Universidade do Minho
Escola de Engenharia

Vitor Alberto Teixeira da Silva

HAL-ASOS Hardware Assisted Linux for Application Specific Operating Systems

Fevereiro de 2022

HAL-ASOS - Hardware Assisted Linux for Application Specific Operating Systems

Vitor Alberto Teixeira da Silva

UMinho | 2022

Universidade do Minho
Escola de Engenharia

Vitor Alberto Teixeira da Silva

HAL-ASOS
Hardware Assisted Linux for Application Specific Operating Systems

Tese de Doutoramento
Programa Doutoral em
Engenharia Eletrónica e de Computadores (PDEEC)
Especialidade de Informática Industrial e Sistemas Embebidos

Trabalho efetuado sob a orientação do
Professor Doutor Adriano José da Conceição Tavares
e do
Professor Doutor Francisco Carlos Afonso

Fevereiro de 2022

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos. Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho

Atribuição-NãoComercial-SemDerivações
CC BY-NC-ND

https://creativecommons.org/licenses/by-nc-nd/4.0/

Acknowledgements

I would like to express my deepest gratitude to Professor Adriano Tavares for the guidance throughout this thesis. His knowledge and expertise were of utmost importance to the achievements in this work. I also would like to thank to Professors João Monteiro and Jorge Cabral for their support helping me to overcome the major obstacles faced during this thesis. I am also grateful to centro ALGORITMI for providing the necessary conditions that made this thesis possible. On a personal note, I would like to thank to my family for every sacrifice made, so that I could pursue my dreams in the world of science. Lastly I would like to acknowledge the financial support received from Portuguese Foundation for Science and Technology (FCT) with the PhD grant SFRH/BD/82732/2011.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

HAL-ASOS - Linux com Aceleração em Hardware para Sistemas Operativos dedicados à Aplicação.

O ecossistema de sistemas embebidos de hoje tornou-se enorme, cobrindo vários e diferentes sistemas, exigindo desempenho e mobilidade completa enquanto atingem autonomias de bateria cada vez maiores. Mas a crescente frequência de relógio que resultou em dispositivos cada vez mais rápidos começou a estagnar antes dos transístores pararem de encolher. Plataformas Field Programmable Gate Array (FPGA) são uma solução alternativa para a implementação de sistemas completos e reconfiguráveis. Fornecem desempenho e eficiência computacional para satisfazer requisitos da aplicação e do sistema embebido. Vários Sistemas Operativos (SO) assistidos por FPGA foram propostos, mas ao estreitar seu foco na síntese do datapath do acelerador de hardware, a grande maioria ignora a integração semântica destes no SO. Ambientes de síntese de alto nível (HLS) elevaram a abstração além da linguagem de transferência de registo (RTL), seguindo uma abordagem específica de domínio enquanto misturam software e abstrações de hardware *ad hoc*, que dificultam as otimizações. Além disso, os modelos de programação para software e hardware reconfigurável carecem de semelhanças, o que com o tempo dificultará a Exploração do Ambiente de Design (DSE) e diminuirá o potencial de reutilização de código. Para responder a estas necessidades, propomos HAL-ASOS, uma ferramenta para implementar sistemas embebidos baseados em Linux que fornece (1) elasticidade no design em conformidade com a natureza evolutiva deste SO, (2) integração semântica profunda de tarefas de hardware nos modelos de programação do Linux, (3) facilidade na gestão de complexidade através de metodologia e ferramentas para apoiar o design, verificação e implementação, (4) orientada por princípios de design híbridos e eficiência no sistema. Para avaliar as funcionalidades da ferramenta, foi implementado um aplicativo criptográfico que demonstra alcance de desempenho enquanto se emprega a metodologia de design. Novos níveis de desempenho são atingidos numa aplicação de Visão por Computador que explora recursos de programação assíncrona-síncrona. Os resultados demonstram uma abordagem flexível na reconfiguração entre hardware e software, e desempenho que aumenta consistentemente com acréscimo de recursos ou frequência de relógio.

palavras chave: FPGA, Linux, Elasticidade Evolutiva, Microcódigo, Aceleração em Hardware.

Abstract

HAL-ASOS - Hardware Assisted Linux for Application Specific Operating System

Today's embedded systems ecosystem became huge while covering several and different computer-based systems, demanding for performance and complete mobility while experiencing longer battery lives. But the rampant frequency that resulted in faster devices began hitting a wall even before transistors stopped shrinking. Field Programmable Gate Array (FPGA) platforms are an alternative solution towards implementing complete reconfigurable systems. They provide computational power, efficiency, in a lightweight solution to serve the application requirements and increase performance in the overall system. Several FPGA-assisted Operating Systems (OS) have been proposed, but by narrowing their focus on datapath synthesis of the hardware accelerator, they completely ignore the deep semantic integration of these accelerators into the OS. State-of-the-art High-Level Synthesis (HLS) environments have raised the level of abstraction beyond Register Transfer Language (RTL) by following a domain-specific approach while mixing *ad hoc* software and hardware abstractions, making harder for performance optimizations. Furthermore, the programming models for software and reconfigurable hardware lack commonalities, which in time will hinder the Design Space Exploration (DSE) and lower the potential for code reuse. To overcome these issues, we propose HAL-ASOS, a framework to implement Linux-based Embedded systems which provides (1) elasticity by design to comply with the evolutive nature of Linux, (2) deep semantic integration of the hardware tasks in the Linux programming models, (3) easy complexity management using methodology and tools to fully support design, verification and deployment, (4) hybrid and efficiency-oriented design principles. To evaluate the framework functionalities, a cryptographic application was implemented and demonstrates performance achievements while using the promoted application-driven design methodology. To demonstrate new levels of performance that can be achieved, a Computer Vision application explores several mixed asynchronous-synchronous programming features. Experiments demonstrate a flexible design approach in terms of hardware and software reconfiguration, and significant performance that increases consistently with the rising in processing resources or clock frequencies.

Keywords: FPGA, Linux, Evolutive elasticity, Microcode, Hardware Acceleration.

Table of Contents

Resumo **vi**

Abstract **vii**

List of Listings **xi**

1 Introduction **1**

1.1 Research questions and Methodology 5

1.2 Scope 6

1.3 State of the Art 6

1.3.1 Native FPGA Acceleration 6

1.3.2 Operating systems for FPGA 7

1.3.3 Application-specific operating systems 12

1.3.4 Microcode-level customizations 12

1.4 Conclusions 13

1.5 Thesis Structure 14

2 Design Methodology **15**

2.1 Design flow 15

2.2 Programming model 18

2.3 Application Development 20

2.4 Software refactoring 23

2.4.1 The File reader task 24

2.4.2 The Encryptor task 25

2.4.3 The Uploader task 26

2.4.4 Functional validation 27

2.5 Application profiling 29
2.5.1 Profile tools 29
2.5.2 Profiling Results 29
2.5.3 Conclusions 31
2.6 Accelerator model 32
2.6.1 Hardware Kernel model 33
2.6.2 Hardware Task model 34
2.6.3 Linux Integration 36
2.6.4 Emulator Model 38
2.7 Computational offloading 39
2.7.1 Hardware specification 40
2.7.2 Emulating Hardware Accelerators 43
2.7.3 Hardware description 46
2.7.4 Co-Simulation model 50
2.7.5 Encryptor co-simulation 52
2.7.6 Encryptor SA co-simulation 58
2.8 Platform deployment 61
2.8.1 Hardware selection 62
2.8.2 Full simulation Model 65
2.8.3 Full system simulation 67
2.9 System Implementation 75
2.10 Conclusions 77

3 First-class Hardware Components 79

3.1 Kernel Core 82
3.1.1 Authentication 84
3.1.2 Control Unit 86
3.1.3 Hardware System Calls 90
3.1.4 Microprogram 96
3.1.5 Time Events 100
3.1.6 System-Level Datapath 103

3.1.7 Kernel Runtime 109
3.1.8 Kernel Call and Response 111
3.1.9 Kernel Procedures 113
3.2 HW-Task 116
3.2.1 Programming Model 117
3.2.2 User Procedures 121
3.2.3 Linux programming interface 125
3.3 Hardware Kernel Interfaces 134
3.3.1 Generic Interface 135
3.3.2 Multi-clock design 136
3.3.3 Synchronizer for generic interface 137
3.3.4 Multi-master design 139
3.3.5 S00 Control Interface 146
3.3.6 S01 Data Interface 150
3.3.7 M00 System Interface 152

4 Auxiliary Hardware Components **157**
4.1 Local-Bus 157
4.2 HW-Mutex 160
4.3 Local RAM 163
4.4 Message-Queue 167
4.5 Local Interrupts 171
4.6 ZeroCopy Unit 173
4.7 Performance Counters 181
4.8 Accelerator Versions 185

5 Experimental Results **190**
5.1 Object detection a case study 190
5.2 Feature detection stage 192
5.3 Software-only Accelerated Feature Detection 195
5.4 Asynchronous-synchronous datapath 201
5.5 Multi-threaded Synchronous design 204

5.6 Stand-alone Synchronous Single-task 211
5.7 Stand-alone Asynchronous Dual-task 216
5.8 Stand-alone Asynchronous Single-task 224
5.9 Performance Comparison 226

6 Conclusions and Future work 234

A Resource Addressing 239

B Clock Synchronizers 249

C Source Listings 253

D Auxiliary Figures 304

E AES Implementation 314

E.1 Computations in the AES 315
E.2 Overview of the AES 318
E.3 Structure of the AES 319
E.4 Cipher Round 320
E.5 The Key Schedule 332
E.5.1 Sequential Key Expansion 334
E.5.2 Pipelined Key Expansion 335
E.6 The Software AES 337
E.6.1 Encrypt process 337
E.6.2 Decrypt process 338
E.7 Hardware Architecture for AES 339
E.7.1 Sequential Encryption 339
E.7.2 Pipelined Encryption 341

F Features from Segment Test 356

F.1 Feature Detection 357
F.1.1 Address Controller 359
F.1.2 Classifier Filter 360

F.1.3 Contiguity Checking 361
F.1.4 Scoring 362
F.1.5 Final Score 362
F.2 Non-maximum Suppression 364
F.2.1 Suppressor 365
F.3 Design Co-simulation 365

References **375**

List of Abbreviations

ABI	Application Binary Interface.
AES	Advanced Encryption Standard.
AMBA	Advanced Microcontroller Bus Architecture.
API	Application Programming Interface.
ASIC	Application Specific Integrated Circuit.
ASOS	Application-Specific Operating Systems.
AXI	Advanced eXtensible Interface.
BRAM	Logic blocks of Random-access memory (RAM).
BSP	Board Support Package.
CPU	Central Processing Unit.
DCR	Device Control Register.
DDS	Data Distribution System.
DMA	Direct Memory Access.
DPI	Direct Programming Interface.
DPR	Dynamic Partial Reconfiguration.
DSE	Design Space Exploration.
DTC	Device Tree Compiler.
EDA	Electronic Design Automation.

FLI Foreign Language Interface.

FPGA Field Programmable Gate Array.

FS File System.

FSM Finite State Machine.

GCC Gnu Compiler Collection.

GPP general purpose processors.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HAL Hardware Abstraction Layer.

HAL-ASOS Hardware Assisted Linux for Application Specific Operating Systems.

HDL Hardware description language.

HLS High-Level Synthesis.

HTC Hybrid Threads Compiler.

HWTI Hardware Thread Interface.

ILP Instruction Level Parallelism.

IP Intellectual Property core.

IP-XACT Standard Structure for Packaging, Integrating, and Reusing IP.

ISA Instruction Set Architecture.

JIT Just-in-time compilation.

LINTC Accelerator Local Interrupt Controller.

LRAM Accelerator Local RAM.

OS Operating System.

PL Programmable Logic.

PLB Processor Local Bus.

POSIX Portable Operating System Interface.

PS Processing System.

RAM Random-access memory.

RTL Register Transfer Language.

SDK Software Development Kit.

SIMD Single Instruction Multiple Data.

SoC System on a Chip.

SPUFS Synergistic Processor Units File System.

stdout standard output descriptor.

TLP Thread Level Parallelism.

UHD Ultra High Definition.

UML Unified Modelling Language.

VFS Virtual File System.

VLIW Very Long Instruction Word.

VM Virtual Machine.

XML Extensible Markup Language.

List of Figures

2.1 The HAL-ASOS design flow. 16

2.2 Simplified UML diagram of template class *Task*. 19

2.3 Overview of the AES-128-bit algorithm. 21

2.4 Machine 1 application - task graphical representation. 22

2.5 The Machine 1 application - SW task version source code. 23

2.6 File reader task - simplified *run()* member. 24

2.7 Encryptor task - simplified *run()* member. 25

2.8 Uploader task - simplified 'run()' member (full listing:C.3) 26

2.9 Machine 1 - SW-only application output for one million digits. 27

2.10 Machine 2 - Server application Output. 27

2.11 Machine 2 - contents of plain data file after decryption. 28

2.12 Machine 2 - contents of encrypted file received. 28

2.13 Profile results of the Machine 1 application. 31

2.14 HAL-ASOS Accelerator model integrated into Host platform. 32

2.15 Hardware Kernel simplified model. 33

2.16 Hardware Task simplified model. 35

2.17 HAL-ASOS file system structure on Linux. 36

2.18 Accelerator Model Emulator - Simplified UML class diagram. 39

2.19 HW Encryptor Task - Simplified Datapath 41

2.20 HW Encryptor Task - Simplified Control unit Finite State Machine (FSM) 42

2.21 Machine 1: emulated HW *Encryptor* software changes 44

2.22 Machine 1: Hardware Task emulator 45

2.23 Machine 1 - HW encryptor using emulator 46

2.24 *Encryptor* - simplified control path. 48

2.25 *Encryptor* SA - simplified control path. 50

2.26 HAL-ASOS simplified Co-Simulation model 51
2.27 Machine 1 - software changes for Co-Simulation using the HW *Encryptor* task. 52
2.28 Co-Simulation - HW Encryptor Accelerator settings 53
2.29 Co-Simulation - HW Encryptor application output 54
2.30 Co-Simulation - HW Encryptor handshake messages 55
2.31 Co-Simulation - HW Encryptor wave plot using Vivado simulator 57
2.32 Machine 1 - Co-Simulation Standalone HW Encryptor Task 58
2.33 Co-Simulation - HW Encryptor SA application output. 59
2.34 Machine 1 - HW Encryptor SA handshake messages using ModelSim 59
2.35 Co-Simulation - HW Encryptor SA wave plot using ModelSim. 60
2.36 Machine 1 -Simplified block design using ZC702 platform. 63
2.37 Programmable Logic - device tree source two accelerators. 64
2.38 HAL-ASOS Full simulation Model 65
2.39 Full Simulation - HW Encryptors block design 67
2.40 Full Simulation - hal_asos_link simulation parameters 68
2.41 Full Simulation - QEMU and Vivado handshake log 69
2.42 Full Simulation - running Linux image on QEMU 70
2.43 Full Simulation - Machine 1 application using HW *Encryptor* task 71
2.44 Full Simulation - HW *Encryptor* simulation waveform. 72
2.45 Full Simulation - Standalone HW Encryptor design 73
2.46 Full Simulation - Stand-Alone HW *Encryptor* simulation waveform. 74
2.47 Machine 1 - Software-only application version running on ZC702 board. 75
2.48 Zynq ZC702 - Application performance of the machine 1 application. 78
2.49 Zynq ZC702 - Application performance of the hardware accelerated designs. 78

3.1 Accelerator Model - and overview of the provided services. 80
3.2 Hardware system call - wait event sequence diagram. 81
3.3 Hardware system call - mutex lock sequence diagram. 82
3.4 Kernel Core internal structure 83
3.5 Authenticator device architecture 84
3.6 Authenticator wave diagram: control operation for *HW-Task* restart. 85
3.7 Kernel Core - Control Unit Overview 86

3.8 Kernel Core - FSM state diagram 87
3.9 Kernel Core - control unit wave diagram. 89
3.10 Kernel Core - system-level datapath and microprogram interaction. 90
3.11 *HW-Kernel* package - system call types, entry and exit records 91
3.12 Microprogram - *opcode* format example in mutex lock step two. 97
3.13 Microprogram - ROM-based internal architecture. 99
3.14 Time Event unit - simplified architecture diagram. 101
3.15 Time Event unit - control logic state diagram. 102
3.16 Time Event unit - Wait for a signal event using a 6-clock timeout. 103
3.17 System-Level datapath - system call signals and target resource. 104
3.18 System-Level datapath - Time event parameters and return signals. 106
3.19 System-Level datapath - Local-BUS and MBUS fields of system call. 107
3.20 Kernel Core - status register signals. 108
3.21 System-Level datapath - Index management service. 109
3.22 Kernel Core - execution progress service. 111
3.23 Kernel Core - input and output interface types using VHDL records. 112
3.24 Kernel Core - execution runtime overview from the *HW-Task* perspective. 114
3.25 Kernel package procedures - wait signal event descriptions. 115
3.26 Kernel package procedures - read local-RAM in burst format descriptions. 116
3.27 Hardware Task - simplified example architecture. 118
3.28 Task Model - concurrent user procedure description example. 119
3.29 HW-Task Model - Sequential units descriptions using VHDL. 121
3.30 User package procedures - scheduling concurrent system calls. 122
3.31 User package - VHDL procedure to burst read the *sysram* memory. 124
3.32 Hardware descriptor - record composition and specialized types. 126
3.33 Kernel package - VHDL excerpt of the file and socket descriptor records. 127
3.34 Kernel-level internal message hierarchy. 128
3.35 Query message - file descriptor query sequence diagram. 129
3.36 User package - procedure to query for a file HW-based descriptor. 130
3.37 Data transfer message - file descriptor write sequence diagram. 132
3.38 Remote call message - file descriptor write data sequence diagram. 133

3.39 Kernel Package - Kernel-level communication tokens. 134
3.40 Generic Interface Master-Slave model 135
3.41 Synchronizer for the generic interface 138
3.42 Generic Bus component - The top level and schedule policies 140
3.43 Generic Bus Arbiter - internal architecture 141
3.44 Generic Bus Arbiter - internal architecture 142
3.45 Generic Bus component - internal architecture 144
3.46 Generic Bus wave plot - timing simulation using A and B interfaces. 145
3.47 Slave decoder component- internal architecture. 147
3.48 S01 interface - multi-master connectivity. 151
3.49 M00 interface - multi-master connectivity. 153
3.50 M00 interface - Read four words in SYSRAM using burst format. 156

4.1 Local-BUS mapping - Low- and High-memory regions. 158
4.2 Local-BUS architecture - slave interfaces and address decoding. 159
4.3 Local-BUS - Resource usage for final design in Zynq ZC702. 160
4.4 HW-Mutex design architecture. 161
4.5 HW-Mutex control unit state diagram. 162
4.6 HW-Mutex wave diagram: concurrency scenarios. 164
4.7 Local RAM - internal architecture for 32-bit accelerator design. 165
4.8 Local-Ram wave diagram: burst read system call. 166
4.9 HW FIFO[1:3] - architecture using 3 and 8 configured parameters. 168
4.10 HW FIFO[1:3] - architecture using 3 and 8 configured parameters. 169
4.11 HW-FIFO [1:3] - Wave plot simulation diagram using Vivado. 170
4.12 Local Interrupt Controller - control and status register. 171
4.13 Local Interrupt Controller - internal architecture diagram. 172
4.14 ZeroCopy unit - Connectivity in the accelerator model. 174
4.15 ZeroCopy unit - architectural design. 176
4.16 ZeroCopy unit - Lock unit. 177
4.17 ZeroCopy unit - Write enable. 177
4.18 ZeroCopy unit - Control unit FSM. 178

4.19 ZeroCopy unit wave diagram - Read the S01 interface and write to the system memory using burst format. 180
4.20 Performance Counter - architecture simplified block diagram. 183
4.21 Performance counter wave diagram - Blocked counter metrics using example *HW-Task*. 184
4.22 HW kernel - architecture configurations for b and f variants. 188
4.23 HAL-ASOS accelerator - architecture components and configuration clause. 189

5.1 Object Detector application block diagram. 191
5.2 Object Detector application test. 192
5.3 Image mapping to Bresenham circle example. 193
5.4 Block Diagram for the Software Full dataset test. 196
5.5 Block Diagram for the Software Non-maximum suppression. 197
5.6 Hardware accelerated feature detection application using HAL-ASOS framework. . . . 198
5.7 Software-only feature detection: application performances when using input and output files and *time* command on the target platform Xilinx ZC-702. 199
5.8 Software-Only feature detection performances using 1080p test images. 200
5.9 An abstract overview of synchronous (a) versus asynchronous (b) pipeline in the *Classi-filter* block design. 202
5.10 Hardware Accelerated Feature detection feature datapath. 203
5.11 Simplified synchronous control unit for HW feature detection. 204
5.12 Extended features for the multi-threaded HW-Task that targets LRAM resource when using the V4_00_B accelerator. 206
5.13 Excerpt of the multi-threaded application source that targets the LRAM resource using the V4_00_B accelerator. 207
5.14 Performances for the multi-threaded HW accelerated feature detention using V4_00_B and V4_00_F accelerators at 50 MHz and 8 kB and 2 kB block sizes. 208
5.15 Application performances between software and synchronous HW accelerated feature detection. 209
5.16 Performances for the multi-threaded HW accelerated feature detention using accelerator versions V4_00_B/F at 142 MHz and 32 kB and 2 kB block sizes. 210
5.17 Excerpt from the source to implement standalone application using synchronous design and the V4_00_F accelerator. 211

5.18 Extended features for the standalone HW-Task that targets SYSRAM resource when using the V4_00_F accelerator. 213
5.19 Performances for the stand-alone synchronous control using V4_00_B/F accelerators at 50 MHz and 8 kB and 2 kB block sizes. 213
5.20 Performance comparison of the synchronous design HW-accelerated applications using 8 kB block and 50 MHz clock frequency. 214
5.21 Performance of the standalone synchronous application using the V4_00_F accelerator and 90,000 corners and varying the input block size and design frequency. 215
5.22 Dual task hardware accelerated feature detection block diagram. 216
5.23 Asynchronous control for hardware accelerated feature detection. 218
5.24 Extended Features design-level for the HW-Task *MFastSA* that targets the *SYSRAM* resource and uses de V4_00_F accelerator version 220
5.25 Performance of asynchronous designs using V4_00_B and V4_00_F accelerators. . . 221
5.26 Performance of the interrupt latency of V4_00_B and V4_00_F accelerators. 222
5.27 Performance of the interrupt latency of V4_00_F accelerators. 223
5.28 Logic Analyzer using interrupt as trigger in the V4_00_F accelerator. 224
5.29 Performances of asynchronous V4_00_A and V4_00_F/B accelerators. 226
5.30 Performances for the asynchronous designs varying the input file. 227
5.31 Performances using dual-task V4_00_F accelerators and 90,000 corners. 228
5.32 Full frame-rate operation using 100 MHz clock and 32 kB input block size. 229
5.33 Hardware accelerated object detection application results. 232
5.34 Software accelerated object detection application results. 233

A.1 S00 interface - Two-level address decoding design. 240
A.2 S00 interface - Two-level address resource usage in Zynq7000. 240
A.3 S00 interface - Single-level address resource usage in Zynq7000. 241
A.4 S00 Interface - Two-level address resource usage in Virtex 5 family. 242
A.5 Slave decoder internal architecture block diagram. 243
A.6 S00 interface - Resource usage for tri-state datapath design in Zynq7000. 243
A.7 S00 interface - Resource usage for MUX datapath design in Zynq ZC702. 244
A.8 Slave decoder internal architecture block diagram. 245
A.9 Page decoder component - internal architecture. 246

A.10 S00 interface - Resource usage final design using ZC702 platform. 246

B.1 Flip-Flop D time constraints setup, hold and propagation delay 249
B.2 Flip-Flop D time constraints violation 250
B.3 Multi-Flop D synchronizer circuit 250
B.4 Multi-Flop D pulse synchronizer circuit 251
B.5 Synchronizer Generic: single clock ACK-based handshake circuit. 252

D.1 Platform Deployment - Machine 1 block design using ZC702 board (Back to Figure 2.36). 304
D.2 Co-Simulation - Standalone HW Encryptor signals using Vivado simulator (back to Figure 2.35) 305
D.3 Time Event unit - Wait for a signal event using 6 as timeout parameter, not exhausted. 306
D.4 Time Event unit - Wait for a signal event using no timeout. 306
D.5 ZeroCopy unit wave diagram - Read system memory and write S01 interface in burst format. 307
D.6 Kernel Core - FSM state diagram for the RAM-based microprogram. 308
D.7 Kernel Core - Control and status registers for the RAM-based microprogram. 308
D.8 Kernel Core - system-level datapath and microprogram interaction for the RAM-based microprogram. 308
D.9 Microprogram - RAM-based internal architecture. 309
D.10 Feature detection test images dataset. 309
D.11 Synchronous control unit for the multithread-based HW-Task design. 310
D.12 Synchronous control unit for the FastSA HW-Task. 311
D.13 Block design for the Dual-Task Asynchronous design targeting ZC702 platform. 312
D.14 Asynchronous control unit for the single Task feature detection. 313

E.1 The AES block interface. 318
E.2 The AES-128 algorithmic structure. 319
E.3 The AES block cipher matrix - The state S matrix. 320
E.4 The AES parallel *SubBytes* block diagram. 321
E.5 The *sbox* function internal blocks. 321
E.6 The software *SubBytes* function using C language. 323
E.7 The software *InvSubBytes* function using C language. 323

E.8 The hardware *SubBytes* step using VHDL language. 323
E.9 The *ShiftRows* step in the AES. 324
E.10 The software *ShiftRows* function using C language. 325
E.11 The software inverse of *ShiftRows* function using C language. 325
E.12 The hardware *ShiftRows* function using VHDL. 326
E.13 The *MixWord* (or *MixColumn*) block in the AES *MixColumns* step. 328
E.14 The AES *MixColumns* - software implementation using C language. 329
E.15 The AES *InvMixWord* - software source using C language. 330
E.16 The AES *MixWord* - hardware description using VHDL. 330
E.17 The AES *AddRoundKey* step block diagram. 331
E.18 The AES *AddRoundKey* step - software implementation using C language. 332
E.19 The AES *AddRoundKey* step - hardware implementation using VHDL. 333
E.20 The AES Key Schedule Algorithm- pipeline-based block diagram. 333
E.21 The AES Key Expansion - software C sequential implementation. 335
E.22 The AES Key Expansion - hardware sequential-based description using VHDL. 336
E.23 The AES software encrypt process - sequential implementation. 338
E.24 The AES software decrypt process - sequential implementation. 339
E.25 The hardware AES - sequential design. 340
E.26 The hardware AES - pipeline design. 342

F.1 HW-Fast or HW-NMS parameter registers. 356
F.2 Hardware Feature detection datapath. 358
F.3 Address controller hardware block diagram. 359
F.4 Classifier filter hardware block diagram. 361
F.5 Contiguity check hardware block diagram. 362
F.6 Scoring hardware block diagram. 363
F.7 Feature detection final score. 363
F.8 Non-Maximum suppression datapath. 364
F.9 Suppression hardware block diagram. 365
F.10 Feature Detection co-simulation - Block design using Vivado. 366
F.11 Feature Detection co-simulation - Single corner input file. 367
F.12 Feature Detection co-simulation - Wave plot of the FAST datapath. 369

F.13 Feature Detection co-simulation - Wave plot of the FAST+NMS datapaths. 370
F.14 Feature Detection co-simulation output logs. 371

List of Tables

1.1 Gap Analysis considering the literature solutions revised in the section. 13

3.1 Kernel Core - HW system call summary description 93
3.2 Microprogram - Binary excerpt from the Microprogram. 98
3.3 The S00 interface - Page 0 internal register mapping. 148
3.4 The S00 interface - Page 1 internal register mapping. 149
3.5 Performance Counter internal register offsets. 150
3.6 The M00 interface - signal description and generic interface mapping. 154

4.1 HAL-ASOS accelerator versions V3 and V4 features. 186

5.1 Hardware Accelerated Feature detection designs comparison. 230
5.2 HW-Accelerated FAST using HAL-ASOS (a) compared to the literature. 231

List of Equations

5.1 Fast algorithm Bright comparison 193
5.2 Fast algorithm Bright comparison 193
5.3 Fast algorithm Scoring function 194

List of Listings

2.1 Profile script used on Machine 1 application. 30
A.1 Page decoder component - entity declaration using VHDL. 247
A.2 Slave decoder component - architecture description using VHDL. 248
C.1 ’Encryptor’ task ’run()’ member specialization (Figure 2.7). 253
C.2 ’File reader’ task ’run()’ member specialization(Figure 2.6). 254
C.3 ’Uploader’ task ’run()’ member specialization (Figure 2.8). 255
C.4 Machine 1:Encryptor HW Task:(back to Figure 2.24). 256
C.5 Machine 1:Encryptor HW Task datapath:(back to text 2.7.3). 257
C.6 Encryptor SA - control path (1/2):(back to Figure 2.25). 258
C.7 Encryptor SA - control path (2/2):(back to Figure 2.25). 259
C.8 Encryptor SA - synchronous control path:(Figure 2.25). 260
C.9 Encryptor SA - synchronous datapath:(Figure 2.25). 261
C.10 Co-Simulation Encryptor - SystemVerilog test-bench file:(sec. 2.7.5). 262
C.11 Full Simulation - SystemVerilog testbench for QEMU. 263
C.12 Machine 1 Application - receive parameters from command line. 264
C.13 Machine 1 Application - HW *Encryptor* software refactoring. 264
C.14 Machine 1 Application - HW *Encryptor* SA software refactoring. 265
C.15 Kernel Package - LBUS register and LRAM write procedures. 266
C.16 Microprogram - VHDL description for the test input MUX. 267
C.17 Microprogram - VHDL description for the outputs DEMUX.. 268
C.18 Microprogram - VHDL description for program in ROM. 269
C.19 Local-BUS - System call configuration descriptions. 270
C.20 Kernel Package - import and export Kernel Core interfaces. 271
C.21 Kernel Package - VHDL procedures declaration (part 1/3). 272
C.22 Kernel Package - VHDL procedures declaration (part 2/3). 273

C.23 Kernel Package - VHDL procedures declaration (part 3/3). 274
C.24 User Package - VHDL procedures declaration (part 1/6). 275
C.25 User Package - VHDL procedures declaration (part 2/6). 276
C.26 User Package - VHDL procedures declaration (part 3/6). 277
C.27 User Package - VHDL procedures declaration (part 4/6). 278
C.28 User Package - VHDL procedures declaration (part 5/6). 279
C.29 User Package - VHDL procedures declaration (part 6/6). 280
C.30 User package - VHDL procedure to read the *sysram* memory. 280
C.31 Data transfer message - file descriptor write sequence diagram. 281
C.32 HAL-ASOS Accelerator - HW-Kernel configurations for the a, c, d and e variants. 282
C.33 File reader software Task *run* member. 283
C.34 CornerDump and CornerUploader software *run* members. 284
C.35 Full Feature detector software thread *run* member(1/2). 285
C.36 Full Feature detector software thread *run* member(2/2). 286
C.37 Software application for the synchronous standalone feature detection. 287
C.38 Synchronous control for HW accelerated feature detection(1/2). 288
C.39 Synchronous control unit for the HW accelerated feature detection(2/2). 289
C.40 Extended features synchronous Control unit for the HW accelerated feature detection using the LRAM(1/2). 290
C.41 Extended features synchronous Control unit for the HW accelerated feature detection using the LRAM(2/2). 291
C.42 Extended features synchronous Control unit for the HW accelerated feature detection using the SYSRAM(1/2). 291
C.43 Extended features synchronous Control unit for the HW accelerated feature detection using the SYSRAM(2/2). 292
C.44 Extended features for the standalone synchronous Control unit of the HW accelerated feature detection using the SYSRAM(1/2). 293
C.45 Extended features for the standalone synchronous Control unit of the HW accelerated feature detection using the SYSRAM(2/2). 294
C.46 Control FSM VHDL description for the FastSA HW-Task (1/2). 294
C.47 Control FSM VHDL description for the FastSA HW-Task (2/2). 295

C.48 Extended features VHDL description for the MFastSA HW-Task. 296
C.49 Extended features VHDL description for the FastSA HW-Task. 297
C.50 Asynchronous control unit for the NonmaxSA HW-Task(1/2). 298
C.51 Asynchronous control unit for the NonmaxSA HW-Task(2/2). 299
C.52 Asynchronous Extended Features the NonmaxSA HW-Task(1/2). 299
C.53 Asynchronous Extended Features the NonmaxSA HW-Task(2/2). 300
C.54 Network changes to asynchronous Extended Features the NonmaxSA. 300
C.55 Standalone Dual-Task network-based Feature Detection. 301
C.56 Ed.Rosten-C network-based Feature Detection (1/2). 302
C.57 Ed.Rosten-C network-based Feature Detection (1/2). 303
E.1 The hardware AES - Pipeline architecture (back to Figure E.26). 344
E.2 The hardware AES - Sequential architecture (back to Figure E.25). 345
E.3 AES Key Expansion - Pipeline architecture (back to E.5.2). 347
E.4 AES Key Expansion - Sequential architecture (back to E.5.1). 349
E.5 AES Cipher Round - architecture description using VHDL. 351
E.6 AES SubBytes - architecture description using VHDL. 351
E.7 AES SubWord - architecture description using VHDL. 352
E.8 AES Sbox - architecture description using VHDL (back to Figure E.8). 352
E.9 AES MixColumns description using VHDL (Figure E.16). 355

List of Algorithms

1 Microprogram to lock an HW-Mutex 94

2 Microprogram to write one word to the system bus 95

3 Microprogram to write word at M00 interface 155

Chapter 1

Introduction

With network connectivity, Operating System (OS) and database integration, today's embedded systems universe became huge, covering several and different computer-based systems, both in size and functionalities. From mobile phones to self-driving car systems, from Ultra High Definition (UHD) cameras to remote health monitors, from a simple smart watch to internet-aware home devices, our world has been shaped by an increasingly sophisticated set of electronic devices. These are the elements in a highly interconnected ecosystem that is smarter, more efficient and strives for multi-functionality and flexibility, thus facing harder and ever-increasing complex designs.

Today's most frequent demands for embedded devices are still grounded to performance, from multiple and concurrent software applications, high quality graphics, to complete mobility, with connectivity everywhere and longer battery lives. But the rampant frequency that resulted in faster devices, began requiring huge cooling systems and state-of-the-art power supplies to keep up with the power-hungry Central Processing Unit (CPU)s. The *Moore's Law* [1] that served as the underlying philosophy that driven processor design, began hitting a wall even before transistors stopped shrinking.

Industry has adopted the multicore-platform to deliver advances in the current and next-generation embedded devices. We live now in the many-core era where Thread Level Parallelism (TLP) has become a dominating factor in computing performance. But the computational performance is not per se guaranteed by an increase in the number of CPU cores in the system. Performance benefits are mainly restricted to the code sections that can be parallelized, from coarse grained process and thread levels, through finer grained instruction and data levels (*Amdahl's Law* [2]). Currently, two issues must be addressed efficiently, the scalability and the heterogeneity. While the processor utilization, throughput and Instruction

Level Parallelism (ILP) are the root drivers of performance in the system, the performance of the many-core demands for scalability, as the system utilization will only be exacerbated by a proper and efficient parallelization. Another consideration is the fact that distinct CPU architectures will map more efficiently into specific types or sections inside an application. The control dominated or event-bounded sections are generally composed by independent code sections and can be efficiently executed in more traditional out-of-order execution processor. Other sections can be more data-centric or processor-bounded, such as image or signal processing, and can also executed in the same machine, but it will experience great performances in a more complex CPU architecture such as Single Instruction Multiple Data (SIMD) or Very Long Instruction Word (VLIW) processors.

The struggle to serve the different application needs pushed chip manufactures into designing systems that mix core architectures and dynamically adjust performance to the computational needs. ARM has been developing the big.LITTLE heterogeneous processing Architecture since 2011. The design uses two classes of processors and in the latest design specifications, the Big processor cluster can include four Cortex-A73, while the Little processor cluster can also include four Cortex-A53. Apple launched the A13 Bionic chip, an ARM-based System on a Chip (SoC) design that includes a 64-bit hexa-core processor with two 2.65 GHz Lightning cores for high-performance processing, and four 1.8GHz Thunder cores for power efficient processing. Intel announced the Hybrid x86 CPU designed with power efficiency in mind, with one x86 Sunny Cove core (big CPU) and four smaller design x86 CPU cores (small CPUs).

However, different core architectures can introduce Instruction Set Architecture (ISA) and Application Binary Interface (ABI) incompatibilities, different memory hierarchies, cache organization or coherence algorithms. Currently, specific OSes are used to deal with these issues, by abstracting the computing platform into a single Virtual Machine (VM), where system calls represent the standard set of operations for each specific machine while providing the designer an established way for structuring its applications. The assumption that such set of virtual operations will be available across different platform distributions, provides the means for portability and consequently guide industry acceptance. Although a valid solution, the specific OSes usually restrict the available application software base and target specific software tools that can raise development effort. Consequently, new design projects are featured by growing software complexity and engineering effort.

Generic purpose OSes usually provide fast application prototyping with eased use, high software integration, increased hardware support, and extended debugging features that are not usual in target specific

OSes. But despite of the expected overhead and performance metrics degradation, most of the 'well-accepted' operating systems, were never created to abstract such a level of heterogeneity into a unified VM model. They are built into the premise of some internal homogeneity and so, they are now struggling with the hardware level asymmetries at many levels of implementation, raising considerable issues to programming and increased computational overhead to conform with the system.

The risen in silicon logic densities, pushed the Field Programmable Gate Array (FPGA) from being applied as glue logic and prototyping towards implementing complete reconfigurable systems. Today's FPGA platforms provide large density fabrics and include the latest multi-core CPU architectures. They represent the desired architecture for most embedded devices and motivated designers to use them not only as development platforms but also as final products. Offloading computation to specialized hardware circuitry is not new as it has been successfully used in the past. The mix of fast CPU cores and fine-grained reconfigurable logic allows to map both sequential or control-dominated code and highly parallel data-centric computations into a single platform. They can provide computational power, efficiency, in a light-weight solution to serve the application requirements, increasing performance, and they can also be considered as complementary to complex heterogenic processor architectures.

However, the programming models for software and reconfigurable hardware lack commonalities, which in time will hinder the Design Space Exploration (DSE) and lower the potential for code reuse. Traditional design techniques were not able to kept with these risen in system complexity. Generally, they do not consider any efficiencies on the purposed programming models. Also, the existing design techniques for these types of reconfigurable devices evolved from the Application Specific Integrated Circuit (ASIC) design and tend to view the specialized hardware as passive processing units in the system.

A new design methodology is demanded for dealing with aforementioned new systems requirements and constraints of multiple functionalities, programmability, heterogeneity, smartness, real-time performance, power consumption and security due to connectivity, that all together have been compounding the design complexity. Essentially, it must raise the abstraction level to design, allowing the user to quickly envision, develop and deploy the application. Specifically, it must be one that: (1) promotes the reconfigurable hardware to first-class computing unit, being able to synchronize, communicate and notify other computing units in the system; (2) provides seamless hardware integration through an automated DSE, guided by an improved metric-driven approach and encompassing an integrated system emulator with multilevel simulation; (3) encourages creativity, exploring the hardware and software synergies, and thus expanding the scope of electronic design beyond its original boundaries.

In spite of the panoply of existing Electronic Design Automation (EDA) tools, supporting capabilities such as high-level synthesis, system profiling, simulation and emulation as well as Standard Structure for Packaging, Integrating, and Reusing Intellectual Property (IP) (IP-XACT) [3] design and overlay architecture for FPGAs, only to mention a few, none of them can efficiently handle such demand for nowadays systems requirements and constraints. A toolset is in need to leverage IP evaluation, quality assurance and mostly a snap-in integration. Most of the existing IPs, are blacked boxed with few or none visibility into, neither software to support for validation and assist in device-driver development, thus forcing the integration task to a level of development effort.

To gain the insight to what such a tool needs to address it is better to start by answering what is the today's system realization. Traditionally, most systems are developed from the bottom up starting with the hardware. The OS already exists and is generally pre-selected and the applications are developed within the limitations of the pre-determined hardware and software stacks. Typically, the application development is largely abstracted from the hardware, and in the absence of a virtual emulator, the application will not be completed until the target hardware is fully available. The system integration and debugging will occur later in the development cycle, and it will usually face schedule delays and quality issues that, ultimately resulted in quality degradation. Applications are pre-sentenced to the underlying hardware and software layers, and must conform to any constraint limitations imposed. Any potential glitch that simultaneously involves the application, OS, and hardware layers is extremely difficult to fix and demands for very time-consuming effort with a lot of iteration and debugging.

To support the narrow range of application needs and still being able to tackle performance and efficient design metrics, one must ditch the hardware-first paradigm and follow a different approach. One that starts by quickly envisioning the application, allowing designers to feel the application needs and then choose the right platform, resources and the software layers. For an effective performance, the hardware and software must to be developed concurrently to better promote efficiency in a resource-aware design approach that fits the solution with just the right needs.

1.1 Research questions and Methodology

We believe that extracting the performance benefits from computational offload to FPGA reconfigurable devices, requires an agnostic approach to the software. One that follows an application-centric design methodology, where the application is driving the system requirements instead of system capabilities driving the application. One that raises the abstraction level to programming and provides transparent reconfigurable hardware devices integration.

For these reasons, this thesis tries to answer to the following questions:

1. How can we transparently and dynamically extend the Linux programming models with reconfigurable hardware devices?
2. How can we lower programmability gap between the hardware and software?
3. Can we provide and automated design flow that mitigates the system complexity, keeps track on development and ensures compliance with the design metrics, while leveraging better computing and resource efficiency?

We address the questions above by applying the following methodology that is anchored to (1) elasticity by design, (2) deep semantic integration, (3) easy complexity management, (4) hybrid and efficiency-oriented design principles, to implement Hardware Assisted Linux for Application Specific Operating Systems (HAL-ASOS):

1. Run a parallelization tuning cycle using profilers and based on several workloads to identify critical Linux kernel- and user-level subsystems that should be tuned for scalability;
2. Propose a high-level programming abstraction at the same level of software task to express hardware translated tasks from Linux, application and middleware components by making reconfigurable hardware first-class computing entities;
3. Leveraging computing and resource efficiency by applying mixed asynchronous–synchronous design, event-driven, microcode dynamicity and laziness approaches;
4. Support system designer in the creation of the full platform solutions, including Board Support Package (BSP), OS, device drivers, middleware and applications software;

5. Provide an integrated solution that automates the development of the software on which the user application will be built by exploring the capabilities of the application-tailored SoC.

HAL-ASOS targets the design of Embedded systems, tailored to the application requirements, which are implemented using CPU+FPGA platforms.

1.2 Scope

The scope of this thesis falls within the development of software and reconfigurable hardware devices, but it is constrained to the hardware accelerated Linux-based embedded systems on CPU+FPGA platforms. It also falls into the scope of design methodologies to ensure an efficient design and ease the programmability gap between software and hardware, while providing the designer with a complete solution for developing reconfigurable systems that, benefit from the synergy among software, hardware and services, and deliver powerful computation solutions that can be built with just the right resources.

1.3 State of the Art

The state-of-the-art for this thesis falls into four areas that will be discussed in the paragraphs below: (1) Native or *ad hoc* FPGA acceleration (2) operating systems for FPGA; (3) Application-Specific Operating Systems (ASOS); and (4) microcode-level customization and update:

1.3.1 Native FPGA Acceleration

Many native FPGA-based acceleration solutions exist, which are hand-optimized for one specific application and FPGA platform, hindering the productivity by demanding for complete rewriting or time-consuming porting. HThreads [4] provided a unified multi-threaded programming model for architectures with reconfigurable components, by delivering mechanisms that implement transparent integration of hardware threads into a heterogenous system. Such mechanisms, implement basic scheduling, synchronization and interrupt handling for the hardware threads. A Hardware Thread Interface (HWTI) abstracts a platform-independent compilation target, for hardware-resident computations. It enables the use of standard thread communication and synchronization across the software/hardware boundary. The system is designed from C code language sources, which are compiled using the HybridThreads compiler (HTC) to create

VHDL code that is integrated into the HThreads synthesis process. A runtime support implementing a hardware-based microkernel provides OS backend to the components in the system. It enables the design of heterogeneous systems using Portable Operating System Interface (POSIX) programming abstractions. From the designer perspective, it provides a multi-threaded programming model where a parent thread creates any number of children threads that will execute transparently on the underlying computational resources. Despite the novelty in this research, the use of dedicated compiler from C to Register Transfer Language (RTL) to implement hardware threads in the application, is very limiting in terms of portability and maintenance.

Luca Pezzarossa et al. [5] evaluated the potential benefits of using Dynamic Partial Reconfiguration (DPR) to implement hardware accelerators in real-time systems by driving the main focus towards: (1) trade-offs between hardware utilization, worst-case performance, and speed-up over a pure software solution and (2) the trade-offs between the use of multiple specialized accelerators combined with DPR instead of the use of a more general accelerator, and the memory footprint of the partial-bit streams. The interaction between software and hardware is based on the control registers of each accelerator and specific shared memory regions. For testing, it implemented a passive coprocessor model where the software is responsible for activating accelerators when input data is ready for processing. The results compare performance of the software using softcore processor over the hardware accelerators in combination with the DPR feature of the FPGAs. The use of DPR can lead to significant reduction in the hardware size when the reconfigured tasks are computationally intensive, and maintain performances gains ranging from 1.2 to 4.1 times over the software execution.

Solutions described above narrow their focus on datapath synthesis of the hardware accelerator, completely ignoring the deep semantic integration of these into operating system, or high-level synthesis (High-Level Synthesis (HLS)) environments as well as DPR-enabled elasticity. In the absence of and OS environment, applications fail to handle reliable software operation as well as the legacy software execution and well-established programming models.

1.3.2 Operating systems for FPGA

There have been many proposals for building operating systems for FPGA, mainly due to the risen in silicon logic densities alongside the differentiating capabilities of FPGAs. Offloading computation to specialized hardware circuitry has been used to provide computational power, efficiency, in a light-weight solution to serve the application requirements and increasing performance, while it can also be considered as complementary to complex heterogenic processor architectures. To reduce development time and to ease a complex design implementation, HLS environments have raised the level of abstraction beyond RTL (i.e., by using high-level languages such C/C++ or OpenCL) and following a domain-specific approach, while mixing *ad hoc* software and hardware abstractions, making harder performance optimizations. Furthermore, design portability is strongly impacted when changing from one HLS environment to another, due to their specific dependencies on custom data type, hardware support IPs, and compiler-specific ”pragmas” [6].

BORPH [7] implements an operating system level support for FPGA-based reconfigurable computers. It introduces the concept of hardware process which is the same as normal UNIX process, but execution is handled by hardware circuits on FPGA. Under BORPH, hardware and software share the same familiar UNIX interface and the same level of support from the OS kernel. An application using BORPH is composed of a collection of files, to implement a predetermined number of processes, that can be software or equivalent hardware processes. The framework is composed of a kernel module and a user Application Programming Interface (API) that provides the set of system calls to interact with the computing resources. The kernel module is responsible for the request handling that mainly correspond to the allocation and configuration of the hardware resources. The API includes a series of system calls that implement a message passing interface allowing a hardware process to access data files or to communicate with other processes using pipes. In design terms, a hardware process is a BORPH executable binary file (BOF), that contains information about the resources. By extending the standard Linux kernel to accommodate the hardware process, it conflicts with its evolutive nature, demanding for a continuous updated patching of the Linux source. No hardware interface is established and the communication is based on passive hardware regions that are populated into the Linux *procfs*, under the process identification (pid) folder. In doing so, a new folder is created with any new execution, thus creating different virtual file locations that are unpredictable to the application.

FUSE [8] implements a framework to abstract HW accelerators and design PetaLinux-based embedded systems. It provides a POSIX compliant thread model that can be implemented using software or hardware resources. To abstract the accelerator model to the software application, it relies on the Top-Level FUSE Component (TLFC) and the Low-Level FUSE Component (LLFC), that operate at Linux user and kernel spaces, respectively. The TLFC is as software library that provides thread creation, initialization, scheduling and destruction. It also implements a set of functions to interact at the LLFC using the OS interface. The LLFC is a kernel module that acts as low-level abstraction and is responsible for the communication, synchronization and monitoring of the hardware tasks. Each hardware task is coupled with an accelerator interface that is orchestrated by the LLFC module. For every such interface, a loadable kernel module is created dynamically at runtime and LLFC maps each hardware interface as peripheral I/O devices. Communication is implemented on the software side, exchanging data using operating system services and thus forcing the hardware task design to a passive coprocessor model.

FOSFOR [9] is a framework that implements a transparent abstraction layer for applications following the Synchronous Data Flow model, deployed on System on Chip architectures. The underlying platform is composed by a combination of reconfigurable hardware regions (RR) and the required number of general purpose processors (GPP)s. It considers a Synchronous Data Flow model description and an architectural mapping to describe associations between processing units and actors. The implementation is based on C/C++ language for the software, and VHDL descriptions for the hardware-based components. The resulting synthesis is a hardware design used to configure the FPGA and perform the actual computation. It implements the concept of hardware threads, that are composed of a static component responsible for the communication, providing local resources and associated control interfaces. A dynamic part is used to implement the application related design. For this, Flexible OS was selected as hardware operating system and RTEMS was chosen for the management of the software parts. A middleware layer implemented using hardware and software parts establishes a single programming interface. It abstracts both implementation and mapping, and offers some mechanisms like the accesses to OS services and the inter-process communication. Focusing on the synchronous data flow model, system portability increases, but communication efficiency becomes a system performance bottleneck.

SPREAD [10] provides a point-to-point streaming oriented programming environment, for architectures with reconfigurable components. It is a thread-based approach where hardware tasks are encapsulated into threads that resemble the POSIX programming interface. It considers three basic operations performed by each thread, in getting the application-related data, the computation itself, and then providing the processing results to the next thread. To facilitate switching between hardware and software, a software delegate is created, and is responsible for the monitoring of the hardware thread on the software side. A Hardware Thread Interface (HTI) abstracts the hardware thread design and enables communication between hardware and software as well as between hardware threads. Streaming channels can be dynamically interconnected at runtime, to provide inter-thread communication. Depending on whether the execution resource is hardware or software it provides three distinct communication channels. When both threads are implemented in software, the communication is implemented using memcopy function. When implemented on hardware, they communicate directly using FIFOs that are synchronized by the HTI. Lastly, when a software and hardware threads need to communicate, they use a Direct Memory Access (DMA) point-to-point connection. A hardware thread manager is also provided to handle hardware threads and is responsible for their creation and termination. Unlike previous frameworks, SPREAD is an integrated solution specifically developed for streaming application design. It does not allow "one-to-many" and "many-to-one" streaming interconnections, which are more common but not so easy automate. The hardware task programming interface is centered around the passive coprocessor model, where control unit blocks waiting for input data, executes processing upon arrival, and concludes producing results on the local resources and signaling the software execution.

ReconOS [11] implements an operating system that extends the software multi-threaded programming model to reconfigurable hardware. From the programmability point of view, it provides an API close to the POSIX programming model where threads can be executed either on CPU or in reconfigurable hardware. Software threads can be implemented using the POSIX library and synchronized using semaphores, mutexes, condition variables and mailboxes. Hardware threads are abstracted by the OS interface (OSIF) that implements synchronization and communication mechanisms. An indexing scheme is also used to implement resource sharing. The software-level inter-thread communication is implemented using shared memory, while communication between hardware and software uses a message-based exchange model. Thread memories are shared leaving consistency and coherence hazards to be handled by the programmer. Such constraint penalizes the performance in the overall system by the increased use of interrupt events and consequent latency involved.

Zongwei Zhu et al. [12] proposes a task scheduling framework on the DPR-based platform that exploits the hardware task cycle accuracy and task preemption overhead to improve scheduling efficiency. The framework is based on general OS and takes the full consideration of the preemption overhead, the reconfiguration time and hardware tasks' cycle accuracy. The scheduling method is based on the predictable execution time of hardware tasks in DPR to improve scheduling efficiency of the whole system. The hardware task participates in the scheduling of the OS through the associated delegate thread and optimizes the task scheduling model, thereby reducing both the number and the overhead of task context switch.

FOS [13] provides an OS that adopts a modular FPGA development flow, to allow each system component to be changed and be agnostic to the heterogeneity of EDA tool versions, hardware and software layers. It dynamically maximizes the utilization transparently from the users by using resource-elastic scheduling to arbitrate the FPGA resources in both the time and spatial domain for any type of accelerators. Moreover, FOS can switch between different accelerator implementations on the fly in order to balance resource allocation for the best performance and load scenarios. It provides an application-centric view to the developers by hiding most of the complexity encountered when using a heterogeneous CPU-FPGA acceleration system with a Linux backend. A user level daemon is responsible for managing FPGA resources and initiate scheduling operations. Updating individual components includes the latency of the partial reconfiguration that ranges from 3.8 to 6.8 milliseconds to replace one hardware accelerator, or 20.7 to 98.4 milliseconds to replace the entire FPGA shell. It abstracts the software design perspective but disregards the efforts required to semantically integrate the hardware accelerators in the application, mapping them by its hardware regions and considering accelerators as passive coprocessors. A design methodology is also proposed but it lacks co-simulation or full-simulation supports to validate the accelerator nodes in the application.

Hoang-Gia Vu et al. [6] propose a hardware task migration scheme assisted by (1) a checkpointing architecture for FPGAs that flattens the structure of nested modules at the hardware description language (HDL) level, (2) a static analysis of the original HDL source code to reduce the cost of hardware and (3) Python-based tool to generate the checkpointing architecture at the HDL level. In the hardware task migration scheme the checkpoint procedures overlap data transfers to minimize downtime to 1.251 milliseconds in the worst-case scenario. When compared to the original design, clock degradations observed vary from 9.73% to 0.15% averaging at 1.66%. The design is limited to a single-clock domain and yet to be ported across different FPGA vendors.

Coyote [14] is a configurable FPGA "OS" for hybrid compute servers, designed mainly for reflecting on the performance and efficiency benefits of importing OS abstractions wholesale to FPGAs. It goes beyond ReconOS's deep semantic integration by supporting secure spatial and temporal multiplexing of the FPGA between tenants, virtual memory, communication, and memory management inside a uniform execution environment. The overhead of Coyote is small and the performance benefit is significant, but more importantly it allows us to reflect on whether importing OS abstractions wholesale to FPGAs is the best way forward. It shows that a coherent and reasonably complete set of OS abstractions can be implemented, and still achieve acceptable performances in throughput, space efficiency, scheduling overhead and memory bandwidth. As FPGAs become larger, the demand for the more traditional OS services will grow. Migrating commonly used OS features to hardware IPs requires the right set of abstractions to prevent them from quickly becoming obsolete.

1.3.3 Application-specific operating systems

Several works have been conducted on performance optimization of different features of an operating system due to the following reasons [15]: (1) OSes are critical to the performance of the running application, especially for system-intensive applications that invoke kernel features extensively, and (2) nowadays in cloud era, many servers only run a single application. Tarax [15] is a one-size-fits-all compiler-based and profile-guided optimization approach for constructing an ASOS. The implementation is based on modified versions of the Linux kernel and Gnu Compiler Collection (GCC), to support kernel instrumentation and profile collection. Detailed analysis has provided insights on how profile feedback helps GCC to perform better optimizations on the Linux kernel in an application-specific manner. The outcome is an optimized kernel image tailored to improve performance of the application and reduce kernel size. Experiments conducted using popular server applications provided a performance increase of 16% in the Linux kernel. Differently from the HAL-ASOS design framework that is assisted by the mainstream and system-wide *OProfile* tool, Tarax does not seamlessly evolve with the Linux OS kernel as it demands both the instrumented Linux OS kernel and GCC.

1.3.4 Microcode-level customizations

Microcode is an abstraction layer between the physical components of a CPU and the programmer-visible instruction set architecture of the computer. Originally, it was purposed to simplify the design of CISC (Complex Instruction Set Computing) CPUs with capability for in-field CPU updating without requiring any special hardware [16]. More recently, x86 microcode-level update capability gains moment by mitigating Spectre and Meltdown vulnerabilities. Benjamin Kollenda et al. [16] reverse engineered the microcode of x86 CPU and proposed a microcode-assisted instrumentation framework, alongside the enclave functionality to realize a small trusted execution environment, leveraging system security defenses such as timing attack mitigations, hardware-assisted address sanitization, and instruction set randomization. CHEx86 processor architecture [17] proposes a transparent capability-based protection scheme enforced through microcode instrumentation, to defend against security exploits targeting temporal and spatial memory safety vulnerabilities. These works are not directly compared to the evolutive elasticity of Hardware Kernel, but similar microcode mechanisms are deployed in both fields.

1.4 Conclusions

To conclude the state-of-art review, Table 1.1 provides the gap analysis based on the above design principles. Any further details are already given on the previous summaries of each individual compared works.

Table 1.1: Gap Analysis considering the literature solutions revised in the section.

	SW Base		**Integration Levels**				**Design**	**Clock**
	Linux	Other	Semantic	P.Model	Elasticity	Support	**Strategy**	**Domains**
HThreads [4]	-	✓	PCoP	TH	ST	-	SYN	SC
Luca et al. [5]	-	✓	PCoP	TH	ST/DPR	-	SYN	SC
BORPH [7]	✓	-	PCoP	PR/FS	ST	-	SYN	SC
FUSE [8]	✓	-	PCoP	TH	ST	-	SYN	SC
FOSFOR [9]	-	✓	PCoP	TH	ST	-	SYN	SC
SPREAD [10]	-	✓	PCoP	TH	ST	-	SYN	SC
ReconOS [11]	✓	✓	OSL	TH	ST/DPR	-	SYN	SC
Zhu et al. [12]	-	✓	PCoP	TH	ST/DPR	-	SYN	SC
FOS [13]	✓	-	PCoP	TH	ST/DPR	M	SYN	SC
Vu et al. [6]	-	✓	PCoP	TH	ST/DPR	-	SYN	SC
Coyote [14]	✓	-	OSL	TH	ST	-	SYN	SC
HAL-ASOS	✓	-	OSL	TH/FS	MP	M,C-FS	A-SYN	MX/SC

PCoP - Passive Coprocessor Model semantic integration; OSL - Operating System Level semantic integration;
TH - Thread-based programming model; PR - Process-based programming model; FS - File System-based programming model;
ST - Static Design approach; DPR - Dynamic Partial Reconfiguration features; MP - Micro-programmable design;
M - Design Methodology; C-FS - Co-Simulation and Full System Simulation;
SYN - Synchronous design; A-SYN - Asynchronous-Synchronous design;
SC - Single Clock domain; MX - Multiple Clock domains.

The works included in Table 1.1 are the most representatives in the field, although several others could be included in this review, such as FISH [18], RIFFA [19], RACOS [20], R3TOS [21] and SEOS [22], just to name a few.

1.5 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 provides an overview of the HAL-ASOS framework. A simple case study is introduced and the design methodology is applied and discussed using the HAL-ASOS framework;

Chapter 3 discusses the first-class components in the Accelerator model, that consist of the *HW-Kernel*, the *HW-Task* and the host system interfaces. In begins with an overview of the *HW-Kernel* provided services, followed by the *Kernel Core* implementation, and concludes by discussing *Hardware Task* programming model in the Linux programming interface.

Chapter 4 discusses the *HW-Kernel* auxiliary components used to provide: (1) synchronization with the host system, by use of the *HW-Mutex*(es) and the Local interrupt controller; (2) control-oriented message service, at *HW-Kernel* and *HW-Task* levels, by use of the *HW-FIFOs*; (3) local storage service using the Local-RAM, (4) the *ZeroCopy* unit; and the (5) Hardware Performance Counters. In each section, it provides architecture details and functional simulations that are based on the hardware system calls that address each component feature.

Chapter 5 applies the HAL-ASOS accelerator model to a computer vision application, to assess many levels of performance while using distinct accelerator versions. It initiates development from the software-only implementation, moving towards initial hardware acceleration concepts, and concludes by refactoring the application using distinct levels of asynchronous design;

Chapter 6 summarizes this thesis and describes the future work considering this framework.

Chapter 2

Design Methodology

The design methodology is a field in science that pushes development towards better productivity by reducing technological gaps and exploring their synergies. But developing hardware and software concurrently requires an efficient design methodology that must be transparent from the engineers' perspective, abstracting away the semantic gap between software and hardware concepts (i.e., a co-design flow that assists in modeling, simulation and verification of a design before committing to hardware). To reduce design and coding efforts, such design methodology must rest on principles of control algorithm refinement, modularity, and best suitability between the algorithm to implement and the chosen hardware platform. The overall design decisions will ultimately be constrained to (1) the identification of kernel functions to be offloaded to FPGA fabrics, (2) ensuring a profitable offloading in terms of performance or any pre-selected design metrics, and (3) using accelerator architecture that promotes the accelerators to the same computing level of the CPU.

In this charter, the development of an application will be discussed within the HAL-ASOS framework. It will also provide an overview of the most relevant functionalities an it will introduce the purposed models that combined make up the HAL-ASOS design methodology. We will begin by describing the HAL-ASOS design flow and applying it to the application, which will structure the reminding descriptions in this chapter.

2.1 Design flow

When following the HAL-ASOS design flow, the Application development can start with a new design, or an existing application that needs to be refactored for functionality, quality of the results or performance at several levels of implementation. Any existing design models can be considered, but the best use of the framework starts with the Unified Modelling Language (UML) design describing the system to be implemented and an application/algorithm task-graph illustrating the semantic relations between the system functional units, while also ensuring a clear understanding in terms dependency and data movement in the application model. Figure 2.1 describes the HAL-ASOS design flow.

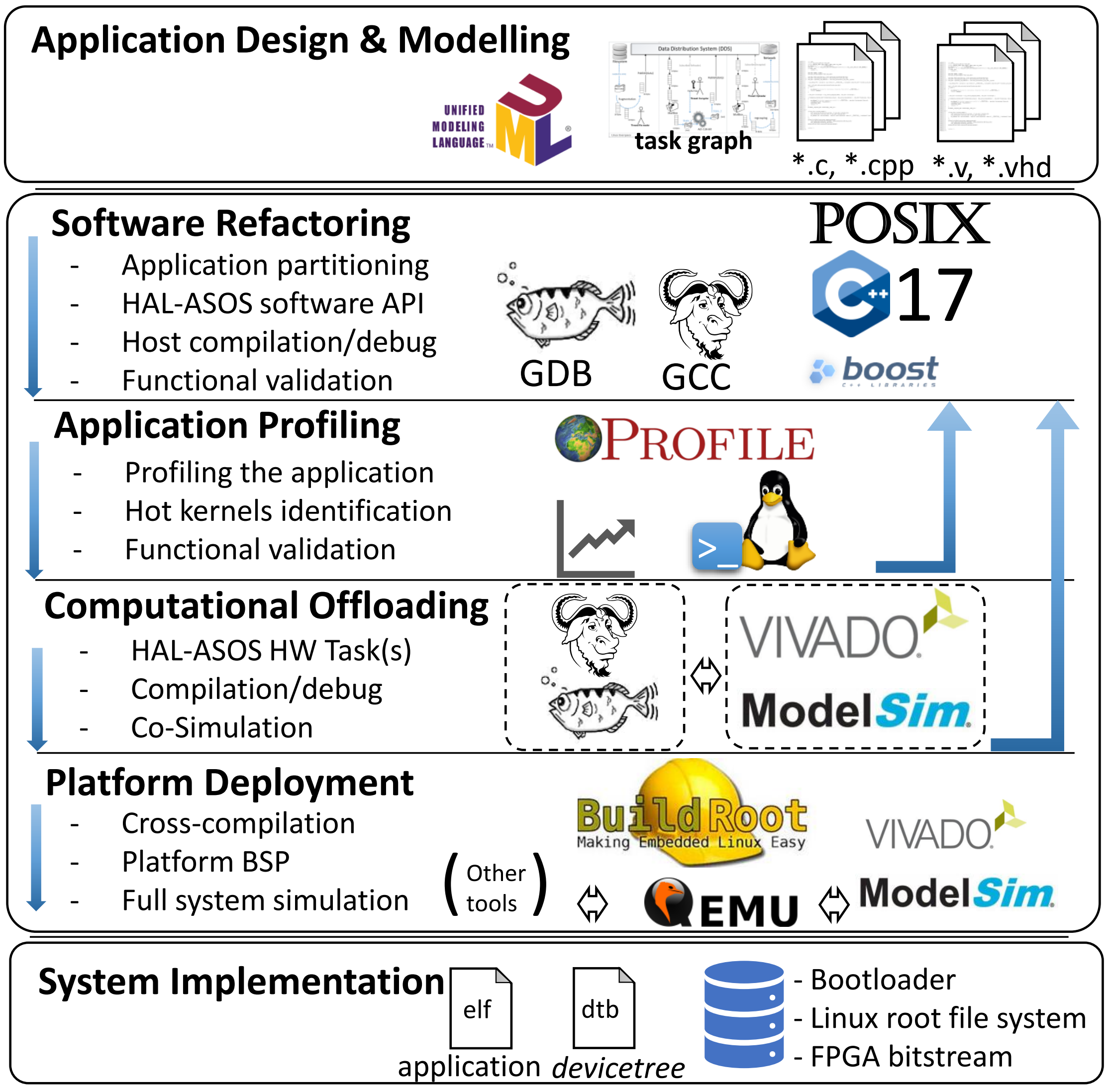


Figure 2.1: The HAL-ASOS design flow.

We enter the design flow in the software refactoring stage where the designer uses the framework by mapping the identified functional units into the HAL-ASOS programming model. The software tools will be used to compile the application prototype that will target the system to be developed. The framework requires compilation using the GCC 7 and distinct framework branches will allow the alternate use of the POSIX native thread runtime model, against the C++ runtime or boost library-based runtime.

For a better understanding of how the application model is translated into the host system CPU resources, a profiling stage needs to be applied, thus exposing the computational demands of the application. The designer will (re)partition the application in the search of the data-level and control-level parallelisms, where it exists and if exists, to take advantage of the multiprocessing units in the system. The potential for performance will be made clear with the computational demands of the application expressed across the set of profiled results. This development stage will put designers in a better position to make hardware and software design decisions, while targeting the pre-selected results and addressing any imposed restrictions. The hardware and software co-design can then proceed in a concurrent development approach to efficiently address each individual application requirements.

Entering the Computational Offloading is a one-step phase if the designer is integrating hardware and software solutions, or it can be a two-step phase when the complex design is the case. The hardware models for the candidates to computational offloading can be addressed in a C/C++ emulator model provided in the HAL-ASOS framework and used within the application prototype. The use of HAL-ASOS emulator name space provides the means for the user to implement a C/C++ based *HW-Task* and integrate it with the application prototype, opening the possibility to integrate high level synthesis tools such as Vivado HLS or MatLab to translate the model to appropriated RTL representation. Besides this, it also provides the designer with a clear understanding about the control algorithm to be implemented, and since not all candidates are suitable for offloading, or will be selected in the final design choice, it can help in anticipating these decisions in the design flow and avoid development and the time-consuming validation efforts of functional units that are not suited for computational offloading.

The accelerator model is applied to the selected candidates when offloading the computation and the model provides a complementary set of functional units, intended to ease the integration of the accelerator within the application. The use of such model is supported by RTL packages at user- and kernel-levels and these were designed to ease the programmability. To assist this development stage in the functional unit's validation, a co-simulation model will unify the application into a single set of functional results. The designer can then access these results, re-evaluate design decisions and iterate between the SW and the HW tasks in the search of the optimal solution. The Co-simulation model provides a unified simulation environment where the user can subject the designed hardware units to the application real demands as opposed to an isolated development and validation. The supported RTL simulation tools are Xilinx's Vivado and Mentor Graphics' ModelSim.

The platform selection is the last phase in the design flow and it occurs when all software and hardware components are fully developed and well consolidated, closing the development phase and sentencing the application to the underlying hardware. To assist in this design stage, the tool provides a Full simulation model that being based on the previous co-simulation principles, it considers the full hardware and software layers in the system. For that, it selected a platform emulator such as QEMU [23], and extended the QEMU functionalities to the surrounding simulation tools. The full simulation model will provide the user with a clear view of the application deployed to the target platform, and it will assist this development stage in addressing any potential glitch that can simultaneously conflict with the Application, the OS and the underlying hardware units. It will allow the user to strip down the several levels of implementation in the system and debug each one as an integrated part, as opposed to the efforts of reproducing a failure in isolated test fixture. For completeness, the platform binaries are also validated and usually the Buildroot [24] tool is used to provide the host with the necessary software packages.

The target validation will proceed beyond this design flow and the designer will confront the developed solution with the initial design expectations. At this stage, any significant change to the system usually represents a costly decision, that will ultimately result in quality degradation.

The main concern of this design methodology was to mitigate the technological gap between the hardware and software concepts, ease the programmability to explore the synergies between the hardware and software concepts, and provide the designer with the means to quickly integrate the implemented hardware units with the application, but also, care was taken to provide the user with the necessary level information that will allow anticipation of potential problems in the early stages of design where they are properly addressed.

2.2 Programming model

In this thesis, an object-oriented multithreading programming model is proposed for HAL-ASOS framework. One that integrates software threads and hardware accelerators in a unified and customized design, that will assist the development in partitioning the application and offloading the critical workload functionality to the accelerator model. The purposed programming model follows the standard of Multithread Programming Models for C/C++ applications on Linux OS, where the elementary processing units are implemented by the software threads.

A software thread usually represents a precise flow of execution inside an application, and is commonly delimited by the smallest sequence of programmed instructions that can be managed independently. In the majority of cases, this flow explores the code cyclic use and for that reason the execution is restricted within an application dependent loop. They can be seen as an integrant part of a process that executes concurrently with other threads in the same application and share memory resources. A multitude of threads in the same application can implement a parallel model that aims to promote the efficient use among the many sources of computation in the system.

The purposed model is centered around the class *Task* that symbolizes the thread in the traditional programming model. The *Task* can be semantically interpreted as software task that maps to the traditional thread, or as hardware task that is deployed into the accelerator model. Figure 2.2 shows a simplified UML of the class *Task*. It can be seen that the implementation is based in the C++ templates metaprogramming and by using specialization we address the configurability inside the application model. The class template qualifiers range between a predefined and a variable template pack that allows the user to configure and extend qualifiers to the application needs. The template parameter pack will be evaluated and matched to the available class implementations during the compilation phase.

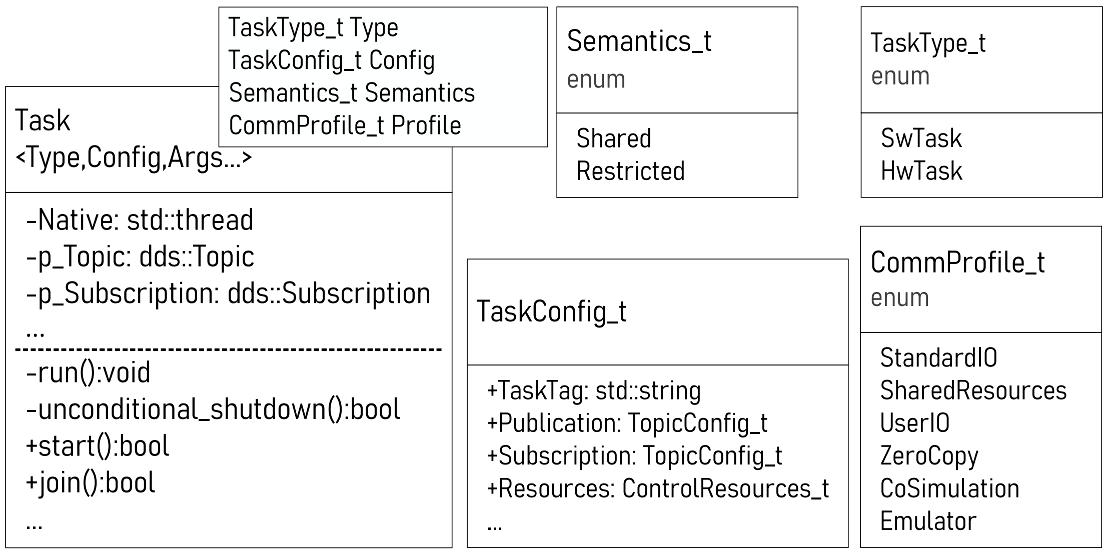


Figure 2.2: Simplified UML diagram of template class *Task*.

Any instance of the class *Task* demands a specialized *run()* member and a unique configuration that links to the class parameters (i.e., *TaskConfig_t*). This configuration includes a string *Tag* used for identification and log messages, a *Topic* and a *Subscription* setting, and a predefined set of members that address resource allocation inside class members.

The use of the *SwTask* as *Task* qualifier will match the class with an implementation that assigns the *Native* thread execution to a specializable *run()* member. The alternate *HwTask* qualifier will select an implementation that assigns the *Native* thread to the class internal services, as result of computation offload to the accelerator model. These services mostly include the boundary crossings between the software and hardware specialized circuitry while exchanging data between the class instance and the accelerator model, and to connect the accelerator with the Linux OS services.

The *Semantics* qualifier establishes data exchange semantics between *Shared* or *Restricted* models. A shared model will allow multiple references for the task data while sharing results with other existing *Tasks*. The restricted model will enforce a unique instance of data that can only be collected by one *Task* instance despite being shared with any definable number of *Tasks*.

The Profile qualifier will be matched in the *HwTask* specialized instances of the class *Task* and establishes the communication model with the hardware resources. The use of this qualifier affects the critical path and latency observed at the processing boundary crossings and it is limited to the predefined set. As an example, the qualifier *StandardIO* which is the default parameter will bind the class resource handling with the traditional set of Linux system calls and exchange data by use of the high and low memory regions. Alternately, the *UserIO* qualifier will reduce the usage of the system calls and map the hardware resources into the application address space.

Throughout the application development several crossed configurations can occur. The multitude of qualifiers combines specialized implementations at different layers of functionality. It is fair to say that for a specific application scenario the correct configuration might not exist. Instead, an optimized solution that explores the distinct trade-off(s) between different class implementations will be achieved. One that during the design space exploration is found to be the best solution for the desired performance metrics within the specificity of the application requirements.

2.3 Application Development

When creating applications, designers usually start with an idea in mind, a solution to a problem, an identified market need, or a technological vision to improve existing solutions. Generally, at this early stage of development, there is no reasonable expectation of what the application will be. To maximize creativity, designers should start by a quick prototype development, to rapidly envision the application and what the outcome results need to be, before facing any restrictions imposed by the commitment to the hardware and software stacks.

When we observe of most today's common computer applications, we realize they are internet-aware, and use the internet to exchange data, by using cloud services or any form of private servers. But within the Internet connectivity, applications are facing a growing need for security, leading developers to rely on complex cryptographic algorithms, that target specific requirements not suited for implementation in the most generic embedded devices.

Traditionally, designers use standard encryption algorithms to cope with security needs and a common example is the use of the Advanced Encryption Standard (AES). The AES is a well-established algorithm that operates on blocks of 128-bit plain data, and is available in three different cipher lengths: 128, 192 and 256-bit [25]. It is classified as symmetric-key algorithm which means that the same cipher is used for both operations, encrypting and decrypting the source data. Each of the 128-bit blocks represents a 4x4 bytes matrix, also designated by state S_i, where i is the consecutive block number in $\{1, ..., n\}$ blocks of data. Figure 2.3 helps to illustrate conceptual implementation of the AES-128 algorithm.

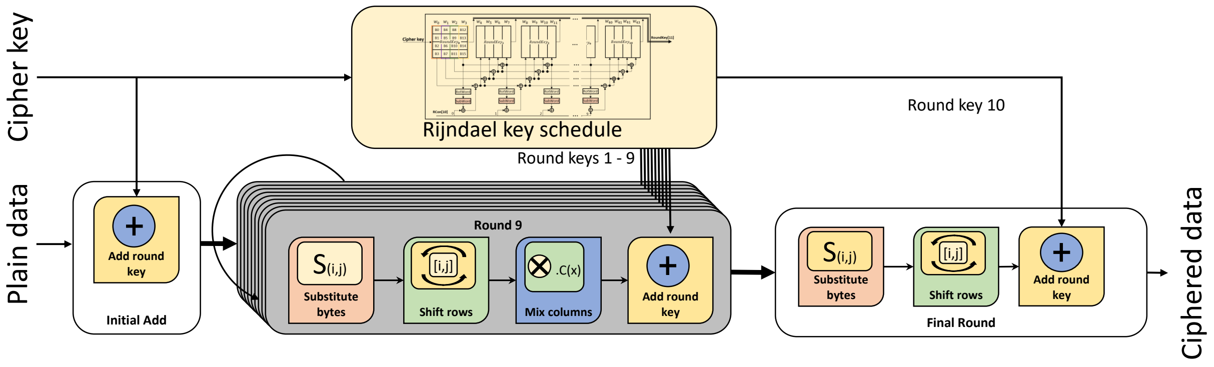


Figure 2.3: Overview of the AES-128-bit algorithm.

Using the Rijndael key schedule, the input cypher will be expanded into ten additional ciphers and used in the successive rounds that establish the encryption/decryption algorithm. As so, the encryption of a state S_i can be summarized to one initial adding operation, using the input cipher, and ten subsequent rounds using one of ten expanded ciphers. The rounds one to nine are identical and include four stages, namely: substituting bytes; shifting rows; mixing columns; and adding the round cipher. The state S_i encryption concludes with the tenth round, similar to the previous nine but skipping the mix columns stage. Implementation details about the AES-128 can be consulted in the Appendix E.

We will consider the example of an application that needs to upload files through the internet and selects the 128-bit AES algorithm to enforce security. The example will target a generic embedded device here referenced as Machine 1.

The simplified architecture for the Machine 1 application can be decomposed into three processing threads: a 'File reader', that polls on the OS file system for files, reads the file contents and fragments data into adequate size before submitting to the application internal structures; an 'Encryptor' thread that implements the AES-128 algorithm, collects the fragmented plain data and converts them into ciphered data; and an 'Uploader' thread that regroups the ciphered fragments and synchronizes the file transfers through the internet. Figure 2.4 depicts this organization.

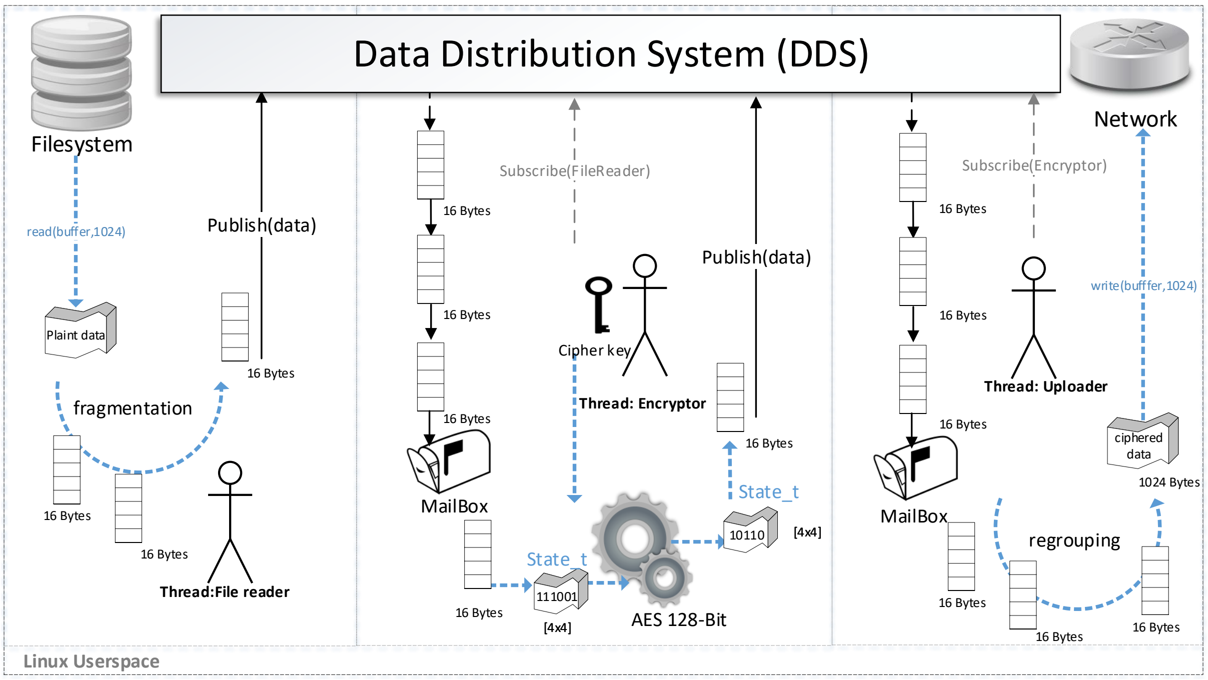


Figure 2.4: Machine 1 application - task graphical representation.

This example gravitates near the data-centric class of applications, where data is the main concern to the system and they tend to view data manipulation as the most important part of the work. Consequently, threads will need an efficient way to exchange data and for that we will use a Data Distribution System (DDS) provided as part of the HAL-ASOS framework. The DDS will link threads through an exchange model based on Publish-Subscribe, where the 'FileReader' will create a topic that is subscribed by the 'Encryptor' thread and consequently, this thread creates another topic that is subscribed by the 'Uploader' thread. The DDS implementation follows a low-memory footprint model, where the multiple attempts on creating the same topic will result on copied references of the same object. If a multitude of subscribers exist in one topic, they will receive memory references of the same 'const' data. The memory resources involved are allocated by any of the topic publishers, and later released when a subscriber attempts to destroy the last memory reference.

2.4 Software refactoring

For the Machine 1 application, the thread 'File reader' will create a topic named *FileReader* using the DDS services. Continuing with processing, all the thread results will be published in that topic. The thread 'Encryptor' will subscribe the *FileReader* topic and will periodically receive any published data to that topic. Similarly, the 'Encryptor' thread will create a topic named *Encrypted* that is subscribed by the 'Uploader' thread. The 'create topic' and 'subscribe topic' operations consider a numeric tag for identification that is created from hashing the *TopicConfig_t* members. The application code for the Machine 1 is presented in Figure 2.5.

```
  9  #include "hal_asos.h"
 10  #define HLEN (sizeof(state_t))//16
     ...
 12  const hal_asos::TaskConfig_t TFRead = { "FileReader", //TaskTag
 13                 { "PlainData",HLEN,1,1 },//Topic
 14                 { "",0 }// No Subscription
 15  };
 16  const hal_asos::TaskConfig_t TEncrypt = { "Encryptor",
 17                 { "CipheredData", HLEN,1,1 },
 18                 { "PlainData",HLEN,1,1 }
 19  };
 20  const hal_asos::TaskConfig_t TUpload = { "Uploader",
 21                 { "",0 },
 22                 { "CipheredData",HLEN,1,1 }
 23  };
     ...
205  void hal_asos_demo::test_aes128_file_sw_threads(void) {
206    using namespace hal_asos;
207
208    Task<SwTask, TFRead> T0;
209    Task<SwTask, TEncrypt>T1;
210    Task<SwTask, TUpload>T2;
211
212    T0.start();
213    T1.start();
214    T2.start();
215    T0.join();
216    T1.join();
217    T2.join();
218  }
```

Figure 2.5: The Machine 1 application - SW task version source code.

Three *TaskConfig_t* structures were specified for each of the *Tasks* (lines 12 to 23). The 'File Reader' and the 'Encryptor' tasks will create the 'PlainData' and 'CypheredData' topics respectively, and use the same topic length specified by the 'HLEN' macro. Each class instance will link with the *TaskConfig_t* structure by template parameter (lines 208 to 210). Using the *SwTask* qualifier, all class instances were parametrized for software resource as result of initial development iteration. Not all qualifiers and configuration parameters were specified and default values predefined by the framework will be used. The task *start()* member will assign the native OS thread execution to the class specialized *run()* member. The execution will *join()* threads until completion of all operations and terminates releasing the allocated resources.

2.4.1 The File reader task

The algorithm for the ’File reader’ task consists of: reading blocks of plain data from the input file; fragment these into adequate size of 128-bit (16 bytes); and publish the resulting fragments in the DDS topic (line 59 to 66). The specialized *run()* member for the class ’File reader’ can be consulted in Figure 2.6. For simplicity, some lines were omitted and the full listing can be consulted on the Attached Listing:C.2.

```
10 #define HLEN (sizeof(state_t))
11 #define BLOCK_LEN 1024
   ...
26 template <>
27 void hal_asos::Task <hal_asos::SwTask, TFRead>::run(void) {
28   std::ifstream input_file;
29   std::shared_ptr<char[HLEN]> p_buff;
30   char* p_local_buff;
31   int InputFileSize, read_len, i, count = 0;
   ...
33   input_file.open(target_file.c_str(),ios::in|ifstream::binary);
   ...
53   input_file.read(p_local_buff+sizeof(int),(BLOCK_LEN-sizeof(int)));
   ...
56   while (this->StatusRunning && Read_len > 0) {
57     for (i = 0; i < Read_len; i += HLEN) {
58       p_buff = std::shared_ptr<char[HLEN]>(new char[HLEN]);
59       copy_len = mmin(HLEN, (int)(Read_len - i));
60       std::copy_n(p_local_buff + i, copy_len, p_buff.get());
61       this->p_Topic->publish(p_buff);
62       count++;
63     }
64     input_file.read(p_local_buff, BLOCK_LEN);
65     Read_len = (long)input_file.gcount();
66   }
67   input_file.close();
68   this->p_Topic->close_topic();
69   LOG_MSG << this->TaskTag << "finished...(" << count << ")\n";
70   delete[] p_local_buff;
71 }
```

Figure 2.6: File reader task - simplified *run()* member.

An input file containing one million digits of *pi* (“3.”+1.000.000 digits) was selected as source of data and is open for read in line 33. Once in the “main” loop, the thread will read successive blocks of 1024 bytes from the source file, fragment each block into smaller 16-byte blocks and publish the resulting data in the DDS topic (lines 58 to 61). After reading all source file content, the thread will exit the main loop by failing the condition at line 56 (*Read_len > 0*) and close the file and the DDS topic (lines 67,68). A closed topic will allow the subscribers to consume the remaining publications. The thread concludes issuing a log message that prints the number of blocks processed and releases all the allocated resources (line 67 to 70).

2.4.2 The Encryptor task

The simplified code for the 'Encryptor' task *run()* member is presented in Figure 2.7. Similarly, the full listing can be consulted attached in Listing C.1.

```
 81 template <>
 82 void hal_asos::Task <hal_asos::SwTask, TEncrypt>::run(void) {
 83   std::shared_ptr<dds::Publication> pLocal;
 84   std::shared_ptr<char[HLEN]> p_cyphered;
 85   std::shared_ptr<const char[]> p_plain;
    ...
 90   set_cypher_key(key);
 91   key_expansion();
 92 
 93   while (this->StatusRunning && ret > 0) {
 94     ret = this->p_Subscription->take_publication(pLocal);
 95     if (ret) {
    ...
101       p_current_state = (state_t*)p_cyphered.get();
102       add_round_key(0);
103       for (round = 1; round < NROUNDS; ++round){
104         subst_bytes();
105         shift_rows();
106         mix_columns();
107         add_round_key(round);
108       }
109     subst_bytes();
110     shift_rows();
111     add_round_key(NROUNDS);
112     ret = this->p_Topic->publish(p_cyphered);
113     }
114   }
    ...
116   this->p_Topic->close_topic();
117   this->p_Subscription->terminate_subscription();
118   LOG_MSG << this->TaskTag << "finished.(" << pcount << ")\n";
119 }
```

Figure 2.7: Encryptor task - simplified *run()* member.

The input cipher is set and the key expansion schedule is executed (lines 90 and 91). Once in the main loop, the thread will collect 4x4 bytes (S_i) of plain data from the DDS subscription (line 94). The ten rounds that complete the AES-128 algorithm will encipher the received plain data, and the iteration of S_i concludes with publishing the resulting data (ciphered data) in the DDS topic (lines 102 to 112). A negative return or a null '*pLocal*' pointer in the *take_publication* call (line 94), will force the execution to break the main loop (line 93). The thread concludes closing the topic, terminating the subscription and issuing a log message that prints the number of blocks processed (lines 116 to 118).

2.4.3 The Uploader task

The simplified code for the specialized *run()* member of the task 'Uploader' is presented in Figure 2.8. The task uses the name space *networking* from HAL-ASOS framework that provides eased access to the OS network subsystem. An instance of *CSocket<Client>* will be used to establish a connection with a network server for uploading files. A successful connection will allow the thread to proceed into the main loop (lines 157-171). The main loop consists of receiving ciphered fragments from the DDS subscription

```
134 template <>
135 void hal_asos::Task <hal_asos::SwTask, TUpload>::run(void) {
136   using namespace hal_asos::networking;
137   int ret = 1, count = 0, index=0;
138   char* p_local_buff;
139   std::shared_ptr<const char[]> p_buff;
140   std::shared_ptr<dds::Publication> pLocal;
141   CSocket<Client> Soc;
    ...
157   this->StatusRunning = Soc.open_connection();
158   while (this->StatusRunning && ret > 0) {
159     while (index < BLOCK_LEN && ret > 0) {
160       ret = this->p_Subscription->take_publication(pLocal);
161       if (ret) {
162         p_buff = pLocal->get_reference();
163         std::copy_n(p_buff.get(),pLocal->get_len(), p_local_buff + index);
164         index += pLocal->get_len();
165         count++;
166       }
167     }
168     if (index > 0) {
169     ret = Soc.safe_write(p_local_buff, index);
170     index = 0;
171     }
172   }
173   Soc.close_connection();
174   this->p_Subscription->close_subscription();
175   delete[] p_local_buff;
176   LOG_MSG<<this->TaskTag<<"finished...("<<count<<")\n";
177 }
```

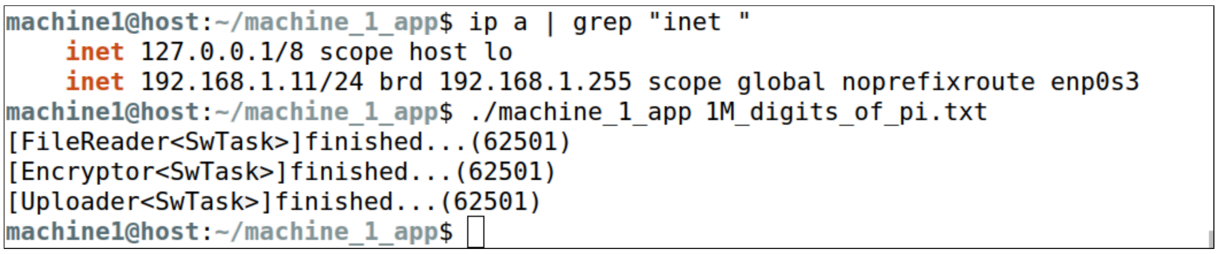
Figure 2.8: Uploader task - simplified 'run()' member (full listing:C.3)

and regrouping data into a network-adequate block size by using *local_buffer* (lines 159-167). Once *local_buffer* is complete the results are transferred over the network (line 169). A negative return on *take_publication* or a network communication error, will break the main loop by failing the condition *'ret > 0'* on line 157. The thread concludes closing the network connection, terminating the subscription and issuing a log message that prints the output results (lines 173-176).

2.4.4 Functional validation

The application for Machine 1 was selected for compilation at Host environment using Linux OS. Similarly, a second machine, Machine 2, was configured with a server application to provide a connection and receive the encrypted data. Details of the Machine 2 application are out of discussion in this thesis but.

Machine 1, using the network subsystem with IP *192.168.1.11*, connects with the Server application on Machine 2 for uploading the encrypted file *1M_digits_of_pi.txt*. The output results of this test are presented in Figure 2.9. The log messages from the three *Tasks*, in Figure 2.9, indicate that 62,501

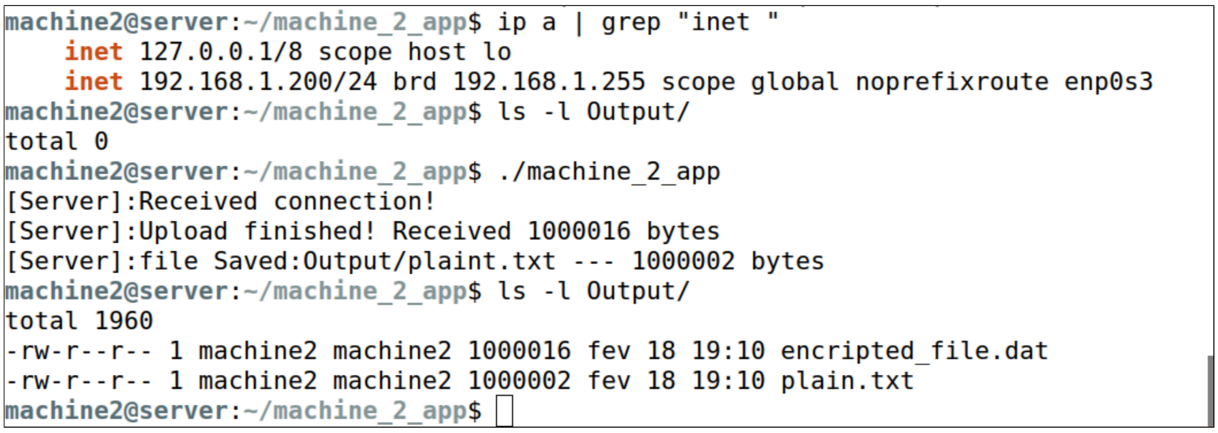


```
machine1@host:~/machine_1_app$ ip a | grep "inet "
    inet 127.0.0.1/8 scope host lo
    inet 192.168.1.11/24 brd 192.168.1.255 scope global noprefixroute enp0s3
machine1@host:~/machine_1_app$ ./machine_1_app 1M_digits_of_pi.txt
[FileReader<SwTask>]finished...(62501)
[Encryptor<SwTask>]finished...(62501)
[Uploader<SwTask>]finished...(62501)
machine1@host:~/machine_1_app$ 
```

Figure 2.9: Machine 1 - SW-only application output for one million digits.

blocks were processed and resulted in a file length of 1,000,016 bytes. The input file *1M_digits_of_pi.txt* contains 1,000,002 bytes and the output exceeds this value in 14 bytes, as result of including a minimal header that indicates the file length and some padding to align the number of fragments with the state matrix length.

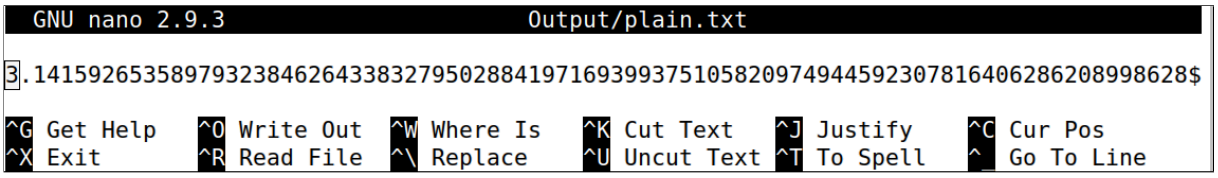
At Machine 2, using IP *192.168.1.200*, a connection was accepted and the file was received successfully. The application log from this test can be consulted in Figure 2.10. The received data resulted in an



```
machine2@server:~/machine_2_app$ ip a | grep "inet "
    inet 127.0.0.1/8 scope host lo
    inet 192.168.1.200/24 brd 192.168.1.255 scope global noprefixroute enp0s3
machine2@server:~/machine_2_app$ ls -l Output/
total 0
machine2@server:~/machine_2_app$ ./machine_2_app
[Server]:Received connection!
[Server]:Upload finished! Received 1000016 bytes
[Server]:file Saved:Output/plaint.txt --- 1000002 bytes
machine2@server:~/machine_2_app$ ls -l Output/
total 1960
-rw-r--r-- 1 machine2 machine2 1000016 fev 18 19:10 encripted_file.dat
-rw-r--r-- 1 machine2 machine2 1000002 fev 18 19:10 plain.txt
machine2@server:~/machine_2_app$ 
```

Figure 2.10: Machine 2 - Server application Output.

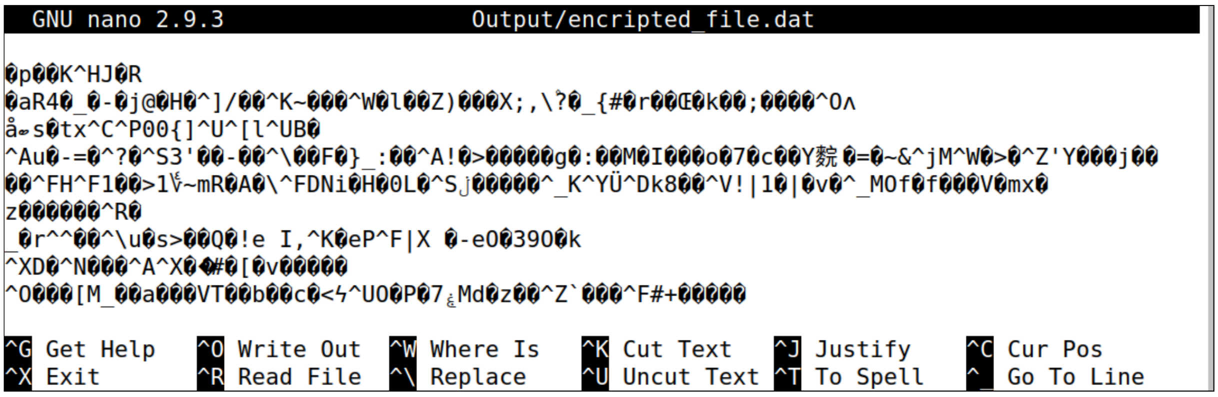
encrypted file containing 1,000,016 bytes that was stored in the *encrypted_file.dat* file. The Inverse form of the AES-128 algorithm was then executed and after removing the transfer header, the remaining 1,000,002 bytes were stored in the *plain.txt* file. A portion of the plain file contents can be seen in Figure 2.11. The *pi* number sequence is represented in a readable form, and that demonstrates the functionality of the system.



```
  GNU nano 2.9.3                         Output/plain.txt

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862$

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell    ^_ Go To Line
```

Figure 2.11: Machine 2 - contents of plain data file after decryption.

The contents of the *encrypted_file.dat* are presented in Figure 2.12, and show no similarities to the original data, thus ensuring a confidentiality level that can be used when sending sensitive data over the Internet.



```
  GNU nano 2.9.3                    Output/encripted_file.dat

�p��K^HJ�R
�aR4�_�-�j@�H�^]/��^K~���^W�l��Z)���X;,\?�_{#�r��Œ�k��;����^O∧
å⌀s�tx^C^P00{]^U^[l^UB�
^Au�-=�^?�^S3'��-��^\��F�}_:��^A!�>�����g�:��M�I���o�7�c��Y䊔�=�~&^jM^W�>�^Z'Y���j��
��^FH^F1��>1V̋~mR�A�\^FDNi�H�0L�^Sȷ�����^_K^YÜ^Dk8��^V!|1�|�v�^_MOf�f���V�mx�
z������^R�
_�r^^��^\u�s>��Q�!e I,^K�eP^F|X �-e0�390�k
^XD�^N���^A^X��#�[�v�����
^O���[M_��a���VT��b��c�<ϟ^UO�P�7ݹMd�z��^Z`���^F#+�����

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell    ^_ Go To Line
```

Figure 2.12: Machine 2 - contents of encrypted file received.

The purpose of this application, the Machine 1, was to introduce some of the HAL-ASOS functionalities, but also to establish a common ground in development between the traditional software design techniques and the hardware accelerated application development, while using the HAL-ASOS framework. No hardware acceleration concepts were yet used, as they need to be applied from design decisions that ensure a profitable resource allocation and satisfy the overall design metrics. Also, no commitment to the underlying hardware was established, as this decision will be made later in the design flow, thus avoiding any constraints to initial development. The example also sets a common ground to well established applications that need to be refactored for performance and offload computation to specialized hardware.

The next step in the design flow is the profiling of the application, and this stage will help to clarify how the application needs are translated into the system resources.

2.5 Application profiling

As soon as the prototype application reaches an acceptable level of functionality, it is time to analyze how well the application performs in the host system and try to understand the program intrinsic behavior. A profiling tool can be used to identify all kind of bottlenecks in the system. Generically, we will be interested in the performance profile, to identify hotspots in the program where the system spends significant amount CPU time. Considering that the framework was specifically designed for Linux embedded systems, we recommend using the *OProfile* tool for this stage in the design flow. Since the profiling will be performed in the host development environment other similar tools can also be used.

2.5.1 Profile tools

The *OProfile* tool is a system profiler for Linux and it is available since kernel version 2.4. This tool will target all parts in a Linux system, from an application, a set of processes or threads, kernel code or interrupt handlers, a subset of system active processors, or ultimately the entire system. Conceptually, this tool is classified as a statistics-based profiler since it operates by collecting strategic data at periodic time intervals. Today's CPU architectures provide hardware performance counters that record the occurrence of specific events without the need for additional code instructions. A timing interruption triggers data collection and signals the profiling application about the existing new data. Post-profiling tools will convert this data into a human readable file that contains the desired profile results. Detailed information about *OProfile* features and events are specific to each CPU architecture and can be consulted at [26].

A similar profiling tool is *gprof*, a superset of the Linux prof command, included in the GCC tools. The tool results also focus on were the CPU spent time inside the application and includes the invocation count to each of the application functions. The *gprof* tool demonstrated higher level of impact in the execution since it links pre- and post-call routines in the application binary. Also, experiences realized with this tool in the HAL-ASOS framework, revealed less stability across many profile trials that due to its strong software dependent nature. Detailed information about this tool can be found at [27].

2.5.2 Profiling Results

The profile results of the Machine 1 application were obtained using *OProfile*. The choice of the *OProfile* was mainly based on the acceptance of the tool in the Linux community, its strong hardware dependent nature, the tool stability across many profiling iterations and high number of supported architectures. For the Machine 1 application we will target the CPU cycle count event that automatically decrements a hardware counter every time the CPU completes an execution cycle. The tool will register the program address(es) every time the counter value reaches zero, and to achieve effective results, a considerable number of executions was used. Listing 2.1 shows a *bash* script used to assist in the profiling of Machine 1 application.

Listing 2.1: Profile script used on Machine 1 application.

```
1   #!/bin/bash
2   if [ $# -eq 3 ]; then
3
4   sudo rm -rf oprofile_data/
5     echo --------------------------------------------------
6     echo "Profiling $1 for $2 iteration(s)"
7     echo --------------------------------------------------
8     for i in $(seq 1 $2)
9     do
10      echo ------------------------------------------------
11      echo interation $i
12      echo ----------------------------------------------------------
13      sudo operf --append --event CPU_CLK_UNHALTED:$3:0:0:1 ./$1
14      echo ----------------------------------------------------------
15    done
16
17    opreport --accumulated\
              --exclude-dependent\
              --exclude-symbols=_GLOBAL_OFFSET_TABLE_\
              --symbols > iterationGlobal.perf
18  else
19    echo "Wrong command <execuable> <n_iter> <sample_rate>"
20  fi
```

The command '*operf*' is used to launch the application binary with the OProfile predetermined settings (line 13). The argument CPU_CLK_UNHALTED indicates the desired hardware event and the desired sample rate was set to 6.000 CPU cycles. The –*append* switch will append the profile data across several profile sessions and the script launched the application for 100 times (lines 7 to 14). The *opreport*, in line 15, outputs the results for the profile session with arguments that are used to establish the accumulated results across the report entry lines and confine them to the application binary.

Figure 2.13 shows a simplified version of the profile results for the Machine 1 application. Analyzing these results, one can conclude that the four AES-128 round used functions (lines 4 to 11) are responsible for nearly 41% of CPU time spent in the application code. The percentage of line 13 shows that *Task<Encryptor>* spends approximately 1% of the CPU time executing local code or out of any function calls, and that between 1% and 2% of the assigned application time is being spent inside the functions that relate to the DDS subsystem (lines 18-71). The workloads are evenly distributed across the AES-128 algorithm as no hotspots consume excessively amounts of CPU time. An estimate of 41% can be faced as potential for performance contribution in the application, and it becomes clear which are the candidates to the computational offload.

```
 1  CPU: Intel Haswell microarchitecture, speed 3500 MHz (estimated)
 2  Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
 3  with a unit mask of 0x00 (No unit mask) count 50000
 4  samples cum. symbol name
 5  15.86% 15.86% mix_columns()
 6  13.75% 29.61% add_round_key(unsigned char)
 7  11.74% 41.35% subst_bytes()
    ...
11  1.44% 52.54% shift_rows()
    ...
13  0.76% 54.43% hal_asos::Task<(hal_asos::TaskType_t)0,TEncryptor>::run()
    ...
18  0.61% 57.67% Topic<(hal_asos::dds::Semantic)1,TFReader>::publish<16>()
    ...
24  0.50% 61.00% hal_asos::dds::Publication::get_len()
    ...
29  0.44% 63.30% Topic<(hal_asos::dds::Semantic)1,TEncryptor>::publish<16>()
    ...
50  0.31% 71.20% Subscription<TEncryptor>::take_publication()
    ...
71  0.29% 72.10% Subscription<TUploader>::put_publication()
    ...
76  0.26% 75.62% hal_asos::Task<(hal_asos::TaskType_t)0,TUploader>::run()
    ...
82  0.23% 79.51% hal_asos::Task<(hal_asos::TaskType_t)0,TFReader>::run()
```

Figure 2.13: Profile results of the Machine 1 application.

2.5.3 Conclusions

The top listed functions in the profile report, generally include the candidates to computational offloading and the decision needs to be validated according to the application requirements. Since requirements are at the root of the application design, they altogether will influence these candidate selections. Such an example could be security-related requirements. When consider that 'static' hardware is not easily patched and forced to abnormal behavior, or that cipher keys are not so easy to access when they are encapsulated by custom and closed hardware, the requirement is favorable to offloading. But the power-aware requirements might conflict with this decision, since more functional units might demand for more energy consumption, depending on the implemented HW behavior when compared with its software counterpart. This can represent a need for design power estimation, or the purposed design being evaluated or re-designed to meet this requirement. When a multitude of choices play in favor or against each other, is fair to say that the right choice might not exist. One can only achieve the optimized solution that attempts on consolidating the overall of the design metrics.

Considering this development stage, and the purpose of these overview, we will proceed with the offload of the 'Encryptor' task. Since the development materializes into a deeming cycle, other solutions might become attractive in-between iterations.

2.6 Accelerator model

The HAL-ASOS Accelerator model can be decomposed into a user defined *HW-Task*, a parametrizable *HW-Kernel* with three differentiated transfer channels that aim to explore distinct bus technology dependent-interfaces. A simplified representation of this model can be seen in Figure 2.14 and includes a minimal Host system representation. To avoid miss confusion with the term host, we clarify that the host development system is the system hosting the toolchain used to compile the several implementation levels of this application, while the Host system is the target platform that hosts the accelerator model.

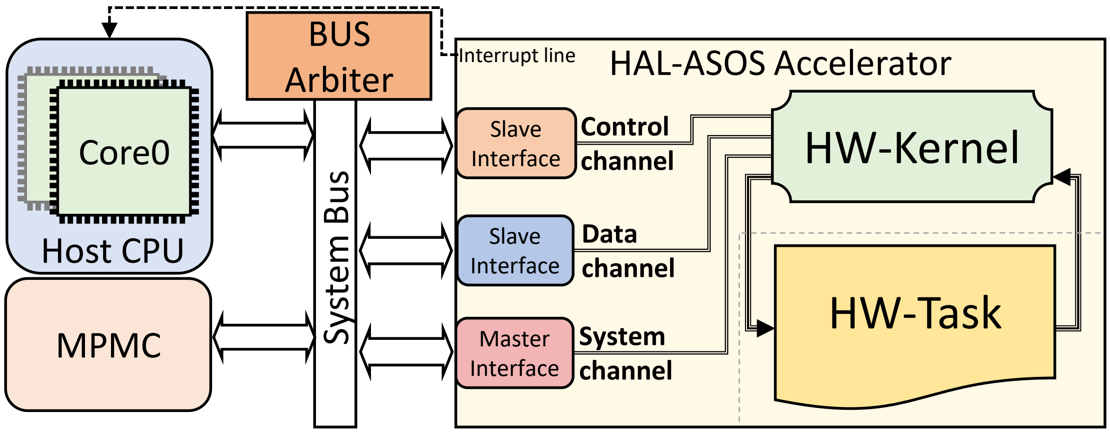


Figure 2.14: HAL-ASOS Accelerator model integrated into Host platform.

The HW-Task plays the central role in the design and uses the *HW-Kernel* to interact with the host. Optional implementations allow a *HW-Task* tightly integrated in the accelerator model or a loosely design *HW-Task*, as an independent component. The transfer channels are platform dependent and establish differentiated data exchange with the Host system. These include: a fast, word-rated and low-bandwidth channel, used for control-oriented transfers; an optimized speed, byte-rated and high-bandwidth channel, used for large and data-oriented transfers; and an optimized speed, byte-rated channel, used by *HW-Task* to access the system memory. Platform-classified model implementations will include PLB or AXI bus interfaces.

The Accelerator is a native 32-bit big-endian machine, but 64-bit word can be applied system-wide. An interrupt line is mandatory and allow the accelerator to synchronize with the Host OS.

2.6.1 Hardware Kernel model

The *Hardware Kernel* model translates the Host system to the *HW-Task* and provides integration at hardware and software levels. The model includes a *Kernel Core* implementing the Control unit with a system-level datapath, and a collection of functional units that implement the service-level Datapath. Figure 2.15 presents a simplified model of the *HW-Kernel*. The Control unit uses single address microcode design to encode the set of HW system calls. The system-level datapath implements the multiplexing and demultiplexing of the system call parameters into the service-level datapath. The *M00_Kernel* and *S00_Task* are the master and slave interfaces of system call bus, used to connect with the interfaces in the *HW-Task*.

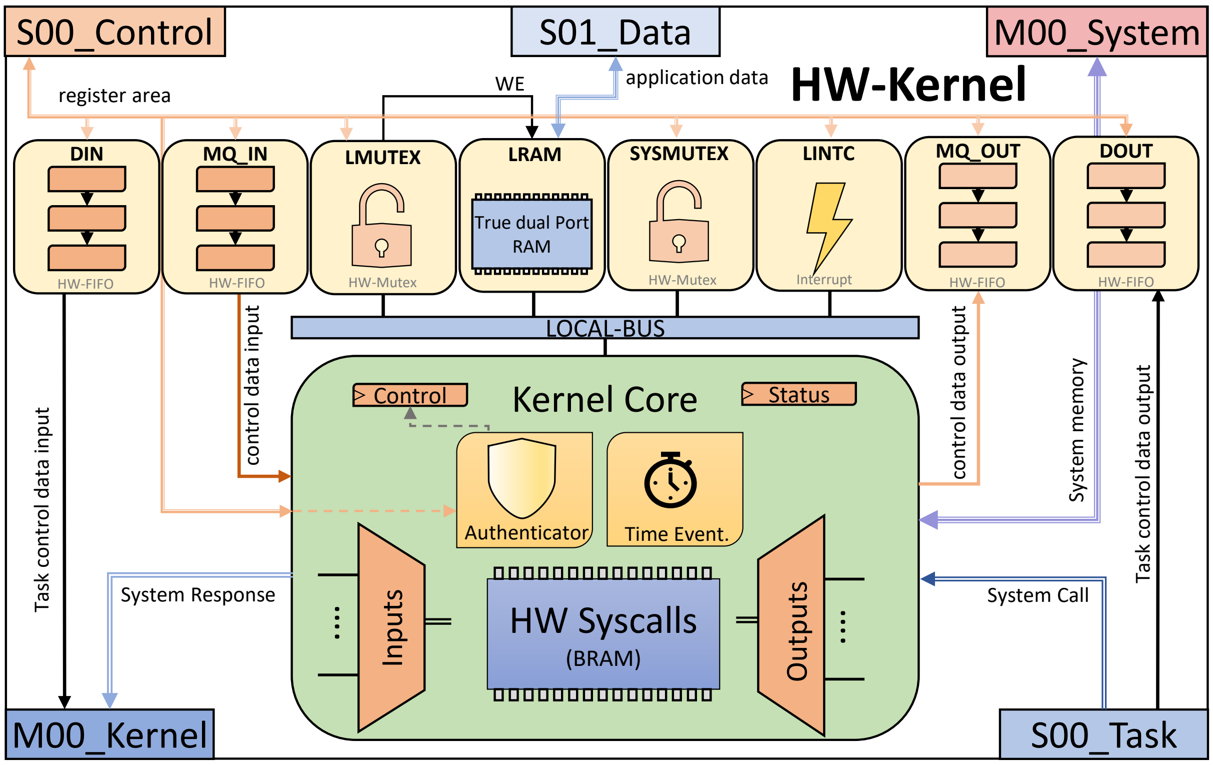


Figure 2.15: Hardware Kernel simplified model.

The *Kernel Core* is responsible for the time management and provides waiting event coupled with time-out functionalities and a parametrizable task sleep. The *Control* and *Status* registers will allow the host system to interact with the *HW-Kernel*. To preserve the *HW-Kernel* status, any control operation issued by the CPU cores, is for-warded via Authenticator unit that validates permissions before authorizing a write operation. As consequence of the microprogramming technology used for the HW system calls, the *Kernel*

Core implementation results in a static unit that is independent of the *HW-Task* implementation and can be configured or changed by applying microcode updates.

A service-level datapath includes (1) a dual-port and bi-directional message-queue used for messaging control information within the host system services, (2) a dual-port bi-directional data-FIFO available for HW-Task generic use (3) a Local Interrupt Controller (LINTC) that allows synchronization with the Linux OS, (4) a true dual-port generic purpose Local RAM (LRAM) for data exchange and temporary storage, and (5) two dual channel *HW-Mutexes* that implement mutual exclusion with the accelerator model. The latter are directly coupled with the LRAM and a system memory region allocated at boot-time. At kernel-side, dedicated interfaces are used to manage each of the *HW-FIFO*, while the remainder of the functional units are accessed through custom *Local-Bus*. The *M00_System* interface is used to access a kernel-specific region in the host system memory.

S00_Control and *S01_Data* offer the control- and data-oriented transfer interfaces for host system accesses to the *HW-Kernel* functional units. The *S01_Data* implements a byte-oriented bidirectional interface used exclusively to access the LRAM. The reminder of the functional units, link to the *S00_Control* in a bidirectional register type interface. The complete set of units that integrate the *HW-Kernel* model are also parametrizable and are made available to the host system through the Linux integration model.

2.6.2 Hardware Task model

The Hardware Task model provided by the HAL-ASOS accelerator follows traditional architecture modelling techniques. The design can be partitioned in a Control Unit and a collection of user-defined functional units that compose the Datapath. Figure 2.16 presents a simplified model of the *HW-Task*. The Control synchronizes the task internal implementation while the Datapath implements the task algorithm. Using the HAL-ASOS VHDL packages, the Control can also synchronize externally with the other tasks in the system. Mainly, these packages provide a set of services that are divided into user- and kernel-levels. The user-level implementation, considers the task-related functionality and only uses local resources, while the kernel-level considers a more service-related implementation, where it is allowed to talk to the local hardware or system resources, including the Linux OS.

Both the user and kernel services are manly implemented using VHDL procedures and the HW-Task will link the datapath to each procedure implementation. A VHDL procedure, denotes a subprogram description that is implemented in zero simulation time. It differs from the VHDL functions, in the sense that, it

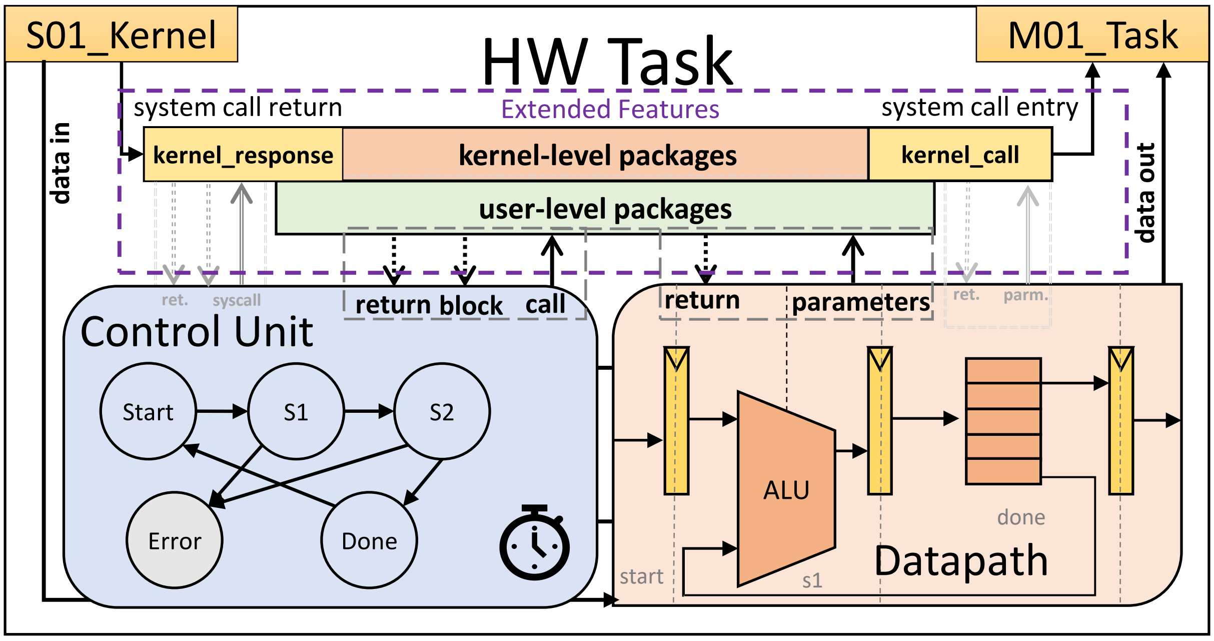


Figure 2.16: Hardware Task simplified model.

can receive parameters as input and output, and using signal driver capabilities, manipulate values that exist outside the subprogram. To accommodate the complexity of each procedure, the kernel provides generic FSMs for each of the RTL layers being concurrently executed. In doing so, the task description scales, implementing procedures from the user or kernel packages. At user package, each procedure is implemented using functionality-based procedures from the kernel package. While executing, the *kernel_call* interface will be updated by the procedure RTL and the kernel will execute HW system calls accordingly.

Generically, once the *HW-Task* control-path implements a procedure call, the Control unit is blocked and the synchronization is transferred to the *Kernel Core* control. The procedure will execute in a predetermined number of clocks that depends on each distinct implementation and value of the passed parameters. These parameters will be copied or forwarded to the procedure circuitry accordingly, and at completion, an exit path will ultimately enable back the task Control Unit. The kernel package uses the *kernel_call* and *kernel_response* registers, that link to the *HW-Kernel* system call infrastructure using *M00_Task* and *S00_Kernel* interfaces. A template for the *HW-Task* design is provided, and it includes a minimal control FSM with blocking functionality. The designer will then extend the FSM states to accommodate the HW-Task algorithm description. The implementation details of the *HW-Task*, the RTL packages or the *HW-Kernel* will be best discussed in Chapter 3.

2.6.3 Linux Integration

The HAL-ASOS Accelerator model integrates with the target platform OS at user- and kernel-levels. Due to the myriad of functional units in the model, a proper OS support requires a collection of device-drivers that efficiently export each functionality into the Linux user-space. Such a collection of drivers is best organized through a customized File System(FS). Figure 2.17 presents the HAL-ASOS FS structure.

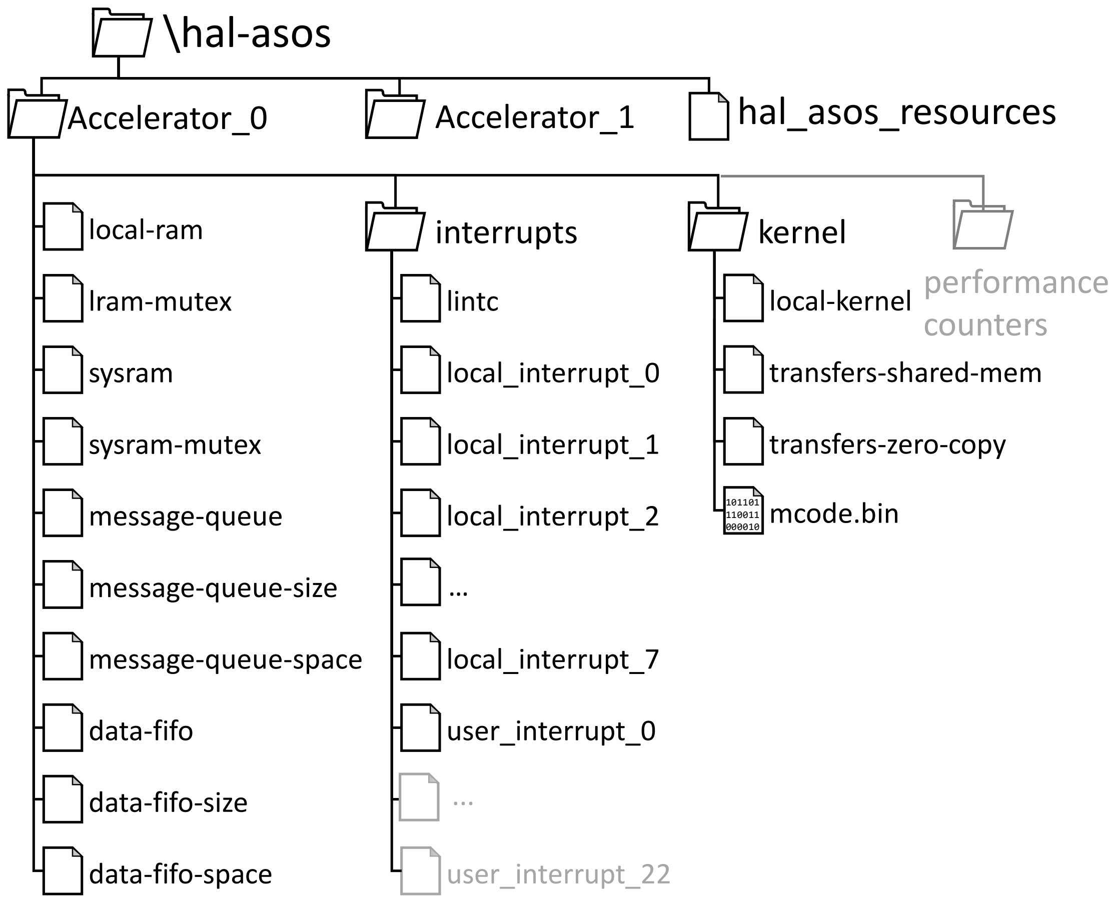


Figure 2.17: HAL-ASOS file system structure on Linux.

The HAL-ASOS file system mounts during Linux start-up and it can be found at the root of the Linux file system in the *hal-asos* folder. Any existing accelerators will be extracted from the device tree information that results from the deployment phase, and mapped into individual *Accelerator_x* folders. The folder name results from a configurable name tag in the accelerator listing, and a sequence number that counts the instances using the same tag. Inside the accelerator folder, the structure is organized in a kernel folder, an interrupt folder and a sub-set of virtual files that map the remainder of functional units in the Accelerator model. These include: the LRAM and the *local-mutex* (LMUTEX); the system memory region (i.e., *sysram*) and associated hardware mutex (*sysram-mutex*); the *HW-Kernel message-queue* with read-only *size* and *space* files; and similarly, the *data-fifo* with the *size* and *space* files.

The *interrupt* folder contains the virtual files that provide the synchronization between the software threads in the system and the accelerator. The *lintc* file represents the local interrupt controller and it uses seven native interrupts that are mapped to the *local_** files. Since a configurable number of user definable interrupts is also available, up until twenty-four *user_** files can be present in this directory.

Some of the FS functionality demand proper registration by using the *local-kernel* file in the *kernel* directory. The registration is based on exclusive ownership model and the *local-kernel* initializes a private structure containing a 32-bit key automatically hashed. The *transfer_** files map distinct profile interfaces between the software framework and the accelerator model, and the *mcode.bin* file is used to update the *Kernel Core* microprogram. These features will require validation using the previously generated key or otherwise will be denied.

A performance counters folder will be found if the accelerator model is active for performance metrics. Generally, these follow a hardware event counter model, coupled with synchronous clock timer model, that register the number of events and associated latency across the *HW-Kernel* functionality. One file for each active performance counter will be found in that directory, and reading them or using a '*cat*' command from the Linux bash, will output the current performance results in a conveniently formatted text message. The Performance functionality will be best discussed in Chapters 4 and 5.

At the application-level, to efficiently handle the exported model, the software framework provides the class *Proxy* that maps the FS functionality into a resource-oriented set of operations. A *Proxy* member will be found in all class *Task* instances that were specialized for *HwTask*. The transfer profiles are differentiated using distinct *Proxy* implementations that explore the different interfaces provided by the FS.

The HAL-ASOS file system is a Linux functionality provided by the framework and exists only in the target generated platform binaries, mostly because of its hardware and device tree dependencies. For that reason, it will only be available on the System implementation phase, or alternately, when applying the Full simulation model to the platform binaries, in the deployment phase. When compiling the application for the development host, an emulated version of the file system is implemented by the Emulator model.

2.6.4 Emulator Model

The Emulator model is a framework feature that assists in mapping the pre-selected candidates to computational offloading into the *HW-Task* structure. By using a software development perspective, it promotes a more flexible environment that exposes the task algorithm requirements and consolidates *HW-Task* interoperability in the application. This stage establishes a design iteration that aims to create a control algorithm that fits the application requirements and it is close to the RTL specification.

The Emulator Model is composed by a set of function-oriented entities that can be divided in two implementation layers: (1) the HW emulation layer, that includes the Accelerator described above; and (2) the software layer that emulates the Linux integration in the HAL-ASOS file system. This model follows a design approach that took the effort in describing the implementation details and available interfaces on each functional unit, and can ultimately be considered as a software description of the HW counterpart. The Emulator model is a C/C++ functionality implemented by the '*hal_asos::emulator*' *namespace*, and considers a unified implementation with purposed programming model. Figure 2.18 shows a simplified UML class diagram for the first layer. In that figure, it can be seen that the *AcceleratorModel* class integrates the *HwTaskModel* class and the *HwKernelModel* class, and the three altogether form a template model qualified by the accelerator configuration. For simplicity, the following section will refer the Emulator model entities by the name of their representative hardware counterparts.

The user packages are private members of the class *HW-Task* and a similar implementation can be found in the *HW-Kernel*, with respect to the kernel packages. A C++ Friend definition will allow the *HW-Task* to access the private parts of the *HW-Kernel* and talk directly with the kernel packages. The *HW-Task* plays the central role in the emulator as it stimulates the model using its *Native* thread member. Among other internal code, the thread loops using the *run_iteration* member, that considers the processing in an equivalent clock cycle of the *HW-Task*. The successive calling of this member will iteratively progress the task algorithm code. The designer will provide a specialized *run_iteration* member that contains the task algorithm implementation, and since it is using the same configuration as template qualifier, it will be linked to the class unique code.

A call to any of the user package members will ultimately result in a kernel HW system call, and the execution proceeds by copying the parameters to the member *KernelICall* that binds the two classes. When executing the *HW-Kernel* member *execute_sys_call*, it will transfer execution to the kernel model and the system call will be implemented. Any data exchange or control updates in the kernel set of

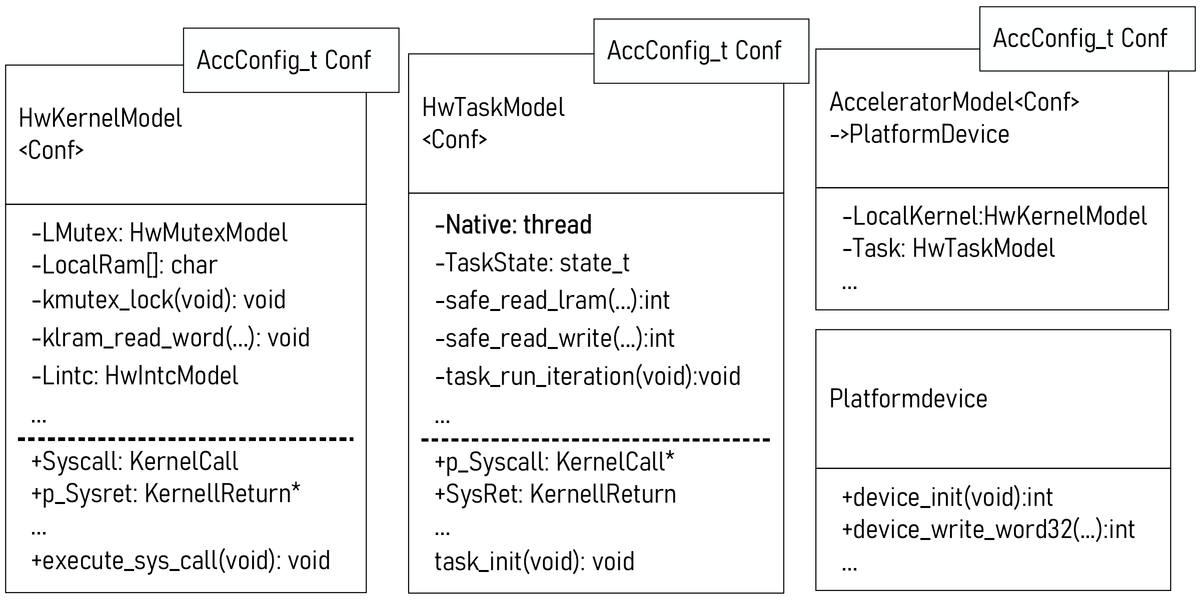


Figure 2.18: Accelerator Model Emulator - Simplified UML class diagram.

functional units will be performed. At conclusion, the resulting data will be found in the *KernelReturn* member and the execution returns to the *HW-Task*

At the Host system, the Accelerator is accessed via *PlatformDevice*. This structure encapsulates the model representation in a common form, and similarly to the Linux device model, interfaces the Accelerator via *read* or *write* set of operations that is used by SW Layer of the Emulator.

2.7 Computational offloading

In this design stage we leverage the computation offload by applying a two-step HW description. We begin with the specialization of the HW *Encryptor* task that includes the selected candidates from the profiling phase. We then apply the Emulator model to ease the HW partitioning and gain perception about the new task implications, and once completed, we apply the Accelerator model and proceed to the hardware description stage. At completion, we verify the design by applying the Co-Simulation model and establish a unified simulation environment, where this new application snapshot will be validated.

In a real design scenario, multiple attempts to map application functionality are most likely to occur, and can result from metric-driven improvements or control algorithm refinements. But to reduce the design iteration on this example application, we anticipate two decisions before proceeding into the computation offload phase. First, using a behavioral modelling style, we will describe the *HW-Task* and map the pre-selected candidates with minimal application refactoring. The task control unit will also be described following a close to emulator description style. Then, the resulting accelerator will be functionally validated using the co-simulation model. And finally, to conveniently explore the interaction between the Accelerator model and Host system, we will execute another design iteration and offload the file and network socket functionalities into a unified *HW-Task* design, that assumes the role of a standalone (SA) task in the application. These two functional units will be confined to specific accelerator implementations that will proceed in the design flow until the System implementation phase, where they will be evaluated and compared. To distinguish the HW from the SW counterparts we will be referring these tasks as the HW *Encryptor* and HW *Encryptor* SA.

2.7.1 Hardware specification

When we look at the source code of the *Encryptor* task in Figure 2.7, we realize that generically, the thread is polling for data using the DDS subscription and, at arrival of such data, it will be encrypted and published back to the DDS topic. As soon as the subscription expires, the thread will break the loop and terminate the execution. To map this outline into the Accelerator model, the best use of resources demands that we use the Accelerator data channel to exchange data with the DDS subsystem. This exchange is implemented by a combination of service request, that use the HW system calls in the accelerator and the Linux kernel system calls, to synchronize the *HW-Task* with the application functionality. To minimize the resulting computation overhead on the Host side, we maximize the transfer length involved, thus lowering the number of requests per-file and consequently lowering the synchronization events at the Linux kernel.

The first design decision will be to properly handle data in blocks of 1024 bytes that will be stored in the Accelerator's Local-RAM. To extract individual fragments of 128-bit (16 bytes) of plain data, we will specify the first loop of the *HW-Task* and name it *encrypt* loop. We will call this the inner-loop. Numerous block transfer requests may be necessary until the file is exhausted, as so, we will use a second loop and call it *exchange* loop. Since the file contents might not conform with the block alignment, at every request iteration we will check how many bytes were effectively transferred, and proceed in the *exchange* loop with a new target value. For now, we decide to use a target length counter, and to comply with the task final message informing the processing results, we will include another counter here referred as total counter.

If we elaborate the set of identified requirements into a block diagram, we obtain the HW *Encryptor* datapath. Such diagram can be seen in Figure 2.19 and includes the connection with the Control unit.

From that figure it can be seen that the input of the data-path is the return data from the HW system call and that similarly, the output data will be forward via HW system call. The two blue functional blocks represent the synchronous registry for local storage and we add a comparison unit to signal when the target and count contents match. The light red blocks represent the combinational units that are kept steady value using the synchronous registry. The white labelled blocks are mere representation of data that exists in those registers.

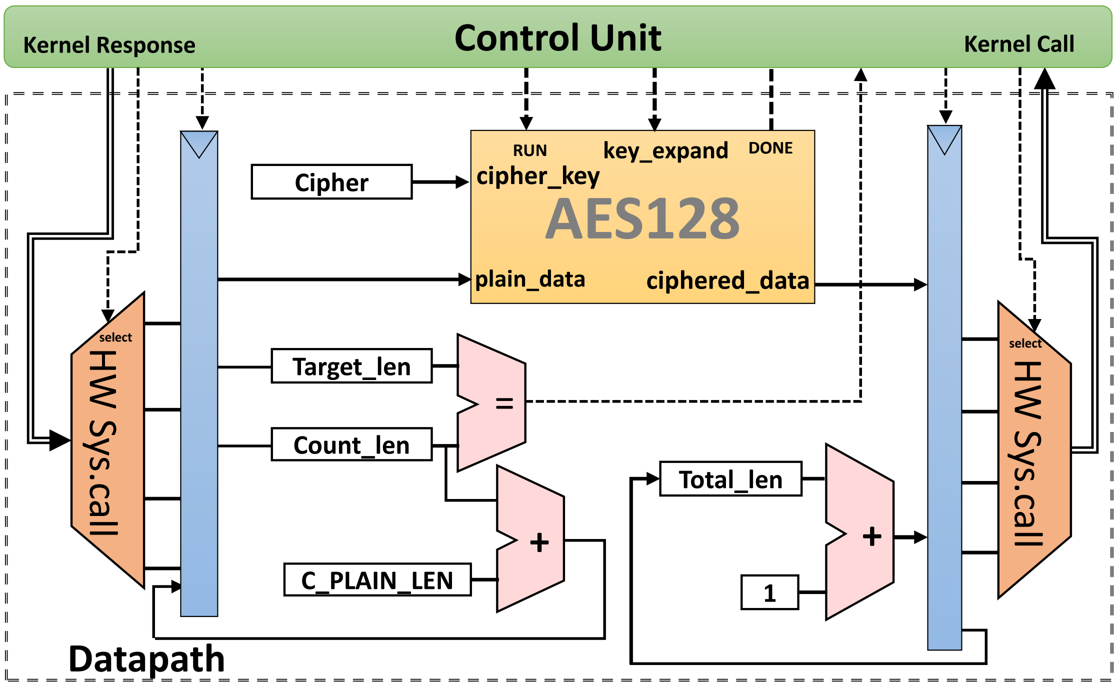


Figure 2.19: HW Encryptor Task - Simplified Datapath

The AES-128 functionality can ultimately be accomplished by integrating any existing functional unit and to serve the purpose of simplicity we will resume the AES description to a top-level functional description. The conceptual implementation was extracted from the diagram depicted in Figure 2.3, and the data inputs and output can be observed here matching the signals in this top-level. The control signals of the AES-128 unit are: the *key_expand*, to expand the cipher in ten additional keys; and the *run* signal that triggers the unit execution. A *done* signal will inform control that the ciphered data is ready to be collected at the output. The AES design follows a twelve-clock pipeline design strategy but to simplify the task specification, this continuous mode of operation will not be explored. Details about HW description of the AES-128 can be consulted in the Appendix C.

The Control Unit for the HW *Encryptor* can be implemented using an FSM-based design, integrating eleven states that sequence the necessary operations, in the true parallel nature of the HW implementation. Such design can be consulted in Figure 2.20.

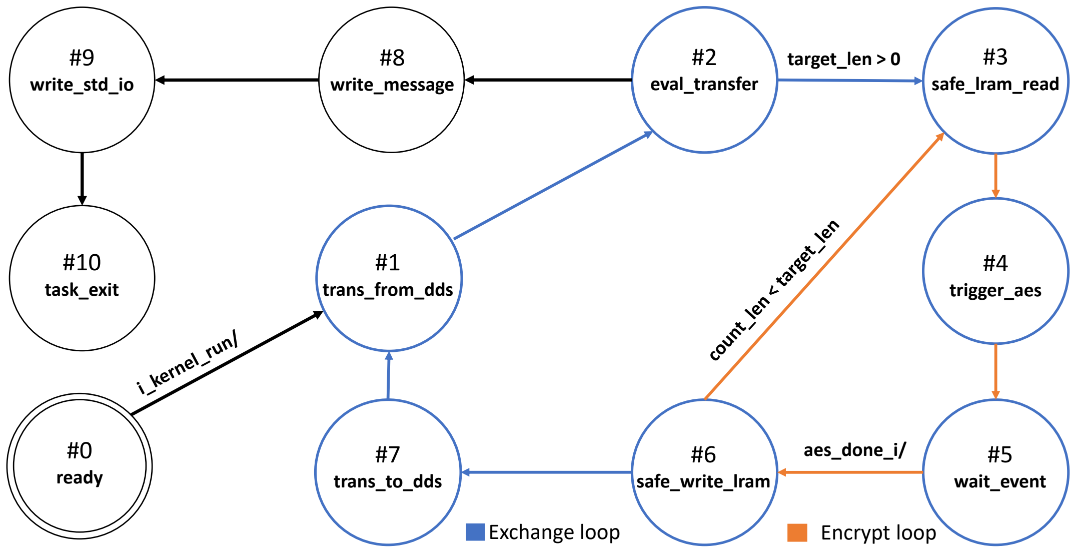


Figure 2.20: HW Encryptor Task - Simplified Control unit FSM

The state zero is a mandatory state for synchronizing the *HW-Task* with the application and the Accelerator enters a sleep state when steady in state zero for more time that a parametrizable time constant. As soon as Control receives the *run* signal from the *HW-Kernel*, the task will issue the first request, demanding a transfer of 1024 bytes from the DDS subscription. The second state will evaluate the transfer result into a new target value and proceed to the *encrypt* loop. Any processing error would result in a negative or null target value that redirects the next state to the write message, thus concluding prematurely.

The target length is accomplished with the state three to state six sequenced loop while the Control Unit is using the user package to: synchronously read blocks of 16-bytes (128-bit) of plain data, here referenced as parametrizable constant C_PLAIN_LEN, and after being submitted to the AES encryption; write the resulting ciphered data to the same address of the Local-RAM. One must say that the fragmenting and regrouping performed by the *FileReader* and the *Uploader* tasks are by now redundant.

In the state four we trigger the AES execution and move to state five where Control issues a wait event system call. Similarly, the clock enable will be cleared and the designer needs to tackle this condition since it needs to keep the AES unit working. The Control unit will reach the seventh state with the counter value matching the target from the *encrypt* loop iteration, and it will conclude by issuing a request to transfer the block of 1024 bytes of ciphered data to the DDS topic, thus restarting the *exchange* loop. The state nine is achieved after failing the target value evaluation in s2, and preparing the message in the standard output format at the Local-RAM on state s8. The Control will issue a request to forward this message and concludes in the state ten, issuing a *task_exit* call from the user package. Beyond this call, the Control unit will not be available until a software reset is issued using *local-kernel* file in the HAL-ASOS file system.

2.7.2 Emulating Hardware Accelerators

When modelling the behavior of the *Encryptor* using C/C++ languages and applying the Emulator model provided by the framework, one must take into account that the sequential statements of the C language will be implemented concurrently if modelling the hardware using HDL. With this idea in mind, care must be taken to avoid the use of temporary data that results from the current flow of execution. For the Emulator model, one clock cycle is implemented by the *run_iteration* call, and one execution of this function will produce results that can only be used in the next cycle.

To develop an emulated *Encryptor* task we start by defining the configuration structure where we specify the accelerator parameterizable resources, namely: the size in number of words of the Local-RAM; the space in number of words in the control FIFOs; the number of parallel words exchanged between *HW-Kernel*, H*W-Task* and *HW-FIFOs*; the number of user-defined interrupts; and the accelerator name or task tag. At the software-side of the application, the *Task* class needs to be reconfigured by the *HwTask* qualifier combined with the Emulator profile, and one accelerator entity needs to be instanced. Figure 2.21 shows the necessary changes to the application when using the Emulator model. The emulated accelerator is instanced in line 208 and uses the configuration as template qualifier. We extend the T1 template pack with the desired profile (line 213) and add one Virtual File System(VFS) instance with the purpose of listing the available accelerators when executing the application (lines 209-210).

The specialized *run_interarion* member is presented in Figure 2.22 and is identified by the compiler using the *HwEncrypter* configuration as template qualifier. The majority of local variables need to be static to preserve data between successive calls that resemble the HW clock cycles. To abstract design from the AES-128 implementation details, we have included this functionality in C++ class (line 43). The FSM implementation uses a switch case block to re-evaluate the *TaskState* member at the beginning of each cycle. Every case entry will conclude by redefining the *TaskStateNext* value and the task iteration concludes with the assignment of the *TaskState* with this new value (not represented).

In state s1 the Control transfers 1024 bytes to the Local-RAM address '0h' and stores the return value in the target length variable. In state s2 this new target determines the next state and, in state s3 the Control

```
  AcceleratorConfig_t HwEncrypter = {"HwEncrypter0", //TaskTag
                      32, //DataFifo Len
                      8, //Message Queue Len
                      1, //DataIn Number of simultaneous words
                      1 //DataOut Number of simultaneous words
  };
...
void hal_asos_demo::test_aes128_file_hw_thread_cypher_3_emulator(void) {
  using namespace hal_asos;

  emulator::AcceleratorModel<hal_asos::emulator::HwEncrypter> A0;
  emulator::VFS& file_system = emulator::VFS::instance();
  file_system.ls();

  Task<SwTask, TFRead> T0;
  Task<HwTask, THwEncrypter, profile<proxy::Emulator>>  T1;
  Task<SwTask, TUpload> T2;

  T0.start();
  T1.start();
  T2.start();
  T0.join();
  T1.join();
  T2.join();
}
```

Figure 2.21: Machine 1: emulated HW *Encryptor* software changes

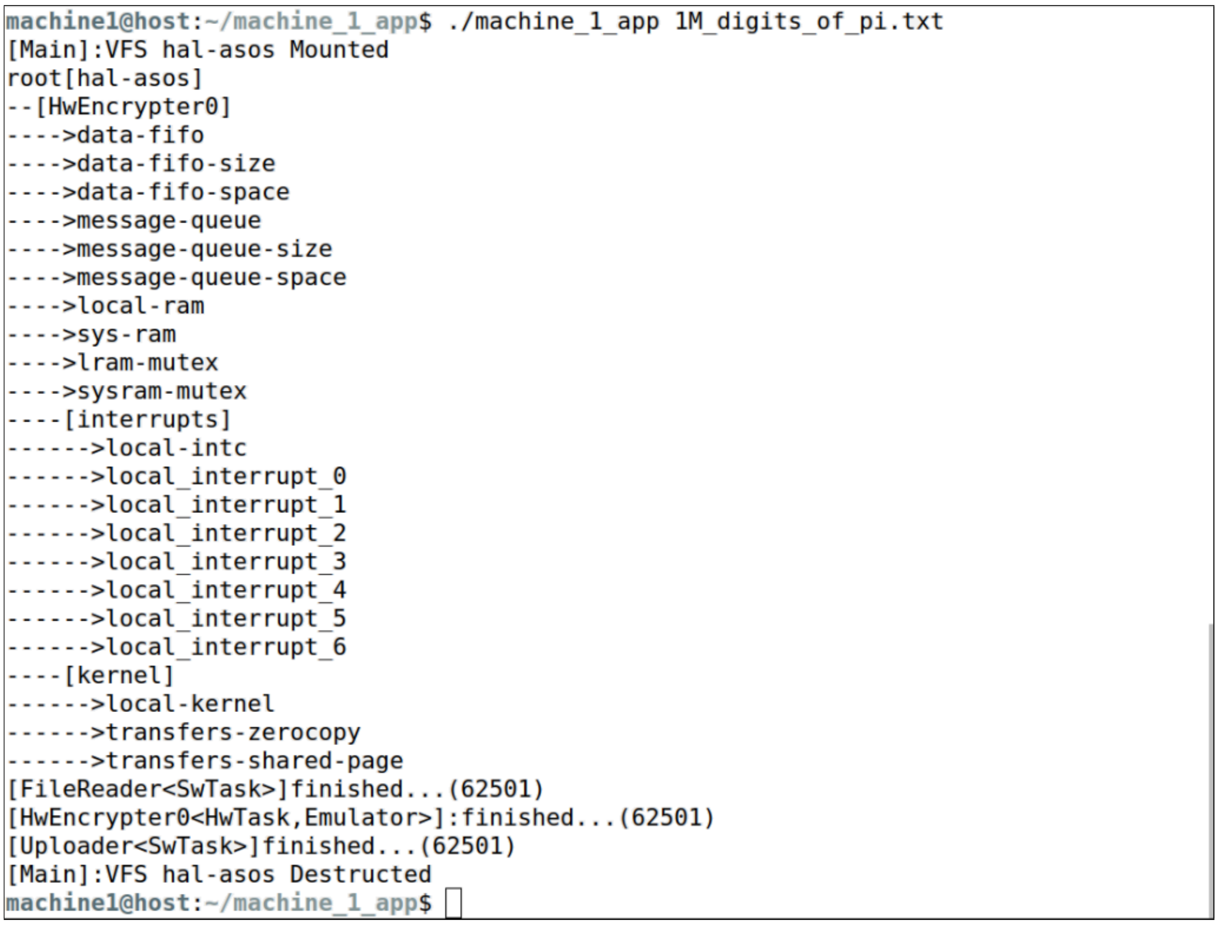
begins the *encrypt* loop implementation. In state s4 the control triggers the AES execution and in s5 the task issues a wait event system call, that when concluded will allow the state to copy the ciphered data to the current block variable. This copy operation represents a synchronous register assignment and as so, such usage should only be considered in the next cycle. The state s6 will conclude the *encrypt* loop by storing the current block to the LRAM, updating counters and re-evaluating loop continuity. In s7 the control closes the *exchange* loop iteration and a null or negative target length will redirect the next state to the exit path. At s8 the message is transferred to the LRAM at the address '80h' and at s90 the control issues a request to write the standard output descriptor (*stdout*). At s99 the Control enters the task exit call that releases the Native thread from the main loop and the emulator concludes the operation.

Experimental log for this emulator stage can be seen in Figure 2.23. The *file_system* object prints the message to inform the user of its service and the *ls* member lists the existing accelerator(s). The remaining four lines are the task conclusion messages, and we can observe the Emulator profile and the *HwTask* qualifiers in use, at the *HwEncrytor0* task log message.

In this design stage the *Encryptor* task was mapped into the Accelerator model using the emulator *namespace*. After dealing with some model-related considerations, we have decided to increase the data exchange between Task's software resources and the Accelerator model, and in doing so, the fragmented and regrouping of the adjacent tasks became redundant. This consideration raised the need to refactor the SW tasks *FileReader* and the *Uploader* but no changes were performed. Instead, we wait for the

```cpp
template <>
void HwTaskModel<HwEncrypter>::task_run_iteration(void)  {
  static int  count_len = 0, target_len, index=0, total_len = 0;
  static block_state_t curr_block;
  static char message_file[] = "finished...(%d)\\n";
  static uint16_t text_len = sizeof(message_file) - 1, mlen = text_len + 4;
  static Ip_AES128_cypher Encrypter;
  switch (this->TaskState){
  case s0:
    if (this->pLKernel->isControlRun()) {
      Encrypter.p_UserKey = &key;
      Encrypter.key_set();
      total_len = 0;
      this->TaskStateNext = s1;}  break;
  case s1:
    transfer_data_from_dds_subscrition(0,1024);
    target_len = cast_return_to_transfer_len();
    count_len = 0;
    index = 0;
    TaskStateNext = s2;  break;
  case s2:
    TaskStateNext = s8;
    if(target_len > 0)
      TaskStateNext = s3;  break;
  case s3:
    safe_read_lram_word32((int*)& curr_block, 4, index);
    count_len = count_len + HLEN;
    TaskStateNext = s4;  break;
  case s4:
    Encrypter.p_plain = (block_state_t*)&curr_block;
    Encrypter.trigger_aes();
    TaskStateNext = s5;  break;
  case s5:
    this->pLKernel->wait_event(Encrypter.Done);
    TaskStateNext = s6;  break;
  case s6:
    std::copy_n((char*)Encrypter.p_cyphered, HLEN, (char*)curr_block);
    safe_write_lram_word32((int*)& curr_block, 4, index);
    index = index + 4;
    total_len = total_len + 1;
    TaskStateNext = s7;
    if(count_len < target_len)
      TaskStateNext = s3;
    break;
  case s7:
    transfer_data_to_dds_topic(0,count_len);
    TaskStateNext = s1;  break;
  case s8:
    safe_write_lram(message_file, mlen, &total_len,4,128);
    count_len = 0;
    TaskStateNext = s90; break;
  case s90:
    write_stdout(text_len,mlen,128);
    TaskStateNext = s99; break;
  case s99:
    this->task_exit();   break;
  default: break;}}
```

Figure 2.22: Machine 1: Hardware Task emulator



```
machine1@host:~/machine_1_app$ ./machine_1_app 1M_digits_of_pi.txt
[Main]:VFS hal-asos Mounted
root[hal-asos]
--[HwEncrypter0]
---->data-fifo
---->data-fifo-size
---->data-fifo-space
---->message-queue
---->message-queue-size
---->message-queue-space
---->local-ram
---->sys-ram
---->lram-mutex
---->sysram-mutex
----[interrupts]
------>local-intc
------>local_interrupt_0
------>local_interrupt_1
------>local_interrupt_2
------>local_interrupt_3
------>local_interrupt_4
------>local_interrupt_5
------>local_interrupt_6
----[kernel]
------>local-kernel
------>transfers-zerocopy
------>transfers-shared-page
[FileReader<SwTask>]finished...(62501)
[HwEncrypter0<HwTask,Emulator>]:finished...(62501)
[Uploader<SwTask>]finished...(62501)
[Main]:VFS hal-asos Destructed
machine1@host:~/machine_1_app$
```

Figure 2.23: Machine 1 - HW encryptor using emulator

standalone *HW-Task* performance results as we believe that it will translate into similar changes and less computation overhead.

The Machine 1 application was validated using this new configuration and from the functional results, we can accept the specifications and proceed in design, implementing the HW *Encryptor* task in the HAL-ASOS Accelerator model.

2.7.3 Hardware description

To implement the HW *Encryptor* we start with a template *HW-Task* provided by the framework that includes a minimal task implementation with control and datapath units. The Control unit uses an FSM-based design that evaluates a task state register to encode the implemented states. This FSM design is composed of concurrent VHDL processes that target synchronous and asynchronous combinational features. The synchronous processes establish synchronism and the required registers, and allow the task to block and resume while interacting at kernel level. The combinational design establishes the control path that dictates the FSM behavior and through the use of the RTL packages, it will allow the task to interact at the application level.

The provided FSM model implements a Moore's machine-based description, where each individual state determines the set of active control outputs. A closed-loop locks the task next state on the current value, until an input change determines a new next state. The next valid clock pulse will store this value and FSM will produce a new set of state dependent control outputs. A clock pulse is considered valid if it produces a synchronous assignment. A non-valid clock pulse can result from the blocked-task condition that disables functional units' clock as result of kernel control interchange. For this reason, the description of the datapath registers uses asynchronous design to implement the blocking and sleeping functionalities.

To describe the FSM discussed in the specification section we implement the task datapath using a set of independent synchronous logic that includes the AES-128 component and the previously identified counters. The task register descriptions follow the name used in the design stage and extends them with the *'_d'* to denote the input connection, and the *'_q'* to denote the output of the registers. These input and output connections link with the control path of the FSM, and so, the input register connections are updated with the control-path outputs and the stored values are then forwarded back to the control-unit inputs.

Describing hardware using HDL languages is analogous to computer programming, in the sense that we structure an inherent functional behavior, through common language constructs such as *if*, *then*, *else* or *case*, and delegate the details in the functional aspects to specific subprograms. Such description denotes similarities with the emulator implementation depicted in Figure 2.22. The *switch* statement from the C language, translates directly to a *case* statement of the VHDL, the *when* entry in the VHDL can implement The *case* entries from the software implementation to create a similar structure that represents the algorithmic level of the *HW-Task*. Figure 2.24 outlines a simplified VHDL process that translates the emulator source to the FSM control path used in the HW *Encryptor* task. The complete listing can be consulted in the attached Listing C.4.

In that figure, it can be seen the *exchange* and *encrypt* loops implemented in states *s1* to *s7* and *s3* to *s6* descriptions. The request to exchange data with the DDS is implemented by the procedure call in lines 122 and 151, and the resulting transfer is read from, and written to, the Local-RAM in lines 133 and 143. The procedure parameter *p_current_d* in line 133, is sourced to the *p_current_q* register that connects with of the AES-128 component using *plain_data_i* signal. The *p_cypher_i* parameter in the procedure call of line 143, establishes the connection from the AES-128 cyphered output. Details of the AES-128 connections can be consulted in the attached Listing C.5. The *wait_signal_event* procedure in line 143, extends the *done_i* port from the AES-128 to the kernel control and blocks the *HW-Task*. A high value in this port will allow the system to return from the kernel system call and the FSM proceeds to the next state.

```
 94 ---------------------------------------------------------------------------------------------
 95 FSM_CONTROL: process(task_state,....)
 96 ---------------------------------------------------------------------------------------------
 97 begin
    ...
114 hal_asos_link_to_kernel(kernel_response,kernel_call);
115   case task_state is
116   when s0_ready=>
117      if s00_kernel_run = '1' then
118       task_state_next <= s1_transfer_from_dds;
119      end if;
120      total_len_d<=0;
121   when s1_transfer_from_dds=>
122      transfer_data_from_dds(kernel_call,kernel_response,0, C_BLOCK_LEN);
123      target_len_d <= cast_return_to_transfer_len(kernel_response);
124      index_d <= 0;
125      count_len_d<=0;
126      task_state_next <= s2_evaluate_transfer;
127   when s2_evaluate_transfer=>
128      task_state_next <= s8_write_message;
129      if(target_len_q >0) then
130       task_state_next <= s3_read_lram;
131      end if;
132   when s3_read_lram=>
133      safe_read_lram_word32(kernel_call, kernel_response,p_current_d,(C_PLAIN_LEN/4),index_q);
134      count_len_d <= count_len_q + C_PLAIN_LEN;
135      task_state_next <= s4_trigger_aes;
136   when s4_trigger_aes=>
137      trigger_aes_i<= '1';
138      task_state_next<= s5_wait_aes;
139   when s5_wait_aes=>
140      wait_signal_event(kernel_call, kernel_response,aes_done_i,Done_d);
141      task_state_next <= s6_write_lram;
142   when s6_write_lram=>
143      safe_write_lram_word32(kernel_call, kernel_response,p_cypher_i,(C_PLAIN_LEN/4),index_q);
144      index_d <= index_q + (C_PLAIN_LEN/4);
145      total_len_d <= total_len_q + 1;
146      task_state_next <= s7_transfer_to_dds;
147      if(count_len_q < target_len_q) then
148       task_state_next <= s3_read_lram;
149      end if;
150   when s7_transfer_to_dds=>
151      transfer_data_to_dds(kernel_call, kernel_response,0, count_len_q);
152      task_state_next<= s1_transfer_from_dds;
153   when s8_write_message=>
154      safe_write_lram(kernel_call, kernel_response, fmessage,
                          std_logic_vector(to_unsigned(total_len_q,32)),0);
155      task_state_next <= s90_print_stdio;
156   when s90_print_stdio=>
157      write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
158      task_state_next <= s99_exit;
159   when s99_exit=>
160      task_exit(kernel_call, kernel_response);
161      task_state_next <=s99_exit;
    ...
165 end process FSM_CONTROL;
166 ---------------------------------------------------------------------------------------------
```

Figure 2.24: *Encryptor* - simplified control path.

At state *s6*, when the counter register matches the target length register value, the task breaks the *encrypt* loop and proceeds in to state *s7*, where it transfers the encrypted data to the DDS subsystem and restarts a new exchange iteration. Once back at state *s2*, a null or negative value in the *target_len* register will break the *exchange* loop, and the FSM proceeds to the state *s8*, where it transfers the results message to the Local-RAM. At state *s90*, the FSM issues a request to write the message in the *stdout*, and in state *s99* the FSM concludes with the *task_exit* procedure, notifying the software class and putting the accelerator core in a dead state. Only a hardware reset or a software demanded initialization can revert the FSM back to state *s0*, and the *HW-Task* will be ready for a new iteration.

The HW *Encryptor* SA can be considered an incremental design iteration that uses the previous task description. The *encrypt* and *exchange* loops will remain in use but the task will exchange data with the input file, and to the network socket, as opposed to the use of the DDS subsystem. To provide the interaction with Linux OS services, the framework translates most of the common Linux models to HW descriptors that maintain the status of each virtual representation in the application scope. Such descriptors are *ifile_q* and *tsocket_q*, which are implemented in the *HW-Task* synchronous process descriptions in its datapath, and they are used by the control path while implementing user-related procedures. Figure 2.25 outlines the most relevant changes needed to describe the combinational procedure for the HW *Encryptor* SA control.

In lines 124 and 130 the control will implement user package procedures to query for the required objects and update the local HW descriptors. The file length is read in line 137 and will be used as header for the encrypted file in line 138. The network socket will open a connection at line 148, that uses the predetermined connection settings, and at the *exchange* loop completion both objects will be closed (lines 189 and 192). The FSM concludes similarly by writing the results message to the Linux *stdout* and issuing a *task_exit* procedure call that will put the accelerator in sleep state. The complete FSM description including the combinational and synchronous procedure can be consulted in the attached Listing C.6, C.7 and C.8. For completeness, the datapath description can also be found in the attached Listing C.9.

When we analyze the HW *Encryptor* SA descriptions listed in this section, and continuing throughout the related attached figures, it can be noticed that the description style used follows a more hardware-oriented, or structural description, when compared to the behavioral style used in the emulated *Encryptor* task description. This style does not demonstrate the same level of similarity, but it reflects the incremental description updates that aim to improve the logic extraction feature used by the implementation tools such

```
  hal_asos_link_to_kernel(kernel_response,kernel_call);
  case task_state is
  ...
  when s1_query_file=>
    pooled_fstream_query(kernel_call,kernel_response,ifile_q, ifile_d);
    task_state_next <= s2_query_socket;
  when s2_query_socket=>
    pooled_socket_query(kernel_call,kernel_response,tsocket_q,tsocket_d);
    task_state_next<= s3_open_file;
  when s3_open_file=>
    pooled_fstream_open(kernel_call,kernel_response,ifile_q,ifile_d);
    status_ret_d <= cast_return_to_transaction_ret(kernel_response);
    task_state_next <= s4_evaluate_file;
    if(status_ret_q < 0)then
      task_state_next <= s17_write_string_lram;
    end if;
  when s4_evaluate_file=>
    pooled_fstream_read_len(kernel_call,kernel_response,ifile_q,file_len_d);
    p_current_d(0) <= std_logic_vector(to_unsigned(file_len_q,32));
    task_state_next <= s16_close_file;
    if(file_len_d >0) then
      task_state_next <= s5_set_word_len;
    end if;
  when s5_set_word_len=>
    safe_write_lram_word32(kernel_call, kernel_response, p_current_q,1,0);
    inc_index <= '1';
    task_state_next<= s6_open_socket;
  when s6_open_socket=>
    pooled_socket_open(kernel_call,kernel_response,tsocket_q,tsocket_d);
    status_ret_d <= cast_return_to_transaction_ret(kernel_response);
    task_state_next <= s7_read_file;
    if(status_ret_q < 0)then
      task_state_next <= s15_close_socket;
    end if
...
  when s14_write_socket=>
    clr_index<= '1';
    pooled_socket_write_word32(kernel_call, kernel_response, tsocket_q,index_q,0);
    task_state_next <= s7_read_file;
  when s15_close_socket=>
    pooled_socket_close(kernel_call, kernel_response, tsocket_q,tsocket_d);
    task_state_next <= s16_close_file;
  when s16_close_file=>
    pooled_fstream_close(kernel_call, kernel_response, ifile_q,ifile_d);
    task_state_next<= s17_write_string_lram;
  ...
end process FSM_DPATH;
-------------------------------------------------------------------------------------
```

Figure 2.25: *Encryptor* SA - simplified control path.

as Vivado. Further details that concern the Accelerator model and some of the package provided features used in this section will be best discussed in Chapter 3 and Chapter 4.

2.7.4 Co-Simulation model

From this point in the design flow, the two *HW-Task* descriptions that where implemented are ready for functional validation. An efficient and proper design validation should consider the application as a unified solution. Since this application will be running on the host development system, and the elaborated designs will exist solely in the simulation tool, we apply the co-simulation model that links the application functionality with the simulation tool. The outcome of this model is a unified simulation environment that simplifies the need for complex test benches that closely emulate the application by applying the real application stimulus to the accelerator model.

At the software side, the framework will use the *CoSimulation* profile that specializes the task's Proxy member and implements the conceptual resource-based set of operations, by using two network channels, and establishing connection with each individual accelerator. Figure 2.26 helps to illustrate the Co-Simulation model in the HAL-ASOS framework. A primary transfer channel will enqueue all of the *Task* class requests and the secondary channel will provide the interrupt related synchronism. The pending operations from the primary channel will be forwarded to the secondary channel for synchronism, before re-attempting the primary channel again.

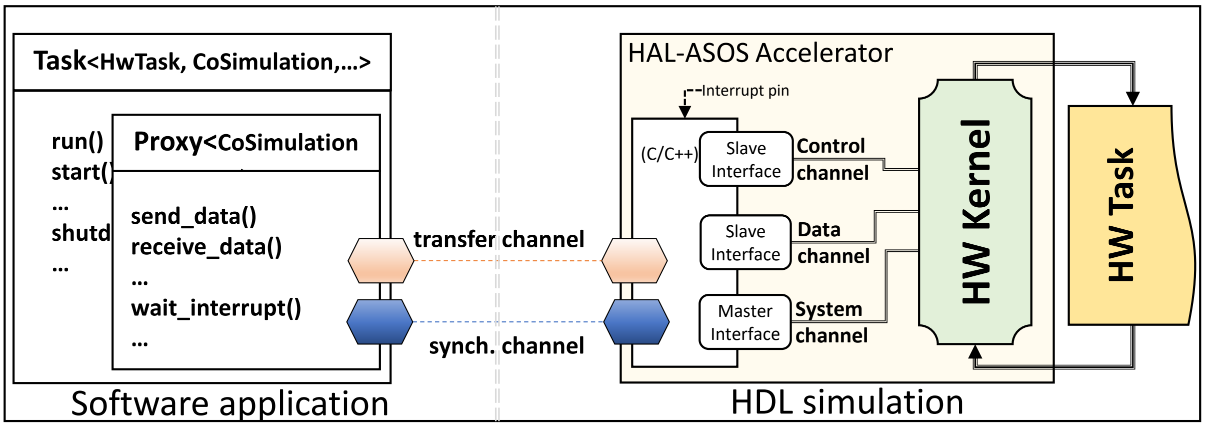


Figure 2.26: HAL-ASOS simplified Co-Simulation model

The Accelerator model interfaces the communication channels using similar network features that provide a bus-abstraction to the original design. This feature is implemented using a mixed topology that combines the RTL description with: (1) VHDL Foreign Language Interface (FLI), establishing a programming interface that provides means to access data in the VHDL elaborated and simulated models; (2) or SystemVerilog Direct Programming Interface (DPI), interfacing the hardware description to foreign languages, namely C/C++ and System-C, by directly calling functions implemented in the foreign language. The choice is simulation-tool dependent since FLI support is, up until now, only available in the ModelSim tool. In either of the Model distinct implementations, the foreign language used was C/C++. Running a test-bench on the accelerator will allow the network channels to listen and accept connections. The software application can then start, and after a quick handshake, it will be running simultaneously at both sides of the implemented design.

2.7.5 Encryptor co-simulation

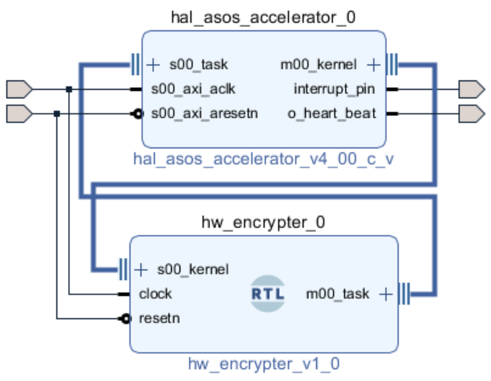
In this section, the Machine 1 application will be executed on the host's development environment using the co-simulation model. The co-simulation model involves the simulation of the software application, containing the *Task* class reconfigured for *HwTask* and using the *CoSimulation* profile, and the developed HW *Encryptor*, interacting with the accelerator model by using an RTL simulation tool. Figure 2.27 shows the Machine 1 application, containing the necessary modifications to perform the functional validation in the co-simulation environment. When we observe the lines in that figure, we can see that the *Task* T1 declaration is using the *THwEncrypter* configuration, the *HwTask* qualifier and extends the template package with the *CoSimulation* profile.

```
void hal_asos_demo::test_aes128_file_hw_cosim_thread_cypher_3(void) {
  using namespace hal_asos;

  Task<SwTask, TFRead> T0;
  Task<HwTask, THwEncrypter,profile<proxy::CoSimulation>>  T1;
  Task<SwTask, TUpload> T2;

  T0.start();
  T1.start();
  T2.start();
  T0.join();
  T1.join();
  T2.join();
}
```

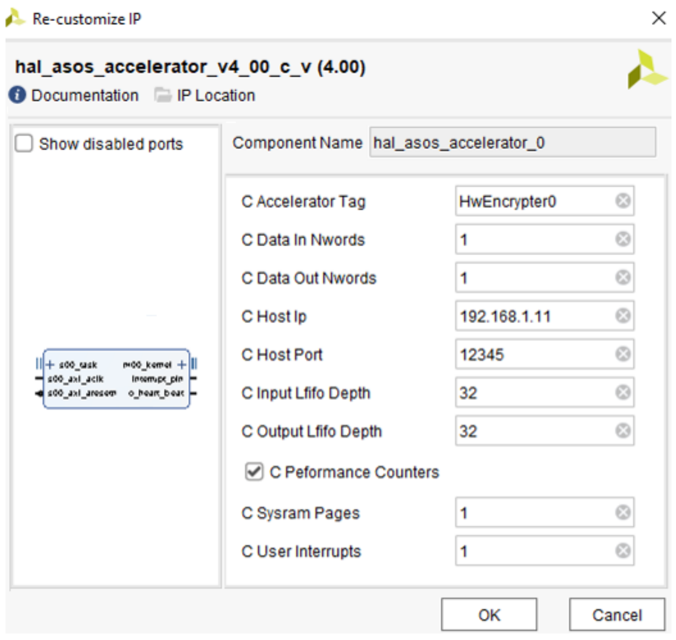
Figure 2.27: Machine 1 - software changes for Co-Simulation using the HW *Encryptor* task.

On the RTL side, we integrate the *HW-Task* in the application using the accelerator component, that can be identified by the suffixes '*_c*' for co-simulation and '*_v*' for bus abstraction based on SystemVerilog. In Figure 2.28a it can be seen the block design used for the co-simulation of the HW *Encryptor* task using a Vivado design project. The upper block represents the accelerator model, where it receives stimuli at clock and reset inputs, and responds with interrupt and heart-bit signals. These outputs are intended for their graphical representations in the tool's wave window, since the interrupt signal is sensed internally by the bus abstraction model, and the heart-bit signal is an interface-based signal that denotes the accelerator operability.

The lower block represents the *HW-Task* that was described in the previous section. This block design is then instantiated as a component, and submitted to a test-bench using an RTL file provided with the accelerator. The source listing of this test-bench can be consulted in the attached Listing C.10. The framework provides distinct implementations of the accelerator model which include Extensible Markup



(a) HW Encryptor accelerator connections



hal_asos_accelerator_v4_00_c_v (4.00)

Component Name: hal_asos_accelerator_0

C Accelerator Tag	HwEncrypter0
C Data In Nwords	1
C Data Out Nwords	1
C Host Ip	192.168.1.11
C Host Port	12345
C Input Lfifo Depth	32
C Output Lfifo Depth	32
C Peformance Counters	✓
C Sysram Pages	1
C User Interrupts	1

(b) Accelerator v4.00.c.v parameters

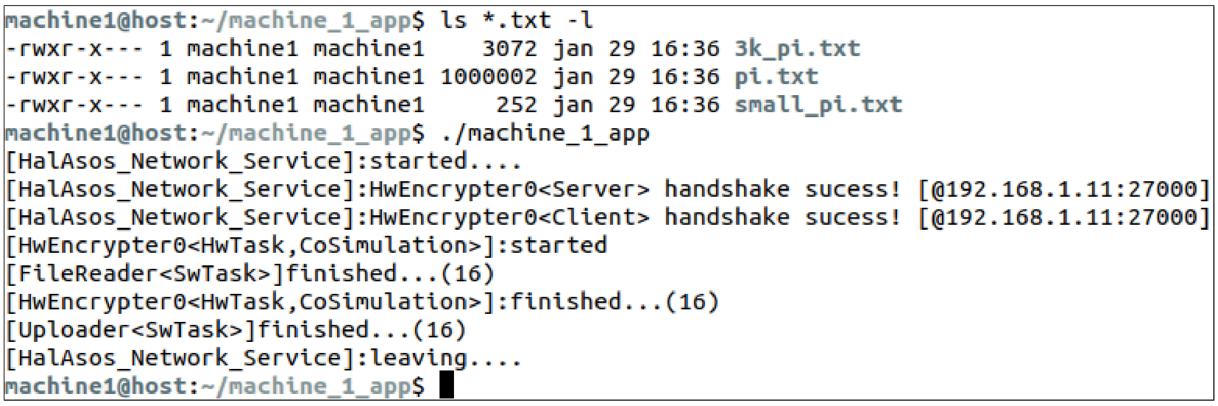
Figure 2.28: Co-Simulation - HW Encryptor Accelerator settings

Language (XML) descriptions in the IP-XACT format, to simplify the block design step by promoting the assisted connection with rule verification features. It also abstracts each group of the top-level logical signals to single interface connections.

In the same image, Figure 2.28b, it is also possible to see the graphical interface used to configure the accelerator. This accelerator uses the host network service and so, it needs parameters that configure the network location where it will register its existence. The IP address and port number are used to connect with the Machine 1 application, and the identification is mostly dependent of the name *tag* that matches the configuration member used in the class *Task*.

When both, the application binary and the elaborated design are ready, the application and the simulation can be started without any particular order. In Figure 2.29 it can be seen message log that took place on the software side of the application. For this purpose, a smaller file containing 252 bytes (file small_pi.txt) was used, thus reducing the simulation and exposing the task algorithm to less cycles than the specified block size to complete the encryption, and to a file length that is not aligned with the accelerator word size.

The HAL-ASOS network management service starts when the first registration attempt is received and issues a message in the application log. Upon receiving the accelerator registration, it will assign new port number used for handshake activities. The *Task* class will also register its particular interest about an accelerator that fits the provided parameters, and when found, it will receive back its IP address and the new port number for the handshake activities that starts right away. The remaining log messages demonstrate the application expected behavior and include the processing results sent by the HwEncrypter0 accelerator, which can be identified by the prefix from the template qualifiers *HwTask* and *CoSimulation*. Each individual task message shows a total of sixteen processed fragments and the application concludes as expected.

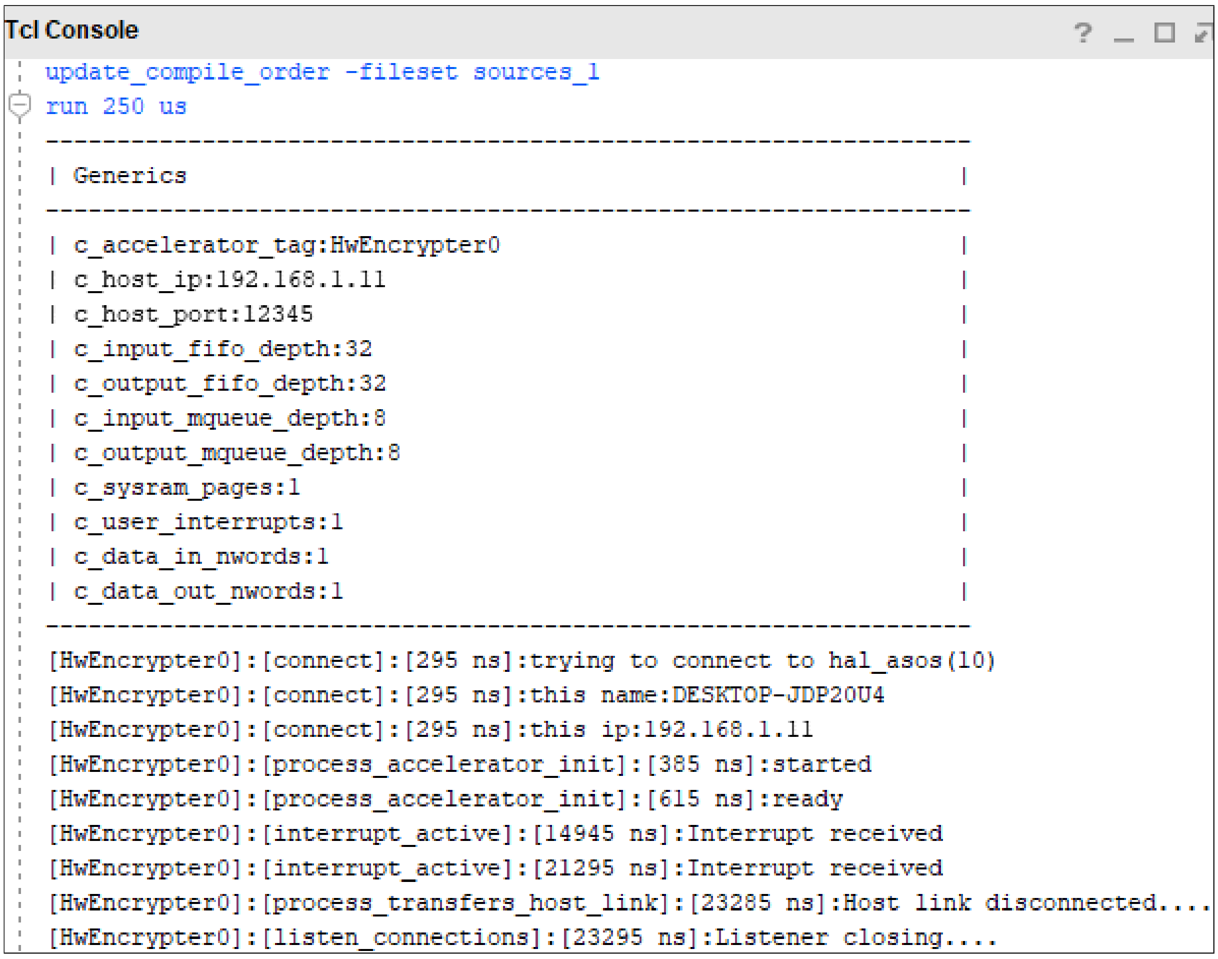


```
machine1@host:~/machine_1_app$ ls *.txt -l
-rwxr-x--- 1 machine1 machine1    3072 jan 29 16:36 3k_pi.txt
-rwxr-x--- 1 machine1 machine1 1000002 jan 29 16:36 pi.txt
-rwxr-x--- 1 machine1 machine1     252 jan 29 16:36 small_pi.txt
machine1@host:~/machine_1_app$ ./machine_1_app
[HalAsos_Network_Service]:started....
[HalAsos_Network_Service]:HwEncrypter0<Server> handshake sucess! [@192.168.1.11:27000]
[HalAsos_Network_Service]:HwEncrypter0<Client> handshake sucess! [@192.168.1.11:27000]
[HwEncrypter0<HwTask,CoSimulation>]:started
[FileReader<SwTask>]finished...(16)
[HwEncrypter0<HwTask,CoSimulation>]:finished...(16)
[Uploader<SwTask>]finished...(16)
[HalAsos_Network_Service]:leaving....
machine1@host:~/machine_1_app$ 
```

Figure 2.29: Co-Simulation - HW Encryptor application output

Back on the RTL side, the simulation was started and the accelerator waits for the task handshake. Figure 2.30 shows the log that results from this simulation in the Vivado's TCL console. It can be seen the list of configuration parameters in use, here referred as *Generics*, and three log messages that encompass the accelerator registration. All the displayed messages lines contain prefixes based on: the name of the accelerator that sends the message; followed by the name of the internal operation that is being executed; and the current simulation time in nanoseconds. The handshake is triggered by the internal *connect* function at 295 nanoseconds of simulation time. After a successful handshake, the *Proxy* member in the class *Task* initializes the accelerator using local provided resources that include a system memory region. The operation started at 385 nanoseconds and completed at 615 nanoseconds. The next two message lines, report interrupts received from the *HW-Kernel* and the simulation concludes with the final two lines that result from the software application disconnection.

From the performed simulation, a wave diagram is also displayed where it's possible to examine the contents in the registers of the elaborated design. Figure 2.31 combines a view of the simulation at the top of the image, and an enlargement aligned with some FSM states at the bottom. At the top of the figure, it can be seen the *exchange* loop iteration, encompassed between the states s1 and s8, and sixteen *encrypt* loop iterations, delimited by the markers at 5.235 microseconds and the at 14.845 microseconds.



```
update_compile_order -fileset sources_1
run 250 us
----------------------------------------------------------------
| Generics                                                     |
----------------------------------------------------------------
| c_accelerator_tag:HwEncrypter0                               |
| c_host_ip:192.168.1.11                                       |
| c_host_port:12345                                            |
| c_input_fifo_depth:32                                        |
| c_output_fifo_depth:32                                       |
| c_input_mqueue_depth:8                                       |
| c_output_mqueue_depth:8                                      |
| c_sysram_pages:1                                             |
| c_user_interrupts:1                                          |
| c_data_in_nwords:1                                           |
| c_data_out_nwords:1                                          |
----------------------------------------------------------------
[HwEncrypter0]:[connect]:[295 ns]:trying to connect to hal_asos(10)
[HwEncrypter0]:[connect]:[295 ns]:this name:DESKTOP-JDP20U4
[HwEncrypter0]:[connect]:[295 ns]:this ip:192.168.1.11
[HwEncrypter0]:[process_accelerator_init]:[385 ns]:started
[HwEncrypter0]:[process_accelerator_init]:[615 ns]:ready
[HwEncrypter0]:[interrupt_active]:[14945 ns]:Interrupt received
[HwEncrypter0]:[interrupt_active]:[21295 ns]:Interrupt received
[HwEncrypter0]:[process_transfers_host_link]:[23285 ns]:Host link disconnected....
[HwEncrypter0]:[listen_connections]:[23295 ns]:Listener closing....
```

Figure 2.30: Co-Simulation - HW Encryptor handshake messages

Through the lower part of the figure, we can observe the behavior across the *encrypt* loop, where reading a fragment from Local-RAM takes twelve clock cycles, marker at 14,245.00 microseconds, in a two clock per word rate, that includes the four words read, and two more words for locking and unlocking the *LMutex*. The words used in the HW *mutexes* are equal and composed from a combination between configuration parameters and hardware signatures from the used *mutex* channel. When we observe the *block_task* signal, we realize that it remains active for the majority of the simulation time, which demonstrates an intensive use of functionalities under the kernel control.

The behavior in the AES-128 top-level signals can also be examined, namely: *run* and *done* control signals, and *plain_data* and *ciphered_data* ports. At the 14,465.00 microseconds marker, the FSM triggers the AES *run* signal and proceeds to the *wait_aes* state, where it blocks in the HW system call. The AES uses a twelve-stage pipeline to encrypt the input *plain_data* and then releases the *done* signal. The kernel event subsystem will acknowledge a steady high value in the *done* signal, and the task returns from the system call in an additional four clocks. The AES connection differentiates blocking by using the kernel *sleep* control signal, and keep its clock active while the task is blocked in the above system calls.

The same twelve clocks are used to write-back the cyphered data to the Local-RAM and the *encrypt* loop restarts, or moves to state *s7* and attempts a new *exchange* loop iteration. The number of clocks used in the states *s1* and *s7* are mostly dependent of the network activity and does not reflect the four clocks used in the HW system call implementation. Two clocks are spent writing a 64-bit message in the kernel message-queue, and upon receiving, two more clocks are spent reading the response. Similar to other transactions in the kernel set of functional units, one clock pulse is used during data exchanging, and another clock pulse is used for reading the signals that acknowledge the operation.

The FSM completes the *exchange* loop with the *count_len* matching the *target_len* and the *total_len_q* register indicating sixteen processed fragments, where it proceeds to state *s8*. Once in s8, the control writes the final results in the Local-RAM, and the operation completes in the state *s90* with the request to transfer the message to the *stdout*. At the 22,505.00 microseconds marker, the FSM achieves the last state and the execution concludes. The *task_exit* procedure notifies the *Task* class and puts the *Kernel Core* in a *dead* state. It also suspends all the *HW-Task* activities with both, the *block_task* and *sleep* signals active, until a hardware reset or a new initialization procedure is received.

A functional validation of the Machine 1 was performed, using the HW *Encryptor* description and the accelerator model for co-simulation. At the software side, minimal reconfiguration was performed by selecting the *HwTask* specialized class, that using the *CoSimuation* profile connects with the hardware part of the application for a unified simulation environment. For convenience, a smaller file was used thus producing a reduced number of cycles that ease the representation in the wave windows and allow for a comprehensive description about the *HW-Task* behavior. We now proceed with the Machine 1 application using the HW *Encryptor* SA task in the co-simulation environment. For completeness, we will be using the ModelSim tool to simulate the accelerator's RTL behavior.

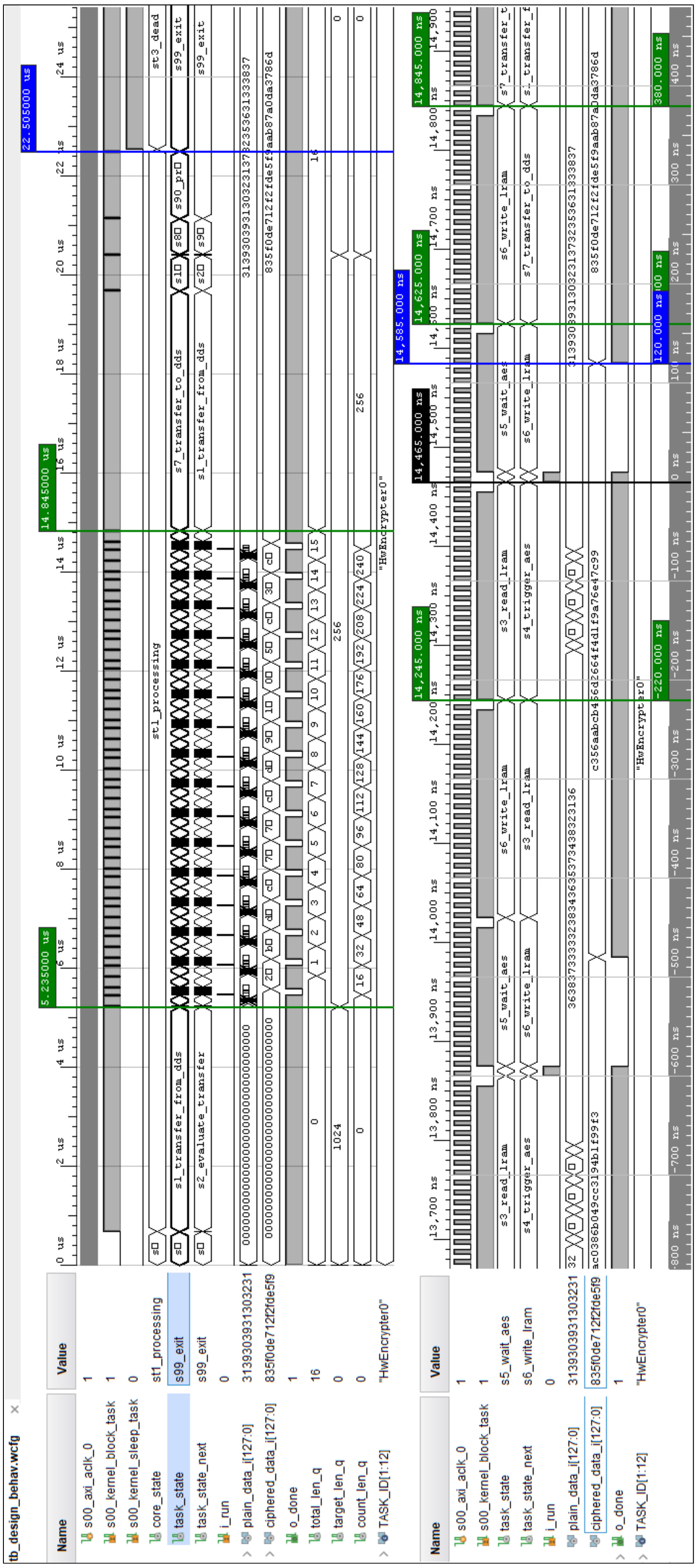


Figure 2.31: Co-Simulation - HW Encryptor wave plot using Vivado simulator

2.7.6 Encryptor SA co-simulation

To address the roles of reading from the input file and writing to the network socket using the standalone *HW-Task*, first the designer needs to create the local references of this objects in the software application. For that it will use a framework software abstraction that encapsulates the implementation details of each specific object in the software class *VirtualObject*.

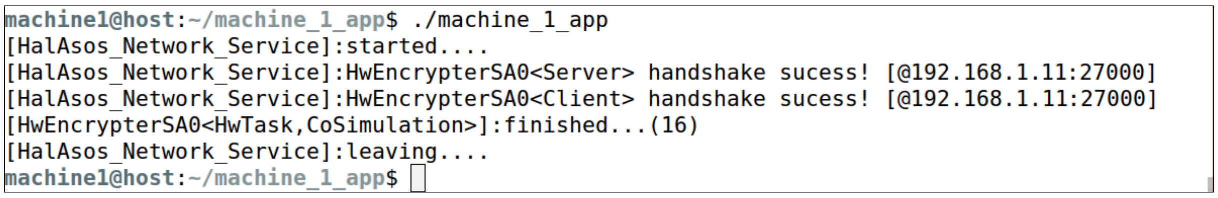
The abstraction reduces the interaction to a message-based communication model, using internal message format that can be sent or received by the accelerator. The template specializations of this class include the most commonly used Linux device models. Figure 2.32 demonstrates the use of this feature in abstracting the input file stream and the network socket for the Machine 1 application.

```
367  void hal_asos_demo::test_aes128_file_hw_cosim_thread_cypher_sa(void) {
368  using namespace hal_asos;
369    hal_asos::networking::CSocket<hal_asos::networking::Client> Soc;
370    CFstream<std::ifstream> Input_file(target_file.c_str());
371
372    Task<HwTask, THwEncrypterSA, profile<proxy::CoSimulation>>  T1;
373
374    Input_file.set_flags(std::ios::in | std::ifstream::binary);
375    Soc.set_ip_address(ip);
376    Soc.set_sock_family(AF_INET);
377    Soc.set_sock_type(SOCK_STREAM);
378    Soc.set_sock_port(PORT_NO);
379
380    T1.submit_to_pool(Input_file);
381    T1.submit_to_pool(Soc);
382
383    T1.start();
384    T1.join();
385  }
```

Figure 2.32: Machine 1 - Co-Simulation Standalone HW Encryptor Task

Such instances are declared in lines 369 and 370 of the same figure, and after some configurations, they are submitted to the task *T1* internal structures in lines 380 and 381. The file and the network socket are kept closed until submission, but alternatively, they could have been opened. The *T1 Task* instance uses the same qualifiers as in previous validation but a new configuration that matches this new accelerator descriptions is used.

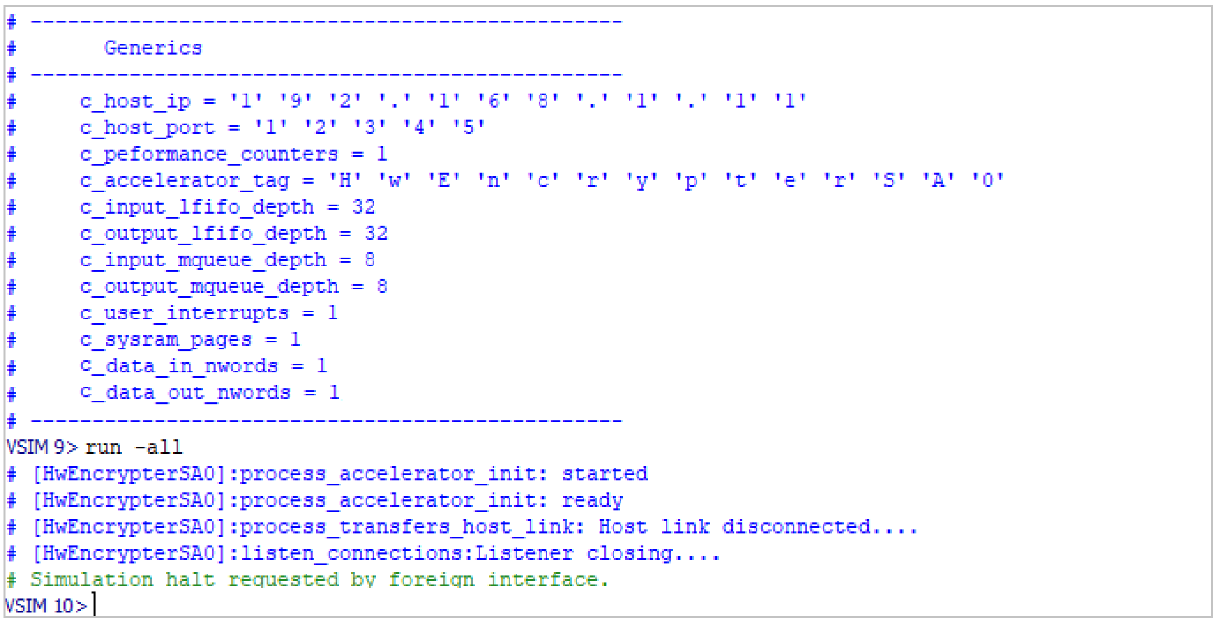
The newly reconfigured application was once more compiled and executed at host environment, and after connecting with the accelerator model, the output results were captured and can be observed in Figure 2.33. Similar output from the HAL-ASOS network manager is reported and it can be noticed only one task result message.



```
machine1@host:~/machine_1_app$ ./machine_1_app
[HalAsos_Network_Service]:started....
[HalAsos_Network_Service]:HwEncrypterSA0<Server> handshake sucess! [@192.168.1.11:27000]
[HalAsos_Network_Service]:HwEncrypterSA0<Client> handshake sucess! [@192.168.1.11:27000]
[HwEncrypterSA0<HwTask,CoSimulation>]:finished...(16)
[HalAsos_Network_Service]:leaving....
machine1@host:~/machine_1_app$ 
```

Figure 2.33: Co-Simulation - HW Encryptor SA application output.

At the ModelSim console, the configured settings were reported and upon receiving connection from the software task, the simulation proceeded in a continuous mode until receiving disconnection. In Figure 2.34 it can be seen the console log from this simulation.



```
# ------------------------------------------------
#       Generics
# ------------------------------------------------
#     c_host_ip = '1' '9' '2' '.' '1' '6' '8' '.' '1' '.' '1' '1'
#     c_host_port = '1' '2' '3' '4' '5'
#     c_peformance_counters = 1
#     c_accelerator_tag = 'H' 'w' 'E' 'n' 'c' 'r' 'y' 'p' 't' 'e' 'r' 'S' 'A' '0'
#     c_input_lfifo_depth = 32
#     c_output_lfifo_depth = 32
#     c_input_mqueue_depth = 8
#     c_output_mqueue_depth = 8
#     c_user_interrupts = 1
#     c_sysram_pages = 1
#     c_data_in_nwords = 1
#     c_data_out_nwords = 1
# ------------------------------------------------
VSIM 9> run -all
# [HwEncrypterSA0]:process_accelerator_init: started
# [HwEncrypterSA0]:process_accelerator_init: ready
# [HwEncrypterSA0]:process_transfers_host_link: Host link disconnected....
# [HwEncrypterSA0]:listen_connections:Listener closing....
# Simulation halt requested by foreign interface.
VSIM 10>
```

Figure 2.34: Machine 1 - HW Encryptor SA handshake messages using ModelSim

A wave plot describing the register contents throughout the performed simulation is depicted in Figure 2.35. In a similar manner, the image is divided in two subplots where we can observe the overall path of the states implemented by the FSM, and a magnified view of the *encrypt* loop. At the bottom of the topmost subplot, we can observe the first marker indicating the open file operation at time 1,940.00 nanoseconds, and consecutively the socket is opened at time 3,150.00 nanoseconds (i.e., *S3_Open_file* and *S6_Open_sock* markers respectively). The hardware file descriptors updated with the current state on the software side and it can be seen that the two virtual objects are kept closed until each specific open state is reached. The complete file encryption starts at the 8,960.00 nanoseconds and finishes at 18,710.00 nanoseconds in a total of sixteen processed fragments, (i.e., *S9_load_plain* and *S13_update_in* markers respectively).

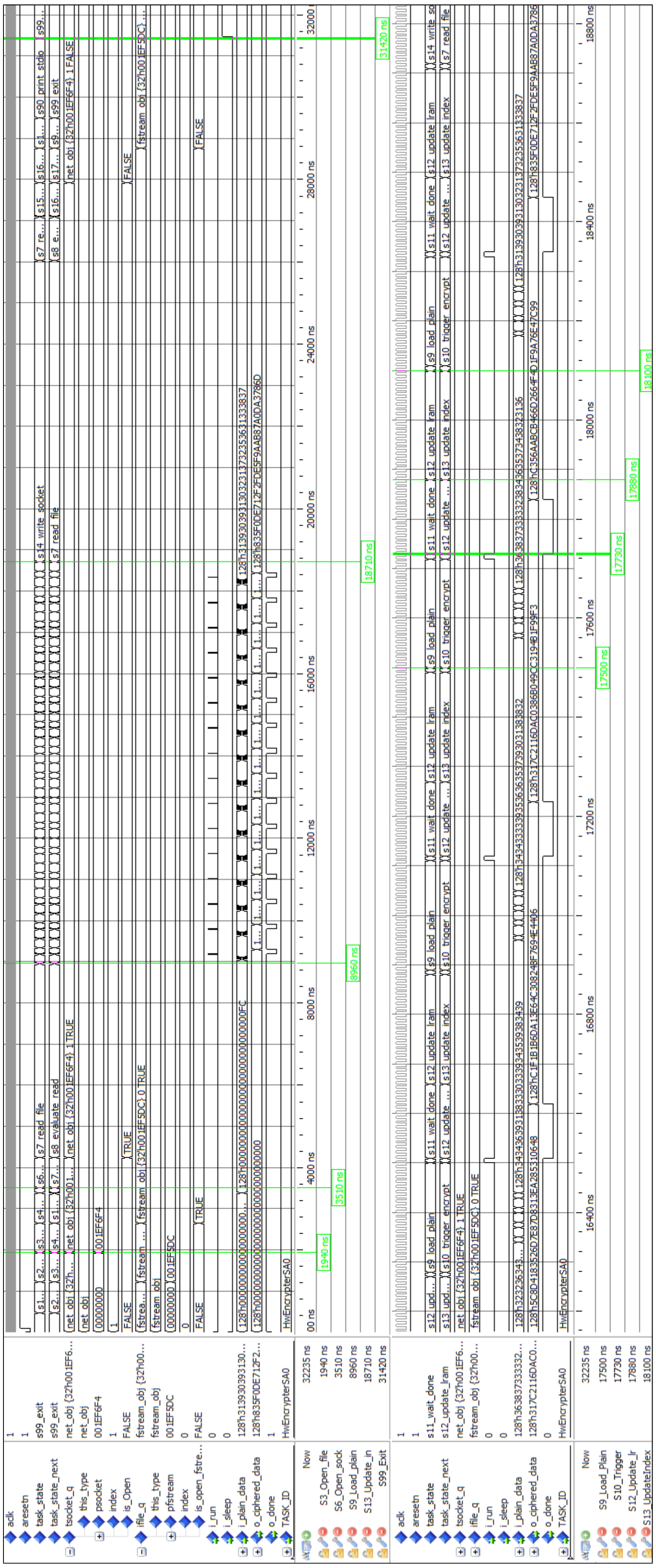


Figure 2.35: Co-Simulation - HW Encryptor SA wave plot using ModelSim.

The bottommost subplot in the same figure, highlights the cyclic behavior in the control unit, and it can be observed the timing diagrams used per cycle. At time 17,500.00 nanoseconds the FSM loads the AES-128 input with the contents of the Local-RAM. At time 17,320.00 nanoseconds it triggers the *run* signal and 10 nanoseconds later achieves the wait state where it stays blocked. At time 17,880.00 nanoseconds the FSM reaches the Local-RAM update state and replaces the current address with the ciphered data, and at time 18,100.00 nanoseconds it evaluates the *encrypt* cycle before proceeding on the *exchange* loop, by writing to the network socket.

The operation concludes at time 31,420.00 nanoseconds (i.e., *S99_Exit marker* in the topmost subplot), where the FSM reaches the final state after closing both virtual objects and transferring the results message to the *stdout*. Both virtual file descriptors were updated to false open and the task remains blocked. The wave plot from an equivalent simulation performed using Vivado simulator tool, can be found in the attached Figure D.2. A close simulation time was achieved, with the task reaching the final state at time 28.3 microseconds. The main differences can be correlated with the states that are dependent on the network operation and ultimately from the tool internal throughput.

From this point in design, the distinct versions of application where functionally validated in the host environment. No commitment to the underlying hardware was established, and the overall design stage provided an appropriate system abstraction that increased comprehension and placed the designer in a better position to make decisions. We now proceed to the platform selection in the deployment phase, implementing the binary files that will instantiate the system.

2.8 Platform deployment

In this section, we address the platform deployment while considering the requirements of the developed application. We choose the underlying hardware that fit such requirements, and produce the binary files that instantiate the system. To assist the designer, we will introduce the Full Simulation model. The Full simulation model allows the designer to validate the complete stack that implements the system. The provided simulation environment considers simulation at the lowermost level, the target architecture and the developed RTL, at the Linux kernel and HAL-ASOS file system, and at the topmost level where the application is instantiated.

2.8.1 Hardware selection

Up until now, we have been developing the application using Xilinx's Vivado, and in the supported set of devices that target Linux embedded systems, we can find the Zynq family. The Zynq family is divided in four ranges of application, namely: (1) the Cost-Optimized range, and includes the Zynq-7000 and Zynq7000S SoC based on Artix devices; (2) the Mid-Range, and includes Zynq UltraScale+ MPSoC CG devices and Zynq-7000 SoC Kintex; (3) the High-End range of UltraScale+ MPSoC EV and EG devices; and (4) the High-End range of UltraScale+ RFSoC devices. We consider that the Machine 1 application requirements do not fit the bills of the Graphics Processing Unit (GPU) provided by the third range, or the High bandwidth RF data converters provided in the fourth range. As so, we will discard such devices and contemplate the use of the Cost Effective or Mid-Range devices. The UltraScale+ device combines a heterogeneous MPSoC using ARM Cortex-A53, a 64-bit multi-core asynchronous processing unit, and ARM Cortex-R5, 32-bit multi-core real-time processing unit, coupled with a system logic cell based programmable logic area. The Zynq-7000 SoC family differs in a logic cell based and less dense programmable area, and single- or dual-core 32-bit ARM Cortex-A9 architectures. We consider that Cost-effective range is the best suit for the application requirements and among the set of available devices we chose the Zynq7000 SoC on the ZC702 board.

The ZC702 board will be selected for this example, as it provides a moderate set of logic resources and also a subset of surrounding hardware that suits the application requirements. It includes the Z-7020-CLG848-1 device, a dual-core ARM Cortex A9 capable of achieving 866MHz CPU clock, and it is the third choice considering dual-core architectures and available logic resources in the programmable area. Figure 2.36 shows the simplified block design for the Machine 1 using a Vivado project that targets the ZC702 board. The Zynq-7000 Processing System (PS) can be seen in the center of the figure, and includes all the static hardware in the system. The remaining functional units are implemented in the Programmable Logic (PL) area, among which stand out the two hardware accelerators coupled with the *HW-Tasks Encryptor* and *Encryptor* SA. To implement the design in the selected platform, we use the HAL-ASOS accelerator V4_00_b component provided by the framework. The V4_00_b provides connectivity with Advanced eXtensible Interface (AXI) bus using the Interconnect IPs for master and slave interfaces. In the current design, we have considered a clock frequency of 100 MHz applied to all logic devices in PL. The detailed representation of this block design can be consulted in the attached Figure D.1.

With respect to the design flow in Vivado, the tool produces a compressed file in *hdf* extension that contains

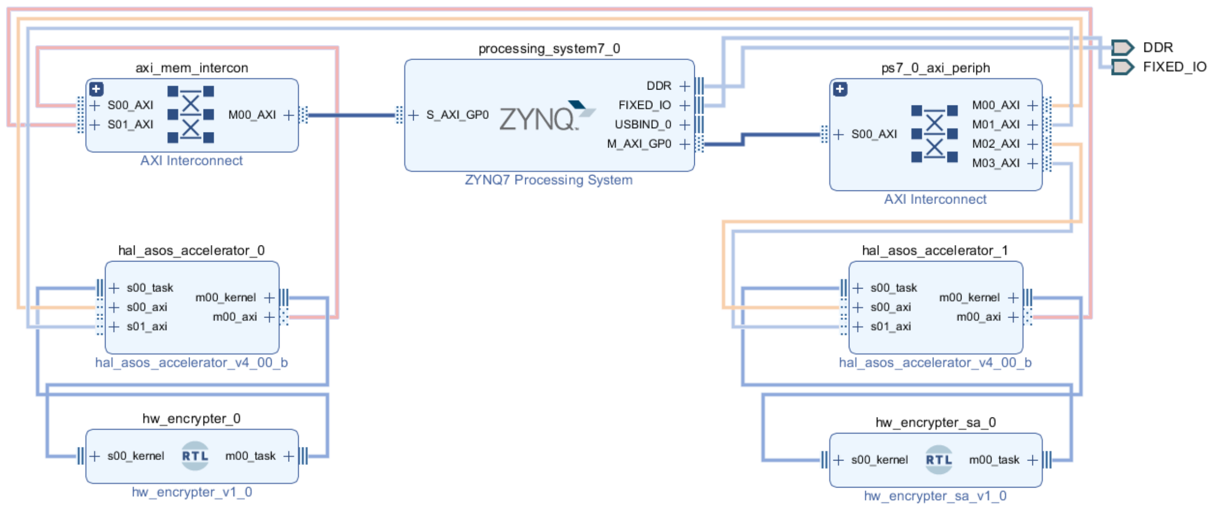
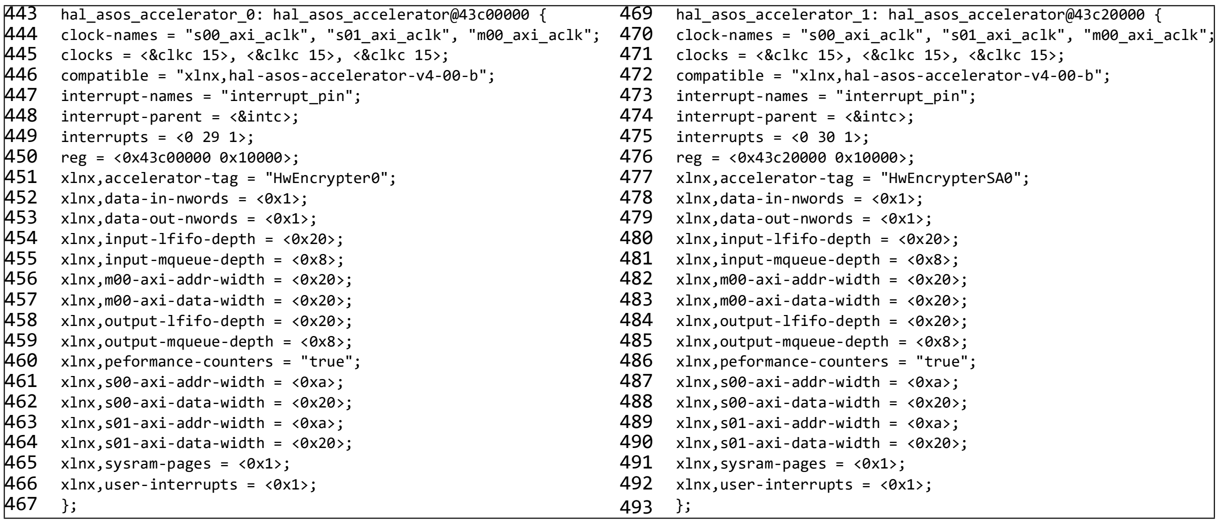


Figure 2.36: Machine 1 -Simplified block design using ZC702 platform.

among other sources, the bitstream file that implements the block design in the PL and an XML file that describes the design. The implemented system can then be exported to an Software Development Kit (SDK) or Vitis project in the Vivado design flow, which provide the necessary compilation toolchain and the platform BSP that allow the designer to build the files that instantiate the system. From this deployment stage two files are produced: (1) a *BOOT.bin* file, that contains a first stage bootloader, the bitstream file and the U-Boot file in the *elf* format; and (2) the *devicetree.dtb* file, a data structure that describes the system in the format capable of being used by the Linux kernel.

The first-stage bootloader is provided by the BSP in one of the build tools, and the second stage bootloader can be generated on a machine-based or custom-based configuration using the U-Boot sources. The device tree can be generated in the build tools and compiled with the Device Tree Compiler (DTC) tool from the host environment. Figure 2.37 outlines the source lines from the device tree that describe the accelerators. Most of these lines refer to parameters configuring the hardware, but are also information to be used by the HAL-ASOS file system while interacting with the implemented hardware.

To show some light in these lines, we can say that the Linux kernel will link the accelerator descriptions with the file system code by using the *compatible* property cell in lines 446 and 472. Two interrupt lines were assigned to each accelerator and they can be seen in the *interrupt* property cells in lines 449 and 475. The first number is 0 and according to Xilinx, it identifies a non-SPI peripheral where the interrupt numbers offset form the number 32. As so, the 29 and 30 numbers correspond to the interrupt lines 61 and 62, and since third number is 1, it specifies the interrupt type as edge-rising. The control interfaces base and range addresses can be read using the *reg* property cells in lines 450 and 476. The accelerator-tag(s) are



```
443 hal_asos_accelerator_0: hal_asos_accelerator@43c00000 {
444 clock-names = "s00_axi_aclk", "s01_axi_aclk", "m00_axi_aclk";
445 clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>;
446 compatible = "xlnx,hal-asos-accelerator-v4-00-b";
447 interrupt-names = "interrupt_pin";
448 interrupt-parent = <&intc>;
449 interrupts = <0 29 1>;
450 reg = <0x43c00000 0x10000>;
451 xlnx,accelerator-tag = "HwEncrypter0";
452 xlnx,data-in-nwords = <0x1>;
453 xlnx,data-out-nwords = <0x1>;
454 xlnx,input-lfifo-depth = <0x20>;
455 xlnx,input-mqueue-depth = <0x8>;
456 xlnx,m00-axi-addr-width = <0x20>;
457 xlnx,m00-axi-data-width = <0x20>;
458 xlnx,output-lfifo-depth = <0x20>;
459 xlnx,output-mqueue-depth = <0x8>;
460 xlnx,peformance-counters = "true";
461 xlnx,s00-axi-addr-width = <0xa>;
462 xlnx,s00-axi-data-width = <0x20>;
463 xlnx,s01-axi-addr-width = <0xa>;
464 xlnx,s01-axi-data-width = <0x20>;
465 xlnx,sysram-pages = <0x1>;
466 xlnx,user-interrupts = <0x1>;
467 };
```

```
469 hal_asos_accelerator_1: hal_asos_accelerator@43c20000 {
470 clock-names = "s00_axi_aclk", "s01_axi_aclk", "m00_axi_aclk";
471 clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>;
472 compatible = "xlnx,hal-asos-accelerator-v4-00-b";
473 interrupt-names = "interrupt_pin";
474 interrupt-parent = <&intc>;
475 interrupts = <0 30 1>;
476 reg = <0x43c20000 0x10000>;
477 xlnx,accelerator-tag = "HwEncrypterSA0";
478 xlnx,data-in-nwords = <0x1>;
479 xlnx,data-out-nwords = <0x1>;
480 xlnx,input-lfifo-depth = <0x20>;
481 xlnx,input-mqueue-depth = <0x8>;
482 xlnx,m00-axi-addr-width = <0x20>;
483 xlnx,m00-axi-data-width = <0x20>;
484 xlnx,output-lfifo-depth = <0x20>;
485 xlnx,output-mqueue-depth = <0x8>;
486 xlnx,peformance-counters = "true";
487 xlnx,s00-axi-addr-width = <0xa>;
488 xlnx,s00-axi-data-width = <0x20>;
489 xlnx,s01-axi-addr-width = <0xa>;
490 xlnx,s01-axi-data-width = <0x20>;
491 xlnx,sysram-pages = <0x1>;
492 xlnx,user-interrupts = <0x1>;
493 };
```

Figure 2.37: Programmable Logic - device tree source two accelerators.

used at file system level to create the task directory, and can be read using the property cells in lines 451 and 477. Some of these parameters will also be used in this section, as they are of utmost importance to handshake of the block design using the Full simulation model. The overall use of this description will be best discussed in the HW implementation details of the Accelerator model in Chapter 3.

The designer will also need a Linux distribution that targets the hardware and software stacks, and in some cases, a customized Linux version that fits in the application requirements with just the right packages is the desirable approach. In such case, an automated compilation tool is commonly used and examples of these are OpenEmbedded [28] or Buildroot. We have selected Buildroot to generate the Linux distribution for Machine 1, and generically, three files are produced: a U-boot executable *elf* format file, *uboot.elf*, that can be used in the SDK to generate the *BOOT.bin* file; a *uImage* file, containing the Linux kernel image in a format that can be used by the bootloader; and a *rootfs.ext2* file that contains the compressed file system for Linux. Also, generic Linux image in the compressed format can also be generated, and such a file, *zImage*, will be used by the Full simulation model.

To support the Accelerator features in the Linux OS, the designer will need to add the HAL-ASOS file system sources to the Linux kernel, by applying a file patch that matches the pre-selected kernel version. Alternately the sources can be compiled separately as kernel module and mounted at boot time. Such compilation can be performed using the host cross-compile environment provided by Buildroot. To comply with this section descriptions, we have compiled the Linux kernel version 4.9.0, in the *zImage* format, with the debug symbols, and already includes the HAL-ASOS file system sources.

2.8.2 Full simulation Model

The Full Simulation model of the HAL-ASOS framework, allows the designer to validate the system in an integrated environment that includes all implementation domains, from the accelerators that exist in the RTL simulation, the hardware devices the exist in the selected board, the Linux kernel binaries and file systems, until the Linux user-space where the application will be executed. For that, it relies on the QEMU platform, that allows to functionally emulate all the existing hardware on the system. To comply with the RTL simulation, using the local device representation, the framework extends the traditional set of QEMU devices with the *hal_asos_qdev* structure.

Generically, within a specific domain, such device implementation uses a shared network connection, and forwards the subsequent read and write requests that result from guest code, to the simulation environment where the target hardware is being simulated. In the opposite direction, another channel is used the provide access to the system resources and trigger interrupt events that can result from the hardware simulation. Figure 2.38 depicts a simplified diagram that describes this model considering the example that integrates an RTL simulation tool.

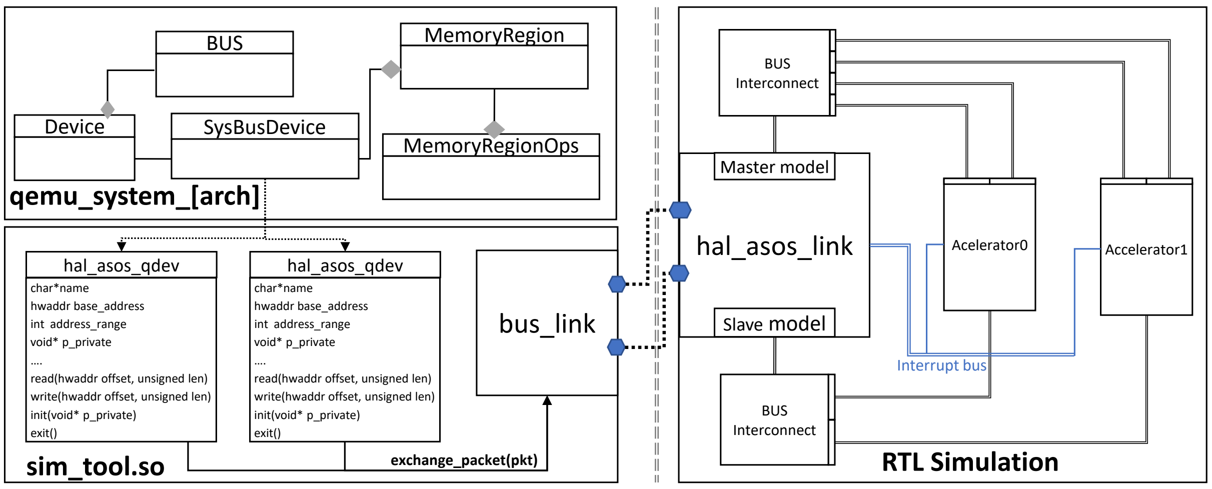


Figure 2.38: HAL-ASOS Full simulation Model

With regard to software implementation, this model can be decomposed into three distinct domains: (1) the extended QEMU device model, that creates an intermediate and generic representation ready to be used with many different simulation tools; (2) the dynamic implementation, that allows reusing the implemented features with the specifics of each simulation tools, and thus reduce excessive footprint in the QEMU sources and consequent recompilations; and finally (3) using the programming interfaces available in each simulation tool, establish the endpoints in the network connections that fit the device network channels. As for the RTL simulation, the model assumes once again two implementation variants, using VHDL FLI- or SystemVerilog DPI-based implementations in the ModelSim tool or SystemVerilog DPI-based implementation in the Vivado simulation tool.

On the RTL side, the model stimulates the design and allows the user to explore how the accelerators relate to the system bus, or how they synchronize with the application on the Host side, here considered QEMU guest. The *hal_asos_link* component initiates the connection with QEMU side, and in the handshake phase, exchange the tool and accelerators intrinsic details. These include the tool name and address range, and accelerator parameters namely, the task tag, base address and address range, and the interrupt offset. Each device is then registered accordingly to map the corresponding accelerator in the QEMU machine virtual representation.

On QEMU side, at the top of the hierarchy, stands the bus where the devices are attached. For every device, there is a memory region and a file operation structure that uses each device specific code. Once in service, and as consequence of the software execution, whenever QEMU needs to access the memory region where the device is mapped, it launches the execution of the read or write functions from the registered object. These implementations evaluate the address and the size of the request and forward and appropriate message to the RTL simulation.

On the RTL side, the receiving requests from the network channel are forwarded to the Master Model. Such model implements the procedures that exchange data with the *HW-Kernel* resources using the accelerator differentiated interfaces, i.e., the *S00_Control* and *S01_Data*. Whenever an interrupt line is triggered, or an accelerator accesses the system's memory, the *hal_asos_link* component intermediates such request by using the Slave Model, and proceeds by forwarding an appropriate message to the QEMU side. Upon receiving such message, the *bus_link* on QEMU side uses native API to interact with the memory or interrupt subsystems and complete the received request.

Despite presenting a wider validation chain, this model can replace the Co-simulation model and reduce the number of iterations in the design flow. For this reason, once achieving the deployment phase, the design flow does not iterate back to the Co-Simulation phase, assuming that it can validate the application in full simulation environment. Care must be taken when dealing with design refinements in this stage and appropriate planning is required.

2.8.3 Full system simulation

To apply the Full simulation model to the Machine 1 application the designer needs to provide the means for connecting the accelerators that exist in the simulation tool with the virtual device representation on QEMU. For that, it will replace the processing system in the block design of Figure 2.36, by the *hal_asos_link* component that fit the SystemVerilog implemented version. Alternatively, it can create a new block design and keep both designs in the Vivado project, to be used them in the subsequent design iterations. Figure 2.39 depicts the block design for Full simulation of the Machine 1 application using Vivado. To highlight the connections, the accelerator interface lines where manually colored in: light purple for the accelerators' master interfaces; light orange for the control-oriented slave interfaces; and light blue for data-oriented slave interfaces.

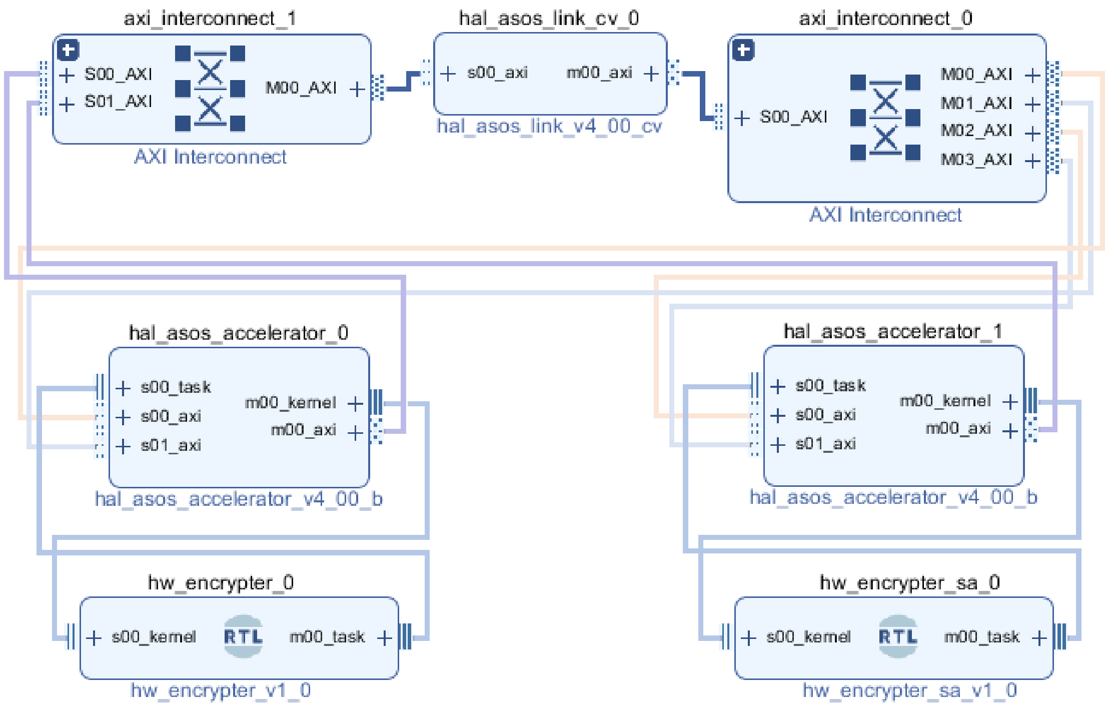
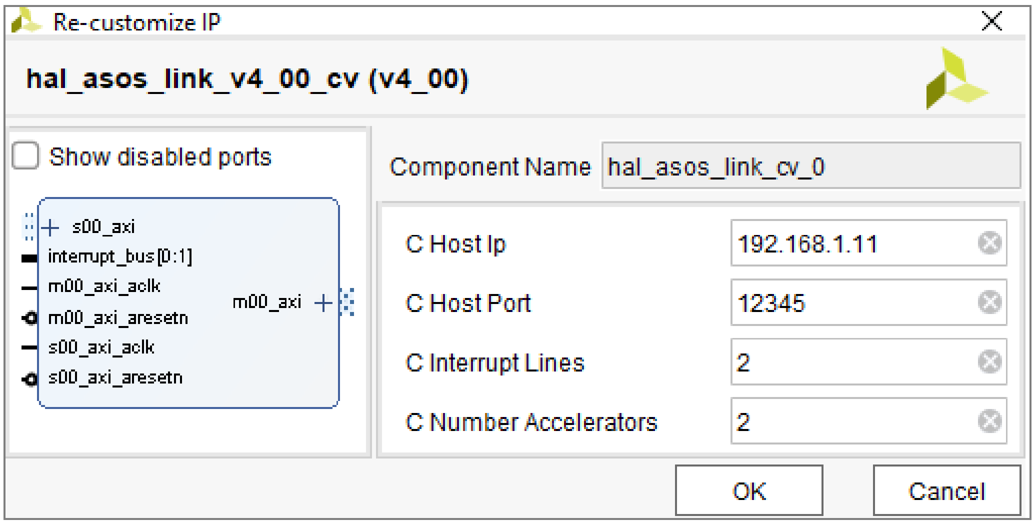


Figure 2.39: Full Simulation - HW Encryptors block design

Similarly, as in the co-simulation model, the *hal_asos_link* component requires parametrizable settings that configure the network connection. Figure 2.40 depicts the Vivado component generated interface, and includes the parameters used in this simulation. Such block design requires one interrupt for accelerator and two accelerators are in the block design. We use the same IP address as in the Co-Simulation but a different port that depends in the tool specific implementation.



Component Name: hal_asos_link_cv_0

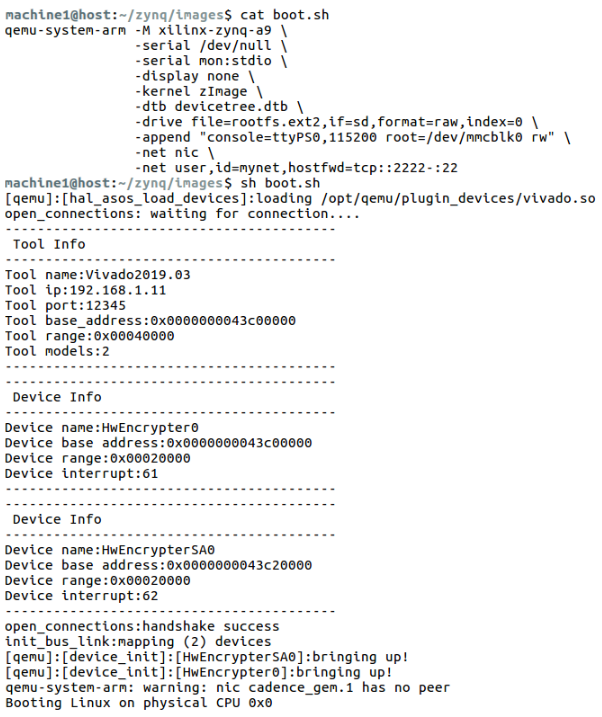
C Host Ip	192.168.1.11
C Host Port	12345
C Interrupt Lines	2
C Number Accelerators	2

Figure 2.40: Full Simulation - hal_asos_link simulation parameters

On the QEMU side a custom implementation for Vivado tool was compiled as dynamic library and during the QEMU machine initialization, the HAL-ASOS extension will load all the *.so* files that can be found in the libraries path. In this example, a *vivado.so* dynamic library is loaded, and QEMU will wait for a parametrizable time until receives connection from the simulation tool, before proceeds with the handshake. Figure 2.41 depicts the handshake log messages at both sides of the simulation.

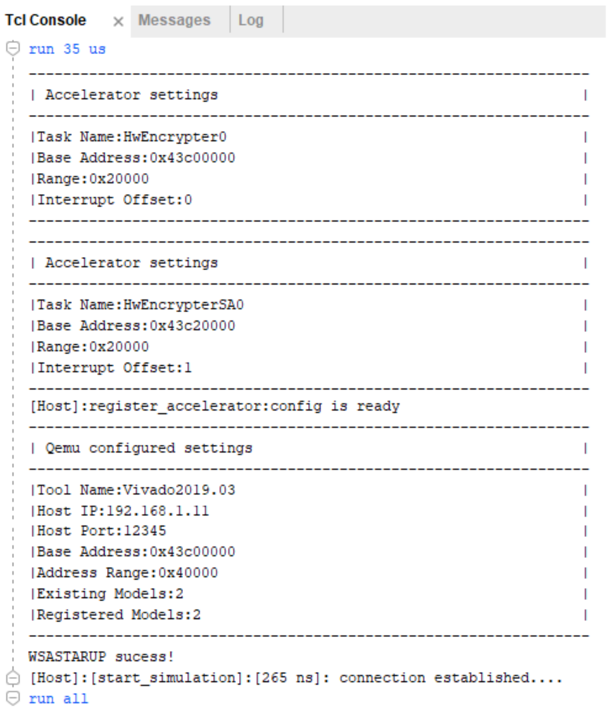
In Figure 2.41a, a *cat* command exposes the used boot settings. We have selected the QEMU pre-defined machine *xilinz_zynq_a9* and the compressed Linux generic image, *zImage* will be used. The *rootfs.ext2* file will be mounted as SD device and the append switch specifies the kernel boot command. After loading the symbols in the *vivado.so* library the *open_connection* of the *bus_link* was executed. Upon receiving connection, the details about the simulation tool and the existing accelerators were exchanged. At concluding the handshake, the simulation parameters were used to register two devices using the device model extension, and once all hardware devices where initialized, the QEMU proceeded by booting the Linux image.

In the Figure 2.41b we can see the handshake log messages in the Vivado TCL console. The first lines list the accelerator parameters and result from SystemVerilog registry functions. At 265 nanoseconds of simulation time, the *hal_asos_link* successfully established connection with QEMU. A continuous run command was issued, and the simulation proceeds until the designer decides to manually stop the simulation, or issues a *shutdown* on the Linux console of the guest machine and closes the QEMU execution. The testbench source used in this simulation is a generic example provided by the HAL-ASOS framework and can be consulted in the attached Figure C.11.



```
machine1@host:~/zynq/images$ cat boot.sh
qemu-system-arm -M xilinx-zynq-a9 \
                -serial /dev/null \
                -serial mon:stdio \
                -display none \
                -kernel zImage \
                -dtb devicetree.dtb \
                -drive file=rootfs.ext2,if=sd,format=raw,index=0 \
                -append "console=ttyPS0,115200 root=/dev/mmcblk0 rw" \
                -net nic \
                -net user,id=mynet,hostfwd=tcp::2222-:22
machine1@host:~/zynq/images$ sh boot.sh
[qemu]:[hal_asos_load_devices]:loading /opt/qemu/plugin_devices/vivado.so
open_connections: waiting for connection....
-----------------------------------------
 Tool Info
-----------------------------------------
Tool name:Vivado2019.03
Tool ip:192.168.1.11
Tool port:12345
Tool base_address:0x0000000043c00000
Tool range:0x00040000
Tool models:2
-----------------------------------------
-----------------------------------------
 Device Info
-----------------------------------------
Device name:HwEncrypter0
Device base address:0x0000000043c00000
Device range:0x00020000
Device interrupt:61
-----------------------------------------
-----------------------------------------
 Device Info
-----------------------------------------
Device name:HwEncrypterSA0
Device base address:0x0000000043c20000
Device range:0x00020000
Device interrupt:62
-----------------------------------------
open_connections:handshake success
init_bus_link:mapping (2) devices
[qemu]:[device_init]:[HwEncrypterSA0]:bringing up!
[qemu]:[device_init]:[HwEncrypter0]:bringing up!
qemu-system-arm: warning: nic cadence_gem.1 has no peer
Booting Linux on physical CPU 0x0
```

(a) QEMU handshake log



```
run 35 us
---------------------------------------------------------------
| Accelerator settings                                        |
---------------------------------------------------------------
|Task Name:HwEncrypter0                                       |
|Base Address:0x43c00000                                      |
|Range:0x20000                                                |
|Interrupt Offset:0                                           |
---------------------------------------------------------------
---------------------------------------------------------------
| Accelerator settings                                        |
---------------------------------------------------------------
|Task Name:HwEncrypterSA0                                     |
|Base Address:0x43c20000                                      |
|Range:0x20000                                                |
|Interrupt Offset:1                                           |
---------------------------------------------------------------
[Host]:register_accelerator:config is ready
---------------------------------------------------------------
| Qemu configured settings                                    |
---------------------------------------------------------------
|Tool Name:Vivado2019.03                                      |
|Host IP:192.168.1.11                                         |
|Host Port:12345                                              |
|Base Address:0x43c00000                                      |
|Address Range:0x40000                                        |
|Existing Models:2                                            |
|Registered Models:2                                          |
---------------------------------------------------------------
WSASTARUP sucess!
[Host]:[start_simulation]:[265 ns]: connection established....
run all
```

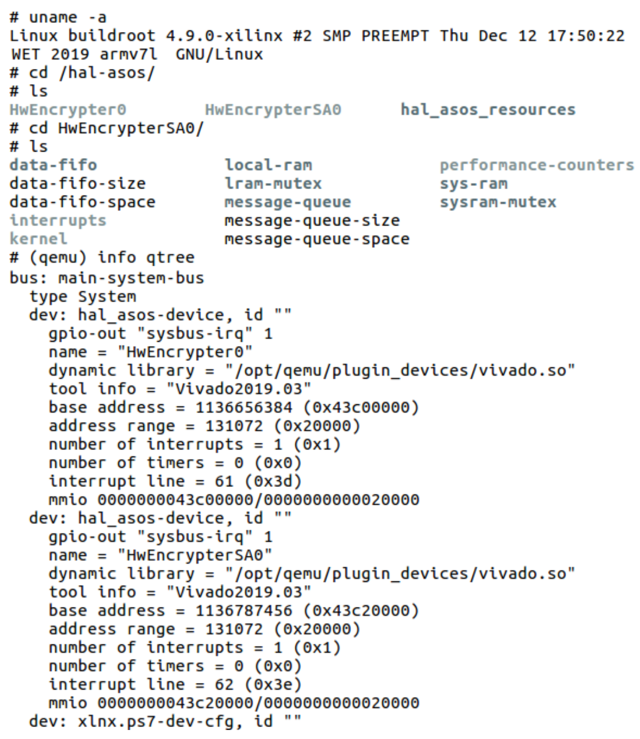
(b) Vivado Simulator handshake log

Figure 2.41: Full Simulation - QEMU and Vivado handshake log

Once QEMU booted the Linux image, a set of commands were executed with the purpose of confirming the correct state of the system, and the results can be seen in Figure 2.42a. A '*uname*' command displays information about the system: a Linux 4.9.0 preemptible kernel for SMP architectures is in use and is based on ARMv7l machines. A '*ls*' command lists the available accelerators in the HAL-ASOS file system. Two directories with the accelerator's names are found, and contain the virtual files that export the accelerator model to the Linux user-space. Entering the QEMU monitor console, an '*info qtree*' command lists the devices in the system and it can be seen the accelerators that were registered in the handshake phase.

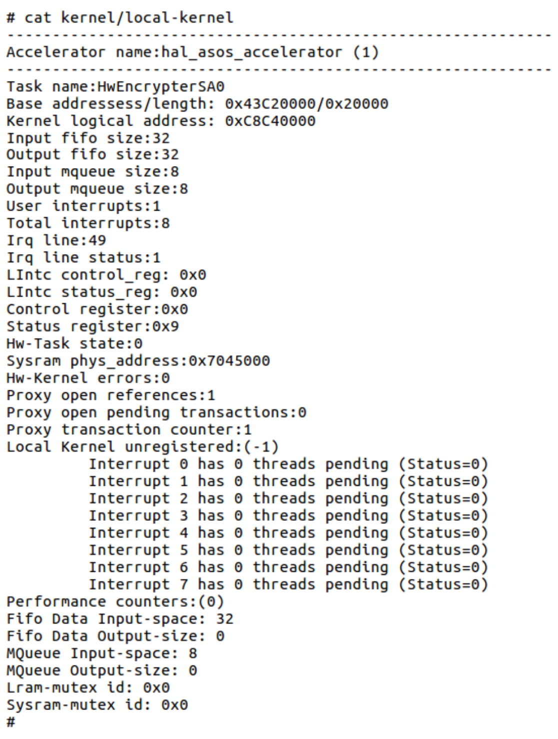
On Figure 2.42b it can be seen a console formatted message, from the local-kernel virtual file read that lists the current accelerator status. It includes information such as: the accelerator name; the base address and combined address range; the configured design parameters; some Linux kernel assigned resources; internal statistics and status of the local-kernel registry; but is also displayed the overall status of the *HW-Kernel* functional units being simulated in the RTL side.

Considering that the Linux image was successfully booted, the handshake established a connection between the emulated hardware in QEMU and the RTL simulation on Vivado, we now proceed with the



```
# uname -a
Linux buildroot 4.9.0-xilinx #2 SMP PREEMPT Thu Dec 12 17:50:22
 WET 2019 armv7l  GNU/Linux
# cd /hal-asos/
# ls
HwEncrypter0          HwEncrypterSA0       hal_asos_resources
# cd HwEncrypterSA0/
# ls
data-fifo              local-ram             performance-counters
data-fifo-size         lram-mutex            sys-ram
data-fifo-space        message-queue         sysram-mutex
interrupts             message-queue-size
kernel                 message-queue-space
# (qemu) info qtree
bus: main-system-bus
  type System
  dev: hal_asos-device, id ""
    gpio-out "sysbus-irq" 1
    name = "HwEncrypter0"
    dynamic library = "/opt/qemu/plugin_devices/vivado.so"
    tool info = "Vivado2019.03"
    base address = 1136656384 (0x43c00000)
    address range = 131072 (0x20000)
    number of interrupts = 1 (0x1)
    number of timers = 0 (0x0)
    interrupt line = 61 (0x3d)
    mmio 0000000043c00000/0000000000020000
  dev: hal_asos-device, id ""
    gpio-out "sysbus-irq" 1
    name = "HwEncrypterSA0"
    dynamic library = "/opt/qemu/plugin_devices/vivado.so"
    tool info = "Vivado2019.03"
    base address = 1136787456 (0x43c20000)
    address range = 131072 (0x20000)
    number of interrupts = 1 (0x1)
    number of timers = 0 (0x0)
    interrupt line = 62 (0x3e)
    mmio 0000000043c20000/0000000000020000
  dev: xlnx.ps7-dev-cfg, id ""
```

(a) QEMU - Survey HAL-ASOS file system and monitor



```
# cat kernel/local-kernel
-------------------------------------------------------------
Accelerator name:hal_asos_accelerator (1)
-------------------------------------------------------------
Task name:HwEncrypterSA0
Base addressess/length: 0x43C20000/0x20000
Kernel logical address: 0xC8C40000
Input fifo size:32
Output fifo size:32
Input mqueue size:8
Output mqueue size:8
User interrupts:1
Total interrupts:8
Irq line:49
Irq line status:1
LIntc control_reg: 0x0
LIntc status_reg: 0x0
Control register:0x0
Status register:0x9
Hw-Task state:0
Sysram phys_address:0x7045000
Hw-Kernel errors:0
Proxy open references:1
Proxy open pending transactions:0
Proxy transaction counter:1
Local Kernel unregistered:(-1)
         Interrupt 0 has 0 threads pending (Status=0)
         Interrupt 1 has 0 threads pending (Status=0)
         Interrupt 2 has 0 threads pending (Status=0)
         Interrupt 3 has 0 threads pending (Status=0)
         Interrupt 4 has 0 threads pending (Status=0)
         Interrupt 5 has 0 threads pending (Status=0)
         Interrupt 6 has 0 threads pending (Status=0)
         Interrupt 7 has 0 threads pending (Status=0)
Performance counters:(0)
Fifo Data Input-space: 32
Fifo Data Output-size: 0
MQueue Input-space: 8
MQueue Output-size: 0
Lram-mutex id: 0x0
Sysram-mutex id: 0x0
#
```

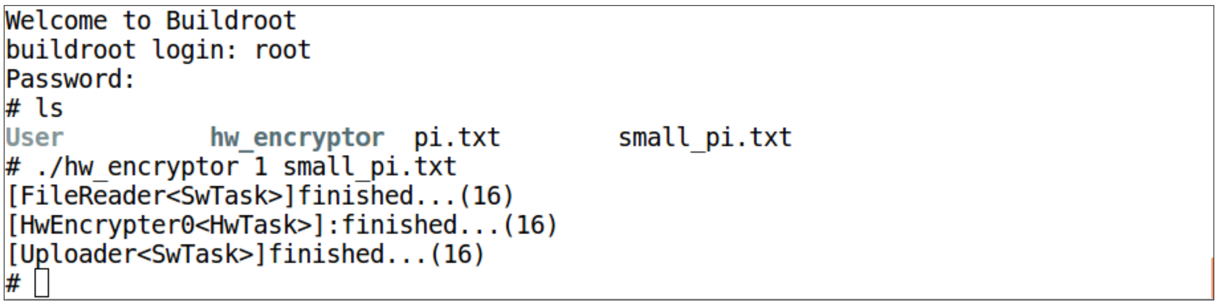
(b) QEMU - File system query to accelerator kernel

Figure 2.42: Full Simulation - running Linux image on QEMU

functional validation of the Machine 1. For this stage, the Machine 1 application was once more compiled, but this time using the host cross-compilation toolchain.

To address the different application variants, the main function was updated to receive parameters form the command line that specify both, the application variant and the input source. The attached Figure C.12 lists these software changes. The Machine 1 application was reconfigured to use the *HwTask* qualifier and recompiled with debug symbols. The attached Figure C.13 lists the corresponding source. Figure 2.43 depicts the output log from the Machine 1 functional that selects the use the HW *Encryptor* task. A '*ls*' command lists the files in the home directory and two text files that contain 1,000,002 bytes and 252 bytes are listed. The file *hw_encryptor* file is the application binary in the *elf* format. The log that results from the execution, demonstrates similar results as in the Co-Simulation stage, and we can observe the *HwTask* as prefix of the *Encryptor* task log. Since the task is configured to use the *StandardIO* profile no qualifier is printed.

On the RTL side, a simulation waveform was plotted and once more the image is divided in wide and magnified views. Figure 2.44 at top, depicts the overall execution time of the *Encryptor* task. At time 1,813.235 microseconds the *HW-Kernel* received the control information to start the *HW-Task* and 151.38



```
Welcome to Buildroot
buildroot login: root
Password:
# ls
User          hw_encryptor  pi.txt        small_pi.txt
# ./hw_encryptor 1 small_pi.txt
[FileReader<SwTask>]finished...(16)
[HwEncrypter0<HwTask>]:finished...(16)
[Uploader<SwTask>]finished...(16)
# 
```

Figure 2.43: Full Simulation - Machine 1 application using HW *Encryptor* task

microseconds later, the task enters the state *s3* to read the first word from the Local-RAM. The encryption of the 256 bytes completes 9.61 microseconds later and after 943.504 microseconds the task completes the transfer to the DDS topic. The execution concludes 20.775 microseconds later. We can observe that due to the multiple implementation levels, the *HW-Task* spends most of the time waiting on the interaction with the simulation on QEMU. The light blue strips, indicate activity in the accelerator bus interfaces. The *S00_AXI* is connected with the control-oriented channel and the *S01_AXI* is connected with the data-oriented channel of the Accelerator model.

At the bottom of Figure 2.44, we zoom in to see the details of the activity on those signals. The purple blocks represent read transactions, and the shaded red blocks represent write transactions. The *read*, *write*, *read* pattern is used to lock the Local-RAM *mutex* with the transaction ID 0x5, and the '1' value on the *o_status*[31] bit, indicates a locked *mutex*. Consecutive write transactions are performed in the *S01_AXI* interface, to transfer the plain data from the DDS subscription to the Local-RAM. The first word 0xfc is the control field and indicates an effective file length of 252 bytes. Three clock cycles per word are used to write the *S01_AXI* interface and an interval of eight clocks is used in the internal operation of the *hal_asos_link* component.

Similarly, the HW *Encryptor* SA task was also validated and Figure 2.45 shows the application log in the Linux command line at QEMU. The attached Figure C.14 lists the used source with the necessary changes. A '*uname*' command lists the information about that system before running the application, and a '*ls*' command repeats the list of contents in the directory. The application is launched with argument '2', which selects the standalone version of the Machine 1, and the same file was used. As expected, we can observe that only one task is in execution, where 16 fragments of plain data where successfully ciphered and consequently uploaded to the Machine 2. The simulation waveform of this execution can be seen in Figure 2.46.

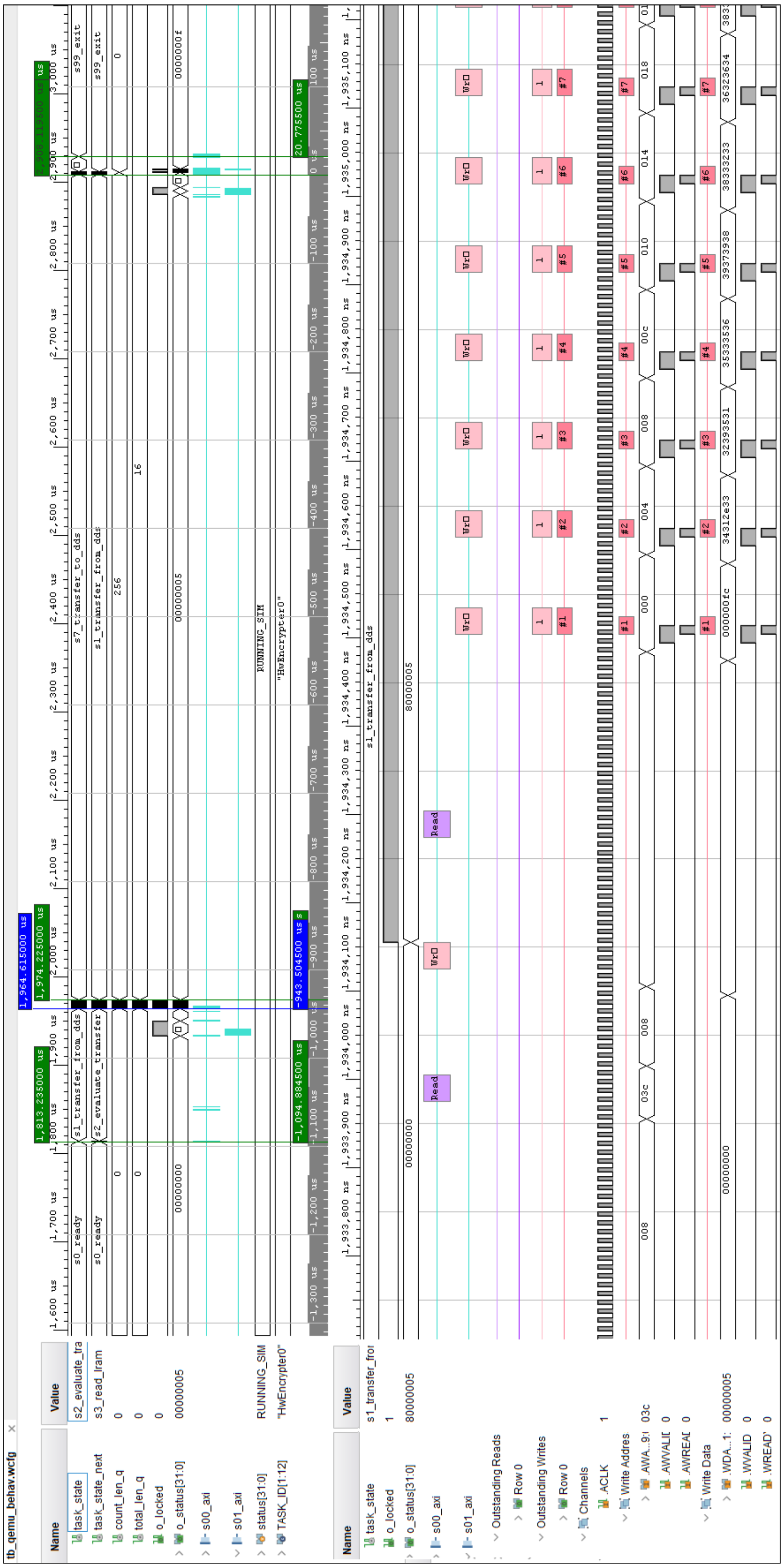
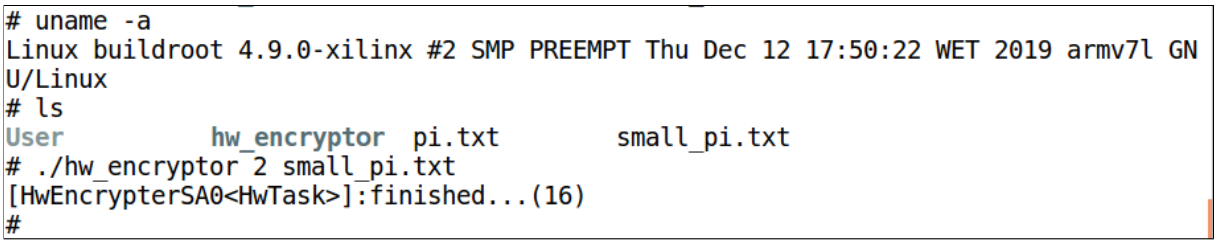


Figure 2.44: Full Simulation - HW *Encryptor* simulation waveform.



```
# uname -a
Linux buildroot 4.9.0-xilinx #2 SMP PREEMPT Thu Dec 12 17:50:22 WET 2019 armv7l GN
U/Linux
# ls
User          hw_encryptor  pi.txt        small_pi.txt
# ./hw_encryptor 2 small_pi.txt
[HwEncrypterSA0<HwTask>]:finished...(16)
#
```

Figure 2.45: Full Simulation - Standalone HW Encryptor design

At top we can see that the application received the run signal at 5,042.705 microseconds and 191.00 microseconds later, the received plain data was ciphered and is ready to be transferred over the network. At time 6,179.695 microseconds, such transfer was completed and 9.95 microseconds later the task concluded. The worst case in the task FSM is once again the network transfer with 916 microseconds duration. It's fair to say that due to the emulated nature of QEMU, the Full-Simulation environment involves a great deal of overhead in the network subsystem, which includes the host development environment, the QEMU network back-end, and the target Linux OS in QEMU.

Analogous bus activity can be observed in the accelerator interfaces. The bottom of Figure 2.46 depicts the last transactions used to read the 64 words of ciphered data from the Local-RAM. Four clocks are spent to read data in a total of eleven clocks per word transaction. The Local-RAM *mutex* remains locked until the transaction completes and once more, the read, write, read pattern appears at the *S00_AXI* interface to unlock the HW *mutex*. The *o_status*[31] bit is cleared indicating the *mutex* is free and the last owner ID was 0x13.

When compared to the results of the Machine 1 Co-Simulation, the Full Simulation requires more clocks per task iteration and it consumes more simulation time. For this reason, the major application design refinements should be addressed during the Co-Simulation phase. The Full simulation provides a powerful simulation environment that is capable of dealing with any potential glitches involving the tree major system components: the Software; the hardware; and the Linux OS. But it also requires a good strategy in applying debug iterations while addressing such faults.

In this section, the binary files that implement the system were cross-compiled to the target hardware and the overall system files were validated. Usually in this phase, we use compilation with debug symbols and as so the binary files might need to be once more re-compiled. We will proceed with the system implementation using the ZC702 board, where the trifecta of application versions will be evaluated.

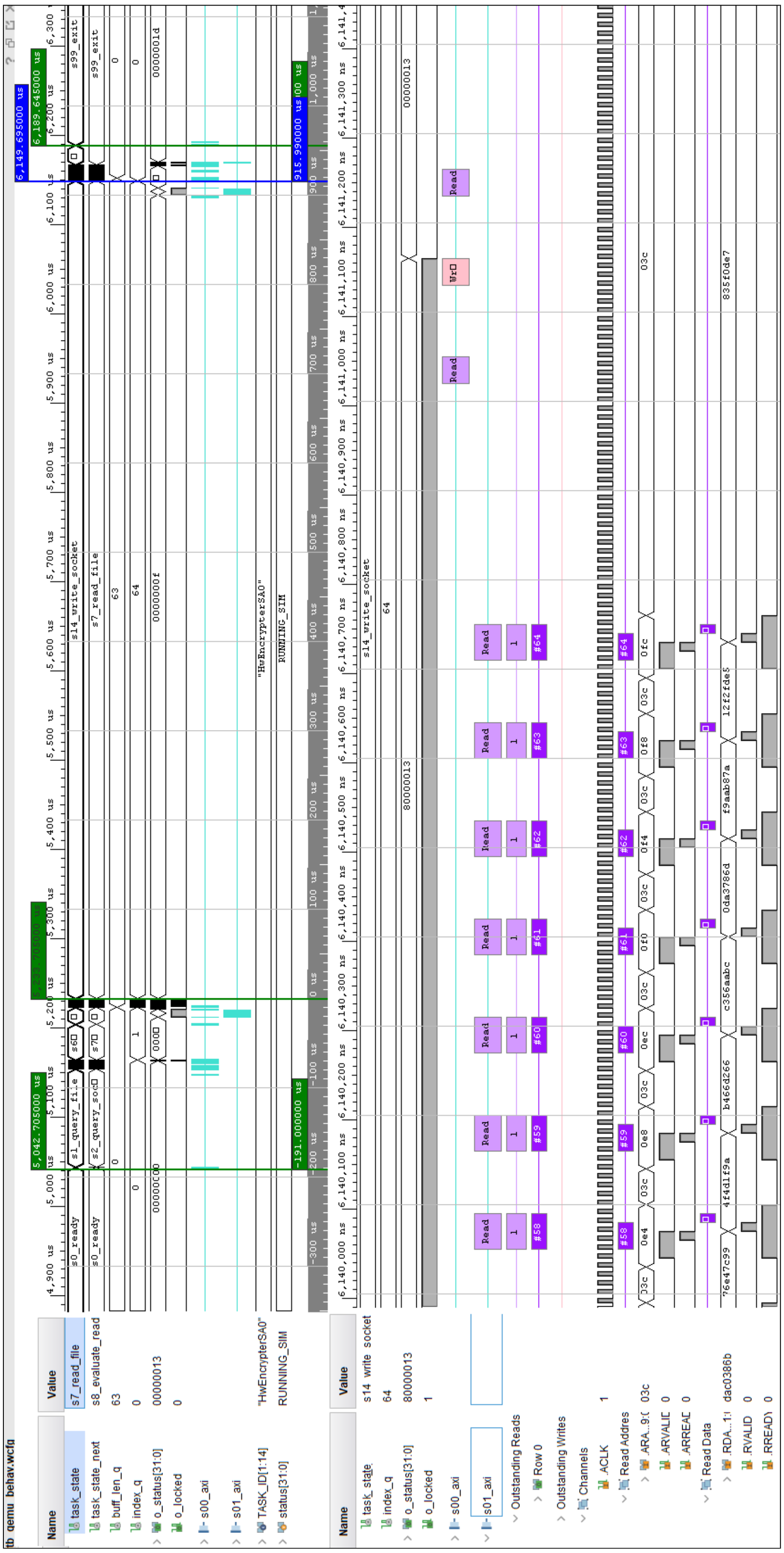
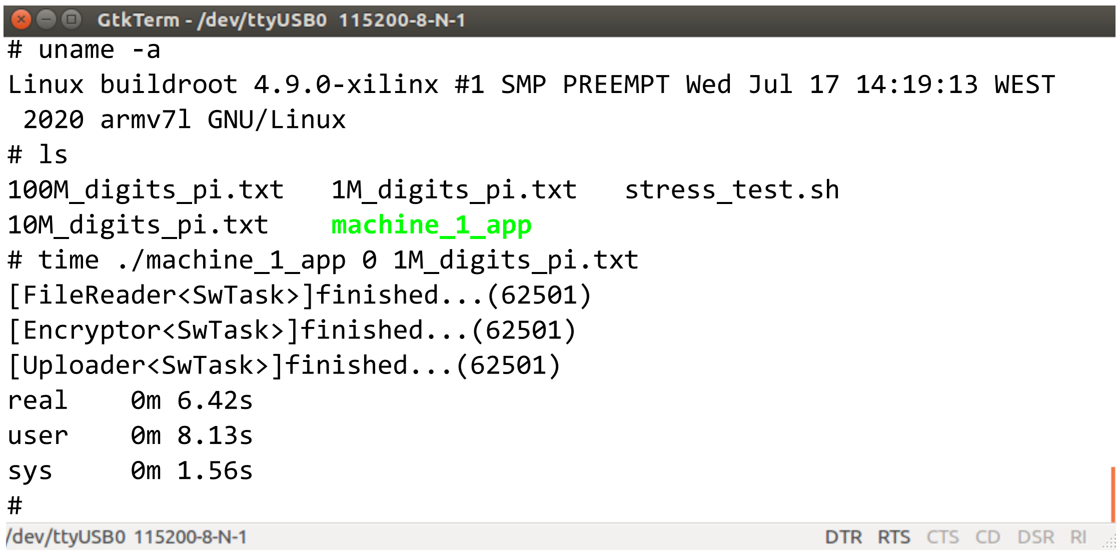


Figure 2.46: Full Simulation - Stand-Alone HW *Encryptor* simulation waveform.

2.9 System Implementation

In this section, we implement the system in the selected board and evaluate the application in the three implementation variants: the software-only, the software and hardware *Encryptor* and the hardware *Encryptor* as standalone. The target system is using the binary files that resulted from the previous section where the bitstream includes the two implemented accelerators using 100 MHz bus clock

In similar manner to the Full simulation tests, the Machine 1 mounts the HAL-ASOS file system at boot-time and two accelerator folders can be found in the *hal-asos* directory, at the root of the Linux file system. The Machine 2 was set to receive the resulting ciphered files and both applications were simultaneously executed. Figure 2.47 shows the results of the Machine 1 running on the ZC702 board. The tests were executed using the '*time*' command that displays an output message with timing statistics about the execution.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# uname -a
Linux buildroot 4.9.0-xilinx #1 SMP PREEMPT Wed Jul 17 14:19:13 WEST
 2020 armv7l GNU/Linux
# ls
100M_digits_pi.txt   1M_digits_pi.txt   stress_test.sh
10M_digits_pi.txt    machine_1_app
# time ./machine_1_app 0 1M_digits_pi.txt
[FileReader<SwTask>]finished...(62501)
[Encryptor<SwTask>]finished...(62501)
[Uploader<SwTask>]finished...(62501)
real    0m 6.42s
user    0m 8.13s
sys     0m 1.56s
#
```

Figure 2.47: Machine 1 - Software-only application version running on ZC702 board.

The output results demonstrate similar log messages from the application. The timing statistics are displayed after the application log and include: (1) the real time between the invocation and the termination, (2) the CPU time spent in the application running the user code and (3) the CPU time spent in the system running the Linux kernel code. The SW-only real time results demonstrate 6.42 seconds spent using one million digits of *pi* as input. The subsequent lines demonstrate that 8.13 seconds were spent running the application code and 1.56 seconds running the system code. The total time value overcomes the real time value and the differences result from the existing parallelism in the application and scheduling the execution using two CPU cores to simultaneously execute the application. Similar messages describe the results of the HW accelerated versions using the same input file and can be seen in Figure 2.48a. The argument '1' selects the HW *Encryptor* implementation, whereas the argument '2' selects the stand-alone version.

It can be seen that using the same input file, 3.58 seconds are spent by the HW *Encryptor* implementation, and 0.30 seconds as spent by HW *Encryptor* SA. Similar results are demonstrated reading the performance counter used to count intervals between task execution. We are using a 100 MHz clock (10 nanoseconds per clock tick), and so the output demonstrates that one task iteration has been counted and took 3.566439440 seconds. The counter starts in the moment that the *run* signal is received from the *HW-Kernel* and stops when it receives the *task_done_i* signal from the *HW-Task*. The application consumes more time when compared to the counter value, as it needs to initialize the internal structures before issuing the start command, and the performance counter stops in the exact time the *HW-Task* reaches the final state. In that moment, the application still needs to complete the file upload and release all allocated resources and thus translating into the additional execution time.

A more accurate test considers statistical data that results from multiple iterations and to best characterize the application behavior, we have increased the file length by 10 and 100 times, and repeated the test in a per-file basis for 10 iterations. Figure 2.48b plots the resulting data from a total of 90 tests (i.e., the 3 application versions, for 3 file lengths, 10 times each trial). The red line indicates the execution time of the SW-Only version, the blue line indicates the HW *Encryptor* and the black line indicates the stand-alone version. We can observe that in the three application versions, the execution time scaled practically linear with the file length.

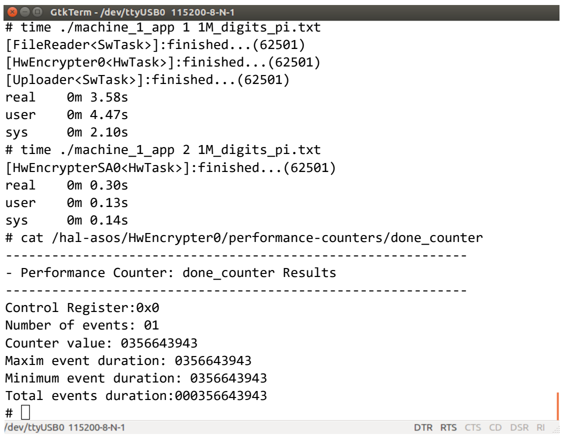
The same data was plotted in a performance ratio comparing the SW-only to the HW accelerated versions. Figure 2.49a depicts these results where the red line presents the performance gain achieved by offloading the *Encryptor* task, and the blue line represents the performance gain achieved with the stand-alone version. When comparing the application with the HW *Encryptor* task, is fair to say that the average gain matches the potential for performance observed in the Profiling stage and becomes more evident when the system utilization increases.

The computing overhead introduced in exchanging data between the software threads and the accelerator was kept in acceptable values, and thus provided the system with effective performance. The stand-alone version completely outranges the performance in the application, achieving a 93% increase in the 100 million digits of *pi*. A fair analysis concludes that such values result mainly from reducing the multi-thread overhead in the synchronization and data exchange threads, and increasing the system performance with the additional accelerator processing.

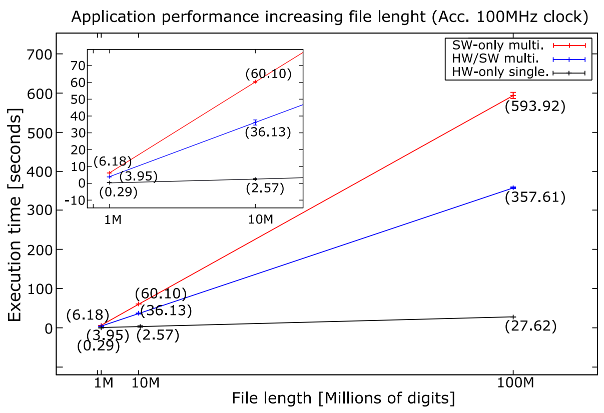
To best characterize the use of resources we have chosen the center value of the three input files, the 10 million digits, and applied different clock sources to the design from 25 MHz to 200 MHz. The SW-only results remain steady since the frequency only affects the accelerators in the PL. Figure 2.49b plots the results from the 100 test iterations. The block design implementation, using the two accelerators at 200 MHz, failed to meet the time constraints in the implementation phase of Vivado design flow. Even though when tested, it was able to produce intermittent results in both designs. The hardware specifications of the clock sources cannot produce 150 MHz clock and the closest match is 142.8 MHz. The two application versions increase performance with clock until the 100 MHz. Beyond this value the performance increase is imperceptible as the Host system cannot respond to the increased demand of the accelerators to avoid starvation in the remanding processes in the system. A final design that does not face power-ware design restrictions can be implemented using the 100 MHz clock, although the 50 MHz designs produce similar performance results with greater efficiency.

2.10 Conclusions

We have already mentioned that the purpose of this chapter is to introduce some of the HAL-ASOS functionalities, and guided by the Design flow, describe the purposed models that compose the design methodology. For this reason, the HW *Encryptor* design followed a simplified design strategy. Several other design techniques could be applied, and it is fair to say that the performance could be improved if for instance, the decision was to use the system channel as opposed to data transfer channel, together with an asynchronous mode of operation between AES pipeline and the memory transfers. In doing so, it would release the potential for performance of the HW true parallel nature by exploring the data-level parallelism of this application. We believe that such level of implementation is not suited to be described in this chapter as it requires increased number of independent functional units and so we have decided for the closeness to SW code approach, that is also capable of being reproduced by most of the commonly used HLS synthesis tools that use C/C++ languages.

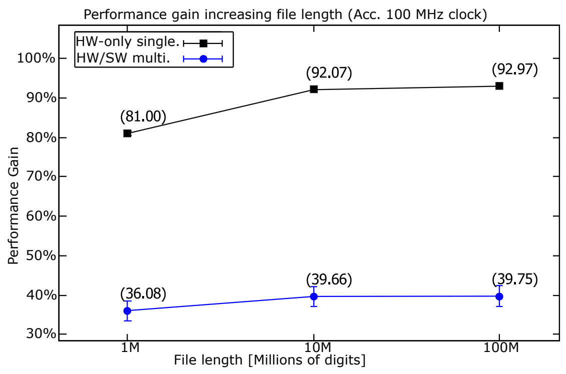

```
# time ./machine_1_app 1 1M_digits_pi.txt
[FileReader<SwTask>]:finished...(62501)
[HwEncrypter0<HwTask>]:finished...(62501)
[Uploader<SwTask>]:finished...(62501)
real    0m 3.58s
user    0m 4.47s
sys     0m 2.10s
# time ./machine_1_app 2 1M_digits_pi.txt
[HwEncrypterSA0<HwTask>]:finished...(62501)
real    0m 0.30s
user    0m 0.13s
sys     0m 0.14s
# cat /hal-asos/HwEncrypter0/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 0356643943
Maxim event duration: 0356643943
Minimum event duration: 0356643943
Total events duration:000356643943
#
```
GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

(a) HW *Encryptors* designs, two accelerators at 100 MHz bus clock.

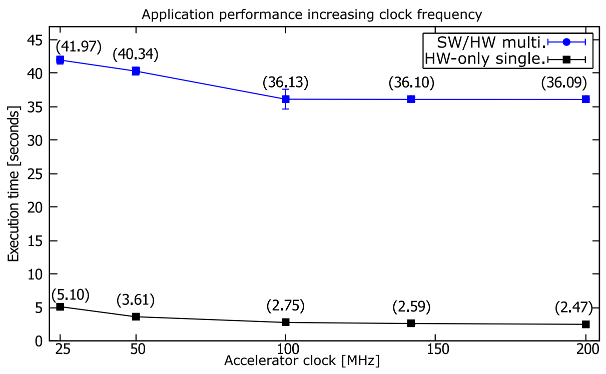


(b) HW *Encryptors* design, two accelerators at 100 MHz bus clock.

Figure 2.48: Zynq ZC702 - Application performance of the machine 1 application.



(a) HW *Encryptors* designs, two accelerators at 100 MHz bus clock.



(b) HW *Encryptors* designs, two accelerators using 10 million digits.

Figure 2.49: Zynq ZC702 - Application performance of the hardware accelerated designs.

Chapter 3

First-class Hardware Components

The accelerator model implements a set of services that integrate the operation of digital logic circuits in software applications for the Linux operating system. For programmability and complexity mitigation, the proposed model redefines the concept of the *HW-Task*, to limit the design scope in the application of such digital circuit. From the point view of the digital circuit, the *HW-Task* represents a local control that integrates its operation in the host system. In the accelerator model, the *HW-Task* and the *Kernel Core* are the prime computing units in the design, that require means to interact with other computing units in the host platform. For this, the *HW-Kernel* implements distinct data- and control-oriented interfaces that allow the host system to address the *HW-Kernel* resources, and allow the *HW-Task* to address the host system memory.

From the accelerator's point of view, an *HW-Task* is a local processing unit that this model promotes to first-class computation entity. For the *HW-Task* operability, the accelerator model provides a *Kernel Core* and establishes an exclusive bi-directional connection between these two components. In this connection, it implements an ambivalent Master-Slave relationship, where the *HW-Task* is the Master that initiates system-level operations in the *Kernel Core*, but behaves as a Slave when these come into service, being subject to the control signals that the execution gives rise to, until a pre-programmed sequence of operations is completed. In this sense, it is fair to say that the concept of computational entity is transferred to the *Kernel Core* when it operates at the service of the *HW-Task*, and is returned back to the caller with the service completion. Figure 3.1 describes the connectivity between the *HW-Task* and the *Kernel Core* while implementing the set of services in the accelerator model.

To handle the distinct service requests, we introduce the concept of HW system calls and deployed these into the *Kernel Core*. An HW system call is a pre-programmed set of control operations that use the

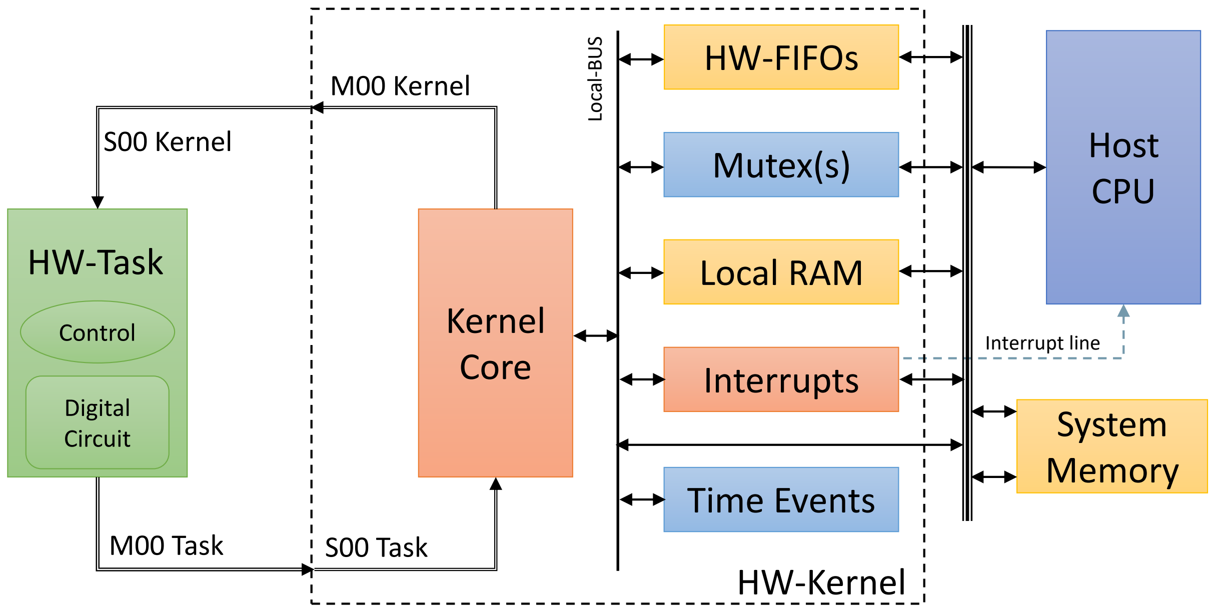


Figure 3.1: Accelerator Model - and overview of the provided services.

local resources to implement services. These services can include time management, synchronization or scheduling of operations, and data movement in the system or local memories. To implement the set of HW system calls, the *Kernel Core* employs micro-code concepts, where all the control and synchronization actions are encoded using a microprogram. In doing so, the *Kernel Core* design benefits from a static and predictable implementation and the microprogram content can be modified to accommodate system architecture impairments. To make these services accessible to the *HW-Task* design, the framework provides a distinct set of procedures that interface the HW system calls in the parameters that these require. Such procedures are defined by the kernel package and are implemented by the *HW-Task* design.

To illustrate the *HW-Task* and *Kernel Core* interoperability, let us consider a simplified design example, where an *HW-Task* that needs to wait for a completion signal in its datapath, before proceeding with the execution. For this, the control unit can implement the wait event procedure while estimating the worst-case scenario in terms of timeout, as opposed to waiting indefinitely. Figure 3.2 depicts a sequence diagram that includes the components involved in this HW system call. In this example, we have established a one-hundred clock period timeout and selected the input *i_flag* as event parameters in the corresponding kernel procedure. At the exact instant the procedure is initiated, the control unit of the *HW-Task* is blocked by the *Kernel Core*, allowing this unit to proceed in configuring and triggering the Time Event unit. After a successful configuration stage, the *Kernel Core* triggers the Time Event Unit to start counting form the specified timeout, and waits for the event detection or an exhausted timeout signal.

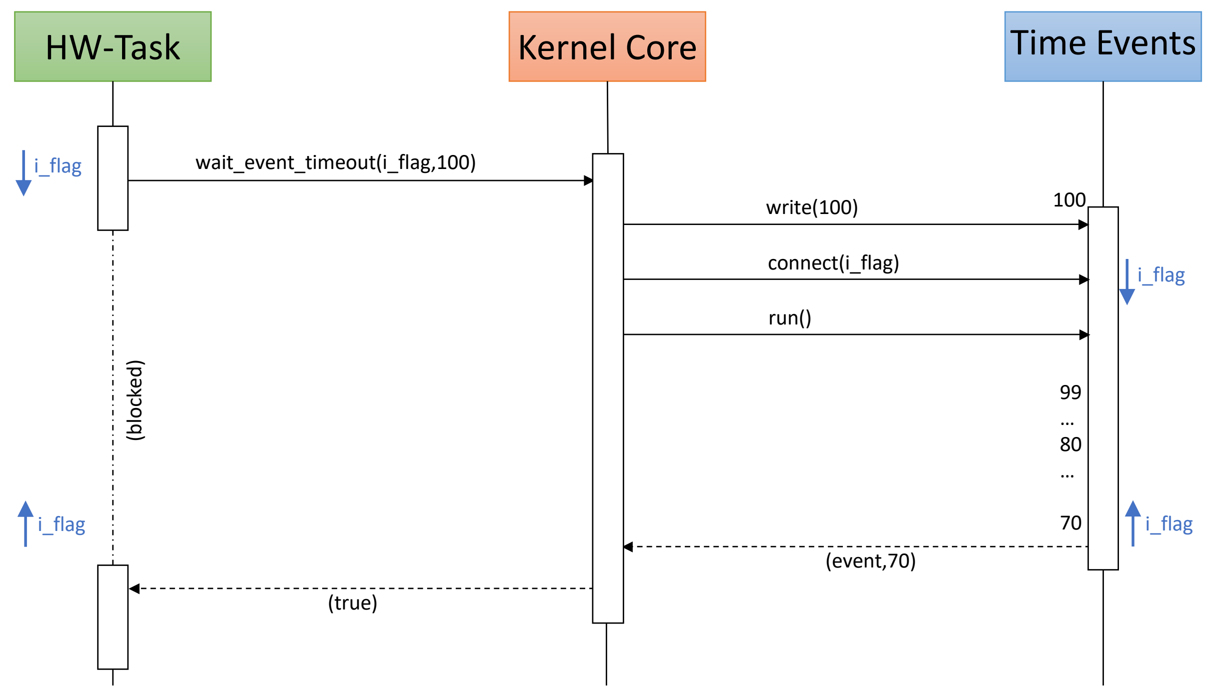


Figure 3.2: Hardware system call - wait event sequence diagram.

Beyond a time-interval of thirty clock cycles, the Time Event unit is triggered by the selected flag and thus completing the wait stage. It then raises the event signal that notifies the *Kernel Core* to conclude the system call, where it establishes a *boolean* response for the event detection and unblocks the *HW-Task*.

Figure 3.3 shows a sequence diagram that describes an operation scenario where the *HW-Task* competes with the Host system for mutual exclusion in the Local-Mutex resource of the accelerator model. Through this feature, it implements contention and concurrency control before writing to the local memory LRAM. In similar way, the *HW-Task* invokes a system call using the *mutex_lock* procedure from the kernel package. It places the *LMutex_offset* as parameter, to specify which of the two HW-Mutex in the accelerator model is the target of the operation. At that moment, the control of the *HW-Task* is blocked and the *Kernel Core* proceeds with the execution of the system call, by writing the *TaskID* in the indicated resource.

At the same time, a SW thread in the Linux OS competes for the same resource, by writing its ID as well. The *Kernel Core* reads the result of its write operation and does not recognize in the received status, as being the owner of this resource. In this condition, it must wait for the availability of the resource while the *HW-Task* remains blocked. A few moments later, the host system thread releases the resource, and the *Kernel Core* reacts by repeating the write operation on the HW-Mutex. The subsequent reading of the status indicates a resource blocked by the kernel, and the *Kernel Core* completes the system call by ensuring exclusivity and releasing the control unit of the *HW-Task*.

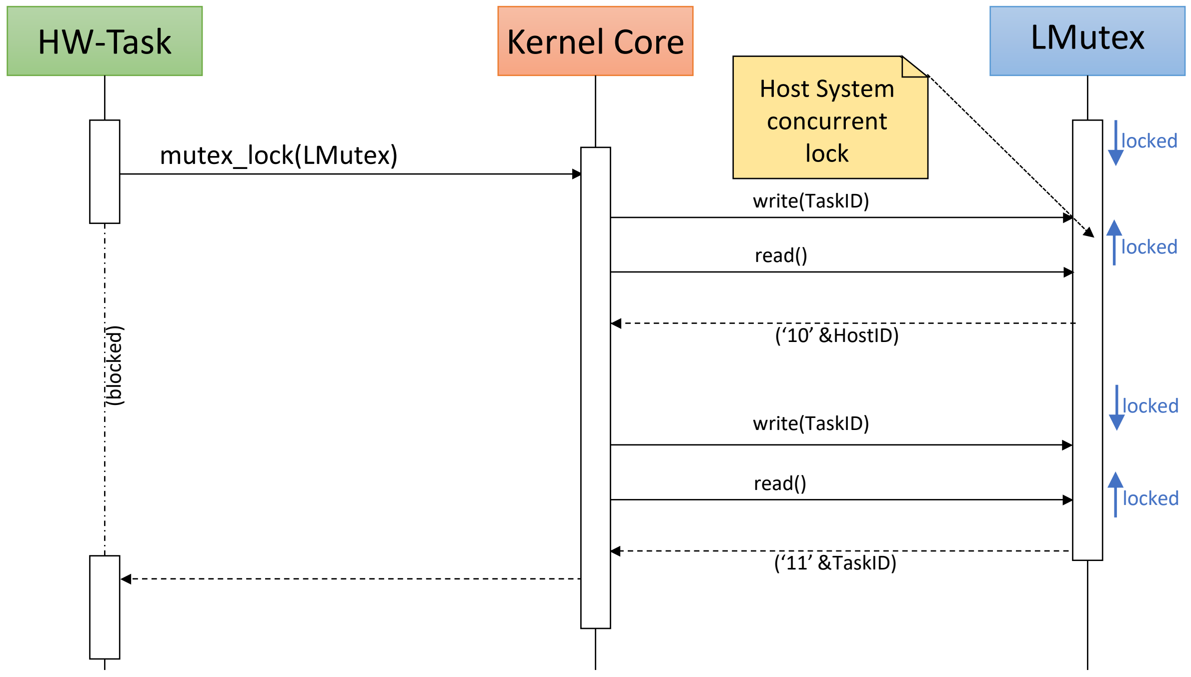


Figure 3.3: Hardware system call - mutex lock sequence diagram.

In the next sections, we discuss the implementation of the *Kernel Core* in the *HW-Task* concept. We then proceed with details of connectivity between the *Kernel Core* and the *HW-Task*, and the available resources in the *Kernel Core*. In section 3.2 we discuss the *HW-Task* template design where we show some application-level operations. We conclude in section 3.3 describing the host system connection with the accelerator model.

3.1 Kernel Core

The *Kernel Core* manages the HW resources in the accelerator model and promotes the integration of the *HW-Task* in the host platform, namely in the interaction with the operating system and the application memory segment. Conceptually, the *Kernel Core* acts like any kernel that can be found in the most elementary OS, by providing a set of services that interact with local resources through system call invocation. In the HAL-ASOS design, these are called HW system calls to distinguish themselves from the similar software-based implementation. To operate at kernel level, the *HW-Task* implements system calls using procedures described in the kernel HDL package. Alternatively, for more complex operations the user HDL procedures can implement consecutive system calls that involve more than one local resource. Procedures can receive input and output parameters that link to resources from the *HW-Task* design. These in turn will allow the system call to access these resources and ultimately update them with execution results.

Figure 3.4 shows a simplified diagram that describes the internal organization of this component. The Control Unit determines the status of the accelerator, and it can be triggered by the active bits in the control register. These in turn can be handled by the host system to address the application functional requirements. Due to the critical nature of these operations, the content of this register is updated under the supervision of the Authenticator device, which validates the received word before authorizing the operation.

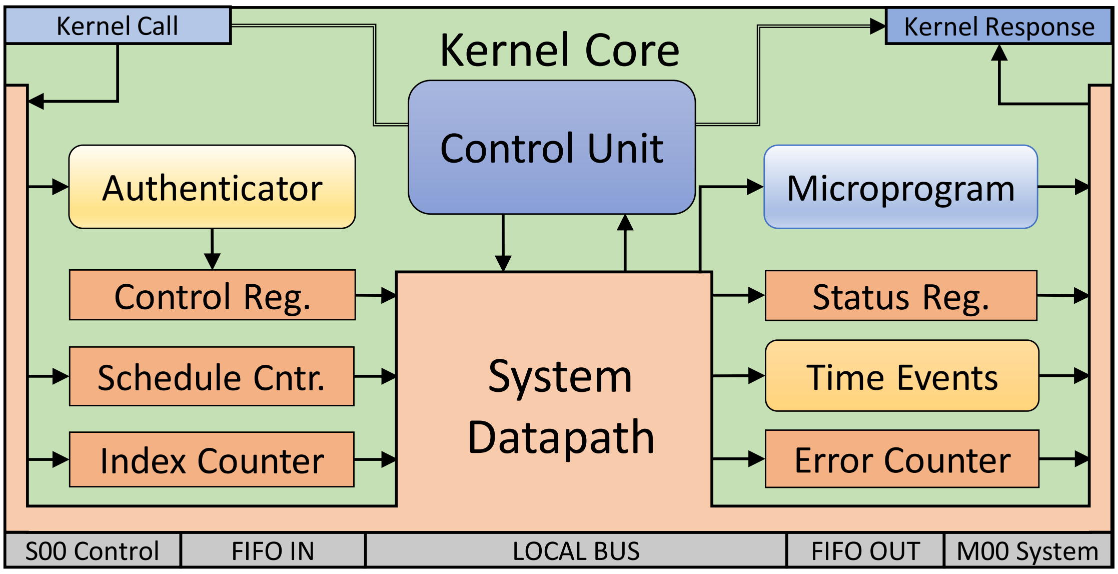


Figure 3.4: Kernel Core internal structure

Once active, the Control unit operates through the system-level datapath, establishing connectivity between the micro-programmable unit and the kernel Call and Response interfaces. These interfaces match directly to the *S00_Task* and *M00_Kernel* signals in the accelerator top-level, and allow the *HW-Task* to trigger the system calls present in the micro-programmable unit. In turn, system calls give rise to a pre-programmed set of control actions, which operate at the system level to handle adequate data manipulation using the existing local resources.

When the *HW-Task* design needs a wait event within the duration of a predetermined number of clock cycles, or to wait for an HW signal restricted to a maximum timeout interval, a system call can interact with the Time Event device to provide this service. In addition, three parametrizable counters are included in the kernel datapath and used internally when scheduling multiple and concurrent system calls, manipulating data through indexes and to account for any errors that may occur as result of the execution.

Once running, the micro-programmable unit suspends the clock signal at strategic points of the *HW-Task* design for all system calls. In doing so, the *HW-Task* context remains suspended while it interacts at kernel level. Pre-programmed control signals are then generated to forward the received parameters using the system-level datapath. At the same time, status information is generated that indicates whether the system call performs a write or read operation, or if it stays blocked waiting for available resources involved. In the final active clock cycle that completes the system call execution, the microprogram re-establishes the context on the *HW-Task*, allowing this entity to proceed with its processing.

3.1.1 Authentication

In the *Kernel Core* design, the Authenticator acts as a shield for the control register. All data write transactions that target this register are forward via the Authenticator device that, after validating the received word, determines whether or not such contents can be written. In Figure 3.5 we can see a simplified diagram that describes the internal design of the Authenticator device.

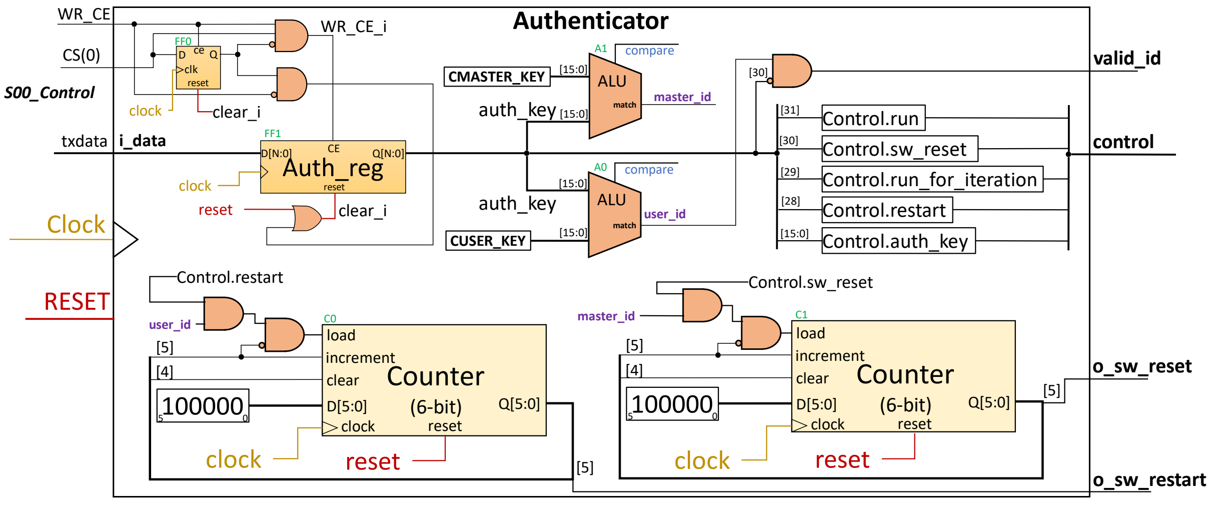


Figure 3.5: Authenticator device architecture

On the left side of the figure, we see the *S00_Control* interface, which is the source of control-oriented data, used by the host system. In this interface, the *CS(0)* line is active when the address matches the control register offset, and the interface specifies a write transaction using the *WR_CE* signal. The logical combination of both signals and logic '0' in the FF0 register, enables the clock (CE) in FF1 for one period. The next active clock transition stores the contents of *i_data* in the internal register *Auth_reg*.

During this clock cycle, the Authenticator makes the word available at the *control* output, while comparing the *auth_key* field with parameter keys in A0 and A1 inputs. The logical combination of *user_id* and negated *control.sw_reset* will activate the output *valid_id*, while at the same time, the input *WR_CE* assumes the logical zero, which activates the *clear_i* internal signal. The subsequent clock cycle will use the *valid_id* signal to store the control output in the *Kernel Core* register, while the clear signal discards the contents in the *Auth_reg*. Figure 3.6 shows a wave plot that results from the functional validation of the authenticator device, where the *S00_Control* interface writes in the control register of this unit.

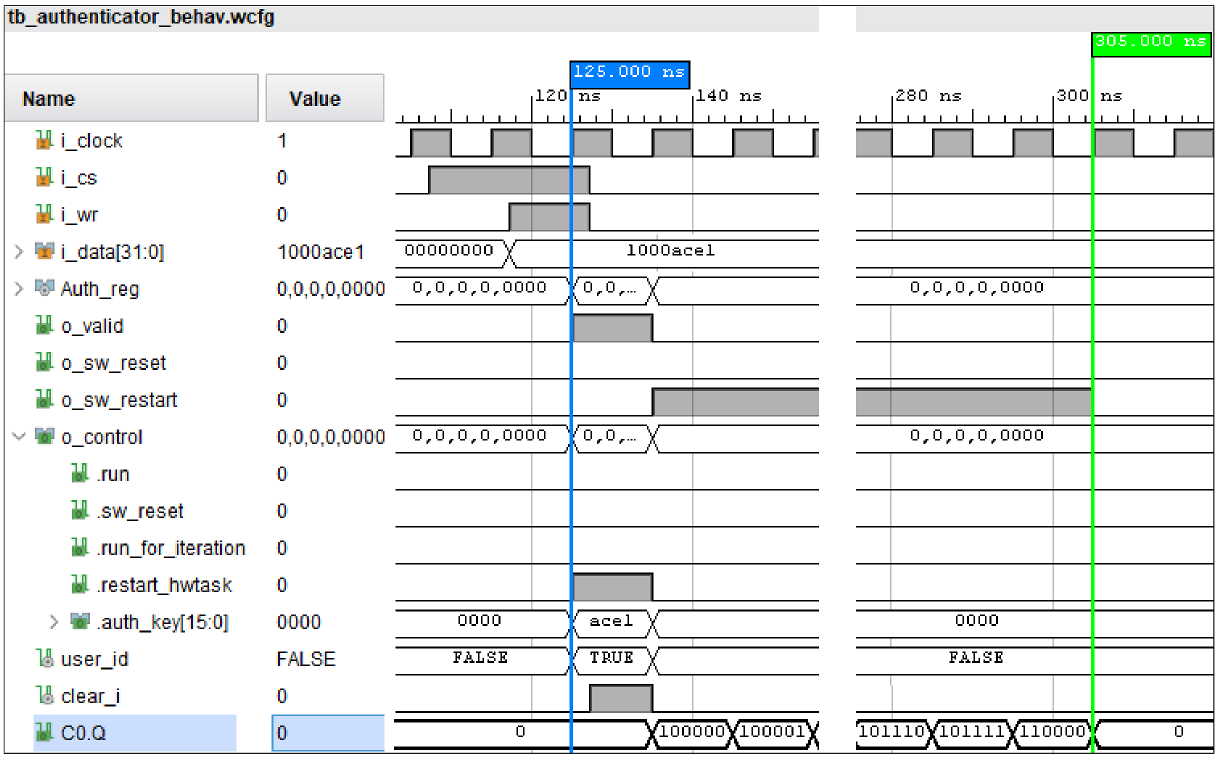


Figure 3.6: Authenticator wave diagram: control operation for *HW-Task* restart.

At the 125 nanoseconds marker, the control word 0x1000ace1 sets the *restart_hwtask* bit, that combined with the *user_id flag*, loads the counter C0. In the next clock cycle, the active 5th bit in Q[5:0] links with the output *o_sw_restart*, and the same bit starts C0 counting. When the count value reaches "110000", the 4th bit clears C0, and at the 305 nanoseconds marker the output Q assumes the value of "000000" that completes the restart operation. The authentication key *CUSER_KEY* gives the host privileges to change all fields in the control register except for *sw_reset* bit. Due to such critical condition and to avoid erroneous data re-use, this operation can only be triggered by the use of distinct *CMASTER_KEY*. In a similar way, when the authenticator receives the *CMASTER_KEY* with the logical value high in the *sw_reset* bit, the combination of this bit with the *master_id* flag loads C1. The output of this counter triggers the SW demanded reset operation for a pre-determined number of clock cycles. The value of both keys, as well as the count intervals of the C0 and C1, are parameters of the Authenticator. These values are defined in the *hal_asos_configs* package that can be adjusted to better suit the design timing constraints. Some of the contents in this package will be discussed in this section.

3.1.2 Control Unit

The Control Unit is responsible for the status of the *Kernel Core*, and reacts to the active bits in the control register and the task status bits. From the host's point of view, this unit establishes synchronism between the OS and the accelerator model. As from the accelerator's point of view, it synchronizes the *HW-Task* with the target application. Figure 3.7 1 describes the top-level signals of this unit, and it also shows the control register implementation and the *HW-Task* status bits dead and done (i.e., *task_dead_i* and *task_done_i*).

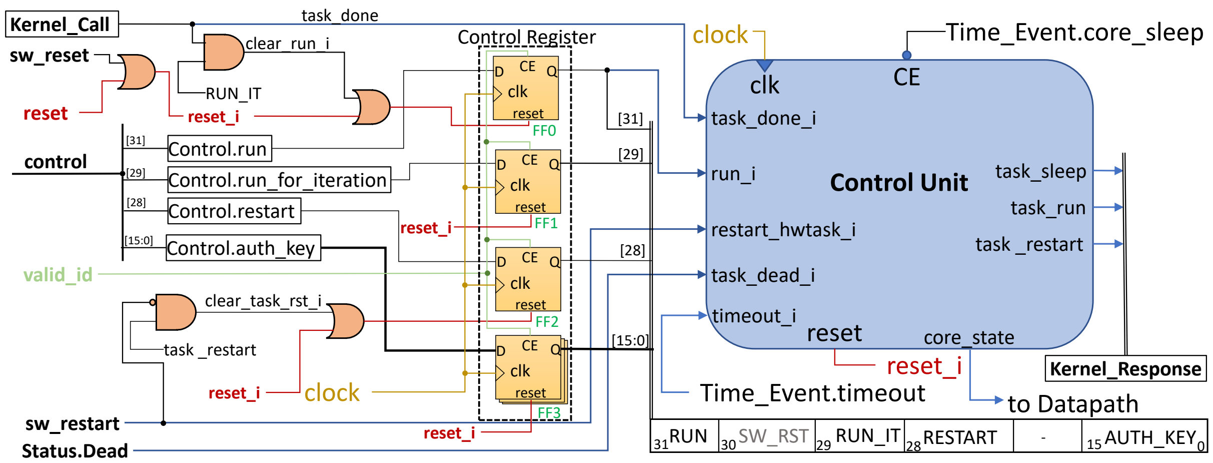


Figure 3.7: Kernel Core - Control Unit Overview

The main source of information for this unit is the control register implemented using FF0 to FF3 logical elements (LEs). When the *RUN* bit is active, the control enables the accelerator to a continuous run, that can be quantifiable in *HW-Task* iterations or processing rounds. When this bit value switches to logical zero, the control remains active until the *HW-Task* completes the current round, and signals completion through the *task_done* bit. Similarly, the combination between *RUN* and *RUN_IT* bits, activates the control unit to run for one *HW-Task* round. At completion, the *task_done* signal is used to set *clear_run_i* and the next active clock transition will clear the RUN bit (FF0) in the control register. In response, the control suspends the *task_run* output, which prevents the *HW-Task* from starting a new round. The *HW-Task* rounds are comprised between the clock cycle in which the control activates the *task_run* output until the clock cycle in which it receives the *task_done* signal.

The implementation of the control unit follows a state logic that can be consulted in Figure 3.8. States #0, #1 and #2 are referred to as online states since control is available to interact with the host system, although from the *HW-Task* perspective, in #2, the internal context is suspended. In the remaining states, the accelerator is considered offline, either because it is in the dead state or because a restart operation has been triggered and it waits in #3 for the *HW-Task* to be ready for execution.

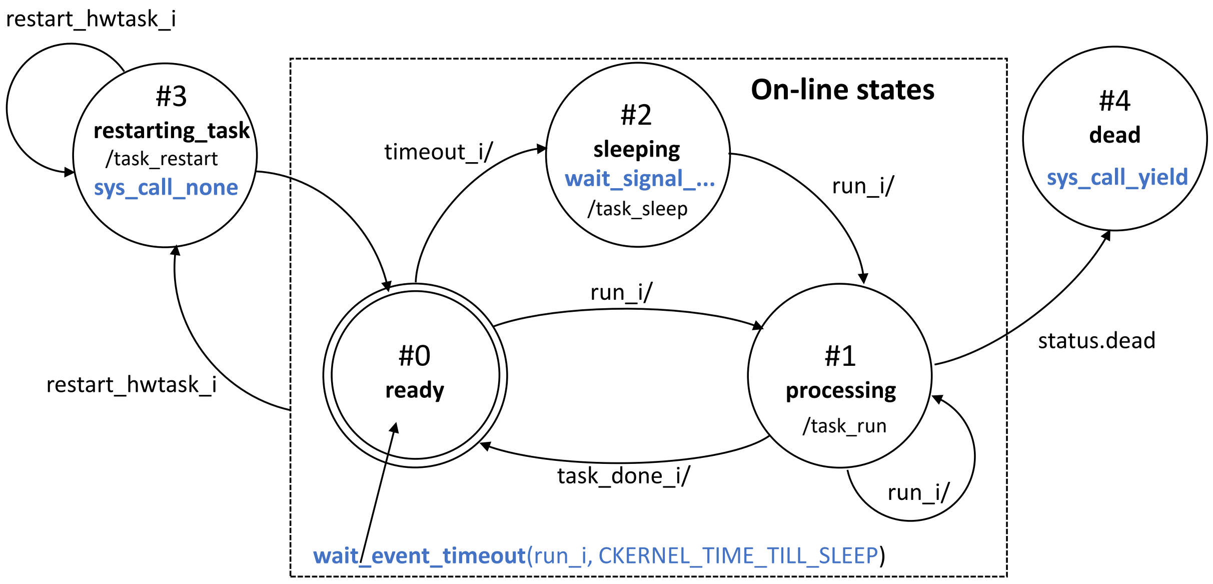


Figure 3.8: Kernel Core - FSM state diagram

The Control unit assumes the initial state *ready* after a system restart or a host demanded SW reset. During this state, it is not possible to predict when the host system will be available to interact with the accelerator, and thus the control initiates the HW system call *waint_event_timeout*. This system call receives the *run_i* signal and the integer *CKERNEL_TIME_TILL_SLEEP* as input parameters. The numeric value expresses the timeout value in clock cycles during which the execution will remain waiting for a high value on the *run_i* signal. If the *run_i* input is not received before the counting expires, the control returns from executing the system call with the timeout bit high, and transitions to #2.

In #2, the Control unit activates the *task_sleep* output to enable the sleep condition on the *HW-Task* design. At the same time, it selects a similar system call but now using the logical combination of the *run_i* or *restart_hwtask_i* as input signal and no timeout value. In doing so, this system call will wait indefinitely for a high value in the input signal parameter. A zero value in the *timeout* parameter activates the *core_sleep* signal that is output in the Time Event unit, and using this signal, strategic points of the *Kernel Core* are kept with the clock signals disabled. If the input signal is detected, the Time Event unit disables the sleep signal and the Control unit transitions to the next state accordingly. Slightly modified control unit is used to address microprogram updates and can be seen attached in Figure D.6.

Once in #1, the Control unit connects the system-level datapath and the micro-programmable unit to the kernel Call and Response interfaces, so that it can answer to *HW-Task* requests. At the same time, the *task_run* output is kept active, signaling the *HW-Task* to proceed with its execution. The *HW-Task* can now execute any of the system calls that the Kernel makes available, until the host system decides to disable the run bit. Alternatively, the *HW-Task* can terminate the execution using the *task_yield* system call. This will activate the Dead flag in the Status register and the Control unit reacts with the transition to #4. Once in #4, the clocks for all strategic points in the *Kernel Core* and *HW-Task* designs are once again deactivated. The Control unit does not return from this state unless a software reset operation is initiated by the host system, or the target platform is restarted.

In Figure 3.9 we can see a wave diagram of a short simulation of this Control unit, including relevant signals that relate to the *Kernel Core* design. After an initial reset the control assumes the *st0_ready* and remains in this state, while waiting for the host to intervene in the control register. At time 1,300.00 nanoseconds, the status register indicates that the *HW-Task* is ready but blocked waiting for the run signal. From the contents of the *sys_call_id* signal, we can see that the microprogram is executing a wait event system call, which a few clock cycles later, gives rise to a timeout signal in the Time Event unit. In response to a timeout, the Control unit advances to the *st2_sleep* state, suspending the clock signal in the *HW-Task* design. Simultaneously, it repeats the execution of the system call, using the logical combination of the run and restart signals, and zero timeout clock cycles as parameters. Five clock cycles later, the Time Event unit activates the *core_sleep* signal that suspends the clock sources in strategic points the *Kernel Core* design. Under this condition, the Control unit enters into suspension mode, where it remains until host system intervention.

At time 1,630.00 nanoseconds, the control register is updated with the user key and the active bits run and restart. One of these two active signals would be enough to trigger the Time Event device, which in response, disables the *core_sleep* signal. With the clock sources enabled, the Control unit recognizes the flags that gave rise to the event and advances to *st4_restart*, where the context of the *HW-Task* is restarted. After ten clock cycles, the restart concludes and the control switches to the *st1_processing* and activates the *task_run* signal.

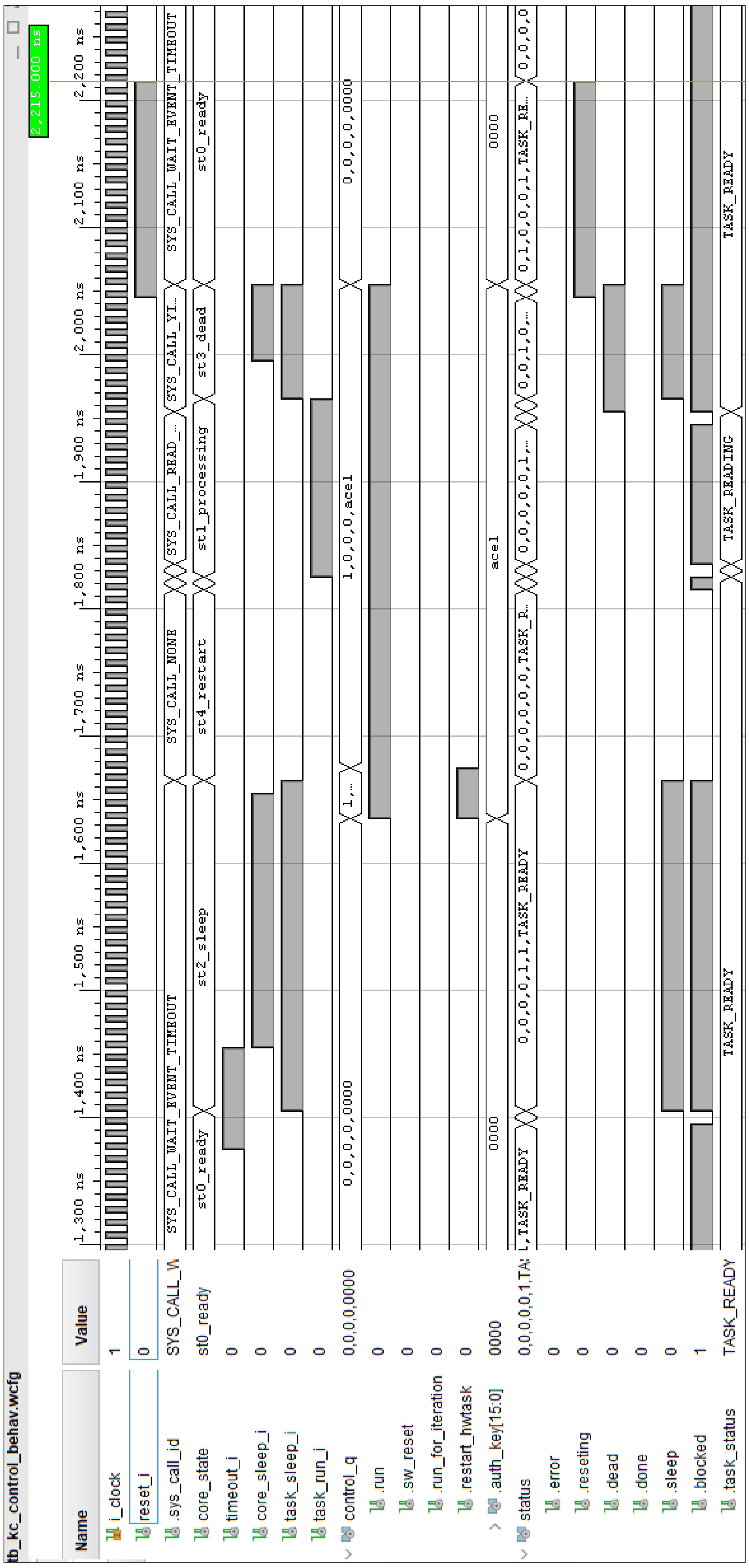


Figure 3.9: Kernel Core - control unit wave diagram.

Once active, the *HW-Task* executes the system call that reads the control-oriented data FIFO, and when completed, it decides to abort the execution by issuing the *task_yield* system call. The control switches the *st3_dead* where it remains blocked with the clock signals suspended once again. A few cycles later, the main reset signal restores the accelerator to its initial state. During these states, the status register is updated with the information from the *HW-Task*, where its operating status is visible, as well as the indication of blocking, dead, sleep and restart, by the use of the corresponding flags.

In Figure 3.10 we can see a functional block diagram depicts the connectivity between the kernel Call and Response interfaces, the micro-programmable unit and the system-level datapath. In M0, the source of parameters for the microprogram and the system-level datapath are determined by the state register of the Control unit. Similarly, using M1, the kernel response interface will receive the outputs from the system

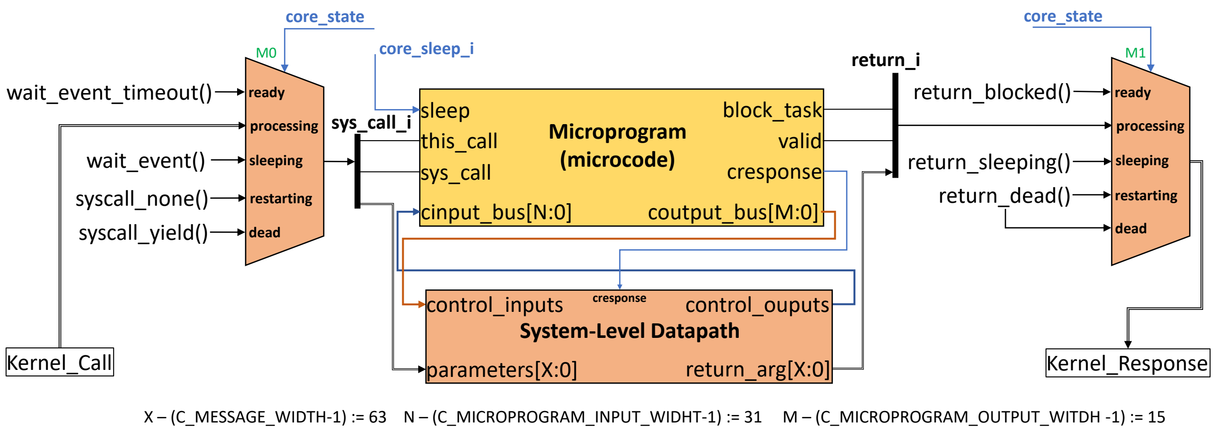


Figure 3.10: Kernel Core - system-level datapath and microprogram interaction.

call execution, or otherwise it will be clamped to constant signals. In the *processing* state, the *HW-Task* is the source and destination of these signals, whereas in the remaining states, M0 disconnects the *HW-Task* from the microprogram and system-level datapath, and M1 reinforces the blocking signals. Details about the microprogram and the system-level datapath will be discussed in the following subsections, after the introduction of the HW system calls.

3.1.3 Hardware System Calls

An HW system call is a sequence of control operations that use a predetermined number of steps, to provide services related to the local resources in the accelerator model. In similarity with the concept applied to the OS environments, the HW system calls virtualize the accelerator through a specific set of features, that allows the designer to structure an *HW-Task*. They are the *Kernel Core* fundamental interface, to handle the local resources and abstract away the complexity that the accelerator model represents. Such level of abstraction, in turn, promotes the design reuse allowing it to be implemented on different platforms, as long as the HW system calls have appropriate implementation. In doing so, we organize the *Kernel Core* design trough an incremental set of programmable features. For simplicity, in this section we will refer to the HW system calls as system calls and to the *Kernel Core* as kernel.

To implement a system call, the *HW-Task* uses procedures in the kernel package that establish the functionality, the involved parameters and the connectivity between these and the kernel microprogram and system-level datapath units. For this, the kernel provides entry and exit points in its interface that establish the required signals. In Figure 3.11, we can see an excerpt of the kernel package that defines the system call interface. Starting at lines 163, 206 and 213, the kernel package defines the *sys_call_t* type as the subset of system calls that the kernel supports and are used in the input and output records to establish the system call interface.

```
    ...
  6 library hal_asos_v4_00_a;
  7 use hal_asos_v4_00_a.hal_asos_configs_pkg.all;
  8 use hal_asos_v4_00_a.hal_asos_utils_pkg.all;
    ...
163 type sys_call_t is (SYS_CALL_NONE,  SYS_CALL_WAIT_EVENT_TIMEOUT, SYS_CALL_READ_LFIFO,
164   SYS_CALL_WRITE_LFIFO, SYS_CALL_READ_MESSAGE,SYS_CALL_WRITE_MESSAGE, SYS_CALL_READ_LBUS,
165   SYS_CALL_WRITE_LBUS, SYS_CALL_MUTEX_LOCK, SYS_CALL_MUTEX_TRY_LOCK, SYS_CALL_MUTEX_UNLOCK,
166   SYS_CALL_READ_MBUS, SYS_CALL_WRITE_MBUS, SYS_CALL_READ_LBUS_BURST,
167   SYS_CALL_WRITE_LBUS_BURST, SYS_CALL_READ_MBUS_BURST, SYS_CALL_WRITE_MBUS_BURST,
168 SYS_CALL_YIELD);
    ...
206 type sys_call_input_t is
207 record
208   this_call: std_ulogic;-- trigger sys_call
209   sys_call_id :sys_call_t;
210   parameters : std_logic_vector(C_MESSAGE_WIDTH-1 downto 0); --field for syscall parameters
211 end record;
212
213 type sys_call_output_t is
214 record
215   valid: std_logic;
216   block_task: std_logic;
217   sys_call_id : sys_call_t;
218   return_arg : std_logic_vector (C_MESSAGE_WIDTH-1 downto 0); -- return sys_call data
219 end record;
    ...
```

Figure 3.11: *HW-Kernel* package - system call types, entry and exit records

When executing system calls, each procedure specifies its arguments according to the desired feature in line 209, and links them to the input *parameters* according to line 210. It then activates the *this_call* flag to signal the kernel for valid inputs and to proceed with the system call. In response, the microprogram activates the *block_task* signal and transfers the received type of system call to the *syscall_id* member according to line 216 and line 217, respectively. Over the course of execution, the kernel updates the *return_arg* output (line 218) with the processing results from the system-level datapath. In the last step of the system call execution, the microprogram activates the signal on line 215, which indicates valid parameters in the return register, and at completion, it disables the *block_task* output to release the *HW-Task* control. The output fields hold their contents until the next system call execution, thus allowing the *HW-Task* to re-use or test them to evaluate results.

It should be noted that the kernel HDL package is inserted in a hierarchy that starts in the tool's configuration package. This establishes, among others, the length of the system-level datapath that is determined by the largest parameter it receives (lines 210 and 218). This parameter is the kernel-level control message and it depends on the parameter target architecture of the host system. As result, the length of the datapath is fixed on two words when the tool targets a 32-bit host, or three words for a 64-bit host. The kernel control messages will be discussed in subsection 3.2.3.

Table 3.1 summarizes the features that the kernel provides through the system calls. The first column lists the specific types that identifies each feature, the second column provides a short semantic description about the service and the third column shows the number of steps required to complete the system call. The first item in the table represents the idle operation for the states of inactivity in the microprogram. Taking the second line as an example, the procedure that implements the system call can receive parameters that include an input signal, and a numeric integer specifying a timeout in clock cycles. In this case, the duration of the system call will vary according to the HW context that has been associated with, but the implementation is based on three valid steps. Here, the first step is used to configure the Time Event device according to the received parameters. In the second step the microprogram waits for signal or timeout events, and in the last valid step, the system call completes by returning the remaining timeout clocks.

In lines three to six, the system calls read and write control-oriented data, to and from the HW-FIFOs, using two valid steps. The first step establishes the control signals in case of read, or control signals and output data in case of write, and waits on available data or space accordingly. In the second step, the system calls handshake with the device and collect received data in case of a read. Similarly, to read or write through the Local-BUS, it requires a fixed two-step implementation that uses one clock cycle in each step. In the first step, the parameters are submitted to the bus and these include the target address and the word to be transferred when the operation is a write. In the second step, the system call performs the handshake operation and collects data when the operation is a read.

Table 3.1: Kernel Core - HW system call summary description

System Call type	**Description**	**Steps**
SYS_CALL_NONE	No operation - Idle system.	-
SYS_CALL_WAIT_EVENT_TIMEOUT	Wait for a hardware event during a parameter number of system clocks.	3
SYS_CALL_READ_LFIFO	Read control data from the Local FIFO to the *HW-Task* control input.	3
SYS_CALL_WRITE_LFIFO	Write from the *HW-Task* control data output to the Local FIFO.	3
SYS_CALL_READ_MESSAGE	Read from the Message Queue to the parameter message.	3
SYS_CALL_WRITE_MESSAGE	Write the parameter message to the Message Queue.	3
SYS_CALL_READ_LBUS	Read one word using the Local-BUS at parameter offset.	2
SYS_CALL_WRITE_LBUS	Write using the Local-BUS at parameter offset and byte length in one word.	2
SYS_CALL_MUTEX_LOCK	HW-Mutex Lock using the offset parameter.	4
SYS_CALL_MUTEX_TRY_LOCK	HW-Mutex try-lock using the offset parameter.	4
SYS_CALL_MUTEX_UNLOCK	HW-Mutex unlock using the offset parameter.	4
SYS_CALL_READ_MBUS	Read one word from the system memory at offset parameter.	4
SYS_CALL_WRITE_MBUS	Write to system memory at offset parameter and byte length in one word.	4
SYS_CALL_READ_LBUS_BURST	Read using the Local-BUS at offset and byte length parameters in burst format.	4
SYS_CALL_WRITE_LBUS_BURST	Write using the Local-BUS at offset and byte length parameters in burst format.	4
SYS_CALL_READ_MBUS_BURST	Read the system memory at offset parameter for burst length words.	4
SYS_CALL_WRITE_MBUS_BURST	Write the system memory at offset parameter for burst length words.	4
SYS_CALL_YIELD	Kill the accelerator.	2

In the HW-Mutex lock, step zero evaluates the state of the resource and implements containment when locked. Beyond this, step one acquires the resource and step two evaluates the final result of the operation. If the locked by channel B flag is set, the final step releases the *HW-Task*, or otherwise, the concurrent race for resource is lost and the microprogram re-attempts the system call until it achieves success. Algorithm 1 describes the implementation of this system call.

In step zero (i.e., Step0), the microprogram generates the *block_task* signal that stops the control unit of the *HW-Task* until completion. At the same time, it generates the Local-BUS read signal to query the status of the HW-Mutex. The system-level datapath links a received address for the HW-Mutex with the Local-BUS, and uses the received data to produce flags Locked by A and Locked by B at the inputs of the microprogram. If the Locked by A flag is set, the microprogram will remain in this step until the status is updated and the flag is cleared. On release, it proceeds to step one writing the resource with the task ID using the Local-BUS. Here, no testing is required and the microprogram reinforces correct behavior using a dummy true test. In doing so, the two possible flows of execution, through the true or false path both lead to step two. In step two the algorithm tests the Locked by flag for success in acquiring the resource. In the occurrence of failure, the microprogram will return to step zero to repeat the system call. On success, it will proceed to step three where the execution completes by transferring the received status to the *return_arg* register, activating the valid signal and releasing the *HW-Task*.

Algorithm 1 Microprogram to lock an HW-Mutex

```
1: pseudocode SYS_CALL_MUTEX_LOCK
2: Step0: produce block_task and lbus_rd_ce test mutex status Locked A flag.
3:     if true then goto step 0.
4: Step1: produce block_task and lbus_wr_ce test true input.
5:     if false then goto step 2.
6: Step2: produce block_task and lbus_rd_ce test mutex status Locked B flag.
7:     if false then goto step 0.
8: Step3: produce valid
9:     exit
```

To avoid containment for undetermined number of clock cycles, the procedures that deal with the *HW-Mutexes* distinct modes of operation between lock and try-lock. In try-lock mode, the system call can fail on step zero as result of locked resource. In this case, it proceeds to step three avoiding contention and concluding by returning a false *boolean* parameter to the *HW-Task*. Otherwise, if the resource is free the system call proceeds to step one, where it attempts the lock operation. In step two, in the occurrence of a lost race for the resource, the system call proceeds to step three and a false return is established. Otherwise, if the resource is locked the system call concludes with true as return parameter.

Reading or writing to the system bus is scheduled in four steps to allow the microprogram to handle distinct platform or bus technologies. Algorithm 2 describes this system call implementation. In step zero the system call tests if the kernel owns a valid memory address and if false, it proceeds to step three and completes the execution with error. Once in step one, the system call sends the write transaction to the M00 interface, providing parameters and activating a request signal. It expects to receive an acknowledgment signal (i.e., *CmdAck*) that can occur after an undetermined number of clock cycles and, in the event of an error, a *CmdErr* signal will also be received. Such error indicates an abnormal transaction handshake and can occur if for instance, the system bus does not accept the specified operation at the provided address, or if the M00 interface is busy with a transaction in progress. In step two, the system call waits for the completion signal (i.e., *Cmplt*) that will always be generated by the M00 interface. During this step, a new error may be received if step one was completed successfully and it predicts a failure during the write operation performed by the host system bus. Upon receiving the *Cmplt* signal, the system call proceeds to step three to finalize the system call and release the *HW-Task*. The *CmdErr* signal is handled at the system-level datapath where it is connected to the error flag input register.

Algorithm 2 Microprogram to write one word to the system bus

```
1: pseudocode SYS_CALL_READ_MBUS
2: Step0: produce block_task and test sysram address
3:     if false then goto step 3
4: Step1: produce block_task and mbus_wr_req and test CmdAck.
5:     if false then goto step 1.
6: Step2: produce block_task and test Cmplt.
7:     if false then goto step 2.
8: Step3: produce valid
9:     exit
```

The use of a similar system call for the burst transfer format requires in step zero to test the active burst mode flag. Such flag is active when there is a valid system memory address in the kernel register and there is burst capability on the interface. It then proceeds to step one or aborts the execution, by proceeding to step three and completing with error. Step one is executed as previously while the step two is repeated for each transferred word, until a burst done signal is set high. Here, the *Cmplt* signal is used at the kernel runtime level to index the next input word. Similarly, in step three the microprogram completes the execution by releasing the *HW-Task* context. The use of burst format system calls will be best discussed in subsection 3.1.7.

In the last system call, the *HW-Task* kills the accelerator putting the kernel control in the dead state. This system call is executed in one step and is generally used in case of task control errors, or to implement a pre-programmed application shutdown. In each case, the procedure used by the *HW-Task* is a user defined procedure that combines two operations to send a kernel control message before triggering the dead signal. Such message is used at host level for notifying the application, and can be used to request shutdown, or otherwise restart the accelerator to handle errors. The *HW-Task* message protocol and related package procedures will be discussed in section 3.2.

3.1.4 Microprogram

To implement the design of the micro-programmable unit, the accelerator model employs single address microcode as design basis. Its operation is based on the flow of microinstructions that implement the microprogram, where each *opcode* activates certain outputs, and selects one input for testing. As result, a program counter advances into the next instruction or takes a jump based on the current address and an implicit offset in the *opcode*. Depending on the microcode technique, the address of the next instruction may also be defined in the *opcode*, for the two possible logical cases. This is the instruction format when using a two-address microcode. Alternatively, using an *N*-bit adder logic element, the address of the next instruction can result from the increment of one unit of the current address value. For the accelerator model, an integer parameter of 5 in the settings package (i.e., *CSYS_CALL_WIDTH*) is applied to *N* to establish the program counter address range. In this way, 2^5 four-step system calls or a 128-byte address space is in use for the chosen microcode, and a 2-bit incremental counter with load input is used instead of the traditional adder.

Figure 3.12 shows the *opcode* format for step two of the system call to lock a HW-Mutex. The value of the program counter results from the address prefix corresponding to the system call, concatenated with the output register from the step counter. In this example, the absolute address 0x22 is applied to the ROM where the microprogram is defined. The resulting word determines that input 10 is used as a test source, 0x20 is the address of the next microinstruction in case of failure, and output 7 remains active as long as the current microinstruction is valid. In the same figure, it is also possible to observe the value of the outputs valid (V) and block task (B) which are transversal to all microinstructions, and for this reason they are located at specific positions in the *opcode*.

The implicit offset in the *opcode* is used when the test result is false and for this reason, this field is called 'next step when false' (NSF). Traditionally, it refers the next state of the microprogram, but the concept has been adapted due to the hierarchy of states that the design implements. Such hierarchy begins in the processing state of the *Kernel Core*, providing the system call services to the *HW-Task*, which in turn translate to an execution state of the caller that each system call represents. Each state, in its turn, evolves in micro-states that implement the necessary steps to complete a system call. As for the choice of jump in case of test failure, we have adopted the concept of continuous execution in line with the program's normality. The opposite case could also be used but the emphasis would be reversed. Ultimately, the choice must be aligned with the predominant default value of the inputs.

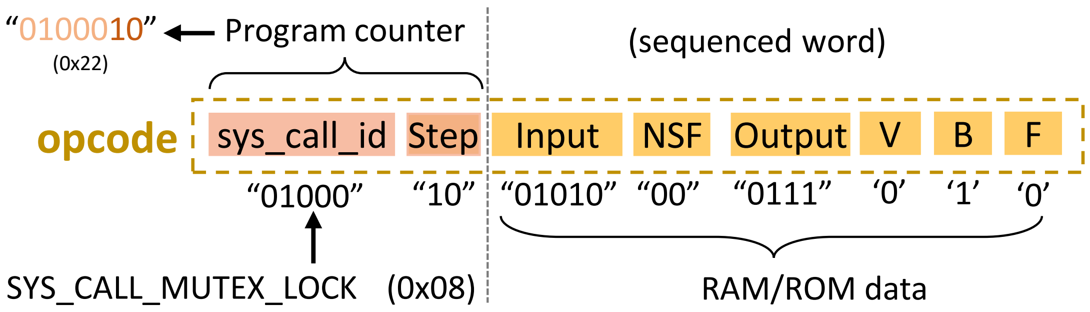


Figure 3.12: Microprogram - *opcode* format example in mutex lock step two.

To select a test input, the design of the microprogram uses a 5-bit field in the *opcode* to implement a multiplexing function of 32 signals to 1. Up until now, 24 of these are in use and can be consulted in the excerpt of HDL descriptions attached to this document, as shown in Listing C.16. In the same *opcode*, a 4-bit field allows the microprogram to activate outputs, by implementing a demultiplexer function from 1 to 16 signals. Listing C.17 attached to this document includes an excerpt from the HDL descriptions of this component, where it can be seen that so far 10 outputs are in use.

Table 3.2 shows an excerpt from the microprogram that includes the microinstructions of two system calls, the mutex lock and try-lock. The contents in this table are ordered according to the microinstruction *opcode* in Figure 3.12, and for completeness, the microprogram description can be found in Listing C.18 attached to this document. In the first line of the mutex lock system call, the microinstruction selects the input 12 for testing the Locked A flag. The microprogram should proceed to the next instruction only when the resource is free. Since this flag indicates a contradictory state, in order to implement a continuous flow of valid tests, it must be complemented before the mux input. In this way, when the Locked A flag is active, the input selection will result in test failure, and the microprogram will jump to the current instruction until the resource is released. On release, the true result increases the step counter, which will give rise to the absolute address 0x21 as result of the concatenation using 0x08 and "01". In this step, the microprogram activates the output 6 to write in the HW-Mutex and implements the dummy test to proceed to step two on any result. For this test, it selects input 31 which has the logic value '1' statically assigned to the multiplexer input.

In the microprogram inputs, only the locked flags A and B are used in complemented value, and we use the latter to test if the mutex has been released by the microprogram. As such, the same flag without the reverse logic is received at input 10, which gives rise to a valid locked test. Such test is used in step two of the lock system call to ensure success in the occurrence of a race condition for the resource. When

Table 3.2: Microprogram - Binary excerpt from the Microprogram.

	Sys Call ID	Step	Input	NSF	Output	Valid	Block
	...	...	...	...	...	...	...
SYS_CALL_MUTEX_LOCK	0x08	00	01100	00	0111	0	1
		01	11111	10	0110	0	1
		10	01010	00	0111	0	1
		11	11110	00	0000	1	0
SYS...MUTEX_TRY_LOCK	0x09	00	01100	11	0000	0	1
		01	11111	10	0110	0	1
		10	01010	11	0111	0	1
		11	11110	00	0000	1	0
	...	...	...	...	...	...	...

success, the microprogram reaches step three by incrementing the counter, or otherwise jumps to address 0x20 and repeats the system call. In step three, it activates the output to indicate valid data in the return register and releases the *HW-Task* by disabling the Block output. At completion, the microprogram needs to jump to step zero in the counter register, so that a new system call can start. Although the increment of the counter gives rise to the similar behavior, the design applies a dummy test of a false input to favor regularity, and jump back in the last step of each system call.

The following lines in the same table describe the implementation of the system call for the try-lock mode. As described in subsection 3.1.3, it differs from the previous one in steps zero and two, where in the first case it proceeds to the last step if the resource is locked, avoiding contention. In the step two, it proceeds to the next step even though it might have lost the dispute for the resource, thus avoiding the repetition of the system call. In this case, both test results lead to step three and a distinction is made at the system-level datapath, which turns the denied Locked B flag into an error and a false logical value for the system call return.

Although some system calls use fewer steps, a set of four microinstructions establish a state in the microprogram. In the fewer steps case, the unused locations were filled with just-in-case corrections, implementing jumps to the beginning of the system call where they are inserted. For this reason, only the blocking output is set to active, the false input is used for testing and the NSF field defines "00" as the next step. Up until now, 18 system calls are mapped in the program memory that requires 72 memory locations. For this purpose, we implement a 128 x 13-bit ROM with an overall usage of 56% of the capacity and exhibiting 12% of memory footprint with self-corrections.

Figure 3.13 describes the ROM-based microprogram architecture using a simplified block diagram where U0 represents the combinational ROM, addressed using the 7-bit program counter (PC). For this signal, the design uses the inputted *i_sys_call* as address prefix and to convert a system call element to an equivalent number, it employs the VHDL position attribute. In doing so, it considers the logical description in the signal by the enumeration value it represents. Then, we concatenate this number with the 2-bit output Q of the counter C0 to form the absolute address used in U0. In order to synchronize the microprogram execution with the system-level datapath, the absolute address is also forwarded to this unit using the output *control*.

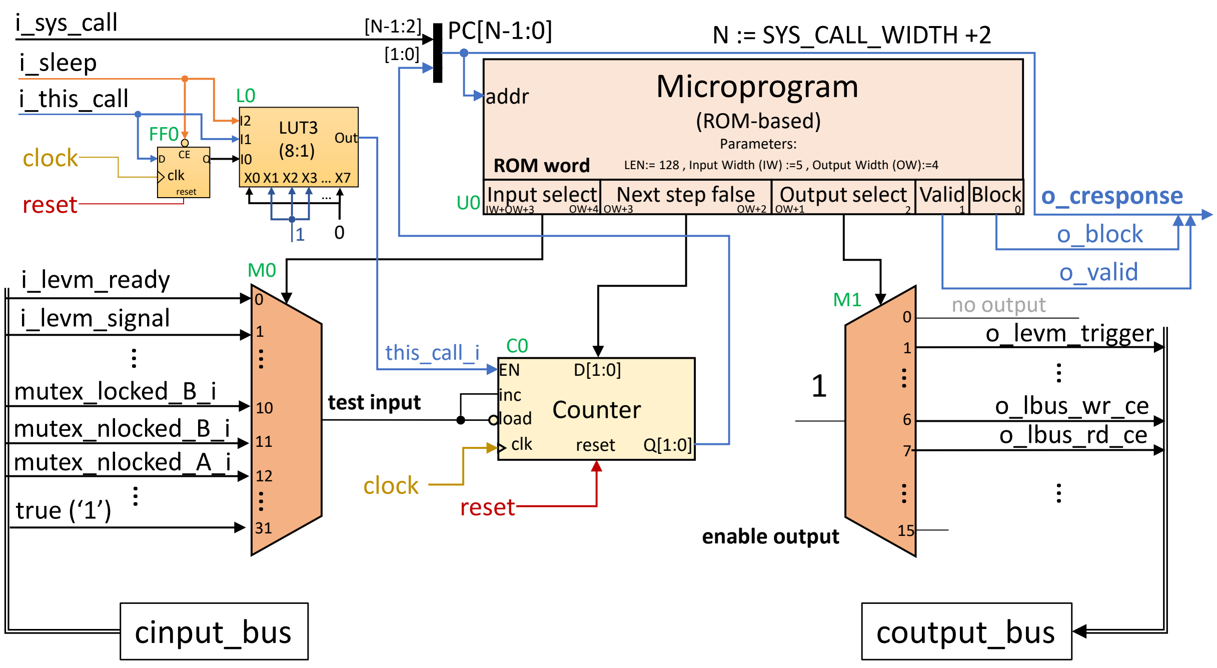


Figure 3.13: Microprogram - ROM-based internal architecture.

The C0 counter is updated in the clock active transitions by incrementing whenever the output in M0 is a logic '1', or loading the value in input D when the same is a logic '0'. For a change in its output, the counter must have the EN input active, which enables the clock signal in the output register Q. The EN signal is connected to the output of the logic expression in L0 that evaluates the *i_this_call* input, this signal in the previous cycle, and the sleep input. The combination of logic '0' at the input I2 and '1' at I1 or I0, establishes a logic '1' at the output. The use of FF0 provides an additional clock pulse in C0, ensuring that, on completion, the microprogram proceeds with step zero of the idle system call. To implement this, L0 receives an 8-bit logic function 'X' as parameter, that contains '1' in the positions 1, 2 and 3, and '0' in the remaining positions. This is an alternative representation of the logical expression after it has been implemented in the FPGA configurable logic blocks (CLBs). Figure C.18 attached to this document, shows an excerpt from the HDL descriptions of the microprogram unit that include the U0 ROM, the PC and the L0 logic. The sleep signal is activated by the Time Event unit operation during step one of the wait system call (line 1 of Table 3.1). In this step, the inputs *i_sys_call* and *i_this_call* cannot change since the *HW-Task* is already blocked, and by disabling the clock in C0 and FF0, the microprogram is considered suspended. Within a signal detection or timeout events, the sleep signal is revoked, allowing for the next active clock to update the C0 output. In doing so, the PC receives the absolute address for step two and the microprogram resumes the execution. Specific points in the accelerator model that use this hibernation signal include the *HW-Task* control, the kernel control and the sequential logic in the kernel's datapath.

The descriptions about each design unit will include the sleep feature. In the next subsection, we shall discuss the Time Event unit operation using a wave plot that includes the microprogram signals, while implementing the wait system call. Alternative RAM-based microprogram architecture uses true dual-port RAM in U0 to allow filesystem-based updates in the microcode contents can be consulted attached in Figure D.9. The binary code is loaded by the HAL-ASOS file system at start-up, and the storage resource is mapped by the S00 interface at supplementary page 8 offset.

3.1.5 Time Events

In the proposed accelerator model, the time wait and event detection features are implemented using the Time Event unit, which provides parametrized services using the wait system call (line 1 of Table 3.1). Through this unit, the kernel provides synchronization services to the control of the *HW-Task*, being based on internal events that it expects to receive. Such events may result from known and synchronous sources, or asynchronous change in the logical signals that this entity receives from its design. Ultimately, waiting for an asynchronous event has a degree of unpredictability, which can be complemented with a timeout parameter in clock cycles.

With this unit, the accelerator model seeks to enclose waiting events in a single resource to avoid consequent replication of counters. At the same time, it seeks to mitigate the energy waste associated with these waiting cycles, by providing a sleep feature that disables the clock at strategic points in the *HW-Task* and *Kernel Core* designs. In doing so, whenever the *HW-Task* enters a sleep state, the kernel will act in accordance, disabling the clock sources of the functional units in its datapath. On the other hand, if the accelerator is not running, the kernel control uses this service to disable the same functional units, while waiting for the host system to intervene in the control register.

Figure 3.14 describes the design of the Time Event unit through a simplified block diagram. The central element in the datapath is the C0 that implements clock cycle counting until it achieves a timeout value that receives on its input. For appropriate semantics, this timeout is converted using two's complement in A1, and in doing so, the count value starts from that result to zero.

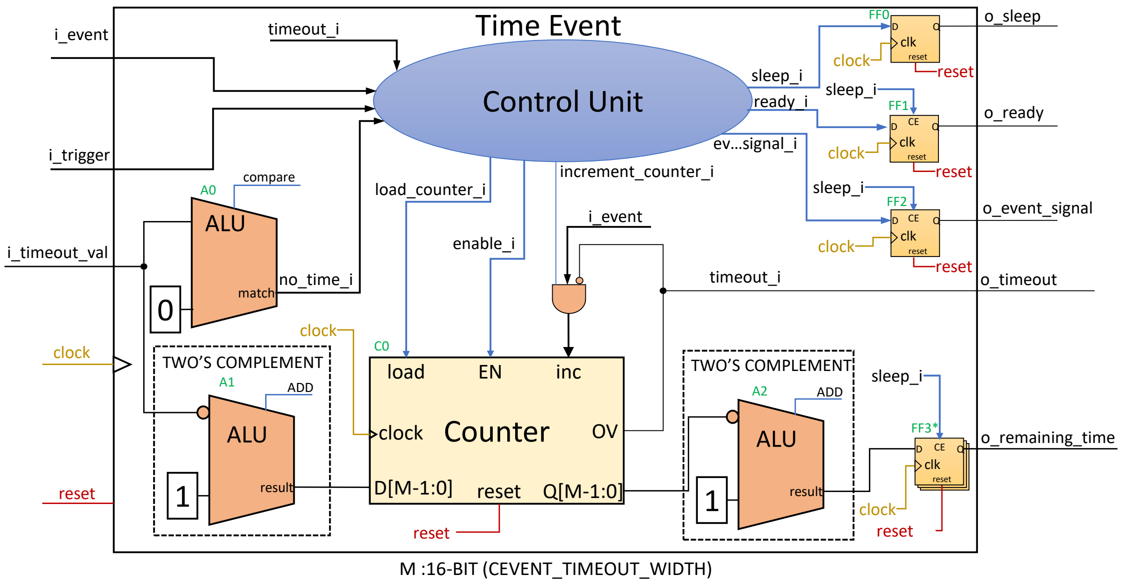


Figure 3.14: Time Event unit - simplified architecture diagram.

Once in service, counting can be interrupted by detecting the event input, or ultimately by exhausting the counter, which is signaled by the internal timeout. At completion, the remaining count is converted back in A2 and returned to the *HW-Task* using the remaining time output. The overall design of the datapath is equivalent to a down-counting circuit that can load absolute count intervals. The resource management is supervised by the control unit that loads, activates and stops the C0 counter. At the same time, the control is sensitive to the *event* and *trigger* inputs while establishing the *ready*, *sleep* and *event* signals on the outputs.

The *event* input is connected to the *HW-Task* sourced signal by means of the system-level datapath in the *Kernel Core*, and the *trigger* signal is activated by the microprogram to start counting. For this, the *ready* output is used, informing the microprogram about the resource state. If the timeout value is null at the input, the internal signal *no_time_i* is set to logical '1'. The control unit reacts by disabling the *enable_i* signal that suspends the C0 counter. At the same time, it will *enable* the sleep output and wait indefinitely for a signal rise at the input *event*. The *sleep* output is distributed by the accelerator design to disable clock sources, and internally, this signal is directly applied to the logic elements FF1, FF2 and FF3. Finally, the output that results from the event detection, is sent back to the microprogram so that it can complete the system call.

Figure 3.15 describes the control of the Time Event unit using a logic state diagram. After an initial reset signal the control logic reaches #0, where it activates the *ready* output and waits for a *trigger* signal to initiate a new event. Once at #1, the control enables the clock in C0 using the internal *enable* signal and together with *load*, enables the write of the timeout value received in its input.

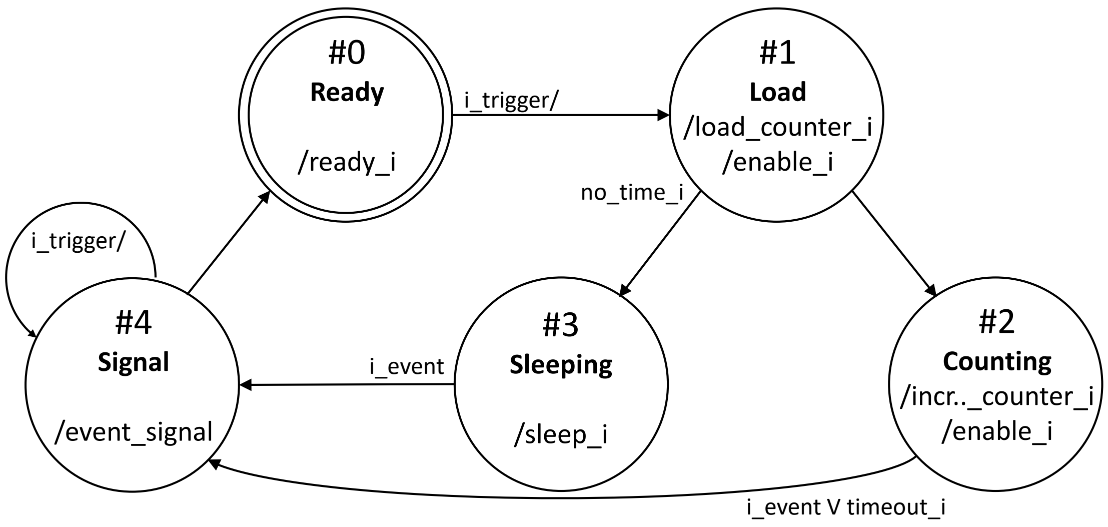


Figure 3.15: Time Event unit - control logic state diagram.

With a non-zero timeout, the control advances to #2, where it keeps the clock active and starts counting, by activating the *increment* signal. The reception of a logic '1' in the *event* input, or the occurrence of a timeout in the counter, completes the counting state and the control advances to #4. If the timeout value at the input is null in #1, the control advances to #3, where it waits indefinitely for the *event* input while keeping the *sleep* output active. With a logic '1' in the *event* input, the sleep state concludes and control advances to #4. Once in #4, the unit activates the *event* output and synchronizes the operating mode, waiting for the *trigger* input to be disabled. Upon disable, the control unit returns to #0 and signals that it is ready for a new processing round.

Figure 3.16 shows a wave plot that results from the simulation of the Time Event unit using Vivado, in the context of the accelerator design where it is implemented. For this purpose, the *HW-Task* implements a system call to wait for an event and considers a timeout of 6 clock cycles. At the bottom of this diagram, it can be seen the parameters in use, as well as the microprogram control signals. The system call starts at

10,505.00 nanoseconds and in step 0, the microprogram evaluates the operationality of the Time Event unit. A clock cycle later, the microprogram reaches step 1 and activates the trigger signal, staying in this step until event detection.

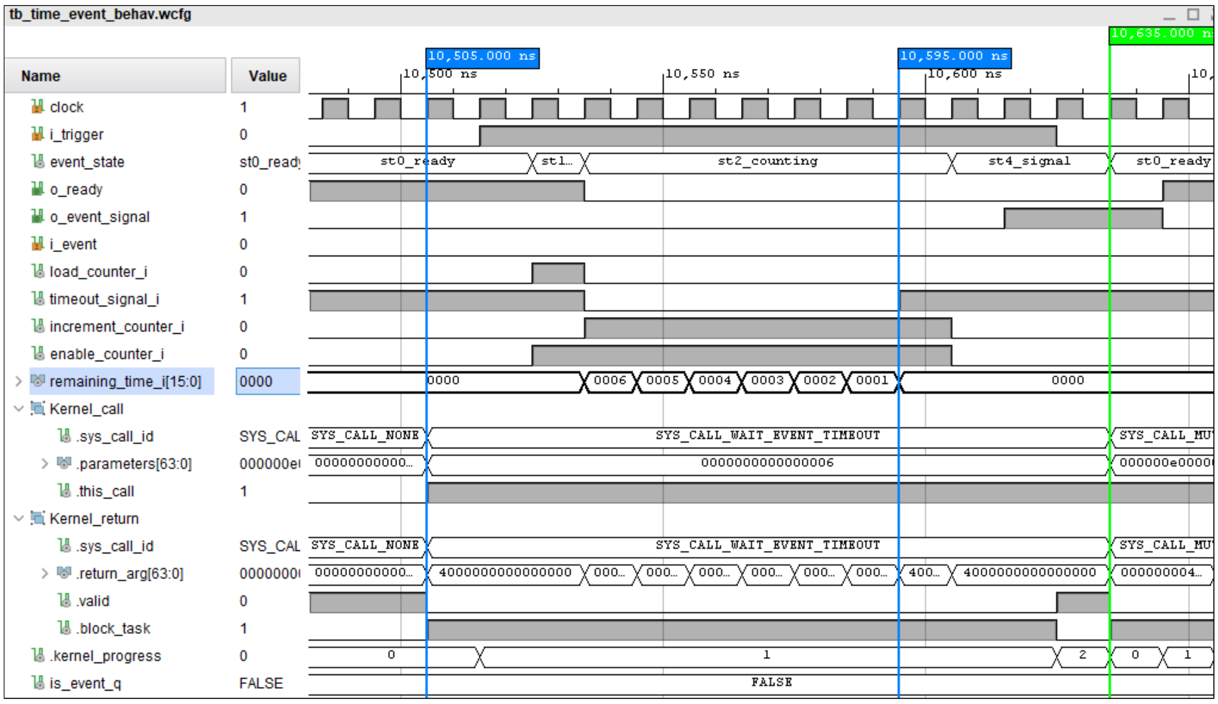


Figure 3.16: Time Event unit - Wait for a signal event using a 6-clock timeout.

The control of the Time Event unit responds to the trigger signal stimulus and advances to state 1, loading the counter with 6, and one clock cycle later, when state 2 is reached, it starts counting. After 6 clock cycles, counter time is exhausted and the internal timeout signal is asserted. The control unit advances to state 4 and signals the event detection through its output *o_event_signal*. Upon receiving this signal, the microprogram proceeds to step 2 and releases the context of the *HW-Task*. At the same time, the system-level datapath transfers the timeout result and the remaining count value to the return register. In the context of the *HW-Task*, the system call concludes with a false *boolean* extracted from the *return_arg* register at position 62 (PARAM_TIMEOUT_STATUS). And in the position range [15:0] (PARAM_TIMEOUT_VAL' range) this unit can check for a remaining time value. For completeness, similar plot using event detection can be consulted in Figure D.3 and Figure D.4 shows the same simulation using a zero-timeout parameter.

3.1.6 System-Level Datapath

The system-level datapath establishes correspondence between the system call interface, the microprogram and the resources in the accelerator model. For this, it selects one of three interfaces that provide access to local resources as it was shown in Figure 3.4. Among these interfaces we can find the HW-FIFOs input and output interfaces, the Local-BUS, and the M00-BUS. Alternatively, the system-level datapath can select the TimeEvent internal peripheral while providing timing services. The design is predominantly combinational and operates in conjunction with the microprogram, to exchange data between individual resources and the members of the system call. In Figure 3.17 we can see the system-level datapath architecture, described using a simplified diagram.

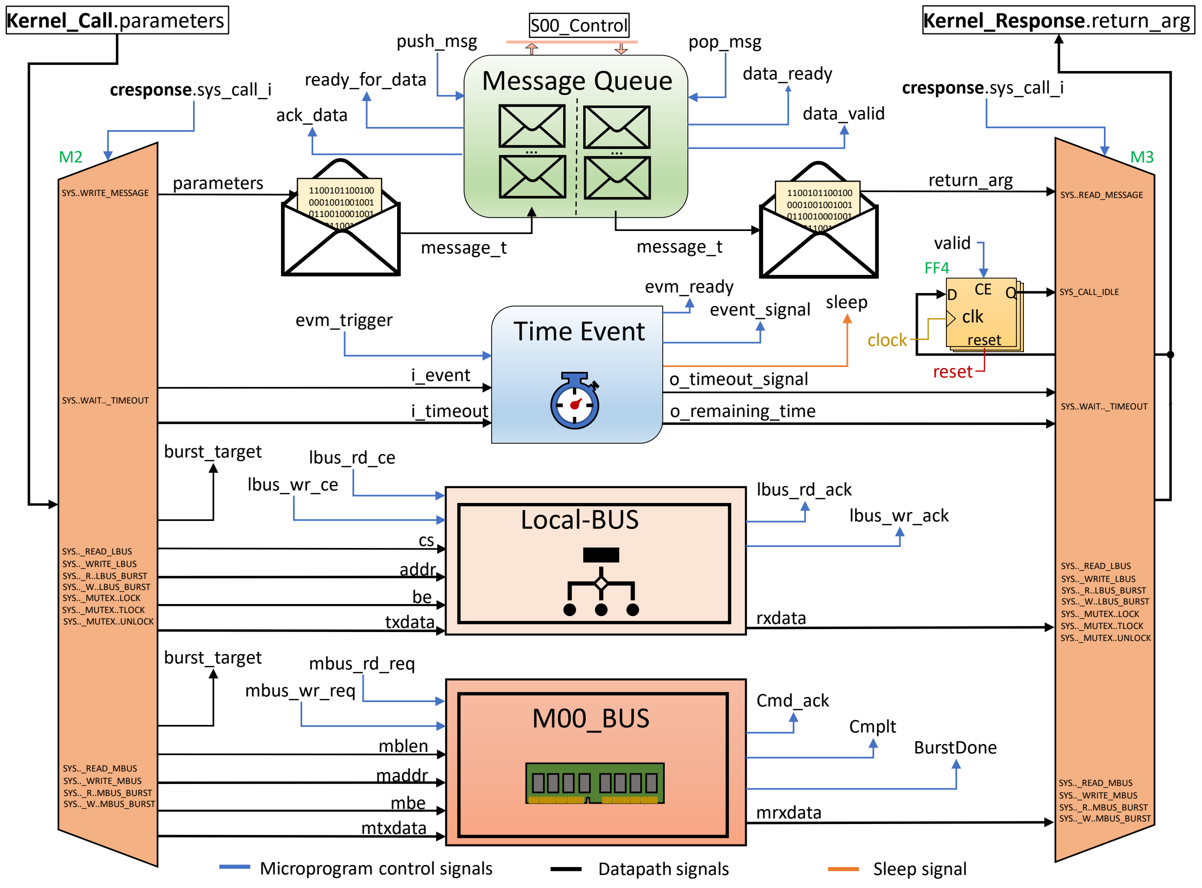


Figure 3.17: System-Level datapath - system call signals and target resource.

At the top of the figure, we can see the system call interface signals that connect with this design, after being selected by the kernel control state, as it was seen in Figure 3.10. Input parameters are received at the M2 demultiplexer, which operates according to the type it receives from the microprogram control output. As such, it extracts predetermined fields from a system call to connect with one of the four logical elements visible in the center of the figure. Conversely, the signals received from each logic element are inputs to the M3 multiplexer, which are selected by the same type as in M2 and concatenated to form the fields in the return of the system call. To handle clock cycle transactions, the return member fields are sent directly from M3 to the kernel output. At completion of each system call, these are stored in FF4, so that they are available beyond execution. For that, the microprogram must reach an idle state (i.e., SYS_CALL_IDLE), or otherwise a new system call will update these registers with new data.

The interface that the kernel provides with the HW-FIFOs, is used to submit or retrieve control words, that the accelerator model distinguishes between committed to kernel or to the *HW-Task*. In these components, the design is aligned with the transfer rate of a clock cycle and takes advantage of parallelism to exchange data above the word length. Figure 3.17 shows a Message Queue used to exchange kernel messages, which is implemented using two HW-FIFOs. In this logical element, the length of the data is specified by the predefined type *message_t*, which in the standard configuration is using two words (64 bits). Whenever the microprogram executes one of the two system calls that exchange messages, the receiving parameters and return members are connected with the input or output of the Message Queue.

Similar resource is the Data-FIFO logical element that despite using the same design, it does not have a pre-defined type. Instead, the length of data is based on a predefined number of words that can be changed at the top-level of each accelerator. In doing so, different lengths between different accelerators can be used in the same design, or even be different between sent and received in the same design. For this reason, only the control signals are provided by the microprogram, and the datapath is not included in the kernel design, being connected directly to the *HW-Task*.

With respect to control signals, three are used in each logical direction of transfer, one being input and two outputs, as can be seen in the same figure. Depending on the direction, the first output signal indicates to the microprogram that each device is ready to receive, or to provide available data. The input signal triggers the exchange and is produced by the microprogram outputs, and the last control signal is used as a handshake, indicating that the data has been received, or should be read at the output. Implementation details will be discussed in section 4.4, and the type of messages that can be exchanged will be discussed in section 3.2.

Figure 3.17 also shows the connection with the Time Event unit. Since it is a local resource, none of the previous interfaces is used while executing the wait system call. For this purpose, three control signals connect with the microprogram, and seventeen bits from the *parameters* and *return* members in the system call records, connect with the inputs and outputs of the Time Event unit. In the *parameters* member, sixteen bits specify the timeout value and the position of one bit is used to establish the connectivity between the received event and the input event in the device. In the *return* member, the same 16-bit position is used to send back the remaining time, and one bit is used to indicate the occurrence of timeout. Once in execution, the microprogram waits for the ready signal before starting a new event. The new event is set using the trigger signal which gives rise to the sleep output that suspends the microprogram execution and strategic points across the accelerator model. Upon receiving an active input event, or an exhausted timeout signal, the sleep output is cleared and the microprogram resumes the execution.

Figure 3.18, shows excerpts of two files containing Time Event related descriptions, used by the system-level datapath. These are part of the configuration package, lines 12 to 47, and the kernel package, lines 10 to 95. It is also possible to observe excerpt descriptions of the system-level datapath in the *Kernel Core* design, lines 562 to 576. In the configuration package, we find two redefinable constants that set the timeout bit width and the time interval before the kernel sleep state, as mentioned earlier in subsection 3.1.2.

```
12  package hal_asos_configs_pkg is
    ...
46  constant C_EVENT_TIMEOUT_WIDTH :natural :=16;
47  constant C_KERNEL_TIME_TILL_SLEEP:
                              std_logic_vector(C_EVENT_TIMEOUT_WIDTH-1downto0):= x"C0DE";
    ...
10  package hal_kernel_pkg is
    ...
93  subtype PARAM_TIMEOUT_VAL IS std_logic_vector(C_EVENT_TIMEOUT_WIDTH-1 downto 0);--[15:0]
94  constant PARAM_SIGNAL_SOURCE: natural:= (C_MESSAGE_WIDTH-1);
95  constant PARAM_TIMEOUT_STATUS: natural:= (C_MESSAGE_WIDTH-2);
    ...
126 architecture Behavioral of kernel_core is
    ...
562 case cresponse_i.sys_call_id is
563    when SYS_CALL_WAIT_EVENT_TIMEOUT =>
564       time_event_selected_timeout <= sys_call_i.parameters(PARAM_TIMEOUT_VAL'RANGE);--[15:0]
565       time_event_selected_event <= sys_call_i.parameters(PARAM_SIGNAL_SOURCE);--[63]
566       dresponse_i.return_arg(PARAM_TIMEOUT_VAL'RANGE)<=time_event_remaining;--[15:0]
567       dresponse_i.return_arg(PARAM_TIMEOUT_STATUS)<=time_event_timeout_signal;--[62]
    ...
```

Figure 3.18: System-Level datapath - Time event parameters and return signals.

In the kernel package, we find positions for the parameters used in the system call members. The timeout received and returned parameters use the least significant bits (e.g., 15 down to 0) defined by the subtype of line 93. The signal event position is defined in line 94 and the timeout status flag position in line 95. Such positions are then used in *Kernel Core* descriptions to establish connectivity with the Time Event unit. Lines 564 and 566 use the timeout bit range in the assignment, and lines 565 and 567 assign the event status and the timeout flag using the above positions. The assignments are active when the system call ID matches the value specified in line 562.

Similarly, Figure 3.19 shows excerpt descriptions of the same design, but this time focusing the connections of two system calls that use the burst format, to exchange data across the Local-BUS and the

M00-BUS interfaces. In this format, a consecutive number of words is exchanged, starting from an initial offset and until a parameter number is achieved. These two resources follow a generic interface model that aims to promote data exchange to a single format, and in doing so, similar descriptions are used to implement the system-level datapath connections. Details about this generic interface will be discussed in section 3.3.

```
126 architecture Behavioral of kernel_core is
    ...
608 case cresponse_i.sys_call_id is
    ...
712 when SYS_CALL_WRITE_LBUS_BURST=>
713     task_status <= TASK_WRITING;
714     lbus_burst_syscall_i <= '1';
715     o_lbus_cs  <= '1';
716     o_lbus_addr<= sys_call_i.parameters(PARAM_LBUS_OFFSET'RANGE);--[53:36]
717     o_lbus_be  <= sys_call_i.parameters(PARAM_LBUS_BE'RANGE);--[35:32]
718     o_lbus_dout<= sys_call_i.parameters(PARAM_LBUS_WORD'RANGE); --[31:0]
719     target_i(target_i'high-1 downto 0)
                          <=signed(sys_call_i.parameters(PARAM_LBUS_BURST_LEN'RANGE)); --[63:54]
720     dresponse_i.return_arg(C_LBUS_DATA_WIDTH-1 downto 0) <= i_lbus_din; --[31:0]
721     dresponse_i.return_arg(PARAM_LBUS_BURST_LEN'range)
               <= std_logic_vector(to_unsigned(cresponse_i.index,PARAM_LBUS_BURST_LEN'length));
722     error_source <= not(i_lbus_wr_ack);
723 when SYS_CALL_READ_MBUS_BURST=>
724     task_status <= TASK_READING;
725     o_kernel2_mbus_addr(31 downto C_MBUS_OFFSET_WIDTH)
                <= sys_ram_addr_reg(31 downto C_MBUS_OFFSET_WIDTH);--[31:20]
726     o_kernel2_mbus_addr(C_MBUS_OFFSET_WIDTH-1 downto C_PAGE_SHIFT)<= std_logic_vector  (
                     UNSIGNED(sys_call_i.parameters(PARAM_MBUS_PAGE_PREFIX'RANGE))
                   + UNSIGNED(sys_ram_addr_reg(C_MBUS_OFFSET_WIDTH-1 downto C_PAGE_SHIFT)) );
727     o_kernel2_mbus_addr(C_PAGE_SHIFT-1 downto 0)
                <= sys_call_i.parameters(PARAM_MBUS_PAGE_OFFSET'RANGE) & "00";--[11:0]<-[45:36]
728     o_kernel2_mbus_tlen <= sys_call_i.parameters(PARAM_MBUS_BURST_LEN'RANGE) ; --[63:54]
729     target_i(target_i'high-1 downto 0)
                <= signed(sys_call_i.parameters(PARAM_MBUS_BURST_LEN'RANGE))-1;
730     dresponse_i.return_arg(PARAM_MBUS_WORD'RANGE) <= i_mbus2_kernel_data;
731     dresponse_i.return_arg(PARAM_MBUS_BURST_LEN'RANGE)
                <= std_logic_vector(to_unsigned(cresponse_i.index,PARAM_LBUS_BURST_LEN'length));
732     error_source <= i_mbus2_kernel_cmd_error  or sys_ram_addr_reg(31);
    ...
```

Figure 3.19: System-Level datapath - Local-BUS and MBUS fields of system call.

As in the previous peripheral, the implementation is based on the settings that the configuration and kernel packages establish, and can be consulted in Listing C.19 attached to this document. For this, the system-level datapath implements connectivity with the inputs signals: word, offset, byte enable (BE) and transfer length, and the system call parameters member. In the opposite direction, the output signals: word and the effective number of words transferred, are connected with the return member. The latter signal is part of the kernel index service which allows the *HW-Task* to manipulate data using array logical structures at kernel-level, and it will be discussed in the next subsection.

In summary, the least significant 32-bit of the parameters and return members are used to exchange one word in both directions (lines 718, 720 and 730). The most significant 32-bit are split into 18 bits to specify the word offset in the interface (lines 716, 725 to 727), 4-bit in the BE field to select bytes that are affected during a word write operation (line 717), and lastly, the remaining 10-bit are used to specify the transfer length (lines 719, 728 and 729). These are default settings that map 256 k-Words on each interface and allow a maximum length of 1024 words (1 k-word) for sequential transfers. For completeness, the next subsection demonstrates the interoperability in these two system calls while describing details of the kernel run-time.

While implementing system calls, the kernel updates the status register with results of the execution. Such register receives information from multiple sources in the kernel design that when combined describe the accelerator state. Throughout the previous excerpts about the system-level datapath, we can find that a task status is updated according to the system call in progress (lines 713 or 724), to reflect the logical state of the kernel's operation. Such information must be complemented with the *blocked* and *sleep* flags from the microprogram and Time Event units, and the *dead* flag that results from the yield system call. Also, an error source is established that can trigger the error flag if the system call is not completed properly. All these flags are source of the status register that can be seen in Figure 3.20.

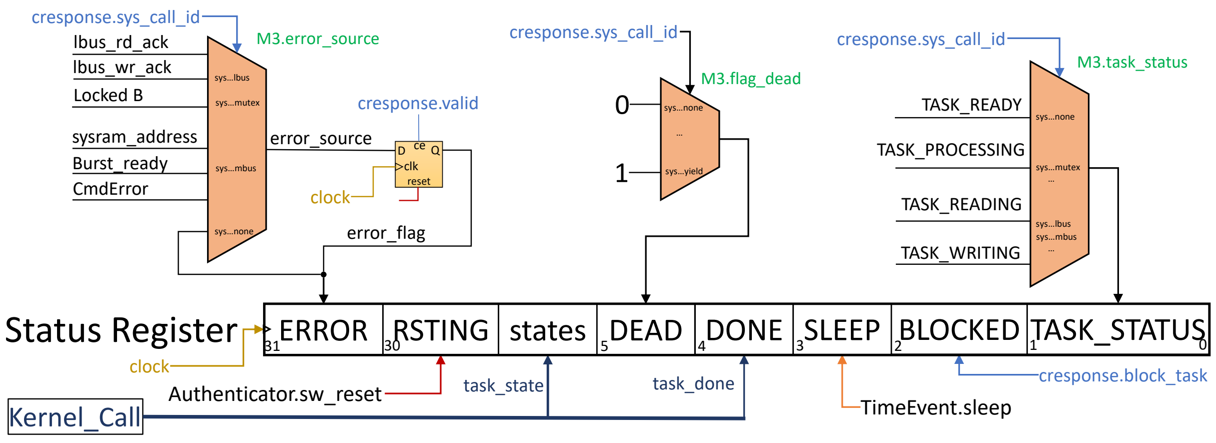


Figure 3.20: Kernel Core - status register signals.

In doing so, using the M3 multiplexer group, the system-level datapath implements the task status, the error source and the dead flag. The task status, sleep and blocked bits, will be updated at the next active clock transition, and the error flag will be triggered if a high logic value persists until completion of the system call. The dead flag signals a control state that results from the yield system call, from which the kernel can only return after a system reset or a software-demanded reset. In the latter case, the *Resetting* bit is kept high for the number of clock cycles that this operation requires. Also, using the *Kernel_Call* interface, the *HW-Task* publishes its control state in the 29- down to 6-bit range of this register, and uses the *Done* bit to signal the completion of its processing.

An error counter records the number of flag occurrences and can be read using the *S00_Control* interface on the accelerator model. The same interface can be used to read the status register, and read or write to the *sysram_address* register. The contents in the *sysram_address* register can activate the error flag in M3 if it is used before receiving a memory address. For this, its value after the reset is negative and the signal bit is used as an error source. Details of the *S00_Control* interface will be discussed in section 3.3.

3.1.7 Kernel Runtime

While executing system calls, the kernel may need to repeat a particular step of the microprogram in order to manipulate data above the word length. On the other hand, if a local resource does not allow such an exchange without prior handshake, the kernel is forced to repeat the system call in its entirety. To deal with these execution variants, the kernel relies on index service to manipulate consecutive words, or the procedure scheduler to establish a logical sequence in the execution of system calls. Figure 3.21 shows a simplified diagram describing the resource logic used to implement the index service.

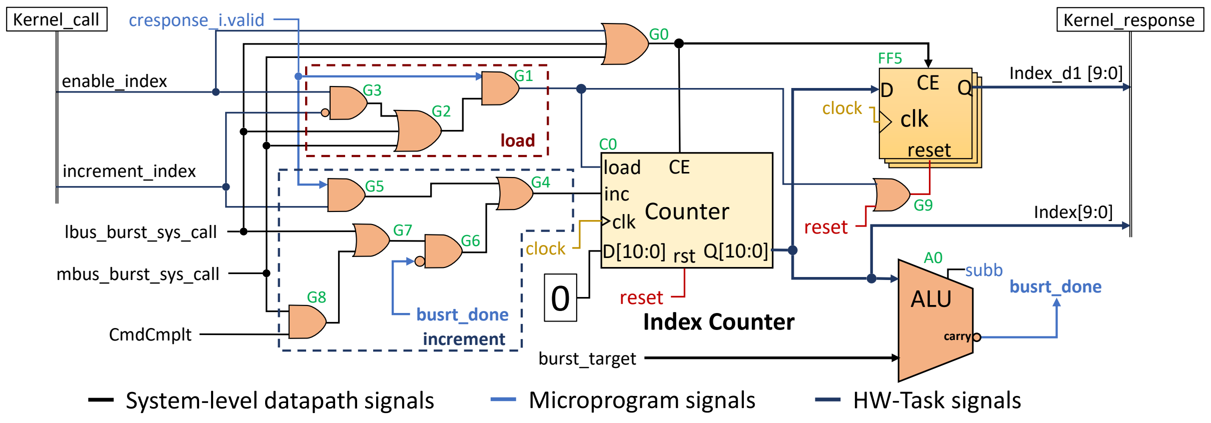


Figure 3.21: System-Level datapath - Index management service.

Such functionality relies on the C0 counter, which a procedure can use to index elements in a logical array. In similar way, the same index can also be used as offset that adds to a base parameter to compute incremental addresses. To cope with reading operations in the next clock cycle, FF5 provides the index value with the delay of one counting cycle. This allows the procedure to specify a new offset or a new word using a new index value, while reading the current word from the return register and storing it using the previous index.

In its design, this logic provides means of internal control using the microprogram and the system-level datapath signals, or explicit control in the procedure descriptions. While using internal control, the logic combination G0, uses one of the burst signals from the system-level datapath, to enable the clock sources of the C0 counter and the FF5 register. The increment of the index is activated by the same signals using G7 or G8 depending on the system call. In this way, the output value will increase with the clock cycle when using G7, or it will be postponed until a clock cycle in which a complete signal is active in G8. A target length is subtracted from the current value of the index using A0 (i.e., index-target= carry), which gives rise to a *done* signal when they are equal, disabling the increment signal using G6. In the burst mode the kernel executes one system call and repeats the step in the microprogram where it exchanges the new received word. At completion, the *valid* signal in G1 sets the counter and the delayed index to the initial values, using one of the active inputs in G2.

A procedure can request this service through the kernel's call and response interfaces by activating the enable and increment signals. The increment signal is combined with the valid signal in G5 which gives rise to the *inc* input at the completion of a system call. In doing so, when the procedure controls the index service, the kernel repeats the system call in its entirety to submit a different word on each execution. In the last word, the procedure disables the increment signal while keeping the clock active and the logical combination in G3, G2 and G1 activates the load input that sets the outputs to its initial values.

To repeat a system call, the kernel can rely on the procedure scheduler that, when active, superimposes the block signal from the microprogram. In this mode of operation, the kernel can repeat the system call until the procedure descriptions suspend the scheduling service, or allow the advance to another concurrent procedure that may be implemented in the same descriptions. This is usually the case when the *HW-Task* implements user package procedures that are composed of kernel procedures. Details of these procedures will be discussed in the following section.

With the scheduler service, concurrent procedures are executed according to a pre-established order that aims to implement composite features in the accelerator model. The scheduler policy is based on incremental counting used to select procedures for execution, multiplexing the kernel interfaces to each procedure accordingly. In Figure 3.22, we can see a logical diagram that describes the implementation of this service. The *HW-Task* can invoke this functionality using the kernel interface, by enabling the service and setting the reschedule signal at each procedure completion. In doing so, it must establish a logic '1' in the enable signal, and in this instant, the zero value in the counter establishes a connection with the procedure in the corresponding sequence.

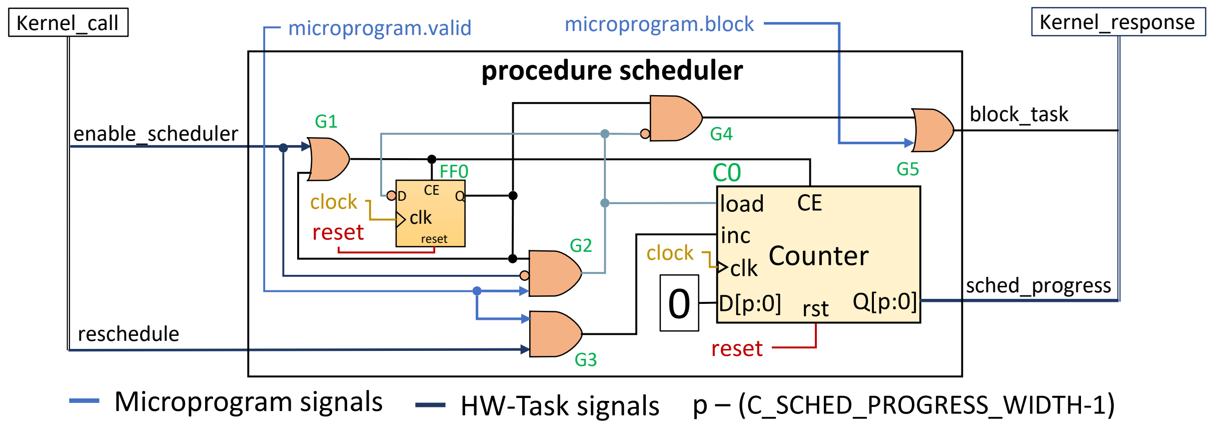


Figure 3.22: Kernel Core - execution progress service.

The enable signal activates the FF0 and C0 clocks using the logical combination in G1, and with a logic ’0’ at the output of G2, FF0 activates the Q output at the next clock transition superimposing the microprogram’s block feature in G5. In this condition, the *HW-Task* cannot return from the procedure execution and the kernel repeats the system call consecutively, until a logical combination of the *reschedule* and the *valid* signals in G3, activate the *inc* input, which gives rise to a new value in the C0 output.

Each new count value will select a different kernel procedure for execution, and in the last step, a logic ’0’ in the enable signal, indicates that the scheduler service must conclude with the execution of the current system call. For this, the logic combination in G2 activates the *load* input of C0 when the *valid* signal is set high, by the last step of the microprogram execution. At that time, the input of FF0 receives a logic ’0’ and in the next active clock transition, the outputs in these two sequential units reload the initial values. A logic ’0’ in FF0 will release the context of the *HW-Task* so that it proceeds to a new state in its logic diagram.

3.1.8 Kernel Call and Response

From the kernel procedure’s point of view, the *Call* and *Response* interfaces are the representation of the *Kernel Core*. Here, we can find the signals that establish correspondence with the components involved in the system call execution. Other signals that connect to the indexing and scheduling services can also be found on this interface, as well as signals that are intended to control the *HW-Task*. Figure 3.23 shows an excerpt from the kernel package that establish these interfaces. In these descriptions, the record members can be grouped by sub-levels of datapath or control. In the datapath sub-level, lines 247 and

260, we can find the *parameters* and *return_arg* members that perform the effective exchange of data in both directions. At the control level, we can find the *this_call*, line 248, and the system call ID, line 246, that correspond to the microprogram inputs, and the signals that in response determine the execution state, such as *block_task*, in line 266, or *sleep_task*, in line 267. A valid response must include the sys-call id in the response field, line 259, together with an active *valid* signal, line 261, and the error flag, line 268, that can also be used in the *HW-Task* context.

```
     ...
 10  package hal_kernel_pkg is
     ...
244  type kernel_input_t is
245    record
246      sys_call_id :sys_call_t;
247      parameters: std_logic_vector (C_MESSAGE_WIDTH-1 downto 0);
248      this_call: std_ulogic;
249      enable_scheduler: std_ulogic;
250      reschedule: std_ulogic;
251      enable_index: std_ulogic;
252      increment_index: std_ulogic;
253      task_state: std_logic_vector(23 downto 0);
254      task_done: std_ulogic;
255  end record;
     ...
257  type kernel_output_t is
258    record
259      sys_call_id : sys_call_t;
260      return_arg:std_logic_vector (C_MESSAGE_WIDTH-1 downto 0);
261      valid:std_ulogic;
262      kernel_progress: natural range 0 to (2**C_KERNEL_PROGRESS_WIDTH)-1;
263      sched_progress: natural range 0 to (2** C_SCHED_PROGRESS_WIDTH)-1;
264      index: natural range 0 to (2**C_KERNEL_INDEX_WIDTH)-1;
265      index_d1: natural range 0 to (2**C_KERNEL_INDEX_WIDTH)-1;
266      block_task: std_ulogic;
267      sleep_task: std_ulogic;
268      error_flag: std_ulogic;
269      task_reset: std_ulogic;
270      task_run  : std_ulogic;
271  end record;
     ...
```

Figure 3.23: Kernel Core - input and output interface types using VHDL records.

To provide the scheduling and index services requests, lines 248 to 252 implement the corresponding enable and trigger inputs, and in lines 263, 264 and 265, the kernel provides each service outputs. Additionally, in line 262 the kernel output interface provides the execution step in the microprogram and the *HW-Task* can use this value in its design when needed. The *task_run* in line 270 and a *task_reset* in line 269, are used to synchronize the *HW-Task* with the control unit of the *Kernel Core*. In the opposite direction, a *task_done* signal in line 254, informs the control that the *HW-Task* has completed a processing round. The 24 bits of the *task_state* can be used by the *HW-Task* to publish its control state in the status register of the accelerator.

To comply with the use of automated block design tools, these interfaces were striped-down to individual signals that can be found in the *HW-Task* and the Accelerator top-level descriptions. Such signals are based on the equivalent standard logic and use appropriate XML descriptions to provide rules that prevent erroneous connections, as well as support the validation features that can be found in today's modern EDA tools. For this reason, the new signals are logically grouped in the *M00_Task* and *M00_Kernel* interfaces that have a corresponding *S00_Task* and *S00_Kernel* connection. To transverse between the kernel *Call* and *Response* interfaces, and the standard logic representations, the kernel package provides import and export procedures that can be seen attached in Listing C.20.

3.1.9 Kernel Procedures

To interface the set of system calls that the *Kernel Core* implements, the HAL-ASOS tool provides the procedures in the kernel package that *HW-Task* can implement in its datapath. These procedures establish the fundamental interface of the system calls and in doing so, they define the type and number of parameters required for each functionality. In essence, such procedures export the system calls to the *HW-Task* design, while seeking to expose the distinct features that a given resource can have. Therefore, the number of procedures that are used to invoke system calls is higher than the set of features that the microprogram implements. An excerpt from the kernel package that declares the most relevant procedures in the accelerator model can be found in Listings C.21, C.22 and C.23 attached to this document.

In their implementation basis, the kernel procedures are distinguished by the exclusive use of kernel design-imposed types, and in this way, the underlying model is limited to a connectivity between parameters and interface members. To address the uniqueness of the kernel interfaces and a four steps model imposed by the microprogram, the calling entity can implement a kernel procedure for each state in its control logic. In concurrent form, it can implement other procedures in the kernel package which aim to obtain an adequate formatting of the parameters involved. For more complex and feature-oriented procedures, the *HW-Task* can implement user package procedures. These will allow the use of more elaborate types that ultimately seek to represent application memory objects or the Linux device model. Implementation details of the user package procedures will be discussed in section 3.2.

Figure 3.24 outlines the connectivity between the *HW-Task* and the *Kernel Core* while it implementing system calls. On the right side of the figure, we can see a representation of the *Kernel Core* that is based on the functional units in the kernel runtime. On the left side of the figure, we can see a simplified example of the *HW-Task* design. It can also be seen the top-level interfaces *M00_TASK* and *M00_KERNEL* discussed in the previous section.

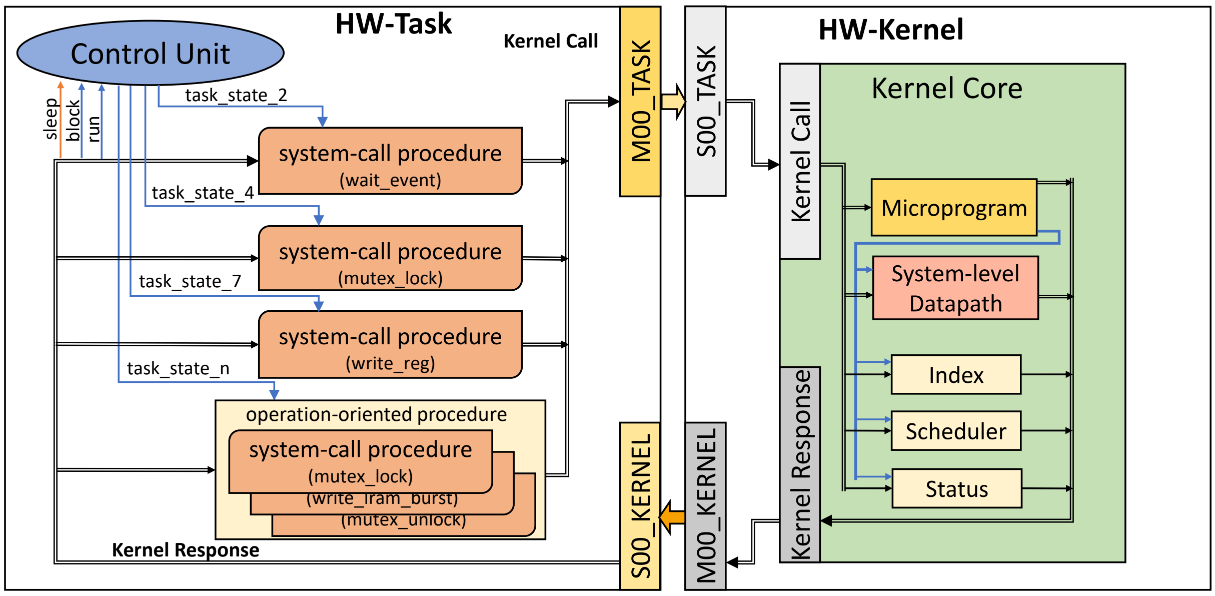


Figure 3.24: Kernel Core - execution runtime overview from the *HW-Task* perspective.

The control unit of the *HW-Task* is responsible to activate concurrent system call procedures in its datapath, while it receives control signals to synchronize with the kernel execution. The moment a system call starts, the control unit suspends, keeping the procedure active until completion of the microprogram execution. Once active, the inputs and outputs of the procedure are updated by the execution of the microprogram and the system-level datapath, and in this sense, it is considered that these signals are controlled by the context of the kernel. To that extent, the procedure circuitry is seen as an HW extension of the *Kernel Core*.

In the same figure, it can also be seen a user package procedure that is composed of three concurrent kernel procedures. Similarly, only one user procedure can be used in each state of the *HW-Task* control, and that particular state is repeated for each system call consecutive execution. For that, the user procedure relies on the kernel scheduling service to select each procedure accordingly. The *HW-Task* design is accountable for the additional logic inside a user procedure, to interconnect each kernel procedure with the kernel interfaces. In that sense, the interconnect logic is considered an extension of the *HW-Task*'s datapath.

In Figure 3.25 it can be seen the descriptions of the *wait_event* procedure in the kernel package. Such procedure, interfaces the wait system call and can be used to put the *HW-Task* into a sleep state while waiting for a signal event in its design. For this, the *HW-Task* must connect the target signal with the event input, and when required, specify a timeout value using the *timeout_val* parameter. Alternatively, it can connect a logic '0' in the event input for using this service as time-based event. The descriptions in this procedure specify the system call ID on line 1152, and provide the received timeout value, on line 1153. In line 1154, it establishes a connection between the target signal and the system call parameter, and at line 1155, the *this_call* signal is set, triggering the microprogram execution.

```
  10  package hal_kernel_pkg is
      ...
1144  -------------------------------------------------------------------------------------------
1145  procedure wait_signal_event(signal i_call       : out kernel_input_t;
1146                              signal o_response   : in kernel_output_t;
1147                              signal i_event      : in std_logic;
1148                              signal is_event     : out boolean;
1149                              constant timeout_val: in integer :=0) is
1150  ---------------------------------------------------------------------------------------------
1151  begin
1152      i_call.sys_call_id <= SYS_CALL_WAIT_EVENT_TIMEOUT;
1153      i_call.parameters(PARAM_TIMEOUT_VAL'RANGE)
                        <= std_logic_vector( to_unsigned(timeout_val,C_EVENT_TIMEOUT_WIDTH));
1154      i_call.parameters(PARAM_SIGNAL_SOURCE) <= i_event;
1155      i_call.this_call <= '1';
1156      is_event <= to_boolean(not(o_response.return_arg(PARAM_TIMEOUT_STATUS)));
1157  end procedure wait_signal_event;
1158  ---------------------------------------------------------------------------------------------
```

Figure 3.25: Kernel package procedures - wait signal event descriptions.

To exchange parameters, the procedure descriptions make use of the kernel's input interface, that connects with the output in this procedure, line 1145, using the *HW-Task* design. Likewise, the kernel output interface connects to the corresponding procedure input in line 1146. In this case, the timeout occurrence signal is used to stimulate the *boolean* output *is_event*, on line 1156. The procedure implementation remains active until the microprogram completes its execution and releases control of the *HW-Task*. In these descriptions, we can see that it completes without returning the remaining time value. As result, the *HW-Task* receives the *boolean* parameter that indicates the event occurrence, and if desired, the remaining time can be consulted using the *return_arg* member after the execution has completed. A simulation wave plot that includes this procedure descriptions was discussed in subsection 3.1.5 and can be seen in Figure 3.16.

In Figure 3.26 we can see a snippet from the kernel package, that describes a procedure which reads from the local memory using the burst format. In the parameters of this procedure, we can also find the kernel input and output interfaces, followed by input parameters, such as transfer length and offset in the LRAM, and output data in the length of a word. In lines 1414 and 1415 the procedure specifies the system call and triggers the execution in the microprogram. The descriptions forward the input parameters to the corresponding signals on the kernel *Call* interface, in lines 1416 to 1418, and in the opposite direction, in line 1419, it forwards the received word to the output parameter *data*.

```
  10  package hal_kernel_pkg is
      ...
1405  ------------------------------------------------------------------------------------------------
1406  procedure lram_read_word_burst (signal i_call : out kernel_input_t;
1407                                  signal o_response: in kernel_output_t;
1408                                  transfer_len:in natural;
1409                                  lram_offset: in natural range 0 to (2**CLRAM_AWWIDTH)-1;
1410                                  data: out std_logic_vector(31 downto 0))is
1411  ------------------------------------------------------------------------------------------------
1412  begin
1413
1414  i_call.sys_call_id <= SYS_CALL_READ_LBUS_BURST;
1415  i_call.this_call <= '1';
1416  i_call.parameters(PARAM_LBUS_BE'RANGE) <= "0000";
1417  i_call.parameters(PARAM_LBUS_ADDR'RANGE)
                     <= '1'& std_logic_vector(to_unsigned(lram_offset,CLRAM_AWWIDTH));
1418  i_call.parameters(PARAM_LBUS_BURST_LEN'RANGE)
                     <= std_logic_vector(to_unsigned(transfer_len,PARAM_LBUS_BURST_LEN'LENGTH));
1419  data := o_response.return_arg(PARAM_LBUS_WORD'RANGE);
1420
1421  end procedure lram_read_word_burst ;
1422  ------------------------------------------------------------------------------------------------
```

Figure 3.26: Kernel package procedures - read local-RAM in burst format descriptions.

Once in execution, the system call will use the system-level datapath in the *Kernel Core*, to activate the index service and the microprogram will repeat the step two until it receives a burst done signal from the index service. In doing so, a different word is sourced at the *data* output parameter, and the *HW-Task* is responsible for implementing the path that consecutively distributes the received words by the distinct destination registers. In this case the *index_d1* field of the kernel interface will be used to select the target register. Once again, the descriptions in this procedure can be translated into wires between the kernel interfaces and the datapath of the *HW-Task*.

The kernel package includes more procedures beyond those that were seen until now. Due to the extensive descriptions that these require, we've decided to include procedures that relate to the accelerator model internal features. Other procedures target the synchronization and manipulation of software features in the Linux OS or the target application. We shall discuss some of these in the following section.

3.2 HW-Task

In the HAL-ASOS accelerator model, the *HW-Task* represents an execution flow that is part of a Linux application and was implemented using equivalent logic circuits that counterparts to a software-based computation. In this way, the base concept of a Linux application is retransformed to a combination of hardware and software execution flows, dictated by the system requirements. When offloading software computations via *HW-Task*(s), an application can benefit from the intrinsic parallel nature in the hardware circuits, which were designed according to the system's requirements and use just the right resources. On the other hand, the boundary crossing between the HW and the SW implementations can overcome the benefits in the mixed design architecture. To avoid multiple and consecutive design iterations, the candidates to computation offload must be identified using the OProfile tool in the design methodology, followed by a functional Co-Simulation as covered in Chapter 2.

In this section, we describe the *HW-Task* design and introduce the HAL-ASOS programming model through examples. We then describe the *HW-Task* interaction with the Linux software application and the OS services.

3.2.1 Programming Model

The *HW-Task* is the processing entity in the HAL-ASOS programming model that integrates FPGA-based computation into Linux software applications. To ease development, the framework provides a template design for the *HW-Task* that is independent from the *HW-Kernel*. It includes RTL descriptions with: (1) a state machine structure for the control unit that is synchronized with the *Kernel Core* system calls; (2) the sleep infrastructure where all sequential units subject to this state will be instantiated; and (3) the connectivity between the *HW-Task* design and the kernel interfaces to and from the top-level signals in this component. Figure 3.27 shows an example of the *HW-Task* template model and to better illustrate the design and the internal connectivity of this component, two black boxed IPs were added to the datapath.

To comply with the traditional design of the logic circuits, such design is divided into a control unit and a datapath. Here, the control unit implements the state logic that establishes the behavior of the *HW-Task* by sensing specific signals from the datapath. Additionally, some of the *HW-Kernel* interface signals are also considered, namely the *task_run* and the *task_error*. The first signal triggers the *HW-Task* activity for the required processing rounds, and the latter signals a system call error that can be used to trigger handling states at the *HW-Task* level.

In this model, the FF0 register establishes the current state of the control logic, where each state determines the set of active control signals and the value of the next state in M0. When a state requires a procedure to implement a system call, the value of the next state can be set concurrently, as if it was a one clock-cycle state. Once reaching such state, the control unit will activate the chosen procedure and

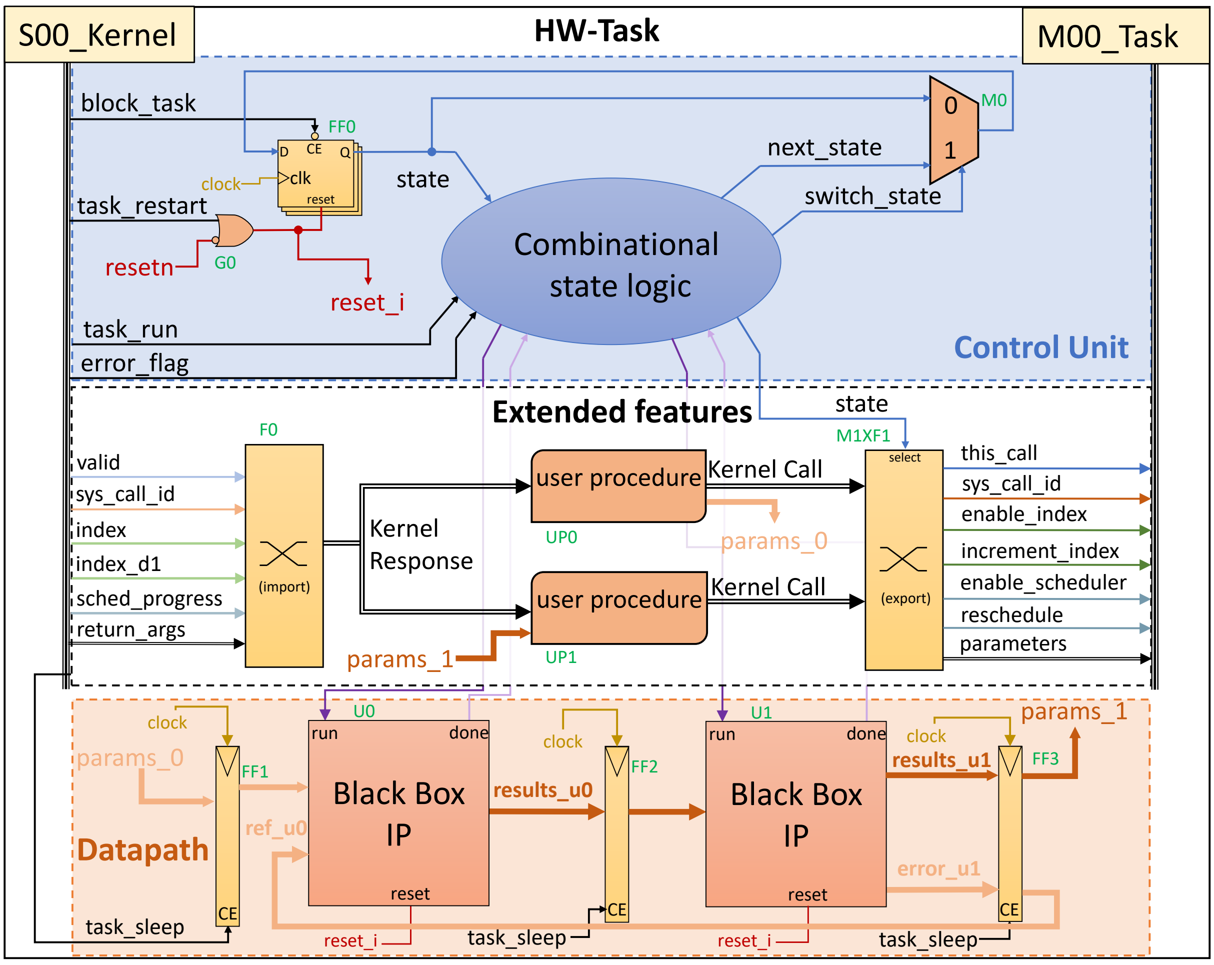


Figure 3.27: Hardware Task - simplified example architecture.

in response, the *HW-Kernel* interface will raise the *block_task* signal. In doing so, the current state will be maintained for the necessary number clock-cycles, thus allowing the procedure to complete. Upon completion, *HW-Kernel* lowers the *block_task* signal, enabling back the clock in FF0 and in the next active clock transition the state register is updated with the next state value.

In its turn, the datapath provides the necessary resources for computing the set of data it receives as input. Generally, a datapath consists of specific design logic functions to compute data and sequential logic for storing results. Depending on the data-level parallelism, a datapath can achieve a fixed length expressed in clock-cycles, usually referred as processing stages. Such stages are intended to promote the throughput of the datapath by maximizing the usage of the logical units that it implements, following a design structure usually referred as the execution pipeline. In the datapath of Figure 3.27, the two IPs U0 and U1 are the implementation of this combinational logic and the registers FF1, FF2 and FF3 are the sequential units that establish the progress of the data across the datapath.

The proposed programming model implements an extended layer with application-based functionality, that uses a mix of control and datapath signals. In this new design level, the kernel or user package procedures are implemented concurrently and are primarily scheduled selecting each *Kernel_Call* at the M1XF1 inputs, by use of the state register. Conversely, each procedure receives a connection with the *Kernel_Response* in parallel with other procedures in the same level. For this, logic function F0 translates signals from *S00_Kernel* to the *Kernel_Response* interface, and similarly, M1XF1 is composed of a logic function that translates the selected *Kernel_Call* to the *M00_Task* interface. For this example, we have included two user procedures, UP0 and UP1, where the topmost procedure outputs the received data to FF1 which is input to the datapath, whereas the bottommost procedure receives data as arguments using FF3 which is output from the datapath. As such, the simplified *HW-Task* example is using the two concurrent procedures to exchange application data using the accelerator model. For a better understanding, Figure 3.28 shows an excerpt from the *HW-Task* that describes this implementation.

```
209  import_kernel_response( s00_kernel_sys_call_id, s00_kernel_return, s00_kernel_valid,
210                          s00_kernel_syscall_progress, s00_kernel_sched_progress,
211                          s00_kernel_index,s00_kernel_delayed_index, s00_kernel_block_task,
212                          s00_kernel_sleep_task, s00_kernel_error_flag, kernel_response);
213
214  export_kernel_call(kernel_call, m00_task_sys_call_id, m00_task_parameters,
215                     m00_task_this_call,m00_task_enable_scheduler, m00_task_reschedule,
216                     m00_task_enable_index,m00_task_increment_index);
217  --------------------------------------------------------------------------------------------------
218  EXTENDED:process(state,params_u0_q, params_u1_q, kernel_response, local_offset_q,
      sys_offset_q)
219  --------------------------------------------------------------------------------------------------
220  begin
221  params_u0_i <= params_u0_q;
222   hal_asos_link_to_kernel(kernel_response,kernel_call);
223   case state is
224     when  st1_read_data=>
225      safe_read_lram_word32_burst(kernel_call,kernel_response,params_u0_i,4, local_offset_q);
226     when st4_write_data=>
227      safe_write_sysram_word32_burst(kernel_call,kernel_response,params_u1_q,8, sys_offset_q);
228     when others=> null;
229   end case;
230  end process EXTENDED;
231  --------------------------------------------------------------------------------------------------
```

Figure 3.28: Task Model - concurrent user procedure description example.

To connect the top-level interfaces with the implemented user procedures, lines 209 and 214 implement two concurrent kernel procedures. The first procedure receives as input the *HW-Kernel* slave top-level signals and outputs the *kernel_response* record, which is the output of the *Kernel Core*'s defined interface. In the opposite direction, the second procedure receives the *kernel_call* record as input and provides the output signals that connect with the *HW-Task* master interface. The two procedure implementations use only connect logic with the purpose of providing the *Kernel Core* interfaces locally. To implement the extended features level, a combinational process in line 218, uses the state register in the sensitivity list, together with *kernel_response* record and the parameters that each procedure requires as input. In these parameters, we can find the signals that connect with the datapath's logic units (i.e., params_u0_q and params_u1_q), and two address registers used in each procedure.

A case statement will activate the user procedures in lines 225 or 227, when the control logic reaches the *st1* or *st4* states, respectively. In doing so, the *kernel_call* will be updated with signals from a specific procedure implementation, while it senses the *kernel_response* received as argument. For as long as the control unit remains in one of these logic states, the corresponding procedure will be set to active to update the correspondent outputs according to the changes in the input parameters that it receives. In response, it will generate the appropriate outputs using the *kernel_response* or the *params_u0_i* signals. When the control logic is implementing the remaining states that are not considered in the enclosed cases, a kernel procedure at the line 222 will keep the connectivity between the *HW-Task* and the kernel interface in a consistent state, using the default logic signals. Similarly, the *params_u0_i* will be tied to the last received results that were stored in the FF1 register, using in a self-closed loop design (line 221).

Figure 3.29 shows an excerpt with descriptions for implementing the sequential logic in the datapath and control unit of the *HW-Task* example. In these, a logic function is used to establish the internal reset of such logic, using the target platform or the kernel software reset sources (line 161). In the line 163, a process describes the FF0 state register that is set to the initial state by the active rise in the clock signal and a logic '1' in the internal reset (in line 168). If otherwise such reset is low, a logic '0' in *block_task* signal will store the next state in the logic sequence. Using this description style, the blocking signal will be source to the clock enable pin of this logic element by using the necessary logic function. In doing so, the clock will be disable when executing kernel procedures, thus providing the necessary clock cycles before storing the next state value and proceed in the state logic.

A similar style is used to describe the sequential elements in the datapath using the process at the line 175. The sleep task signal in the kernel interface is used to enable the clock in these logic elements. In this way, the clock will be active while executing kernel procedures thus allowing the output arguments to be stored in these elements. Once in sleep condition, the clock source is disabled in the logic described by both processes and since such condition is the result of the wait system call, it also ensures a block condition until the input event is detected. Nevertheless, a distinction must be made according to the necessary instance in time where the provided source is available. A block condition will cease in the last step of the system call execution, whereas the sleep condition will be completed in the next state in the

```
reset_i <= s00_kernel_swrst or not(resetn);
-----------------------------------------------------------------------------------------------
ff0:process(clock)
-----------------------------------------------------------------------------------------------
begin
    if rising_edge(clock) then
        if reset_i = '1' then
            state <= st0_ready;
        elsif kernel_response.block_task = '0' then
            state <= next_state;
        end if;
    end if;
end process ff0;
-----------------------------------------------------------------------------------------------
dpath_seq:process(clock)
-----------------------------------------------------------------------------------------------
begin
    if rising_edge(clock) then
        if reset_i = '1' then
            params_u0_q <= (others=>(others=>'0'));   --reset ff1
            result_u0_q <= (others=>(others=>'0'));   --reset ff2
            params_u0_q <= (others=>(others=>'0'));   --reset ff3 high
            u1_error_q  <= (others=>(others=>'0'));   --reset ff3 low
        elsif kernel_response.sleep_task = '0' then
            params_u0_q <= params_u0_i;   --ff1
            result_u0_q <= result_u0_o;   --ff2
            params_u1_q <= params_u1_o;   --ff3 high
            u1_error_q  <= u1_error_i;    --ff3 low
        end if;
    end if;
end process dpath_seq;
-----------------------------------------------------------------------------------------------
```

Figure 3.29: HW-Task Model - Sequential units descriptions using VHDL.

logic sequence after implementing the wait procedure.

3.2.2 User Procedures

User package procedures, or user procedures, are a subset of low-level operations, wrapped inside a VHDL language procedure that can be instantiated when developing *HW-Task*s for the HAL-ASOS accelerator model. Such procedures operate at the extended level in *HW-Task* and can be seen as an auxiliary subprogram, similar to a library used in software-based applications. In its essence, a user procedure provides means to implement a specific feature that is based on more than one HW system call. Thus, they establish interface with the required input and output parameters while implementing the correspondence with the required system call procedures. In addition, user procedures are also distinguished by providing advanced data formats, such as unconstrained arrays or multiple words, or user-defined types that are not supported at kernel-level, and a semantic abstraction of the Linux programming interface.

Figure 3.30 shows a typical architecture of a user procedure that implements concurrent system calls from the kernel package, and uses the scheduler service in the *Kernel Core* to ensure a proper execution.

The scheduling service divides the procedure into n steps, and establishes a sequence in the concurrent system calls, by sensing the *reschedule* signal that is provided internally. In doing so, the procedure descriptions implement M0 that ensures an exclusive connection between the *Kernel_Call* interface and the subset of outputs from each system call, where all inputs connect with the *Kernel_Response* and receive input parameters in parallel.

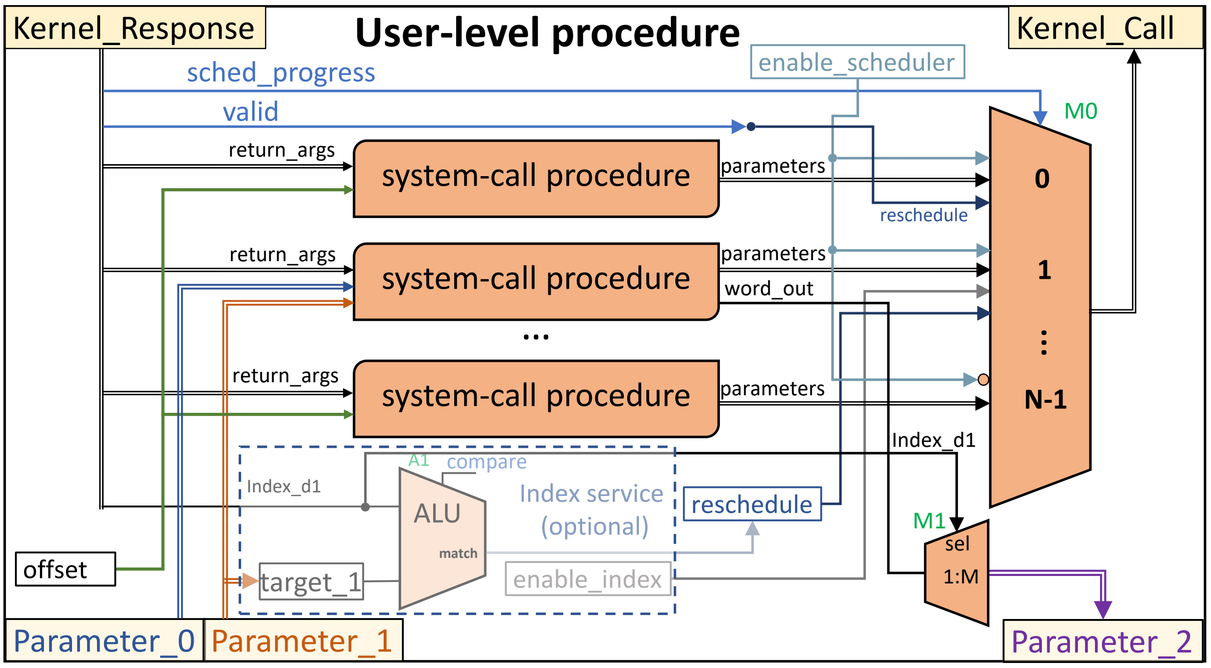


Figure 3.30: User package procedures - scheduling concurrent system calls.

From the bottom of the same figure, it can also be seen that the user procedure receives inputs and provides outputs with the *HW-Task* context. Here, two inputs and one output parameters are used (i.e., *Parameter_0*, *Parameter_1* and *Parameter_2*, respectively), where together the input parameters are read-only signals and the output parameter represents *HW-Task* resources, handled by the execution in the user procedure. A constant offset is also provided internally and connects with the top- and bottom-most system calls. These two system calls are a common example of the mutual exclusion invocation, using the *HW-Mutex* lock and unlock procedures in the kernel package.

The input parameters provided by the *HW-Task* context, are connected to the system call used in step 1, and is a typical example of a multiple word read that expects an address offset and transfer length as input, and where a consecutive number of words can be provided as output. In such case, the system call implements a burst format transfer and the target length is used at kernel-level to repeat the word exchange in the required number of times. If otherwise the involved resources do not allow such format, an alternative single word exchange system call can be selected to implement an equivalent operation. For this reason, the target parameter will be used at the *HW-Task* level to implement repeated executions of the same step, until the transfer length is completed. Optionally, to control the repetition process, ALU A1 is used to compare the index register with the received target length.

To establish the desired scheduling, in step 0, the *Valid* flag is used to trigger the first scheduling operation. Similarly, in step 1, the same flag can be used if the system call provides burst format, or alternatively, if a system call repeat is in use, the reschedule signal can result from the *match* signal in A1. Considering that an array of data is provided as output of the user procedure, a demultiplexer M1 is implemented locally to connect each of the 32-bit words to the corresponding position in the array. Such component, uses the index register in the delay of one clock cycle, to select the appropriate position of each word of data that is received from the system call. In the last step, the scheduler service is deactivated, resulting in the *HW-Task* context release with the completion of the system call, and therefore, the conclusion of the user procedure.

For completeness, Figure 3.31 shows a user procedure description that implements similar architecture to the diagram of Figure 3.30. Such procedure allows to safely read a consecutive number of words from the *SYSRAM* memory region in the host system. For this, in steps 0 and 2, it invokes the system calls to lock and unlock the required HW-Mutex, making use of the internal offset as parameter in each procedure (e.g., lines 1775 and 1783, *CSYSMUTEX_WOFFSET*, defined externally in the configuration package).

In step 1 starting at line 1775, the procedure invokes the system call to read a consecutive number of words, expressed by the input parameter *word_len* and starting at the specified input parameter *offset*, while using the burst transfer format. In doing so, it requires the use of the *index_d1* value to index the correct position in the *pbuff* array, as shown in line 1779. To ensure a proper memory address that is aligned with the host system, the received word-based offset is multiplied by four or by eight according to the parameter *C_HOST_ARCH* defined in the configuration package. The resulting logic are simple wires implementing a misaligned connection between input parameter *offset* and the variable *param_woffs* at line 1760, which in turn is an input parameter of the system call at line 1778.

An *if* clause is used at line 1764 to prevent the procedure from running with transfer length zero. It immediately disables the scheduler and releases the *HW-Task*, preventing any connection with the *i_call* interface. If otherwise the *word_len* is greater than zero, the scheduler service is activated in line 1769, blocking the *HW-Task* for multiple system calls. Once reaching step 2 starting at line 1781, the service

```
package hal_asos_user_pkg is
...
-------------------------------------------------------------------------------------------
procedure safe_read_sysram_word32_burst (signal i_call : out kernel_input_t;
                      signal o_response: in kernel_output_t;
                      signal pbuff: out t_array_slv_32;
                      constant word_len: in natural;
                      constant offset: in natural)is
-------------------------------------------------------------------------------------------
constant ALNMNT: natural:=POW2(C_HOST_ARCH/8);
variable param_woffs: unsigned(C_HOST_ARCH-1 downto 0)
         := to_unsigned(offset,C_HOST_ARCH) rol ALNMNT;
variable rcvd_word: std_logic_vector(C_MACHINE_WIDTH-1 downto 0);

begin
if word_len = 0 then
  i_call.enable_scheduler <= '0';
  i_call.reschedule       <= '0';

else
 i_call.enable_scheduler <= '1';
 i_call.reschedule       <= '1';

  case o_response.sched_progress is
    when 0=>
          mutex_lock(i_call,o_response,CSYSMUTEX_WOFFSET);
    when 1=>
          i_call.reschedule<= o_response.valid;

          mst_bus_read_word_burst(i_call, o_response, word_len,
                                std_logic_vector(param_woffs),rcvd_word);
          pbuff(o_response.index_d1)<= rcvd_word;
    when 2=>
          mutex_unlock(i_call,o_response,CSYSMUTEX_WOFFSET);
          i_call.enable_scheduler <= '0';
    when others=>null;
  end case;
end if;
end procedure;
-------------------------------------------------------------------------------------------
```

Figure 3.31: User package - VHDL procedure to burst read the *sysram* memory.

is disabled at the same time the *mutex_unlock* is invoked. In doing so, the microprogram will ensure a blocked state until the completion of the current system call. Complementary procedure that uses multiple repetitions of a single-word based system call, can be consulted in attached Listing C.30.

When implementing user package procedures, the *HW-Task* design scales using combinational logic that consecutively selects each kernel procedure and connects it to the received parameters. The logic used is considered an extension from the *HW-Task* datapath and in doing so, care must be taken in order to avoid a critical path degradation. In some cases, to meet the performance requirements, the design must be refactored to upgrade the system calls scheduling, by promoting them to the level of the *HW-Task*'s *Control* unit while introducing the necessary state logic to efficiently select each required system call. An excerpt of the most relevant user package procedures can be found attached in Listings C.24 to C.29. They show procedures that relate to composite operations in the accelerator model, but most of all, the user procedures are intended to provide a semantic abstraction of the Linux programming interface.

3.2.3 Linux programming interface

The above user procedures implement composite operations that target local resources in the accelerator model. Other procedures in this package, aim at composite operations to integrate the behavior of the *HW-Task* in the target application, running on the Linux operating system. This integration explores a mixed implementation between HW and SW, to provide a semantic abstraction on the Linux programming interface, allowing indirect access from the *HW-Task* design to resources in the host system.

Generally speaking, the Linux programming interface brings together a set of functionalities that by themselves constitute a level of abstraction on resources installed on the host system. As a derivative of the Unix system, the granularity in this abstraction is the file, and within this category we can highlight different types such as: regular files used for storing information; virtual files used in IPC, such as sockets, shared memory, message queues, pipes or the standard IO; and virtual files used to represent HW resources or devices that altogether are of particular interest to the *HW-Task* design. The handling of such files is provided by the set of system calls implemented by the Linux Kernel, which allow the fundamental operations of reading and writing, or control, namely in the opening and closing of files, or establishing control flags.

With the inherent parallelism observed in the structure of today's IT systems, and the consequent sharing of resources between the various processing flows, synchronism and concurrency control become fundamental aspects in ensuring system performance, correctness and integrity of data. While some of the above mechanisms implement FIFO-type data formats, which by ordering and uniqueness in the stream of data provide some level of synchronism, others implement contention at kernel-level that is based on the system calls behavior. For resources such as shared memory regions, auxiliary mechanisms are generally employed and can include mutual exclusion via mutex, or a state change using condition variables. Alternatively, when these features are extended at the process-level, the use of named semaphores with its counter structure, allows to reproduce functionalities equivalent to the mutex and condition variable.

The Linux programming interface allows a vast set of additional features that are of less relevance to the *HW-Task* design, for the simple reason that their use in the HW does not translate into an increase in benefits for the host system. Among these we can refer: the creation and handling of processing threads; the memory subsystem; timing and sleep features; inter process signals; file system handling and others.

While some of these functionalities are replaced by features in the accelerator model, others can still be used in a complementary way through the SW programming model implemented by the HAL-ASOS framework.

To abstract the *HW-Task* design from the Linux programming interface, the kernel package in the HAL-ASOS framework provide a local representation of resources by using hardware descriptors as shown in Figure 3.32. For that, it once again defines appropriate VHDL types, to describe a logical structure that is aligned with the Linux descriptor approach, and is based on integer numbers. In these, positive non-null values are used to index a file table that contains a list of files opened by the application, and negative values are used to portray the occurrence of errors.

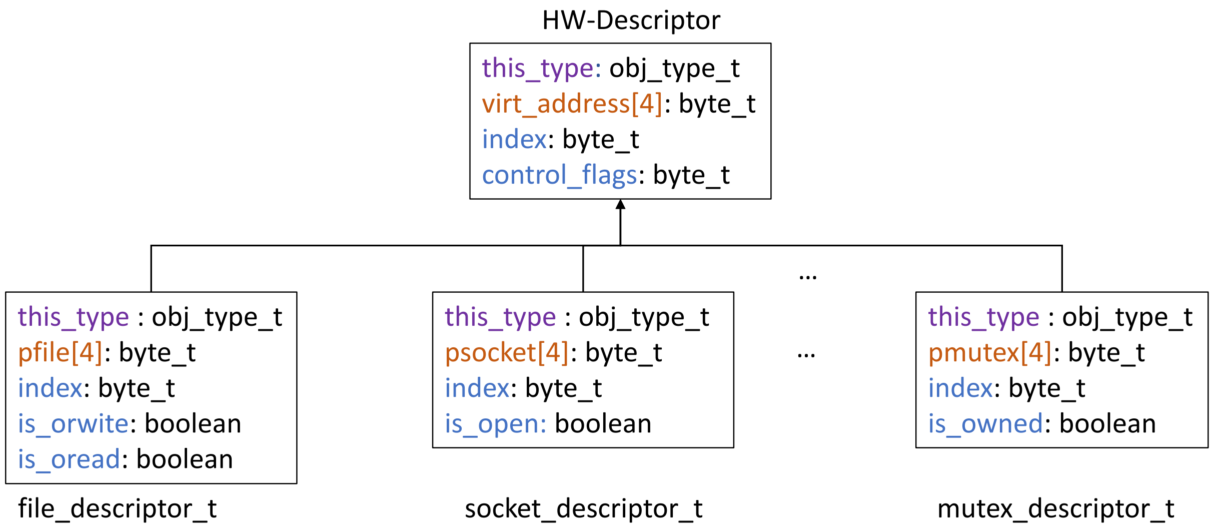


Figure 3.32: Hardware descriptor - record composition and specialized types.

In addition to the indexing feature, the *HW descriptor* is extended with additional fields that promote correct handling and attempt on minimizing interactions between the *Kernel Core* and the software in the host system. Such fields include: a type qualifier, to distinguish between different types of resource, or to identify an uninitialized or null descriptor; a Linux user-space memory address, associated with the resource in the file descriptor; and a field of state control flags, such as open or closed status, read or write permissions and whether it is blocked or free, among others, which can vary in format to comply with the descriptor states.

In the HAL-ASOS framework, the hardware descriptors are bound to memory objects that were previously instantiated in the target application, and submitted to the list of pooled resources in a specific *Task* class. Each *Task* class can be a software-based processing resource or a hardware-based processing resource that represents an HAL-ASOS accelerator. The *HW-Task* design must also create the required hardware descriptors as part of the local resources, to exchange information with the pooled object in the software application. Figure 3.33 shows an excerpt of the kernel package that includes the records of two hardware-descriptors used in the regular files and network socket files, respectively.

```
package hal_kernel_pkg is
...
type obj_type_t is ( null_obj, mutex_obj, semaphore_obj, conditional_obj,
            array_obj, fifo_obj, fstream_obj, vfile_obj, file_obj, net_obj,
            shd_mem_obj, npipe_obj, custom_obj );
...
type file_descriptor_t is
record
 this_type: obj_type_t;
 pfile: signed(C_MACHINE_WIDTH-1 downto 0);
 index: integer range -128 to 127;
 is_owrite:boolean;
 is_oread: boolean;
end record;
...
type socket_descriptor_t is
record
  this_type: obj_type_t;
  psocket: signed(C_MACHINE_WIDTH-1 downto 0);
  index: integer range -128 to 127;
  is_Open: boolean;
end record;
...
```

Figure 3.33: Kernel package - VHDL excerpt of the file and socket descriptor records.

The descriptors manipulation is performed using a control message protocol implemented at kernel-level in the accelerator model. Such protocol, is fundamentally based on three message formats, which are classified as query, remote execution and data transfer. Each type of message provides the means for the *HW-Task* to first make a query request about a certain descriptor. The query message returns an updated descriptor according to previously specified parameters. Subsequently, the *HW-Task* will be able to change the object's state through remote call messages, which allow to open a descriptor for reading or writing, or to increment a shared semaphore, among others. Finally, when applicable, the *HW-Task* will use data transfer messages to exchange data through the descriptor. For the effective exchange, it can use the LRAM or the SYSRAM storage, as the source or destination of data, according to the specified operation field *xcode*.

Figure 3.34 shows a message hierarchy implemented in the HAL-ASOS framework that is based on the three message types. A top of the hierarchy, stands a generic message that is composed of a *xcode* field that specifies an operation using a kernel package specific type, and establishes a message structure that includes a 3-byte control field and a 4-byte payload. The *xcode* field is common to all messages and is primarily used at software-level to establish a received message format. For the implicit message semantic, distinct messages implements specific fields, in the control and payload areas, following space limits dictated by the generic message.

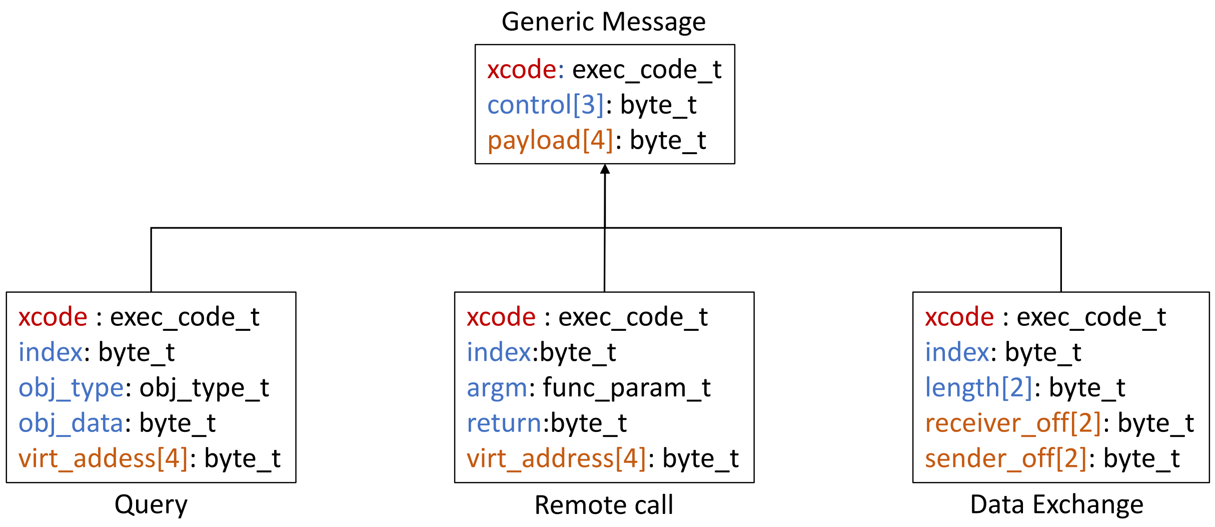


Figure 3.34: Kernel-level internal message hierarchy.

Regarding the query message, its structure matches the composition of the hardware descriptor and is mostly used to update the descriptors in the *HW-Task* context. A remote call message provides additional fields to update the descriptor status by using the appropriate *xcode* type and one additional function parameter. The software in the *Task* class will provide a return code that indicates the execution result of the internal handler. A data exchange message includes a 16-bit control field to specify the desired transfer length and can be used to indicate number of bytes or words, according to the *xcode* provided. In the payload fields, it provides a sender and a receiver offsets to be used when applicable, and one of them will refer the local storage. As in the previous transfer parameter, the specified value can be interpreted as word or byte offsets.

For efficient handling of the descriptors in the software application, the HAL-ASOS framework provides storage resources in each *Task* class that include a local descriptor table, and auxiliary computing resources used to respond to messages sent by the accelerator kernel and complete the requested operation. To perform the effective transfer of data, the same class can explore the different memory models which aim to minimize the impact on processing and on the memory footprint, by use of extended class profiles such as shared memory, user-space IO and zero copy.

To prepare the hardware descriptor, *HW-Task* must initialize the local register with the expected descriptor type and index. For this, the software application can provide the initial values using a task-level control message, or the design can assume a pre-established order that corresponds to the behavior of the software application. Figure 3.35 shows a diagram that describes the sequence of operations involved in a query message. The actions performed between *Task* class and *Kernel Core* simplify the joint operation involving the class and the HAL-ASOS file system.

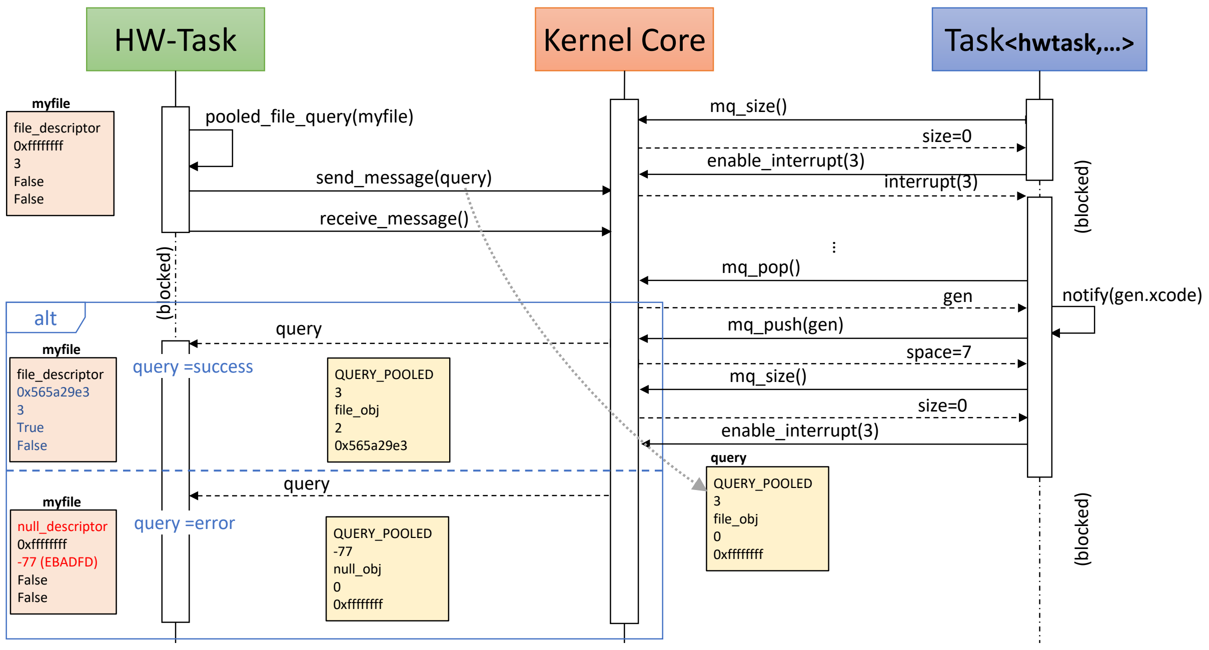


Figure 3.35: Query message - file descriptor query sequence diagram.

In the described example, *HW-Task* implements a hardware descriptor called *myfile*, used for handling storage files, where it specifies an *index* value of 3. By invoking the appropriate user procedure, the *HW-Task* submits the descriptor for status update. In its implementation, this procedure follows a synchronous message model, where it executes a system call to send the descriptor fields through a query message and suspends the *HW-Task* until it receives a return message. Alternatively, an asynchronous implementation could be used. A representation of the query message sent confirms the descriptor values, where a *xcode* field specifies the *QUERY_POOLED* command, followed by an index of 3 and a *file_obj* type is indicated. The *obj_data* field is set to 0 and no virtual address is sent. For this purpose, it is assumed the *Task* previously placed the processing resource on wait and activated the interrupt related to messages in the *Message Queue* resource.

Writing a message in the *Message Queue* triggers an interrupt to the host system, that concludes at the Linux kernel, with the reactivation of the processing resource. Once executing, it reads the message using *mq_pop* and returns to user space to execute the notify command. At completing the execution, the processing resource re-enters the kernel space to submit the response message, which contains the updated values. By receiving a response message, the *Kernel Core* completes the read system call and returns the received data to the user procedure. In its turn, the user procedure concludes by copying the message fields to the descriptor.

Depending on the result in the class execution, the descriptor can be returned with the updated values in case of success, and in this alternative representation, it can be seen that the descriptor was found in the class resources, at the specified index and is already open for reading. In case of error, the descriptor is returned with a null type and the *Index* field contains the error extracted from the Linux error standard. Implementation details of the user procedure can be seen in the excerpt of the user package, in Figure 3.36.

```
 10 package hal_asos_user_pkg is
    ...
646 --------------------------------------------------------------------------------------------------
647 procedure pooled_file_query(signal i_call : out kernel_input_t;
648                             signal o_response: in kernel_output_t;
649                             signal ufile_q: in file_descriptor_t;
650                             signal ufile_d: out file_descriptor_t)is
651 --------------------------------------------------------------------------------------------------
652 variable qmsg : kpool_query_m;
653 variable this_type:std_logic_vector(PARAM_OBJ_TYPE'range):=
654            std_logic_vector(to_unsigned(obj_type_t'pos(file_obj),8));
655 begin
656 i_call.reschedule<= '1';
657 i_call.enable_scheduler <= '1';
658 case o_response.sched_progress is
659   when 0 =>
660     qmsg.xcode    := std_logic_vector(to_unsigned(exec_code_t'pos(QUERY_POOLED),8));
661     qmsg.obj_type := this_type;
662     qmsg.virt_address := (others=>'0');
663     qmsg.index    := std_logic_vector(to_signed(ufile_q.index,8));
664     qmsg.obj_data := (others=>'0');
665 
666     send_message(i_call,o_response,qmsg);
667   when 1 =>
668     receive_message(i_call,o_response);
669     qmsg           := cast_return_to_query_message(o_response);
670 
671     ufile_d.this_type<= file_obj;
672     ufile_d.pfile    <= signed(qmsg.virt_address);
673     ufile_d.index    <= to_integer(signed(qmsg.index));
674     ufile_d.is_owrite<= std_logic_to_boolean(qmsg.obj_data(1));
675     ufile_d.is_oread <= std_logic_to_boolean(qmsg.obj_data(0));
676 
677     i_call.enable_scheduler<= '0';
678   when others => null;
679 end case;
680 end procedure;
681 --------------------------------------------------------------------------------------------------
```

Figure 3.36: User package - procedure to query for a file HW-based descriptor.

To receive the file descriptor data, the procedure uses the *ufile_q* parameter, as input on line 649, which has a connection with the output of the register implemented in the datapath of the *HW-Task*. A query message is implemented as internal variable on line 652, and on line 653 the file object type is translated to the equivalent enumeration. As already seen in descriptions of other procedures, the implementation makes use of the scheduler service to split the execution into two steps. In step 0 starting at line 659, the message is filled with the descriptor values, while at the same time, it is passed as a parameter in the system call on line 666. With the completion of the system call, the execution returns from the *Kernel Core* with the increment of the progress register in the scheduler service.

Upon returning, the scheduler service maintains the blocked state of the *HW-Task*, until completing the next system call. As such, the query procedure is repeated in step 1 with the invocation of the system call on line 668. The execution of the system call blocks until the reply message is received, and upon receiving an update, the system call is resumed with a return register that contains fields of the received message. The implementation makes use of a kernel function to extract the contents in the return register, as shown at line 669, that establishes an ordered connection with the internal variable *qmsg*. An alternative function that returns a descriptor of type *file_obj* could also be used.

The resulting connection is source of the *ufile_d* in lines 671 to 675, which is an output parameter in the procedure top-level at line 650. As such, this VHDL formal mechanism must connect with the descriptor register input that belongs to the *HW-Task* datapath, in order to provide the returning data. Upon deactivating the scheduler service, at the line 667, the procedure concludes releasing the *HW-Task*, that once active, writes the descriptor register with the values of *ufile_d*, and abandons the procedure by executing the next state in its control logic.

At this moment, the datapath of *HW-Task* stores updated descriptor that is open for writing operations. To proceed in the manipulation of such descriptor, the *HW-Task* must provide data from its processing using the LRAM, and request for their transfer using procedures which for this purpose can also be found in the user package. For a better use of computing resources, the design of the *HW-Task* can promote the parallelism of operations, taking advantage of the wait states in the return messages to initiate processing of the data used in the next write transaction.

In Figure 3.37, it can be seen a sequence diagram that describes the operations using a loop cycle, in which the design of the *HW-Task* asynchronously transfers the processing results using the hardware descriptor initialized in the previous diagram. In this example, the *HW-Task* starts with processing before reaching the main loop. Once there, it implements an asynchronously aligned write operation that initiates with the writing procedure and completes the cycle, by finalizing the write operation of the previously processed data. For that, the write procedure is divided in two independent implementations which can be distinguished by the *async* prefix, and the *conclude* suffix, respectively.

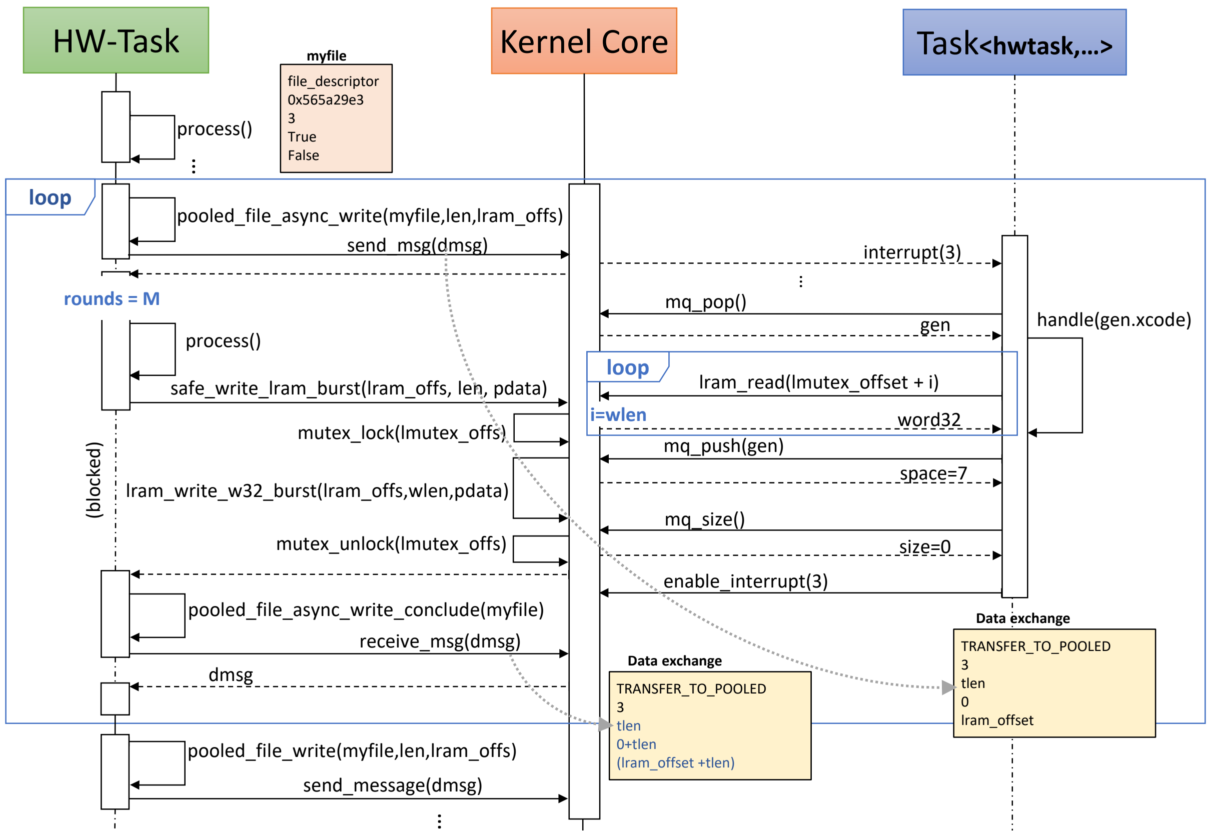


Figure 3.37: Data transfer message - file descriptor write sequence diagram.

By invoking the first procedure, the *HW-Task* design provides the location in the LRAM containing the previously processed data, which completes by sending a data transfer message. In the representation of the message sent, it can be seen the *xcode* value, which indicates a transfer from the LRAM towards the descriptor, and uses an 8-bit reference size in a *tlen* transfer length. The offset in the receiver is zero since the file is manipulated continuously and the offset in the LRAM has a value that results from previous processing states. Alternatively, a 32-bit word-based procedure could be used to maximize the transfer length field.

To avoid a long sequence diagram, greater detail was only used in the steps that occur inside the *loop* structure, where it can be observed that after completing the process stage, the *HW-Task* implements a procedure composed of three system calls to write the results in the local storage. In parallel, *Task* class will be able to read the LRAM data that was written before the *HW-Task* entering in the loop (not shown) and, in doing so, avoid an address collision. For the read operation, the class implements a *wlen* based loop without acquiring the *LMutex* exclusivity that at this moment may belong to the Accelerator's kernel.

With the completion of the user procedure, the *HW-Task* returns from the kernel and initiates a new processing round, where it writes the results in new LRAM positions, and finishes the write procedure of the previous data. In doing so, it implements the second part procedure to *conclude* the write operation, which returns a response message used to align the *HW-Task* processing according to the results from the software side. To minimize the number of operations, the *Task* class provides the offset values to use in the next transfer through the arithmetic represented in the return message, that is based on the number of bytes it was able to transfer. The *HW-Task* may abort in case of error, include partial results of delayed processing in the next transfer, or start a new processing round that initiate with the transfer request of the data processed in the previous iteration. After *M* processing cycles, the *HW-Task* design concludes writing the remaining data with a synchronous procedure, as shown in Figure 3.37. For completeness, the asynchronous procedures used in the main loop can be consulted attached in Listing C.31.

To conclude the descriptor manipulation, the *HW-Task* design can complete the sequence of logical operations that ends closing of the descriptor in the *Task* class. For this, the *HW-Task* invokes the *close* procedure in the user package, which is based on the exchange of *remote call* messages. Figure 3.38 shows the sequence diagram that describes the use of the close procedure.

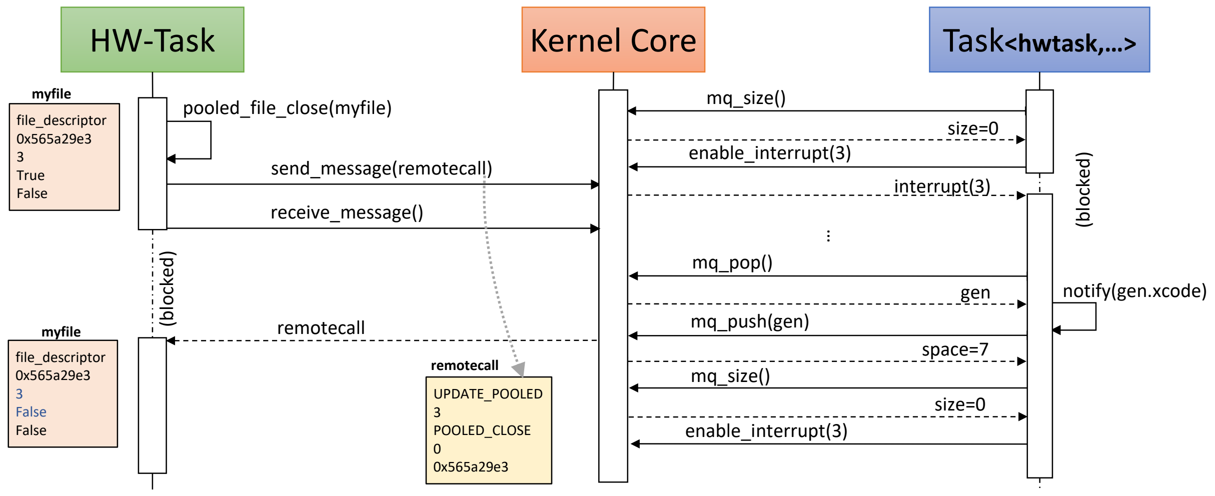


Figure 3.38: Remote call message - file descriptor write data sequence diagram.

In the representation of the sent message, it can be seen the *xcode* field that indicates a descriptor update. This *xcode* is complemented by the parameter *POOLED_CLOSE* to indicate the desired action, and the response field is set to zero. The final result of the procedure can be extracted from the descriptor representation, that updated the internal flags of read or write permissions to false, keeping the other parameters valid. With this, it allows the *HW-Task* to implement a new procedure to open the file using the descriptor it has or to reuse the descriptor to manipulate other files, by invoking a new query.

For completeness, in Figure 3.39 it can be seen an excerpt of the kernel package that includes the communication tokens used at kernel-level in the accelerator model. In line 108, the *exec_code_t* type is defined to provide *xcode* tokens sent in all message formats. The first token is a null code used in case of empty messages, the second is exclusively used in the query message, and the third token is used in remote call messages. Most of the remaining tokens are used in data transfer messages with exception of the last two tokens. A custom code is provided at line 111, for user definable features and must be complemented by providing the appropriate handle to the *Task* class. The last code, at the end of line 111, is used by the yield system call, and sent using a generic message with empty control and payload fields.

```
 10  package hal_kernel_pkg is
     ...
106  type exec_code_t is (NULL_EXCODE, QUERY_POOLED, UPDATE_POOLED, TRANSFER_TO_POOLED,
107  TRANSFER_FROM_POOLED, TRANSFER_FROM_POOLED_W32, TRANSFER_TO_POOLED_W32,
108  TRANSFER_TO_POOLED_SYSRAM, TRANSFER_FROM_POOLED_SYSRAM, TRANSFER_FROM_POOLED_W32_SYSRAM,
109  TRANSFER_TO_POOLED_W32_SYSRAM, TRANSFER_TO_HOST_SWFIFO, TRANSFER_FROM_HOST_SWFIFO,
110  TRANSFER_CONTROL_TO_DDS_TOPIC, TRANSFER_CONTROL_FROM_DDS_SUBSCRIPTION,
111  TRANSFER_DATA_TO_DDS_TOPIC, TRANSFER_DATA_FROM_DDS_SUBSCRIPTION,
112  TRANSFER_SYSRAM_DATA_FROM_DDS_SUBSCRIPTION, TRANSFER_SYSRAM_DATA_TO_DDS_TOPIC, TRANSFER_TO_STDIO,
113  TRANSFER_FROM_STDIO, TRANSFER_TO_STDERR, CUSTOM_CODE, TASK_YIELD);
     ...
278  type func_param_t is (POOLED_RELEASE, POOLED_LOCK, POOLED_TRY_LOCK, POOLED_SIGNAL,
279  POOLED_FORCE_SIGNAL, POOLED_POST, POOLED_WAIT, POOLED_TRY_WAIT, POOLED_OPEN,
280  POOLED_CLOSE, POOLED_GET_LEN, POOLED_SEEK, POOLED_ERROR);
     ...
```

Figure 3.39: Kernel Package - Kernel-level communication tokens.

When using the third token, i.e., *UPDATE_POOLED*, a remote call message combines it with additional *func_param_t*, that can also be seen at the bottom of Figure 3.39. Such additional parameters, in lines 278 to 280, instruct the *Task*'s processing resource about the required update feature, as it was seen in the close file example of Figure 3.38.

3.3 Hardware Kernel Interfaces

The accelerator model provides three interfaces that integrate the underlying HW in the target platform. In These: (1) the *S00_Control* interface is used in the control operations, (2) the *S01_Data* interface is used to exchange data with local memory (LRAM), and (3) the *M00_System* interface is used to address the system memory using the accelerator resources. Such interfaces are composed by a generalization of signals, which implement the Generic Bus. Using the generic interface, we abstract the design from the technology in the target platform and so promote design re-use. In doing so, distinct accelerator versions implement specific circuits that translate target platform bus to each interface. In addition to the top-level interfaces, the generic definition can be found on the Local-BUS in the *HW-Kernel* design, and therefore, on all peripherals that are accessed through this bus, promoting the exchange of data to a single format.

3.3.1 Generic Interface

In its conceptual model, the Generic Bus, or *Gen-BUS*, follows a Master-Slave model that allows several slaves coupled to a single master, or to several masters using an arbiter. Figure 3.40 depicts a connection between two generic interfaces, Master and Slave, illustrating an example of the signal distribution through the Slave implementation. With respect to Slave addressing, we can see: a N-bit wide *ADDR* register, that specifies the offset in bytes, as a parameter of the Slave interface; a 4-bit wide byte enable (BE) register that enables writing in the bytes of one word; and two signals, *WR_CE* and *RD_CE*, which enable the clock for data writing or reading operations. Typically, these two CE signals are common in the Slave interface internal circuits and the distinction is made by the exclusive connection of the chip-select (CS) line, which the Slave interface distributes according to the address map it implements.

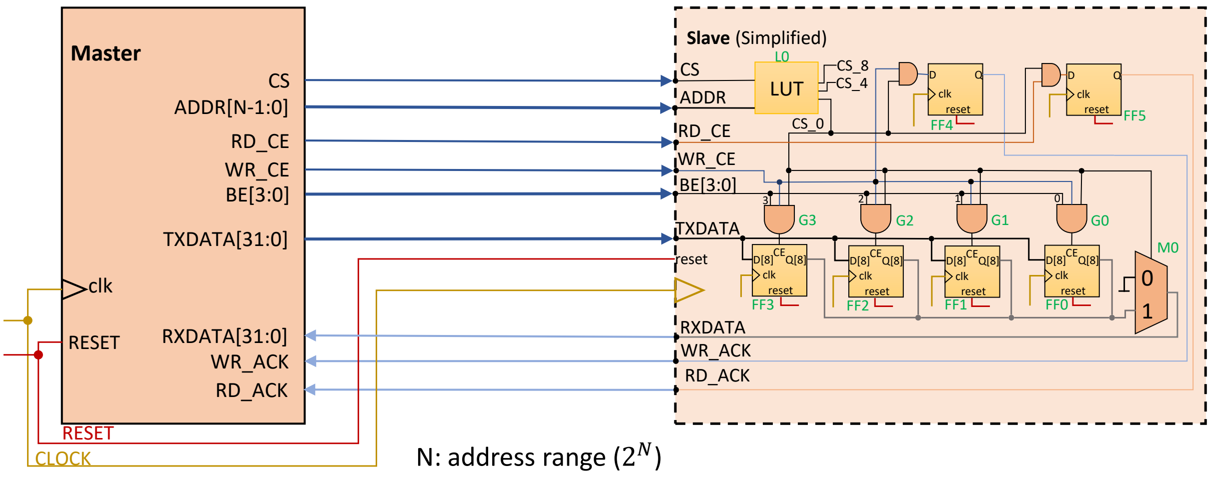


Figure 3.40: Generic Interface Master-Slave model

With regard to data exchange, two 32-bit buses are used, one in each direction to allow reading the written value during a single transaction. All transactions are confirmed by a handshake protocol that involves the corresponding *WR_ACK* and *RD_ACK* signals, both asserted for one clock period. The *ACK* signals, indicate to the Master that it is safe to change the contents of the *TXDATA* bus since it has already been written, or to read the contents of the *RXDATA* bus since it contains the stable data that results from the read transaction. With respect to the Slave implementation, the internal design is illustrative and in line with the style used by the framework. Here, we can see the encoding of three words of 32-bit, and by using L0, three select lines are extracted: *CS_0*, *CS_4* and *CS_8*. The logical combination of *CS_0* with the *WR_CE* input and the four bits of the *BE* bus, selects the individual bytes in the word zero for a write transaction, through G0 to G3 gating. The logic elements FF0 to FF3 represent 8-bit registers that receive their input through the TXDATA bus, aligned in a Little-Endian ordering in which the first 8-bits (7:0) are FF0 inputs, and consecutively the last 8-bits (31:24) are inputs at FF3. The next active clock transition, stores the TXDATA input bytes in the registers that have the CE inputs asserted.

The same clock transition gives rise to the *WR_ACK* signal using FF4. With the *CS_0* line active and by using MUX M0, all bytes of word zero are placed on the *RXDATA* bus, following the same ordering as the input. The combination of this signal with *RD_CE* input, gives rise to the *RD_ACK* signal in FF5, in the next active clock transition. In more elaborate designs, where the interface implements several internal offsets, usually the M0 inputs scale to the appropriate number, and each respective ACK signal can be combined logically using OR gates that connect to the single outputs of the Slave interface.

3.3.2 Multi-clock design

With the increase in complexity and the demand for multifunctionality, the design of FPGA-based systems evolved and grown into a multitude of systems that co-exist on a single chip, each representing an abstraction to a specific feature in the overall design. Distinct systems face specific time requirements and as such, require specific clock sources forming isolated clock domains. Ultimately, a network of multiple clock sources allows the designer to balance contradictory metrics such as computational power and energy efficiency.

The exchange of signals between logic circuits implies some degree of synchronism, and whenever a signal is sent to a logic circuit with a different clock source, this condition is known as clock domain crossing (CDC). Clock domains can relate to each other in frequency and phase differences, and if they belong to the same clock hierarchy in a design, they became synchronous and predictable. On the other hand, when domains belong to different clock hierarchies, they have asynchronous relationships that can be unpredictable. The CDC requires a set of precautions to ensure that time constraints of the logic elements in the receiver circuit are satisfied.

In most cases, it is common to adapt the design with synchronizing circuits, which fundamentally aim to contain the metastable state of the captured signal, for the appropriate time of stabilization. In a complementary way, it must also be ensured that signal events, especially of short duration, are detected by the receiving logic circuit. For this purpose, handshake protocols such as the ACK signals are used, in which the transmitter waits for a confirmation, before reading data or moving into a new exchange step. As synchronizers are typically diluted in design, they are sometimes a source of bugs due to miss-use or connection errors often motivated by the internal complexity with which they are associated, or due to the condition of a closed proprietary design.

To address some of these problems, the accelerator model adopts an asynchronous-synchronous design strategy to implement the generic interface. In such design, control signals are captured using synchronizer circuits, and are later used to unlock stable vector signals using MUX (es). While sending data from the Master circuit, the active control signals impose a steady value on vector signals which are only allowed to change after the handshake protocol is completed. On the slave circuit, MUX inputs are locked to prevent the propagation of unstable values, and are unlocked after the control signal capture time. In the opposite direction of data, same rules are applied between data vectors and the ACK signals. To reduce the possibility of miss-connection errors, the design applies the complete set of logical elements in a single isolated component that allows configuring different cases of connectivity through specific architectures.

3.3.3 Synchronizer for generic interface

The implementation of the multi-clock design strategy for the accelerator model is concentrated on the generic synchronizer component (*Sync_gen*) for the generic interface. To a better understanding of this component, one can refer to synchronizer variants found in the community: (1) the conventional multi-flop synchronizer and (2) a toggle synchronizer variant. The first one generally uses the receiver clock to capture an external domain signal and refer it to the local domain. For signals with the duration of a transmitter's clock pulse, the conventional synchronizer is not suitable due to the lost data phenomenon. For this reason, in the design of the synchronizer for the generic interface, we apply the (2) synchronizer circuit while enforcing the ACK-based handshake protocol. Implementation details about these two circuits can be found attached in Appendix B.

Figure 3.41 describes the architecture of the *Sync_gen* component using a logic element diagram. On the left side we can see the layout of the Master interface and it's set of signals, while the corresponding layout of the Slave interface is represented on the right side. One must notice an extra a *BUS_VALID* signal returning from the Slave circuit, that is used in the *Hold* circuits to store the control signals while waiting for availability on the interface. Normally, such signal is always active, except for the cases when the design is operating in multi-master mode. In that case, the BUS_VALID signal is controlled by an arbitrator device. We shall discuss this multi-master arbitration in the flowing section.

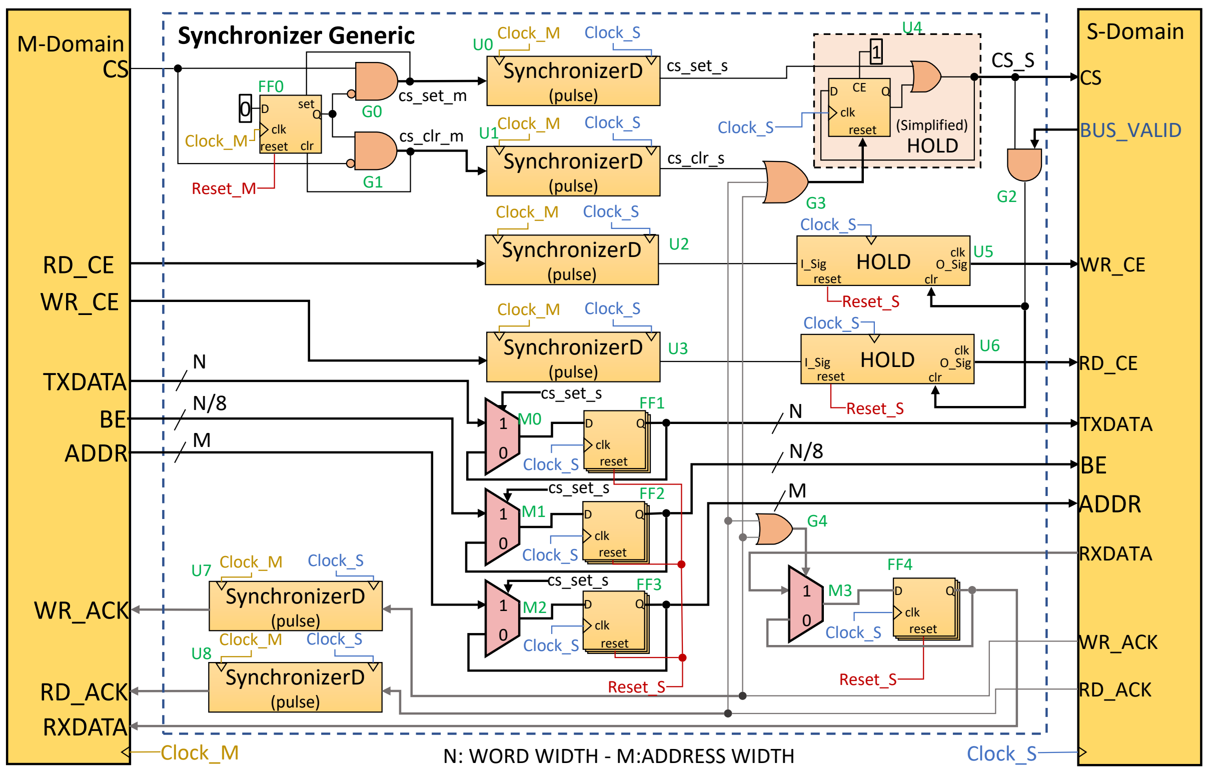


Figure 3.41: Synchronizer for the generic interface

In FF0 the CS signal from the master is decomposed into set and clear pulses using the logical combination with G0 and G1. The two pulses are referred to the Slave clock domain through the *synchronizers_dual* in U0 and U1. The set pulse signals the start of a transaction by activating the CS_S output that remains active through the Hold circuit. The same pulse is used in M0, M1 and M2, allowing to store the vector inputs in FF1, FF2 and FF3. The content of these buses remained stable since the beginning of the transaction and during the time of CS synchronization in U0. The corresponding clear signal at the output of U1, can disable this CS_S signal in the following clock cycles, if the Master gives up on the bus transaction.

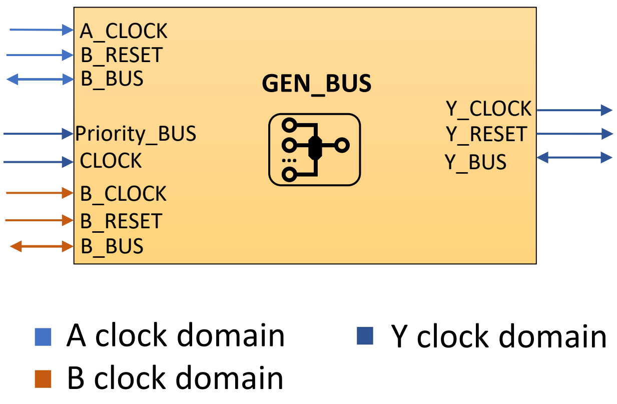
When one of two CE pulses is received in the Slave circuit, the output of U2 or U3 is hold stable by circuits U5 or U6. This allows to synchronize the CE signal with the CS_S signal if the latter has lost a clock pulse in the metastability region, but mostly it implements a pending transaction waiting for the arbitration. When the bus is available, the logic combination G2 determines the deactivation of the CE signals on the next active clock transition. In response to one of the two active CE lines, the Slave interface reacts by activating the corresponding ACK signal. One of these signals will select the contents of the RXDATA output through G4 and M3, being stored in FF4 until the next transaction, and G3 disables the CS_S output which anticipates the end of the transaction. Furthermore, the Master expects to receive the same signal through U7 or U8.

It should be noted that this type of connection does not meet the performance requirements of interfaces where transactions occur at the rate of a clock cycle. In such cases, the chosen multi-clock strategy establishes that the design accommodates the clock hierarchies to favor the performance of the interface. For this, it merges the Slave interface with the Master's clock domain. Therefore, this is a choice that in some cases degrades the design's clock cycle or energy efficiency metrics. When the Slave interface belongs to the Master clock domain, it is possible to select a dedicated architecture that without making changes to the established design, it suppresses the synchronizer circuits. The layout of this architecture can be found attached in Figure B.5.

Alternatives to this merger would include to intermediate forms of asynchronous FIFO storage. These components are more complex circuits, which take advantage of specific FPGA blocks such as true dual-port RAM, using gray code numerical systems for address control. With these coding systems, the scalar progression of the address register, varies at the rate of one bit and therefore, conventional synchronizers can refer this value to the clock domain of the control unit in this component.

3.3.4 Multi-master design

In multi-master topologies the generic interface requires implementation as an active component. This component, will handle synchronization, signal connectivity and schedule Master transactions to distribute access to the Slave interface. For this, it makes use of an internal arbiter device that implements two distinct scheduling policies. In Figure 3.42 we can see the Top-level of the *Gen-BUS* component for a connection using two masters, A and B and the Slave interface Y. On the right side of the figure, one can see the truth table that determines the scheduling between A and B. The conditions that determine a scheduling action are the inputs: *Priority*, from priority bus; *ReqB* and *ReqA*, from each of the CS lines; and the internal register *BusMaster* that establishes the active master on the bus. The output of the schedule policy is the *NextMaster* register that is source of the *BusMaster* when the scheduling actually takes place.



Policy	Priority	ReqB	ReqA	BusMaster	NextMaster
Static	1	X	X	X	A
	2	X	X	X	B
	3	X	X	X	A
Round-Robin	0	0	0	X	IDLE
	0	0	1	IDLE	A
	0	1	0	IDLE	B
	0	1	1	IDLE	A
	0	0	1	A	A
	0	1	0	A	B
	0	1	1	A	B
	0	0	1	B	A
	0	1	0	B	B
	0	1	1	B	A

Figure 3.42: Generic Bus component - The top level and schedule policies

As it can be seen from the figure, the internal arbiter of this component, implements two scheduling policies that are based on: (1) static priorities, and (2) round-robin. In doing so, the static priority replaces the round-robin policy with the highest priority in the system. Here the least significant bit of the Priority Bus determines the highest priority found regardless of the value in other inputs. Usually, priority sources are extracted from external auxiliary components that implement mutual exclusion such as the HW-Mutex.

When no priority signals are active, the next master is determined by the round-robin policy. Therefore, this new value depends on the *ReqA* and *ReqB* inputs and the value in the current *BusMaster* register. This policy determines that a master can access the bus in consecutive transactions while it is the only active master on the bus. When concurrency scenarios are observed, the arbiter distributes access to each master alternately, putting the master that lost the dispute on hold. Each transaction is also subject to a configurable time limit in number of clock cycles. If the bus master exceeds this time, an internal timeout event is generated that interrupts the transaction. As response, the arbiter re-schedules the next master and the one that was withdrawn will have to repeat the transaction.

In Figure 3.43 we can see a diagram that describes the internal implementation of the Arbiter for the Generic Bus component (*Gen-BUS*). The request and priority inputs are the stimulus for the two scheduling policies implemented using U0 and U1. Although in concurrent mode, the source selection in M0 determines that the value of *next_master_i* signal is established by the Round-Robin encoder, only if the priority lines are at rest. Under this condition, the signal prx_activity_i from U1 has the logical value zero that using M0, selects the source of U0 as the next master.

The design of the U0 encoder is purely combinational where the input is concatenated with the corresponding numerical value of the current master to form a new address value. Such address, is then used

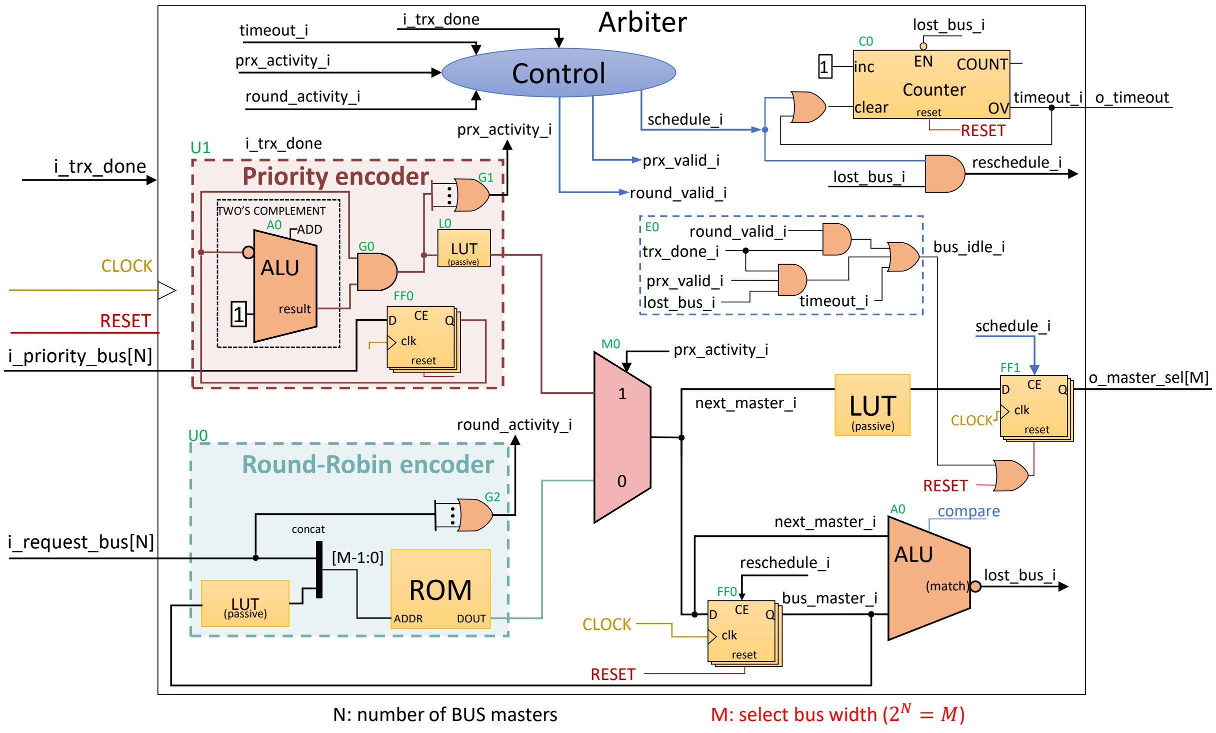


Figure 3.43: Generic Bus Arbiter - internal architecture

to determine the value for the next bus master using a program in ROM. Here, the LUT transformation aims to convert the corresponding value of the FF0 register into a corresponding numerical value. Depending on the number of masters, this circuit can be removed during the implementation of the design, since the advanced data type for the bus master results from an enumeration.

The implementation of the U1 encoder is also combinational and results from the *AND* logic using the priority at the input with its equivalent two's complement. The result returns the least significant active bit in the input. Using an input register ensures the constant value of the *priority_bus* signal for at least one clock cycle. If at least one input is active, the value of *prx_activity_i* is set to the logical one, which using M0, selects the output from this encoder as the source for the next master.

The value of *next_master* is compared with the value of BusMaster using ALU A0. The result of this comparison gives rise to the active *lost_bus_i* signal if the difference between the two is valid. With this signal active, counter C0 starts, which may give rise to the timeout event if the current transaction does not complete in 8 clock cycles. Such number of cycles is a parameter in the accelerator settings in the *hal_asos_configs* package.

A scheduling action is only triggered after the transaction has completed, either in response to the *trx_done_i* signal or the timeout event. At this moment, the logical combination E0 becomes active and forces the output *mst_sel* to assume idle state during a clock pulse, at FF1. In response, the Control activates the *schedule_i* signal to trigger the scheduling of the next master. This next master can be the same one when the bus reaches the idle state, between two consecutive transactions. Otherwise, the combination of *schedule_i* with the *lost_bus_i* signal, gives rise to the *reschedule_i* signal that enables writing to the *BusMaster* register, in FF0. Therefore, the same value will also be written in FF1 in the next clock cycle, after completed the idle state.

The control unit for this component is divided in two distinct flows that represent scheduling policies. The state logic that implements this control unit can be consulted in Figure 3.44. After a reset signal has been asserted, the Control assumes state #0, where it activates the *schedule_i* signal to generate a bus master. If the *prx_activity_i* input is active, the control assumes state #1 until it establishes synchronism with the beginning of the transaction. As soon as the transaction starts, the control unit assumes state #2 and remains in this state for consecutive transactions, until the *lost_bus_i* and *trx_done_i* signals are asserted. It then completes the priority-based logic sequence by returning to state #0 which gives rise to a new bus master. Alternatively, if the *timeout_i* signal is asserted, the control unit likewise leaves #2 to originate a new master in #0. Such signal, is generated by the counter C0 in the datapath, which starts counting when the *lost_bus_i* signal is asserted, and overflows before receiving a high value from the *schedule_i* signal, which is asserted in state #0 (see also Figure 3.43).

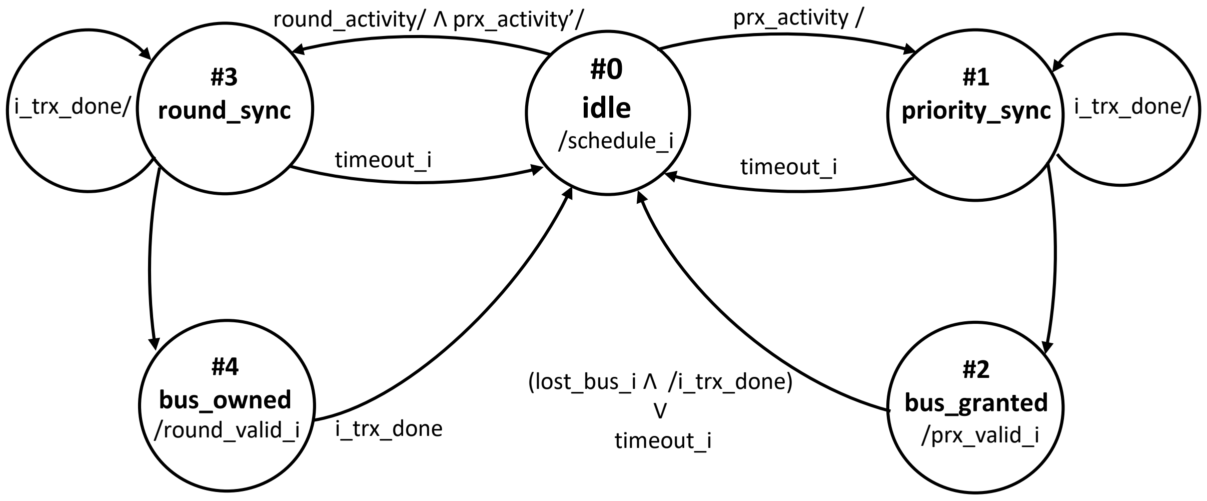


Figure 3.44: Generic Bus Arbiter - internal architecture

State #3 is assumed when once in #0, there is no active priority and there is activity in the round-robin encoder. Similarly, the control remains in state #3 for synchronism before switching to state #4. As long as a concurrent transaction does not arise, control assigns the bus to the current master, until it terminates and abandons access, or another master initiates a transaction. In this scenario, the control distributes the access among the active masters in a sequence of states of #0, #3 and #4, for each Master transaction.

The arbiter architecture is extensible to multiple masters, but the accelerator model uses no more than two. Regarding scalability, the design of this component is made less efficient by the resource consumption of the U0 encoder. The chosen alternative would be a synchronous ring architecture, using a binary token. When compared, the limit of two masters in the design favored the combinational architecture that is faster and still efficient.

In Figure 3.45, we can see a diagram that describes the internal architecture of the *Gen-BUS* device. The two master-type inputs, A and B, are routed through components U0 and U1 for synchronization purposes, giving rise to the internal signals that connect to M0 and M1. The MUX (es) establish the connectivity between the Slave Y-BUS interface and the master chosen by the Arbiter. The result of this choice is received via the *mst_sel_i* signal, which is also used in L0 to give rise to the *BUS_VALID* signals specific to each master.

When no master is connected to Y, this slave interface receives zero values on all signals. This is an idle state that is achieved whenever a master is disconnected and lasts at least one Y clock cycle, before another master takes its place. The beginning of a new transaction is determined by the logical value zero on the line *trx_done_i*, as result of the active CS line on the Y-BUS. The upward transition of this signal gives rise to the *cs_rise* signal in the *SynchTRX* component, which in the next clock cycle disables *trx_q* in FF1.

Upon receiving an ACK signal, the value of the *trx_done_i* assumes the high logic value through G2, and the same signals are used to reset the logic value to FF1 in the next clock cycle. In response to this high signal, the bus reaches the idle state, which by turning off the CS line establishes a low logic value in FF0 at the next active clock transition. If, on the other hand, there is a delay in the reception of ACK and a timeout occurs, this signal resets the value FF1, at the same time that Y reaches the idle state again.

The different variants that can be found for the *Gen-BUS*, specify distinct architectures on the *Synch_gen* component which aims to accommodate the relationships between the clocks of the Master and the Slave interfaces. The available choices allow a design using a single clock domain, the Y domain belonging to A or B, or ultimately, three distinct clock domains. In Figure 3.46 shows a timing simulation wave diagram of

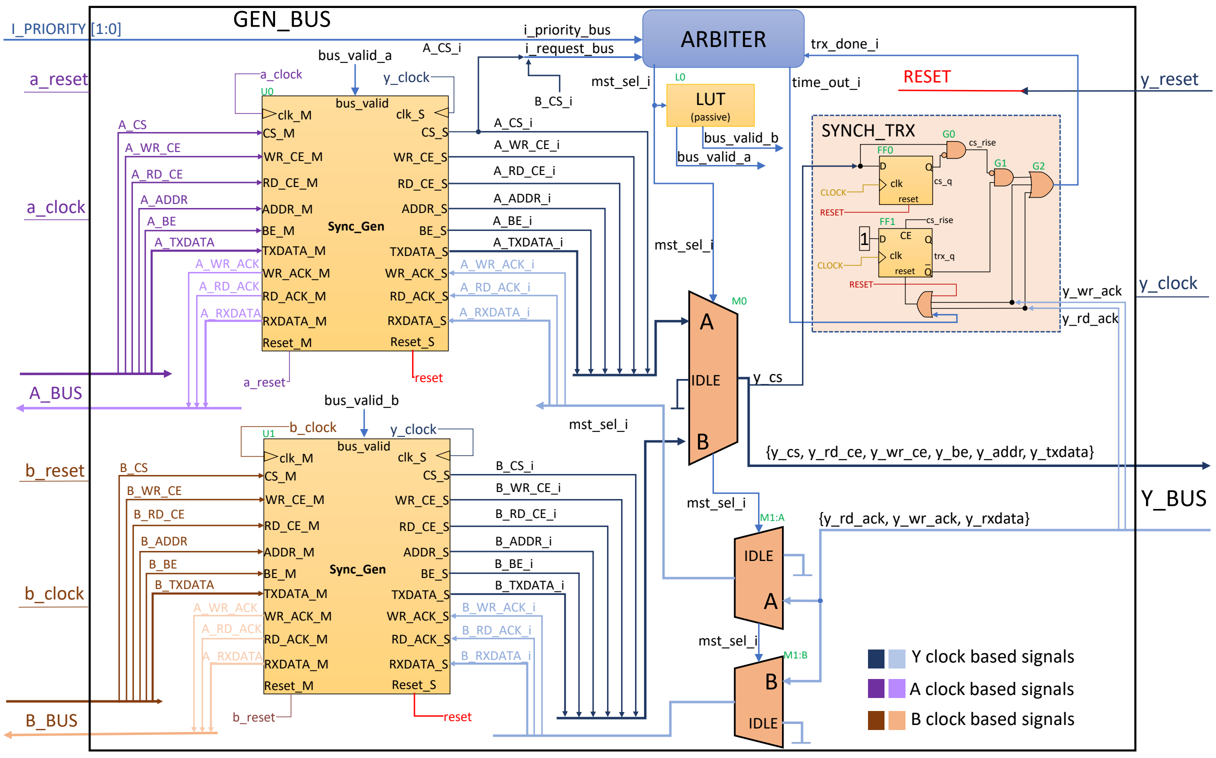


Figure 3.45: Generic Bus component - internal architecture

the *Gen-BUS* component, using an architecture where the Y clock belongs to the Master B clock domain. This diagram expresses a temporal simulation that is based on the resulting synthesis of the component. Here, the clock of Master A has a period of 6.3 nanoseconds and the same clock of B and Y, have a period of 10 nanoseconds.

The *priority_bus* input takes the value of “2” at time instant of 170 nanoseconds specifying B as the priority master. At time instant of 197 nanoseconds, we can see that A and B compete simultaneously for the bus, but B proceeds with two write transactions at offsets 0x00 and 0x04, with the duration of 2 clock cycles each. As Y belongs to the domain of B, and since the priority is established, the transaction proceeds with no delay. The same priority is revoked at time instant of 240 nanoseconds, and B still performs a read transaction of the offset zero. From the values in the *mst_sel* output, we can see that source ”2” was selected three times in a row, alternating with source “0”, for an idle cycle between each transaction.

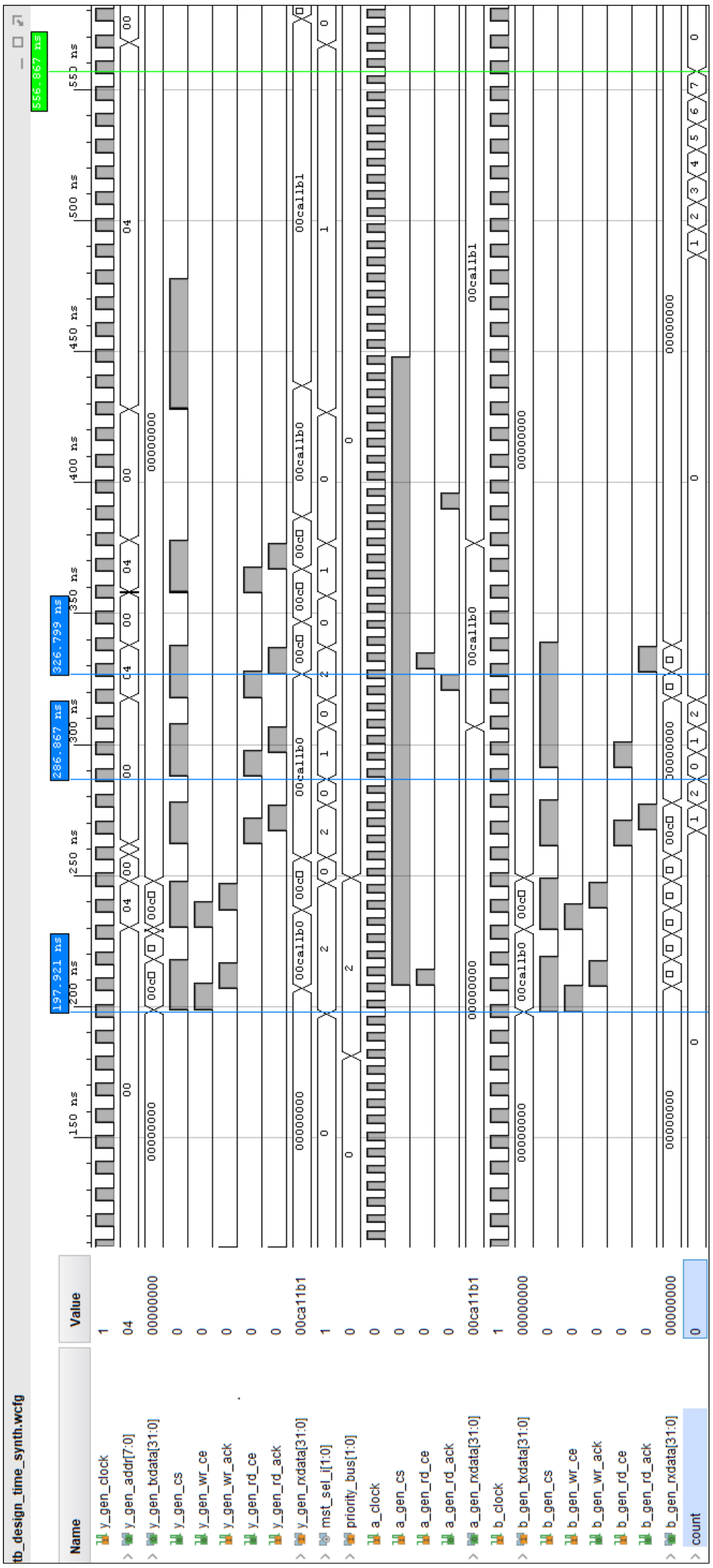


Figure 3.46: Generic Bus wave plot - timing simulation using A and B interfaces.

At time instant of 286 nanoseconds, master A starts the Y reading transaction, with an 83 nanoseconds delay. The transaction lasts two clock cycles from Y, while B is put on hold. At completion of A transaction, B proceeds in a second read at offset ”4”, while A waits for confirmation. The acknowledge is received at time instant of 326 nanoseconds, and Master A initiates a new read operation 6.3 nanoseconds later. This transaction reaches interface Y at time instant of 350 nanoseconds and is committed at interface A at time instant of 380 nanoseconds. At this moment Master A violates the protocol and keeps the CS line active while delaying the CE line. This condition leads to a timeout that occurs at the time instant of 556 nanoseconds, interrupting the transaction.

3.3.5 S00 Control Interface

The *S00_Control* interface implements a control-oriented channel that allows the host system to access the accelerator register area. This memory region maps all the HW resources in the accelerator model, with exception of the local memory that qualifies itself as a data flow-oriented storage space. For this reason, the local memory can be accessed using a dedicated channel provided by the S01 interface. This design choice aims to differentiate data manipulation, and opens the possibility of mapping this address space through a specific bus that is suitable for exchanging data using word-based transfers. In doing so, the accelerator model can benefit from a dedicated control channel if and when supported by the target platform. This condition allows flexible choices in the clock source of this interface in adequate ratio to the clock used by the CPU cores in the host system, and at the same time deviate this control information from the usual intense data flow of the main system bus.

Fundamentally, this control channel allows the accelerator to implement synchronization mechanisms at the application and operating system levels. For this, the interface allows reading or writing operations that target: the accelerator control and status registers; two FIFO-based bidirectional channels for exchanging control messages; two HW-Mutexes that allow the operating system to implement exclusivity mechanisms; and a local interrupt controller to establish the synchronization between the accelerator and other processing entities in the host system. In addition to these, the S00 interface allows to initiate transfers using the ZeroCopy unit, and access the performance metrics area of the accelerator. To map these resources, the S00 interface demands for an address range of 512 bytes or 128 words, thus nine address lines are required. To organize the design, we decided to split decoding in two levels that page the address range using 16-word memory blocks. Figure 3.47 describes such memory layout where it can be seen the 8 pages and their correspondence with the HW resources.

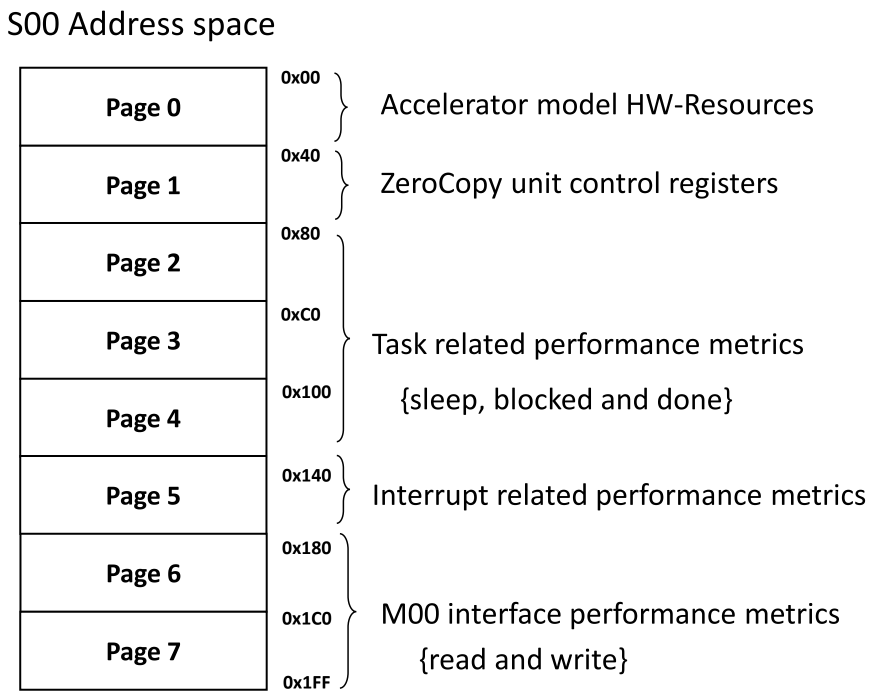

S00 Address space
Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7
0x00
0x40
0x80
0xC0
0x100
0x140
0x180
0x1C0
0x1FF
Accelerator model HW-Resources
ZeroCopy unit control registers
Task related performance metrics
{sleep, blocked and done}
Interrupt related performance metrics
M00 interface performance metrics
{read and write}

Figure 3.47: Slave decoder component- internal architecture.

In this two-level address design, each logical unit is responsible for page decoding in the range where it is mapped, and so, page 0 decoding is implemented by the *HW-Kernel* to address all local resources in the accelerator model. Similarly, the ZeroCopy unit is responsible for page 1 decoding to address its internal registers. In the performance metrics (i.e., pages 2 to 7), we use the same design in the 6 pages of memory, where identical offsets are used to address the internal registers that each unit implements.

Table 3.3 lists the offsets used in page 0 to map the accelerator model resources. To access this page, the host system must compose an absolute address starting from the base address of the *S00_Control* interface. This value usually results from the choices between bus types, bus hierarchies and ultimately the technology in the target platform. Then it must add the page offset and the register offset within the page. Although two offset fields can be seen in the table, the host system must always consider the byte offset to compose the desired address. The word-based offset is used internally in the interface design to select the target register.

Conceptually, all registers can be read and written, and the cases which only one operation applies are referred in the description column. The read transactions involving write-only locations will be accepted by

Table 3.3: The S00 interface - Page 0 internal register mapping.

Page offset	Internal register	Word offset	Byte offset	Description
0x0	Kernel control	0x0	0x0	Kernel control register.
	Kernel status	0x1	0x4	Kernel status register, read-only.
	LINTC control	0x2	0x8	Local interrupt control register.
	LINTC status	0x3	0xC	Local interrupt status register.
	MQueue space	0x4	0x10	Message Queue space and size registers,
	MQueue size	0x5	0x14	read-only.
	Data FIFO space	0x6	0x18	Data FIFO space and size registers,
	Data FIFO size	0x7	0x1C	read-only.
	MQueue Output	0x8	0x20	Message Queue output register, read-only.
	Data FIFO Output	0x9	0x24	Data FIFO output register, read-only.
	MQueue Input	0xA	0x28	Message Queue input register, write-only.
	Data FIFO input	0xB	0x2C	Data FIFO input register, write-only.
	Sysram address	0xC	0x30	Sysram base address register.
	Sys.Call errors	0xD	0x34	HW system call error counter, read-only.
	SysMutex status	0xE	0x38	Sysram dedicated HW-Mutex register.
	LMutex status	0xF	0x3C	Local RAM dedicated HW-Mutex.

the interface and return the control word 0xfe11dead. Read-only locations do not have connectivity with the *WR_CE* signal at internal level, but the transactions are confirmed accordingly in the next clock cycle.

Table 3.4 lists the offsets used in page 1 to map the ZeroCopy unit internal registers. The ZeroCopy unit is a component that provides services for the HAL-ASOS file system on Linux, in alignment with the zero-copy strategy of Linux OS. Optionally, these services can be integrated into the accelerator model using this unit in a shared mode. In order to avoid an access collision between host and *Kernel Core*, the internal registers of this unit are accessed in protected mode. This mode imposes that it is only possible to write in the register area after ensuring the exclusivity in the HW-Mutex of the accelerator model.

The description column, shows that only the status and error counters are not included in this register class, as they are dedicated to the host system. After a successful exclusivity step, the host system will be able to write the lock ID register, which will close this unit to other entities. Thus, it acquires writing privileges in the parameters area and control register. The parameter area allows to configure the offset in the LRAM, the length of a transfer and the physical address in the main memory. After this configuration step, the host system will be able to initiate a transfer through the control register, and finish checking the result in the status register. In the last area, the host system can check the total errors that may occur during writing or reading transactions since the unit started operating. Details about the implementation

Table 3.4: The S00 interface - Page 1 internal register mapping.

Page offset	**Internal register**	**Word offset**	**Byte offset**	**Description**
0x40	ZCU Control	0x0	0x0	ZeroCopy unit control register (WP).
	ZCU Status	0x1	0x4	ZeroCopy unit status register, read-only.
	Lock ID	0x2	0x8	Lock ID register (WP).
	LRAM offset	0x3	0xC	LRAM offset register (WP).
	Transfer Length	0x4	0x10	Transfer-length register (WP).
	MBUS Address	0x5	0x14	Main bus address register (WP).
	-	0x6	0x18	
	-	0x7	0x1C	
	M00 WR Errors	0x8	0x20	M00 write errors counter, read-only.
	M00 RD Errors	0x9	0x24	M00 read errors counter, read-only.

(WP) - write-protected registers.

of this protected mode, as well as the internal architecture of the unit, will be discussed in section 4.6.

Table 3.5 describes the internal offsets used on pages 2 through 7 to map the registers in the performance metrics area. This area allows the mapping of five event counters, used to measure performance metrics such as: sleep or blocking time, or execution of the *HW-Task*, in total or in execution rounds; host system latency in responding to accelerator interrupt events; or performances in accessing the system memory interface while writing and reading data, respectively. The contents of these counters are exported to the user space on Linux using the HAL-ASOS FS.

Each page is mapped using one of the offsets indicated in the first column, and implements the registers that can be seen in the second column. The vast majority of these registers are 64-bit wide and as such, they are mapped through word zero (W0) and word one (W1). Only the count totals register has a length of 128-bit and as such, is mapped using 4 words. With the exception of the control, all remaining registers are read-only. Writing to this register will activate or suspend the unit, or clear the stored contents, placing them in the initial values. Optionally, the design of each unit can be configured without clock cycle metrics. In this case, only the event counter and the control registers are implemented. Reading the remaining registers will return the control word 0xfe11dead.

With the adoption of a single data exchange format, the addressing of hardware resources follows the generic interface design considerations. As described above, the *S00_Control* interface splits the available address range using 8 pages of 16 words length, and in pages 0 and 1, we can find resources that are at the service of the *Kernel Core* and as so, they are also mapped in the Local-BUS address space. These

Table 3.5: Performance Counter internal register offsets.

Page(s) offset	Internal register	Word offset	Byte offset	Description
0x80 0xC0 0x100 0x140 0x180 0x1C0	Event Counter W0 Event Counter W1	0x0 0x1	0x0 0x4	Event counter register, 64-bit wide {W1,W0}, read-only.
	Clock Counter W0 Clock Counter W1	0x2 0x3	0x8 0xC	Event duration clock counter register, 64-bit wide {W1,W0}, read-only.
	Max Clock count W0 Max Clock count W1	0x4 0x5	0x10 0x14	Maximum count achieved register, 64-bit wide {W1,W0}, read-only.
	Min Clock count W0 Min Clock count W1	0x6 0x7	0x18 0x1C	Minimum count obtained register, 64-bit wide {W1,W0}, read-only.
	Total Clocks W0 Total Clocks W1 Total Clocks W2 Total Clocks W3	0x8 0x9 0xA 0xB	0x20 0x24 0x28 0x2C	Total clocks in successive events register, 128-bit wide {W3,W2,W1,W0} read-only.
	Counter Control	0xC	0x30	Performance Counter control register.

resources, in turn, were designed for concurrent access of two distinct masters, and for that, they provide specific interface connections labeled A and B. To implement resource addressing in the accelerator model, this interface uses the Slave and Page decoder components as part of the HAL-ASOS framework. These are associated with each other following a hierarchy from one Slave decoder to eight Page decoders. Details about the resource addressing or the Slave and Page decoder design considerations can be found attached in Appendix A.

In the multi-clock accelerator versions, a generic synchronizer component is used to connect the control interface (i.e.,*S00_Control*) to the local resources in the HW-Kernel design. Since most of these are at the service of the *Kernel Core*, they belong to the same clock domain of this unit. In doing so, output signals from the synchronizer are connected to the Slave decoder forming a single-master bus at the service of the host system.

3.3.6 S01 Data Interface

The S01 is the accelerator fundamental interface, characterized by the intensive use while exchanging data between the main system memory and the local storage on the accelerator. With this interface, the accelerator model distinguishes application-oriented data, such as the processing results, from control-oriented data used for synchronizing and controlling the application or the accelerator model. In doing so, for the overall system performance, it is of utmost importance that S01 is implemented using adequate connectivity with the system bus and a clock cycle that suits the target platform memory interface.

The S01 has a parameterizable address range that is used by the accelerator design to establish the effective size of its local storage. Such parameter is the *s01_addr_width* and the default configuration preset its value in 10 bits, which corresponds to 1 Kbyte of local storage. Optionally, this width can be increased until 16 bits, or 64 Kbytes of storage, a limit established by the design of the *Kernel Core*. The selected width maps the local memory (LRAM) as exclusive slave peripheral in this interface.

In the accelerator model the S01 interface is also used to establish connection between the ZCU and the LRAM which can have distinct connectivity attributes in terms of clock domain crossings. For that, specific accelerator versions target specific memory models that aim to balance the host CPU effort in the task of data movements. In this way, the S01 operates in multi-master mode with the existing CPU cores connecting as master A and the ZCU connecting as master B. Figure 3.48 shows a schematic that exemplifies this connectivity.

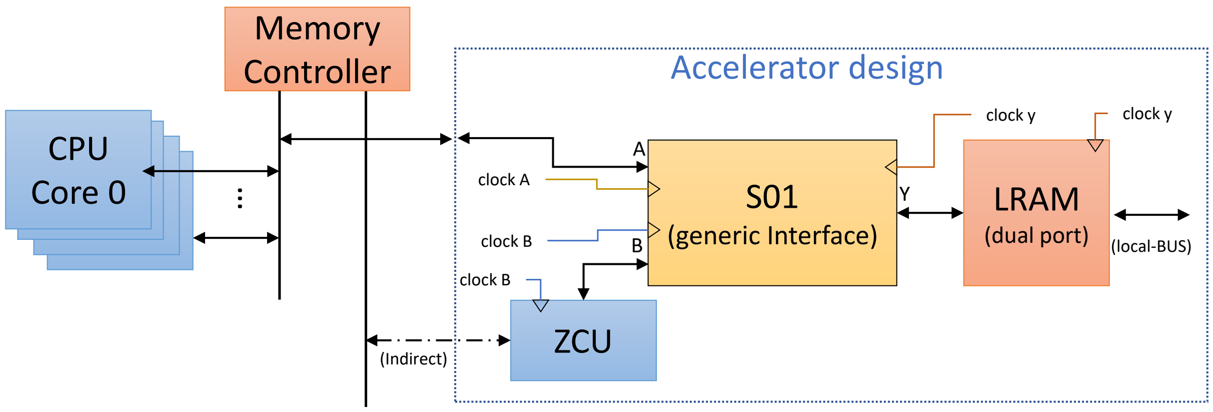


Figure 3.48: S01 interface - multi-master connectivity.

To consolidate each model requirements, the B, D, and E, F terminated string versions of the accelerator are distinguished in the clock domains and interface capabilities. In the E versions, the accelerator design extends the master A clock domain to slave Y, to favor the transfers performed by the CPU of the target platform. And in the F versions, the accelerator design extends master B clock domain to slave Y, to favor transfers performed by the ZCU. Ultimately, in version C and D, each master operates on equal terms and using a single clock domain. In section 4.8 we shall discuss each accelerator version in more detail.

3.3.7 M00 System Interface

When the accelerator model needs to manipulate data beyond the dimensions of the local storage, or perform data movements between the application memory segment and the LRAM, the M00 interface provides access to the system's memory using the appropriate HW system calls. Each accelerator will be able to manipulate a memory region designated as SYSRAM, and for that, its pre-allocation is required by use of the numerical parameter *sysram pages* in the accelerator configuration interface.

Such parameters are evaluated by the HAL-ASOS file system during the host OS initialization phase. It then identifies the accelerators that require access to this memory region and triggers the necessary operations to carry out an appropriate memory allocation. Each allocation concludes by returning a valid memory reference that the file system makes available through a specific entry file in its virtual structure. These memory references are then translated to a physical address and the file system concludes by registering each address in the correspondent accelerators.

The registry operation of the *SYSRAM* physical address on the accelerator activates an internal flag that allows the microprogram to proceed in the execution of the correspondent HW system calls. All HDL procedures that manipulate this memory, implement system calls that are based on offset parameters within this region. During each operation, the *Kernel Core* composes the target address by adding the offset parameter to the physical address using the *SYSRAM_ADDRESS* register. In practical terms, the Parameters length in the system call HDL record establishes limits of 18-bit word offset parameter, and a 10-bit transfer word length parameter. In this way, the maximum addressable size in this memory region is set to 1024 kB, and consecutive transfers are possible using a page range of 4 kB.

By choosing the appropriate accelerator version, it will allow the designer to improve this region throughput against the overall consumed resources. The F terminated string version, provides specific implementation that interfaces this memory region using burst mode transfers. In this mode, the M00 interface carries out sequential data transfers that require a base address and a transfer length. It takes on a higher performance format since the handshaking steps on the bus are implemented for the first word only. Depending on the target system a transfer of N sequential words may be fragmented into N / B bus transfers, where B is the maximum number of words in the burst mode for the particular bus technology. Despite the improved performance of burst transfer format, the M00 interface imposes latencies much higher than LRAM, since it is connected to a more complex memory interface and can experience delays due to bus internal schedule. The overall advantage arises by combining this with the use of implementation profiles in the software side. These will allow to map this region in the application memory segment. Under such condition, the application will be able to produce or reuse data in a memory region within the range of the design in the accelerator. In similar way to the previous interface, the M00 is used in multi-master mode where the *Kernel Core* is the master A and ZCU is the master B. Figure 3.49 depicts a connection diagram of the M00 interface.

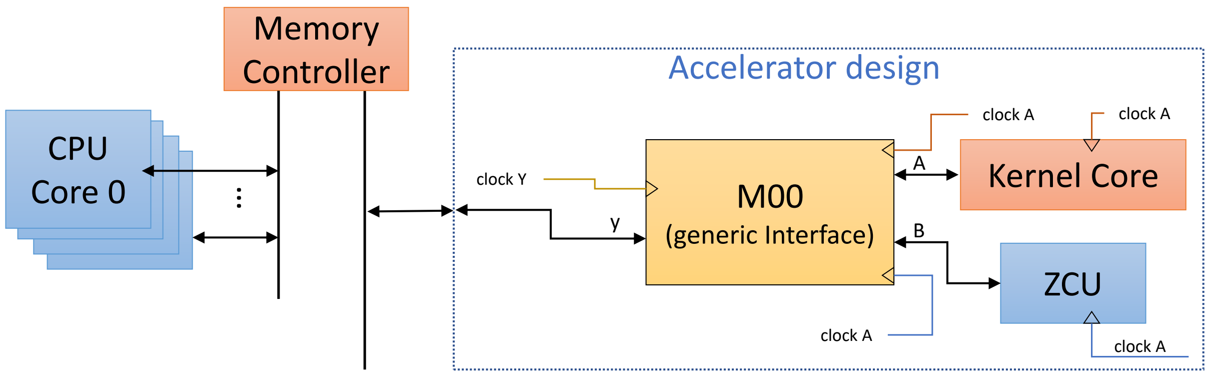


Figure 3.49: M00 interface - multi-master connectivity.

In doing so, the M00 interface is implemented using the generic bus component and since these two masters share the same clock domain, the same configuration is used in all accelerator versions. For this, clock Y is the source of clock A that connects to the *Kernel Core* and *ZCU* units, and consequently is distributed across the accelerator design. Fundamentally, the M00 interface follows the considerations of the Generic interface and performs data exchange using this format. Additional signals required that are beyond the generic interface definition are capability-based control signals that can be connected in parallel on both masters.

To handle greater complexity in bus transfers, the M00 provides a control channel that trigger the transfer on the system bus. Table 3.6 lists the top-level of the M00 interface, where two logical group of signals can be seen. At top, the data channel contains the signals that regulate data exchange within the accelerator model. These are configured to match the host architecture using 32- or 64-bit data and addressing widths (M, N). At bottom, we can see the signals used for the transfer control channel. The connectivity of this channel with the system bus is technology dependent and for this reason they are not listed.

Considering that writing and reading to this interface is implemented by means of multiple internal operations that are technology or protocol dependent, the control signals that specify the direction of data are suffixed by request (Req). The acceptance of each transfer by the M00 interface is later marked the

Table 3.6: The M00 interface - signal description and generic interface mapping.

M00 interface			**Description**	**Generic Mapping**
Data channel	CS	in	Same signals specified by the generic interface.	CS
	ADDR[M-1:0]	in		ADDR
	BE[(N/8)-1:0]	in		BE
	TXDATA[N-1:0]	in		TXDATA
	RXDATA[N-1:0]	out		RXDATA
	WR_ACK	out		WR_ACK
	RD_ACK	out		RD_ACK
Control channel	WR_Req	in	Write transfer request.	WR_CE
	RD_Req	in	Read transfer request.	RD_CE
	TLEN[B-1:0]	in	Transfer length using word units.	-
	CMDACK	out	Transfer acceptance acknowledge signal.	-
	ERROR	out	Transfer request or transaction errors.	-
	CMPLT	out	Transaction complete.	-
	BURST_DONE	out	Burst transfer done.	-
	RESTART	in	Restart the interface.	-

(N, M and B) - Host system definable architecture and burst length widths.

by *CMDACK* signal after host bus acceptance. When the interface implements transfers in the simplified format, the *BURST_DONE* signal will have the constant high value in the output, which indicates to the accelerator kernel that transfers are limited to transactions of a word of data. Each new transaction requires the request for a new transfer and the end of transfer is always marked by the active complete signal (Cmplt). In the event of errors, this signal is accompanied by the active *ERROR* signal. In the multiple sequential byte format, the *BURST_DONE* signal will have the logical value '0' and the *TLEN* parameter will specify the transfer length. The *Cmplt* signal is active at each word transaction and the transfer completes with the *BURST_DONE* signal active for one clock period. In the event of persistent anomalies, the design can optionally trigger a software reset on the M00 interface using the RESTART signal. To handle transactions in this interface, the *Kernel Core* provides four HW system calls that allow read or write transfers of single or multiple sequential words.

Algorithm 3 shows the pseudo-code describing the microprogram that requests a single write using the M00 interface. Similarity to other system calls, the implementation is scheduled in four steps. In the beginning, Step 0 suspends the context of the *HW-Task* using the *block_task* signal, while evaluating the *sysram_address_ok* flag. This control flag indicates the initialized value of the address register. In case of failure, the microprogram ends abruptly by advancing to Step 3 and releasing the *HW-Task*. If everything is correct, it proceeds to Step 1, where it maintains the *HW-Task* blocked and produces the *Wr_req* signal to initiate the write transfer. At the same time, it waits on the *CmdAck* input that allows the microprogram to proceed to Step 2. Once in Step 2, it waits for the transfer to complete, which concludes with the *Cmplt* signal. By reaching the Step 3, the microprogram releases the *HW-Task* and signals the completion of the system call with the valid flag. At the same time, the system-level datapath evaluates the inputs to generate the error flag and increment the error counter. It expects a logical '0' that results from the *ERROR* signal combined with the complemented control flag. When the kernel implements the HW-system call in the multiple sequential words format, Step 0 evaluates a combination of the same control flag with the complemented *BURST_DONE* signal, and in Step 3 the execution completes with the same signal active. The system call procedure activates the index service which is incremented at the pace of the respective ACK signals. Using this service, the procedure implementation manipulates the input data to generate the word for the next transaction.

Algorithm 3 Microprogram to write word at M00 interface

```
1: pseudocode SYS_CALL_WRITE_MBUS
2: Step0: produce block_task and test sysram address flag.
3:     if false then goto step 3.
4: Step1: produce block_task and test CmdACK input.
5:     if false then goto step 1.
6: Step2: produce block_task and test Cmplt input.
7:     if false then goto step 2.
8: Step3: produce valid
9:     exit
```

Figure 3.50 shows a wave plot for the system call that implements a burst read transfer of four words. The execution starts at time 12,465.00 nanoseconds with the *kernel_progress* equal to zero. One cycle later (i.e., marker 12,475.00 nanoseconds), the microprogram activates the request signal *i_mbus_rd_req*. The contents of the *i_tlen* input indicate a transfer length of four and the initial address 0x07044000 is used at the input *i_mbus_addr*. The interface acknowledges the transfer using the *o_mbus_cmdack* signal at 12,535.00 nanoseconds, and the first transaction concludes at marker 12,625.00 nanoseconds. The three remaining words are received using a two-clock cycle rate, accompanied by an increment of the index register. The transfer concludes with the *o_bdone* signal active at 12,695.00 nanoseconds, and the microprogram produces the valid signal for one clock period. At the bottom of the wave plot, we can also see the array of registers *bdata_out_i*, in the context of the *HW-Task*, that receive the four consecutive words that result from this transfer.

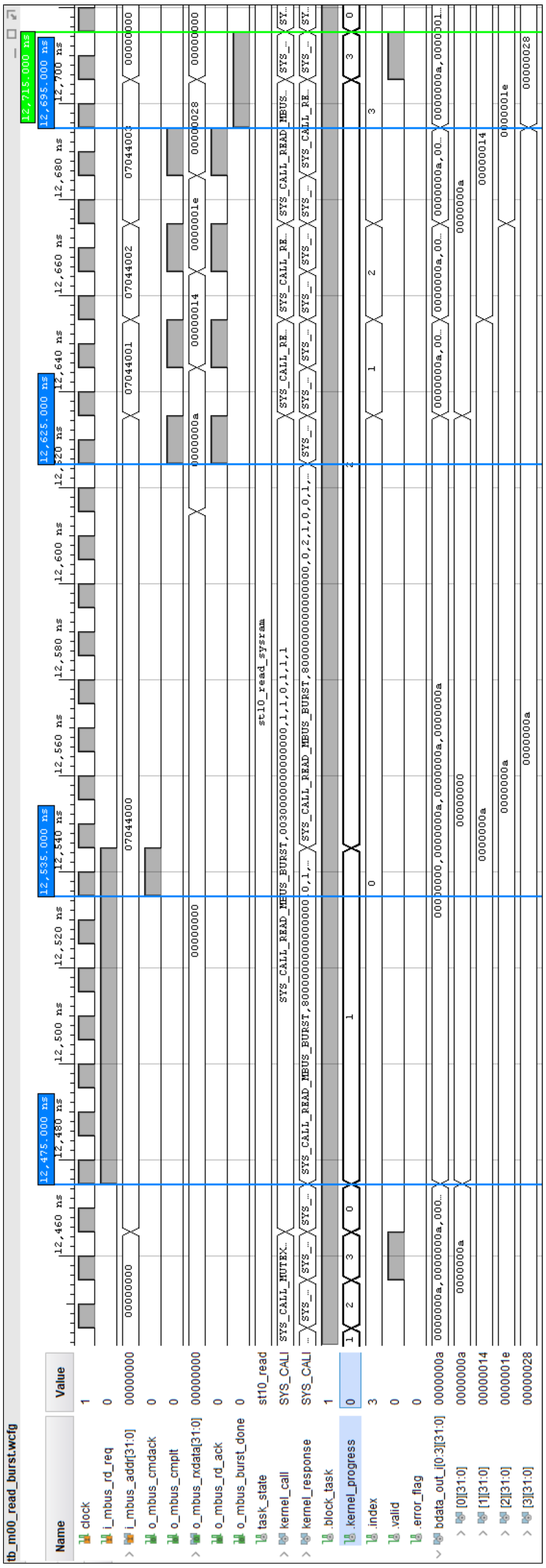


Figure 3.50: M00 interface - Read four words in SYSRAM using burst format.

Chapter 4

Auxiliary Hardware Components

The Accelerator Model implements a set of local resources to assist in the integration of the *HW-Task*, in software applications for the Linux operating system. In these, we can find the synchronization at the OS and application levels, the data exchange and local storage. In the previous chapter we have discussed the *HW-Kernel* interfaces that provide means for the host system to exchange information with the accelerator model. For this, it targets the set of local resources provided by the auxiliary components in the *HW-Kernel*. To provide connection with the Kernel Core, the *HW-Kernel* implements the Local-BUS, that follows the design considerations of the generic interface. In this way, specific interfaces are required to implement the concurrent access of the host system and the *Kernel Core*.

In the following sections we discuss the implementation details in each resource starting with the Local-BUS. For completeness, throughout these sections we have included examples of system calls that target the correspondent resource. We then conclude this chapter by summarizing the distinct accelerator versions and comparing them in the micro-architectural differences that each represents.

4.1 Local-Bus

To address resources that do not have a dedicated interface the *Kernel Core* makes use of the Local-BUS (LBUS) In its internal structure, the address range is 1024 kB and is logically organized in two distinct memory regions of equivalent size. The first area is the low-memory region and is intended for register-based addressing. As such, this address range is divided in 16 words of 32-bit per page, and pages 0 and 1 are physically mapped. The second memory region is reserved for addressing the LRAM which can scale up to 256 kB. Figure 4.1 depicts these two memory regions of the LBUS where the addressable registers are visible.

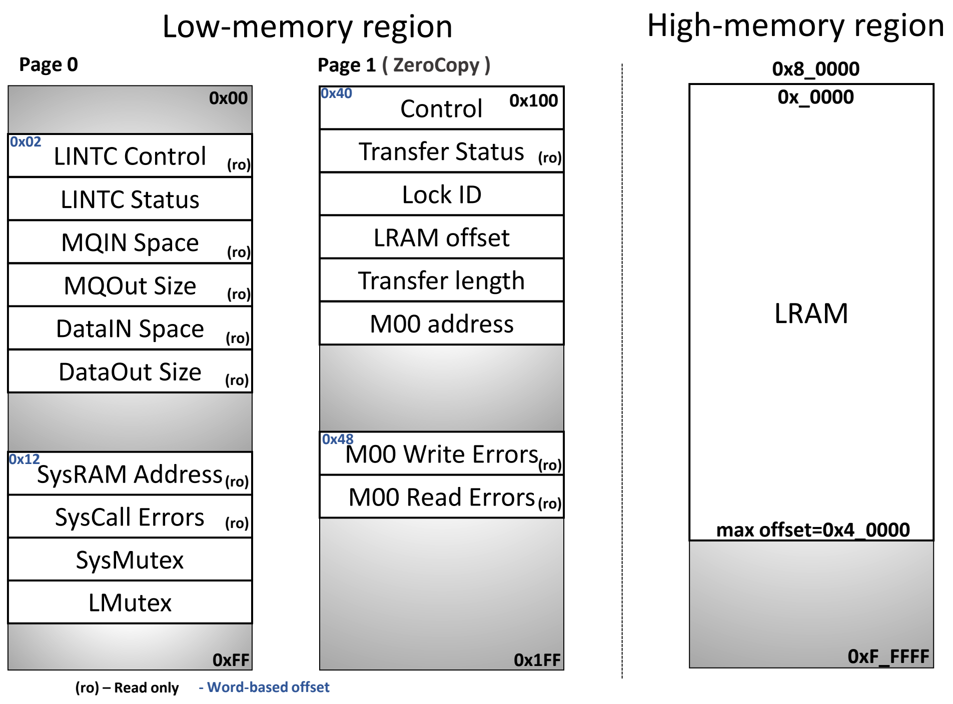


Figure 4.1: Local-BUS mapping - Low- and High-memory regions.

The low memory region maps the local resources in the *HW-Kernel* and to promote transparency in the overall design, the same offsets of the S00 interface are used. Most of these are mapped as read-only registers, and the *Kernel Core* is allowed to write: the status of the *HW-Mutexes*, to lock the resource; the status of the local interrupt controller, to trigger user-definable interrupts; or in the *ZeroCopy* unit registers, to trigger concurrent memory transfers. In the high-memory region, we can see the LRAM space whose size is a parameter on the accelerator interface. To address this memory regions the *Kernel Core* executes two system calls that implement the read and write transactions, and uses a 20-bit address field in the parameters member of the system call.

The LBUS design follows the generic interface format and allows one master, which is the *Kernel Core*. To map the resources on pages 0 and 1, as well as the LRAM memory range, this bus implements a two-level partial address decoder using lines A19 and A6. In level zero, it combines a hierarchy of one slave decoder to map four pages of asymmetric range. In level one, it implements two-page decoders for the register area (i.e., pages 0 and 1) and the remaining two pages are merged into the LRAM interface using appropriate line selection. Figure 4.2 describes the internal organization of this component showing three interfaces dedicated to each memory location. Implementation details about the Slave or Page decoders, as well as the design considerations for the resource addressing, are discussed in Appendix A.

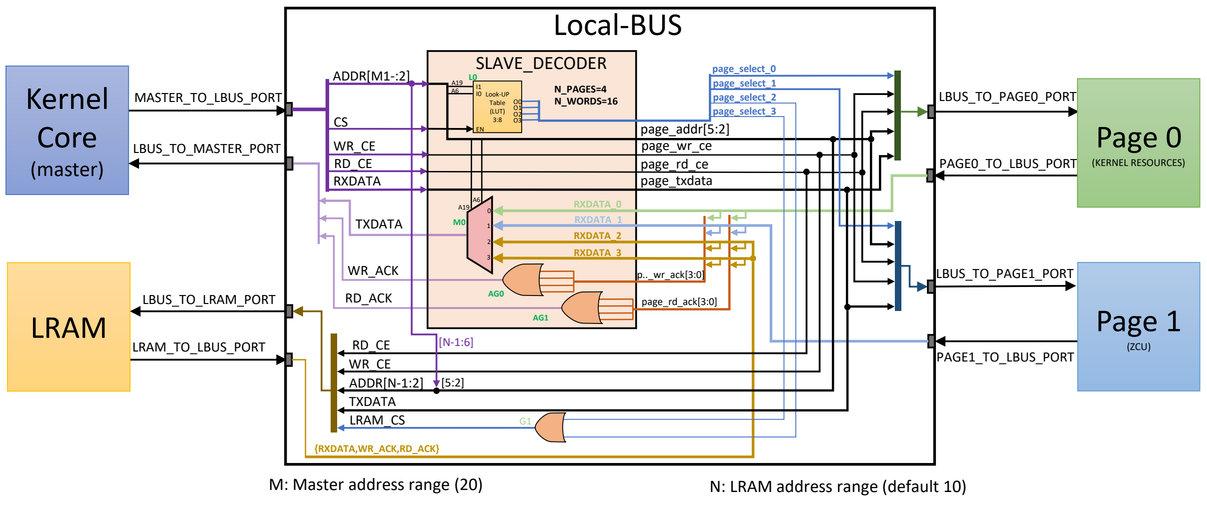
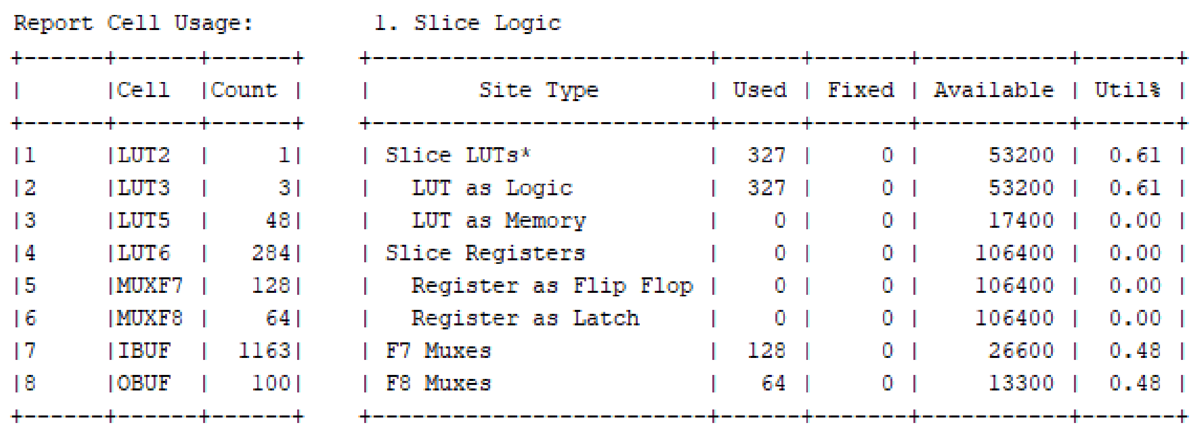


Figure 4.2: Local-BUS architecture - slave interfaces and address decoding.

At the output of L0, the first two lines of page select are connected to the interfaces of pages 0 and 1, respectively. The remaining lines, 2 and 3, are logically combined to form the LRAM select signal (LRAM_CS). Likewise, in M0 we can observe the connection with pages 0 and 1 in the corresponding inputs, whereas the inputs 3 and 4 are connected using the same signals from LRAM. As a consequence, the accesses to low memory using addresses above page 1, will hit pages 0 and 1 in alternate sequence. To avoid possible memory collisions in the asymmetric ranges, the kernel package applies a static 7-bit limit of the offset parameter (i.e., *CLBUS_REG_WIDTH* constant), in the procedures that access the low memory region. An excerpt of the kernel package containing the write procedures that target the low and high memory regions can be found attached in Listing C.15.

To expand the high-memory region beyond the two pages, the page address bus is extended with the necessary bits from the absolute address. This connection uses the LRAM's length parameter at the top-level of the accelerator, to make the correct assignment in the page bus and in the interface with the LRAM. In procedures that manipulate data using this local resource, accesses are always relative to offset 0x0000, and it is not possible to specify absolute addresses. An example of a procedure for writing one word in the LRAM can also be seen in the attached Listing C.15.

For completeness, in Figure 4.3 we can see the excerpt from the resource consumption report in the synthesis phase, using the target platform ZC702. These include the implementation of the slave decoder parameterized using 4 pages of 16 words, and two-page decoders referring to pages 0 and 1.



```
Report Cell Usage:       1. Slice Logic
+------+------+------+    +-------------------------+------+-------+-----------+-------+
|      |Cell  |Count |    |        Site Type        | Used | Fixed | Available | Util% |
+------+------+------+    +-------------------------+------+-------+-----------+-------+
|1     |LUT2  |     1|    | Slice LUTs*             |  327 |     0 |     53200 |  0.61 |
|2     |LUT3  |     3|    |   LUT as Logic          |  327 |     0 |     53200 |  0.61 |
|3     |LUT5  |    48|    |   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
|4     |LUT6  |   284|    | Slice Registers         |    0 |     0 |    106400 |  0.00 |
|5     |MUXF7 |   128|    |   Register as Flip Flop |    0 |     0 |    106400 |  0.00 |
|6     |MUXF8 |    64|    |   Register as Latch     |    0 |     0 |    106400 |  0.00 |
|7     |IBUF  |  1163|    | F7 Muxes                |  128 |     0 |     26600 |  0.48 |
|8     |OBUF  |   100|    | F8 Muxes                |   64 |     0 |     13300 |  0.48 |
+------+------+------+    +-------------------------+------+-------+-----------+-------+
```

Figure 4.3: Local-BUS - Resource usage for final design in Zynq ZC702.

4.2 HW-Mutex

The *HW-Mutex* implements exclusivity in the accelerator model in a similar way to the memory object used by the C/C ++ language, which is a containment mechanism implemented through lock and unlock operations. There are two HW-Mutex units in the accelerator model, the first is Local-Mutex (*LMutex*), which ensures exclusivity in writing to the LRAM, and the second is the System-Mutex (*SysMutex*), to implement exclusivity in the system memory region that may have been allocated by the HAL-ASOS file system. Figure 4.4 shows a simplified diagram that describes the internal organization of this component.

This unit is based on two concurrent channels, A and B, that can read from, and write to, the status register. The write operation can store a key that distinguishes the exclusive owner of the resource, while keeping the device locked to one channel and ignoring successive attempts to write keys in the other channel. Therefore, this resource can only be released by writing the same key that blocked the device while using the same physical interface. Once in the locked state, it activates an output signal that can be used to enable write operations on associated resources, such as the LRAM or the ZCU unit. In the accelerator design, channel A is dedicated to the host and is connected to S00 interface, while channel B is dedicated to the *Kernel Core* and is connected to the Local-BUS. In its internal control logic, the *HW-Mutex* favors channel B in cases of concurrent lock attempts, on the assumption that the latency in the operations to be carried out is much lower since it is a local resource in the design of the accelerator.

The key that identifies the owner is stored in the status register (FF4) and results from a 30-bit composition (WID [29:0]), containing a unique numeric ID which is concatenated with 1-bit (CHID) that distinguishes the channel where this key was presented. Using K0 and K1, the resulting composition prevents the B-lock key from being equivalent to the A-Lock key, or vice versa. In turn, distinct WIDs allow the Host to have

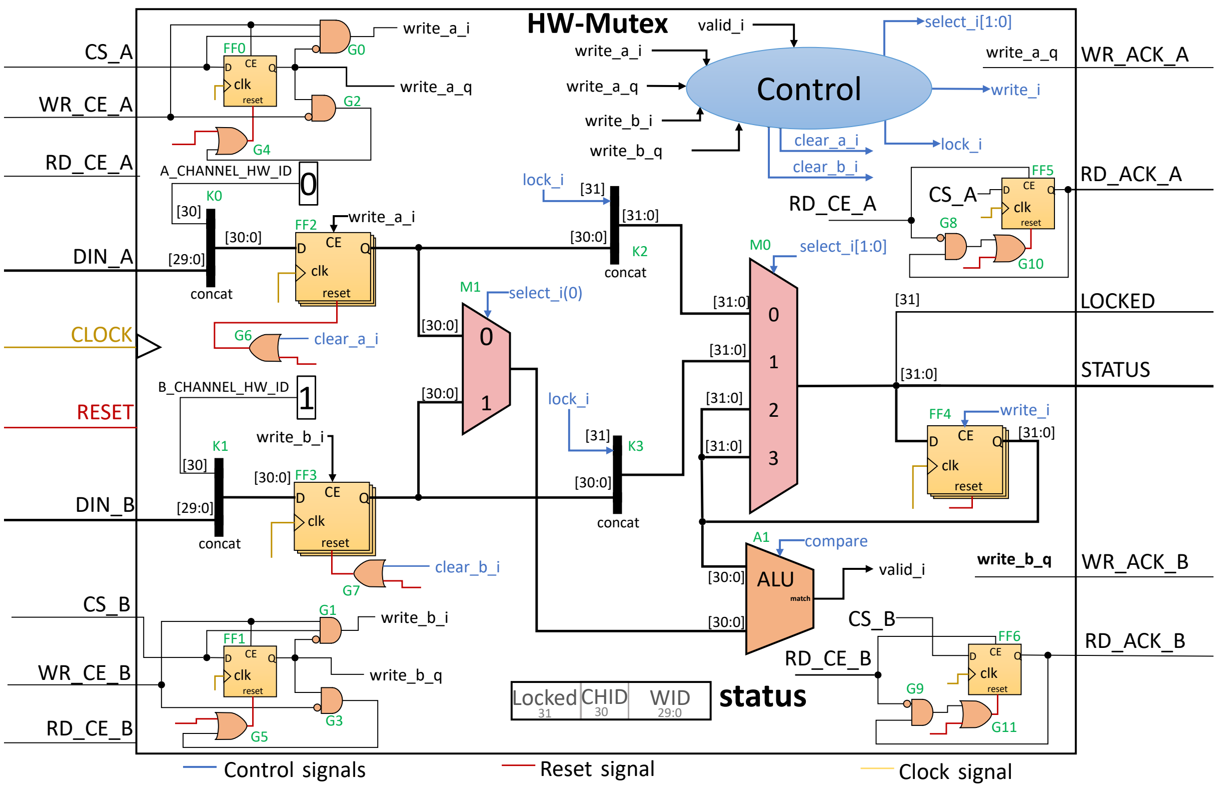


Figure 4.4: HW-Mutex design architecture.

different entities competing for the same resource, while side B is accessed by a single entity, i.e., the *HW-Task* using a WID extracted from its *TaskName* parameter.

Once in free state, the device keeps the status register closed on itself, by selecting input 3 of M0 and disabling the CE input in FF4. At the same time, this choice places the contents of the status register in the output of the device. A write operation performed in A or B, first reaches registers FF2 or FF3, which are inputs 0 and 1 of M0, after being concatenated with the control signal *lock_i* (K2, K3), indicating the status of the device. The writing in these registers is triggered by the logical combination G0 and G2 for channel A or G1 and G3 for B, which enables clock signal for a single period. These write signals are also inputs from the control unit and they can determine the subsequent forwarding of the chosen key to the FF4 register. For this, the control unit selects the appropriate input in M0 and activates the *write_i* signal in FF4 for one clock period. At the same time, depending on the selected channel, it activates the clear signal in G6 or G7, thus clearing the accepted WID. In the next clock cycle, the control closes the status register on itself again, selecting the inputs 2 or 3 in M0, accordingly to the chosen channel.

The least significant bit of the *select_i* signal (bit 0), is used in M1, to choose the channel that owns the resource and to compare it with the current value of the status register in A1. This comparison indicates to the control unit if the key that holds the exclusivity in the resource was presented again at the input. In this context, a new write operation can only be enabled after FF0 or FF1 storing the logic'0', using G2 and G4 or G3 and G5, respectively. With the *valid_i* signal active, the control writes the new key in FF4, which has the *locked_i* signal disabled in its composition. This action establishes the status register value as free on the next active clock transition. For transaction handshake on the interfaces of the two channels, S00 and Local-BUS, distinct signals are generated in the next clock cycle using FF0 and FF1 for write acknowledge, and FF5 and FF6 for read acknowledge. The Status output is updated in response to the write transactions, and depending on the lock flag, it uses one clock cycle when locking the device, or two clock cycles if the device was previously locked.

Figure 4.5 shows a state logic diagram which describes the implementation of the *HW-Mutex* control unit. The initial state, #0, is assumed after a reset signal and remains active as long as the resource is free. During this state, the *select_i* output locks the Status register with the logical value "11". This unit implements two independent flows that differentiate channel B from A in the sequence of states. The flow on the right is triggered by the channel B write request independently of A request, and determines the sequence of states #0, #1, #2 where it remains locked at the service of B. The flow on the left is triggered by the channel A write request when channel write B is not asserted, and determines the sequence of states #0, #4, #5 where it remains locked at the service of A.

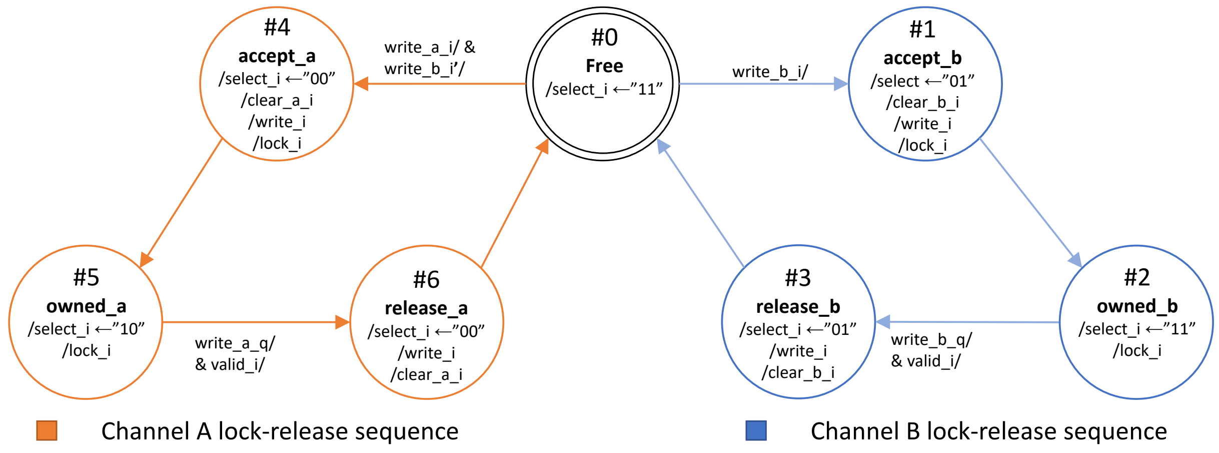


Figure 4.5: HW-Mutex control unit state diagram.

When A or B intend to release the resource, they re-write the ID in the device. In response, the control unit evaluates the logical value of the *write_a_q* or *write_b_q* and *valid_i* inputs, before proceeding to the release states, #3 or #6. Once in these states, the Control chooses the corresponding ID and produces the write signal which updates the status register. In the next clock cycle, the control unit reaches the free state where it remains until a new lock request.

In Figure 4.6 we can see a behavioral diagram that results from the simulation of this device using Vivado. For simplicity some signals have been omitted. At simulation time instant of 45 nanoseconds, A and B compete for the resource with the keys 0xace0 and 0xace1 respectively. Since the control design favors the B channel, the state register assumes *st1_accept_b* value and in the next clock cycle, the output *o_locked* is set high indicating that the device is locked. By analyzing the *status_q* output, we can see that the channel B holds the resource exclusivity with the composition 0xc000ace1. At time instant of 85 nanoseconds, input A replicates the composition in Status register at the input A, but control is no longer active on this channel and remains in the *st2_owned_b* state. At time instant of 125 nanoseconds, channel A acquires the resource with the composition 0x8000ace0 after B release, and at time instant of 175 nanoseconds releases the resource. The control unit reaches the free state two clock cycles later and the Status register holds the composition ID that released the resource.

4.3 Local RAM

In the accelerator model, the local RAM (LRAM) implements the memory segment used for storing variables in the scope of the *HW-Task*. In its conceptual model, this storage space is largely similar to the model of a cache, and ultimately is used by the accelerator kernel to exchange data with the host system's memory. In the accelerator design this storage space is addressable by means of the S01 interface and as such, by specifying the address lines of S01, it establishes the size of LRAM. In the set of configurable parameters, the *C_S01_ADDR_WIDTH* is by default set to 10-bit, thus resulting in a total LRAM storage capacity of 1024 bytes. To promote parallelism the LRAM is implemented following the Synchronous True dual port RAM design, where two concurrent interfaces, A and B, can be connected each using its own clock source. In doing so, the channel A is connected to the S01 interface and channel B is connected with the Local-BUS, allowing the host system and the *Kernel Core* to simultaneously access to the LRAM contents. In order to avoid collisions, writing operations are allowed only to the channel that acquires the *HW-Mutex*. Such resource is the *LMutex*, from which the CHID in the status register and the locked output signal, are combined logically to establish write-enable (WE) signals for channels A and B in the LRAM. In doing so, the write privileges will be established to channel A when logic zero CHID is in the locked composition, or to channel B when logical one CHID.

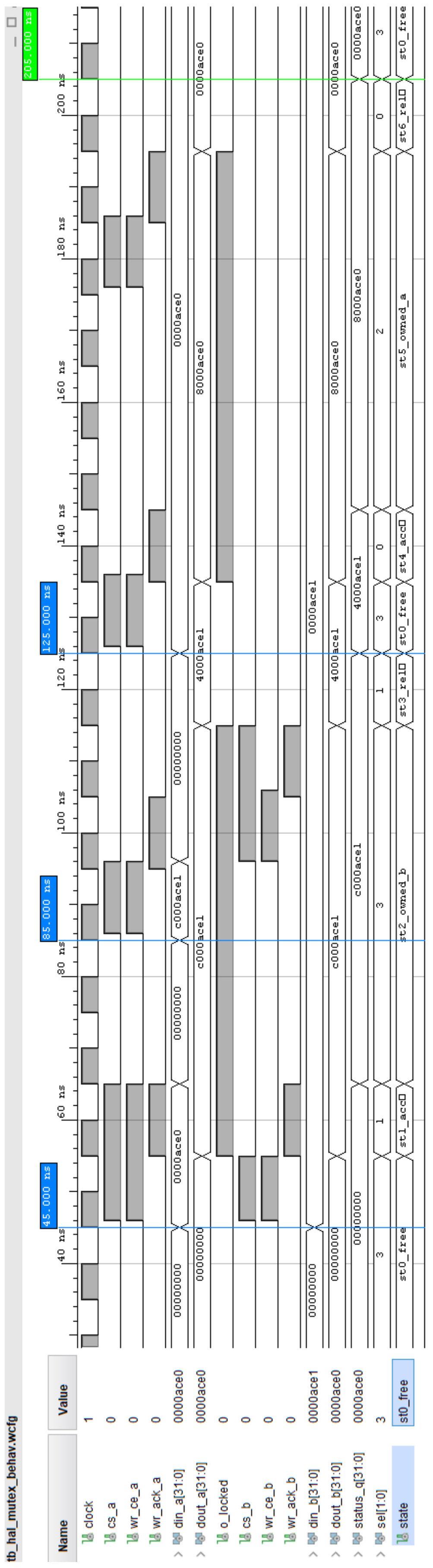


Figure 4.6: HW-Mutex wave diagram: concurrency scenarios.

In Figure 4.7, it can be seen the architectural design used to implement the LRAM component. For a 32-bit target architecture, the design is internally organized in four 8-bit memories and follows a little-endian ordering. In each of the channels, with the activation of the chip-select signal (CS), the next clock transition updates the output (DOUT) with the contents in the position specified by the address bus (ADDR). Thus, when reading this device, such operation always refers to the total number of bytes in a word and the BE bus is not considered. If otherwise, the WR and WE inputs are active, the logical combination with the BE bus, enables writing in the bytes of one word, at the position specified by the ADDR bus. In this case, the next clock cycle will update DOUT with the new values received at the DIN input, and the existing ones that were not affected. Both operations are confirmed using the correspondent acknowledge signals *WR_ACK* and *RD_ACK*.

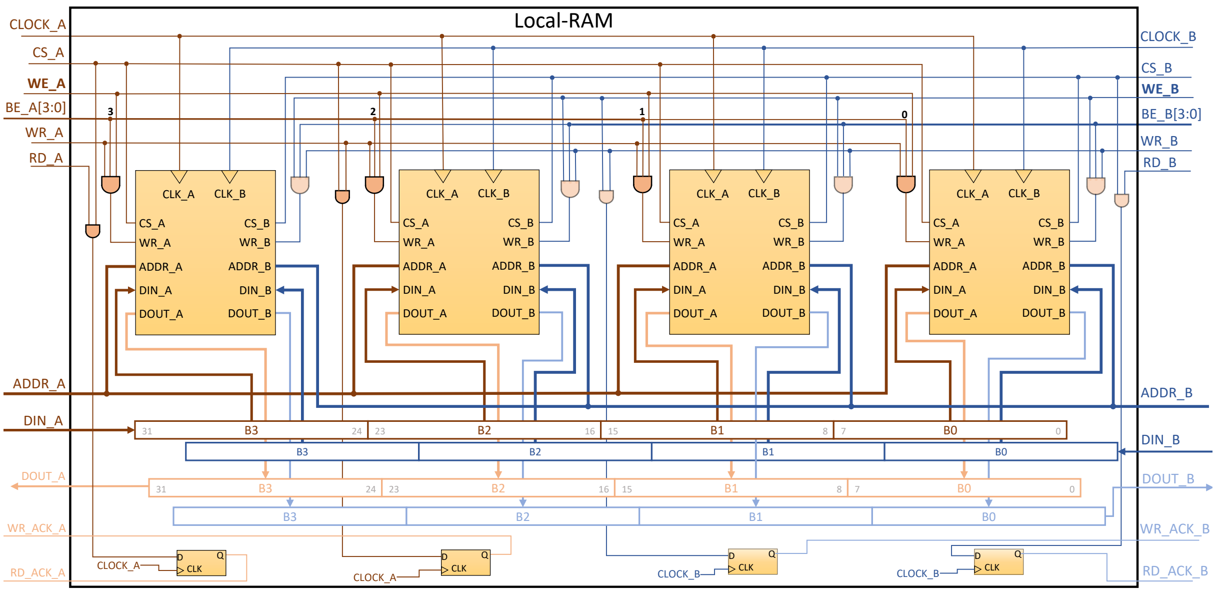
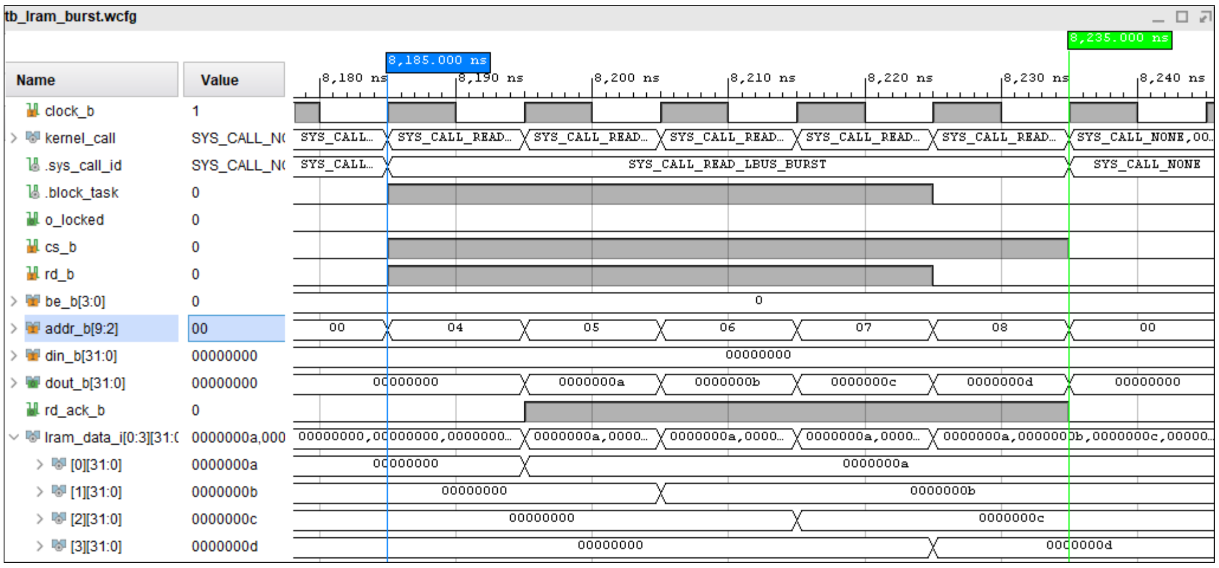


Figure 4.7: Local RAM - internal architecture for 32-bit accelerator design.

To access the contents of this memory, the *HW-Task* design implements the system calls that interact with the Local-BUS. In its generic form, each read or write system call consumes two clock cycles, which allow the microprogram to effectively exchange data and deal with the bus handshake signals. Alternatively, for transactions that involve a consecutive number of words in memory, the *HW-Task* can implement the system calls that provide a burst-like transfer mode. Usually, in this transfer mode, each word is exchanged following one clock cycle rate, by the elimination the initial steps in each transfer. Given the simplicity of the Local-BUS, all transfer steps can be implemented while exploring the parallelism in the design. For that, the beginning of a new transfer overlaps the handshake of the transfer in progress, and in doing so, each word can be exchanged at one clock cycle rate. The system call completes with an extra clock cycle that handshakes the transfer of the last word. In Figure 4.8 it can be seen a wave diagram that describes this transfer mode for a consecutive reading of four words.


tb_lram_burst.wcfg

Name	Value
clock_b	1
kernel_call	SYS_CALL_N(
.sys_call_id	SYS_CALL_N(
.block_task	0
o_locked	0
cs_b	0
rd_b	0
be_b[3:0]	0
addr_b[9:2]	00
din_b[31:0]	00000000
dout_b[31:0]	00000000
rd_ack_b	0
lram_data_i[0:3][31:(	0000000a,000
[0][31:0]	0000000a
[1][31:0]	0000000b
[2][31:0]	0000000c
[3][31:0]	0000000d

Figure 4.8: Local-Ram wave diagram: burst read system call.

In the same figure, it can be seen the kernel call interface, the *block_task* signal from kernel response interface, the locked signal in *LMutex*, the interface B in the LRAM and the internal variable *lram_data_i* in the context of the *HW-Task*. At time instant of 8,185.00 nanoseconds, the *HW-Task* implements the system call to read four words from the LRAM starting at address 0x4, and as response the kernel activates the *block_task* signal. For this purpose, the exclusivity in the *LMutex* resource was not acquired and this signal remains low for the entire system call. At each *clock_b* cycle, the interface B address is incremented once, using a count based on the number of words.

The first read acknowledge occurs one clock cycle later, at time instant of 8,195.00 nanoseconds. At this time, the LRAM makes the content of the word at address 0x04 available at *DOUT_B*, with the value of 0x0000000a. Simultaneously, it receives a new request to read the contents at address 0x5. For each active cycle of *RD_ACK*, the *lram_data_i* is updated with the LRAM output using the correspondent position. This pattern of operation repeats itself for three clock cycles and in the fifth cycle, at time instant of 8,225.00 nanoseconds, the system call receives acknowledge of the last word read. In response, it updates the last position of the *lram_data_i* variable and completes the operation by clearing the *block_task* signal.

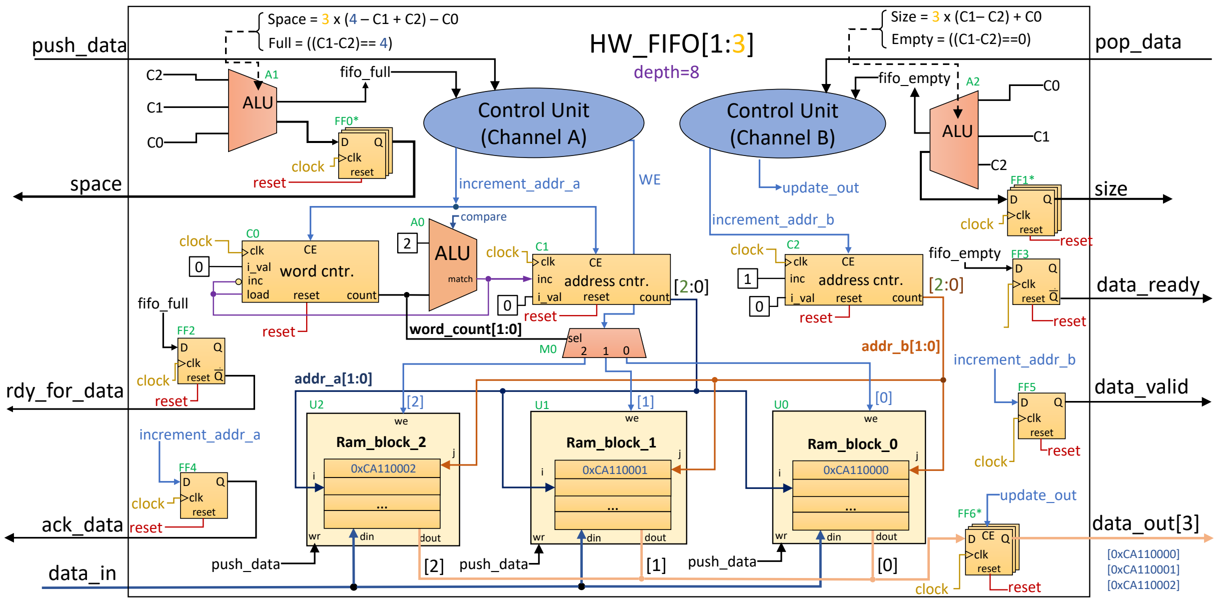
4.4 Message-Queue

In the HAL-ASOS framework, the message-queue is the established interface for bi-directional and control-oriented communication, used by the *Kernel Core* and *HW-Task* designs. At the *Kernel Core* level, the accelerator programming model dictates that for each message sent there must be a return message, and in doing so, the message-queue is reshaped into a bidirectional logical channel implemented using two *HW-FIFOs*. The reduced number of control signals required by this component allows for the accelerator model to implement a dedicated interface for each *HW-FIFO*, and thus providing an efficient and orderly exchange of data that can handle multiple words per clock cycle. By its simplicity, it also minimizes the impact on the microprogram and at the same time provides a containment mechanism which can be used to synchronize the control logic in the *HW-Task*. At the *HW-Task* level, the use of this component is optional and the designer should parameterize each component to meet the application requirements. The *HW-Task* design is provided with a dedicated interface for each *HW-FIFO* that is resumed to the signals that provide effective exchange of data. The control signals for each component are still managed by the microprogram and are activated by the appropriate system calls invocation.

To map the specified settings into an efficient design, the accelerator model provides two distinct *HW-FIFO* architectures in a single component, which are applied to the receive and transmit channels of the *HW-Task* and *Kernel Core* units. In the receive channel, each *HW-FIFO* is written by the *S00_Control* interface and, as such, the design establishes a fixed single word input and a parameterizable number of words output, i.e., [1:M]. On the transmit channel, different number of words can be written in each HW-FIFO, which connects the read output with the same generic interface, and in this way, the HW-FIFO design establishes a parameterizable number of input words and a fixed one output word, i.e., [N:1].

To promote the efficiency while exchanging multiple words per clock cycle, each design exploits the FPGA's memory blocks by replicating the storage units according to the number of words it receives as a parameter. Figure 4.9 shows the HW-FIFO architecture for the receive input. For the selected architecture, the HW-FIFO component receives as parameters three output words and a storage space of eight words of 32-bit. To provide the desired number of words, the design replicates the storage using three true-dual port memory blocks. For computing the desired storage space, the received number of outputs is normalized to the nearest value in the power of two range, as described at the bottom of the figure, which determines a storage space of four words per Block RAM (BRAM). As a consequence, the effective storage scales to twelve words, as result of three block RAMs using four words capacity.

To ensure concurrency and consistent access to storage resources, this design implements two independent control units, A and B. Each unit controls an address register that is common to all memory blocks which are incremented using counters C0 and C1 for address register A (*addr_a*) , and C2 for address register B (*addr_b*). To avoid address collisions, the read address (B) can never exceed the write address (A), and similarly, the write operation can never go further than the effective storage space over the read address. For this purpose, each control unit senses a datapath flag that indicates the status of full, for the writing operation, and empty, for the reading operation. The logical value of each flag results from the arithmetic operations represented by ALUs A1 and A2.



(Depth = M , N_OUTPUTS=O) : (8,3) –> Least_Pow2(3)=>2; RAM_SIZE = 8/2 =4 WORDS; ADDR_WIDTH=2 (N) –> Effective Size(E) = 4x3 = 12 WORDS

Figure 4.9: HW FIFO[1:3] - architecture using 3 and 8 configured parameters.

Due to the asymmetric number of input and output words, the control unit A uses C0 to select one of three available RAM blocks, and C1 to control the write address. Writing a word to a specific RAM block depends on the value of counter C0, which is used in M0 to establish connection with the active WE control signal. The control unit authorizes writing by activating this signal, after evaluating the logical value of the full flag. When the *push_data* signal is active, the *din* input is written in the RAM block that has the active WE input, at the position indicated by the *addr_a* register. In response to the *push_data* signal, the control unit A activates the *increment_addr_a* signal which advances C0, and reevaluates the *fifo_full* flag to produce a consistent value for the WE signal. When the C0 count is equal to the number of output words minus one (i.e.,2 in the provided example), the comparison in A0 activates the *match* flag, which enables the *inc* input of C1. The next increment signal advances C1 thus producing a new write address, and loads C0 with the value zero for a new word count.

When the difference between the C0, C1 and C2 reaches the existing storage space, the *fifo_full* flag will be asserted. As consequence, the control unit A will deactivate the *WE* signal suspending the write operation to all RAM blocks. Each RAM block will still receive the *WR* signal that requests a write operation, but since none of them has the *WE* input active, the existing data will be preserved. Under this condition, the write address will no longer be incremented until progress is registered in the read address.

Whenever the C1 advances beyond the number of output words above C2 (i.e., size equal to 3 in the provided example), control unit B receives the zero value from the *fifo_empty* flag, and in response, it activates the *update_out* signal, anticipating the received set of words to output register FF6*. This register enforces a logical grouping between the three output words and prevents the information from existing in an intermediate inconsistent state. In this mode of operation, the update in the output B occurs at a rate of three words per clock cycle, and the read address is only incremented after the *pop_data* signal has been activated and the data handshake on channel B has been completed. Beyond this handshake phase, if there is no readable data, i.e., *fifo_empty* is active, the output values remain unchanged and the output *data_ready* is disabled, signaling that the current *data_out* values have been read. Therefore, the control unit B remains in a blocking state where it waits for a progress in the C1 to increment the C2 and produce a new output value. For completion, in Figure 4.10 we have included two state logic diagrams of the control units A and B that implement the behavior according to the descriptions above.

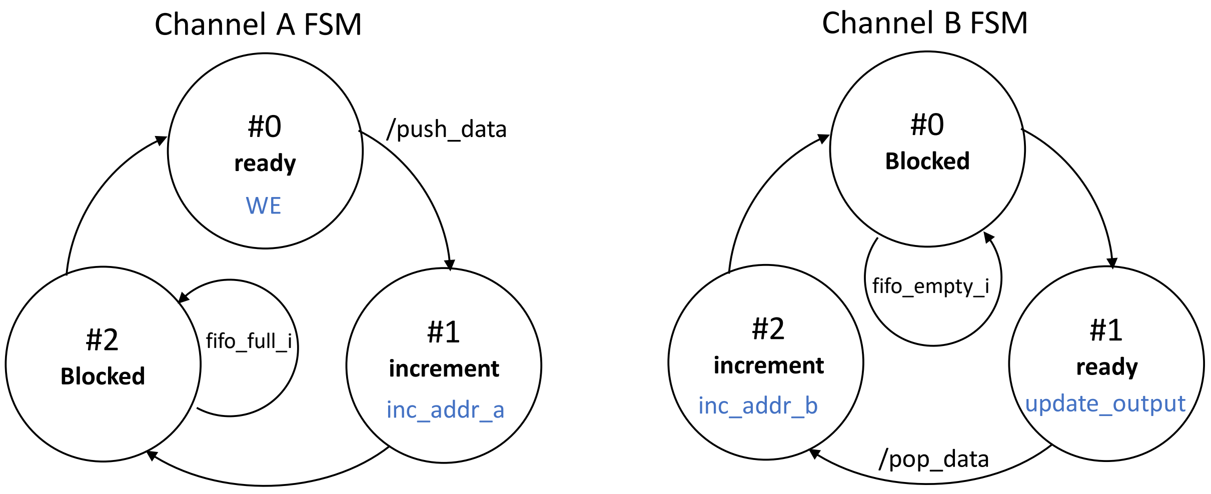


Figure 4.10: HW FIFO[1:3] - architecture using 3 and 8 configured parameters.

A similar design is used in the transmit channels and allows the *Kernel Core* to write a configurable number of words per clock cycle into each *HW-FIFO*. It also includes two independent control units and a word counter is used to select each output word in similar way as in M0, but now using a 32-bit MUX that connects its inputs to all memory blocks and the selected output is stored in the FF2* output register.

Figure 4.11 shows a wave plot diagram that results from the simulation of the *HW-FIFO* component using a concurrent write and read scenario. For this purpose, the same parameters used for the previous descriptions were considered, as can be seen in the last two lines. As consequence, an effective storage space of 12 words is available, as shown in the *o_space* output value (line 3), after the reset signal has reached the logic value '0'. During the simulation, channel A receives 18 write requests, where consecutive values from 0 to 17 are used in the *i_data_port* signal. For each valid *push_data* signal at the input, the HW-FIFO responds with the *o_ack_data* signal, indicating that it has registered the received value. The write operations conclude at the time 575 nanoseconds, indicated by the green marker, with the control unit reaching the *st2_blocked* state due to lack of space. At this instant of time, the *o_ready_for_data* signal has been disabled and the available space is 0.

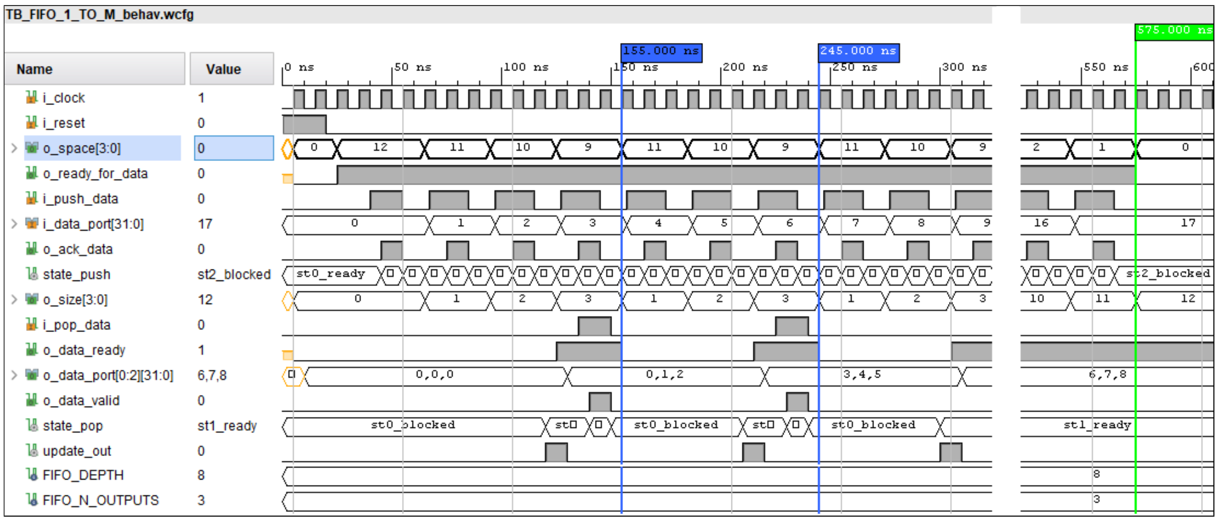


Figure 4.11: HW-FIFO [1:3] - Wave plot simulation diagram using Vivado.

On channel B, the *update_out* signal is activated by the control unit when the value of *o_size* reaches the number of outputs for the first time (i.e., 3). In the next clock cycle, the *o_data_port* output is updated with the value of the first three words received, and the *o_data_ready* output goes to the active state. In response to this signal, the *pop_data* input receives the logical value '1' and after a handshake cycle involving the *o_data_valid* signal, the control advances to the s*t0_blocked* state since the size registers the value 1, which is less than the number of outputs. This cycle repeats every time *o_size* reaches the value of 3 and only once the *pop_data* signal is received. The simulation concludes with the *o_data_ready* signal active, indicating that the third group of words in the output has not yet been read. The value of *o_size* registers 12 words available for reading, as the result of 18 words received minus 6 words read.

4.5 Local Interrupts

To establish synchronization between the accelerator model and the Linux operating system, the design makes use of the LINTC. In summary, this component operates by multiplexing an interrupt line that it receives from the target platform, by the set of interrupt sources that the accelerator model provides. In this set, seven interrupt sources are considered native since they are directly related to the HW resources in the design. Native interrupt sources are used to implement synchronization at the operating system level, while exchanging data between the Host system and a particular HW resource. Additionally, the accelerator's top-level settings allow to specify interrupt sources dedicated to the context of the *HW-Task*. Similarly, the user definable interrupt sources implement synchronization between the *HW-Task* design and the target application. The Host system uses the S00 interface that maps the control and status registers of this component, to mask each input source, enable the interrupt signal and acknowledge the resulting flags. Figure 4.12 shows the composition of the control and status registers.

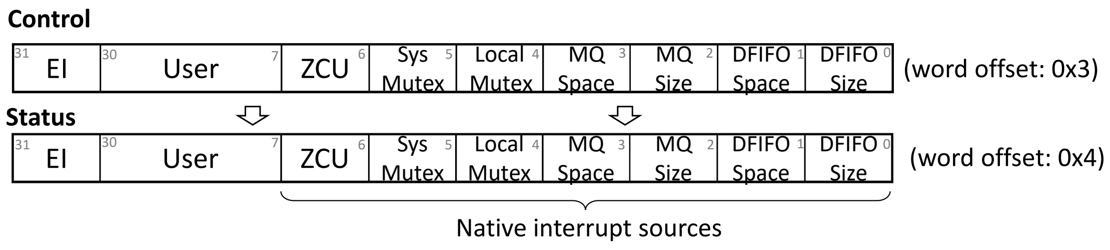


Figure 4.12: Local Interrupt Controller - control and status register.

To mask an interrupt source, the host system activates the corresponding bit in the control register and in the occurrence of a high value at the specified source, the status register is updated with the active flag in the same position. When the most significant bit of the control register (EI) is active, the transition from zero to one in a status flag, triggers the interrupt signal that is sent to the host platform.

Figure 4.13 describes the implementation of the local interrupt controller. In FF0 we can see the control register used to mask interrupt sources. When the Host system writes to the control register, the S00 activates the CS line that selects the 0x03 word offset. Considering that the status register is used to flag masked interrupt sources, the host system can write to the 0x04 offset to acknowledge such flags. In doing so, it can activate bits in the handle register (FF1), that are used to clear the corresponding status flags. When writing to the status register, the active bits in FF1 are used to clear the flags in FF8, FF12 or FF15. The instant the *write_status* signal returns to zero, a logical combination with FF2 resets the FF1 bits. The contents of the control and status registers are made available using the 32-bit outputs that can be read by the host system using the same interface, or by the *HW-Task* by means of adequate system calls that use the Local-BUS. In the same way, the *HW-Task* can trigger user interrupt sources by writing to the offset of the status register. For the occurrence of such interrupts, the corresponding bit in the control register must be previously activated by the host system.

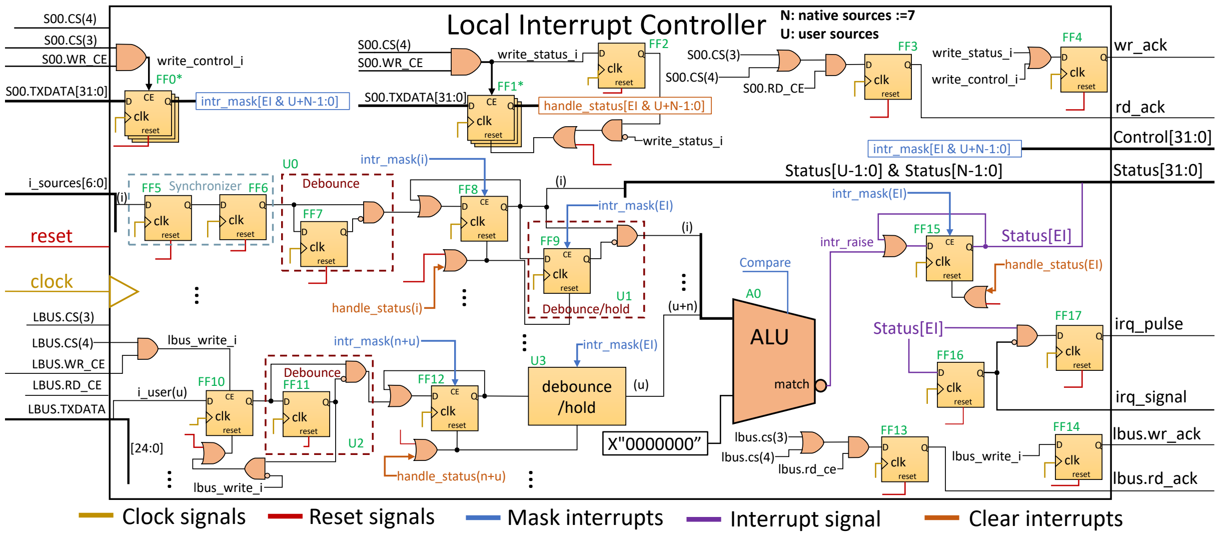


Figure 4.13: Local Interrupt Controller - internal architecture diagram.

Native interrupt signals connect to the 7-bit input sources. Here, multi-flop synchronizers (FF5 and FF6) can be activated to handle distinct clock domains in the overall design. The U0 debounce circuit allows the source input at position 'i' to be recognized during a clock period. As such, the same signal must reach a logical zero during at least one clock period before generating a new interrupt source. If the bit in the 'i' position of the control register is set, the CE input of FF8 allows to activate the flag in the corresponding position of the status register. Such a flag will generate a high value at the input of A0 until the EI bit of the control register is activated. A logic one in this bit will store the *intr_raise* signal in FF15 and at the same time, the combination of FF8 and FF9 invalidates the signal at the input of A0, until the output of FF8 reaches a new logical zero. In doing so, U1 prevents a status flag from giving rise to more than one interrupt source, before being acknowledge by the host system. The output of FF15 corresponds to bit 31 of the status register, and the active state of this flag, is used to generate a persistent signal in FF16 that can be connected to the host interrupt line. In FF17 a pulse is generated that can be used as an alternative in the same line.

The seven native sources use a similar datapath as described in the figure, were more flip-flops such as in FF8 implement the status flags. Similarly, each bit in the TXDATA input from the Local-BUS, can be captured in FF10 and FF11 to activate the corresponding flags in the same register, as depicted by FF12. To handshake these interfaces, each read or write operation gives rise to acknowledge signals using FF3 and FF4 for S00, and FF13 and FF14 for the Local-BUS.

4.6 ZeroCopy Unit

When applying the HAL-ASOS programming model to a new design, or when refactoring an application, the resulting memory layout will be distributed between the system memory and LRAMs on each accelerator. Such layout will require a memory data movement that may translate into an increased processing and contribute to a degradation of performance metrics. Any performance degradation will be closely linked to the length of data that the application will handle, as well as the clock ratio of the existing CPU cores and the different buses through which these memories are addressable.

In order to deal with the frequent movement of data in memory DMA devices are often used. This device allows to free the CPU from the task of copying data between memories, so that they can be allocated to operations that relate to their local segment or caches. Examples are data movements carried out through the network subsystem, which require frequent transfers in both directions. In some cases, the movements performed by the DMA are affected by the location of data and make access to the memory temporarily unavailable to the CPU. Depending on such location it can establish connections between different memory regions and as such, it may require two different bus interfaces. In doing so, the DMA transactions generally involve more than one form of intermediate storage, either at the system level or at the level of its the internal organization.

With the ZeroCopy unit (ZCU) we attempt on reducing the impact that the data movements represent, but at the same time by embedding this feature in the accelerator model, we try to keep resource usage at acceptable levels by re-using existing HW interfaces. In a broader perspective, this unit cooperates with the zero-copy strategy, in the HAL-ASOS file system for Linux, being part of the set of services it provides. Within this strategy, it aims to minimize the copy overhead at the operating system level. In doing so, this unit provides services to the Host system by moving data between the system memory and the local storage (i.e., the LRAM) on the accelerator where it is implemented. In addition, this functionality can be extended to the application-level, promoting parallelism in the exchange of data by the initiative of the *HW-Task*.

Figure 4.14 shows a simplified diagram of the ZCU connectivity with the host system, the *Kernel Core* and the memory interfaces. As an integral part of the accelerator model, this unit benefits from the location of the data and therefore, no additional interfaces are needed to perform the data movement. To access the distinct memory locations, interfaces M00 and S01 are used, following a multi-master implementation where connections A and B are the masters on the interface, and connection Y represents the target of the operations to be carried out. In this way, each interface implements a scheduling policy through the *Gen-BUS* component.

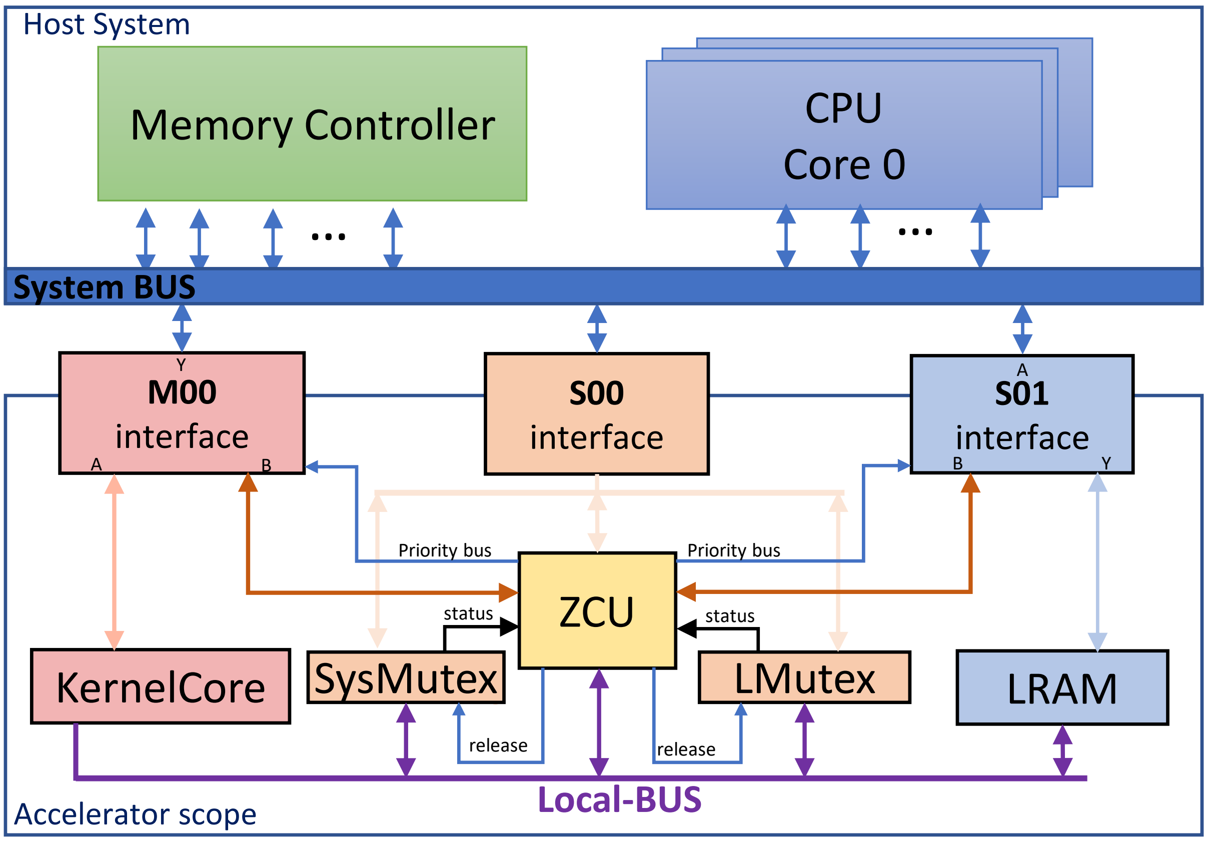


Figure 4.14: ZeroCopy unit - Connectivity in the accelerator model.

To access the ZCU, the host system uses the S00 interface that maps all the registers in the accelerator model. The set of registers for the ZCU unit includes a *Control*, *Status*, *Lock ID* and *Transfer* parameters. On the accelerator side, the *Kernel Core* uses the Local-BUS to access the same registers. To avoid a collision between these two interfaces, the unit imposes a containment model that establishes configuration privileges to the entity that successfully acquired the exclusivity. Such model is based on the internal *Lock*

ID register and the external status of the HW-Mutexes, and provides access to the ZCU to initiate single data transfers using the specified parameters.

When the ZCU needs to access the system memory, it uses the interface M00, and at the same time to access the LRAM, it uses the interface S01. The priority source for each interface is managed by the containment model which in idle, establishes priority to the master A on both interfaces. In doing so, the slave Y is connected to the host system in S01, and to the *Kernel Core* in M00 for the time that ZCU is not in use. Once in service, the priority changes to master B at the moment A completes the current transaction. The ZCU will transfer the required data length and, on completion, automatically remove the configuration privileges while releasing the *HW-Mutexes*. A new data transfer requires again the exclusivity in the dedicated hardware resources, and the Host operating system can contribute with scheduling policies that promote the distribution of ZCU services between the different processing entities that it owns.

Figure 4.15 describes in simplified manner the internal organization of the ZCU. We can see the connections of the S00 and Local-BUS interfaces that allow access to the unit's register area. In the accelerator memory layout, these registers are mapped at the second memory page, and the displayed offsets are aligned with the limits of such paging. The generality of the registers is shared using the containment model and only the *Status* register is specific for each interface. This condition allows one interface to initiate a new transfer before the other has consulted this register.

To control the data transfers, the ZCU datapath implements three counters of parameterizable dimension namely C0, C1 and C2. These specify the offset for the interfaces M00 and S01, and the transfer length using word units. Such counters are parametrized according to the LRAM (N) storage capacity on the accelerator and the system memory page size (P) of the Linux OS. Although the transfer counter is using words, a byte-based transfer length is the accepted parameter. When the specified length is not a sub-multiple of the word, in the last transaction, the control unit adds an extra word to complete the transfer of the remaining bytes. With regard to the effective data movement, the ZCU implements a cross exchange between each interface, where the words from M00 are forwarded to the interface at S01 and vice versa. The control signals for each interface are managed by the ZCU's control unit, as it can be seen in the same figure.

The control register allows to initiate a read (bit 16) or write (bit 17) transfer in the LRAM, or to trigger reset operations (bit 30) and restart (bit 28), to the ZCU unit or M00 interface, respectively. Similarity to other

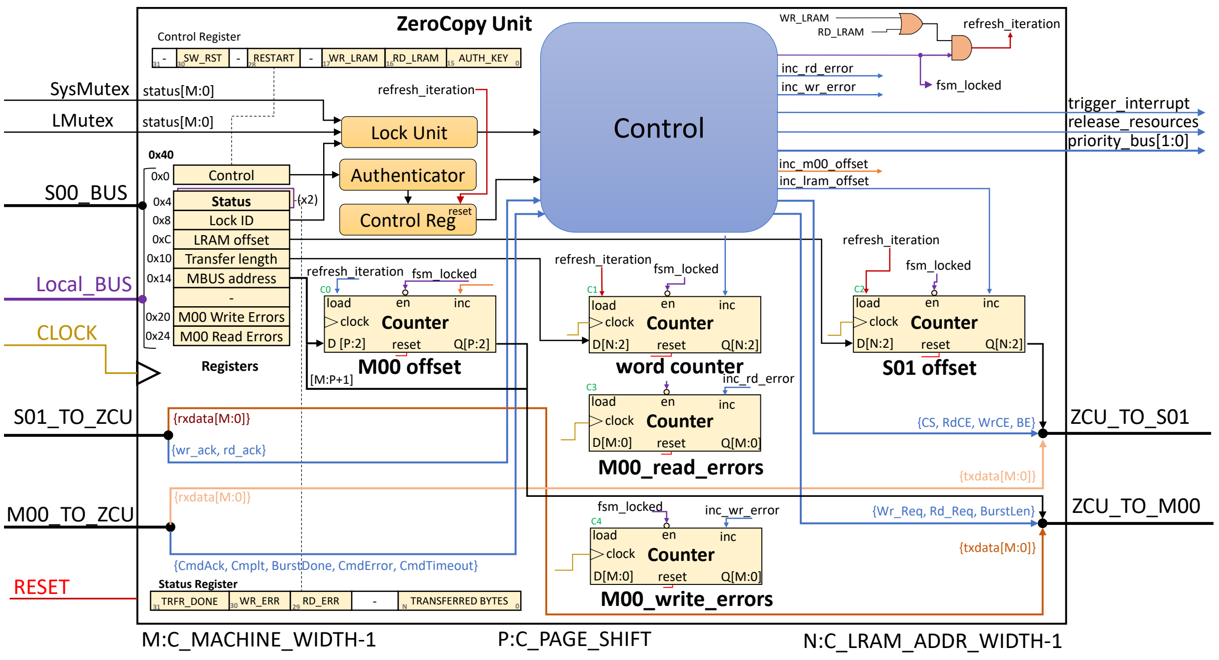


Figure 4.15: ZeroCopy unit - architectural design.

control operations in the accelerator model, writing to this register requires authentication to protect the unit from a control loss in the system. For this purpose, the same authentication unit that can be found in the accelerator kernel is used. At transfer completion, in the status register it is possible to consult the effective number of bytes transferred, or to check for the occurrence of errors. The most significant bit in this register signals that the transfer is complete, as shown at the bottom left of the figure.

Since multiple processing entities that can exist in the host system, may compete with each other for the ZCU resource, it is imposed that such entities can only write on the *Lock ID* register after having successfully acquired the exclusivity through the HW-Mutexes. As such, writing to the transfer registers will only be possible after providing the appropriate word to the Lock ID register. This register, along with the two HW-Mutex status registers are source of the Lock unit. It is through this unit that the ZCU implements the containment model and the diagram that describes its implementation can be seen in Figure 4.16.

The *Lock* unit design encompasses the three input words: *SysMutex* status; *LMutex* status; and *Lock ID*, and the two control signals that synchronize this datapath, *fsm_locked_i* and *trfr_done_i*. At the output, the *host_locked* signal is used to grant writing privileges to the S00 interface or to the Local-BUS, and the *one_time_lock* signal triggers the ZCU from idle to locked state, where it can receive the transfer parameters. The locked flag (bit 31) and CHID flag (bit 30) in the status of the HW-Mutexes, are logically

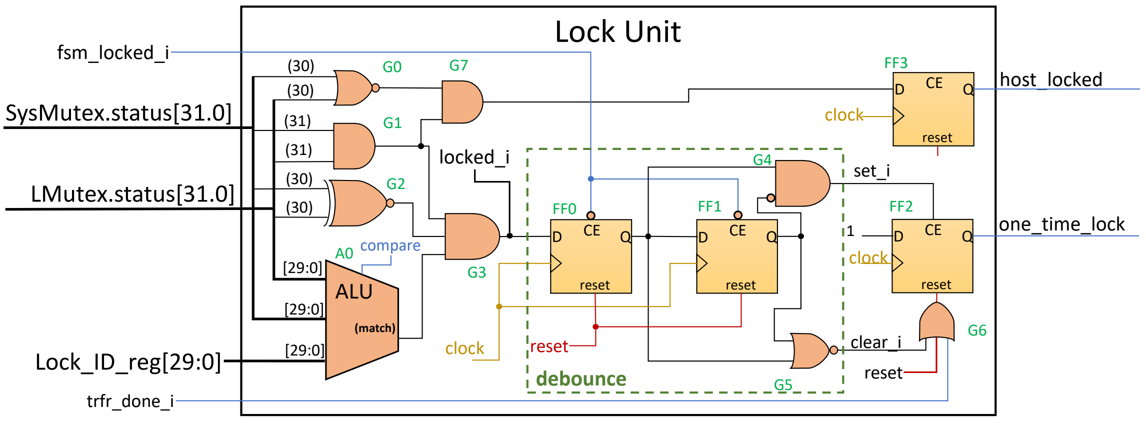


Figure 4.16: ZeroCopy unit - Lock unit.

combined in G1 and G2 to ensure that the two mutexes have been acquired and are at the service of the same interface: S00 or Local-BUS. Using G7, the active low CHD bit will set the *host_locked* signal in FF3. The triple correspondence between the thirty least significant bits, activates the *match* signal in A0, which combined logically in G3 allows to activate the *locked_i* signal. This combination must remain valid for three clock cycles to give rise to the *one_time_lock_i* output. This signal is cleared automatically with the transfer completion by using signal *trfr_done_i* in G6, or otherwise, by releasing the exclusivity in HW-Mutexes, which gives rise to the signal *clear_i* in G5.

Finally, in figure 4.17 we can see a connection diagram that establishes the write privileges between the S00 and Local-BUS interfaces. The clock enable signal (CE), activates the writing of one of the sources A or B in the register. The logical combination of *WR_CE* with the zero value of the *fsm_locked_i* signal, establishes the writing privileges to the Lock ID register (at left) depending on the logical value of the *host_locked* signal.

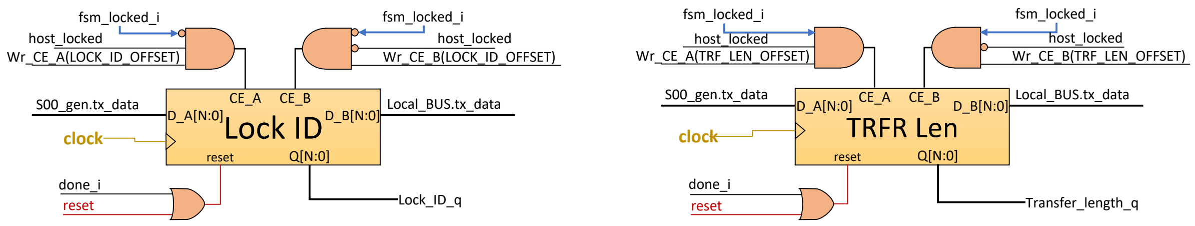


Figure 4.17: ZeroCopy unit - Write enable.

When the corresponding ID is supplied to the Lock ID register, the control unit activates the *fsm_locked_i* signal, and prevents its change at least until the completion of one transfer. On the other hand, the same signal enables writing in the parameter registers, as it can be seen in the diagram that represents the transfer length register (at right). The corresponding CE signal is asserted by with the logical combination of the *WR_CE*, the *fsm_locked*, and according to the logical value of *host_locked* signal. Writing to the control register is determined by the same signal dependence, but the written value first hits the authentication unit, before being transferred to the control register.

The control of the ZCU is implemented according to a state logic that is divided into the two transfer flows: from the LRAM to the system memory; or from the system memory to the LRAM. The control actions include ensuring the one-time-lock transfer, and the subsequent interaction with the M00 and S01 interfaces. Figure 4.18 describes the implementation of the control unit according to a state diagram, which depicts the writing to LRAM flow using the orange color, and the reading from the LRAM flow using the blue color.

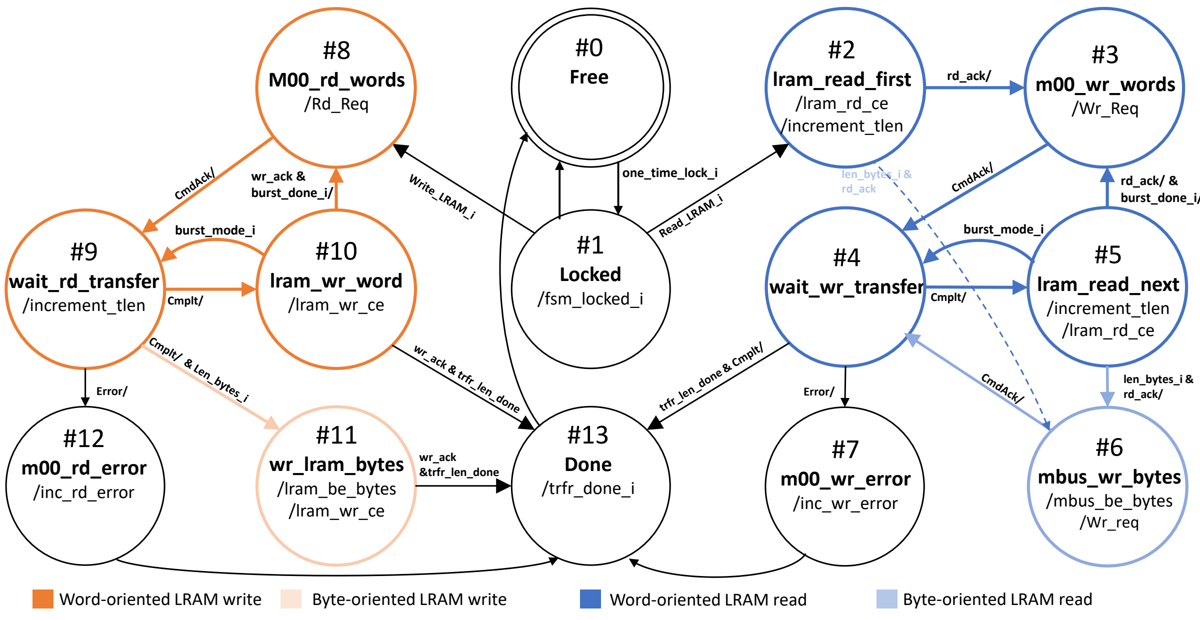


Figure 4.18: ZeroCopy unit - Control unit FSM.

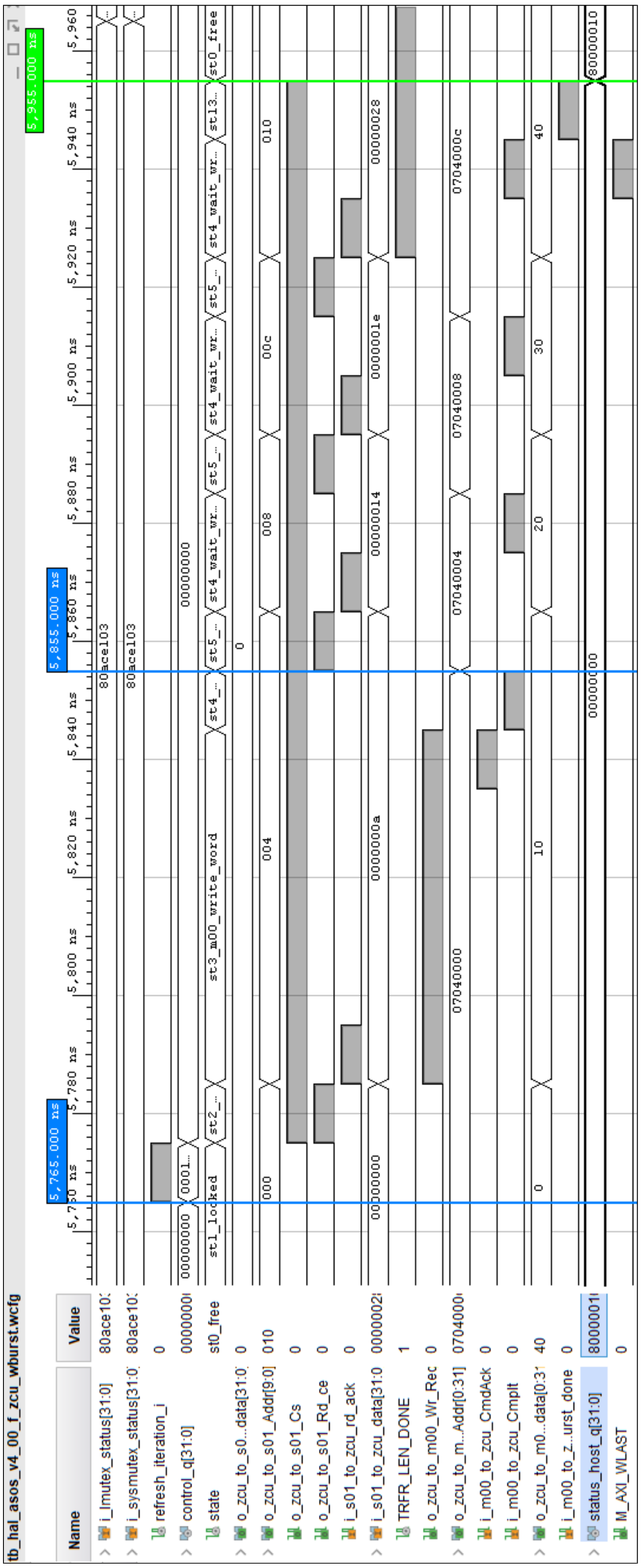
After a reset signal, or a reset operation triggered by software, the unit assumes state #0. During this state it remains idle, turning off the clock sources in the sequential components that make up its datapath by means of a low *fsm_locked_i* signal. With the exclusivity in the resources, the *one_time_lock_i* signal allows the control to switch to #1, *Locked*, and enable writing privileges in the register area.

With the beginning of a new transfer, #2 or #8 are reached, according to the specified operation. State two, #2, is distinguished by the initial assessment of the transfer length, in which, although not desirable, can determine a length inferior to a word and complete the transfer with just one transaction on the M00 interface in #6. The analogous procedure for the write operation can later take place in #9, after the control has read a complete word through the M00 interface, and complete #11 by writing the desired number of bytes in the LRAM.

A transaction on the M00 interface is made in two phases that comprise an unpredictable number of clock cycles. The number of clock cycles depends mainly on the bus technology and the level of congestion to which it is subject to. As such, in the first phase, the desired operation and the address value are specified, followed by the first word of data if the operation writes to this interface. For this, the #3 or #8 is used. The interface responds with the *CmdAck* signal that determines the progress in the state machine. During the second phase, the control waits for the result of the transaction, incrementing the word count and keeping all signals steady, in #4 or #9. In the event of an error, a state that increments the respective counter is assumed, #7 or #12. If otherwise a *Cmplt* signal is received, the control logic initiates a new cycle of reading or writing in the LRAM, through #5 or #10, respectively. At the same time, the address counters for the M00 and S01 interfaces are incremented, using a logical combination that depends on the *increment_tlen* signal and the *Cmplt* signal input.

If the M00 interface allows burst mode transfers, the state machine submits or retrieves the consecutive number of words that comprise the length of a burst. In doing so, it iterates between #5 and #4 when the transaction reads, and #9 and #10 when the transaction writes in the LRAM. The *burst_mode_i* and *burst_done_i* signals are used in burst capable interfaces and allow the control unit to fragment received the transfer length in successive burst transfers on the M00 interface. If otherwise, the M00 interface does not have this feature, the *burst_done_i signal* remains active, imposing the request of a new transaction with every word. In such cases, control switches to #3 and #8 by the time it completes #5 and #10, respectively. Upon reaching the specified transfer length, the control switches to #14 where it signals the completion, giving rise to the interruption stimulus of this device. This state lasts only one clock cycle and the control returns to #0 while releasing the hardware resources.

In Figure 4.19 we can see a wave diagram from a simulation performed on the accelerator V4_00_F, using a burst capable AXI4 interface. In this diagram, the ZCU *Control* implements the flow of states on the right, which reads data from the LRAM to write the system's memory. For simplicity, most of the visible signals are control while others were omitted. Also, the time in the diagram has been moved to the moment when the ZCU receives the control operation. By analyzing the contents in the HW-Mutexes (0x80ace103), we can see that the ZCU operates at the service of the host system with the ID 0xace103.


tb_hal_asos_v4_00_f_zcu_wburst.wcfg

Name	Value
i_lmutex_status[31:0]	80ace10:
i_sysmutex_status[31:0]	80ace10:
refresh_iteration_i	0
control_q[31:0]	0000000(
state	st0_free
o_zcu_to_s0...data[31:0]	0
o_zcu_to_s01_Addr[9:0]	010
o_zcu_to_s01_Cs	0
o_zcu_to_s01_Rd_ce	0
i_s01_to_zcu_rd_ack	0
i_s01_to_zcu_data[31:0]	0000002:
TRFR_LEN_DONE	1
o_zcu_to_m00_Wr_Rec	0
o_zcu_to_m...Addr[0:31]	0704000(
i_m00_to_zcu_CmdAck	0
i_m00_to_zcu_Cmplt	0
o_zcu_to_m0...data[0:31]	40
i_m00_to_z...urst_done	0
status_host_q[31:0]	8000001(
M_AXI_WLAST	0

Figure 4.19: ZeroCopy unit wave diagram - Read the S01 interface and write to the system memory using burst format.

The control remains in #1 until at time instant of 5,765.00 nanoseconds, when the content of *control_q* is updated with the desired operation, concatenated with the authentication key. The datapath of the ZCU receives the refresh iteration signal that reloads counters C0 to C3, and the control switches to #2. One clock cycle is used to read the content of LRAM at the offset zero, which returns the first word 0x0000000a, and with the receiving of the acknowledge (**rd_ack*) the control switches to #3. In this state, the control initiates the memory transaction through the signal **Wr_Req* and waits for the acknowledge. The target address is 0x07040000 and the first word received from interface S01 is present in the output signal **Wr_d*. With the **_CmdAck* signal, #4 is achieved, and a clock cycle later the M00 interface is ready to receive the second word. At time instant of 5,855.00 nanoseconds, control reaches #5 and the sequence of states repeats itself through #4 and #5, following a three-clock cycle per transferred word rate.

At the bottom of the wave diagram, it is possible to see two AXI handshake signals, which regulate data exchange with the M00 interface using a ready-to-receive vs data-write-valid model, which completes with the signal indicating the last word written(**_WLAST*). At time instant of 5,955.00 nanoseconds, the ZCU is available for a new operation and waits for the ID of the next entity. In the status register we can observe the indication of 16 transferred bytes (0x10), concatenated with the active bit done (bit 31). For completeness, the wave plot diagram that resulted from the complementary operation, in reading the system memory using and writing to the interface S01 can be found attached in Listing D.5.

4.7 Performance Counters

In the accelerator model, performance counters are a special purpose set of registers that target specific performance metrics based on the activity of the assigned *HW-Task*. These metrics include: (1) sleep and (2) blocked states, and (3) complete processing rounds in the *HW-Task*; (4) the interrupt latency in the host system; and performance results while accessing the system memory (i.e., using *M00_System*) in (5) write and (6) read transactions. Each metric is based on a record of a particular event, in the duration and number of occurrences, that are subsequently classified between minimum and maximum, and a total duration of the various events. Since this logic resources do not perform any effective processing inside the application scope, the performance counters are an optional feature that can be enabled through corresponding selection in the accelerator's top-level (see Figure 2.28b). In doing so, the *HW-Kernel* instantiates six performance components and maps them in the *S00_Control* address range using specific memory pages, as described in section 3.3.

In the six performance related events, a *Sleep* counter records the number of clock cycles that the *HW-Task* spent while in a sleep state and another counter will record the number of times it went to sleep. Similarly, a *Blocked* counter records clock cycles spent in the blocked state while executing in the *Kernel Core* context. A *Done* counter records clock cycles required to complete each *HW-Task* processing round. To access the performance impact while synchronizing with the host system, an Interrupt counter records the clock cycles spent waiting for the interrupt status acknowledge. Additionally, distinct performance counters record the clock cycles spent waiting for the write or read transactions in the *M00_System* interface. Conducting a low-level performance analysis will allow the assessment of the three *HW-Task* performance metrics, to extract the sleep or blocked results from the overall done results. In a similar way, the blocked results may also be separated from an effective kernel execution or the waiting periods that are mostly related to the synchronization or the *M00_System* transactions.

A single performance counter design is used, which internally relies on: (1) two clock cycle counters for the number of occurrences and the duration of each event; and (2) three ALUs for classifying the minimum and maximum values, and to compute a total of clock cycles from the intermediate results. Figure 4.20 describes the architecture of this performance counter using a logic diagram, where it is possible to observe, in FF0*, a 3-bit control register that allows the starting or stopping the performance counter (*run*, bit 31), or clearing the current metrics (*clear*, bits 1 and 0 simultaneously).

The active *run* bit enables the clock signal in FF1 and counters C0 and C1 and through this register, i.e., FF1, the design creates two internal flags that signal the rise and fall states of the *i_event* input. Such flags, are then used to trigger load and increment inputs on C0 and C1, respectively, or to enable the clock in the FF2*, FF3* and FF4* registers. In this way, C0 will count the clock cycles for as long as the input signal is asserted, and C1 will record the number of rise occurrences in the same signal. The duration of each event is compared with the minimum (FF2*) and maximum (FF3*) registers using A0 and A1, respectively. The logic comparison flag is combined with the *event_fall* flag to enable the clock signal and allow a write of a new value in one of these registers. Similarly, in A2 the current count value is added with the total clock cycle count register (FF4*), and the result of the addition is stored with the active *event_fall* flag.

Figure 4.21 shows a wave plot that results from the simulation diagram using an *HW-Task* that implements six states (i.e., *task_state* in line 2) that allow: (*st1_*) to read a control message; (*st2_*) to process the received message; (*st3_*) to process an array of four words; (*st4_*) to write the processing results to LRAM;

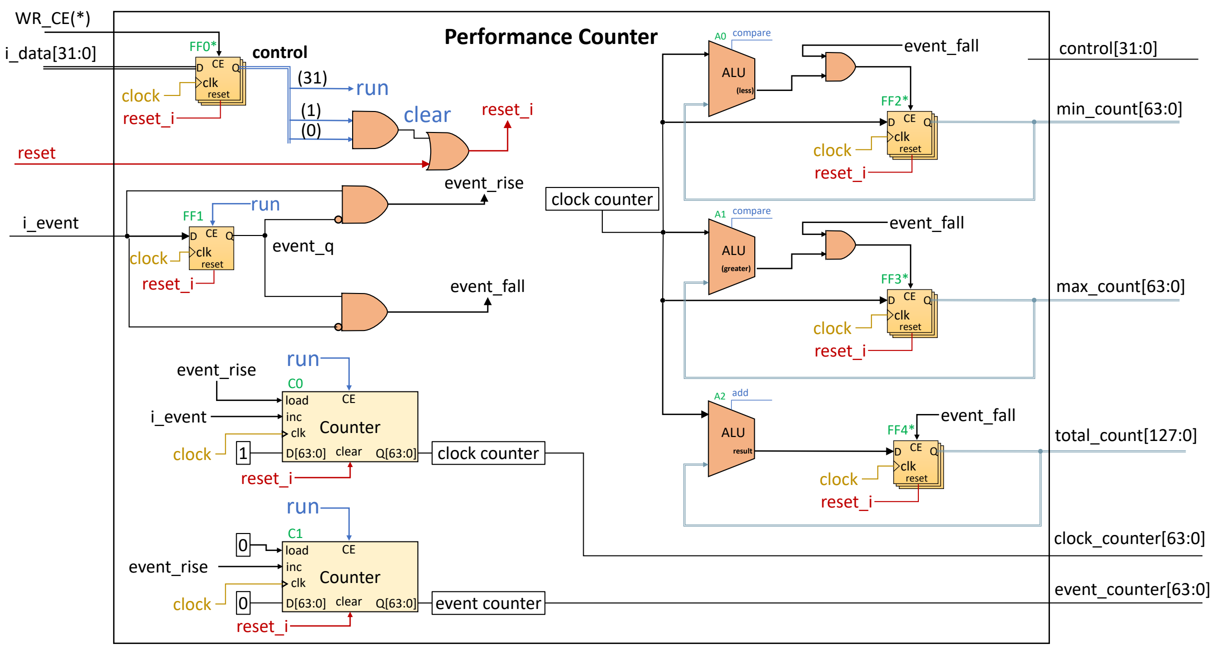


Figure 4.20: Performance Counter - architecture simplified block diagram.

and (*st5_*) to signal the host that the data has been processed. In the same figure, it can be seen an excerpt of the kernel response interface that includes the system call ID, the progress of the microprogram and scheduler, the *block_task* and the valid signals. The bottom of the figure, also shows the internal registers of the *Blocked* performance counter, and the *event_counter* and *clock_counter* registers of the *Done* performance counter.

An analysis of the *HW-Task* execution, shows the processing distribution across the accelerator model, that starts at the instant of time 1,985.00 nanoseconds when the *HW-Task* control receives the *run* signal and transitions to the *st1_* state. In *st1_*, the *HW-Task* invokes a system call to read the Control FIFO in 3 clock cycles, where in the 3^{rd} cycle the *HW-Task* stores the received message in its datapath and sets the next state in the control logic as *st2_*. The execution of this system call results in 2 cycles of blocked status, which are added as minimum, maximum and total of the first detected event. Since the performance counters are based on the *Kernel Core* status register, the stimuli are received in the performance counters one clock cycle after they have occurred and as such the results in the registers reflect this delay. Simultaneously, the *Done* performance counter started counting with the run signal, and records in *st3_* the value of 4 clock cycles for the first detected event. During the course of *HW-Task* processing this performance counter is kept active until completion.

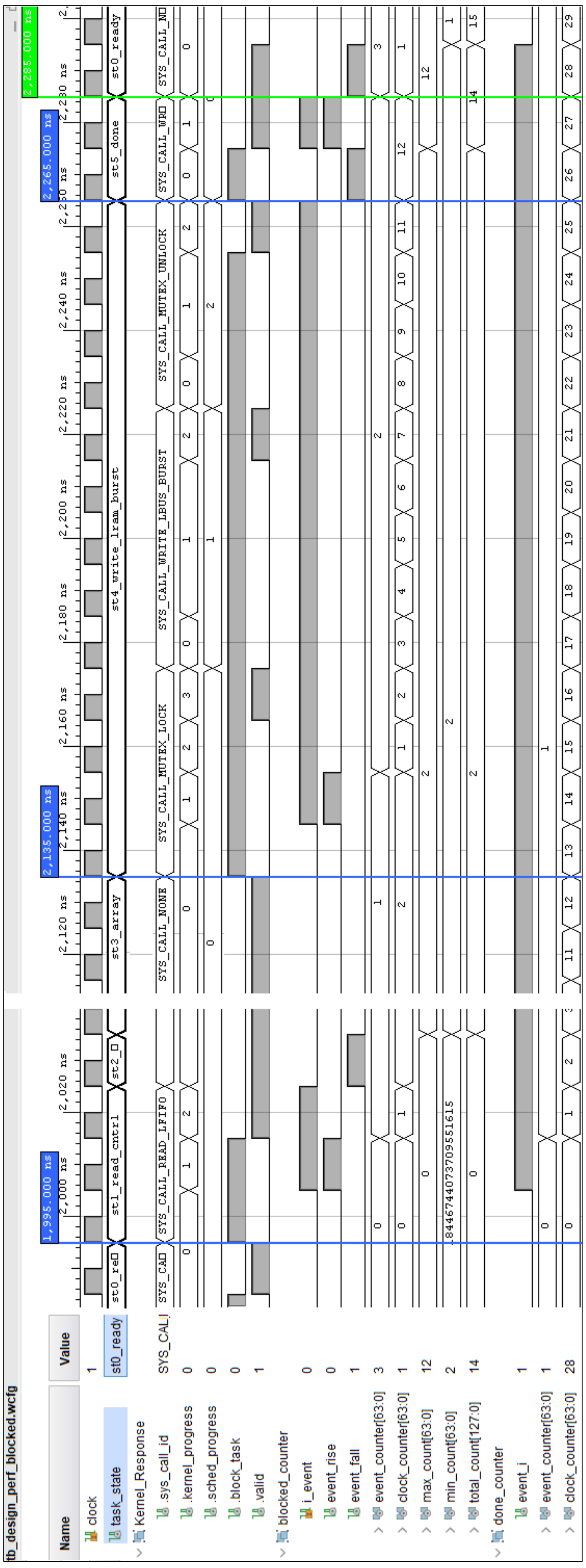


Figure 4.21: Performance counter wave diagram - Blocked counter metrics using example *HW-Task*.

At time instant of 2,135.00 nanoseconds, the control reaches *st4_* and it invokes a user procedure to safely update the LRAM contents, that involves: (0) locking *LMutex*, (1) writing an array of data to *LRAM*, and (2) unlocking *LMutex*. The execution of this procedure completes at instant of time 2,265.00 nanoseconds, with a *Blocked* cycle count of 12 cycles, which are added to the total register that achieves 14 cycles.

At *st5_*, the *Done* counter registers an increase of 13 clock cycles over the 14 cycles registered in the previous state (i.e., 27 cycles). During this state, the *HW-Task* invokes a system call that activates an interrupt through LINTC, and lasts 2 clock cycles, thus registering the minimum *Blocked* event duration with 1 cycle. At the instant of time 2,285.00 nanoseconds, the *HW-Task* completes the processing round with 29 cycles, of which 15 were executed in kernel space, with the longest execution interval of 12 cycles. It should be noted that the *st0_* state is considered a synchronization state, where the kernel disconnects the *HW-Task* from the microprogram while waiting for the host intervention, and for this reason, the performance counters remain disabled.

4.8 Accelerator Versions

The HAL-ASOS framework provides two main versions of the proposed accelerator model that aim at specific bus technologies, most frequent on today's FPGA platforms. The purpose of each version is to provide a flexible design choice, favoring different communication channels and to ease the accelerator connectivity with the specifics of each bus topology. Although a single *HW-Kernel* design fits in the accelerator model, distinct accelerator implementations exist and can be distinguished by the mapping of special purpose interface capabilities with the host system. Up until now, the framework allows to select accelerators based on IBM CoreConnect bus architecture, using Processor Local Bus (PLB) and Device Control Register (DCR) bus in the V3 based versions , or the competing Advanced Microcontroller Bus Architecture (AMBA), using AXI4 bus for the V4 based versions. In each of the versions, the termination letters distinct specific of the bus interfaces. Examples are the use of AXI4 with and without burst capabilities in the data-oriented interfaces, besides providing distinct clock domains in each interface design.

Table 4.1 lists the accelerator features in each of the two numbered versions. Both versions are split between multi- or single-clock designs. The first column, refers to the 'A' terminated version, only available through the HDL sources, describing a tightly coupled design which includes the *HW-Task* on the accelerator model. The 'A' terminated versions are selected for advanced use and to promote changes in the accelerator model. As opposed to 'A', the 'B' to 'F' terminated string versions are eligible from an IP-XACT repository, and they implement loosely coupled designs that plug into an *HW-Task* component. These can also be divided according to a single- or multi-clock design strategy. The 'B' versions provide the simplest design of the accelerator model, that targets low resources in the interface components while providing a single-clock design that eases validation.

Table 4.1: HAL-ASOS accelerator versions V3 and V4 features.

	Single-clock				Multi-Clock	
	Sources	IP-XACT repository			IP-XACT repository	
Versions 3	V3_00_A	V3_00_B	V3_00_C	V3_00_D	V3_00_E	V3_00_F
M00_System	PLB v4.6	PLB v4.6	Co-Simulation	FSL 2.1	PLB v4.6	DCR v2.9
S01_Data	PLB v4.6	PLB v4.6		PLB v4.6	PLB v4.6	PLB v4.6
S00_Control	PLB v4.6	PLB v4.6		PLB v4.6	PLB v4.6	PLB v4.6
Versions 4	V4_00_A	V4_00_B	V4_00_C(_V)	V4_00_D	V4_00_E	V4_00_F
M00_System	AXI4-Lite	AXI4-Lite	Co-Simulation	AXI4-Lite	AXI4-Lite	AXI4-Lite
S01_Data	AXI4-Lite	AXI4-Lite		AXI4	AXI4	AXI4-Lite
S00_Control	AXI4-Lite	AXI4-Lite		AXI4	AXI4-Lite	AXI4

The 'C' versions are used for the Co-Simulation and implement a connectivity through the network service of the operating system where the RTL simulation tool is running. For this, distinct implementations rely on the framework software API and a *proxy* component connects with the *HW-Kernel* interfaces to exchange data with the target application. To provide a simple validation in the early design stages a single-clock design is provided. In the Co-Simulation mode, the framework supports two RTL simulation tools: (1) ModelSim and (2) Vivado Simulator. Since different tools implement specific programming technologies, in ModelSim the accelerator model uses VHDL FLI, which allows a dynamic library to be loaded as a target architecture of a component in the design. With Vivado, the DPI interface for SystemVerilog is used, which similarly allows the calling of functions implemented in a static software library. To distinguish these two versions, we extend a suffix '_V' in the specific versions for Co-Simulation using the Vivado tool.

The 'D' terminated versions allow the application to scale in performance by providing the full capabilities in the data-oriented interfaces. In the CoreConnect based accelerator versions, the *V3_00_D* was intended to explore the Fast Simplex Link (FSL) interface when targeting the MicroBlaze soft-core processor, but it was abandoned mostly due to performance limitations and resource usage in the overall design. In the AMBA-AXI4 based accelerator versions the design uses the same clock domain in S01 and M00, and provides the full capabilities in both interfaces.

To ease the design timing constrains, the multi-clock version will allow an improvement of the design critical path at the expense of a balance in the interface feature capabilities. Each interface has a specific clock domain which include the generic synchronizers as discussed in section 3.3. The design strategy accommodates clock domains to provide the burst format compatible performance. Therefore, a choice can be made between 'E', a host-dependent data movement model relying on the Host system CPU to exchange data between the LRAM and the system memory, or 'F', an accelerator-dependent data movement model relying on the *HW-Task* or the ZCU for the same data exchange. For this reason, the 'E' and 'F' terminated string versions are distinguished by an exclusive burst format in S01 or M00, respectively.

For harmonizing each specific interface capabilities, the *HW-Kernel* design employs configuration concepts using VHDL language features, which establish the appropriate binding between entity descriptions and a specialized target architecture in strategic components. Figure 4.22 shows an excerpt from the *HW-Kernel* descriptions that lists configurations for the *V4_00_B* and *V4_00_F* accelerators. In line 1651, the configuration descriptions instruct the syntheses tool to use a *blank* and a *single_clock* architecture descriptions in the XS00, XS01 and XM00 components. In this way, the design is configured for a single-clock domain and using arbiters to schedule the concurrent transactions that may occur in S01 and M00 interfaces.

Line 1707, shows the configuration for the multi-clock design that instructs the synthesis tool to bind the XS00 component to the *dual_clock* architecture, as depicted in Figure 3.41, and the XS01 and XM00 components to the multi-clock architectures of the *gen_bus* as depicted in Figure 3.45. In this case, the chosen architectures merge the Slave Y clock domain with the Master B clock domain to promote burst transfers format in the M00 interface. As consequence, the burst format transfers must only be used in the arbiter priority scheduling policy, enforced by the *HW-Mutexes* in the accelerator model. A complementary configuration is used in the *V4_00_E* variant that promoted Slave Y to the Master A clock domain. In this case, the burst transfer capability is used by the host system to exchange application data using the S01 interface. The remaining configurations for the V4 variants can be seen in the attached Listing C.32.

```
167 architecture V4_00_b of hal_asos_accelerator is
168 ...
167 for acc_kernel:xhal_kernel use configuration hal_kernel_v4_00_b_config;
168 ...
357 begin
358 ...
369 M01_CONTRL:entity axi_lite_ipif_master
370            generic map (C_M_AXI_ADDR_WIDTH=> C_M00_AXI_ADDR_WIDTH,
371 ...
416 S01_DATA:entity axi_lite_slave_ram_ifif
417          generic map(C_S_AXI_DATA_WIDTH => C_S01_AXI_DATA_WIDTH,
418 ...
501 S00_CNTRL:entity axi_lite_slave_regs_ipif
502           generic map (C_S_AXI_DATA_WIDTH=> C_S00_AXI_DATA_WIDTH,
503 ...
504 ACC_KERNEL: xhal_kernel
505             generic map(C_PEFORMANCE_COUNTER=>C_PEFORMANCE_COUNTERS,
506 ...
625 end V4_00_b;
```

```
193 architecture V4_00_f of hal_asos_accelerator is
194 ...
315 for acc_kernel:xhal_kernel use configuration hal_kernel_v4_00_f_config;
316 ...
383 begin
384 ...
392 M01_CONTRL: entity axi_full_master_ipif
393             generic map (C_M_AXI_ADDR_WIDTH=> C_M00_AXI_ADDR_WIDTH,
394 ...
480 S01_DATA:  entity axi_lite_slave_ram_ifif
481            generic map(C_S_AXI_DATA_WIDTH => C_S01_AXI_DATA_WIDTH,
482 ...
521 S00_CNTRL: entity axi_lite_slave_regs_ipif
522            generic map (C_S_AXI_DATA_WIDTH=> C_S00_AXI_DATA_WIDTH,
523 ...
568 ACC_KERNEL: xhal_kernel
569             generic map(C_PEFORMANCE_COUNTER=>C_PEFORMANCE_COUNTERS,
570 ...
695 end V4_00_f;
```

Figure 4.22: HW kernel - architecture configurations for b and f variants.

Figure 4.23 shows an excerpt of the HAL-ASOS accelerator architecture descriptions for 'B' and 'F' terminated versions using the AMBA AXI4 bus, while promoting the use of the configurations listed in Figure 4.22. At the top of the figure, line 167 specifies the configuration applied to the *acc_kernel* component, indicating that the configuration 'B' should be used, and includes the architectures for each of the unresolved components in the *HW-Kernel* design that perform the appropriate binding.

In lines 369, 416 and 501, the components that connect to each of the *HW-Kernel* interfaces are instantiated, providing the functionalities as specified in Table 4.1 for version *V4_00_B*. In this case, all components implement a correspondence with AXI4-Lite bus, to connect the distinct generic interfaces with the host system. In line 504, the *HW-Kernel* component is instantiated to finalize the design of the accelerator model. It must be said that in the loosely coupled design approach, the *HW-Task* is an external component that connects with the *HW-Kernel* through the *M00_Kernel* and *S00_Task* interfaces of the accelerator.

Similar descriptions are used for the accelerator component in the 'F' version, as can also be seen at the bottom of the same figure. Likewise, in line 315 the configuration of the *acc_kernel* component is specified, and in lines 392, 480 and 521, the components that establish the correspondence between the host system and the *HW-Kernel* interfaces are instantiated. These provide the functionality according to Table 4.1 for the *V4_00_F* version. In this case, the *M00_SYSTEM* component is based on the AXI4 bus, allowing the generic interface to perform burst format transfers while accessing the main system memory. The remaining components implement a correspondence with the interfaces according to the AXI4-Lite bus specification and in line 568, the *HW-Kernel* instantiation completes the accelerator description.

```
167 architecture V4_00_b of hal_asos_accelerator is
168 ...
167 for acc_kernel:xhal_kernel use configuration hal_kernel_v4_00_b_config;
168 ...
357 begin
358 ...
369 M01_CONTRL:entity axi_lite_ipif_master
370            generic map (C_M_AXI_ADDR_WIDTH=> C_M00_AXI_ADDR_WIDTH,
371 ...
416 S01_DATA:entity axi_lite_slave_ram_ifif
417          generic map(C_S_AXI_DATA_WIDTH > C_S01_AXI_DATA_WIDTH,
418 ...
501 S00_CNTRL:entity axi_lite_slave_regs_ipif
502           generic map (C_S_AXI_DATA_WIDTH=> C_S00_AXI_DATA_WIDTH,
503 ...
504 ACC_KERNEL: xhal_kernel
505             generic map(C_PEFORMANCE_COUNTER=>C_PEFORMANCE_COUNTERS,
506 ...
625 end V4_00_b;


193 architecture V4_00_f of hal_asos_accelerator is
194 ...
315 for acc_kernel:xhal_kernel use configuration hal_kernel_v4_00_f_config;
316 ...
383 begin
384 ...
392 M01_CONTRL: entity axi_full_master_ipif
393             generic map (C_M_AXI_ADDR_WIDTH=> C_M00_AXI_ADDR_WIDTH,
394 ...
480 S01_DATA:  entity axi_lite_slave_ram_ifif
481            generic map(C_S_AXI_DATA_WIDTH > C_S01_AXI_DATA_WIDTH,
482 ...
521 S00_CNTRL: entity axi_lite_slave_regs_ipif
522            generic map (C_S_AXI_DATA_WIDTH=> C_S00_AXI_DATA_WIDTH,
523 ...
568 ACC_KERNEL: xhal_kernel
569             generic map(C_PEFORMANCE_COUNTER=>C_PEFORMANCE_COUNTERS,
570 ...
695 end V4_00_f;
```

Figure 4.23: HAL-ASOS accelerator - architecture components and configuration clause.

Chapter 5

Experimental Results

In this chapter, we will evaluate different versions of the HAL-ASOS accelerator synchronous and asynchronous design models that can be applied to HW-Task, using appropriate procedures from the HDL packages. Some of the accelerator versions will be evaluated considering metrics in the overall performance of the target application, by varying resource usage through data size and clock frequencies. To fit different accelerators into the overall performance, the results from the Linux *time* command and the performance counters in the HAL-ASOS file system will be combined and analyzed. The chapter concludes with a gap analysis by evaluating different implemented versions of accelerators and comparing the top performance design with similar state-of-the-art implementations.

To stimulate the design of the HW-Task, a feature detection algorithm was selected as the case study. It is often used as an initial step in many computer vision applications to detect corners of objects in image frames or live video streams. A large number of feature detectors exist in the literature, however, it is still true that when processing live video streams at full frame rate, most of them leave little time for further processing. For better understanding and functional validation, the chosen algorithm was embedded into a test application that locates an object in image frames. The following two sections describe the structure of this application and the chosen detection algorithm.

5.1 Object detection a case study

For a case study, the developed application employs algorithms from the *OpenCV* library [29] to detect and extract image features that will be used to match a known object in the target image frame. Therefore, attributes that characterize specific points (corners) for the chosen object are previously collected and stored using appropriate binary descriptors. These points will be compared with points of interest found in the target images. The application is divided into four different processing steps: (1) the entire image is analyzed to extract the points of interest; (2) a set of binary descriptors is created for the detected points; (3) these points are compared with descriptors of the chosen object, and if a match is found; (4) the located object is circumscribed in the image. Figure 5.1 represents this sequence of steps through a block diagram, where it is possible to observe which of the *OpenCV* library algorithms were selected.

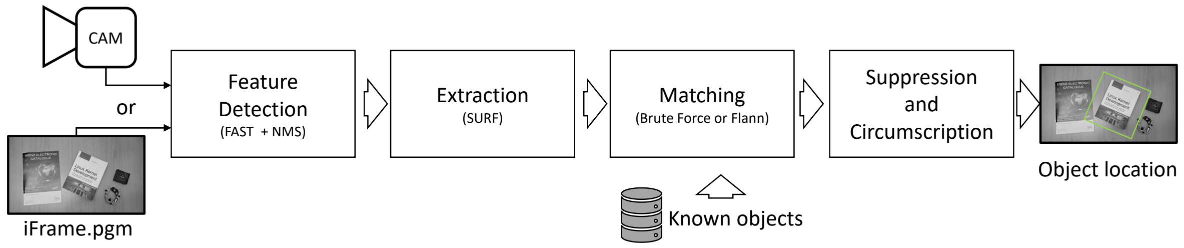
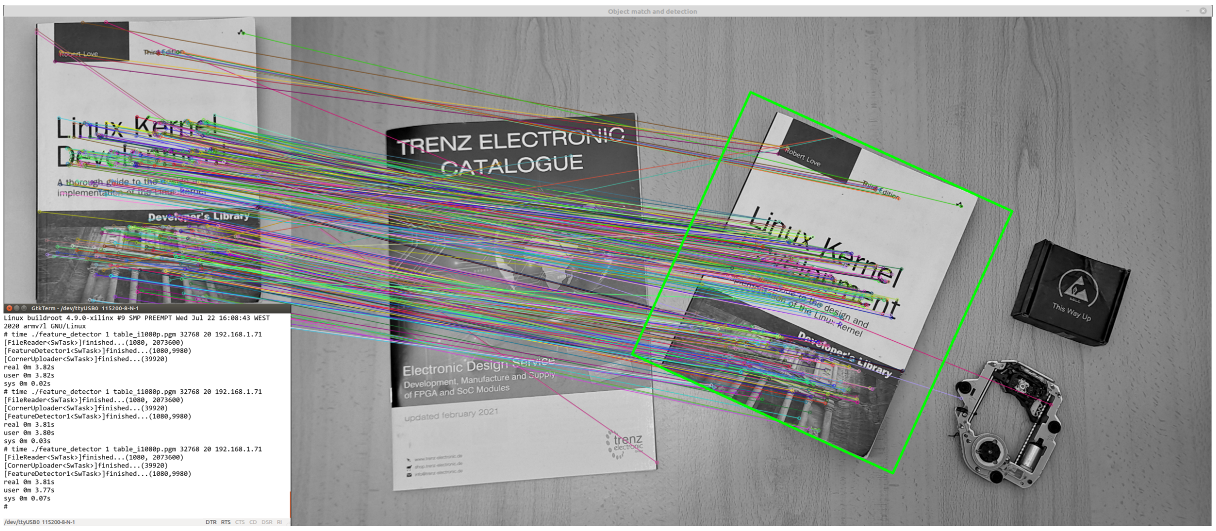


Figure 5.1: Object Detector application block diagram.

The first step uses the FAST+NMS algorithm, and the second step, use SURF algorithm format to generate binary descriptors for the corners found. Using the Brute Force matcher [30] algorithm, the third step establishes a match between descriptors of the chosen object and those from the target image. In doing so, it compares each corner of the chosen object with all corners from the target image and returns the closest classified value using the Euclidean distance. The final step operates on the set of established matches and excludes values with a distance above a specified threshold. Once excluded, the application seeks to delimit the image region described by the remaining matches. It draws a four-point geometric shape defined by the farthest coordinates that can be found in the set of matches.

The above block diagram was implemented following a distributed approach, where the first step (i.e., the feature detection) is deployed into a Xilinx ZC702 platform capable of implementing the HW acceleration models provided by the HAL-ASOS framework. Using the Linux network subsystem, corner points are detected and uploaded while traversing the target image. The receiver of this data is a server system that proceeds with the implementation of the remaining steps, providing the final representation of the image containing the circumscribed object. For the performed tests, different image resolutions and different objects were previously tested under different scenarios, ensuring applicability and compatibility between the algorithms. In Figure 5.2 we can see the application output for an image frame with 1080p resolution.

The upper left corner shows the target object image that was used in the match step. On the right side, it is image frame that was analyzed by the feature extraction step, which includes the target object. After


Object match and detection
GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# time ./feature_detector 1 table_i1080p.pgm 32768 20 192.168.1.71
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetector1<SwTask>]finished...(1080,9980)
[CornerUploader<SwTask>]finished...(39920)
real 0m 3.82s
user 0m 3.82s
sys 0m 0.02s
# time ./feature_detector 1 table_i1080p.pgm 32768 20 192.168.1.71
[FileReader<SwTask>]finished...(1080, 2073600)
[CornerUploader<SwTask>]finished...(39920)
[FeatureDetector1<SwTask>]finished...(1080,9980)
real 0m 3.81s
user 0m 3.80s
sys 0m 0.03s
# time ./feature_detector 1 table_i1080p.pgm 32768 20 192.168.1.71
[FileReader<SwTask>]finished...(1080, 2073600)
[CornerUploader<SwTask>]finished...(39920)
[FeatureDetector1<SwTask>]finished...(1080,9980)
real 0m 3.81s
user 0m 3.77s
sys 0m 0.07s
#
```

Figure 5.2: Object Detector application test.

establishing probable matches and excluding those with a higher distance, the application draws lines between the corners on the object and scene images, and completes by circumscribing the object with the square in green. In the lower left corner, the terminal window of the ZC702 platform is also visible, which in this example is using a software-based *Task* class (i.e., *SW-Task* FeatureDetector1) to detect and extract corners on the entire scene image. In this software-only version, two adjacent *SW-Tasks* (i.e., *FileReader* and *CornerUploader*) are used to read image pixels and send the detected corners to the server application, respectively.

Consecutive iterations of the application show stable quantitative results, in which the *FileReader* processes 2,073,600 bytes, representing 1080 blocks (same as image lines) of 1920 pixels. *FeatureDetector1* extracts 9,980 corners in the scene image and finally, *CornerUploader* converts the corners found in a block of 39,920 bytes that it transmits to the server system. This block is composed of a two-dimensional coordinate, x and y for each 32-bit corner. To complement these results, the command *time* was used to measure the execution duration, which records an average time of 3.81 seconds

5.2 Feature detection stage

For the *Feature Detection* stage, the FAST algorithm (accelerated segment test features) [31] was selected. Purposed by Rosten, E. and Drummond, T. [32], is based on pixel intensity comparison using an 8-bit black and white color scale. Such comparison aims to detect corners by selecting the darkest or brightest pixels and extract their location using xy image coordinates. To delimit the brightest or darkest region, the FAST algorithm employs a geometric shape described by a Bresenham circle, which establishes a comparison periphery and places the pixel under test in the central position, I_p. Figure 5.3 shows an example of this geometric shape considering a three-pixel radius, which results in a comparison of bright and dark with 16 pixels.

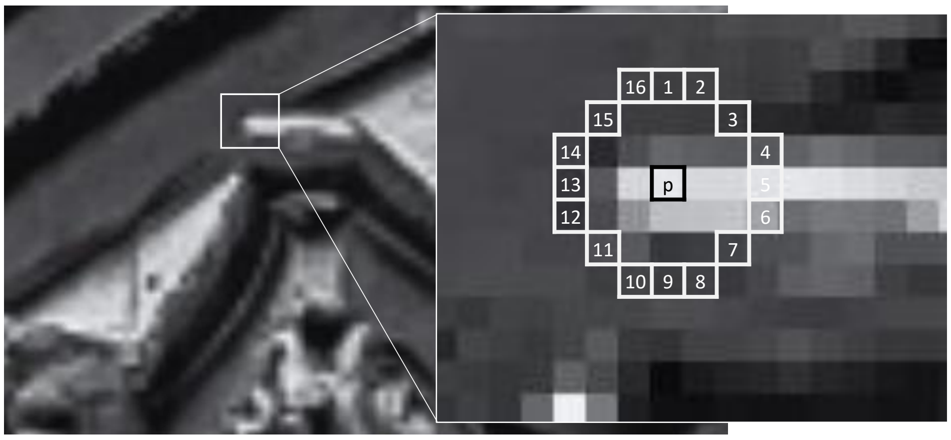


Figure 5.3: Image mapping to Bresenham circle example.

If the intensity of the central pixel (I_p) is higher (darker) or lower (brighter) than the intensity of the pixels in its periphery, and verifies at least 9 consecutive pixels in the arc described by the circle (i.e., Fast9-16), the I_p pixel is considered a key point. To eliminate cases of proximity between pixels, the absolute difference between intensities must be greater than or equal to a threshold value, t, which is a parameter in the FAST algorithm. In such way, the pixel on the periphery ($I_{p \to n}$) is darker or brighter than the central pixel if one of the following expressions is satisfied:

$$x \text{ is brighter} : I_{p \to x} < I_p - t \tag{5.1}$$

$$x \text{ is darker} : I_{p \to x} > I_p - t \tag{5.2}$$

Smaller radius-based versions of FAST exist, where it tests for a darker or brighter contiguity at 5 of 8 pixels (i.e., Fast5-8), or at 7 of 12 pixels (i.e., Fast7-12). In the proposed use case, we selected the FAST9-16 which is the most robust of the three algorithm versions.

For a key point to be considered a corner, first the absolute difference must be quantized through a score function (V), and it also must have a maximum V in its region of adjacent pixel scores. Three forms of computation are possible and these are based: (1) on the maximum number of pixels in the circle, for which p remains corner: (2) on the maximum value of t for which p remains corner; and (3) the sum of the absolute differences between the pixels in the contiguity arc and the center pixel. Definitions (1) and (2) are highly quantized measures but many pixels share the same value of them. For speed of computation, a slightly modified version of (3) can also be used. For the *Feature detection* stage, we have selected (3) since it provides a wither score range. Equation 5.3 represents the modified speed version used in this work. Subsequently, a non-maximum suppression (NMS) rule is applied to eliminate candidates that have adjacent key points with a higher score value. The final set consists of the central pixels that are corners in the image.

$$V = max\left(\sum_{x\epsilon S_{bright}} |I_{p\to x} - I_p| - t, \sum_{x\epsilon S_{dark}} |I_p - I_{p\to x}| - t\right) \tag{5.3}$$

In its original form, the FAST algorithm establishes two acceleration techniques: (1) a high-speed test that pre-examines pixels 1,5, 9, and 13 (the four compass directions). Pixel p will never be a key point if it does not check at least three of the previous pixels as brighter or darker simultaneously, in which case it will be excluded avoiding the consequent processing. The full test with all 16 pixels on the periphery is applied to the candidates who have passed this first test.

Alternatively, (2) is based on machine learning, and for this case the algorithm in format (1) is used to previously analyze images similar to the chosen scenery. From this analysis, a set of key points should be obtained containing the result of the 16 auxiliary pixels, classified into three categories: brighter, darker and similar. These three data subsets are applied recursively using the ID3 (decision tree classifier) algorithm to create a decision tree that can be used to analyze the scenery or other similar images. The acceleration in (2) restricts the application to a specific image type and for this reason it was discarded. The FAST algorithm provided by the *OpenCV* software libraries and a C/C++ implementation provided by the author of the algorithm are based in (1). Both options were tested and used as a performance comparison term with distinct HW accelerated implementations that will be discussed in this chapter. We will refer these implementations as *OpenCV* and *Ed.Rosten-C*, respectively.

A software algorithm that employs the full dataset test and uses the performance-optimized score function of equation 5.3, was also implemented. Its design is based on the C/C++ language and serves as a starting point for the development of the HW accelerated feature detection. The scoring function was selected based on the dimension and precision of produced results. When compared with *OpenCV* and

Ed.Rosten-C algorithms, the suppression of maximums considers 11-bit of dimension of scores against 8-bit of the previous ones. As such, it allows greater precision in the exclusion of non-maximum, and when used in real life images, it demonstrated increased corner location accuracy.

Fundamentally, the software implementation of the full dataset test establishes the starting point in the development of *HW-Task*'s datapath. In doing so, it provides means to validate its internal structure before offloading the computation to specialized hardware circuitry. In its final version, the implementation mirrors the datapath behavior used in the different *HW-Tasks* that will be analyzed in this chapter. The following section outlines the software implementation for the full dataset test.

5.3 Software-only Accelerated Feature Detection

Both algorithms, *OpenCV* or *Ed.Rosten-C*, are characterized by a test dataset that operates on a complete image frame. Such condition implies that the scene image must be completely loaded in the application's memory before processing. Another implementation detail of the previous algorithms is related to the fact that the first three and last three lines of each frame are not considered, as well as the first three and last three columns of the same frame. On the other hand, results will only be obtained when completing the processing of the entire frame.

Unlike the previous ones, the new full dataset test software is centered on seven lines of the frame, and with each new line, the oldest received line is discarded and the processing is repeated for this new set of seven lines. Although it has not received the first seven lines, the implementation proceeds using white pixel lines provided internally, which are later replaced by the lines received from the target image. All frame rows and columns are considered and the first three pixels of each row are processed with the last three pixels in the same rows to form a circle of radius 3. Within this design structure it is also possible to start processing a new image frame while the algorithm processes the last five lines of the previous frame. If, on the other hand, the algorithm receives the last line that completes a frame and there is no new frame to process, the processing is considered complete, leaving 3 lines that did not reach the center of the circle. Figure 5.4 describes the internal architecture for this processing step.

To form a Bresenham circle over the seven lines of the image, a 7x7 pixel matrix structure is used. Each pixel that occupies the center of the matrix is the central pixel p, and together with the 16 pixels in its periphery, they undergo through three processing stages, namely: *Classifilter*, *Contiguity check* ,

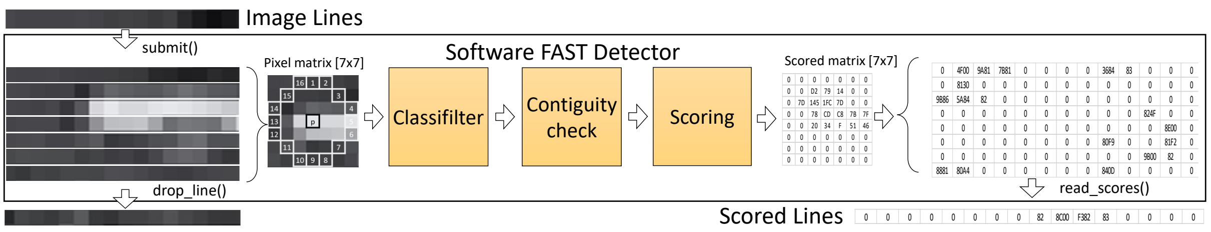


Figure 5.4: Block Diagram for the Software Full dataset test.

and *Scoring*. The *Classifilter* stage calculates the absolute differences to each of the 16 pixels on the periphery and checks whether each of them is brighter or darker than the central pixel p. The *Contiguity* check stage evaluates whether p is darker or brighter for a contiguity of 9 consecutive pixels in the arc describing the circle. If contiguity is verified, pixel p is classified as a key point in the image. The last stage, *Scoring*, calculates the score of the central pixels that have been classified as key points. For each calculated central pixel, the matrix advances one column in the image lines, and the processing resumes with a new circle of pixels for the new central pixel p until this position reaches the end of the line. At this point, a new image line is required for further processing.

The result of the first traversal of the scenery image transforms each line of pixels, into lines of 16-bit scores, where pixels that were not considered key points received a null score. Further processing implements a new traverse over this new image representation and applies non-maximum suppression to discard key points that are not considered corners. In this step, only 3 score lines are considered and the algorithm operates in a similar way to the previous step, discarding the oldest line for each new line received. Processing starts by receiving the second line while considering an internal third line of null scores. Because these operations are independent, the non-maximum suppression stage competes with the previous stages, allowing early memory release and producing results before the entire frame is processed. Figure 5.5 describes the structure and memory layout used for the NMS stage.

The 3x3 matrix of scores uses a 16-bit dimension that shifts across the entire length of lines, while being input parameters to the suppress stage. Such stage will determine if the central score in the position (1,1) is the maximum value between the 8 scores in the proximity. When the condition is verified, the coordinates of the pixel that occupies the center of the input matrix are stored in a FIFO of corners. For this, the algorithm implements counting of lines and columns, which starts with zeros upon receiving the first line. The count is incremented with advances in the matrix position and whenever a new line is sent for processing, until the number of lines in the image is reached. When the last center line score is compared, processing stops until a new line is submitted, starting a new processing iteration. As in the previous algorithm, the first and last center line scores are compared using the last and first values in each of the three lines, to favor regular execution while analyzing all values.

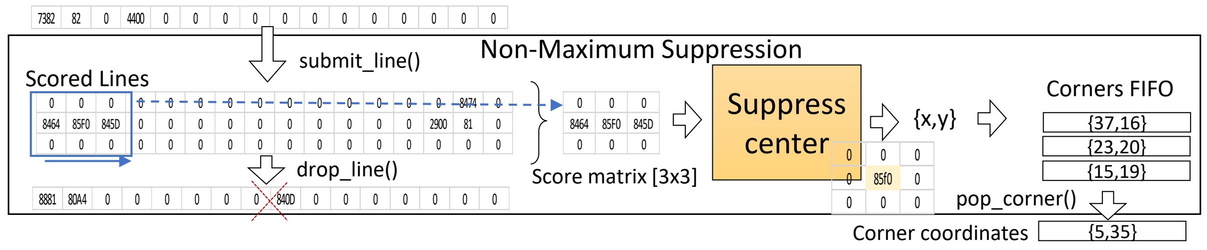


Figure 5.5: Block Diagram for the Software Non-maximum suppression.

As previously mentioned, the software feature detection was parallelized using three processing threads, to take advantage of the dual-core architecture on the target platform CPU. The implementation makes use of the *Task* class in the HAL-ASOS C/C++ framework and applies the *SwTask* template qualifier to specify the desired computational resource. In this way, the first *Task* class, *FileReader*, manipulates the input image by fragmenting data into memory blocks with the dimension of the image width. Such blocks are then published in a topic called *ImageLines* that this task creates using the DDS of the HAL-ASOS framework. The second *Task* class, *FeatureDetector*, subscribes to the topic of the previous task to receive the target image lines, and in its turn, creates a topic called *Corners* to publish the results of its processing. In its implementation, this task stimulates the FAST+NMS algorithms with the image lines it receives from the subscription, exchanging score lines between these two algorithms. When the storage in the corners FIFO reaches the topic length, the execution publishes the intermediate results in the *Corners* topic and resumes processing until a new length is achieved or its subscription is terminated by the *FileReader* task.

Two implementations are available for the third *Task* class which subscribes to the topic *Corners* in both cases. The first one, *CornerUploader*, uploads data through the network subsystem so that the execution can proceed in the server system computational resources. The second, *CornerDump*, stores the processing results locally using a file in binary format. In doing so, it allows for further comparison of results between hardware and software implementations of the FAST+NMS algorithm. On the other hand, it allows for multiple and repeated iterations to measure the average execution times, while excluding delays of the network handshake phase and uploading of corners. Figure 5.6 shows source excerpts in C/C++ language used to implement these two software applications using the HAL-ASOS framework.

On the right side, Figure 5.6a, is represented the code excerpt of the network-based application. In this excerpt, the three SW-based *Task* class instantiations are visible in lines 928 to 930. The T2 instantiation specializes the class using the configurations of the *TCornerUploader* structure, and therefore, it is compiled and linked to the specialized the *run* member that implements the corner upload using the network service. On the left side, line 953 specializes the *Task* class T2 for the configurations of the *TCornerDump* structure, and specialized *run* member that dumps the detected corners in the binary output file. The configurations of each *Task* class as well as each specialized run members can be consulted in the attached Listing C.34 to Listing C.36.

```
925  void hal_asos_demo::
926  feature_detector::sw_mxthread_network(void) {
927      using namespace hal_asos;
928
929      Task<SwTask, TFileRead> T0;
930      Task<SwTask, TFeatureDetector>  T1;
931      Task<SwTask, TCornerUploader > T2;
932      p_Detector_Task = &T1;
933
934      T0.start();
935      T1.start();
936      T2.start();
937      T0.join();
938      T1.join();
939      T2.join();
940  }
```

(a) Network-based source code.

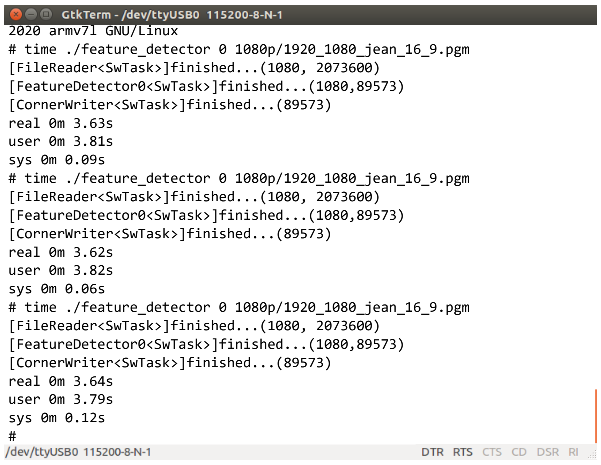
```
947  void hal_asos_demo::
948  feature_detector::sw_mxthread_dump(void){
949      using namespace hal_asos;
950
951      Task<SwTask, TFileRead> T0;
952      Task<SwTask, TFeatureDetector>  T1;
953      Task<SwTask, TCornerDump> T2;
954      p_Detector_Task = &T1;
955
956      T0.start();
957      T1.start();
958      T2.start();
959      T0.join();
960      T1.join();
961      T2.join();
962  }
```

(b) File dump-based source code.

Figure 5.6: Hardware accelerated feature detection application using HAL-ASOS framework.

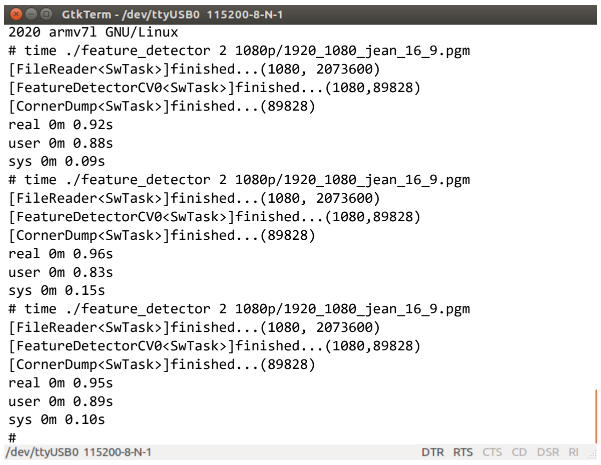
Conceptually, this software application could be structured with a single thread that entirely reads the image, implements the FAST+NMS algorithms and publishes the results of found corners. In terms of compatibility with *OpenCV* and the *Ed.Rosten-C*, this would be the preferable option since these algorithms demands for the full image acquisition in a single shot. On the other hand, the *Ed.Rosten-C* implementation requires the complete image at consecutive virtual memory addresses. As such, the HAL-ASOS and *OpenCV* based detectors will be evaluated in both formats, multi-threaded and single-threaded, while the *Ed.Rosten-C* version will only be evaluated in the single-threaded format. In Figure 5.7 we can see the results of tests performed on detectors based on HAL-ASOS full dataset test, *OpenCV*, and *Ed.Rosten-C* implementations.

Figure 5.7a shows that the detector was executed with the parameters 0 and path to a test image with 1080p resolution. The first parameter selects the multi-threaded version of the HAL-ASOS tool, and the second parameter indicates the image to process. The *time* command was used to measure the execution of the application for three consecutive executions. The total execution time varies between 3.62 and 3.64


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

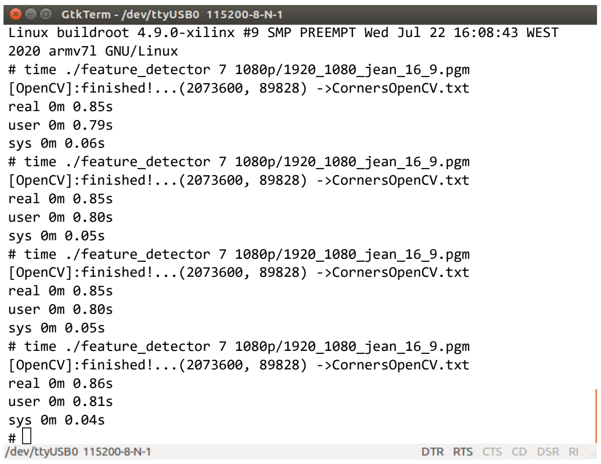
```
2020 armv7l GNU/Linux
# time ./feature_detector 0 1080p/1920_1080_jean_16_9.pgm
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetector0<SwTask>]finished...(1080,89573)
[CornerWriter<SwTask>]finished...(89573)
real 0m 3.63s
user 0m 3.81s
sys 0m 0.09s
# time ./feature_detector 0 1080p/1920_1080_jean_16_9.pgm
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetector0<SwTask>]finished...(1080,89573)
[CornerWriter<SwTask>]finished...(89573)
real 0m 3.62s
user 0m 3.82s
sys 0m 0.06s
# time ./feature_detector 0 1080p/1920_1080_jean_16_9.pgm
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetector0<SwTask>]finished...(1080,89573)
[CornerWriter<SwTask>]finished...(89573)
real 0m 3.64s
user 0m 3.79s
sys 0m 0.12s
#
```

(a) software multi-threaded full dataset test detection.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

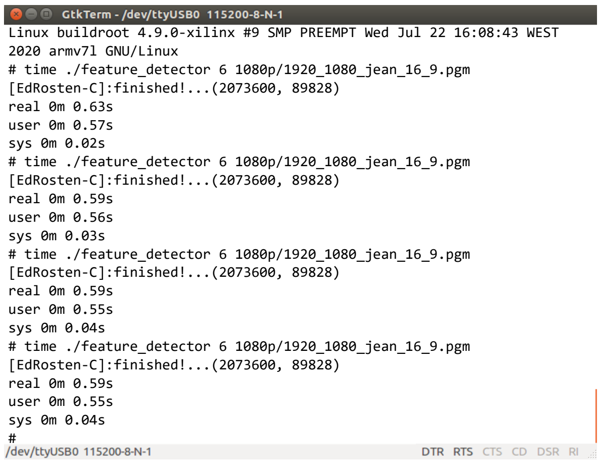
```
2020 armv7l GNU/Linux
# time ./feature_detector 2 1080p/1920_1080_jean_16_9.pgm
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetectorCV0<SwTask>]finished...(1080,89828)
[CornerDump<SwTask>]finished...(89828)
real 0m 0.92s
user 0m 0.88s
sys 0m 0.09s
# time ./feature_detector 2 1080p/1920_1080_jean_16_9.pgm
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetectorCV0<SwTask>]finished...(1080,89828)
[CornerDump<SwTask>]finished...(89828)
real 0m 0.96s
user 0m 0.83s
sys 0m 0.15s
# time ./feature_detector 2 1080p/1920_1080_jean_16_9.pgm
[FileReader<SwTask>]finished...(1080, 2073600)
[FeatureDetectorCV0<SwTask>]finished...(1080,89828)
[CornerDump<SwTask>]finished...(89828)
real 0m 0.95s
user 0m 0.89s
sys 0m 0.10s
#
```

(b) software multi-threaded OpenCV-based detection.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# time ./feature_detector 7 1080p/1920_1080_jean_16_9.pgm
[OpenCV]:finished!...(2073600, 89828) ->CornersOpenCV.txt
real 0m 0.85s
user 0m 0.79s
sys 0m 0.06s
# time ./feature_detector 7 1080p/1920_1080_jean_16_9.pgm
[OpenCV]:finished!...(2073600, 89828) ->CornersOpenCV.txt
real 0m 0.85s
user 0m 0.80s
sys 0m 0.05s
# time ./feature_detector 7 1080p/1920_1080_jean_16_9.pgm
[OpenCV]:finished!...(2073600, 89828) ->CornersOpenCV.txt
real 0m 0.85s
user 0m 0.80s
sys 0m 0.05s
# time ./feature_detector 7 1080p/1920_1080_jean_16_9.pgm
[OpenCV]:finished!...(2073600, 89828) ->CornersOpenCV.txt
real 0m 0.86s
user 0m 0.81s
sys 0m 0.04s
#
```

(c) software single-threaded OpenCV-based detection.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# time ./feature_detector 6 1080p/1920_1080_jean_16_9.pgm
[EdRosten-C]:finished!...(2073600, 89828)
real 0m 0.63s
user 0m 0.57s
sys 0m 0.02s
# time ./feature_detector 6 1080p/1920_1080_jean_16_9.pgm
[EdRosten-C]:finished!...(2073600, 89828)
real 0m 0.59s
user 0m 0.56s
sys 0m 0.03s
# time ./feature_detector 6 1080p/1920_1080_jean_16_9.pgm
[EdRosten-C]:finished!...(2073600, 89828)
real 0m 0.59s
user 0m 0.55s
sys 0m 0.04s
# time ./feature_detector 6 1080p/1920_1080_jean_16_9.pgm
[EdRosten-C]:finished!...(2073600, 89828)
real 0m 0.59s
user 0m 0.55s
sys 0m 0.04s
#
```

(d) software single-threaded Ed.Rosten/C-based detection.

Figure 5.7: Software-only feature detection: application performances when using input and output files and *time* command on the target platform Xilinx ZC-702.

seconds, and in all tests the *user + sys* processing time exceeds the real time since the two cores were at the service of the application. From the messages sent by each of the tasks, we can see that *FileReader* published 1080 blocks of data in a total of 2,073,600 bytes. The *FeatureDetector0* task received 1080 blocks of data representing the image lines and was able to detect 89,600 corners that it publishes in the *Corners* topic. Finally, the task *CornerDump* received the same number of corners that it submits in the binary file for further analysis.

In Figure 5.7b, it can be seen the tests performed to *OpenCV* multi-threaded version using the *Task* class from the HAL-ASOS framework. The same file was used as input image and the test was repeated three times wrapped in the Linux *time* command. The processing results differ in the number of corners found since the *OpenCV* scoring function is based on the maximum threshold as mentioned in the section 5.2. The total execution time varies between 0.92 and 0.96 seconds for 89,928 corners. Results of *OpenCV* single-threaded software can be seen in Figure 5.7c with a total time of 0.85 seconds for the same input image. Finally, Figure 5.7d shows the results of the *Ed.Rosten-C* software test where a total time of 0.59 seconds was needed for an execution that detected the same number of corners as *OpenCV*.

When compared, we can say that *OpenCV*'s execution times are aggravated by multithreading since this algorithm, despite allowing adding lines to the image matrix, cannot start processing until obtaining the entirety of the image. Despite this condition, a better performance is observed in all three versions when compared to the full dataset test software. The tests used an input file that produces a high number of corners, which means that the SW accelerated algorithms are suited to image-controlled environments where the number can be contained. As expected, the performance of the full dataset test version is much lower than the two algorithms with software accelerations, and its implementation helps to translate the costs of this algorithm when using a dual core A9 ARM architecture. For a better understanding of the performance of each application when influenced by the number of detected corners, the tests were repeated with different images at the same resolution, the results can be seen in the graph of Figure 5.8.

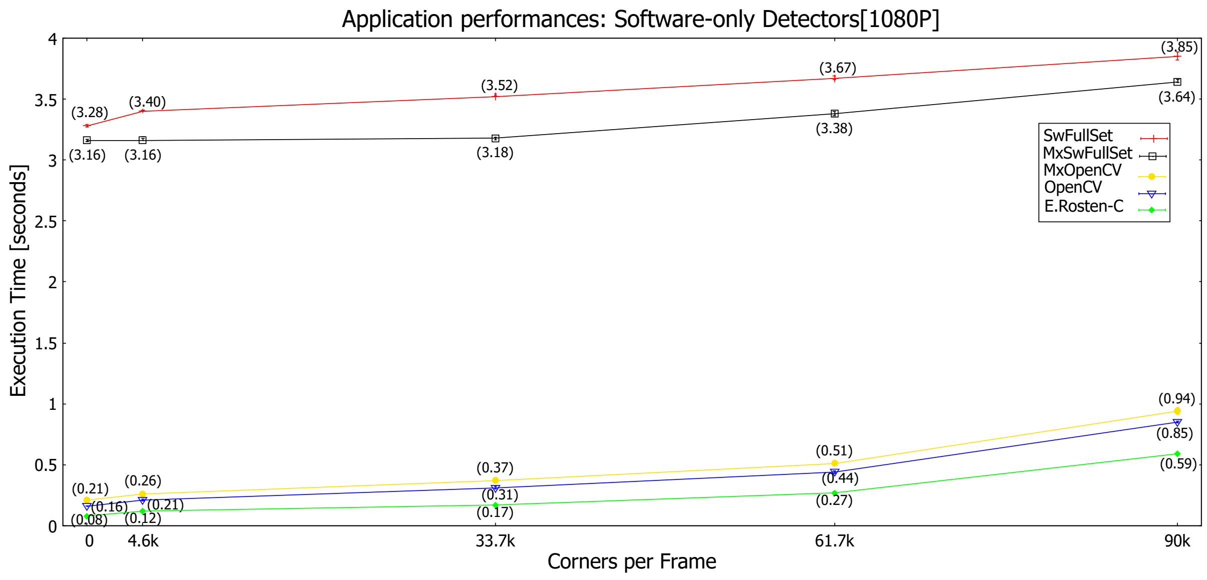


Figure 5.8: Software-Only feature detection performances using 1080p test images.

With zero detected corners, the application based on the *Ed.Rosten-C* algorithm traverses the entire image in 80 milliseconds. Single-threaded *OpenCV* takes an average execution time of 160 milliseconds and suffers a 40-millisecond slump in multi-threaded version for the same image conditions. The execution times of the full dataset are in great number superior, about 19 and 39 times higher to the two previous algorithms when the target image does not present too many number of corners. The differences are reduced to 4 and 6 times with an increase in the number of corners for the last mark (90k) as previously shown in Figure 5.7. The tests performed applied a threshold parameter of 30 to scan the four images that can be consulted in the attached Figure D.10. For the zero corners test, the threshold parameter was raised to the maximum value (i.e., 255), and the *table.pgm* image file was used.

5.4 Asynchronous-synchronous datapath

Traditional FPGA IPs use only synchronous design style due to a rapid development and eased functional verification. In this approach, all state holding elements update their values upon arriving of an active and global clock edge and the pipeline operates like a giant shift-register. Generically, this type of designs suffers penalties in the power consumption metrics by using dynamic power. Also, they face performance limitations due to unbalanced pipelines that result in all stages typically having roughly balanced delays.

Asynchronous designs inherently provide elasticity by allowing a variable number of data items to appear in the pipeline at any time. If there is no congestion, data items are injected at wide intervals, widely spaced in the pipeline and travel rapidly through it. Instead of clock-driven data movement, asynchronous pipelines use some form of handshaking protocol where items travel to the pipeline in availability fashion. Such design inherently offers underflow and overflow protection in the local storage resources and automatic flow control. Since switching activity occurs on data items being processed, asynchronous pipelines consume dynamic power only on demand.

In the HAL-ASOS design, we have been using asynchronous-synchronous design approach that ensures that synchronous circuits are used for computation only and asynchronous circuits ensure that functional units maintain their clock input actives only when it is necessary. By design choice, the *HW-Task* already provides the kernel control signals *sleep* and *blocked* that can be used to disable synchronous functional units on its datapath. Such signals are generally assigned to clock-enable inputs of the synchronous elements when disabling them not in required, such as when the *HW-Task* enters kernel space to execute system calls, or when the *HW-Task* control achieves the sleep or dead states. Conversely, some of these functional units operate concurrently with the *HW-kernel* execution and in these cases, datapath-level signals must be used to ensure that required synchronous elements remain active for the desired clock pulses. Such signals, implement a local bidirectional interstate communication, that typically includes both data and control. Practical example is the Generic Bus handshake protocol.

In HW accelerated Feature detector design, we implement an asynchronous-synchronous datapath that is also concurrent to the kernel-level execution. Therefore, the bidirectional handshake protocol is based on data-valid and ready-to-receive signals, keeping the logical units active for the duration that such input signals remain valid. Figure 5.9 provides an abstract overview of both designs considering the *Classifilter* block. Details about the asynchronous-synchronous datapath can be consulted attached in Appendix F.

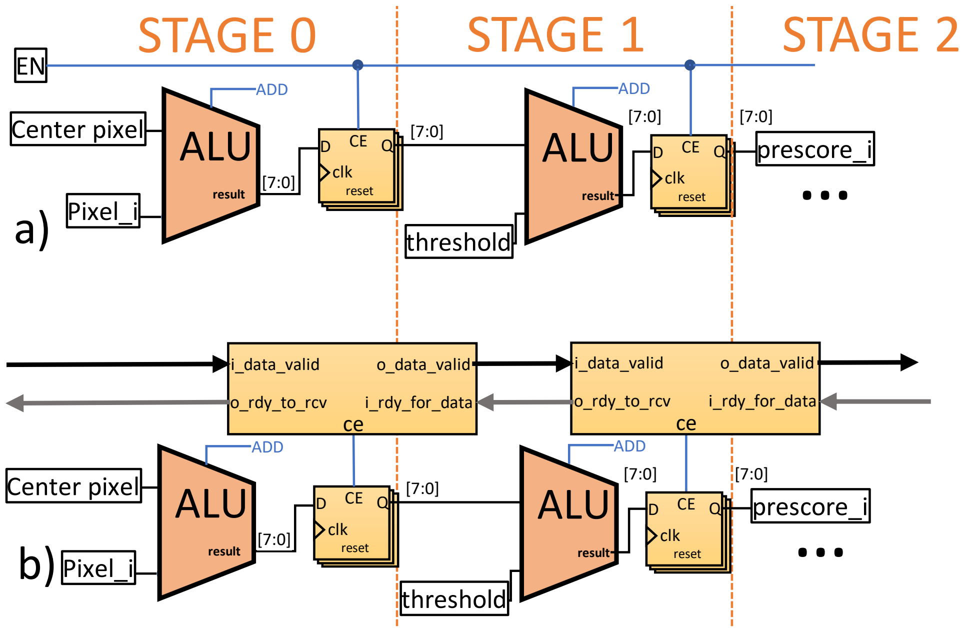


Figure 5.9: An abstract overview of synchronous (a) versus asynchronous (b) pipeline in the *Classifilter* block design.

In the synchronous example, data moves to the subsequent stage by enable-disable control. Such signal is usually handled by a control unit that implements pipeline stall when no data is available at input(s). On the other hand, in asynchronous design, a data-valid versus ready-for-data signals pair is feed across internal designs to move data across each pipeline stage, even when it is a single data in the complete pipeline chain.

To implement hardware-accelerated corner detection, the full dataset test algorithm that was fully deployed into an HDL design, dividing the implementation in two IPs: one for the detection of key points and another for the suppression of non-maximum. These two IPs were used in the datapath of the *HW-Task* and two initial versions where developed. The first one is based on the execution of the multi-threaded application exchanging data using the HAL-ASOS DDS. The second version is based on the single-threaded implementations that operates in a standalone mode reading the image file, processing pixels and writing corners in the output file. These two versions use the same datapath that will also be used in the subsequent *HW-Tasks* discussed in this chapter. Figure 5.10 shows a block diagram that describes such datapath.

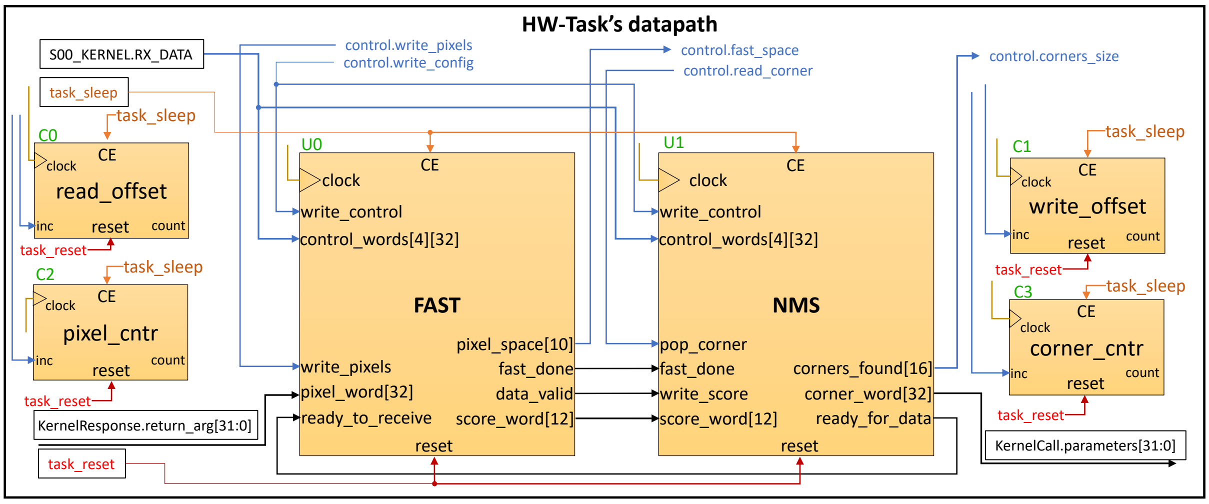


Figure 5.10: Hardware Accelerated Feature detection feature datapath.

To write pixels into the U0 detector (here called FAST), or read the corners found at the U1 suppressor (here called NMS) the *HW-Task* uses a set of packages from the HAL-ASOS framework. It will provide the necessary subset of HW-system calls to exchange data with the DDS subsystem in the target application, or manipulate files that were mapped into the application virtual space. As such, the *pixel_word* input will be connected to the output of the kernel HW-system call interface. In doing so, four pixels can be written in U0 at one clock cycle when the intermediate storage is the LRAM, or two cycles when the intermediate storage is the system memory.

Similarly, the *corner_word* will be connected to the input of the *HW-Kernel* system call interface, thus allowing the control unit in the *HW-Task* to write one corner per clock cycle in the LRAM, or two cycles in the system memory. Such connectivity is rewired by the *Extended Features* design level in the *Hardware Task* model, and is established when the corresponding procedures are set to active state by the control unit. Parametrizable counters C0 to C3 are used to increment memory offsets, account for the computing progress and issue messages about the total number of pixels processed, or the corresponding corners found. Each counter control signal is also rewired to the extended features level, and connected with appropriate HW-system call signals to handshake with results from the kernel-level execution.

5.5 Multi-threaded Synchronous design

In this preliminary design phase, the *HW-Task* was implemented following a processing contention strategy with the control unit describing synchronous behavior. This version is distinguished by control actions that write pixels in the datapath, after all operations to be performed by the Host system are positively confirmed during the HW system call execution. In doing so, the control unit for the *HW-Task* gets simplified, since the containment implicit in the model allows to significantly reduce the number of control states. The state diagram that results from this control strategy can be consulted in Figure 5.11.

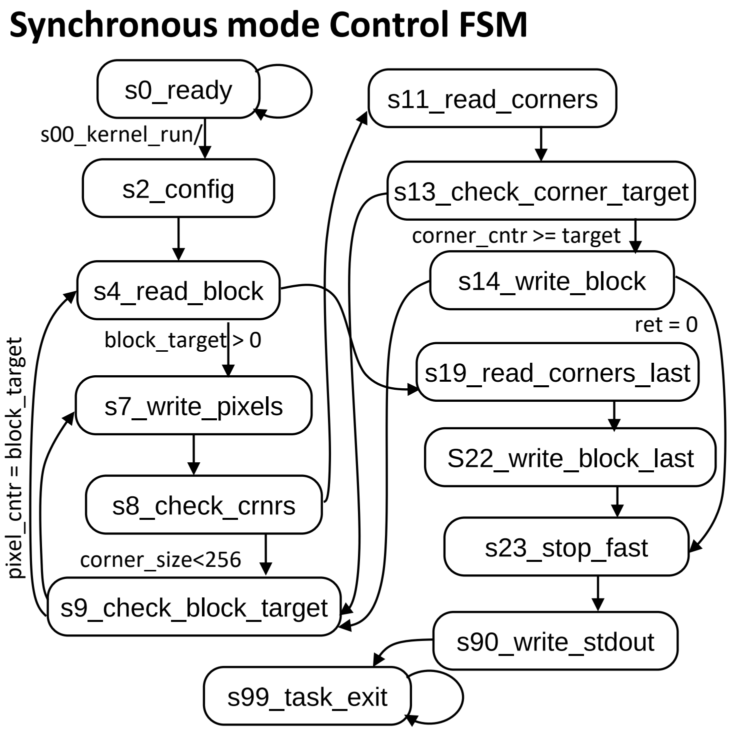


Figure 5.11: Simplified synchronous control unit for HW feature detection.

Upon receiving the run bit in its control register, the *HW-Kernel* wakes up from the sleep event and signals the control unit of the *HW-Task* to initiate processing, by asserting the *s00_kernel_run* flag. The first control action queries the target application for the existence of control data for the *HW-Task*. The retrieved data is used to configure the image format, the threshold value and optimize transfers for a specific block size. The block size is directly linked to the dimension of local storage and main memory resources, whose influence on the final performance will be studied during the tests carried out. For this purpose, a maximum size of 32 kB was established, which represents a 64 kB required storage to allow concurrent transfers in subsequent versions. The destination of such configuration is the *block_len* register in the

HW-Task datapath and three control registers of U0 and U1. These, in turn, are set using appropriate control signals (see Figure 5.10).

Once configured, in s2, the control unit requests the first block of data into the local storage zero offset, which when received sets the *block_target* parameter. Upon reaching s4, distinct *HW-Task* versions will use the system bus or the Local-BUS to transfer bursts of 256 words into the datapath. When the block target is set to the maximum value, i.e., 32 kB, the control unit will iterate between s7, s8 and s9 to implement 32 consecutive burst transfers. At the completion of each burst transfer, the control checks the NMS internal storage for corners. It moves to s11 to transfer the output corners to local storage using the 32 kB high memory region. In s13 it checks for at least 2 kB of stored data, and in s14 requests a transfer to the DDS topic. If it fails, from there the control proceeds to s9 and resumes previous cycle.

Consecutive blocks of pixels will be requested until a zero-block target is received. Control then moves to s19 to read the remaining corners and in s22 it commands the upload of any existing block size into the DDS topic. The processing is concluded by stopping the U0 and U1 IPs and writing a result message in the *stdout*. In doing so, it uses s23 and s99 respectively, and upon reaching s99 a task exit system call will put the accelerator into a dead state. Detailed block diagram of the synchronous control unit for the HW accelerated feature detection can be consulted in the attached Figure D.11. A complete description of this FSM can also be consulted in the attached Listings C.38 and C.39.

To exchange data with resources in the accelerator model, the extended features design implements at least four system calls. These are used to interact at application-level using the DDS services and to read from, or write to, the *HW-Task* datapath using intermediate storage resources, such as the LRAM or the SYSRAM. Figure 5.12 shows an excerpt of the extended features for the *HW-Task* that targets the LRAM. The resulting *HW-Task* can be used with any accelerator version, but is best used in the V4_00_E or V4_00_B for an appropriate use of the specific memory interfaces as well and each distinct clock domains. In the Zynq7000 SoC of the ZC702 platform, the V4_00_E accelerator design is not fully supported since only AXI4-Lite interface is available for the processing system to address the processing logic area. For this reason, the V4_00_B accelerator version was selected to target this *HW-Task* design version.

When an *HW-Task* connects to accelerator version V4_00_B, it generally targets the LRAM to exchange data in the application memory segment. In doing so, in line 474, it requests a block transfer from the DDS to the LRAM at zero offset and length expressed by the *block_len* register. At line 483, it writes consecutive four-pixel words to the datapath, that it receives from the procedure call in line 481. The

```
441  ------------------------------------------------------------------------------------------
442  EXTENDED_FEATURES: process(task_state,...
     ...
456  hal_asos_link_to_kernel(kernel_response,kernel_call);
457  case task_state is
     ...
473    when s4_read_block=>
474      transfer_data_from_dds(kernel_call,kernel_response,0, block_len);
475      pixels_target_d <= cast_return_to_transfer_len(kernel_response);
480   when s7_write_pixels=>
481      lram_read_word_burst(kernel_call,kernel_response,in_burst_q,index_read_q,RAM_DATA);
482      write_pixel_word <= cast_return_to_push_data(kernel_response);
483      pixels_word_d<=RAM_DATA;
     ...
486  when s11_read_crnrs=>
487      lram_write_word_burst(kernel_call,kernel_response,out_burst_q,
         index_write_q,corners_word_i);
488      inc_windex_i<=kernel_response.block_task;
       ...
492    when s14_write_block=>
493      transfer_data_to_dds(kernel_call, kernel_response, 32768, count_crnrs_bytes_q);
494      trfr_len_d <= cast_return_to_transfer_len(kernel_response);
     ...
524  end case;
525  end procedure EXTENEDED_FEATURES;
526  ------------------------------------------------------------------------------------------
```

Figure 5.12: Extended features for the multi-threaded HW-Task that targets LRAM resource when using the V4_00_B accelerator.

concurrent procedure on line 482 establishes connectivity for a one-way handshake between the read acknowledge (*RD_ACK*) of the LRAM and the write pixels signal of the IP U0

In the opposite direction, line 487 writes corners from the datapath to the LRAM using the address range above 32768. Here, the *out_burst_q* register specifies the transfer length and results from a balance between the number of existing corners, a parameter to limit the burst transfer, and the space in words available on this high region of the LRAM. In design terms no such limit exists, but to avoid starvation of concurrent datapath writing procedures, a 256-word limit was established. To control the write offset it relies on counter C1, which is incremented by the *inc_windex_i* signal at line 488. The same signal asserts the *pop_corners* input of U1 to provide the next corner at the output of this IP.

To complete the exchange, line 493 requests a block transfer of corners that already copied to the LRAM, to the corresponding topic in the DDS. The transfer length is triggered by the write offset of at least 2 kB and the value of counter C1 is used as transfer length, after being shifted two positions to the left. The descriptions for this design region of the *HW-Task* can be consulted in the attached Listings C.40 and C.41.

For comparison, the same HW-Task design was refactored to use the SYSRAM memory region, while burst transferring pixels to the datapath and writing corners in the opposite direction. For this, the states s4, s7, s11 and s14 where replaced with the corresponding procedures and the new *HW-Task* was attached to an accelerator version V4_00_F. Complete descriptions of this extended features design can also be found in the attached Listings C.42 and C.43.

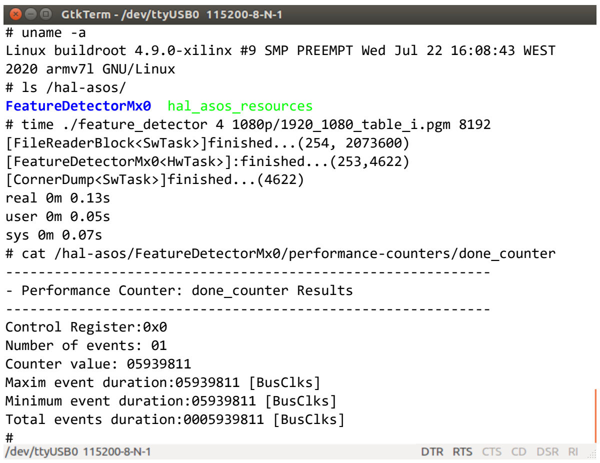
At this stage, the developed application operates in a multi-threaded format, and each of *HW-Tasks* can interact with the *FileRead* and *CornerDump Tasks* through the DDS subsystem. The application was updated to use each design according to a parameter passed as argument and distinct versions of the same application were implemented. Figure 5.13 shows changes required to reconfigure the application to use the accelerator that targets the LRAM.

```
  51 hal_asos::TaskConfig_t
  52  TFeatureDetectorMx = { "FeatureDetectorMx0",
  53 { "Corners",CORNER_LEN,1,1 },
  54 { "Imagelines",BLOCK_LEN,1,1 },
  55 { 1,1,1,1 }
     };
     ...
2223 void hal_asos_demo::feature_detector::
2224  test_fast_detector_std_mx_single(void){
2225   using namespace hal_asos;
2226 
2227   Task<SwTask, TFileReadBlock> T0;
2228   Task<HwTask, TFeatureDetectorMx,
     segment_len<(BLOCK_LEN<<1)>> T1;
2229   Task<SwTask, TCornerDump> T2;
2230   p_Detector_Task = (hal_asos::Task<> *)&T1;
2231 
2232   T0.start();
2233   T1.start();
2234   T2.start();
2235   T0.join();
2236   T1.join();
2237   T2.join();
2238 }
```

Figure 5.13: Excerpt of the multi-threaded application source that targets the LRAM resource using the V4_00_B accelerator.

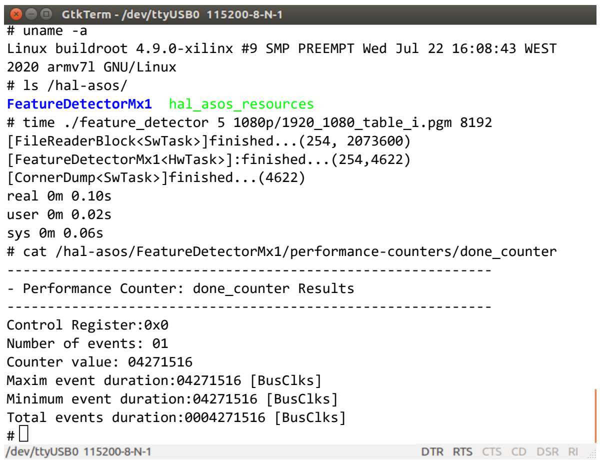
Lines 51 to 55 implement the *Taskconfig_t* structure that specify the desired task-name, the required DDS subscription used to receive pixel blocks, and the output *Topic* used to publish the detected corners. Line 2228 implements the *Task* class and by receiving the previous structure as template qualifier, while binding the processing to the accelerator mapped to the HAL-ASOS file system using the same task-name. At the same time, it also receives the *HwTask* and *segment_len* qualifiers that altogether, specialize the class for a *HW-Task* based computing resource, optimizing memory allocation to use twice the maximum block size.

Both accelerators were deployed to the Zynq ZC702 platform targeting a clock frequency of 50 MHz. A tag-name *FeatureDetectorMX0* was set to the design that targets the LRAM and similar tag, *Feature-DetectorMX1*, was set to the design that targets the SYSRAM. Figures 5.14a and 5.14b show the best performance results obtained when using the *table.pgm* image on each application version. The tests performed considered a block size of 8 kB and a threshold of 30, and the output resulted in a binary file containing 4,622 corners.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# uname -a
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# ls /hal-asos/
FeatureDetectorMx0  hal_asos_resources
# time ./feature_detector 4 1080p/1920_1080_table_i.pgm 8192
[FileReaderBlock<SwTask>]finished...(254, 2073600)
[FeatureDetectorMx0<HwTask>]:finished...(253,4622)
[CornerDump<SwTask>]finished...(4622)
real 0m 0.13s
user 0m 0.05s
sys 0m 0.07s
# cat /hal-asos/FeatureDetectorMx0/performance-counters/done_counter
-------------------------------------------------------------
- Performance Counter: done_counter Results
-------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 05939811
Maxim event duration:05939811 [BusClks]
Minimum event duration:05939811 [BusClks]
Total events duration:0005939811 [BusClks]
#
```

(a) Tests using V4_00_B accelerator.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# uname -a
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# ls /hal-asos/
FeatureDetectorMx1  hal_asos_resources
# time ./feature_detector 5 1080p/1920_1080_table_i.pgm 8192
[FileReaderBlock<SwTask>]finished...(254, 2073600)
[FeatureDetectorMx1<HwTask>]:finished...(254,4622)
[CornerDump<SwTask>]finished...(4622)
real 0m 0.10s
user 0m 0.02s
sys 0m 0.06s
# cat /hal-asos/FeatureDetectorMx1/performance-counters/done_counter
-------------------------------------------------------------
- Performance Counter: done_counter Results
-------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 04271516
Maxim event duration:04271516 [BusClks]
Minimum event duration:04271516 [BusClks]
Total events duration:0004271516 [BusClks]
#
```

(b) Tests using V4_00_F accelerator.

Figure 5.14: Performances for the multi-threaded HW accelerated feature detention using V4_00_B and V4_00_F accelerators at 50 MHz and 8 kB and 2 kB block sizes.

On the left-side image, the application using the *FeatureDetectorMx0* accelerator achieved the best execution time of 130 milliseconds. The *stdout* messages show that the block size has been fixed at 8,192 bytes, giving rise to 254 image fragments which represent a total of 2,073,600 pixels. Analyzing the done performance counter, it can be seen that the *HW-Task* used 5,939,811 clock cycles to process all pixels, and this value can be translated to a 119-millisecond execution time (i.e., 5,939,811/50MHz) or an average processing rate of 2.87 clocks per pixel.

On the right-side image, the application using the *FeatureDetectorMX1* accelerator achieved the best execution time of 100 milliseconds for the same input parameters. The done performance counter registers 4,271,516 clocks to process the same image, which translates to an execution time of 82.5 milliseconds, or a 2.06 clocks per pixel rate. As expected, the total execution time of both applications differs from the times of each *HW-Task* as a result of allocation and consequently freeing the software required resources. When comparing these results with the software full dataset execution, the hardware accelerated versions are 24 times (i.e., 3.16/0.13) and 31 times (i.e., 3.16/0.1) faster.

To compare the HW accelerated versions with the *OpenCV* and the *Ed.Rosten-C* applications Figure 5.15 plots the execution time when using the four input files in the test dataset. The application based in the FeatureDetectionMX1 HW-task outperforms both of the software accelerated versions. On the other hand, the FeatureDetectionMX0 outperforms the *OpenCV*, but it exceeds the execution time of the *Ed.Rosten-C* in 10 milliseconds in the 4.6k corner mark. With the increase in the number of corners per image, the accelerator performance suffers a marginal deterioration when compared to software implementations. The processing performed in the detection and suppression of non-maximum is the same but a greater number of corners needs to be written into the output file.

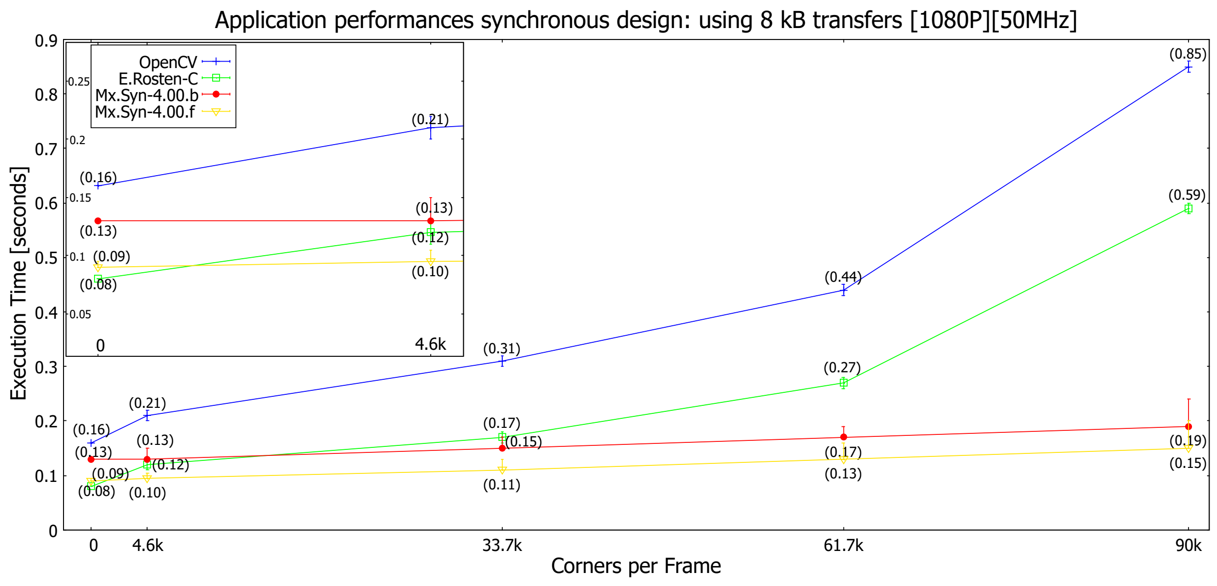
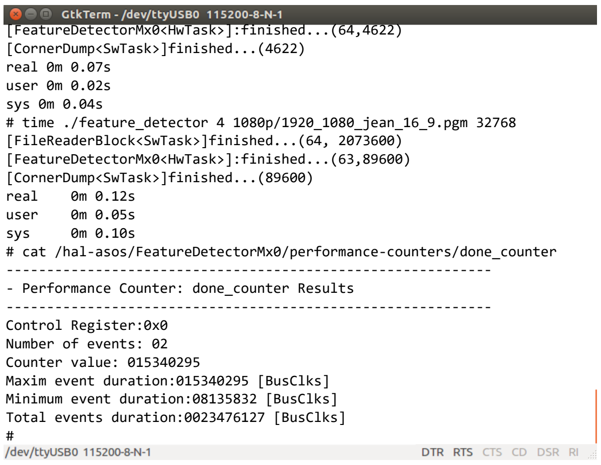


Figure 5.15: Application performances between software and synchronous HW accelerated feature detection.

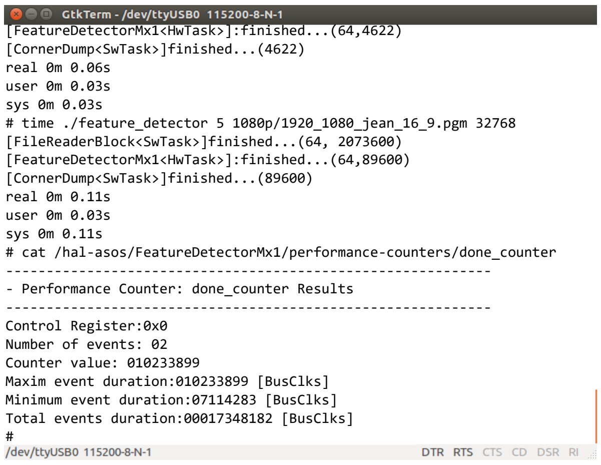
Although, both software-only applications inherently require more memory than the 8 kB and 2 kB block allocations can achieve, such block sizes can be considered a low value to fit in the performance provided by the CPU architecture on the target platform. As such, it can introduce some degree of computation overhead with rescheduling operations that fundamentally result from a low CPU usage in the number of transfers performed. An increased variance is observed in the files that provide greater number of corners in the output file. By varying the frequency applied to the design and the size of the transfer block at the input, the performance results show significant improvements. Figure 5.16a shows the best execution results in the tests performed using images of 4,622 and 89,600 corners. Each of the accelerators was stimulated with a clock frequency of 142.8 MHz and a block size of 32 kB at the input.

Tests performed to the *FeatureDetectorMx0* application showed execution times of 70 milliseconds for the


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
[FeatureDetectorMx0<HwTask>]:finished...(64,4622)
[CornerDump<SwTask>]finished...(4622)
real 0m 0.07s
user 0m 0.02s
sys 0m 0.04s
# time ./feature_detector 4 1080p/1920_1080_jean_16_9.pgm 32768
[FileReaderBlock<SwTask>]finished...(64, 2073600)
[FeatureDetectorMx0<HwTask>]:finished...(63,89600)
[CornerDump<SwTask>]finished...(89600)
real    0m 0.12s
user    0m 0.05s
sys     0m 0.10s
# cat /hal-asos/FeatureDetectorMx0/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 015340295
Maxim event duration:015340295 [BusClks]
Minimum event duration:08135832 [BusClks]
Total events duration:0023476127 [BusClks]
#
```

(a) Tests using V4_00_B accelerator.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
[FeatureDetectorMx1<HwTask>]:finished...(64,4622)
[CornerDump<SwTask>]finished...(4622)
real 0m 0.06s
user 0m 0.03s
sys 0m 0.03s
# time ./feature_detector 5 1080p/1920_1080_jean_16_9.pgm 32768
[FileReaderBlock<SwTask>]finished...(64, 2073600)
[FeatureDetectorMx1<HwTask>]:finished...(64,89600)
[CornerDump<SwTask>]finished...(89600)
real 0m 0.11s
user 0m 0.03s
sys 0m 0.11s
# cat /hal-asos/FeatureDetectorMx1/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 010233899
Maxim event duration:010233899 [BusClks]
Minimum event duration:07114283 [BusClks]
Total events duration:00017348182 [BusClks]
#
```

(b) Tests using V4_00_F accelerator.

Figure 5.16: Performances for the multi-threaded HW accelerated feature detention using accelerator versions V4_00_B/F at 142 MHz and 32 kB and 2 kB block sizes.

4,622 corners image and 120 milliseconds for the 89,600 corners image. Compared to the results of the same design in the previous conditions, the performance improves 1.84 times or 46.1 % in the file with the fewest corners and 1.58 times or 36.8 % for the largest number of corners in the output files. Regarding the results of *OpenCV* (210 and 850 milliseconds) and *Ed.Rosten-C* (120 and 590 milliseconds) this new mark outperforms both by 2 and 4 times in the lower number of corners, and 5 and 7 times for the image with more corners. The done performance counter shows minimum and maximum numbers of clock per execution of 8,135,832 and 15,340,295 respectively. Those can be translated to *HW-Task* execution times of 57.3 milliseconds and 108 milliseconds.

Similar results were observed for the FeatureDetectorMX1-based application, in the figure on the right, which surpasses the previous accelerator by 10 milliseconds and therefore, the two applications based on software accelerated algorithms. For the lower number of corners, it achieved an execution time of 60 milliseconds while 110 milliseconds was measured for the image with more corners. The results in the done performance counter, show execution times of 50.1 and 70.07 milliseconds for the HW-Task. In summary, we can say that with increasing design frequency the difference in performance of both HW accelerated designs is purely marginal (i.e., about 10 milliseconds). The LRAM transfers have become about 3 times faster and with that, it compensates for the low performance interface provided by the target platform.

5.6 Stand-alone Synchronous Single-task

To mitigate the influence of the scheduling operations, the software application was refactored to a single *HW-Task* implementing a stand-alone design. This application version has a similar behavior to the single-threaded version used in the previous software tests. To simplify the design of the *HW-Task* the input file was previously parsed to extract the image parameters and provide a configuration structure for the datapath IPs. Then, together with the output file, these two files were pooled to the Task class to be handled by the accelerator design. Figure 5.17 shows an excerpt of the necessary changes to the software application to implement the standalone synchronous feature detection.

```
 136  hal_asos::TaskConfig_t
 137  TFastDetectorSA = { "FastDetectorSA1",
 138  { "",0 },
 139  { "",0 },
 140  { 4,1,1,1 }
 141  };
      ...
2409  void hal_asos_demo::feature_detector::
2410    test_fast_detector_std_alone_single_sysram(void) {
2411      using namespace hal_asos;
      ...
2419     std::shared_ptr<StreamData> Conf;
2420      Task<HwTask, TFastDetectorSA, segment_len<(BLOCK_LEN << 1)>> T1;
2421     CFstream<std::ifstream> Input_file(scene_img.c_str());
2422     Input_file.set_flags(std::ios::in | std::ifstream::binary);
2423     CFstream<std::ofstream> Output_file("Corners_std_single_sysrsam.txt");
2424     Output_file.set_flags(std::ios::out |std::ios::trunc|std::ios::binary);
2425
2426     Conf = std::make_shared<StreamData>(16);
2427     detector::config_words* p_config = (detector::config_words*) Conf.get();
      ...
2443     Input_file.get_line(header1);
      ...
2451     ss.str(header1);
2452     ss >> p_config->image_width >> p_config->image_height;
2453     ss.clear();
      ...
2463     p_config->threshould = threshould;
2464     p_config->block_len = block_size;
2465
2466     T1.submit_to_pool(Input_file);
2467     T1.submit_to_pool(Output_file);
2468
2469     T1.submit_data(Conf);
2470
2471     T1.start();
2472     T1.join();
2473  }
```

Figure 5.17: Excerpt from the source to implement standalone application using synchronous design and the V4_00_F accelerator.

In that Figure, lines 136 to 141 show the Task configuration structure that binds the class to the accelerator. Such configuration is used by the application that targets the SYSRAM memory region in line 2409.

The *Task* class is declared in line 2420 and the following four lines open the two files using the necessary control flags. Lines 2443 to 2453 parse the input file to fill the parameters in the *config_words* structure. The application proceeds by submitting these three memory objects to the T1 class and issuing the *start* member to initiate processing at the accelerator side. The join member in line 2472 will put the main thread on hold until the *HW-Task* executes the yield system call. A similar source list was implemented for the application version that uses the LRAM-based design. A distinction is made by providing a task configuration structure that binds the class with the accelerator that matches the task name. For completeness, the full source of this application version that fit both designs, can be seen attached in Listing C.37.

The previous *HW-Task* designs where refactored to implement a standalone design. Thus, the design demands for use of hardware file descriptors from the HAL-ASOS framework, which provide means for the *HW-Task* to exchange data with the input and output files. Fundamentally, the control unit maintains its sequence of states if the input and output files are previously prepared by the software side of the application. Therefore, the extended features region has been updated with the proper procedures for handling the two files. The required changes can be seen in Figure 5.18. This excerpt shows the procedures that target the SYSRAM memory region, and to obtain the maximum transfer performance, the correspondent *HW-Task* was attached to the accelerator V4_00_F version.

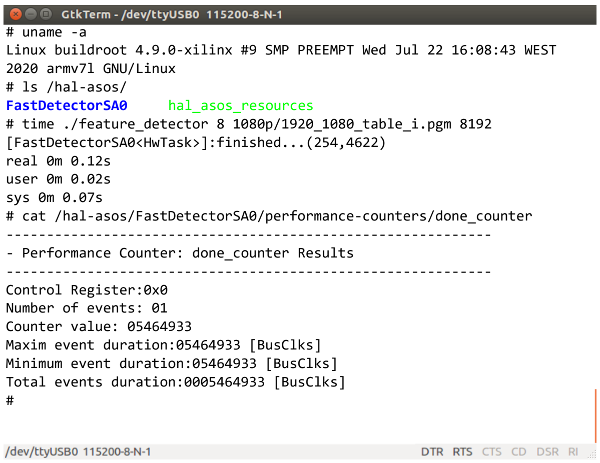
In lines 350 and 352, the procedure calls exchange information with the *Task* class to update the local descriptors *i_file_q* and *o_file_q* respectively. Lines 369 and 380 exchange data with the input and output files using the main system memory. In these, the *_sysram* suffix distinguishes the memory resource in use. The corresponding data is copied to and from its system memory region in lines 372 and 376. In these states (i.e., s11 and s14), the concurrent *cast* procedure handshakes the *write_pixel_word* and *pop_crnrs_i* with the acknowledge signals of the *M01_System* interface. As such, they are asserted at a two clock per word rate which is two times slower than the LRAM resource, but also two times faster than the datapath pixel rate. The complete HDL description for the extended features region can be consulted in the attached Listings C.44 and C.45.

Tests performed demonstrate an overall application performance improvement of 10 milliseconds for the same conditions as the previous version, i.e., 50MHz clock frequency, and 8kB and 2kB for block sizes. Figure 5.19 shows the results for applications based on these two *HW-Task* designs. On the right-side image, the application based on the accelerator V4_00_B is using tag-name *FastDetectorSA0* and completed its processing for the input file with 4,622 corners in 120 milliseconds. On the left-side image,

```
---------------------------------------------------------------------------------------------
EXTENDED_FEATURES: process(task_state, ifile_q, ofile_q,...
...
hal_asos_link_to_kernel(kernel_response,kernel_call);
case task_state is
 when s1_query_ifile=>
   pooled_fstream_query(kernel_call,kernel_response,ifile_q, ifile_d);
 when s2_query_ofile=>
   pooled_fstream_query(kernel_call,kernel_response,ofile_q, ofile_d);
 when s3_query_conf=>
   transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
...
 when s6_read_file=>
   pooled_fstream_read_sysram(kernel_call,kernel_response, ifile_q, block_len,0);
   pixels_target_d <= cast_return_to_transfer_len(kernel_response);
 when s11_push_pixels=>
   safe_safe_read_sysram_word32_burst(kernel_call,kernel_response, pixels_word_d,
fifo_in_burst_q,index_read_q);
   write_pixel_word <= cast_return_to_push_data(kernel_response);
   inc_rindex_i<=cast_return_to_push_data(kernel_response);
 when s14_write_block=>
   safe_write_sysram_word32_burst(kernel_call,kernel_response,corners_word_i,
fifo_out_burst_q,index_write_q);
   pop_crnrs_i <= cast_return_to_pop_data(kernel_response);
   inc_windex_i <= cast_return_to_pop_data(kernel_response);
 when s18_fstream_write=>
   pooled_fstream_write_sysram(kernel_call, kernel_response,ofile_q,
count_crnr_bytes_q, 32768);
          ofile_len_d <= cast_return_to_transfer_len(kernel_response);
...
end case;
end procedure EXTENEDED_FEATURES;
---------------------------------------------------------------------------------------------
```

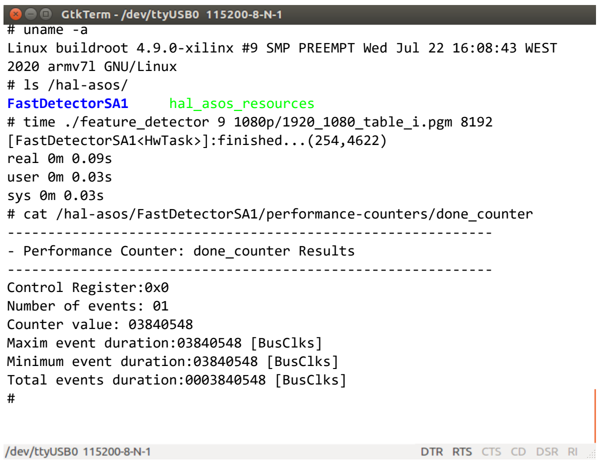
Figure 5.18: Extended features for the standalone HW-Task that targets SYSRAM resource when using the V4_00_F accelerator.

the application based on accelerator V4_00_F is using the tag-name *FastDetectorSA1* and completed its processing in 90 milliseconds under the same input conditions.



```
GtkTerm - /dev/ttyUSB0  115200-8-N-1
# uname -a
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# ls /hal-asos/
FastDetectorSA0      hal_asos_resources
# time ./feature_detector 8 1080p/1920_1080_table_i.pgm 8192
[FastDetectorSA0<HwTask>]:finished...(254,4622)
real 0m 0.12s
user 0m 0.02s
sys 0m 0.07s
# cat /hal-asos/FastDetectorSA0/performance-counters/done_counter
--------------------------------------------------------------
- Performance Counter: done_counter Results
--------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 05464933
Maxim event duration:05464933 [BusClks]
Minimum event duration:05464933 [BusClks]
Total events duration:0005464933 [BusClks]
#

/dev/ttyUSB0 115200-8-N-1                DTR  RTS  CTS  CD  DSR  RI
```

(a) Tests using V4_00_B accelerator.



```
GtkTerm - /dev/ttyUSB0  115200-8-N-1
# uname -a
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# ls /hal-asos/
FastDetectorSA1      hal_asos_resources
# time ./feature_detector 9 1080p/1920_1080_table_i.pgm 8192
[FastDetectorSA1<HwTask>]:finished...(254,4622)
real 0m 0.09s
user 0m 0.03s
sys 0m 0.03s
# cat /hal-asos/FastDetectorSA1/performance-counters/done_counter
--------------------------------------------------------------
- Performance Counter: done_counter Results
--------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 03840548
Maxim event duration:03840548 [BusClks]
Minimum event duration:03840548 [BusClks]
Total events duration:0003840548 [BusClks]
#

/dev/ttyUSB0 115200-8-N-1                DTR  RTS  CTS  CD  DSR  RI
```

(b) Tests using V4_00_F accelerator.

Figure 5.19: Performances for the stand-alone synchronous control using V4_00_B/F accelerators at 50 MHz and 8 kB and 2 kB block sizes.

Regarding the performance of the *HW-Tasks*, the *FeatureDetectorSA0* completes its processing using

5,464,933 clock cycles, which represents an execution time of 109.3 milliseconds. On average, the value of the clock per pixel improves to 2.64 when compared to the previous 2.87. The FeatureDetectorSA1 completes its processing with 69.61 milliseconds using 3,480,548 clock cycles. In this case, the average value of the clock per pixel rate improves to 1.68, when compared to the previous 2.06.

Figure 5.20 shows the execution results of the four developed HW accelerated applications, when the input file varies the number of detected corners from 0 to 90k while the input block size is maintained at 8 kB. The multi-threaded versions (i.e., Mx.) are less regular than standalone due to scheduling delays. This approach aggravates results since synchronization between software threads consumes extra time and lacks determinism. At this point, the standalone single *Task* applications (i.e., Sin.) will benefit from kernel-level operations while executing the HAL-ASOS file system code. In these, execution threads are usually driven by hardware interrupts that can have a higher priority than most threads in the system. Lastly, a performance degradation is observed by increasing the number of detected corners, but when compared to pure software executions such degradation is almost imperceptible.

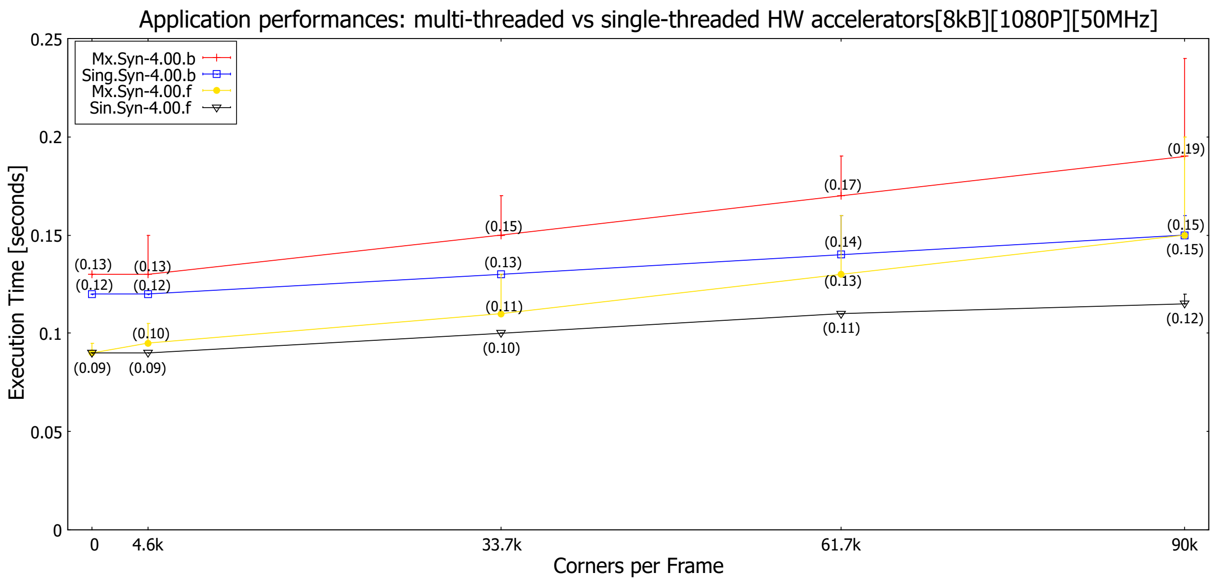


Figure 5.20: Performance comparison of the synchronous design HW-accelerated applications using 8 kB block and 50 MHz clock frequency.

In the same figure, it can be seen that the applications based in the V4_00_B accelerator versions, require more time due to the low bandwidth interface provided by the platform. In terms of performance achievement, the V4_00_F accelerated designs showed that an improvement would be possible by increasing the block sizes. In functional terms, the HAL-ASOS framework establishes a limit for this memory region at 64 k-words, i.e., 256 kB. For this, the *HW-Task* would have to change the control metrics to be based on word rate transfers, where 16-bit are used to specify the required transfer lengths. Appropriate user procedures are available to implement such word-rated transfers and are generally suffixed by *_word32*. Such a memory region will put greater strain on resources at the HAL-ASOS file system level, with a permanent allocation of 64 pages of 8 kB from the kernel space region. On the other hand, such degree of transfers length could congest the main system memory. At this point, there are other possibilities to explore and for that reason this option was not considered beyond the 64 kB, which are intended to be used as 2 blocks of 32 kB for concurrent transfers.

Last but not least, choosing the fastest accelerator design, we evaluated the influence exerted by the input block size while using four possible design frequencies and considering the extreme scenario of 90k corners. Figure 5.21 shows the execution time results of the various tests performed. From this analysis, we can see that increasing block size is more beneficial to the application performance using design frequencies above 50 MHz and has little influence beyond 16 kB. Performance differences bounce around 10 milliseconds between the three highest frequencies. The accelerator gets 20 milliseconds faster at 50 MHz when the block size ranges from 8 kB to 32 kB, and 10 milliseconds faster at 142 MHz for the same range. Lastly, and as purely qualitative metric, the application using the V4_00_F accelerator with a design frequency of 50 MHz and 32 kB input block, or using 142 MHz and a 16 kB input block, is 36 times and 45 times faster than the equivalent software running solely on the ARM A9 dual core of the ZC702 platform.

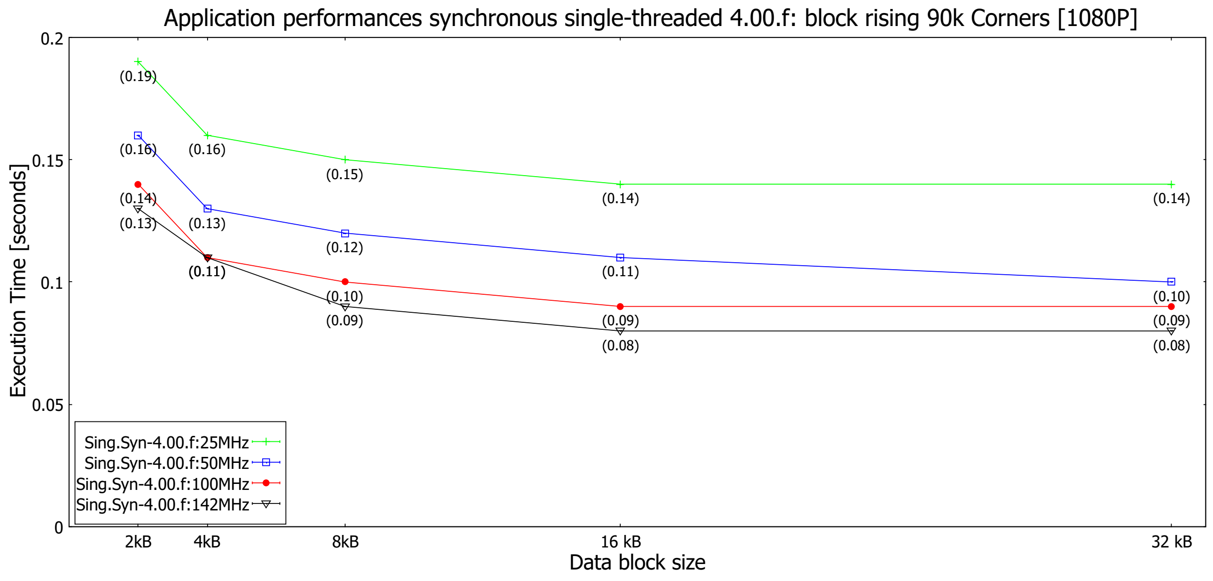


Figure 5.21: Performance of the standalone synchronous application using the V4_00_F accelerator and 90,000 corners and varying the input block size and design frequency.

At this point, it is considered that the design reaches a performance stagnation and that improvements will have to be implemented based on other assumptions that are not memory and clock frequency increases. In the next section, we will cover design changes to the *HW-Task* that outline an asynchronous control unit behavior using the same accelerators versions.

5.7 Stand-alone Asynchronous Dual-task

To improve the performance of the *HW-Task*, the control unit design was refactored to implement an asynchronous behavior strategy. Consequently, the extended features level has been updated with the corresponding procedures, which can be found in the user-level packages provided by the HAL-ASOS framework. To lower the level of complexity, the previous design was divided into distinct *HW-Tasks* by splitting the control unit in (1) reading blocks of pixels from the input file and subsequent writing to the datapath, and (2) handling the corners found and the corresponding writing in the output file. To conform with this new control strategy, the datapath was also split between the two *HW-Task* designs. In addition, this change also aims to improve throughput in data transfers, exploring the dual-core architecture at kernel-level together with the duplicated transfer interfaces in each of the *HW-Kernels*. Figure 5.22 shows this new dual task design using a task-level block diagram.

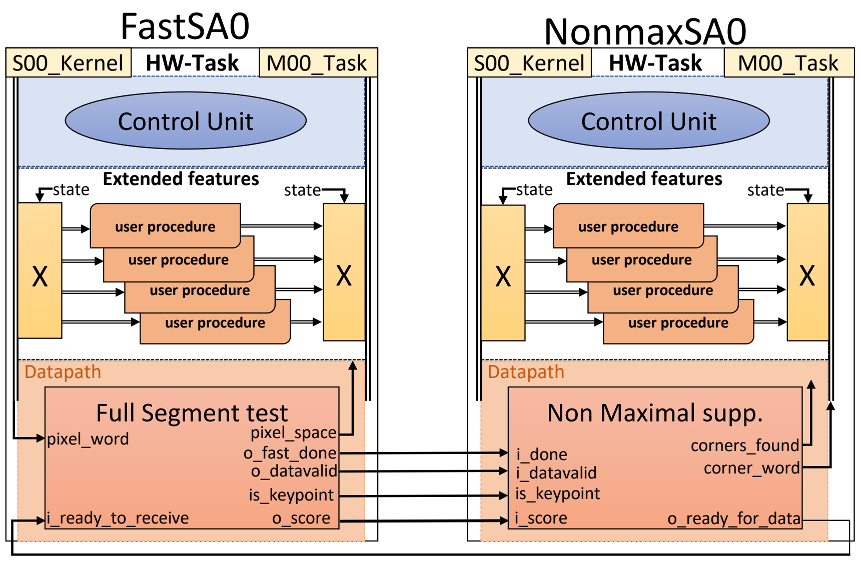


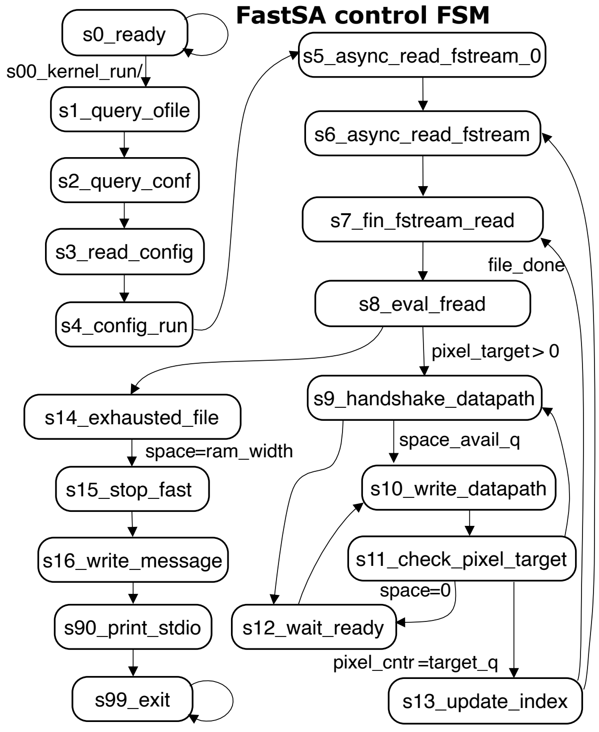
Figure 5.22: Dual task hardware accelerated feature detection block diagram.

On the left side is shown the *HW-Task* that reads the input file in parameterizable block sizes and therefore implements the hardware of the full dataset segment test. As in the previous design, for each pixel submitted to the feature detection, there is an output score that refers the intensity of the central pixel to the intensity of the sixteen pixels on the periphery of the Bresenham circle. This 16-bit value remains in the score output while the datapath handshakes with the non-maximum suppression in the *HW-Task* on the right. The handshake is performed using the *data_valid* and *ready_to_receive* signals, which in this design were promoted to top-level signals for each *HW-Task*, together with the *keypoint* and *fast_done* flags.

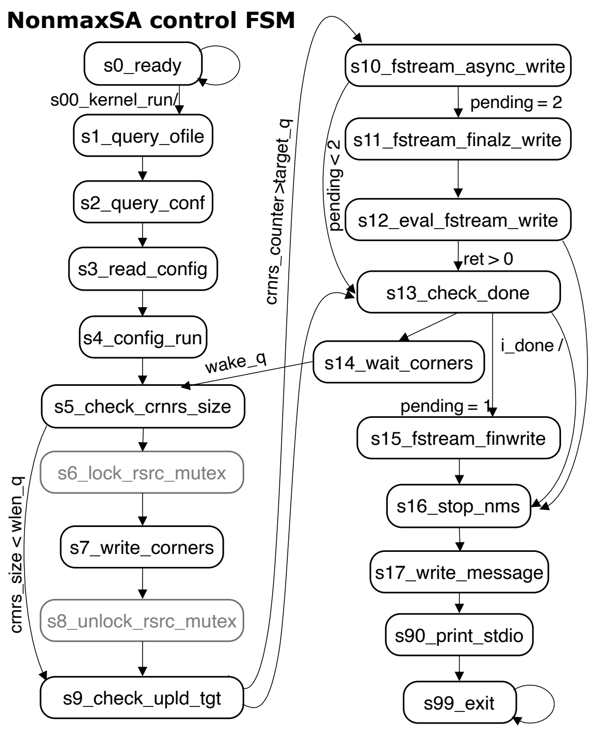
The control unit implements a purely asynchronous behavior semantics, that aims to interact with the host system at the same time it interacts with the datapath. In this control strategy, the datapath must follow a purely data flow-oriented design, capable of dealing with the invariability of data at the input or space availability at the output. The datapath design discussed in the previous section already fulfills this requirement, since it was implemented following an asynchronous-synchronous design, making it compatible with each of the control strategies used.

In general terms, this asynchronous control strategy anticipates transfers between memory regions to raise the CPU-usage on the host-side. For greater efficiency, the control unit must avoid the exclusivity on the shared memory regions, thus preventing any blocking caused by the mutual exclusion resources. For that purpose, it implements a memory layout of multiple blocks of memory, that are accessed using a message-based handshake protocol. Furthermore, a lazy confirmation of message reception is followed, as control postpones the confirmation until the precise moment it reaches an unconfirmed memory block. At that time, if the host system did not complete the corresponding transfer, the control unit will inevitably enter a blocking stage, but ensuring that the CPU was already instructed to proceed into the next block of pixels. On the host system side, such laziness approach promotes continuous transfer tasks because there is always a pending request at the time the CPU acknowledges the current transfer. By doing this, it also reduces scheduling workload by minimizing the number of interrupt events thrown by each accelerator. Figure 5.23 shows the refactored state diagrams for the control units of each *HW-Task* according to this asynchronous design strategy.

Figure 5.23a shows the sequence diagram for the control unit of the *HW-Task FastSA*. In this diagram, an initialization phase is implemented using states s0 and s4, where the control updates the datapath settings by storing the input file descriptor, image format and input block size. In s5 it requests the transfer of the first block and initiates a cyclic behavior, where it requests a new block before confirming that the previous one is ready. Thus, s6 and s7 must ensure a response message containing the number of bytes transferred, to establish the new pixel target for the current cycle. In s8, the control checks this value before starting the copy of the pixel block to its datapath.



(a) Asynchronous control unit for the HW-Task FastSA.



(b) Asynchronous control unit for the HW-Task NonmaxSA.

Figure 5.23: Asynchronous control for hardware accelerated feature detection.

Once in s9, it handshakes with the space available signal on the datapath, to assert the burst transfer register and proceeds to s10 for the effective copy. If otherwise this signal is low, it proceeds to s12 and enters a sleep state. The same signal is forwarded to the *HW-Kernel* to wake the accelerator when available space allows for a new burst transfer. In s11, the control checks for the pixel counter value (C2 in Figure 5.10) to handle the copy progress. If not completed, it will fall back to s9 to assert a new burst transfer or it can be forced into a sleep state, s12, if the datapath space is less than the optimized length of 256 words. At target completion, it proceeds to s13 to prepare the new block address and begin a new cycle. A *file_done* flag can be established by an incomplete block size in the pixel target. It will break the cycle in s13 for the last confirmation using s7 and s8. The control unit will move to s14 to ensure completion of the datapath before proceeding to s15 and stopping U0. It will conclude by preparing the results message in s16 and writing to the *stdout* in s90. In s99 the control will issue a task exit system call that puts the accelerator into a dead state. Excerpt containing the HDL descriptions of this state logic can be found attached in Listing C.46 and Listing C.47.

The control unit for the *NonmaxSA*, Figure 5.23b), assumes the complementary behavior, by writing corners from the datapath to the physical memory and demanding block transfers to the output file. Similar s0 to s4 states are used and the control unit will wait for the first processing results in s14, after having evaluated: the absence of corners in s5; a copied length less than the block size in s9; and a false *fast_done* flag in s13. When the number of corners reaches 256 words, the control wakes up in s14 and proceeds to s5 for a handshake with the datapath. At s5, it asserts the burst length register before proceeding to s6 and write corners in the physical memory. States s6 and s8 are only used when the storage resource is LRAM, and are forced by a design requirement to enable write operations on the channel that holds the exclusivity in the *LMutex*. Such condition can compromise performance on the host system by blocking the mutual exclusion mechanism and forcing the software to reschedule the execution.

Once in s7, the control transfers 256 corners to the storage resource in the current block of data, and in s9, it evaluates the copied length using C3 to see if it has achieved the pre-set block target (here 2 kB). When the block is not completed, it proceeds to s13 to check if the *FastSA* is still active and resumes the wait state in s14. If otherwise a block target is ready, it proceeds to s10 to transfer the current block to the output file. When there are at least two blocks pending for confirmation, it switches to s11 and evaluates the transfers results in s12. If no error is received, it resumes the wait state before initiating a new transfer block, or otherwise it aborts processing by moving to s16.

Upon completing a block, in s13, an active *fast_done* input will break this cyclic behavior and force control to finalize the transfer of any existing corners in s15. The same flag can also awake the control unit in s13 to complete current block and finalize the transfer. It will ultimately conclude by stopping U1 in s16, writing the results message in the *stdout* using s17 and s90, and issuing a task exit system call.

To complete each *HW-Task* design, the extended features level was also refactored and divided according each of to the previous control FSMs in each design. Figure 5.24 shows a VHDL excerpt of the extended features for the *FastSA*. In this excerpt, the design targets the system memory to asynchronously transfer pixel blocks from the input file and in this case, it should be connected with accelerator V4_00_F for an optimized performance. To distinguish this design from the equivalent *HW-Task* that uses the LRAM, the accelerator's tag-name was prefixed with the letter 'M', in association with the *M00_System* interface or the role of a bus master. This tag-name convention was also applied to the *NonmaxSA* design.

In line 457, the control requests the first block of pixels for the zero offset of the SYSRAM region at the main system memory. In line 459, the same procedure repeats the operation, using the offset specified

```
423 --------------------------------------------------------------------------------------------
424 EXTENDED_FEATURES: process(task_state,...
    ...
437 hal_asos_link_to_kernel(kernel_response,kernel_call);
438 case task_state is
    ...
456 when s5_async_read_fstream_0=>
457     async_pooled_fstream_read_sysram(kernel_call,kernel_response, ifile_q,blen_param_q,0);
458 when s6_async_fstream_read=>
459     async_pooled_fstream_read_sysram(kernel_call,kernel_response,
                                         ifile_q,blen_param_q,w_address_offset_q);
460 when s7_fin_fstream_read=>
461     async_finalize_pooled_fstream_read_sysram(kernel_call,kernel_response);
462     pixels_target_d <= cast_return_to_transfer_len(kernel_response);
463 when s10_write_datapath=>
464     unsafe_safe_read_sysram_word32_burst(kernel_call,kernel_response,
                pixels_word_d, burst_target_q,index_read_q);
465     write_pixel_word_i <= cast_return_to_push_data(kernel_response);
466     inc_rindex_i<=cast_return_to_push_data(kernel_response);
467 when s12_wait_space=>
468      wait_signal_event(kernel_call,kernel_response,space_available_q,is_event_d,0);
    ...
479 end case;
480 end if;
481 end procedure EXTENEDED_FEATURES;
482 --------------------------------------------------------------------------------------------
```

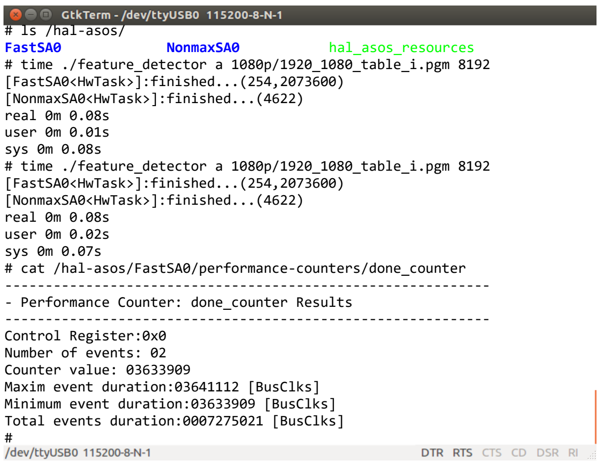
Figure 5.24: Extended Features design-level for the HW-Task *MFastSA* that targets the *SYSRAM* resource and uses de V4_00_F accelerator version

by the *w_address_offset_q* register, which may have a value between 2 kB and 32 kB depending on configuration received. At line 461, the procedure finalizes the transfer of the current block, by storing the return value in the pixel target register. Multiple 256-word burst transfers will then copy the received pixels to the datapath by repeating the procedure in line 464. Finally, in line 468, the wait system call will put the control unit and the *Kernel Core* into a sleep state, leaving strategic functional units in the Datapath with the clock inputs active. The wake condition will be asserted when the datapath space reaches 256 words or 1024 kB.

The remainder of procedures that complete this design-level are related to exchanging control information and printing processing results. The complete source can be consulted attached in the attached Listing C.48. To target the alternative LRAM resource, appropriate procedures were implemented and the same control logic was used. The details of such design can also be consulted attached in Listing C.49. Similar implementation was used to describe the extended features design of the NonmaxSA, that target the SYSLRAM or the LRAM storage resources. We will not introduce those designs to limit the descriptions in this chapter, but they can still be consulted in the attached Listings C.50 to C.53.

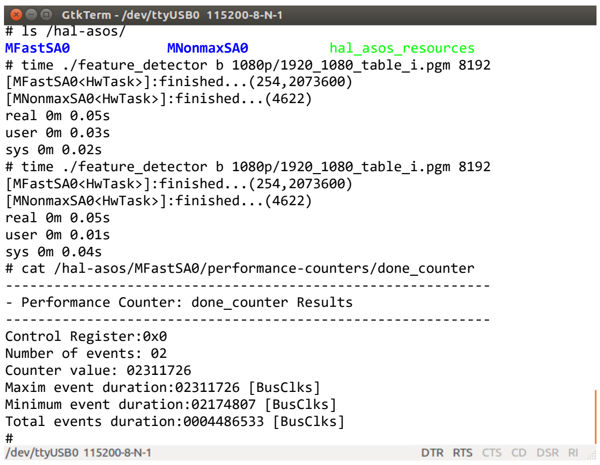
In line with the tests performed in the previous section, each application was tested separately to distinguish the storage resource used in the data exchange. The accelerator that reads the input file requires a memory size of 64 kB divisible to a maximum of two 32 kB blocks. The accelerator that writes the output results uses 4 kB of memory, split into two fixed-size 2 kB blocks. The tests performed vary the block size at the input and the best execution times obtained for the intermediate size of 8 kB and 50 MHz clock frequency can be seen in Figure 5.25.

On the right side, Figure 5.25a show the results of two consecutive tests using the file with 4,622 corners on the accelerator versions V4_00_B. The two consecutive testes obtained a total execution time of 80 milliseconds. When compared with the synchronous stand-alone design (i.e., 120 milliseconds, Figure 5.19a) the asynchronous design is 33.3% milliseconds faster. In the same figure, it can also be seen the results of the performance counter that registers the execution time of the *HW-Task*. Such counter registered 3,633,909 clock cycles and 3,641,112 clock cycles for minimum and maximum durations, respectively. These numbers can be translated to 72.68 milliseconds and 72.82 milliseconds execution times respectively and a 1.75 ratio of clocks per pixel.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
FastSA0                 NonmaxSA0               hal_asos_resources
# time ./feature_detector a 1080p/1920_1080_table_i.pgm 8192
[FastSA0<HwTask>]:finished...(254,2073600)
[NonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.08s
user 0m 0.01s
sys 0m 0.08s
# time ./feature_detector a 1080p/1920_1080_table_i.pgm 8192
[FastSA0<HwTask>]:finished...(254,2073600)
[NonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.08s
user 0m 0.02s
sys 0m 0.07s
# cat /hal-asos/FastSA0/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 03633909
Maxim event duration:03641112 [BusClks]
Minimum event duration:03633909 [BusClks]
Total events duration:0007275021 [BusClks]
#
```

(a) Tests using V4_00_B accelerators at 50 MHz and 8 kB.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

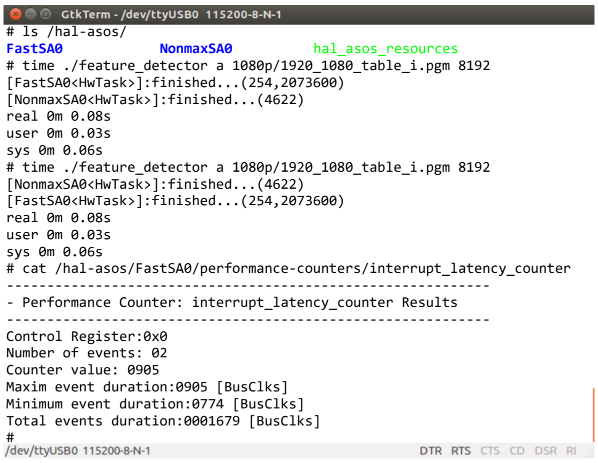
```
# ls /hal-asos/
MFastSA0                MNonmaxSA0              hal_asos_resources
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.05s
user 0m 0.03s
sys 0m 0.02s
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.05s
user 0m 0.01s
sys 0m 0.04s
# cat /hal-asos/MFastSA0/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 02311726
Maxim event duration:02311726 [BusClks]
Minimum event duration:02174807 [BusClks]
Total events duration:0004486533 [BusClks]
#
```

(b) Tests using V4_00_F accelerators at 50 MHz and 8 kB.

Figure 5.25: Performance of asynchronous designs using V4_00_B and V4_00_F accelerators.

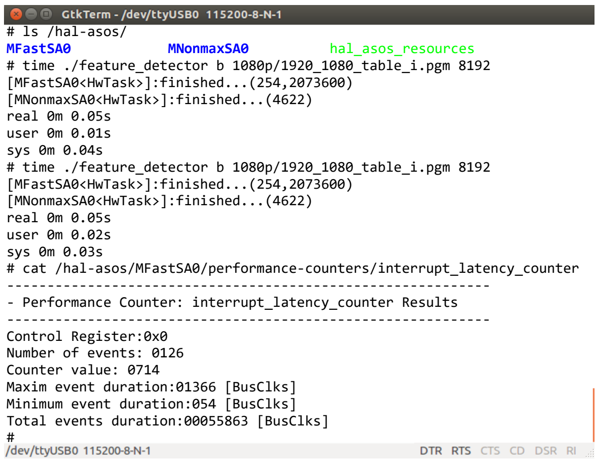
On the left side, Figure 5.25b show the same tests performed to the application based on the system memory, i.e., through accelerator V4_00_F. The execution time took 50 milliseconds for each consecutive run in two tests. When compared to the standalone accelerator with the synchronous control unit (i.e., 90 milliseconds, figure 323) and the same accelerator version, the asynchronous design results are 44% faster. The done performance counter registers execution times of 43.50 milliseconds and 46.23 milliseconds for the MFastSA0. In turn, these translate into a processing ratio of 1.05 clocks per pixel.

The previous tests were repeated using the same conditions in order to evaluate the interrupt events triggered to the host system. Figure 5.26 shows the performances of both applications using the *interrupt_latency* counter. On the right, Figure 5.26a shows the tests run in the application based on V4_00_B accelerators. As in the previous tests, a block size of 8 kB was used, fragmenting the input image in 254 blocks which require equal number of transfers to the LRAM. Results demonstrate equivalent execution times and the *FastSA* interrupt latency counter is showing two interrupt events from the two executions. The least latency time observed was 774 clock cycles (i.e., 0.0155 milliseconds), and comprises the time interval since the interrupt signal was activated until the instant when the host system acknowledges the event by clearing the status of interrupt controller.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
FastSA0            NonmaxSA0          hal_asos_resources
# time ./feature_detector a 1080p/1920_1080_table_i.pgm 8192
[FastSA0<HwTask>]:finished...(254,2073600)
[NonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.08s
user 0m 0.03s
sys 0m 0.06s
# time ./feature_detector a 1080p/1920_1080_table_i.pgm 8192
[NonmaxSA0<HwTask>]:finished...(4622)
[FastSA0<HwTask>]:finished...(254,2073600)
real 0m 0.08s
user 0m 0.03s
sys 0m 0.06s
# cat /hal-asos/FastSA0/performance-counters/interrupt_latency_counter
--------------------------------------------------------------
- Performance Counter: interrupt_latency_counter Results
--------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 0905
Maxim event duration:0905 [BusClks]
Minimum event duration:0774 [BusClks]
Total events duration:0001679 [BusClks]
#
```

(a) Tests using V4_00_B accelerators at 50 MHz and 8 kB block size.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

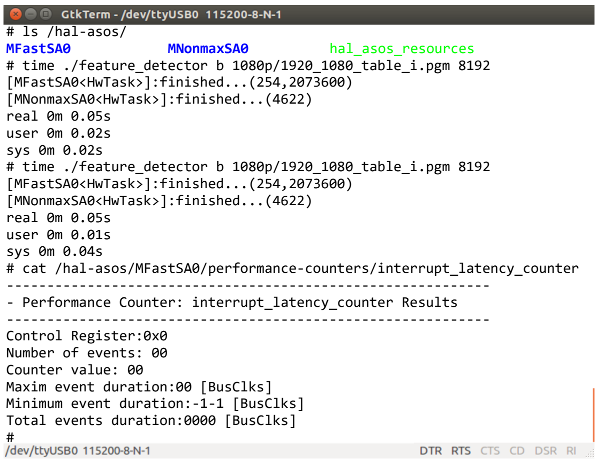
```
# ls /hal-asos/
MFastSA0            MNonmaxSA0          hal_asos_resources
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.05s
user 0m 0.01s
sys 0m 0.04s
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.05s
user 0m 0.02s
sys 0m 0.03s
# cat /hal-asos/MFastSA0/performance-counters/interrupt_latency_counter
--------------------------------------------------------------
- Performance Counter: interrupt_latency_counter Results
--------------------------------------------------------------
Control Register:0x0
Number of events: 0126
Counter value: 0714
Maxim event duration:01366 [BusClks]
Minimum event duration:054 [BusClks]
Total events duration:00055863 [BusClks]
#
```

(b) Tests using V4_00_F accelerators at 50 MHz and 8 kB block size.

Figure 5.26: Performance of the interrupt latency of V4_00_B and V4_00_F accelerators.

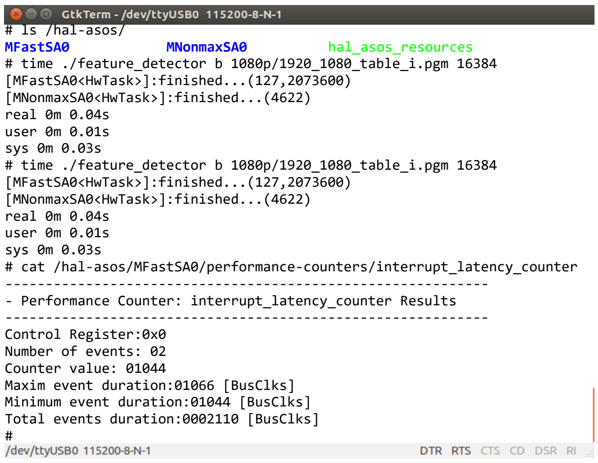
Figure 5.26b shows the results of the design based on V4_00_F accelerators, for the same two runs and 8kB block size. The MFastSA performance counter registers 126 interrupt events for the two runs, with a response of 54 cycles as the best time. The impairment related to the interrupt events between the two versions is closely linked to the performance observed at the host system. Although in the first case the total execution time is higher (i.e., 80 milliseconds compared to 50 milliseconds), the *HW-Task* surpasses the performance of the host system that is using a low-bandwidth bus to exchange data. For this reason, the control unit waited for the new block before starting processing, practically in every transfer performed by the host system.

On the other hand, when the design uses the system memory to exchange data, the performance on the host system increases significantly and the accelerator is overtaken in some transfers. To find out how these two parameters influence the application that uses system memory, the tests were repeated by (1) raising the frequency to 100 MHz and maintaining the block size, and (2) using the new design frequency and raising the block size to 16 kB. Figure 5.27 shows the results for these two tests performed with accelerators V4_00_F.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
MFastSA0            MNonmaxSA0           hal_asos_resources
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.05s
user 0m 0.02s
sys 0m 0.02s
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.05s
user 0m 0.01s
sys 0m 0.04s
# cat /hal-asos/MFastSA0/performance-counters/interrupt_latency_counter
------------------------------------------------------------
- Performance Counter: interrupt_latency_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 00
Counter value: 00
Maxim event duration:00 [BusClks]
Minimum event duration:-1-1 [BusClks]
Total events duration:0000 [BusClks]
#
```

(a) Tests using V4_00_F accelerators at 100 MHz and 8 kB block size.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
MFastSA0            MNonmaxSA0           hal_asos_resources
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 16384
[MFastSA0<HwTask>]:finished...(127,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.04s
user 0m 0.01s
sys 0m 0.03s
# time ./feature_detector b 1080p/1920_1080_table_i.pgm 16384
[MFastSA0<HwTask>]:finished...(127,2073600)
[MNonmaxSA0<HwTask>]:finished...(4622)
real 0m 0.04s
user 0m 0.01s
sys 0m 0.03s
# cat /hal-asos/MFastSA0/performance-counters/interrupt_latency_counter
------------------------------------------------------------
- Performance Counter: interrupt_latency_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 01044
Maxim event duration:01066 [BusClks]
Minimum event duration:01044 [BusClks]
Total events duration:0002110 [BusClks]
#
```

(b) Tests using V4_00_F accelerators at 100 MHz and 16 kB block size.

Figure 5.27: Performance of the interrupt latency of V4_00_F accelerators.

The results in Figure 5.27a, show that raising the design frequency to 100MHz while maintaining the block size does not improve the performance of the application, i.e., achieving the same 50 milliseconds mark. On the other hand, the results of the interrupt latency counter indicate that the V4_00_F accelerator outperforms the host system since zero interrupt events were required. For the current application parameters, the block size in use introduces an already mentioned degree of scheduling computation overhead, thus preventing the decrease of the total execution time. As so, it is fair to say that the accelerator outperforms host system due to preemption and delays in the rescheduling operations.

Figure 5.27b shows the test performed to the same design, using 100 MHz clock frequency and raising the input block size. The two tests show the same application reducing its the total execution time to 40 milliseconds. In the two consecutive executions, two interrupts were sent to the host system with very similar response times, which in the expected balance between both computing resources, it can be attributed to writing on the Linux *stdout*. To confirm, the design was connected to Vivado's logic analyzer and the results can be seen in Figure 5.28.

In this test, the trigger event was configured for the occurrence of an interrupt through the accelerator's *interrupt_pin* signal, with position 20000 in a trigger window of 32768 samples. In addition, *MFastSA0* internal signals were added to include: the control state register; available space in the datapath register; six control signals from the *M00_System* interface; the *SysMutex* locked signal; and the trigger source signal. As it can be seen on the 'T' marker in red, the interrupt signal was activated after the *MFastSA0* control reached the *s90_print_stdio* state.

Looking back in the plot window, it can also be seen that the control reaches s7 in sample 6,365 while waiting 4,312 clock cycles from the confirmation of the current block. In the 7,860 sample marker, the datapath completes the processing of the data stored in the line zero ram with an available space of 512 words. The effective space is based on the approximate size of the line (2048 bytes for 1920 pixels). The active *sysmutex_locked* signal indicates the instant when the host system transfers the last memory block and in the sample marker 10,677, the control receives transferred block confirmation.

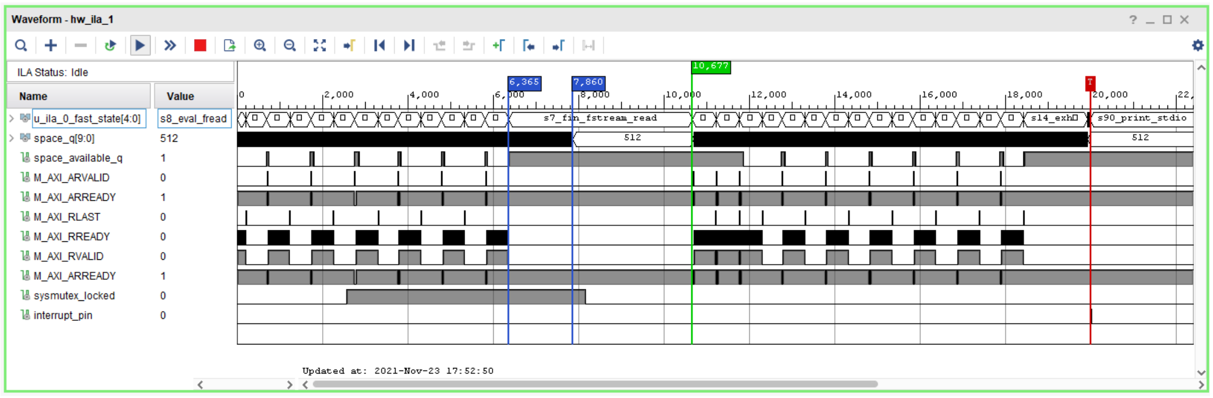


Figure 5.28: Logic Analyzer using interrupt as trigger in the V4_00_F accelerator.

The datapath resumes processing for the remaining pixels of the image, and from the AXI bus signals it can be seen that three consecutive bursts, transfer data from the SYSRAM memory to the datapath , which quickly runs out of space. From this moment on, transfers are interspersed with sleep states and while the last 9,216 pixels of the image are copied in 9 bursts of 256 words of 4 pixels (i.e., 2,073,600 pixels are fragmented in 126 complete 16 kB blocks, plus 9216 pixels in the last incomplete block). At s14 (i.e., beyond sample 18,100), the control waits for the last pixel on line zero RAM to enter the pixel matrix, before proceeding until s90 to print results to the *stdout*.

5.8 Stand-alone Asynchronous Single-task

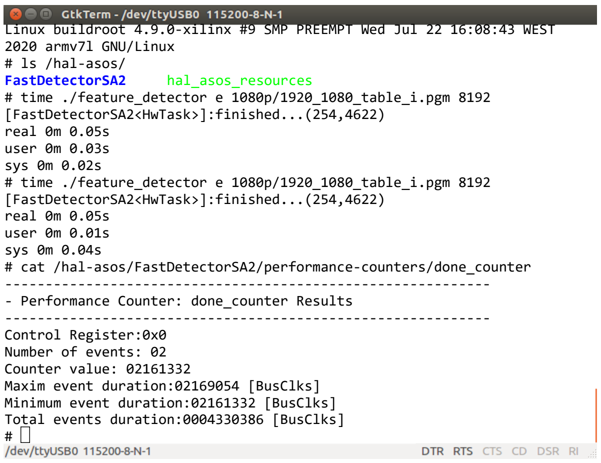
Once the functionality of the asynchronous design is proven, the next step considers a *HW-Task* that merges the two state machines of Figure 5.23 into a Single-task design. Previous tests carried out showed that the pixel reading is crucial for the application's performance, and for this reason, the design maintains the use of system memory to manipulate the input file. To maximize the use of resources, the writing of the output file uses the LRAM memory to store the detected corners, allowing the control to keep concurrent transfers in the two storage resources. The equivalent dual-task design is implemented using two accelerators V4_00_F and V4_00_B, and the previous *HW-Tasks*. For completeness, such design will be used for a performance comparison.

The control unit for this accelerator is divided into three logical level stages. In the first stage, the control promotes maximum performance at the input while monitoring the existence of corners at the output of the datapath. Whenever this storage space holds a number of at least 64 corners, control transfers them to LRAM while trying to keep reading at maximum performance. This copy frees the output space in the datapath to avoid congestion, and it can occur whenever the control finishes an input burst transfer of 256 words. When this stage completes an input data block, the control evaluates the need to transfer the corners to the output file. If there are at least 512 words in the LRAM, the control advances to level 2 stage, otherwise it resumes processing for another block of pixels.

Once in the second stage, the control implements corner transfer while continuing to transfer pixels to the input of the datapath at the rate of 64-word bursts. This duality divides the control actions between the two operations, in an effort to maintain the input of data while writing to the output file. All existing corners in LRAM are transferred and the worst-case scenario can occur when the input block exhausts during this stage. In this case, the control completes the current transfer and returns to the previous stage. To avoid the contention implicit in the transfer handshake, the control monitors the size of the input *message_queue*, allowing it to enter the finalize transfer state after the host system provide a response.

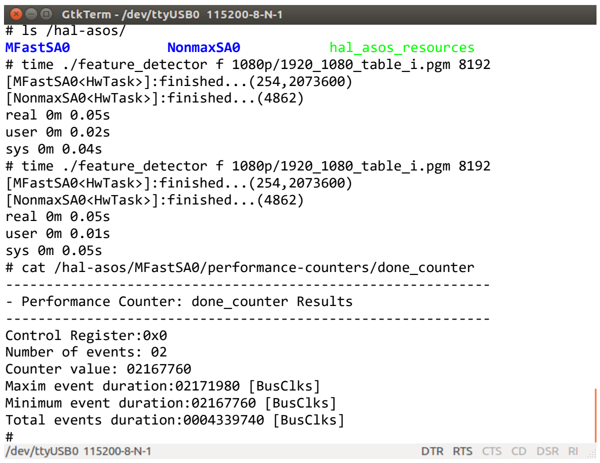
For most of the operating time, the control of the datapath is ensured by the first two stages. The final stage is achieved when the input file has been completely transferred. The control unit focuses the entire operation on the output file while the datapath finishes processing the last received line. Due to the great complexity of this control system, the analysis of its FSM was not considered, but it can be consulted in the attached Figure D.14.

To promote simultaneous performance on both interfaces, accelerator V4_00_A was used to modify the configurations of the *S01_Data* and *M00_System* interfaces to a single-clock design (see section 4.8). Also, to store a large number of corners per block while the input block is totally transferred, the LRAM storage space was raised to equal the input storage (i.e., 2x 32 kB). The resulting *HW-Task* was configured with the tag-name FastDetectorSA2, similar to previous accelerators. Figure 5.29 shows the best results of the performed tests using the 4,622 corners input file, an 8 kB input block size and a design frequency of 50 MHz, for the two equivalent designs. The left side of Figure 5.29a, shows the tests carried out on the single-task asynchronous control that uses the accelerator version V4_00_A. Although the control task is heavily conditioned by concurrent data reading and writing operations, the test performed records a total time of 50 milliseconds in each execution, and the *done* performance counter records a total of 2,161,332 clock cycles for the best *HW-Task* execution.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# ls /hal-asos/
FastDetectorSA2     hal_asos_resources
# time ./feature_detector e 1080p/1920_1080_table_i.pgm 8192
[FastDetectorSA2<HwTask>]:finished...(254,4622)
real 0m 0.05s
user 0m 0.03s
sys 0m 0.02s
# time ./feature_detector e 1080p/1920_1080_table_i.pgm 8192
[FastDetectorSA2<HwTask>]:finished...(254,4622)
real 0m 0.05s
user 0m 0.01s
sys 0m 0.04s
# cat /hal-asos/FastDetectorSA2/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 02161332
Maxim event duration:02169054 [BusClks]
Minimum event duration:02161332 [BusClks]
Total events duration:0004330386 [BusClks]
# 
```

(a) Tests using V4_00_A accelerator at 50 MHz.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
MFastSA0            NonmaxSA0           hal_asos_resources
# time ./feature_detector f 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[NonmaxSA0<HwTask>]:finished...(4862)
real 0m 0.05s
user 0m 0.02s
sys 0m 0.04s
# time ./feature_detector f 1080p/1920_1080_table_i.pgm 8192
[MFastSA0<HwTask>]:finished...(254,2073600)
[NonmaxSA0<HwTask>]:finished...(4862)
real 0m 0.05s
user 0m 0.01s
sys 0m 0.05s
# cat /hal-asos/MFastSA0/performance-counters/done_counter
------------------------------------------------------------
- Performance Counter: done_counter Results
------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 02167760
Maxim event duration:02171980 [BusClks]
Minimum event duration:02167760 [BusClks]
Total events duration:0004339740 [BusClks]
#
```

(b) Tests using V4_00_F/B accelerators at 50 MHz.

Figure 5.29: Performances of asynchronous V4_00_A and V4_00_F/B accelerators.

On the right side, Figure 5.29b shows the tests performed to the design that uses the two accelerators V4_00_F and V4_00_B. In equivalent way, such design reads the input file using the SYSRAM storage, and writes the output file using the LRAM. The same datapath is in use but it was split between the two *HW-Tasks*, as in the previous asynchronous designs. In the two consecutive tests the application executed in 50 milliseconds and the done performance counter shows the best execution for the *MFastSA0* of 2,167,760 clock cycles.

5.9 Performance Comparison

To better distinguish between the four asynchronous designs, Figure 5.30 plots the average performance of the application, when the file is varied to increase the number of corners detected. In the same graph, it is possible to see the performance results of the application when using the software accelerated algorithms. Compared to HW accelerators in asynchronous design, all accelerators outperform software-only versions using 50 MHz clock frequency and input block size limited to 8 kB.

A comparison between the four asynchronous versions confirms a dependence of the detection algorithm for the final performance of the application. When raising the number of corners, the designs that use the V4_00_B accelerators show a slight performance degradation. The design that uses two accelerators V4_00_F have a dedicated system memory interface for each accelerator to minimize bus transaction delays. For that reason, the application's overall performance remains stable with increasing corners in the output file. For completeness, Figure D.13 shows detailed block diagram connections of the design that targets the Zynq7000 on the ZC702 platform, when using V4_00_F version accelerators.

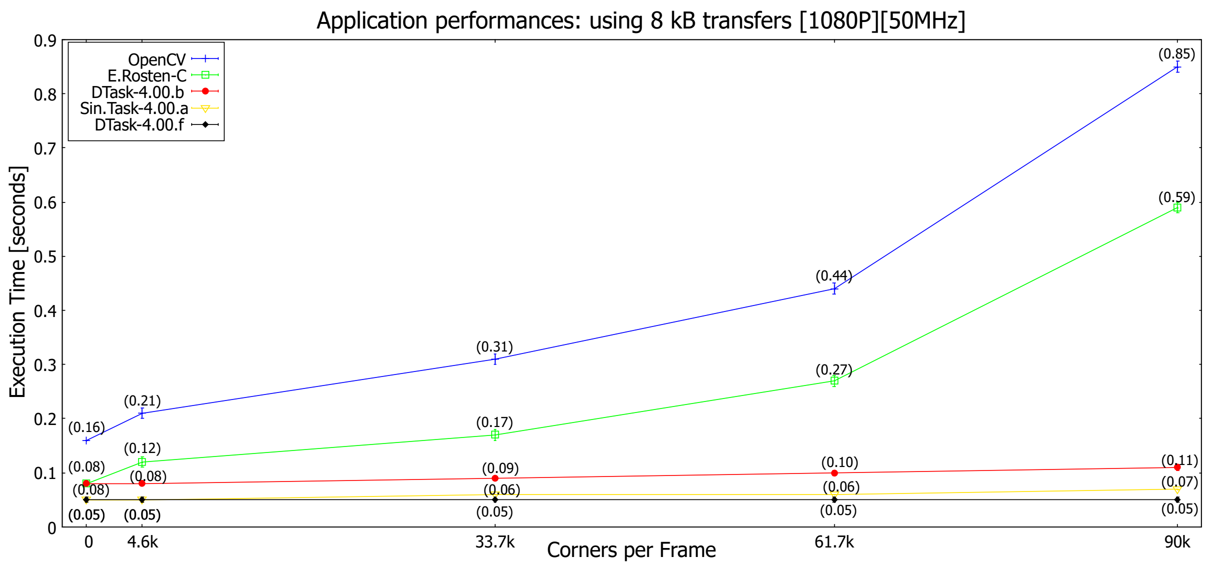


Figure 5.30: Performances for the asynchronous designs varying the input file.

To assess the effects of input block size and frequency applied to the design, Figures 5.31 plots the performance results for the fastest design, (i.e., Dual-Task based on V4_00_F accelerators) using four frequencies while varying the input block size. In all tests, the file with the largest number of corners was used, i.e., *jean.pgm* file with 90,000 corners. With a clock frequency of 25 MHz, the application performance drops to 100 milliseconds and it is not influenced by the increase in block size. For frequencies beyond 50 MHz, the overall application performance raises with the input block size, and the frequency increase has an effect on the overall performance for block sizes larger than 8 kB. Tests performed at 142 MHz did not reflect a performance increase when compared to the previous 100 MHz mark, and the 16 kB block size was sufficient to maintain the performance level.

Table 5.1 summarizes the distinct hardware accelerators discussed in this chapter, categorized by asynchronous and synchronous design strategy. It considers the target platform ZC702 from Xilinx at 100 MHz as target frequency and 16 kB as optimized block sizes for transferring input pixels. The first column

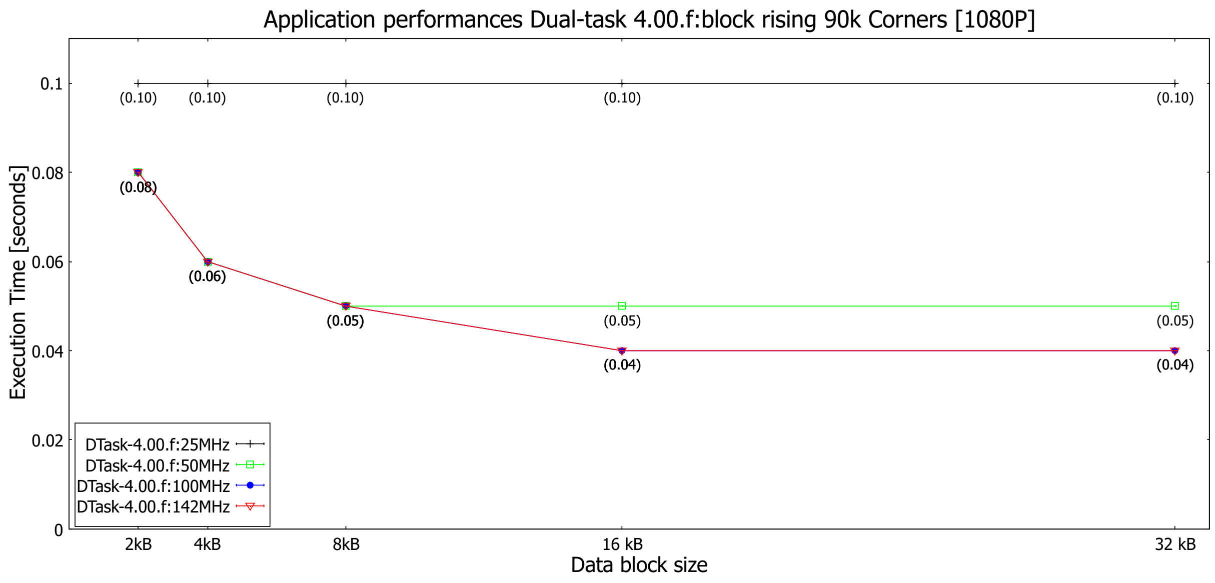


Figure 5.31: Performances using dual-task V4_00_F accelerators and 90,000 corners.

identifies the accelerator by the *HW-Task* name and the second column shows the accelerator version to which it was connected to. The third and fourth columns show the performance obtained and establish the order between the accelerators placing, the fastest ones at the top.

At top stand the designs that rely on the SYSRAM memory region to read the input file. Compared to the LRAM based design (line 4), the performance difference is marginal (i.e., 10 milliseconds) when both designs are stimulated at 100 MHz. Due to a low, or null number of interrupt events in the system, the *FastDetectorSA2* design is the most interesting and it also consumes the least logical resources. On the other hand, this is the most complex design and thus more difficult to update, maintain and functionally verify.

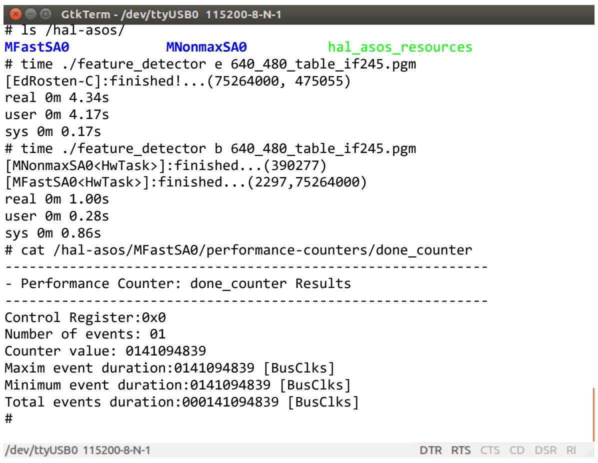
In terms of resource usage, we can say that consumption does not increases drastically when the design scales using two accelerators. From the two topmost rows (right side of the table) it can be seen that the consumption of LUTS and FFS goes up 6.6% and 4.6% respectively. In contrast, the required RAM and BRAM logic goes down by 2.6% and 8.2% respectively. Is fair to say that the second accelerator logic overhead is neglectable when compared to the resource consumed by the overall single accelerator design.

To distinguish the two control strategies discussed, we can say that the application performance improves about 30% with the asynchronous strategy. Beyond this point, the asynchronous model is more beneficial to the system because it reduces the number of interrupt events to less than half, or completely as in line

2. Compared to synchronous examples, the difference between the two fastest designs is 33%, but in contrast, the synchronous design is simpler, faster to develop and verify, and can be implemented using half of the memory resources. For the above reasons we can say that the design that is best positioned to integrate the final application is the asynchronous dual-task using two accelerators in version V4_00_F.

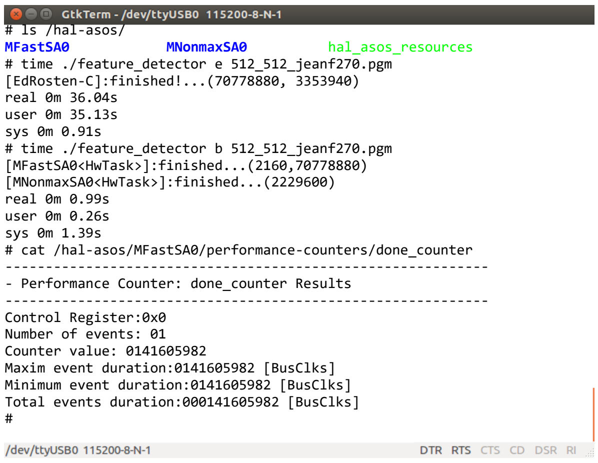
Lastly, the software and hardware accelerated application versions were compared in similar scenarios that can be found in the literature. Usually, these evaluate performance on a frame-per-second (FPS) rate using resolutions of 512x512, or 640x480, connected with digital cameras at the input of the datapath. An alternative test consists of reproducing this frame rate in numerous frames, by converting the files from the test dataset into the two above resolutions and replicated them in two input files. The files containing the lowest and highest number of corners were selected. Figure 5.32 shows results of tests performed using these two files.

On the left side, Figure 5.32a shows the tests carried out using the *table.pgm* file converted to the 640x480 resolution and replicated 245 times in the input file. For this frame length, the software application lasted 4.32 seconds which is equivalent to 56 FPS. The application using the MFastSA0 and MNonmaxSA0 accelerators, an input block size of 32 kB and a design frequency of 142 MHz, takes 1 second to process the same input, which is equivalent to 245 FPS. The *done* performance counter shows that the execution of the MFastSA0 accelerator lasted 993.63 milliseconds.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
MFastSA0            MNonmaxSA0          hal_asos_resources
# time ./feature_detector e 640_480_table_if245.pgm
[EdRosten-C]:finished!...(75264000, 475055)
real 0m 4.34s
user 0m 4.17s
sys 0m 0.17s
# time ./feature_detector b 640_480_table_if245.pgm
[MNonmaxSA0<HwTask>]:finished...(390277)
[MFastSA0<HwTask>]:finished...(2297,75264000)
real 0m 1.00s
user 0m 0.28s
sys 0m 0.86s
# cat /hal-asos/MFastSA0/performance-counters/done_counter
-------------------------------------------------------------
- Performance Counter: done_counter Results
-------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 0141094839
Maxim event duration:0141094839 [BusClks]
Minimum event duration:0141094839 [BusClks]
Total events duration:000141094839 [BusClks]
#
```

(a) Tests using V4_00_F accelerators and 640x480 resolution.


GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# ls /hal-asos/
MFastSA0            MNonmaxSA0          hal_asos_resources
# time ./feature_detector e 512_512_jeanf270.pgm
[EdRosten-C]:finished!...(70778880, 3353940)
real 0m 36.04s
user 0m 35.13s
sys 0m 0.91s
# time ./feature_detector b 512_512_jeanf270.pgm
[MFastSA0<HwTask>]:finished...(2160,70778880)
[MNonmaxSA0<HwTask>]:finished...(2229600)
real 0m 0.99s
user 0m 0.26s
sys 0m 1.39s
# cat /hal-asos/MFastSA0/performance-counters/done_counter
-------------------------------------------------------------
- Performance Counter: done_counter Results
-------------------------------------------------------------
Control Register:0x0
Number of events: 01
Counter value: 0141605982
Maxim event duration:0141605982 [BusClks]
Minimum event duration:0141605982 [BusClks]
Total events duration:000141605982 [BusClks]
#
```

(b) Tests using V4_00_F accelerators and 512x512 resolution.

Figure 5.32: Full frame-rate operation using 100 MHz clock and 32 kB input block size.

On the right side, Figure 5.32b shows the test based on the *jean.pgm* file, with a resolution of 512x512 in and input file containing 270 frames. For this frame-rate the software-based application lasted 38.04 seconds which equates to 7 FPS. The same hardware accelerated application lasted 0.99 seconds which

HW-Task(s)	**Accelerator** (version)	**Time** (ms)	**AVG** (Clocks)	**LINTC** (Evts.)	**RAM** (kB)	**LRAM** (kB)	**LUTs** cnt. (%)	**LUTRAMs** cnt. (%)	**FFs** cnt. (%)	**BRAM** cnt. (%)
Asynchronous design 100 MHz 16 kB block transfers										Zynq7000-Soc (Xilinx ZC702 platform)
MFastSA0	V4_00_F	40	2978000	1	64	1	12113 (22.77)	804 (4.62)	12564 (11.81)	10 (7.14)
MNonmaxSA0	V4_00_F		2954773	9	4	1				
FastDetectorSA2	V4_00_A	40	3255000	0	64	64	8160 (16.18)	1262 (7.25)	8027 (7.54)	21.50 (15.36)
MFastSA0	V4_00_F	40	3311000	1	64	1	11804 (22.19)	834 (4.79)	12576 (11.86)	10 (7.14)
NonmaxSA0	V4_00_B		2878388	73	-	4				
FastSA0	V4_00_B	50	4015000	1	-	64	10982 (20.64)	784 (4.51)	12010 (11.29)	24 (17.14)
NonmaxSA0	V4_00_B		3782638	37	-	64				
Synchronous design 100 MHz 16 kB block transfers										Zynq7000-Soc (Xilinx ZC702 platform)
FastDetectorSA1	V4_00_F	60	5437404	136	64	1	7786 (14.64)	722 (4.15)	7783 (7.40)	7.50 (5.36)
MxFastDetectorSA1	V4_00_F	70	5817009	136	64	1	7708 (14.49)	722 (4.15)	7856 (7.38)	7.50 (5.36)
FastDetectorSA0	V4_00_B	70	6203700	135	-	64	7255 (13.64)	720 (4.14)	7642 (7.18)	21.50 (15.36)
MxFastDetectorSA0	V4_00_B	80	7151521	133	-	64	7269 (13.66)	712 (4.09)	7559 (7.10)	21.50 (15.36)

Table 5.1: Hardware Accelerated Feature detection designs comparison.

equates to 270 FPS, and the result of the performance counter shows a duration of 997.23 milliseconds for the MFastSA0.

While in previous tests, the hardware accelerated designs achieved a processing rate of one pixel per clock, this level of performance cannot be permanent when operating in an OS environment such as Linux. This OS has a base scheduling policy that ensures a fair distribution of the CPU resource to the other tasks in the system. From the tests carried out on the hardware, we can say that the design on the FPGA allows a frame-rate of 460 FPS for a resolution of 640x480, and a frame-rate of 541 FPS for a resolution of 515x512. In the Linux environment, such performance can only be achieved by providing a video stream locally, using a digital camera connected at the input of its datapath. Example of such connectivity is the DCMI parallel interface from STMicroelectronics [33] used in some digital cameras with this resolution ranges.

To the best of our knowledge no tests have been found in the literature that consider the Linux OS and achieve the experimental performances of our designs. In most cases, the target image is already available in the design local storage consuming all RAM-based logic in the FPGA and do not consider the transfer time. Table 5.2 lists publications that can be compared to this design.

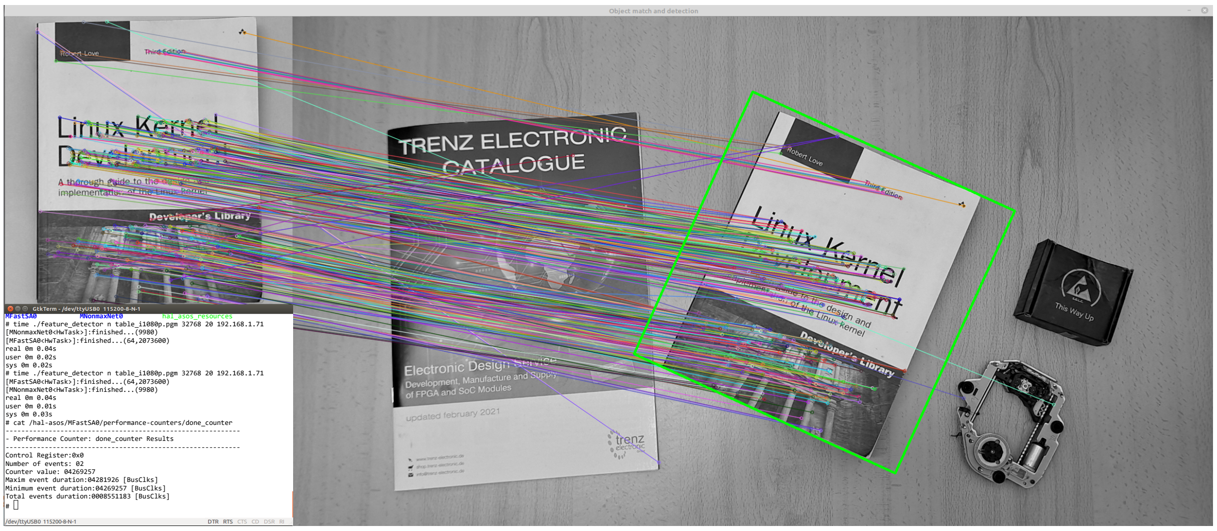
SOA	Target resource	Frame size	fps	f MHz	Platform	FFs	LUTs	BRAMs
[34]	FPGA	512x512	500	130	XC3S200-4	1547	2638	12
[35]	FPGA	515x512	310	100	XC72K325T	112166	80472	35
[36]	FPGA	640x480	55	100	Zynq-7000	3187	4257	576
(a)	FPGA	512x512	541	142	Zynq-7000 (ZC-702)	12564	12935	10
		640x480	460					
		1080p	68					
	CPU+FPGA (Linux OS)	512x512	270	142				
		640x480	245					
		1080p	36					

Table 5.2: HW-Accelerated FAST using HAL-ASOS (a) compared to the literature.

Finally, to integrate the *Feature Detection* stage in the application, the *MNonmaxSA0* was refactored to handle network TCP sockets instead of the output binary files. Changes to extended features are minimal since both resources arise from Linux virtual file model. Required changes can be seen in the attached Listing C.54. The application was also refactored and changes can be seen in the attached Listing C.55. Performed tests demonstrated a consistent total execution time of 40 milliseconds using 142 MHz, a threshold parameter of 20 and an input block size of 32 kB. Experimental results can be seen in Figure 5.33.

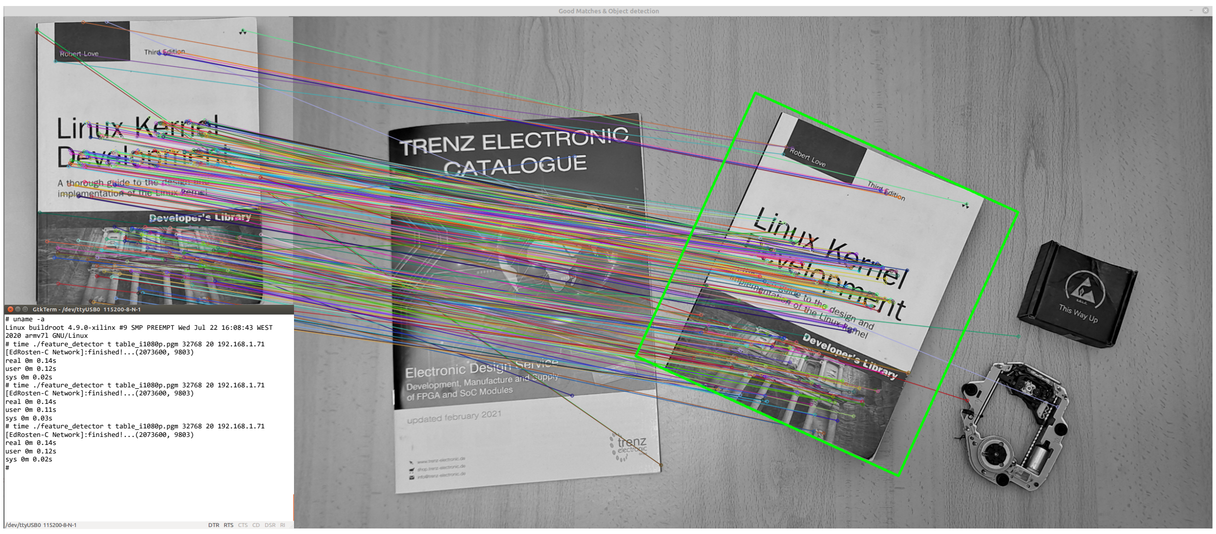
In similar conditions, the application using software *Ed.Rosten-C* was adapted to upload data using the network subsystem. Changes to the software application can be seen attached in Listing C.56 and Listing C.57. For the same conditions the software-only application achieved a total execution time of 140 milliseconds. Experimental results can be found in Figure 5.34.

The software accelerated version is 3.5 times the duration of the hardware accelerated version. Qualitatively speaking, the difference between both is imperceptible. Both applications shown similar erroneous matches that are result of the same Brute-Force algorithm in use. A distinction can be made in terms of the number of successful matches which is higher when using the hardware accelerator.


Object match and detection
GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
MFastSA0            MNonmaxNet0           hal_asos_resources
# time ./feature_detector n table_i1080p.pgm 32768 20 192.168.1.71
[MNonmaxNet0<HwTask>]:finished...(9980)
[MFastSA0<HwTask>]:finished...(64,2073600)
real 0m 0.04s
user 0m 0.02s
sys 0m 0.02s
# time ./feature_detector n table_i1080p.pgm 32768 20 192.168.1.71
[MFastSA0<HwTask>]:finished...(64,2073600)
[MNonmaxNet0<HwTask>]:finished...(9980)
real 0m 0.04s
user 0m 0.01s
sys 0m 0.03s
# cat /hal-asos/MFastSA0/performance-counters/done_counter
-------------------------------------------------------------
- Performance Counter: done_counter Results
-------------------------------------------------------------
Control Register:0x0
Number of events: 02
Counter value: 04269257
Maxim event duration:04281926 [BusClks]
Minimum event duration:04269257 [BusClks]
Total events duration:0008551183 [BusClks]
# 
```

Figure 5.33: Hardware accelerated object detection application results.


Good Matches & Object detection
GtkTerm - /dev/ttyUSB0 115200-8-N-1
/dev/ttyUSB0 115200-8-N-1
DTR RTS CTS CD DSR RI

```
# uname -a
Linux buildroot 4.9.0-xilinx #9 SMP PREEMPT Wed Jul 22 16:08:43 WEST
2020 armv7l GNU/Linux
# time ./feature_detector t table_i1080p.pgm 32768 20 192.168.1.71
[EdRosten-C Network]:finished!...(2073600, 9803)
real 0m 0.14s
user 0m 0.12s
sys 0m 0.02s
# time ./feature_detector t table_i1080p.pgm 32768 20 192.168.1.71
[EdRosten-C Network]:finished!...(2073600, 9803)
real 0m 0.14s
user 0m 0.11s
sys 0m 0.03s
# time ./feature_detector t table_i1080p.pgm 32768 20 192.168.1.71
[EdRosten-C Network]:finished!...(2073600, 9803)
real 0m 0.14s
user 0m 0.12s
sys 0m 0.02s
#
```

Figure 5.34: Software accelerated object detection application results.

Chapter 6

Conclusions and Future work

This thesis proposes the HAL-ASOS framework and its accelerator model to ease the development of Linux based embedded systems using hardware acceleration in CPU+FPGA platforms. Throughout this document we have demonstrated how the HAL-ASOS framework is positioned as an integrated design solution that provides means to design systems that fit in the requirements of the target application. By use of its design methodology, the framework promotes the reuse of legacy software, where strategic pieces can be offloaded to FPGA computation. In a controlled way, the advance in the methodology stages, promotes design scalability guided by the achievement of the system performance metrics. The proposed accelerator model integrates the Linux OS and its programming models into a unified design that was usually and independently split between hardware and software layers. At the same time, this model benefits from the existing computational resources, placing the CPU of the target platform and the FPGA resource at the service of the application.

In Chapter 2 we have applied the design methodology to a case study application that deals with file encryption using the 128-bit AES algorithm. It was possible to modify the application, guided by a performance achievement that was based on the CPU workloads, observed in its original software-only format. This chapter demonstrates the applicability as well as each of the development stages when using the HAL-ASOS framework.

In Chapter 3 the main components in the accelerator model were discussed. We introduced some of the features implemented by the microprogram and the *Kernel Core* design that ultimately support the proposed programming model. The Extended Layer in the *HW-Task* concept and the HAL-ASOS programming model were also discussed. Using the HDL procedure packages and the corresponding hardware system calls, it was possible to extend the Linux programming models to the *HW-Task* design. The chapter ends describing the fundamental interfaces provided to the surrounding hardware. With a single format in data exchange proposed by the Generic Bus, the design becomes more flexible, partitioned into multiple clock domains that connected through a bidirectional handshake protocol.

From the design discussed in this chapter resulted a journal publication entitled: *HAL-ASOS accelerator model: evolutive elasticity by design*.

In chapter 4, singularities of the accelerator model that specialize the design in the different existing versions were discussed. The indispensable components that sustain the local storage and the synchronism in the proposed model were also evaluated. Other optional components, such as the performance counters and the *ZeroCopy* units were also discussed. The chapter concludes by distinguishing the specific versions of the accelerator model, implemented from a single description and using HDL configurations to instruct the synthesis and implementation phases of each design.

In chapter 5, we put the different versions of the accelerator to a test, using a case study that falls within the scope of computer vision applications. In this chapter, we were able to summarize the different characteristics and application details that target each accelerator version. The design was explored to its limit in the application performance when targeting the Xilinx Zynq-7000 SoC in the ZC702 platform. Following a synchronous design topology for FPGA-based circuits, the application was implemented and tested on the target platform. The performance results were compared with algorithms that use software-based acceleration and can be found in the literature.

Subsequently, the asynchronous programming paradigm supported by the HAL-ASOS accelerator model was applied. This design change improved performance results, dropping the execution times by 33% to 45% within the different accelerator versions used. On the other hand, it also proved the ability to mitigate the number of interrupt events launched by each accelerator, turning its behavior into a purely asynchronous entity. In turn, the final design was compared with designs that can be found in the literature under the similar test conditions. We considered that the performance gains were significant when compared to state-of-the-art implementations and due to its uniqueness in the application of asynchronous circuits, this chapter results in a publication that is under revision.

When compared to other accelerator models in the literature, the HAL-ASOS model provides an easy and complete integration of the design in the Linux operating system, promoting the accelerator to a first-class computation entity, in which Linux is the target operating system of this model. No other accelerator allows an *HW-Task* design with this level of programmability. With the microprogram features, the accelerator model becomes more flexible, a deterministic design that is easy to modify and validate. In this sense, future changes will expand the microprogram concept to the design of the *HW-Task*, allowing it to be reused and reconfigured by the HAL-ASOS file system. This kind of changes would allow a behavior modifiable control unit, or a change in the extended features level, and thus providing means for a single *HW-Task* design to be used in the different scenarios discussed in Chapter 5.

The early-stage Co-Simulation and late-stage Full system simulation supports, allow first to achieve initial expectations about what the application needs to be before commitment to hardware, and later, the complete validation of the target system. It considers the distinct technological stacks, starting from the software application, and moving towards a complete system that includes the HAL-SOS file system, the Linux operating system and the design of the accelerator(s). To the best of our knowledge, there is no tool that integrate this level of support to a single DSE. On the other hand, its availability strongly depends on the QEMU tool and the target platforms that this software makes available.

In the current implementation, the proposed accelerator model lacks a native memory region within the Linux OS. Currently the HAL-ASOS file system allocates all required memory from the kernel space region. This condition will limit the accelerator model applicability when the design requires storage with magnitudes in the megabytes order and establishes a continuous memory allocation as a condition. Possibly the memory layout observed in graphic chipsets would be a better approach, but this hypothesis was not considered because it is strongly dependent on the operating system evolutive nature and also in the memory hierarchies in distinct target platforms. Alternatively, this region could be redesigned through multiple memory blocks, facilitating its allocation in different memory locations, leaving the design of the *HW-Kernel* and its microprogram in charge of virtualizing the resulting blocks, through a single continuous memory space.

Once recognized the benefits of a flexible design that results from the easily modifiable microprogram, this design-level lacks ways to interact with more than one entity in the *HW-Task* design. This singularity represents a limiting factor when the design scales concurrently through independent control flows. Usually, this condition forces a partitioning of the design using two or more accelerators. This was the scenario of the first phase of asynchronous implementation in Chapter 5, while attempting to containing complexity in the resulting design. The final design encompassed all possible states in the concurrency between the two independent flows, while establishing an unbalanced processing that leans towards reading the input file. We consider that a multi-task interface could be implemented using an arbitration similar to that existing in the Generic Bus. The two scheduling policies present will allow the HW-Kernel to handle execution requests from more than one control unit.

The elasticity that the microprogram provides in the model accelerator can, at the same time, be a weak point if proper precautions are not ensured. The occurrence of runtime errors, either due to incorrect matching between system call invocation and accelerator version, or software changes in the host system, will ultimately cause a microprogram failure and consequently abort the execution, rendering the design to an unusable state. On the other hand, the framework does not support the development and compilation of new system calls which can lead to incorrect representations of the microprogram. A response system call was planned, to interact at the file system level, reporting this malfunction condition and, therefore, issuing a microprogram update. This condition was tested and revealed to be a sensitive requirement that overexposes *Kernel Core* to the level of correct functioning and security. As such, it requires a broader response to the system's technological stack level to handle erroneous microprogram updates while ensuring some level of security against abnormal operation.

Last but not least, all word transfers are one clock cycle rated, with the exception of the system bus. This is a design limitation that results from an abstraction for this interface alongside with the resource shared condition between *Kernel Core* and *ZeroCopy* units. Being a significant interface in the accelerator model, it should be improved as it can compromise its effective performance.

Summing up, this work was driven by the following design principles, (1) evolutive elasticity, (2) deep semantic integration, (3) mixed asynchronous–synchronous approach and, (4) performance and power efficiency as a sort of by-product, leveraged through event-driven, laziness and microcode dynamicity approaches. Comparing to the state-of-art on FPGA assisted acceleration, the main contributions of this work are:

1. The **elasticity by design** is tightly coupled with the evolutive nature of the Linux OS and uses microcode updates to prevent changes from impacting synchronization, memory, or ABIs.

2. A **deep semantic integration** with Linux, promoting transparency and the HW-Task programmability towards an accelerator model as first-class computing unit.

3. To handle complexity, we provide a **methodology and tools to fully support the technology stack design**, including verification and deployment of complete solutions that fit in the application

requirements using just the right resources.

4. A novel **mixed asynchronous–synchronous design** that is event-driven, communicates by means of handshake protocols and provides a fast design capable of real-time operation.

5. **Performance and power efficiency as a sort of by-product**, offloading software computation to specialized hardware circuitry and minimizing the energy consumption by enabling clocks only when there is computation to be performed. As such, we believe that this mixed asynchronous–synchronous approach will be widely used in the future.

Appendix A

Resource Addressing

In the accelerator model, we can find three distinct areas of resource addressing, when using the S00 and S01 interfaces, and in the Local-BUS. Since in S01 all addressing is implemented by the FPGA memory blocks, the design challenge was concentrated on the S00 and on the Local-BUS. To implement transparency between memory layouts, the accelerator design adopts a unique page format in the register area. In this way, the Local-BUS uses the same page formats and offsets as the S00 interface, despite only mapping the first two pages and the LRAM. With respect to LRAM, it uses the addressing capability in this peripheral, in similar way to S01 interface. Therefore, the addressing task is more demanding in the case of the S00 interface, since it maps a register area four times larger. With that in mind, throughout this section we consider the layout of the S00 memory as a reference for the design, while ensuring that the final format must be applicable to the Local-BUS.

Generically, address decoding is known to be resource-greedy and so we tried to keep the consumed resources under acceptable levels. In this design we implement a two-level address decode, that in level 0 uses the 3 most significant bits (A8:A6) to divide the address range in 8 pages. The remaining 4 bits (A5:A2) are routed to the level 1 altogether with a new 8-line page select bus. This subgroup of address lines is distributed over the 8 pages in the address range. The target address is now specified by the value of this subgroup alongside with the individual selection line on each page. Figure A.1 shows a block diagram that describes this design.

To decode the target page, the component implements a LUT of 4 inputs (L0) that gives rise to the page select bus, by using the three address bits together with the CS line in its inputs. The next decoding level follows the same approach and implements LUTs of five inputs (L1 to L8), using the four address bits that

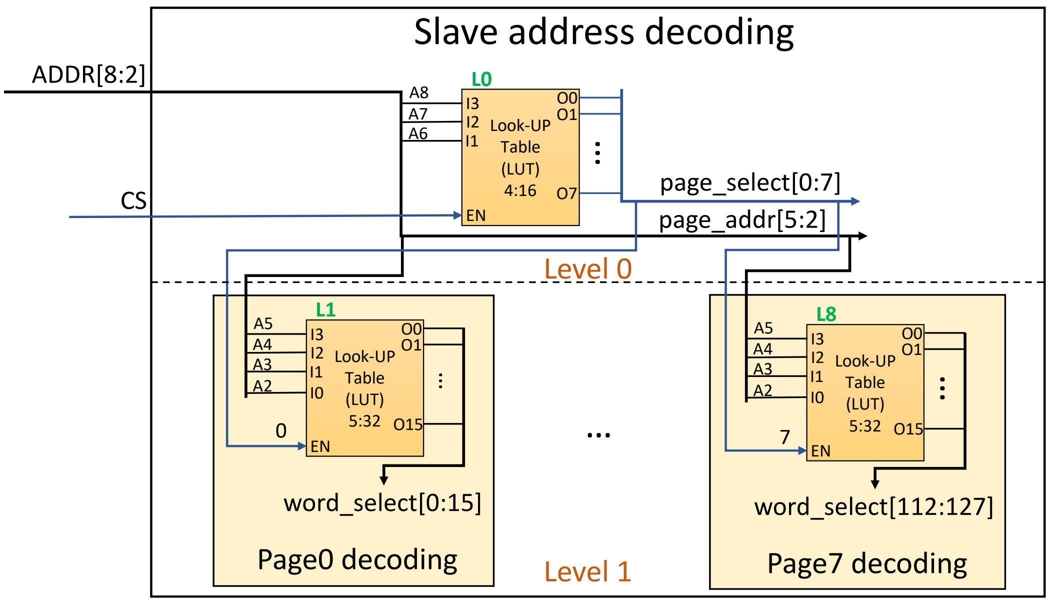
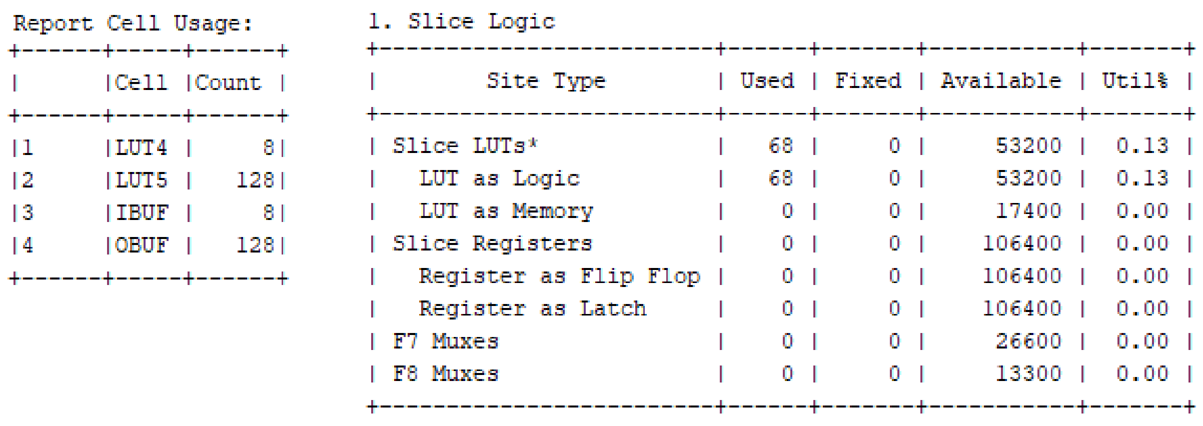


Figure A.1: S00 interface - Two-level address decoding design.

it receives together with each individual page select. The output from this level is a word select bus that identifies one of the 16-word within a target page.

To assess the impact on the consumed resources of this two-level address decoding approach, we have selected a Xilinx platform ZC702 and a synthesis of the HDL description was generated. In Figure A.2 we can see an excerpt from the resource report obtained at synthesis stage from, given by Vivado 2019.03. In the first line of the report cell usage, we can identify the 8-bit LUT4 used to implement L0, and in the second line, we can see 128 LUT5 used to implement L1 to L8. At this stage, the design is using 7 address lines together with 1 CS line as inputs to provide 8 buses of 16 word select lines as outputs. According to the Slice Logic results, we can conclude that the design maps to this FPGA device using 68 slices where only LUT logic elements are used.

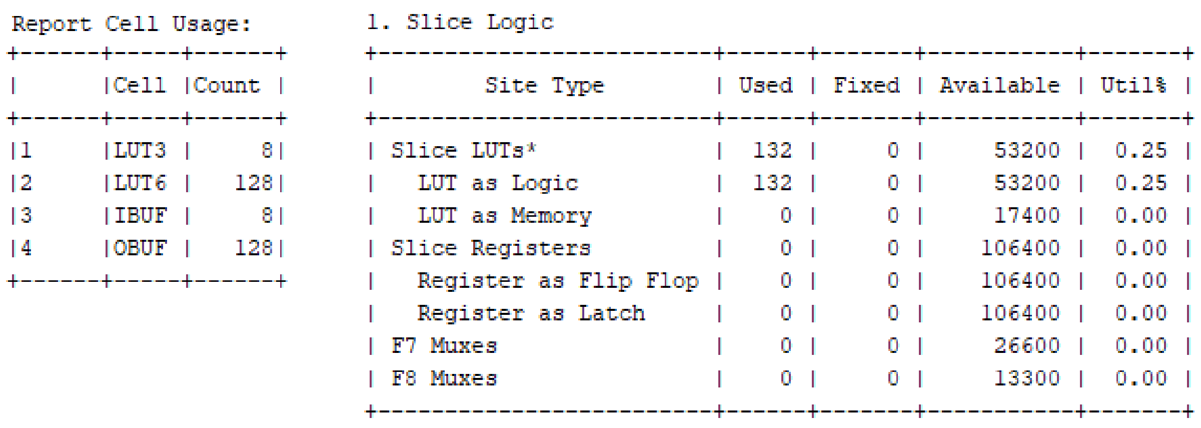


```
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |LUT4 |     8|
|2     |LUT5 |   128|
|3     |IBUF |     8|
|4     |OBUF |   128|
+------+-----+------+
```

```
1. Slice Logic
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   68 |     0 |     53200 |  0.13 |
|   LUT as Logic          |   68 |     0 |     53200 |  0.13 |
|   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
| Slice Registers         |    0 |     0 |    106400 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    106400 |  0.00 |
|   Register as Latch     |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |     13300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Figure A.2: S00 interface - Two-level address resource usage in Zynq7000.

For comparison, the same ratio of inputs and outputs was applied to a single level of address decoding design. This HDL description seeks to implement a single LUT of 8 inputs for 128 outputs. The excerpt of consumed resources after synthesis stage can be seen in Figure A.3. From the report cell usage, we can see that in the absence of such LUT, the strategy on the Vivado led to a similar two-level address decoding using 3-input and 6-input LUTs, thus producing equivalent implementation.



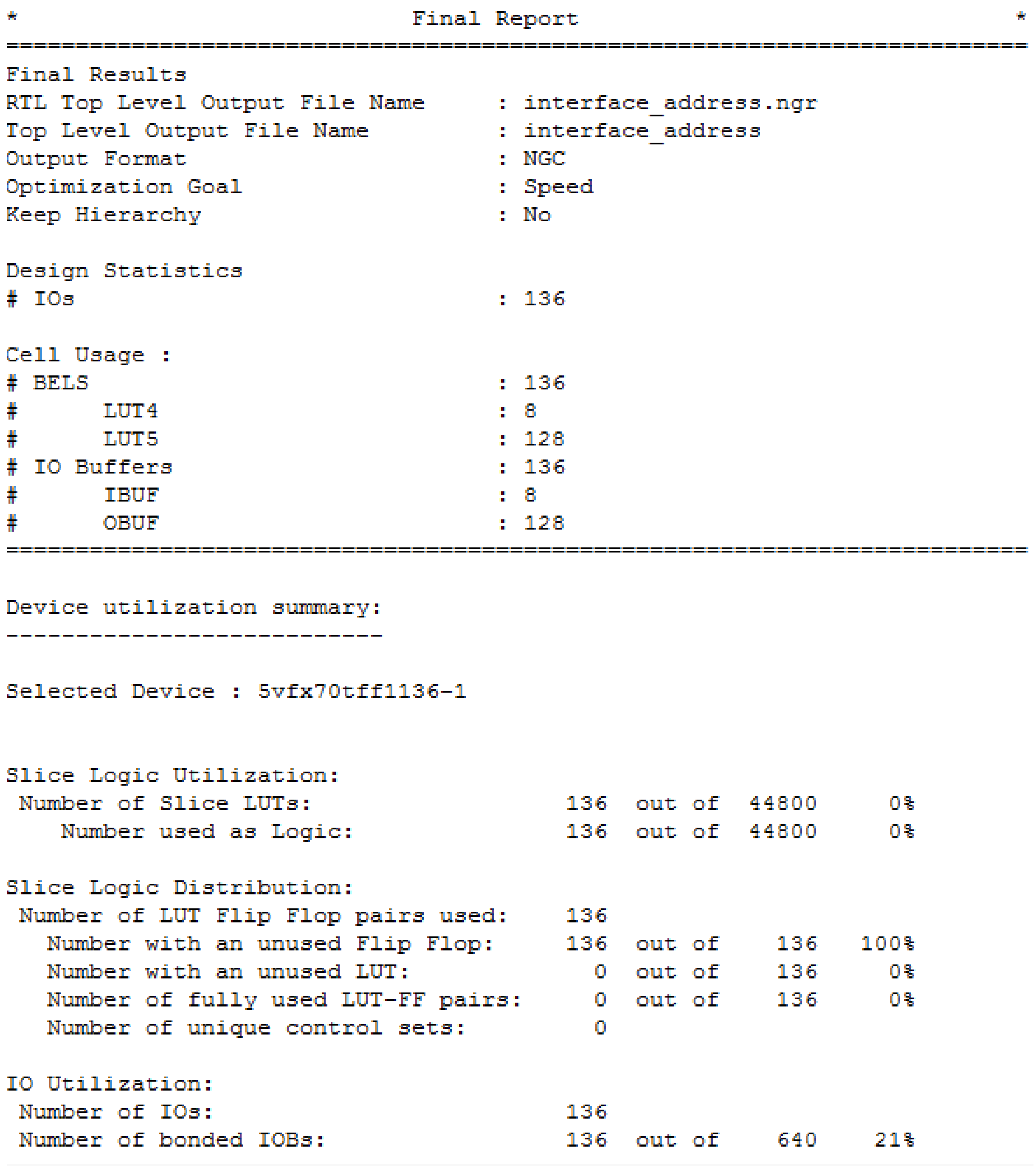
```
Report Cell Usage:            1. Slice Logic
+------+-----+------+         +-------------------------+------+-------+-----------+-------+
|      |Cell |Count |         |        Site Type        | Used | Fixed | Available | Util% |
+------+-----+------+         +-------------------------+------+-------+-----------+-------+
|1     |LUT3 |     8|         | Slice LUTs*             |  132 |     0 |     53200 |  0.25 |
|2     |LUT6 |   128|         |   LUT as Logic          |  132 |     0 |     53200 |  0.25 |
|3     |IBUF |     8|         |   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
|4     |OBUF |   128|         | Slice Registers         |    0 |     0 |    106400 |  0.00 |
+------+-----+------+         |   Register as Flip Flop |    0 |     0 |    106400 |  0.00 |
                              |   Register as Latch     |    0 |     0 |    106400 |  0.00 |
                              | F7 Muxes                |    0 |     0 |     26600 |  0.00 |
                              | F8 Muxes                |    0 |     0 |     13300 |  0.00 |
                              +-------------------------+------+-------+-----------+-------+
```

Figure A.3: S00 interface - Single-level address resource usage in Zynq7000.

A similar result was obtained using a combination of LUT4 and LUT5 in 136 slices consumed, using the ML507 target platform from the Virtex5 family. For compatibility reasons, the Xilinx ISE 14.7 tool was selected and the excerpt of this report can be found attached in Figure A.4. When analyzing the overall results, we can conclude that the 16-word page addressing model is suitable for mapping registry-based resources in FPGA devices. With the LUT6 logic element available in the ZC702 platform, it would be possible to efficiently implement 32-word pages, although this choice is not suitable for the previous platforms in the HAL-ASOS tool.

When looking back at the basics of parallel data buses, we can define the TXDATA bus as input for all words in the address range. The write operation can be determined by the word selection line in combination with the *WR_CE* line active during a clock pulse. To read the content of each word, we can implement an equivalent design, where all words are connected in parallel forming the RXDATA bus. The word that has the active selection line establishes the output signals equivalent to its logical content, while all the others remain in the high impedance state. This uniqueness condition determines a single active logical group of signals that sources the 32-bit RXDATA output.

Traditionally, this output circuit is implemented using tri-state buffers in all signals, where the value in each bit can be alternated between its corresponding logical value or an intermediate state of high impedance



```
*                              Final Report                              *
==========================================================================
Final Results
RTL Top Level Output File Name     : interface_address.ngr
Top Level Output File Name         : interface_address
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : No

Design Statistics
# IOs                              : 136

Cell Usage :
# BELS                             : 136
#      LUT4                        : 8
#      LUT5                        : 128
# IO Buffers                       : 136
#      IBUF                        : 8
#      OBUF                        : 128
==========================================================================

Device utilization summary:
---------------------------

Selected Device : 5vfx70tff1136-1


Slice Logic Utilization:
 Number of Slice LUTs:                   136  out of  44800     0%
    Number used as Logic:                136  out of  44800     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     136
   Number with an unused Flip Flop:      136  out of    136   100%
   Number with an unused LUT:              0  out of    136     0%
   Number of fully used LUT-FF pairs:      0  out of    136     0%
   Number of unique control sets:          0

IO Utilization:
 Number of IOs:                          136
 Number of bonded IOBs:                  136  out of    640    21%
```

Figure A.4: S00 Interface - Two-level address resource usage in Virtex 5 family.

denoted as 'Z'. To apply this approach in the slave decoder, the design evolves to the simplified architecture that can be seen in Figure A.5. In that figure, the LUTs that implement the two level-address decode and the resulting select lines are visible using blue color. For simplicity only the pages 0 and 7 are represented, and we can see 16 words attached to each page decoder. These words are purely figurative and are intended to describe the association in parallel using arrays of tri-state buffers, ATR0 to ATR127, of 32 lines each.

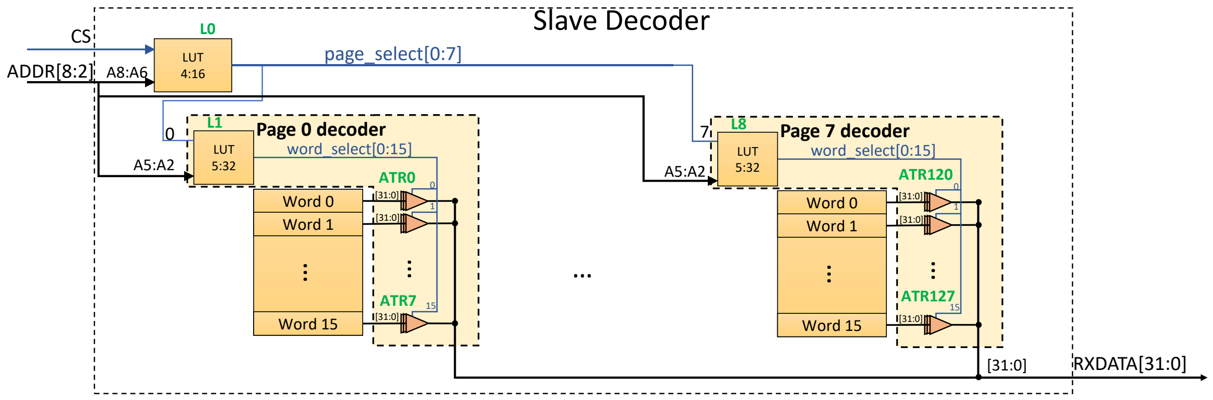
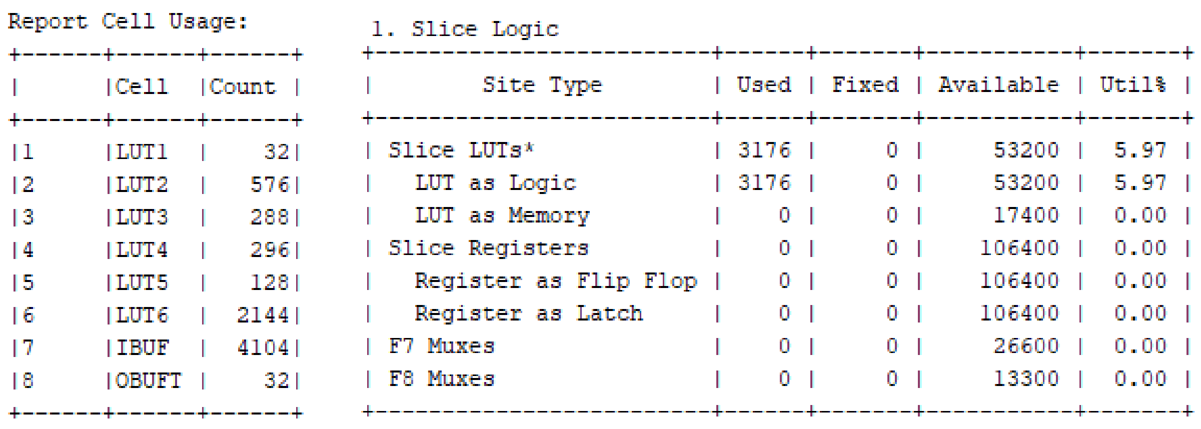


Figure A.5: Slave decoder internal architecture block diagram.

Similar excerpt from the resource usage of this design can be seen in Figure A.6. It must be said that since the 128 words are not part of the design, the tri-state buffers were connected to top-level inputs of the slave decoder. In doing so, we expect to see an additional use of 4096 input buffers (IBUF) that will not be part of the final design.



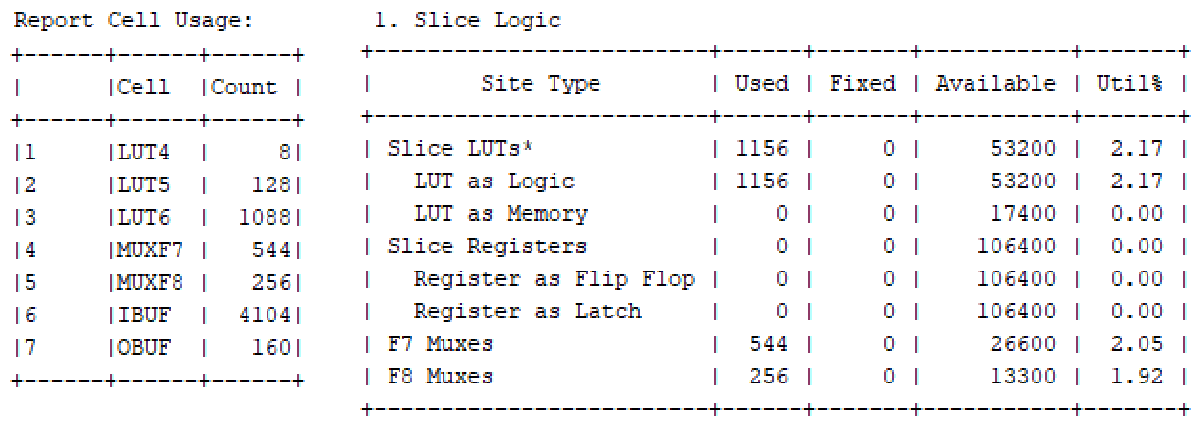
```
Report Cell Usage:
+------+------+------+
|      |Cell  |Count |
+------+------+------+
|1     |LUT1  |    32|
|2     |LUT2  |   576|
|3     |LUT3  |   288|
|4     |LUT4  |   296|
|5     |LUT5  |   128|
|6     |LUT6  |  2144|
|7     |IBUF  |  4104|
|8     |OBUFT |    32|
+------+------+------+
```

```
1. Slice Logic
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             | 3176 |     0 |     53200 |  5.97 |
|   LUT as Logic          | 3176 |     0 |     53200 |  5.97 |
|   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
| Slice Registers         |    0 |     0 |    106400 |  0.00 |
|   Register as Flip Flop |    0 |     0 |    106400 |  0.00 |
|   Register as Latch     |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                |    0 |     0 |     26600 |  0.00 |
| F8 Muxes                |    0 |     0 |     13300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Figure A.6: S00 interface - Resource usage for tri-state datapath design in Zynq7000.

An analysis of the report cell usage shows that this design does not qualifies for implementation using FPGA devices. The parallel word associations have been transformed into equivalent LUT-based circuits, and we can notice that only 32 tri-state buffers (OBUFT) are in use. This transformation resulted in a dispersion in the number of LUT inputs, with a predominance in LUT6 and LUT2. In summary, it can negatively impact the quality factor of the design, since the use of LUT1 to LUT4 are implemented by reusing LUT5 or LUT6 available in each slice. As a result, the use of resources reaches a total consumption of 3176 slices, which represents 5.97% of the available resources.

A viable alternative is to apply multiplexing and take advantage of the MUX (F7 and F8) that can be found in each slice. Applying the same principle of the two-level addressing, the page-level can multiplex the 16 words into a 32-bit output. The decoder-level can multiplex the 8 inputs from each page, to give rise to the 32-bit RXDATA of the slave interface. To ease design mapping, each 32-bit word is connected using a parallel of 4 MUX with 16 inputs of 8 bits, and each page word is connected to a parallel of 4 MUX with 8 inputs of 8 bits. In Figure A.7 we can see the resource usage for this alternative.



```
Report Cell Usage:          1. Slice Logic
+------+------+------+     +-------------------------+------+-------+-----------+-------+
|      |Cell  |Count |     |        Site Type        | Used | Fixed | Available | Util% |
+------+------+------+     +-------------------------+------+-------+-----------+-------+
|1     |LUT4  |     8|     | Slice LUTs*             | 1156 |     0 |     53200 |  2.17 |
|2     |LUT5  |   128|     |   LUT as Logic          | 1156 |     0 |     53200 |  2.17 |
|3     |LUT6  |  1088|     |   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
|4     |MUXF7 |   544|     | Slice Registers         |    0 |     0 |    106400 |  0.00 |
|5     |MUXF8 |   256|     |   Register as Flip Flop |    0 |     0 |    106400 |  0.00 |
|6     |IBUF  |  4104|     |   Register as Latch     |    0 |     0 |    106400 |  0.00 |
|7     |OBUF  |   160|     | F7 Muxes                |  544 |     0 |     26600 |  2.05 |
+------+------+------+     | F8 Muxes                |  256 |     0 |     13300 |  1.92 |
                           +-------------------------+------+-------+-----------+-------+
```

Figure A.7: S00 interface - Resource usage for MUX datapath design in Zynq ZC702.

From the report cell usage, we can conclude that the design is using the desired logical elements. The use of LUTs is significantly lower and is influenced by the slice level organization. Here, each MUXF7 allows an 8:1 multiplexing function but it receives bits from LUT6 or LUT5 in its inputs. In turn, the outputs of these are connected to the MUXF8 inputs which together allow a 16:1 multiplexing function. The final result of the design translates into a total use of 2.17% of the available slices.

To implement the first level of address decoding, the HAL-ASOS design uses the Slave decoder component, that follows the above design considerations. To promote design reuse, this component allows a parameterized number of pages and words per page. In Figure A.8 we can see a block diagram that describes its internal architecture.

On the left side, the component implements the generic interface and is usually connected to top-level signals. On the right side, the component implements the page-level signals that may vary in number and

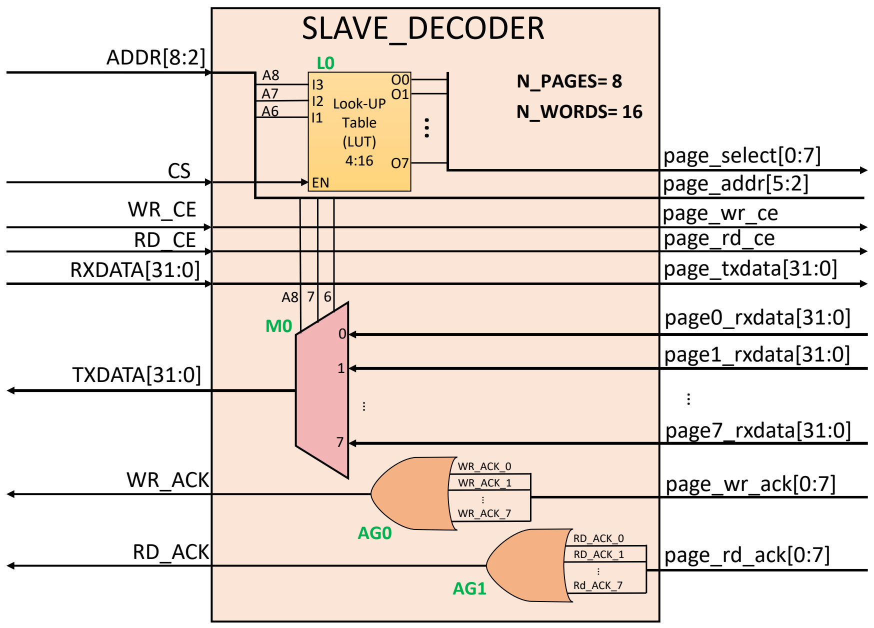


Figure A.8: Slave decoder internal architecture block diagram.

length, as a result of the chosen parameters. This degree of variability was implemented using advanced types of unconstrained number of elements, where we apply mathematical functions that are based on the received parameters, to establish the appropriate number of input and output signals in the design. For better organization, the component extends the remaining signals from the generic interface to the page-level using appropriate nomenclature. Since each data exchange demands two handshake signals, the decoder expects to receive two page-level acknowledge buses. These signals are combined internally using logic OR gates that give rise to the *WR_ACK* and *RD_ACK* signals. For completeness, the VHDL description of this component can be seen in Listing A.1 and in Listing A.2 we've included the VHDL description for its architecture.

In the second level of the address decoding, the HAL-ASOS design implements the page decoder component. Here, we follow the same approach as in the previous level but using page-level inputs and word-level outputs. In Figure 3.47 we can see a diagram that describes its internal design. The main difference lies in the connection at the M0 control inputs being closed by the address bus, instead of being extended in sub-levels. As result of the single parameter, the implementation uses twice as many resources when comparing the 16 inputs with the first level of decoding.

To address the local resources in the S00 interface, the design of the accelerator implements a hierarchy of one slave decoder to eight page decoder components. The resource consumption of the final version in the synthesis stage can be seen in Figure A.10. It is worth noting that, in order to assess this level of

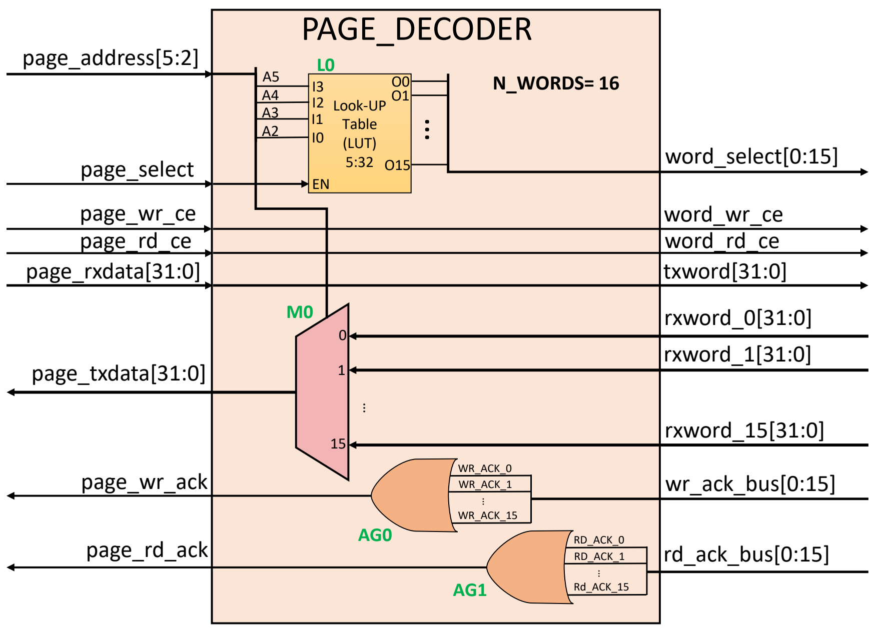


Figure A.9: Page decoder component - internal architecture.

detail, the design of the interface decoder was isolated from the remaining accelerator circuits. With the insertion of OR logic in both components, the implementation slightly increases the consumption of LUT6 and LUT3.

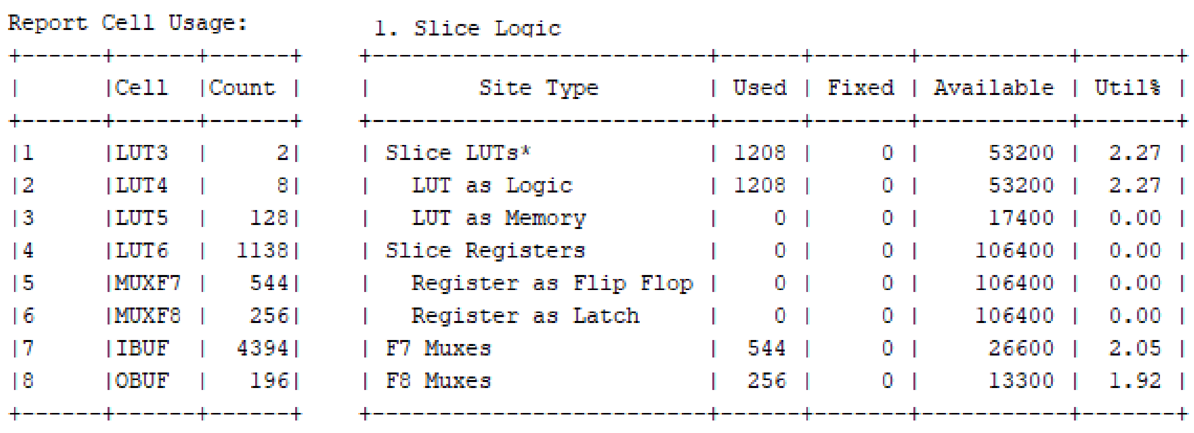

```
Report Cell Usage:          1. Slice Logic
+------+------+------+      +-------------------------+------+-------+-----------+-------+
|      |Cell  |Count |      |        Site Type        | Used | Fixed | Available | Util% |
+------+------+------+      +-------------------------+------+-------+-----------+-------+
|1     |LUT3  |     2|      | Slice LUTs*             | 1208 |     0 |     53200 |  2.27 |
|2     |LUT4  |     8|      |   LUT as Logic          | 1208 |     0 |     53200 |  2.27 |
|3     |LUT5  |   128|      |   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
|4     |LUT6  |  1138|      | Slice Registers         |    0 |     0 |    106400 |  0.00 |
|5     |MUXF7 |   544|      |   Register as Flip Flop |    0 |     0 |    106400 |  0.00 |
|6     |MUXF8 |   256|      |   Register as Latch     |    0 |     0 |    106400 |  0.00 |
|7     |IBUF  |  4394|      | F7 Muxes                |  544 |     0 |     26600 |  2.05 |
|8     |OBUF  |   196|      | F8 Muxes                |  256 |     0 |     13300 |  1.92 |
+------+------+------+      +-------------------------+------+-------+-----------+-------+
```

Figure A.10: S00 interface - Resource usage final design using ZC702 platform.

The Slice Logic utilization shows a total consumption of 2.27% of available slices, and by this number, we can consider this two-level addressing is suitable for mapping in FPGAs. This value is an indicator of preliminary consumption that may vary in the final stages of implementation, being affected by the time constraints that the accelerator model faces. These can lead to the merging of combinational logic that aims to decrease the critical path, or to reuse partial resources in other slices closer in the design.

Alternatives to this addressing model could include the use of FPGA memory blocks, programmed with a binary address map. This can be particularly advantageous in asymmetric memory layouts, where the decoder aims to accommodate the distinct address ranges of the target resources.

Listing A.1: Page decoder component - entity declaration using VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library hal_asos_v4_00_a;
use hal_asos_v4_00_a.hal_asos_configs_pkg.C_MACHINE_WIDTH;
use hal_asos_v4_00_a.hal_asos_utils_pkg.t_array_slv_32;
use hal_asos_v4_00_a.hal_asos_utils_pkg.POW2;
use hal_asos_v4_00_a.MUX32;

entity SLAVE_DECODER is
 GENERIC( N_PAGES: natural:=8;
         N_WORDS: natural:=16);
    Port ( CS : in STD_LOGIC;
           WR_CE : in STD_LOGIC;
           RD_CE : in STD_LOGIC;
           ADDR : in STD_LOGIC_VECTOR ((POW2(N_PAGES)+ POW2(N_WORDS)+2-1 downto 2);
           RXDATA : in STD_LOGIC_VECTOR (C_MACHINE_WIDTH-1 downto 0);
           TXDATA : out STD_LOGIC_VECTOR (C_MACHINE_WIDTH-1 downto 0);
           WR_ACK: out STD_LOGIC;
           RD_ACK: out STD_LOGIC;
           PAGE_SELECT : out STD_LOGIC_VECTOR (0 to N_PAGES-1);
           PAGE_WR_CE : out STD_LOGIC;
           PAGE_RD_CE : out STD_LOGIC;
           PAGE_ADDR: out STD_LOGIC_VECTOR(POW2(N_WORDS)+2-1 DOWNTO 2);
           PAGE_TXDATA : out STD_LOGIC_VECTOR (C_MACHINE_WIDTH-1  downto 0);
           PAGE_RXDATA: in t_array_slv_32 (0 to N_PAGES-1);
           PAGE_WR_ACK: in STD_LOGIC_VECTOR(0 to N_PAGES-1);
           PAGE_RD_ACK: in STD_LOGIC_VECTOR(0 to N_PAGES-1);
end SLAVE_DECODER;
```

Listing A.2: Slave decoder component - architecture description using VHDL.

```vhdl
architecture Behavioral of SLAVE_DECODER is
CONSTANT ADDR_WIDTH: NATURAL:= POW2(N_PAGES) + POW2(N_WORDS)+2;
signal page_address   : std_logic_vector (ADDR_WIDTH-1 downto POW2(N_WORDS)+2);
signal page_cs_i       : std_logic_vector (0 to N_PAGES-1);
signal page_wr_ack_i:  std_logic_vector (0 TO N_PAGES-1);
signal page_rd_ack_i:  std_logic_vector (0 TO N_PAGES-1);

begin
PAGE_WR_CE <= WR_CE;
PAGE_RD_CE <= RD_CE;
PAGE_TXDATA <= RXDATA;
page_address <= ADDR(ADDR_WIDTH-1 downto POW2(N_WORDS)+2);
-----------------------------------------------------------------------------------------------
PG_CS:process(page_address,CS)
-----------------------------------------------------------------------------------------------
variable index: natural;
begin
    index := to_integer(unsigned(page_address));
    page_cs_i<= (others=>'0');
    if (cs = '1') then
        page_cs_i(index)<= '1';
    end if;
end process PG_CS;
-----------------------------------------------------------------------------------------------
PAGE_SELECT <= page_cs_i;
PAGE_ADDR <= addr(POW2(N_WORDS)+2-1 downto 2);
-------------------------------------------------------------
MUX: entity MUX32
-------------------------------------------------------------
    GENERIC MAP(N_INPUTS=>N_PAGES)
    Port MAP( ADDR=> page_address,
       DIN=> PAGE_RXDATA,
       DOUT=>TXDATA);
-------------------------------------------------------------
RD_ACK <= page_rd_ack_i(0);
WR_ACK <= page_wr_ack_i(0);
----------------------------------------------------------
GEN_OR:FOR I IN 0 TO N_PAGES-2 GENERATE
----------------------------------------------------------
 BEGIN
    page_rd_ack_i(I) <= PAGE_RD_ACK(I) OR page_rd_ack_i(I+1);
    page_wr_ack_i(I) <= PAGE_WR_ACK(I) OR page_wr_ack_i(I+1);
 END GENERATE;
----------------------------------------------------------
page_rd_ack_i(N_PAGES-1)    <= PAGE_RD_ACK(N_PAGES-1);
page_wr_ack_i(N_PAGES-1)    <= PAGE_WR_ACK(N_PAGES-1);
----------------------------------------------------------
end Behavioral;
```

Appendix B

Clock Synchronizers

Fundamentally, all flip-flops with a clock signal have time constraints such as the setup time (t_{su}) and hold time (t_h), dictating that the input signal must remain stable in order to produce a stable output, after an expected time interval (t_{dq} in the case of D flip-flop). Figure B.1 describes, in the form of a time diagram, a satisfactory input for this flip-flop. Here, the input signal at D, remains stable for the time constraints and thus produce a stable output at Q after the t_{dq} interval.

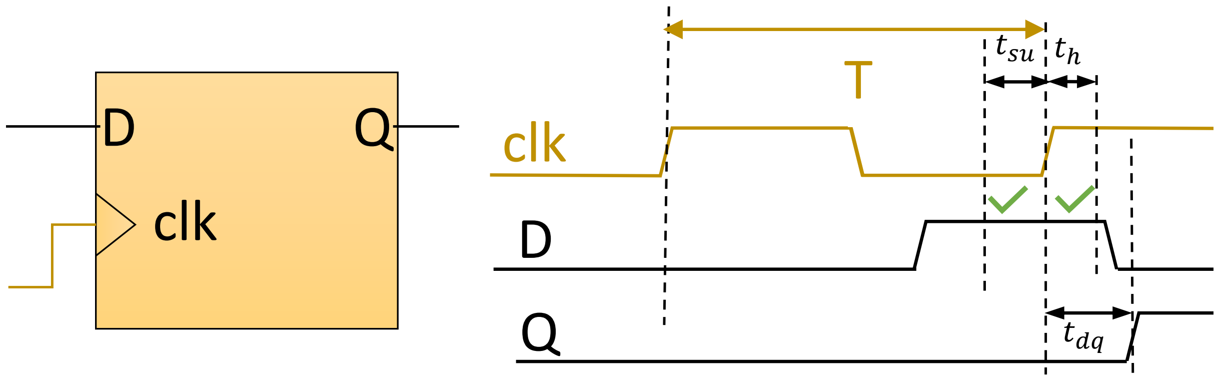


Figure B.1: Flip-Flop D time constraints setup, hold and propagation delay

The violation of time constraints produces failures in the signal capture, which can lead to: (1) the meta-stability of the flip-flop, or (2) the signal loss or data inconsistency when using vector signals. Figure B.2 describes in the form of a time diagram, an occurrence of meta-stability and two common examples of data loss or data inconsistency. In the temporal diagram on the left of the image, it is possible to observe that the input D violates the configuration constraint and this violation gave rise to an oscillating output. This condition is maintained for an undetermined time interval t_{osc}, also known as metastable signal time (t_{meta}). If propagated to the other logical units of a design, this state of oscillation may trigger undetermined control states or excessive current consumption in the design.

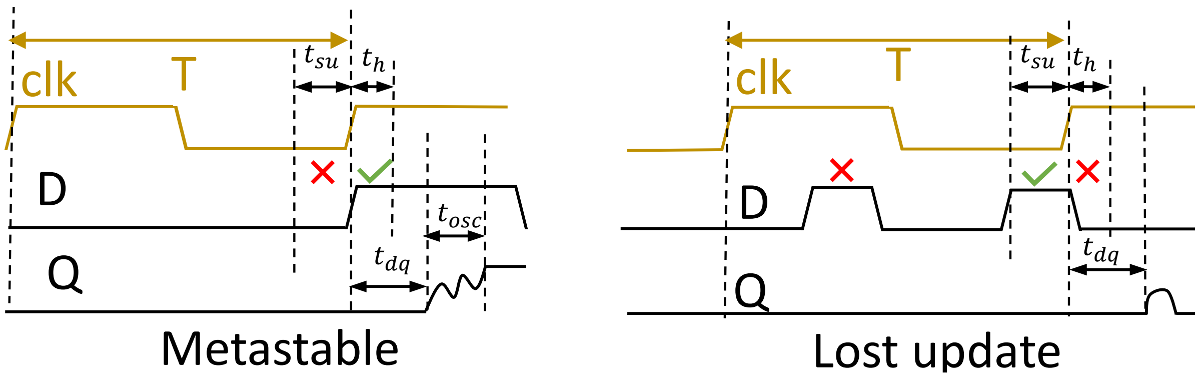


Figure B.2: Flip-Flop D time constraints violation

On the right side we can observe two phenomena of lost update, or lost data, in which the two events at input D result in loss either due to the occurrence between the active transitions of the clock, or because it has violated the hold time. In the latter case, the result of output Q evolves to the previous value and when this failure occurs in vector-type signals, different bits may have different t_{meta} durations and unpredictably evolve to the input or to their previous values. Altogether, these failures can become intermittent, difficult to debug or reproduce, which often only occurs on the final hardware, being pointed out as the second biggest cause of product respin.

The synchronizer variants found in the community can be distinguished in two types of functionality: (1) the conventional multi-flop synchronizer and (2) a toggle synchronizer variant. Figure B.3 describes the implementation of the conventional multi-flop synchronizer, where the transmitter and receiver are marked by the letters A and Y, and only the clock and reset signals from the receiver circuit are used.

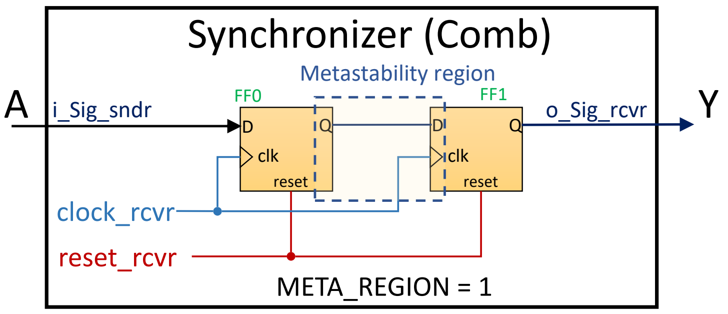


Figure B.3: Multi-Flop D synchronizer circuit

The input *i_Sig_sndr* receives the control signal sent from A to Y, and the output *o_Sig_rcvr* produces the signal captured by this component. The metastable state of the captured signal can occur at output Q of FF0, being contained by the input D of FF1 until the next active clock transition, time that it is allowed to stabilize. The probability of a meta-stable signal at the output of FF1 is very low or practically zero, but even so, for extreme high frequency scenarios, it is possible to scale the meta-stability region with more flip-flops using the *META_REGION* parameter.

For signals with the duration of a transmitter's clock pulse, we cannot use the conventional synchronizer for the reasons already mentioned in the lost data example. Instead, the pulse synchronizer (2) is applied since this circuit is not dependent on the duration of the event, but on the active transitions of the signal. Figure B.4 describes the implementation of this synchronizer where the clock and reset signals from both circuits are used.

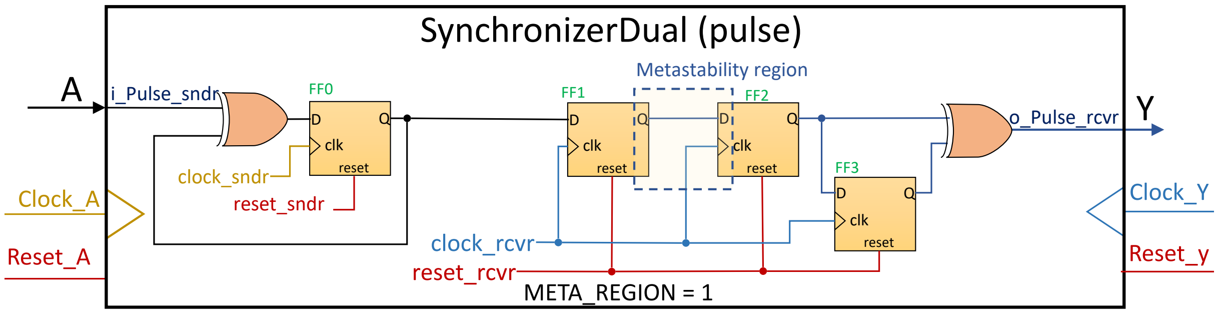


Figure B.4: Multi-Flop D pulse synchronizer circuit

The active transition of the pulse in the input sets the value of output Q in FF0 to its complementary value (toggle). This signal keeps the logic value stable until the next active transition of the input. Two Y clock cycles allow to deal with metastability using the conventional multi-flop scheme, which reproduces the stable value at the output of FF2. Then, a logical combination between this output and FF3 produces a high logic value at the output during a Y clock pulse.

It should be noted that this synchronization can only be applied to clock pulse signals, and in situations where the meta-region signals evolved to its previous value, this pulse can suffer an additional clock cycle Y delay. For signals active for more than one clock cycle, the design of the accelerator employs a decomposition in set and clear pulses, which occur in the upward and downward transitions of the original signal.

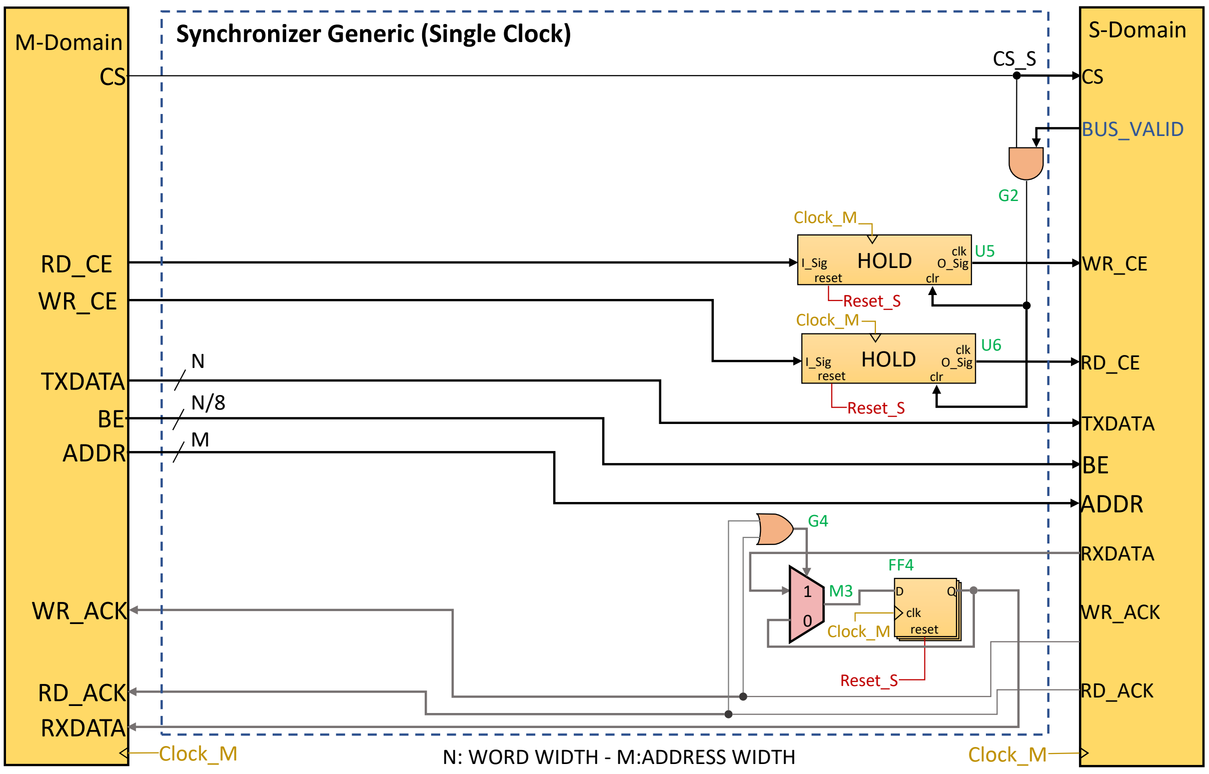


Figure B.5: Synchronizer Generic: single clock ACK-based handshake circuit.

Appendix C

Source Listings

Listing C.1: 'Encryptor' task 'run()' member specialization (Figure 2.7).

```
template<>
void hal_asos::Task<hal_asos::SwTask, TEncrypt>::run(void) {
  std::shared_ptr<dds::Publication> pLocal;
  std::shared_ptr<char[HLEN]> p_cyphered;
  std::shared_ptr< const char[]> p_plain;
  int pcount = 0;
  size_t ret = 1;
  uint8_t round = 0;

  set_cypher_key(key);
  key_expansion();

  while (this->StatusRunning && ret > 0) {
  ret = this->p_Subscription->take_publication(pLocal);
  if (ret) {
    pcount++;
    p_plain = pLocal->get_reference();
    p_cyphered = std::shared_ptr<char[HLEN]>(new char[HLEN]);
    std::copy_n(p_plain.get(),HLEN,p_cyphered.get());

    p_current_state = (state_t*)p_cyphered.get();
    add_round_key(0);
    for (round = 1; round < NROUNDS; ++round){
      subst_bytes();
      shift_rows();
      mix_columns();
      add_round_key(round);
    }
    subst_bytes();
    shift_rows();
    add_round_key(NROUNDS);
    ret = this->p_Topic->publish(p_cyphered);
    }
  }
  this->StatusRunning = false;
  this->p_Topic->close_topic();
  this->p_Subscription->close_subscription();
  LOG_MSG << this->TaskTag << "finished.(" << pcount << ")\n";
```

Listing C.2: 'File reader' task 'run()' member specialization(Figure 2.6).

```
template <>
void hal_asos::Task <hal_asos::SwTask, TFRead> ::run(void) {
  std::ifstream input_file;
  std::shared_ptr<char[HLEN]> p_buff;
  char* p_local_buff;
  int InputFileSize, Read_len, i, count = 0;
  long copy_len;
  input_file.open(target_file.c_str(), std::ios::in | std::ifstream::binary);
  if (!input_file.is_open()) {
    LOG_MSG << this->TaskTag << ":error opening input file!\n";
    this->shutdown_unconditional();
    return;
  }

  p_local_buff = new char[BLOCK_LEN];
  if (p_local_buff == nullptr) {
    input_file.close();
    LOG_MSG << this->TaskTag << ":failed memory allocation!\n";
    this->shutdown_unconditional();
    errno = ENOMEM;
    return;
  }

  input_file.seekg(0, std::ios::end);
  InputFileSize = (long)input_file.tellg();
  input_file.seekg(0, std::ios::beg);
  *((int*)(p_local_buff)) = InputFileSize;
  input_file.read(p_local_buff + sizeof(int), (BLOCK_LEN - sizeof(int)));
  Read_len = (int)input_file.gcount() + sizeof(int);

  while (this->StatusRunning && Read_len > 0) {
    for (i = 0; i < Read_len; i += HLEN) {
      p_buff = std::shared_ptr<char[HLEN]>(new char[HLEN]);
      copy_len = mmin(HLEN, (int)(Read_len - i));
      std::copy_n(p_local_buff + i, copy_len, p_buff.get());
      this->p_Topic->publish(p_buff);
      count++;
    }
    input_file.read(p_local_buff, BLOCK_LEN);
    Read_len = (long)input_file.gcount();
  }
  input_file.close();
  this->p_Topic->close_topic();
  LOG_MSG << this->TaskTag << "finished...(" << count << ")\n";
  delete[] p_local_buff;
}
```

Listing C.3: 'Uploader' task 'run()' member specialization (Figure 2.8).

```
template <>
void hal_asos::Task<hal_asos::SwTask, TUpload>::run(void) {
  using namespace hal_asos::networking;
  int ret = 1, count = 0, index=0;
  char* p_local_buff;
  std::shared_ptr<const char[]>p_buff;
  std::shared_ptr<dds::Publication> pLocal;
  CSocket<Client> Soc;

  p_local_buff = new char[BLOCK_LEN];
  if (p_local_buff == nullptr) {
    LOG_MSG << this->TaskTag << ":failed memory allocation\n";
    this->shutdown_unconditional();
    errno = ENOMEM;
    return;
  }

  Soc.set_ip_address(ip);
  Soc.set_sock_family(AF_INET);
  Soc.set_sock_type(SOCK_STREAM);
  Soc.set_sock_port(PORT_NO);


  this->StatusRunning = Soc.open_connection();
  while (this->StatusRunning && ret  > 0) {
    while (index < BLOCK_LEN && ret > 0) {
      ret = this->p_Subscription->take_publication(pLocal);
      if (ret) {
        p_buff = pLocal->get_reference();
        std::copy_n(p_buff.get(),pLocal->get_len(), p_local_buff + index);
        index += pLocal->get_len();
        count++;
      }
    }
    if (index > 0) {
      ret = Soc.safe_write(p_local_buff, index);
      index = 0;
     }
    }
  Soc.close_connection();
  this->p_Subscription->close_subscription();
  delete[] p_local_buff;
  LOG_MSG<<this->TaskTag<<"finished...("<<count<<")\n";
}
```

Listing C.4: Machine 1:Encryptor HW Task:(back to Figure 2.24).

```vhdl
FSM_CONTROL: process(task_state, s00_kernel_run, s00_kernel_rxdata, index_q, count_len_q,
target_len_q, resetn_i, kernel_response, p_cypher_i, total_len_q, p_current_q, aes_done_i,
p_cypher_i,kernel_call)
-------------------------------------------------------------------------------------------
begin
task_done_i     <= '0';
index_d         <= index_q;
count_len_d     <= count_len_q;
total_len_d     <= total_len_q;
target_len_d    <= target_len_q;
p_current_d     <= p_current_q;
trigger_aes_i   <= '0';
task_state_next<= task_state;
if resetn_i = '0' then
  index_d <= 0;
  total_len_d<=0;
  reset_sys_call(kernel_call);
else
  hal_asos_link_to_kernel(kernel_response,kernel_call);
  case task_state is
  when s0_ready=>
     if s00_kernel_run = '1' then
      task_state_next <= s1_transfer_from_dds;
     end if;
     total_len_d<=0;
  when s1_transfer_from_dds=>
     transfer_data_from_dds(kernel_call,kernel_response,0, C_BLOCK_LEN);
     target_len_d <= cast_return_to_transfer_len(kernel_response);
     index_d <= 0;
     count_len_d<=0;
     task_state_next <= s2_evaluate_transfer;
  when s2_evaluate_transfer=>
     task_state_next <= s8_write_message;
     if(target_len_q >0) then
      task_state_next <= s3_read_lram;
     end if;
  when s3_read_lram=>
     safe_read_lram_word32(kernel_call, kernel_response,p_current_d,(C_PLAIN_LEN/4),index_q);
     count_len_d <= count_len_q + C_PLAIN_LEN;
     task_state_next <= s4_trigger_aes;
  when s4_trigger_aes=>
     trigger_aes_i<= '1';
     task_state_next<= s5_wait_aes;
  when s5_wait_aes=>
     wait_signal_event(kernel_call, kernel_response,aes_done_i,Done_d);
     task_state_next <= s6_write_lram;
  when s6_write_lram=>
     safe_write_lram_word32(kernel_call, kernel_response,p_cypher_i,(C_PLAIN_LEN/4),index_q);
     index_d <= index_q + (C_PLAIN_LEN/4);
     total_len_d <= total_len_q + 1;
     task_state_next <= s7_transfer_to_dds;
     if(count_len_q < target_len_q) then
      task_state_next <= s3_read_lram;
     end if;
  when s7_transfer_to_dds=>
     transfer_data_to_dds(kernel_call, kernel_response,0, count_len_q);
     task_state_next<= s1_transfer_from_dds;
  when s8_write_message=>
     safe_write_lram(kernel_call, kernel_response, fmessage,
                    std_logic_vector(to_unsigned(total_len_q,32)),0);
     task_state_next <= s90_print_stdio;
  when s90_print_stdio=>
     write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
     task_state_next <= s99_exit;
  when s99_exit=>
     task_exit(kernel_call, kernel_response);
     task_state_next <=s99_exit;
  when others=> null;
  end case; end if; end process FSM_CONTROL;
```

Listing C.5: Machine 1:Encryptor HW Task datapath:(back to text 2.7.3).

```
---------------------------------------------------------------------------
plain_data_i        <= array_32_to_slv(p_current_d);
---------------------------------------------------------------------------
ip_aes: entity aes128
---------------------------------------------------------------------------
port map (clock             => clock,
            reset           => reset_i,
            i_run           => trigger_aes_i,
            i_sleep         => s00_kernel_sleep_task,
            i_key_expand    => initial_it,
            o_done          => aes_done_i,
            i_plain_data    => plain_data_i,
            i_cipher_key    => cipher_key_i,
            o_ciphered_data => ciphered_data_i);
---------------------------------------------------------------------------
p_cypher_i           <= slv_to_array_32(ciphered_data_i);
---------------------------------------------------------------------------
TASK_REGS:process(clock)
---------------------------------------------------------------------------
begin
  if rising_edge(clock) then

     if resetn_i = '0' then
        index_q       <= 0;
        total_len_q   <= 0;
        count_len_q   <= 0;
        target_len_q  <= 128;
        initial_it    <= '1';
        p_current_q<= (others=>(others=>'0'));
     else

      if kernel_response.sleep_task = '0' then
         p_current_q<= p_current_d;
      end if;

      if kernel_response.block_task = '0' then
        index_q <= index_d;
        total_len_q <= total_len_d;
        count_len_q <= count_len_d;
        target_len_q <= target_len_d;
      end if;

      if kernel_response.block_task = '0' and aes_done_i = '0' then
           initial_it <= '0';
      end if;

     end if;
  end if;
end process TASK_REGS;
---------------------------------------------------------------------------
```

Listing C.6: Encryptor SA - control path (1/2):(back to Figure 2.25).

```
-----------------------------------------------------------------------------------
FSM_CONTROL:process(task_state,s00_kernel_run,status_ret_q,file_len_d,buff_len_q,
 count_blocks_q, index_q, reset_i, kernel_response, ciphered_data_i, file_len_q, ifile_q,
done_i, tsocket_q, target_read, p_current_q,p_result_q,kernel_call)
-----------------------------------------------------------------------------------
begin
task_state_next  <= task_state;
inc_index        <= '0';
inc_count_blocks <='0';
clr_index        <= '0';
trigger_aes_i    <= '0';
status_ret_d     <= status_ret_q;
 p_current_d     <= p_current_q;
p_result_d       <= p_result_q;
ifile_d          <= ifile_q;
tsocket_d        <= tsocket_q;
file_len_d       <= file_len_q;
buff_len_d       <= buff_len_q;
task_done_i      <= '0';

if reset_i = '1' then
    reset_sys_call(kernel_call);
else
  hal_asos_link_to_kernel(kernel_response,kernel_call);
  case task_state is
  when s0_ready=>
    if s00_kernel_run = '1' then
      task_state_next <= s1_query_file;
    end if;
  when s1_query_file=>
    pooled_fstream_query(kernel_call,kernel_response,ifile_q, ifile_d);
    task_state_next <= s2_query_socket;
  when s2_query_socket=>
    pooled_socket_query(kernel_call,kernel_response,tsocket_q,tsocket_d);
    task_state_next<= s3_open_file;
  when s3_open_file=>
    pooled_fstream_open(kernel_call,kernel_response,ifile_q,ifile_d);
    status_ret_d <= cast_return_to_transaction_ret(kernel_response);
    task_state_next <= s4_evaluate_file;
    if(status_ret_q < 0)then
      task_state_next <= s17_write_string_lram;
    end if;
  when s4_evaluate_file=>
    pooled_fstream_read_len(kernel_call,kernel_response,ifile_q,file_len_d);
    p_current_d(0) <= std_logic_vector(to_unsigned(file_len_q,32));
    task_state_next <= s16_close_file;
    if(file_len_d >0) then
      task_state_next <= s5_set_word_len;
    end if;
  when s5_set_word_len=>
    safe_write_lram_word32(kernel_call, kernel_response, p_current_q,1,0);
    inc_index <= '1';
    task_state_next<= s6_open_socket;
  when s6_open_socket=>
    pooled_socket_open(kernel_call,kernel_response,tsocket_q,tsocket_d);
    status_ret_d <= cast_return_to_transaction_ret(kernel_response);
    task_state_next <= s7_read_file;
    if(status_ret_q < 0)then
      task_state_next <= s15_close_socket;
    end if
  when s7_read_file=>
    pooled_fstream_read_word32(kernel_call,kernel_response, ifile_q,target_read,index_q);
    buff_len_d <= cast_return_to_transfer_len(kernel_response);
    clr_index <= '1';
    task_state_next <= s8_evaluate_read;
  when s8_evaluate_read=>
    task_state_next <= s15_close_socket;
    if(buff_len_q > 0) then
      task_state_next <= s9_load_plain;
    end if;
  when s9_load_plain=>
    safe_read_lram_word32(kernel_call, kernel_response, p_current_d,4,index_q);
    task_state_next <= s10_trigger_encrypt;
  when s10_trigger_encrypt=>
    trigger_aes_i <='1';
    task_state_next <= s11_wait_done;
  when s11_wait_done=>
    wait_signal_event(kernel_call, kernel_response,done_i,event_done_d);
    p_result_d <= slv_to_array_32(ciphered_data_i);
    task_state_next <= s12_update_lram;
  when s12_update_lram=>
    inc_count_blocks<= '1';
    inc_index <= '1';
    safe_write_lram_word32(kernel_call, kernel_response, p_result_q,4,index_q);
    task_state_next <= s13_update_index;
  when s13_update_index=>
    task_state_next<=s14_write_socket;
    if(index_q < buff_len_q) then
      task_state_next <= s9_load_plain;
    end if;
  when s14_write_socket=>
    clr_index<= '1';
    pooled_socket_write_word32(kernel_call, kernel_response, tsocket_q,index_q,0);
    task_state_next <= s7_read_file;
  when s15_close_socket=>
    pooled_socket_close(kernel_call, kernel_response, tsocket_q,tsocket_d);
    task_state_next <= s16_close_file;
  when s16_close_file=>
    pooled_fstream_close(kernel_call, kernel_response, ifile_q,ifile_d);
    task_state_next<= s17_write_string_lram;
  when s17_write_string_lram=>
    safe_write_lram(kernel_call,kernel_response,fmessage,
             std_logic_vector(to_unsigned(count_blocks_q,32)),128);
    task_state_next <= s90_print_stdio;
  when s90_print_stdio=>
    write_stdio(kernel_call, kernel_response,fmessage'high,m_len,128);
    task_state_next <= s99_exit;
  when s99_exit=>
    task_done_i <= '1';
    task_exit(kernel_call, kernel_response);
    task_state_next <=s99_exit;
  when others=> null;
    end case;
    end if;
end process FSM_CONTROL;
--------------------------------------------------------------------------------
```

Listing C.7: Encryptor SA - control path (2/2):(back to Figure 2.25).

Listing C.8: Encryptor SA - synchronous control path:(Figure 2.25).

```
target_read <= INIT_WBLOCK_LEN when initial_it = '1' else WBLOCK_LEN;
increment_val <= 1 when initial_index else 4;
---------------------------------------------------------------------------
process(index_q,increment_val)
---------------------------------------------------------------------------
begin
  next_index <= index_q + increment_val;
end process;
---------------------------------------------------------------------------
FSM_SYNC:process(clock)
---------------------------------------------------------------------------
begin
  if rising_edge(clock) then

     if reset_i = '1' then
      p_current_q   <= (others=>(others=>'0'));
      p_result_q    <= (others=>(others=>'0'));
      file_len_q    <= 0;
      event_done_q  <= false;

      buff_len_q    <= 0;
      status_ret_q  <=0;
      ifile_q       <=(fstream_obj,(others=>'0'),0,false);
      tsocket_q     <=(net_obj,(others=>'0'),1,false);
      task_state    <= s0_ready;

     else
             if kernel_response.sleep_task = '0' then
                 p_current_q  <= p_current_d;
                 p_result_q   <= p_result_d;
                 file_len_q   <= file_len_d;
                 event_done_q <= event_done_d;
             end if;

             if kernel_response.block_task = '0' then
                task_state    <= s0_ready;
                buff_len_q    <= buff_len_d;
                status_ret_q  <=status_ret_d;
                ifile_q       <= ifile_d;
                tsocket_q     <= tsocket_d;
             end if;
     end if;
  end if;
end process FSM_SYNC;
---------------------------------------------------------------------------
```

Listing C.9: Encryptor SA - synchronous datapath:(Figure 2.25).

```
--------------------------------------------------------------------------
plain_data_i <= array_32_to_slv(p_current_q);
--------------------------------------------------------------------------
ip_aes: entity aes128
--------------------------------------------------------------------------
  port map (clock      => clock,
         reset         =>  reset_i,
         i_run         => trigger_aes_i,
         i_sleep       => kernel_response.sleep_task,
         i_key_expand  => initial_it,
         o_done        => done_i,
         i_plain_data  => plain_data_i,
         i_cipher_key  => ciper_key_i,
         o_ciphered_data => ciphered_data_i);
------------------------------------------------------------------------
p_result_d  <= slv_to_array_32(ciphered_data_i);
------------------------------------------------------------------------
TASK_DPATH: process(clock)
------------------------------------------------------------------------
begin
  if rising_edge(clock) then
     if reset_i = '1' then
         initial_it      <= '1';
         initial_index   <= true;
         count_blocks_q  <= 0;
     else
       if clr_index = '1' and kernel_response.block_task= '0' then
         initial_index   <= false;
       end if;

       if done_i = '0' and kernel_response.sleep_task = '0' then
         initial_it      <= '0';
       end if;

       if inc_count_blocks = '1' and kernel_response.block_task = '0' then
         count_blocks_q  <= count_blocks_q + 1;
       end if;

     end if;
  end if;
end process TASK_DPATH;
--------------------------------------------------------------------------
counter_index: process(clock)
--------------------------------------------------------------------------
begin
  if rising_edge(clock)  then
     if (clr_index = '1' and kernel_response.block_task = '0') or reset_i = '1'  then
        index_q    <= 0;
     elsif inc_index = '1' and kernel_response.block_task = '0'  then
        index_q    <= next_index;
     end if;
  end if;
end process counter_index;
--------------------------------------------------------------------------
end encrypter_sa_level;
```

Listing C.10: Co-Simulation Encryptor - SystemVerilog test-bench file:(sec. 2.7.5).

```
`timescale 1ns / 1ps

module tb_design();

    logic interrupt_pin_0;
    logic o_heart_bit_0;
    logic s00_axi_aclk_0=0;
    logic s00_axi_aresetn_0=0;

    always #5ns s00_axi_aclk_0=~s00_axi_aclk_0;

 design_1_wrapper DUT(
    .s00_axi_aclk_0(s00_axi_aclk_0),
    .s00_axi_aresetn_0(s00_axi_aresetn_0),
    .o_heart_bit_0(o_heart_bit_0),
    .interrupt_pin_0(interrupt_pin_0));

//--------------------------------------------------------------
  initial
//--------------------------------------------------------------
    begin
        s00_axi_aresetn_0=0;
        #100ns;
        s00_axi_aresetn_0=1;
        #200ns;
        DUT.design_1_i.hal_asos_accelerator_0.u0.ubus.start_proxy();
        #100ns;
   end
//-------------------
endmodule
```

Listing C.11: Full Simulation - SystemVerilog testbench for QEMU.

```
`timescale 1ns / 1ps

import hal_asos_sv_pkg::*;
module tb_qemu( );

     const int C_BASE_ADDRESS = 32'h043C00000;
    AcceleratorInfo acc0 = { "HwEncrypter0" ,C_BASE_ADDRESS,32'h20000,0};
    AcceleratorInfo acc1 = { "HwEncrypterSA0" ,C_BASE_ADDRESS + 32'h20000,32'h20000,1};
    simulation_status_t status= DEAD_SIM;
    logic clock = 0;
    logic resetn = 0;

    always #5ns clock=~clock;

 //------------------------------------------------
    task timestamp(output int x);
 //------------------------------------------------
        x = $time;
    endtask;

 //------------------------------------------------
    task delay(input int x);
 //------------------------------------------------
        int count = x;
        while(count > 0)begin
            #10 count--;
        end
    endtask;
//------------------------------------------------
    export "DPI-C" task delay;
    export "DPI-C" task timestamp;

    design_2_wrapper DUT(
       .m00_axi_aclk_0(clock),
       .m00_axi_aresetn_0(resetn));
//--------------------------------------------------------------
  initial
//--------------------------------------------------------------
    begin
        resetn=0;
        #200ns;
        resetn=1;
        #50ns
        DUT.design_2_i.hal_asos_link_cv_0.inst.u0.register_accelerator(acc0);
        #10ns
        DUT.design_2_i.hal_asos_link_cv_0.inst.u0.register_accelerator(acc1);
        #10ns
        DUT.design_2_i.hal_asos_link_cv_0.inst.u0.start_simulation(status);

//         #20ns;
//DUT.design_1_i.Host_0.inst.u0.read_bus_w32(C_BASE_ADDRESS +
MQUEUE_IN_SPACE_REG_OFFSET,data);
     ;

   end
endmodule
```

Listing C.12: Machine 1 Application - receive parameters from command line.

```cpp
#include <iostream>
#include "aes_128_client_code/client_machine.h"

extern std::string target_file;
int main(int argc, char** argv){

  char select = -1;
  if (argc > 2) {
    select = *argv[1];
    target_file = std::string(argv[2]);
  }

  switch (select) {
  case '0':
    hal_asos_demo::test_aes128_file_sw_threads();
    break;
  case '1':
    hal_asos_demo::test_aes128_file_hw_thread_cypher_3();
    break;
  case '2':
    hal_asos_demo::test_aes128_file_hw_thread_cypher_sa();
    break;
  case '3':
    hal_asos_demo::test_aes128_file_hw_thread_cypher_3_user_io();
    break;
  case '4':
    hal_asos_demo::test_aes128_file_hw_thread_cypher_sa_uio();
    break;
  default:
    break;
  }
  return 0;
}
```

Listing C.13: Machine 1 Application - HW *Encryptor* software refactoring.

```cpp
void hal_asos_demo::test_aes128_file_hw_thread_cypher_3(void) {
  using namespace hal_asos;

  Task<SwTask, TFRead> T0;
  Task<HwTask, THwEncrypter, semantic<dds::Restricted>>  T1;
  Task<SwTask, TUpload> T2;

  T0.start();
  T1.start();
  T2.start();

  T0.join();
  T1.join();
  T2.join();
}
```

Listing C.14: Machine 1 Application - HW *Encryptor* SA software refactoring.

```
void hal_asos_demo::test_aes128_file_hw_thread_cypher_sa(void) {
  using namespace hal_asos;

  Task<HwTask, THwEncrypterSA>  T1;
  hal_asos::networking::CSocket<hal_asos::networking::Client> Soc;

  CFstream<std::ifstream> Input_file(target_file.c_str());
  Input_file.set_flags(std::ios::in | std::ifstream::binary);

  Input_file.open_file();
  Input_file.get_file_len();


  Soc.set_ip_address(ip);
  Soc.set_sock_family(AF_INET);
  Soc.set_sock_type(SOCK_STREAM);
  Soc.set_sock_port(PORT_NO);

  T1.submit_to_pool(Input_file);
  T1.submit_to_pool(Soc);

  T1.start();
  T1.join();
}
```

Listing C.15: Kernel Package - LBUS register and LRAM write procedures.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library hal_asos_v4_00_a;
use hal_asos_v4_00_a.hal_asos_configs_pkg.all;


package hal_kernel_pkg is
...

-----------------------------------------------------------------------------------------
procedure lbus_write_reg (signal i_call : out kernel_input_t;
                          signal o_response: in kernel_output_t;
                          reg_offset: in natural range 0 to (2**CLBUS_REG_WIDTH)-1;
                          data: in std_logic_vector(31 downto 0)) is
-----------------------------------------------------------------------------------------
begin
      i_call.sys_call_id <= SYS_CALL_WRITE_LBUS;
      i_call.this_call <= '1';
      i_call.parameters(PARAM_LBUS_WORD'RANGE)<=data;
      i_call.parameters(PARAM_LBUS_BE'RANGE) <= "1111";
      i_call.parameters(PARAM_LBUS_ADDR'RANGE)
                                <=std_logic_vector(to_unsigned(reg_offset,CLBUS_AWWIDTH));
end procedure lbus_write_reg;
-----------------------------------------------------------------------------------------
...
-----------------------------------------------------------------------------------------
procedure lram_write_word (signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1;
                           data: in std_logic_vector(31 downto 0))is
-----------------------------------------------------------------------------------------
begin
       i_call.sys_call_id <= SYS_CALL_WRITE_LBUS;
       i_call.this_call <= '1';
       i_call.parameters(PARAM_LBUS_BURTS_LEN'RANGE)
                         <=std_logic_vector(to_unsigned(1, PARAM_LBUS_BURTS_LEN'length));
       i_call.parameters(PARAM_LBUS_WORD'RANGE)<=data;
       i_call.parameters(PARAM_LBUS_BE'RANGE) <= "1111";
       i_call.parameters(PARAM_LBUS_ADDR'RANGE)
                      <= '1'& std_logic_vector(to_unsigned(lram_address,CLRAM_AWWIDTH));

end procedure lram_write_word;
-----------------------------------------------------------------------------------------
```

=> back to Figure 4.2

Listing C.16: Microprogram - VHDL description for the test input MUX.

```vhdl
------------------------------------------------------------------------------------------
--INPUT_MUX_SEL
------------------------------------------------------------------------------------------
CONSTANT LEV_EVM_READY_SEL         :natural:=0;--00000
CONSTANT LEV_EVM_SIGNAL_SEL        :natural:=1;--00001
CONSTANT LFIFO_DATA_READY_SEL      :natural:=2;--00010
CONSTANT LFIFO_DATA_VALID_SEL      :natural:=3;--00011
CONSTANT LFIFO_READY_FOR_DATA_SEL  :natural:=4;--00100
CONSTANT LFIFO_ACK_DATA_SEL        :natural:=5;--00101
CONSTANT MQ_LINK_DATA_READY_SEL    :natural:=6;--00110
CONSTANT MQ_LINK_DATA_VALID_SEL    :natural:=7;--00111
CONSTANT MQ_LINK_READY_FOR_DATA_SEL:natural:=8;--01000
CONSTANT MQ_LINK_ACK_DATA_SEL      :natural:=9;--01001
CONSTANT MUTEX_STATUS_B_LOCKED_SEL :natural:=10;--01010
CONSTANT MUTEX_STATUS_B_LOCKED_NOT_SEL:natural:=11;--01011
CONSTANT MUTEX_STATUS_A_LOCKED_NOT_SEL:natural:=12;--01100
CONSTANT LBUS_WR_ACK_SEL           :natural:=13;--01101
CONSTANT LBUS_RD_ACK_SEL           :natural:=14;--01110
CONSTANT MBUS2_KERNEL_CMD_ACK_SEL   :natural:=15;--01111
CONSTANT MBUS2_KERNEL_CMD_CMPLT_SEL :natural:=16;--10000
CONSTANT SYSRAM_ADDRESS_OK_SEL     :natural:=17;--10001
CONSTANT BURST_MODE_ACTIVE_SEL     :natural:=18;--10010
CONSTANT BURST_DONE_SEL            :natural:=19;--10011
CONSTANT MBUS2_KERNEL_CMD_BURST_COMPLT:natural:=20;--10100
CONSTANT MBUS2_KERNEL_CMD_BURST_DONE :natural:=21;--10101
CONSTANT MBUS2_KERNEL_CMD_BURST_READY:natural:=22;--10110
--
CONSTANT FALSE_SEL                  :natural:=2**(input_sel_t'LENGTH)-2;--(32-2)--11110
CONSTANT TRUE_SEL                   :natural:=2**(input_sel_t'LENGTH)-1;--(32-1)--11111
-----------------------------------------------------------------------------
---------------------------------------------------
INPUT_MUX: process( input_sel, i_time_event_ready,i_time_event_signal,
...
i_mbus2_kernel_burst_done, i_mbus2_kernel_burst_mode)
---------------------------------------------------
begin
inc_load_not <= '0';
  case to_integer(unsigned(input_sel)) is
     when LEV_EVM_READY_SEL        =>inc_load_not <= i_time_event_ready;
     when LEV_EVM_SIGNAL_SEL       =>inc_load_not <= i_time_event_signal;
     when LFIFO_DATA_READY_SEL      =>inc_load_not <=i_lfifo_data_ready;
     when LFIFO_DATA_VALID_SEL      =>inc_load_not <=i_lfifo_data_valid;
     when LFIFO_READY_FOR_DATA_SEL  =>inc_load_not <=i_lfifo_ready_for_data;
     when LFIFO_ACK_DATA_SEL        =>inc_load_not <=i_lfifo_ack_data;
     when MQ_LINK_DATA_READY_SEL    =>inc_load_not <=i_msg_queue_data_ready;
     when MQ_LINK_DATA_VALID_SEL    =>inc_load_not <=i_msg_queue_data_valid;
     when MQ_LINK_READY_FOR_DATA_SEL=>inc_load_not<=i_msg_queue_ready_for_data;
     when MQ_LINK_ACK_DATA_SEL      =>inc_load_not <=i_msg_queue_ack_data;
     when MUTEX_STATUS_B_LOCKED_SEL =>inc_load_not <=mutex_chb_locked;
     when MUTEX_STATUS_B_LOCKED_NOT_SEL=>inc_load_not <= mutex_chb_locked_not;
     when MUTEX_STATUS_A_LOCKED_NOT_SEL=>inc_load_not <= mutex_cha_locked_not;
     when LBUS_WR_ACK_SEL              =>inc_load_not <=i_lbus_wr_ack;
     when LBUS_RD_ACK_SEL              =>inc_load_not <=i_lbus_rd_ack;
     when MBUS2_KERNEL_CMD_ACK_SEL      =>inc_load_not <=i_mbus2_kernel_cmd_ack;
     when MBUS2_KERNEL_CMD_CMPLT_SEL    =>inc_load_not <=i_mbus2_kernel_cmd_cmplt;
     when SYSRAM_ADDRESS_OK_SEL        =>inc_load_not <=i_sysram_buffer_address_ok
     when BURST_MODE_ACTIVE_SEL        => inc_load_not<=i_burst_mode;
     when MBUS2_KERNEL_CMD_BURST_COMPLT => inc_load_not <=i_mbus2_kernel_burst_complt;
     when MBUS2_KERNEL_CMD_BURST_DONE   => inc_load_not <=i_mbus2_kernel_burst_done;
     when MBUS2_KERNEL_CMD_BURST_READY  =>
                  inc_load_not <= i_mbus2_kernel_burst_mode AND i_sysram_buffer_address_ok;
  --
     when BURST_DONE_SEL               =>INC_LOAD_NOT <= i_burst_done;
     when FALSE_SEL                    =>inc_load_not <='0';
     when TRUE_SEL                     =>inc_load_not <='1';
     when others                       => null;
  end case;
end process INPUT_MUX;
---------------------------------------------------
```

Listing C.17: Microprogram - VHDL description for the outputs DEMUX..

```
----------------------------------------------------------------------
-- OUTPUT MUL SET
----------------------------------------------------------------------
CONSTANT NO_OUPTUT      :  natural:=0;
CONSTANT LEVM_TRIGGER   :  natural:=1;
CONSTANT POP_DATA       :  natural:=2;
CONSTANT PUSH_DATA      :  natural:=3;
CONSTANT POP_MQUEUE     :  natural:=4;
CONSTANT PUSH_MQUEUE    :  natural:=5;
CONSTANT LBUS_WRITE     :  natural:=6;
CONSTANT LBUS_READ      :  natural:=7;
CONSTANT MBUS_WRITE     :  natural:=8;
CONSTANT MBUS_READ      :  natural:=9;
----------------------------------------------------
...
----------------------------------------------------
out_puts:process(output_sel)
----------------------------------------------------
variable select_out: natural;
begin

select_out :=  to_integer(unsigned(output_sel));
o_levm_trigger_evm<='0';
o_lfifo_pop_data<='0';
o_lfifo_push_data<='0';
o_msg_queue_pop_data<='0';
o_msg_queue_push_data<='0';
o_lbus_wr<='0';
o_lbus_rd<='0';
o_kernel2_mbus_wr_req<='0';
o_kernel2_mbus_rd_req<='0';
case select_out is
    when LEVM_TRIGGER   =>    o_levm_trigger_evm    <= '1';
    when POP_DATA       =>    o_lfifo_pop_data      <= '1';
    when PUSH_DATA      =>    o_lfifo_push_data     <= '1';
    when POP_MQUEUE     =>    o_msg_queue_pop_data  <= '1';
    when PUSH_MQUEUE    =>    o_msg_queue_push_data <= '1';
    when LBUS_WRITE     =>    o_lbus_wr             <= '1';
    when LBUS_READ      =>    o_lbus_rd             <= '1';
    when MBUS_WRITE     =>    o_kernel2_mbus_wr_req <= '1';
    when MBUS_READ      =>    o_kernel2_mbus_rd_req <= '1';
    when others         =>    NULL;
 end case;
end process out_puts;
----------------------------------------------------
```

back to Figure 3.12

Listing C.18: Microprogram - VHDL description for program in ROM.

```
architecture Behavioral of control_logic is
...
-----------------------
-- kernel mcode
-----------------------
signal program: rom :=(
0=>"1111000000010",
1=>"1111000000000",
2=>"1111000000000",
3=>"1111000000000",
4=>"0000000000001",
5=>"0000101000101",
6=>"1111000000010",
7=>"1111000000010",
8=>"0001000000001",
9=>"0001101001001",
10=>"1111000000010",
11=>"1111000000010",
12=>"0010000000001",
13=>"0010101001101",
14=>"1111000000010",
15=>"1111000000010",
16=>"0011000000001",
17=>"0011101010001",
18=>"1111000000010",
19=>"1111000000010",
20=>"0100000000001",
21=>"0100101010101",
22=>"1111000000010",
23=>"1111000000010",
24=>"0111101011101",
25=>"1111000000010",
26=>"1111100000010",
27=>"1111100000010",
28=>"1111101011001",
29=>"1111000000010",
30=>"1111100000010",
31=>"1111100000010",
32=>"0110000011101",
33=>"1111110011001",
34=>"0101000011101",
35=>"1111000000010",
36=>"0110011000001",
37=>"1111110011001",
38=>"0101011011101",
39=>"1111000000010",
40=>"0101010011101",
41=>"0110101011001",
42=>"1111000000010",
43=>"1111000000010",
44=>"1000111000001",
45=>"0111101100101",
46=>"1000010000001",
47=>"1111000000010",
48=>"1000111000001",
49=>"0111101100001",
50=>"1000010000001",
51=>"1111000000010",
52=>"1001010011101",
53=>"1001101011101",
54=>"1111000000010",
55=>"1111000000010",
56=>"1001010011001",
57=>"1001101011001",
58=>"1111000000010",
59=>"1111000000010",
60=>"1011011000001",
61=>"0111101100101",
62=>"1010110000001",
63=>"1111000000010",
64=>"1011011000001",
65=>"0111101100001",
66=>"1010110000001",
67=>"1111000000010",
68=>"1111000000001",
69=>"1111000000001",
70=>"1111000000001",
71=>"1111000000001");
---------------------
...
this_call_i <= (call_d1 or i_this_call) and not(i_sleep);
load_i<= not(inc_load_not);
----------------------------------------------------

uc: entity counter_load
----------------------------------------------------
    generic map( COUNT_WIDTH => C_KERNEL_PROGRESS_WIDTH)
    Port map( i_clock => clock,
           i_clear => reset,
           i_load => load_i,
           i_increment => inc_load_not,
           i_enable => this_call_i,
           i_val => to_integer(unsigned(dout(load_addr_t'range))),
           o_count => temp_control_progress);

control_progress_i <= to_unsigned(temp_control_progress,control_progress_i'length);
----------------------------------------------------
-- ROM_ADDRESSING
----------------------------------------------------
PC:process(i_sys_call_id, control_progress_i)
begin
  program_counter(CSYS_CALL_LEN+1 downto 2)
                           <=to_unsigned(sys_call_t'pos(i_sys_call_id),CSYS_CALL_LEN);
  program_counter(1 downto 0) <= unsigned(control_progress_i);
end process;
----------------------------------------------------
LOCAL_ROM:process(program_counter,program)
----------------------------------------------------
begin
  dout <= program(to_integer(program_counter));
end process LOCAL_ROM;
----------------------------------------------------
...
```

back to Table 3.2 or back to Figure 3.13.

Listing C.19: Local-BUS - System call configuration descriptions.

```
package hal_asos_configs_pkg is
...
-------------------------------------------------------------------------------------------
--lbus
-------------------------------------------------------------------------------------------
constant C_LBUS_DATA_WIDTH   : natural := C_MACHINE_WIDTH;--32
constant C_LBUS_AWIDTH       : natural := 20; --512KB + 512KB
constant C_LBUS_PAGE_SIZE    : natural := (16*C_MACHINE_WIDTH/8);--16 WORDS
constant C_LBUS_PAGE_WIDTH   : natural := POW2(C_LBUS_PAGE_SIZE);--7
constant C_LBUS_REG_WIDTH    : natural := C_LBUS_PAGE_WIDTH+1;--[P0:P1]
constant C_LBUS_BE_WIDTH     : natural := C_MACHINE_WIDTH/8;
constant C_LBUS_AWWIDTH      : natural := C_LBUS_AWIDTH-POW2(C_LBUS_BE_WIDTH); --18 BITS
constant C_LBUS_BUSRT_WIDTH  : natural
                       := C_MACHINE_WIDTH - C_LBUS_BE_WIDTH - C_LBUS_AWWIDTH; --10bits(1kW)
--------------------------------------------------------------------------------------------
--mbus
--------------------------------------------------------------------------------------------
constant C_PAGE_SIZE         : natural := 4096;
constant C_PAGE_SHIFT        : natural := POW2(C_PAGE_SIZE);--12
constant C_MBUS_DATA_WIDTH   : natural := C_HOST_ARCH;--32
constant C_MBUS_BE_WIDTH     : natural := C_HOST_ARCH/8; --4
constant C_MBUS_OFFSET_WIDTH : natural := 20;
constant C_MBUS_OFFSET_WWIDTH: natural := C_MBUS_OFFSET_WIDTH-POW2(C_HOST_ARCH/8);--18
constant C_MBUS_BUSRT_WIDTH  : natural
                                := C_MACHINE_WIDTH-C_MBUS_BE_WIDTH-C_MBUS_OFFSET_WWIDTH;
...
package hal_kernel_pkg is
...
subtype PARAM_LBUS_WORD is std_logic_vector(C_MACHINE_WIDTH-1 downto 0);--[31:0]
subtype PARAM_LBUS_BE is
std_logic_vector(PARAM_LBUS_WORD'high+C_LBUS_BE_WIDTHdowntoPARAM_LBUS_WORD'high+1);--[35:32]
subtype PARAM_LBUS_OFFSET is
   std_logic_vector(PARAM_LBUS_BE'high+C_LBUS_AWWIDTH downto PARAM_LBUS_BE'high+1);--[53:36]
subtype PARAM_LBUS_BURST_LEN is
   std_logic_vector(C_MESSAGE_WIDTH-1 downto PARAM_LBUS_ADDR'high+1);--[63:54]

subtype PARAM_MBUS_WORD is std_logic_vector(C_MACHINE_WIDTH-1 downto 0);--[31:0]
subtype PARAM_MBUS_BE is
 std_logic_vector(PARAM_MBUS_WORD'HIGH+C_MBUS_BE_WIDTH downto PARAM_MBUS_WORD'HIGH+1);
                                                                                 --[35:32]
subtype PARAM_MBUS_OFFSET is
  std_logic_vector(PARAM_MBUS_BE'HIGH+C_MBUS_OFFSET_WWIDTH downto PARAM_MBUS_BE'HIGH+1);
                                                                                  --[53:36]
subtype PARAM_MBUS_PAGE_OFFSET is
  std_logic_vector(PARAM_MBUS_BE'HIGH+C_PAGE_SHIFT-2 downto PARAM_MBUS_BE'HIGH+1);--[45:36]
subtype PARAM_MBUS_PAGE_PREFIX is
   std_logic_vector(PARAM_MBUS_OFFSET'HIGH downto PARAM_MBUS_PAGE_OFFSET'HIGH+1); --[53:46]
subtype PARAM_MBUS_BURST_LEN is
  std_logic_vector(C_MESSAGE_WIDTH-1 downto PARAM_MBUS_OFFSET'HIGH+1); --[63:54]
...
```

back to Figure C.19.

Listing C.20: Kernel Package - import and export Kernel Core interfaces.

```vhdl
...
package body hal_kernel_pkg is
--------------------------------------------------------------------------------------------
procedure import_kernel_call(
               signal sys_call_id: in std_logic_vector(CSYS_CALL_LEN-1 downto 0);
               signal parameters:  in std_logic_vector ((C_MACHINE_WIDTH*2)-1 downto 0);
               signal this_call:   in std_logic;
               signal enable_scheduler:  in std_logic;
               signal reschedule:  in std_logic;
               signal enable_index:in std_logic;
               signal increment_index:  in std_logic;
               signal task_state:  in std_logic_vector(23 downto 0);
               signal task_done:   in std_ulogic;
               signal kernel_call: out kernel_input_t) is
--------------------------------------------------------------------------------------------
begin

 kernel_call.sys_call_id            <= sys_call_values(to_integer(unsigned(sys_call_id)));
 kernel_call.parameters             <= parameters ;
 kernel_call.this_call              <= this_call;
 kernel_call.enable_scheduler       <= enable_scheduler;
 kernel_call.reschedule             <= reschedule ;
 kernel_call.enable_index           <= enable_index;
 kernel_call.increment_index        <= increment_index;
 kernel_call.task_state             <= task_state;
 kernel_call.task_done              <= task_done;

end procedure import_kernel_call;
--------------------------------------------------------------------------------------------
...
---------------------------------------------------------------------------------------------
procedure export_kernel_response( signal i_system_response: in kernel_output_t;
       signal sys_call_id : out std_logic_vector(CSYS_CALL_LEN-1 downto 0);
       signal return_parameter: out std_logic_vector ((C_MACHINE_WIDTH*2)-1 downto 0);
       signal valid: out std_logic;
       signal syscall_progress:  out std_logic_vector(C_KERNEL_PROGRESS_WIDTH-1 downto 0);
       signal sched_progress:  out std_logic_vector(C_SCHED_PROGRESS_WIDTH-1 downto 0);
       signal index: out std_logic_vector(C_KERNEL_INDEX_WIDTH-1 downto 0);
       signal index_d1: out std_logic_vector(C_KERNEL_INDEX_WIDTH-1 downto 0);
       signal block_task: out  std_ulogic;
       signal sleep_task: out  std_ulogic;
       signal error_flag:  out std_ulogic;
       signal task_reset: out std_ulogic;
       signal task_run: out std_ulogic ) is
---------------------------------------------------------------------------------------------
begin
sys_call_id  <=std_logic_vector(to_unsigned(
                 sys_call_t'pos(i_system_response.sys_call_id),CSYS_CALL_LEN));
return_parameter <= i_system_response.return_arg;
valid            <= i_system_response.valid;
syscall_progress <= std_logic_vector(
                    to_unsigned(i_system_response.kernel_progress,C_KERNEL_PROGRESS_WIDTH));
sched_progress <= std_logic_vector(
                    to_unsigned(i_system_response.sched_progress,C_SCHED_PROGRESS_WIDTH));
index          <= std_logic_vector(
                  to_unsigned(i_system_response.index,C_KERNEL_INDEX_WIDTH));
index_d1       <= std_logic_vector(
                  to_unsigned(i_system_response.index_d1,C_KERNEL_INDEX_WIDTH));
block_task          <= i_system_response.block_task;
sleep_task          <= i_system_response.sleep_task;
error_flag          <= i_system_response.error_flag;
task_reset          <= i_system_response.task_reset;
task_run            <= i_system_response.task_run;

end procedure export_kernel_response;
--------------------------------------------------------------------------------------------
```

Listing C.21: Kernel Package - VHDL procedures declaration (part 1/3).

```
package hal_kernel_pkg is
...
procedure reset_record_event_in (signal obj: inout leventm_inputs_t);

procedure raise_interrupt( signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           interrupt_number: integer range 1 to 25);

procedure condition_from_hw_event( signal i_event: in std_logic;
                                   signal condition_id: in std_logic_vector;
                                   signal obj: out t_condition);

procedure condition_wait( signal i_call : out kernel_input_t;
                          signal o_response: in kernel_output_t;
                          signal sucess: out boolean;
                          constant task_id: in std_logic_vector;
                          signal obj: inout t_condition);

procedure condition_wait_for( signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal sucess: inout boolean;
                              constant task_id: in std_logic_vector;
                              signal obj: inout t_condition;
                              timeout_val: integer:=2**C_EVENT_TIMEOUT_WIDTH -1);

procedure wait_event_timeout( signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              timeout_val: integer :=2**C_EVENT_TIMEOUT_WIDTH -1);

procedure wait_signal_event (  signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal i_event:  in std_logic;
                              signal is_event: out boolean;
                              constant timeout_val: in integer :=0);

procedure control_from_word( signal word: in std_logic_vector(31 downto 0);
                             signal control: out control_register_t);

procedure word_from_control( signal control: in control_register_t;
                             signal word: out std_logic_vector(31 downto 0));

procedure status_from_word( signal word: in std_logic_vector(31 downto 0);
                             signal status: out status_register_t);

procedure word_from_status( signal status: in status_register_t;
                            signal word: out std_logic_vector(31 downto 0));

procedure lfifo_pop_data ( signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t);

procedure lfifo_push_data ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t);

procedure receive_message ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t);

procedure receive_message ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            msg: out message_t);

procedure receive_message ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            msg: out kremote_call_m);

procedure send_message ( signal i_call : out kernel_input_t;
                         signal o_response: in kernel_output_t;
                         msg: in message_t);
```

Listing C.22: Kernel Package - VHDL procedures declaration (part 2/3).

```
procedure send_message ( signal i_call : out kernel_input_t;
                         signal o_response: in kernel_output_t;
                         tmsg: in kremote_call_m);

procedure send_message ( signal i_call : out kernel_input_t;
                         signal o_response: in kernel_output_t;
                         rmsg: in kpool_query_m);

procedure send_message ( signal i_call : out kernel_input_t;
                         signal o_response: in kernel_output_t;
                         dmsg: in kdata_transfer_m);

procedure mutex_lock ( signal i_call : out kernel_input_t;
                       signal o_response: in kernel_output_t;
                       mutex_addr: in natural range 0 to (2**C_LBUS_AWWIDTH)-1);

procedure mutex_try_lock ( signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           mutex_addr: in  natural range 0 to (2**C_LBUS_AWWIDTH)-1;
                           signal sucess : out boolean);

procedure mutex_unlock( signal i_call : out kernel_input_t;
                        signal o_response: in kernel_output_t;
                        mutex_addr:in  natural range 0 to (2**C_LBUS_AWWIDTH)-1);

procedure lbus_read_reg ( signal i_call : out kernel_input_t;
                          signal o_response: in kernel_output_t;
                          reg_offset: in natural range 0 to (2**C_LBUS_REG_WIDTH)-1;
                          data: out std_logic_vector(31 downto 0));

procedure lbus_write_reg ( signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           reg_offset: in natural range 0 to (2**C_LBUS_REG_WIDTH)-1;
                           data: in std_logic_vector(31 downto 0));

procedure lram_read_word ( signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1;
                           data: out std_logic_vector(31 downto 0));

procedure lram_write_word ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1;
                            data: in std_logic_vector(31 downto 0));

procedure lram_read_word_burst ( signal i_call : out kernel_input_t;
                                 signal o_response: in kernel_output_t;
                                 transfer_len:in natural; --len
                                 lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1;
                                 data: out std_logic_vector(31 downto 0));

procedure lram_write_word_burst (signal i_call : out kernel_input_t;
                                 signal o_response: in kernel_output_t;
                                 transfer_len:in natural; --len-1
                                 lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1;
                                 data: in std_logic_vector(31 downto 0));

procedure lram_read_byte ( signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           lram_address8: in natural range 0 to (2**(CLRAM_AWWIDTH+2))-1;
                           data: out std_logic_vector(7 downto 0));

procedure lram_write_byte (signal i_call : out kernel_input_t;
                           signal o_response: in kernel_output_t;
                           lram_address8: in natural range 0 to (2**(CLRAM_AWWIDTH+2))-1;
                           data: in std_logic_vector(7 downto 0));
```

Listing C.23: Kernel Package - VHDL procedures declaration (part 3/3).

```
...
procedure byte_to_lbus_word( signal data: in std_logic_vector(7 downto 0);
                             signal be: in std_logic_vector(3 downto 0);
                             signal lbus_data: out std_logic_vector(31 downto 0));

procedure byte_from_lbus_word( signal lbus_data: in std_logic_vector(31 downto 0);
                               signal be: in std_logic_vector(3 downto 0);
                               signal data: out std_logic_vector(7 downto 0));

procedure sysram_get_virt_address ( signal i_call : out kernel_input_t;
                               signal o_response: in kernel_output_t;
                               signal sysram_address: out unsigned(31 downto 0));

procedure mst_bus_read_word (signal i_call : out kernel_input_t;
                             signal o_response: in kernel_output_t;
                             bus_address: in std_logic_vector(31 downto 0);
                             data: out std_logic_vector(31 downto 0));

procedure mst_bus_write_word (signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              bus_address: in std_logic_vector(31 downto 0);
                              data: in std_logic_vector(31 downto 0));

procedure mst_bus_read_word_burst (signal i_call : out kernel_input_t;
                                   signal o_response: in kernel_output_t;
                                   burst_len:in natural; --len-1
                                   bus_address: in std_logic_vector(31 downto 0);
                                   data: out std_logic_vector(31 downto 0));

procedure mst_bus_write_word_burst ( signal i_call : out kernel_input_t;
                                     signal o_response: in kernel_output_t;
                                     burst_len:in natural; --len-1
                                     bus_address: in std_logic_vector(31 downto 0);
                                     data: in std_logic_vector(31 downto 0));

procedure task_yield( signal i_call : out kernel_input_t;
                      signal o_response: in kernel_output_t);

procedure sys_call_return( signal kernel_response: in kernel_output_t;
                           signal ret: out boolean);
...
```

Listing C.24: User Package - VHDL procedures declaration (part 1/6).

```
package hal_asos_user_pkg is
-------------------------------------------------------------
procedure reset_pooled_object(signal umutex: out pooled_mutex_t );

procedure reset_pooled_object(signal usem: out pooled_semaphore_t );

procedure reset_pooled_object(signal ucond: out pooled_condition_t );

procedure reset_pooled_object(signal ubuff: out pooled_array_t);

procedure reset_pooled_object(signal udev: out pooled_device_t );

procedure reset_pooled_object(signal ufile: out pooled_file_t );

procedure reset_pooled_object(signal uobj: out pooled_object_t );
-------------------------------------------------------------
procedure pooled_mutex_query( signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal umutex_q: in pooled_mutex_t;
                              signal umutex_d: out pooled_mutex_t);
-------------------------------------------------------------
procedure pooled_semaphore_query(signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal usem_q: in pooled_semaphore_t;
                              signal usem_d: out pooled_semaphore_t);
-------------------------------------------------------------
procedure pooled_condition_query(signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal ucond_q: in pooled_condition_t;
                              signal ucond_d: out pooled_condition_t);
-------------------------------------------------------------
procedure pooled_buffer_query(signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal pbuff_q: in pooled_array_t;
                              signal pbuff_d: out pooled_array_t);
-------------------------------------------------------------
procedure pooled_file_query(signal i_call : out kernel_input_t;
                            signal o_response  : in kernel_output_t;
                            signal ufile_q: in pooled_file_t;
                            signal ufile_d: out pooled_file_t);
-------------------------------------------------------------
procedure pooled_device_query(signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal udev_d: in pooled_device_t;
                              signal udev_q: out pooled_device_t);
-------------------------------------------------------------
procedure pooled_mutex_lock(signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            signal umutex_d: in pooled_mutex_t;
                            signal umutex_q: out pooled_mutex_t);
-------------------------------------------------------------
procedure pooled_fstream_query(signal i_call : out kernel_input_t;
                               signal o_response  : in kernel_output_t;
                               signal ustream_q: in pooled_fstream_t;
                               signal ustream_d: out pooled_fstream_t);
-------------------------------------------------------------
procedure pooled_socket_query( signal i_call : out kernel_input_t;
                               signal o_response  : in kernel_output_t;
                               signal usock_q: in pooled_socket_t;
                               signal usock_d: out pooled_socket_t);
-------------------------------------------------------------
procedure pooled_mutex_unlock( signal i_call : out kernel_input_t;
                               signal o_response: in kernel_output_t;
                               signal umutex_d: in pooled_mutex_t;
                               signal umutex_q: out pooled_mutex_t)
```

Listing C.25: User Package - VHDL procedures declaration (part 2/6).

```
-------------------------------------------------------------
procedure pooled_semaphore_wait(signal i_call : out kernel_input_t;
                                signal o_response: in kernel_output_t;
                                signal usem_d: in pooled_semaphore_t;
                                signal usem_q: out pooled_semaphore_t);
-------------------------------------------------------------
procedure pooled_semaphore_post(signal i_call : out kernel_input_t;
                                signal o_response: in kernel_output_t;
                                signal usem_q: in pooled_semaphore_t;
                                signal usem_d: out pooled_semaphore_t);
-------------------------------------------------------------
procedure pooled_condition_signal(signal i_call : out kernel_input_t;
                                  signal o_response: in kernel_output_t;
                                  signal ucond_q: in pooled_condition_t;
                                  signal ucond_d: out pooled_condition_t);
-------------------------------------------------------------
procedure pooled_condition_wait(signal i_call : out kernel_input_t;
                                signal o_response: in kernel_output_t;
                                signal ucond_q: in pooled_condition_t;
                                signal ucond_d: out pooled_condition_t);
-------------------------------------------------------------
procedure transfer_control_to_dds  ( signal i_call : out kernel_input_t;
                                    signal o_response: in kernel_output_t;
                                    constant len: in natural);
-------------------------------------------------------------
procedure transfer_control_from_dds(signal i_call : out kernel_input_t;
                                    signal o_response: in kernel_output_t;
                                    constant len: in natural );
-------------------------------------------------------------
procedure transfer_data_to_dds( signal i_call : out kernel_input_t;
                                signal o_response: in kernel_output_t;
                    constant lram_address:in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1);
                                constant len: in natural);
-------------------------------------------------------------
procedure transfer_data_from_dds(signal i_call : out kernel_input_t;
                                 signal o_response: in kernel_output_t;
                    constant lram_address:in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1);
                                    constant len: in natural );
-------------------------------------------------------------
procedure transfer_to_host_swfifo( signal i_call : out kernel_input_t;
                                   signal o_response: in kernel_output_t;
                                   constant len: in natural);
-------------------------------------------------------------
procedure transfer_from_host_swfifo(signal i_call : out kernel_input_t;
                                    signal o_response: in kernel_output_t;
                                    constant len: in natural);
-------------------------------------------------------------
procedure safe_write_lram_word32 ( signal i_call : out kernel_input_t;
                                   signal o_response: in kernel_output_t;
                                   signal pbuff: in t_array_slv_32;
                             constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                      constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
-------------------------------------------------------------
procedure safe_read_lram_word32 ( signal i_call : out kernel_input_t;
                                  signal o_response: in kernel_output_t;
                                  signal pbuff: out t_array_slv_32;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                     constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
-------------------------------------------------------------
procedure safe_write_lram_word32_burst (signal i_call : out kernel_input_t;
                                        signal o_response: in kernel_output_t;
                                        signal pbuff: in t_array_slv_32;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                     constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
```

Listing C.26: User Package - VHDL procedures declaration (part 3/6).

```
------------------------------------------------------------
procedure safe_read_lram_word32_burst ( signal i_call : out kernel_input_t;
                                        signal o_response: in kernel_output_t;
                                        signal pbuff: out t_array_slv_32;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                     constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
------------------------------------------------------------
procedure unsafe_read_lram_word32_burst (signal i_call : out kernel_input_t;
                                         signal o_response: in kernel_output_t;
                                         signal pbuff: out t_array_slv_32;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                     constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
------------------------------------------------------------
procedure safe_write_sysram_word32 (signal i_call : out kernel_input_t;
                                    signal o_response: in kernel_output_t;
                                    signal pbuff: in t_array_slv_32;
                          constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                      constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
------------------------------------------------------------
procedure safe_read_sysram_word32 (signal i_call : out kernel_input_t;
                                   signal o_response: in kernel_output_t;
                                   signal pbuff: out t_array_slv_32;
                           constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                    constant lram_address: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
------------------------------------------------------------
procedure safe_write_lram ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            signal pbuff: in t_array_slv_8;
                           constant len: in natural range 1 to 2**(CLRAM_AWWIDTH+2)-1;
                  constant lram_address: in natural range 0 to 2**(CLRAM_AWWIDTH+2)-1);
------------------------------------------------------------
procedure safe_read_lram (signal i_call : out kernel_input_t;
                          signal o_response: in kernel_output_t;
                          signal pbuff: out t_array_slv_8;
                          constant len: in natural range 1 to (2**(CLRAM_AWWIDTH+2)-1);
                  constant lram_address: in natural range 0 to 2**(CLRAM_AWWIDTH+2)-1);
------------------------------------------------------------
procedure safe_write_lram ( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            constant fmessage: in string;
                            constant parameters: in std_logic_vector;
                 constant lram_address: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
------------------------------------------------------------
procedure safe_write_sysram_word32_burst (signal i_call : out kernel_input_t;
                                          signal o_response: in kernel_output_t;
                                          signal pbuff: in t_array_slv_32;
                          constant word_len: in natural  range 1 to (2**CLRAM_AWWIDTH);
                          constant offset: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
------------------------------------------------------------
procedure safe_read_sysram_word32_burst (signal i_call : out kernel_input_t;
                                         signal o_response: in kernel_output_t;
                                         signal pbuff: out t_array_slv_32;
                           constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
                           constant offset: in natural range 0 to (2**CLRAM_AWWIDTH)-1);
------------------------------------------------------------
procedure pooled_buffer_write (signal i_call : out kernel_input_t;
                               signal o_response: in kernel_output_t;
                               signal shd_buff: in pooled_array_t;
                         constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
                               constant buff_offset: in natural range 0 to 65535;
                    constant lram_address:in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
```

Listing C.27: User Package - VHDL procedures declaration (part 4/6).

```
-------------------------------------------------------------
procedure pooled_buffer_read( signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal shd_buff: in pooled_array_t;
                        constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
                              constant buff_offset: in natural range 0 to 65535;
                  constant lram_address:in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
-------------------------------------------------------------
procedure pooled_file_write (signal i_call : out kernel_input_t;
                             signal o_response: in kernel_output_t;
                             signal ufile: in pooled_file_t;
                       constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
             constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
-------------------------------------------------------------
procedure pooled_file_read( signal i_call : out kernel_input_t;
                            signal o_response: in kernel_output_t;
                            signal ufile: in pooled_file_t;
                        constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
              constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
-------------------------------------------------------------
procedure pooled_device_write (signal i_call : out kernel_input_t;
                               signal o_response: in kernel_output_t;
                               signal udevice: in pooled_device_t;
                        constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
             constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
-------------------------------------------------------------
procedure pooled_device_read( signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal udevice: in pooled_device_t;
                        constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
              constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
-------------------------------------------------------------
procedure pooled_file_close( signal i_call : out kernel_input_t;
                             signal o_response: in kernel_output_t;
                             signal ufile: inout pooled_file_t);
-------------------------------------------------------------
procedure pooled_file_open( signal i_call : out kernel_input_t;
                            signal o_response : in kernel_output_t;
                            signal ufile: inout pooled_file_t);
--------------------------------------------------------------
procedure pooled_fstream_open(signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal ufstream_q: in pooled_fstream_t;
                              signal ufstream_d: out pooled_fstream_t);
-------------------------------------------------------------
procedure pooled_fstream_close(signal i_call : out kernel_input_t;
                               signal o_response  : in kernel_output_t;
                               signal ufstream_q: in pooled_fstream_t;
                               signal ufstream_d: out pooled_fstream_t);
-------------------------------------------------------------
procedure pooled_fstream_write (signal i_call : out kernel_input_t;
                                signal o_response: in kernel_output_t;
                                signal ustream: in pooled_fstream_t;
                         constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
               constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
-------------------------------------------------------------
procedure pooled_fstream_read(signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal ustream: in pooled_fstream_t;
                    constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
               constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
```

Listing C.28: User Package - VHDL procedures declaration (part 5/6).

```
--------------------------------------------------------------
procedure pooled_fstream_write_word32 (signal i_call : out kernel_input_t;
                                       signal o_response: in kernel_output_t;
                                       signal ustream: in pooled_fstream_t;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
             constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
--------------------------------------------------------------
procedure pooled_fstream_read_word32(signal i_call : out kernel_input_t;
                                     signal o_response: in kernel_output_t;
                                      signal ustream: in pooled_fstream_t;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
              constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
--------------------------------------------------------------
procedure pooled_fstream_read_len(signal i_call : out kernel_input_t;
                                  signal o_response: in kernel_output_t;
                                  signal ustream: in pooled_fstream_t;
                                  signal flen:out integer);
--------------------------------------------------------------
procedure pooled_socket_open( signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal usock_q: in pooled_socket_t;
                              signal usock_d: out pooled_socket_t);
--------------------------------------------------------------
procedure pooled_socket_close( signal i_call : out kernel_input_t;
                               signal o_response  : in kernel_output_t;
                               signal usock_q: in pooled_socket_t;
                               signal usock_d: out pooled_socket_t);
--------------------------------------------------------------
procedure pooled_socket_write ( signal i_call : out kernel_input_t;
                                signal o_response: in kernel_output_t;
                                signal usocket: in pooled_socket_t;
                       constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
             constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
--------------------------------------------------------------
procedure pooled_socket_read( signal i_call : out kernel_input_t;
                              signal o_response: in kernel_output_t;
                              signal usocket: in pooled_socket_t;
                        constant byte_len: in natural range 1 to (2**(CLRAM_AWWIDTH+2));
              constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
--------------------------------------------------------------
procedure pooled_socket_write_word32 (signal i_call : out kernel_input_t;
                                      signal o_response: in kernel_output_t;
                                      signal usocket: in pooled_socket_t;
                            constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
              constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
--------------------------------------------------------------
procedure pooled_socket_read_word32( signal i_call : out kernel_input_t;
                                     signal o_response: in kernel_output_t;
                                     signal usocket: in pooled_socket_t;
                          constant word_len: in natural range 1 to (2**CLRAM_AWWIDTH);
            constant source_laddress: in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
--------------------------------------------------------------
procedure pooled_device_close( signal i_call : out kernel_input_t;
                               signal o_response  : in kernel_output_t;
                               signal udev: inout pooled_device_t);
--------------------------------------------------------------
procedure pooled_device_open( signal i_call : out kernel_input_t;
                              signal o_response : in kernel_output_t;
                              signal udev: inout pooled_device_t);
```

Listing C.29: User Package - VHDL procedures declaration (part 6/6).

```
----------------------------------------------------------
procedure write_stdio( signal i_call : out kernel_input_t;
                      signal o_response : in kernel_output_t;
                      constant slen:in natural range 1 to (2**(CLRAM_AWWIDTH+2));
                      constant mlen:in natural range 0 to (2**(CLRAM_AWWIDTH+2));
             constant lram_address:in natural range 0 to (2**(CLRAM_AWWIDTH+2)-1));
----------------------------------------------------------
procedure task_exit(signal i_call : out kernel_input_t;
                    signal o_response: in kernel_output_t);
----------------------------------------------------------
```

Listing C.30: User package - VHDL procedure to read the *sysram* memory.

```
package hal_asos_user_pkg is
...
------------------------------------------------------------------------------------------
procedure safe_read_sysram_word32 (signal i_call : out kernel_input_t;
                      signal o_response: in kernel_output_t;
                      signal pbuff: out t_array_slv_32;
                      constant word_len: in natural;
                      constant offset: in natural)is
------------------------------------------------------------------------------------------
constant ALNMNT: natural:=POW2(C_MACHINE_WIDTH/8);
variable waddr: natural :=offset + o_response.index;
variable param_woffs: unsigned(C_MACHINE_WIDTH-1 downto 0)
         := to_unsigned(waddr, C_MACHINE_WIDTH) rol ALNMNT;
variable word: std_logic_vector(31 downto 0);
begin
if word_len = 0 then
  i_call.enable_scheduler <= '0';
  i_call.reschedule       <= '0';
else
 i_call.enable_scheduler <= '1';
 i_call.reschedule       <= '1';

  case o_response.sched_progress is
     when 0=>
          mutex_lock(i_call,o_response,CSYSMUTEX_WOFFSET);
     when 1=>
          i_call.enable_index <= '1';
          i_call.increment_index<= '1';
          i_call.reschedule<= '0';
          mst_bus_read_word(i_call,
                o_response,std_logic_vector(param_woffs),word);
          pbuff(o_response.index) <= word;
          if o_response.index = word_len-1 then
            i_call.reschedule<= '1';
            i_call.increment_index<= '0';
          end if;
     when 2=>
          mutex_unlock(i_call,o_response,CSYSMUTEX_WOFFSET);
          i_call.enable_scheduler <= '0';
     when others=>null;
  end case;
end if;
end procedure;
------------------------------------------------------------------------------------------
```

back to Figure 3.31.

Listing C.31: Data transfer message - file descriptor write sequence diagram.

```
package hal_asos_user_pkg is
...
------------------------------------------------------------------------------------
procedure pooled_file_async_write (signal i_call : out kernel_input_t;
                 signal o_response: in kernel_output_t;
                 signal ufile_q: in file_descriptor_t;
                 constant byte_len: in natural range 1 to (2**(C_LRAM_AWWIDTH+2));
                 constant source_laddress: in natural range 0 to (2**(C_LRAM_AWWIDTH+2)-
1))is
------------------------------------------------------------------------------------
variable dmsg: kdata_transfer_m;
begin
   dmsg.xcode    := std_logic_vector(to_unsigned(exec_code_t'pos(TRANSFER_TO_POOLED),8));
   dmsg.index    := std_logic_vector(to_signed(ufile_q.index,8));
   dmsg.transfer_len:= std_logic_vector(to_unsigned(byte_len,dmsg.transfer_len'length));
   dmsg.receiver_offset:= (others=>'0');
   dmsg.sender_offset
            := std_logic_vector(to_unsigned(source_laddress,dmsg.sender_offset'length));
   send_message(i_call,o_response,dmsg);
end procedure;
------------------------------------------------------------------------------------

------------------------------------------------------------------------------------
procedure pooled_file_async_write_complete (signal i_call : out kernel_input_t;
               signal o_response: in kernel_output_t;
               signal ufile_q: in file_descriptor_t;
               signal byte_len: out natural range 1 to (2**(C_LRAM_AWWIDTH+2));
               signal source_laddress: out natural  range 0 to (2**(C_LRAM_AWWIDTH+2)-1))is
------------------------------------------------------------------------------------
variable dmsg : kdata_transfer_m;
variable received_index: integer range -128 to 127;
begin

  receive_message(i_call,o_response);
  dmsg := cast_return_to_kdata_transfer_message(o_response);
  received_index:=to_integer(signed(dmsg.index));

  byte_len <= 0;
  source_laddress<= 0;
  if received_index = ufile_q.index then
     byte_len <= to_integer(unsigned(dmsg.transfer_len));
     source_laddress<= to_integer(unsigned(dmsg.sender_offset));
  end if;
end procedure;
------------------------------------------------------------------------------------
```

back to Figure 3.37.

Listing C.32: HAL-ASOS Accelerator - HW-Kernel configurations for the a, c, d and e variants.

```
architecture hwkernel of hal_kernel is
...
end hwkernel

configuration hal_kernel_v4_00_a_config of hal_kernel is
for hwkernel
    for XS00:xsync_gen
        use entity hal_asos_v4_00_A.sync_gen(blank);
    end for;
    for XS01:xsgen_bus
        use entity hal_asos_v4_00_A.gen_bus(single_clock);
    end for;
    for XM00: xmgen_bus
        use entity hal_asos_v4_00_a.gen_bus(single_clock);
    end for;
end for;
end configuration hal_kernel_v4_00_a_config;
...

configuration hal_kernel_v4_00_c_config of hal_kernel is
for hwkernel
    for XS00:xsync_gen
        use entity hal_asos_v4_00_A.sync_gen(blank);
    end for;
    for XS01:xsgen_bus
        use entity hal_asos_v4_00_A.gen_bus(single_clock);
    end for;
    for XM00: xmgen_bus
        use entity hal_asos_v4_00_a.gen_bus(single_clock);
    end for;
end for;
end configuration hal_kernel_v4_00_c_config;

configuration hal_kernel_v4_00_d_config of hal_kernel is
for hwkernel
    for XS00:xsync_gen
        use entity hal_asos_v4_00_A.sync_gen(dual_clock);
    end for;
    for XS01:xsgen_bus
        use entity hal_asos_v4_00_A.gen_bus(single_clock);
    end for;
    for XM00: xmgen_bus
        use entity hal_asos_v4_00_a.gen_bus(single_clock);
    end for;
end for;
end configuration hal_kernel_v4_00_d_config;

configuration hal_kernel_v4_00_e_config of hal_kernel is
for hwkernel
    for XS00:xsync_gen
        use entity hal_asos_v4_00_A.sync_gen(dual_clock);
    end for;
    for XS01:xsgen_bus
        use entity hal_asos_v4_00_A.gen_bus(mxclock_a_is_y);
    end for;
    for XM00: xmgen_bus
        use entity hal_asos_v4_00_a.gen_bus(single_clock);
    end for;
end for;
end configuration hal_kernel_v4_00_e_config;
...
```

back to Figure 4.22.

Listing C.33: File reader software Task *run* member.

```
hal_asos::TaskConfig_t
TFileRead = { "FileReader", //TaskTag
            { "Imagelines",BLOCK_LEN,1,1 },//Topic
            { "",0 }// No Subscription
};
...
hal_asos::Task<>* p_Detector_Task;
template<>
void hal_asos::Task <hal_asos::SwTask, TFileRead> ::run(void) {
  std::ifstream input_file; std::string s;  std::stringstream ss;
  std::string header0 = ""; std::string header1= ""; std::string header2= "";
  std::string version, width, height;
  std::shared_ptr<char[BLOCK_LEN]> p_line;
   std::shared_ptr<StreamData> Conf = std::make_shared<StreamData>(16);
   detector::config_words *p_config = (detector::config_words*)Conf.get();
  int file_length, Read_len=0, count = 0, pixel_len = 0;
  input_file.open(hal_asos_demo::feature_detector::scene_img.c_str(), std::ios::in |
 std::ifstream::binary);
  if (!input_file.is_open()) {
    LOG_MSG << this->TaskTag << ":error opening input file!\n";
    this->shutdown_unconditional();
    return;
  }// First line : version
  getline(input_file, header0);
  if (delimiter.compare("P5") != 0) {
    LOG_MSG << "Wrong file format or version\n";
    this->shutdown_unconditional();
    return;
  }
  // secondline: wxh
  getline(input_file, header1);
  ss.str(header1);
  ss >> p_config->image_width >> p_config->image_height;
  if (p_config->image_width > BLOCK_LEN) {
    this->StatusRunning = false;
    LOG_MSG <<this->TaskTag<< "Topiclen is smaller than image width\n";
    this->shutdown_unconditional();
    return;
  }
  //third line
  getline(input_file, header2);
   p_config->threshould = hal_asos_demo::feature_detector::th;
  file_length = p_config->image_width * p_config->image_height;
  Read_len = p_config->image_width;
  if (p_Detector_Task) {
         p_config->block_len = hal_asos_demo::feature_detector::block_size;
         p_Detector_Task->submit_data(Conf);
   }
  while (this->StatusRunning && Read_len > 0) {
    p_line = std::shared_ptr<char[BLOCK_LEN]>(new char[p_config->image_width], [](char* p)
        { delete[] p; });
    input_file.read(p_line.get(), p_config->image_width);
    Read_len = (int)input_file.gcount();
    if (Read_len) {
      file_length -= Read_len;pixel_len += Read_len;
      this->p_Topic->publish(p_line, Read_len);count++;
      p_line = nullptr;
    }
    else p_line = nullptr;
  }
  input_file.close();
  this->p_Topic->close_topic();
  LOG_MSG << this->TaskTag << "finished...(" << count << "," << " " << pixel_len << ")\n";
}
```

back Figure 5.6.

Listing C.34: CornerDump and CornerUploader software *run* members.

```
hal_asos::TaskConfig_t
TCornerDump = { "CornerDump",
{ "",0},
{ "Corners",CORNER_LEN,1,1 },
{ 1,1,1,1 }};
...
template<>
void hal_asos::Task <hal_asos::SwTask, TCornerDump> ::run(void) {
    std::ofstream CornersFile;
    int ret = 1, count = 0;
    std::shared_ptr<const char[]> p_char;
    std::shared_ptr<dds::Publication> pLocal;
    std::stringstream lines;
    CornersFile.open("Corners.bin", std::ios::out | std::ios::trunc|std::ios::binary);
    this->StatusRunning = true;
    while (this->StatusRunning && ret > 0) {
        ret = this->p_Subscription->take_publication(pLocal);
        if (ret) {
            p_char = pLocal->get_reference();
            ret = pLocal->get_len();
            CornersFile.write(p_char.get(), ret);
            count+=(ret>>2);
        }
    }
    CornersFile.close();
    this->p_Subscription->close_subscription();
    LOG_MSG << this->TaskTag << "finished...(" << count << ")\n";
}
...
template<>
void hal_asos::Task <hal_asos::SwTask, TCornerUploader> ::run(void) {
    using namespace hal_asos::networking;
    int ret = 1, index = 0, count = 0; FrameControl iFrame;
    std::shared_ptr<const char[]> p_block;
    std::shared_ptr<dds::Publication> pLocal;
    hal_asos::networking::CSocket<hal_asos::networking::Client> Soc;

    Soc.set_ip_address(hal_asos_demo::feature_detector::image_ip);
    Soc.set_sock_type(SOCK_STREAM); Soc.set_sock_family(AF_INET);
    Soc.set_sock_port(IMAGE_PORT_NO);

    this->StatusRunning = Soc.open_connection();
    if (!this->StatusRunning) {
        Soc.get_error_message(hal_asos_demo::feature_detector::image_ip);
        LOG_MSG << hal_asos_demo::feature_detector::image_ip << "\n";
        return;}
    iFrame.top = TOPSYMBOL; iFrame.delimitor = CONTROLSYMBOL;
    std::copy_n(scene_img.c_str(),scene_img.length(), iFrame.filename);
    iFrame.filename[hal_asos_demo::feature_detector::scene_img.length()] = 0;
    iFrame.th = FEATURE_THRESHOLD; iFrame.block_len = CORNER_LEN;

    this->StatusRunning = true;
    Soc.safe_write((char*)&iFrame, sizeof(struct FrameControl));
    while (this->StatusRunning && ret > 0) {
        ret = this->p_Subscription->take_publication(pLocal);
        index += ret;
        if (ret){
         p_block = pLocal->get_reference();
         count = Soc.safe_write(p_block.get(), ret);
         if (count < ret) {
            LOG_MSG << "socket write failed to transfer " << (index - count) << "bytes\n";
    }}}
    this->p_Subscription->close_subscription();
    LOG_MSG << this->TaskTag << "finished...(" << index << ")\n";
    Soc.safe_read((char*)&ret, 4); Soc.close_connection();
}
```

back Figure 5.6.

Listing C.35: Full Feature detector software thread *run* member(1/2).

```
hal_asos::TaskConfig_t TFeatureDetector = { "FeatureDetector0",
{ "Corners",CORNER_LEN,1,1 },
{ "Imagelines",BLOCK_LEN,1,1 },
{ 1,1,1,1 } };
...
template<>
void hal_asos::Task <hal_asos::SwTask, TFeatureDetector> ::run(void) {
 detector::config_words *p_config_local;
 std::shared_ptr<StreamData> p_Config;
 bool fast_line; uint16_t* p_uint16;
 int block_size = 0, count_blocks = 0, count_corners = 0;
 int index = 0, line = 0, target = 0, corner_index = 0;
 std::shared_ptr<uint16_t[]> p_score_line;
 std::shared_ptr<const char[]> p_block;
 std::shared_ptr<char[]> p_line;
 std::shared_ptr<char[CORNER_LEN]> p_corner_block;
 std::shared_ptr<detector::corner_t> p_corner;
 std::shared_ptr<dds::Publication> pLocal;

 detector::fast::Fast F1;
 detector::nonmaximal::NonMaxSupression N1;

 block_size = this->p_Subscription->take_publication(pLocal);
 if (block_size <= 0) {
  this->StatusRunning = false;
  this->shutdown_unconditional();
  this->p_Topic->kill_topic();
  LOG_MSG <<this->TaskTag<< "Failed reading subscription\n";
  return; }

 count_blocks++;
 this->pop_data_in(p_Config);
 p_config_local = (detector::config_words*) p_Config->get();
 F1.start_fast(p_config_local);
 N1.start_nms(p_config_local);

 target = p_config_local->image_width;

 p_score_line = std::shared_ptr<uint16_t[]>(new uint16_t[target]);
 p_corner_block = std::shared_ptr<char[CORNER_LEN]>(new char[CORNER_LEN]);
 if (p_score_line == nullptr || p_corner_block == nullptr) {
  this->StatusRunning = false;
  this->shutdown_unconditional();
  this->p_Topic->kill_topic();
  LOG_MSG << this->TaskTag << "Failed reading subscription\n";
  return;}
 p_block = pLocal->get_reference();
 block_size = pLocal->get_len();

 if ((unsigned)block_size < target) {
  this->StatusRunning = false;
  this->shutdown_unconditional();
  this->p_Topic->kill_topic();
  LOG_MSG << this->TaskTag << "Failed - topic len is lower\n";
  return;  }

 p_line = std::shared_ptr<char[]>(std::const_pointer_cast<char[]>(p_block));
 this->StatusRunning = true;
```

continue...

Listing C.36: Full Feature detector software thread *run* member(2/2).

```
while (this->StatusRunning && block_size > 0) {
  fast_line = F1.submit_line(p_line);
   if (fast_line) {
     F1.pop_scores(p_score_line);
     N1.submit_line(p_score_line);
     while (N1.corners_found()) {
       N1.pop_corners(p_corner);
       p_uint16 = (uint16_t*)&p_corner_block[corner_index];
       p_uint16[0] = p_corner->x_coord;
       p_uint16[1] = p_corner->y_coord;
       corner_index += 4; count_corners++;
       if (corner_index == CORNER_LEN) {
         this->p_Topic->publish(p_corner_block, corner_index);
         corner_index = 0;
         p_corner_block = std::shared_ptr<char[CORNER_LEN]>(new char[CORNER_LEN]);
         if (!p_corner_block) {
         this->StatusRunning = false;
         block_size = 0;line = 0;
         LOG_MSG << "Failed memory allocation\n";}
        }
      }
      if (p_score_line == nullptr)
        p_score_line = std::shared_ptr<uint16_t[]>(new uint16_t[target]);
   }
   block_size = this->p_Subscription->take_publication(pLocal);
    if (block_size) {
      p_block = pLocal->get_reference();
      p_line = std::shared_ptr<char[]>(std::const_pointer_cast<char[]>(p_block));
     count_blocks++;}
 }
 if (corner_index)
 this->p_Topic->publish(p_corner_block, corner_index);
 this->p_Subscription->close_subscription();
 this->p_Topic->close_topic();
 LOG_MSG << this->TaskTag << "finished...("<<count_blocks<<","<<count_corners<<")\n";
}
```

back to Figure 5.6.

Listing C.37: Software application for the synchronous standalone feature detection.

```
void hal_asos_demo::feature_detector::
  test_fast_detector_std_alone_single_sysram(void) {
    using namespace hal_asos;

    std::string s;
    std::string header0 = "";
    std::string header1 = "";
    std::string header2 = "";
    std::string version, width, height;
    std::stringstream ss;
    std::shared_ptr<StreamData> Conf;
   Task<HwTask, TFastDetectorSA, segment_len<(BLOCK_LEN << 1)>> T1;

    Conf = std::make_shared<StreamData>(16);
    detector::config_words* p_config = (detector::config_words*) Conf.get();


    CFstream<std::ifstream> Input_file(scene_img.c_str());
    Input_file.set_flags(std::ios::in | std::ifstream::binary);

    CFstream<std::ofstream> Output_file("Corners_std_single.txt");
    Output_file.set_flags(std::ios::out |std::ios::trunc|std::ios::binary);


    if (Output_file.open_file() < 0) {
        LOG_MSG << "[" << __FUNCTION__ << "]:error opening output file!\n";
        return;
    }

    if (Input_file.open_file() < 0) {
        LOG_MSG << "[" << __FUNCTION__ << "]:error opening input file!\n";
        return;
    }

     Input_file.get_line(header0);
    if (header0.compare("P5") != 0) {
        LOG_MSG << "[" << __FUNCTION__ << "]:Wrong file format or version\n";
        return;
    }

    Input_file.get_line(header1);

    ss.str(header1);
    ss >> p_config->image_width >> p_config->image_height;
    ss.clear();

    if (p_config->image_width > BLOCK_LEN) {
        LOG_MSG << "[" << __FUNCTION__ << "Topiclen is smaller than image width\n";
        return;
    }


    Input_file.get_line(header2);

    p_config->threshould = th;
    p_config->block_len = block_size;

    T1.submit_to_pool(Input_file);
    T1.submit_to_pool(Output_file);

    T1.submit_data(Conf);

    T1.start();
    T1.join();
}
```

back to Figure 5.17.

Listing C.38: Synchronous control for HW accelerated feature detection(1/2).

```
read_corners_i   <= '1' when fifo_size_block_q >= fifo_out_burst_q and fifo_out_burst_q >
0 else '0';
write_block_i    <= '1' when count_crnr_bytes_q >=C_TRGT_UPLOAD else '0';
line0_exausted_i <= '1' when to_integer(unsigned(line0_in_size_Q)) = 0 else '0';
read_inblock_i   <= '1' when pixel_counter_q >= pixels_target_q else '0';
-------------------------------------------------------------------------------------
CONTROL_FSM: process(task_state,s00_kernel_run,resetn_i, kernel_response,
line0_in_burst_q,line0_exausted_i,line0_in_word_space,line0_in_size_q,count_crnr_bytes_q,
pixels_target_q,read_inblock_i, pixel_counter_q, fifo_size_block_q,trfr_len_q,
read_corners_i, write_block_i)
---------------------------------------------------------------------------------------
begin
task_done_i<= '0'; task_state_next <= task_state; WR_CE_i <= '0'; inc_rindex_i<='0'
clr_pixel_counter_i <= '0'; load_rindex_i<='0'; load_windex_i<='0';
inc_pixel_block_counter<='0';
case task_state is
    when s0_ready=>
       if s00_kernel_run = '1' then
          task_state_next <= s1_read_config;
       end if;
    when s1_read_config=>
       task_state_next <= s2_write_config;
    when s2_write_config=>
       WR_CE_i <= '1';
       if kernel_response.index = (C_CONF_LEN/4)-1 then
          task_state_next <=s3_config_run;
       end if;
    when s3_config_run=>
       WR_CE_i <= '1';
       load_windex_i <= '1';
       task_state_next <=s4_read_block;
    when s4_read_block=>
       clr_pixel_counter_i <='1';
       load_rindex_i<='1';
       task_state_next <= s5_eval_read;
    when s5_eval_read=>
       task_state_next <= s16_exhausted_file;
       inc_pixel_block_counter<= '1';
       if(pixels_target_q > 0) then
         task_state_next <= s6_recheck_burst_in;
       end if;
    when s6_recheck_burst_in=>
       task_state_next <= s8_check_crnrs;
       if(line0_in_burst_q > 1) then
         task_state_next <= s7_write_pixels;
       elsif (line0_in_burst_q = 1) then
         task_state_next <=s7_write_pixel;
       end if;
    when s7_write_pixel=>
       inc_rindex_i<='1';
       task_state_next <= s8_check_crnrs;
    when s7_write_pixels=>
       inc_rindex_i<='1';
       task_state_next <= s8_check_crnrs;
    when s8_check_crnrs=>
       task_state_next <= s9_check_block_target;
       if read_corners_i= '1' then
          task_state_next <= st10_lock_rsrc_mutex;
       end if;
    when s9_check_block_target =>
       task_state_next <= s6_recheck_burst_in;
       if(read_inblock_i = '1') then
          task_state_next <= s4_read_block;
       end if;
...
```

Listing C.39: Synchronous control unit for the HW accelerated feature detection(2/2).

```
    when st10_lock_rsrc_mutex=>
       task_state_next <=s11_read_crnrs;
    when s11_read_crnrs=>
       task_state_next <=s12_unlock_rsrc_mutex;
    when s12_unlock_rsrc_mutex=>
       task_state_next <=s13_check_crnrs_target;
    when s13_check_crnrs_target=>
       task_state_next <= s9_check_block_target;
       if write_block_i = '1' then
         task_state_next <= s14_write_block;
       end if;
    when s14_write_block=>
      task_state_next<= s15_eval_write_block;
    when s15_eval_write_block=>
      load_windex_i <= '1';
      task_state_next <=  s9_check_block_target;
      if trfr_len_q = 0 then
         task_state_next <= s23_stop_fast_nms;
      end if;
    when s16_exhausted_file=>
      task_state_next <= s16_exhausted_file;
      if(line0_exausted_i = '1') then
        task_state_next <= s17_check_fifo_last;
      end if;
    when s17_check_fifo_last=>
      task_state_next <= s21_recheck_corners_last;
      if fifo_size_block_q >0 then
         task_state_next <= s18_lock_rsrc_mutex_last;
      end if;
    when s18_lock_rsrc_mutex_last=>
      task_state_next <=s19_read_corner_last;
      if fifo_size_block_q> 1 then
         task_state_next <=s19_read_corners_last;
      end if;
    when s19_read_corner_last=>
      task_state_next <=s20_unlock_rsrc_mutex_last;
    when s19_write_corners_last=>
      task_state_next <=s20_unlock_rsrc_mutex_last;
    when s20_unlock_rsrc_mutex_last=>
      task_state_next <= s21_recheck_corners_last;
    when s21_recheck_corners_last=>
      task_state_next <= s23_stop_fast_nms;
      if count_crnr_bytes_q > 0 then
         task_state_next <= s22_write_block_last;
      end if;
   when s22_write_block_last=>
      task_state_next<= s23_stop_fast_nms;
   when s23_stop_fast_nms =>
      WR_CE_i <= '1';
      task_state_next <= s24_write_message;
   when s24_write_message=>
      task_state_next <= s90_print_stdio;
   when s90_print_stdio=>
      task_state_next <= s99_exit;
   when s99_exit=>
      task_done_i<= '1';
      task_state_next <=s99_exit;
   when others=> null;
     end case;
end process CONTROL_FSM;
---------------------------------------------------------------------------
```

back to Figure 5.11.

Listing C.40: Extended features synchronous Control unit for the HW accelerated feature detection using the LRAM(1/2).

```vhdl
-----------------------------------------------------------------------------------------
EXTENEDED_FEATURES: process(task_state, resetn_i, kernel_response,
line0_in_burst_q,block_len,count_crnr_bytes_q,pixel_block_counter,pixels_target_q,
index_read_q,index_write_q,fifo_out_burst_last_q,fifo_out_burst_q,
count_crnrs_q,corners_word_i,trfr_len_q,data_in_i)
----------------------------------------------------------------------------------------------
variable RAM_DATA:STD_LOGIC_VECTOR(31 DOWNTO 0);
begin
RAM_DATA:=(OTHERS=>'0'); pixels_target_d  <= pixels_target_q;
config_d<= (others=>(others=>'0'));OFFSET_i<= (others=>'0');
pixels_word_d <= (OTHERS=>'0');pop_crnrs_i <= '0';trfr_len_d <= trfr_len_q;
inc_windex_i <='0';write_pixel_word <= '0';
if resetn_i = '0' then
   reset_sys_call(kernel_call);
else
   hal_asos_link_to_kernel(kernel_response,kernel_call);
case task_state is
   when s1_read_config=>
      transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
   when s2_write_config=>
      kernel_call.enable_index <= '1';
      kernel_call.increment_index <= '1';
      config_d(0)<=data_in_i(0);
      config_d(1)<=data_in_i(1);
      config_d(2)<=data_in_i(2);
      OFFSET_i<= std_logic_vector(to_unsigned(kernel_response.index,3));
      if kernel_response.index = (C_CONF_LEN/4)-1 then
         kernel_call.increment_index <= '0';
      end if;
   when s3_config_run=>
      config_d(0)<= x"00000080";
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s4_read_block=>
      transfer_data_from_dds(kernel_call,kernel_response,0, block_len);
      pixels_target_d <= cast_return_to_transfer_len(kernel_response);
   when s7_write_pixel=>
      lram_read_word(kernel_call, kernel_response,index_read_q,RAM_DATA);
      write_pixel_word <= cast_return_to_push_data(kernel_response);
      pixels_word_d<=RAM_DATA;
   when s7_write_pixels=>
      lram_read_word_burst(kernel_call,kernel_response,
line0_in_burst_q,index_read_q,RAM_DATA);
      write_pixel_word <= cast_return_to_push_data(kernel_response);
      pixels_word_d<=RAM_DATA;
   when st10_lock_rsrc_mutex=>
      mutex_lock(kernel_call,kernel_response,CLMUTEX_WOFFSET);
   when s11_read_crnrs=>
      lram_write_word_burst(kernel_call, kernel_response, fifo_out_burst_q,
 index_write_q,corners_word_i);
      pop_crnrs_i<=kernel_response.block_task;
      inc_windex_i<=kernel_response.block_task;
   when s12_unlock_rsrc_mutex=>
      mutex_unlock(kernel_call,kernel_response,CLMUTEX_WOFFSET);
   when s14_write_block=>
      transfer_data_to_dds(kernel_call, kernel_response,32768/4, count_crnr_bytes_q);
      trfr_len_d <= cast_return_to_transfer_len(kernel_response);
   when s18_lock_rsrc_mutex_last=>
      mutex_lock(kernel_call,kernel_response,CLMUTEX_WOFFSET);
   when s19_read_corner_last=>
      lram_write_word(kernel_call, kernel_response,index_write_q,corners_word_i);
      pop_crnrs_i<=kernel_response.block_task;
      inc_windex_i <= kernel_response.block_task;
```

back to Figure 5.12.

Listing C.41: Extended features synchronous Control unit for the HW accelerated feature detection using the LRAM(2/2).

```
   when s19_read_corners_last=>
      lram_write_word_burst(kernel_call,kernel_response,fifo_out_burst_last_q,
 index_write_q,corners_word_i);
      inc_windex_i <=kernel_response.block_task;
      pop_crnrs_i<=kernel_response.block_task;
   when s20_unlock_rsrc_mutex_last=>
      mutex_unlock(kernel_call,kernel_response,CLMUTEX_WOFFSET);
   when s22_write_block_last=>
      transfer_data_to_dds(kernel_call, kernel_response,32768/4, count_crnr_bytes_q);
      trfr_len_d <= cast_return_to_transfer_len(kernel_response);
   when s23_stop_fast_nms=>
      config_d(0)<=(others=>'0');
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s24_write_message=>
      safe_write_lram(kernel_call,kernel_response,fmessage,
std_logic_vector(to_unsigned(count_crnrs_q,32) & to_unsigned(pixel_block_counter,32)),0);
   when s90_print_stdio=>
      write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
   when s99_exit=>
      task_exit(kernel_call, kernel_response);
   when others=> null;
  end case;
end if;
end process EXTENEDED_FEATURES;
--------------------------------------------------------------------------
```

back to Figure 5.12.

Listing C.42: Extended features synchronous Control unit for the HW accelerated feature detection using the SYSRAM(1/2).

```
---------------------------------------------------------------------------------------
EXTENEDED_FEATURES: process(task_state, resetn_i, kernel_response,
line0_in_burst_q,block_len,count_crnr_bytes_q,pixel_block_counter,pixels_target_q,
index_read_q,index_write_q,fifo_out_burst_last_q,fifo_out_burst_q,
count_crnrs_q,corners_word_i,trfr_len_q,data_in_i)
---------------------------------------------------------------------------------------
variable RAM_DATA:STD_LOGIC_VECTOR(31 DOWNTO 0);
begin
RAM_DATA:=(OTHERS=>'0'); pixels_target_d  <= pixels_target_q;
config_d<= (others=>(others=>'0'));OFFSET_i<= (others=>'0');
pixels_word_d <= (OTHERS=>'0');pop_crnrs_i <= '0';trfr_len_d <= trfr_len_q;
inc_windex_i <='0';write_pixel_word <= '0';
if resetn_i = '0' then
   reset_sys_call(kernel_call);
else
   hal_asos_link_to_kernel(kernel_response,kernel_call);
case task_state is
   when s1_read_config=>
      transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
   when s2_write_config=>
      kernel_call.enable_index <= '1';
      kernel_call.increment_index <= '1';
      config_d(0)<=data_in_i(0);
      config_d(1)<=data_in_i(1);
      config_d(2)<=data_in_i(2);
      OFFSET_i<= std_logic_vector(to_unsigned(kernel_response.index,3));
      if kernel_response.index = (C_CONF_LEN/4)-1 then
         kernel_call.increment_index <= '0';
      end if;
   when s3_config_run=>
      config_d(0)<= x"00000080";
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
```

Listing C.43: Extended features synchronous Control unit for the HW accelerated feature detection using the SYSRAM(2/2).

```
   when s4_read_block=>
      transfer_data_from_dds_sysram(kernel_call,kernel_response,0, block_len);
      pixels_target_d <= cast_return_to_transfer_len(kernel_response);
   when s7_write_pixels=>
      sysram_read_word_burst(kernel_call,kernel_response,
line0_in_burst_q,index_read_q,RAM_DATA);
      write_pixel_word <= cast_return_to_push_data(kernel_response);
      pixels_word_d<=RAM_DATA;
   when st10_lock_rsrc_mutex=>
      mutex_lock(kernel_call,kernel_response, CSYSMUTEX_WOFFSET);
   when s11_read_crnrs=>
      sysram_write_word_burst(kernel_call, kernel_response, fifo_out_burst_q,
 index_write_q,corners_word_i);
      pop_crnrs_i<=kernel_response.block_task;
      inc_windex_i<=kernel_response.block_task;
   when s12_unlock_rsrc_mutex=>
      mutex_unlock(kernel_call,kernel_response, CSYSMUTEX_WOFFSET);
   when s14_write_block=>
      transfer_data_to_dds_sysram(kernel_call, kernel_response,32768/4,
 count_crnr_bytes_q);
      trfr_len_d <= cast_return_to_transfer_len(kernel_response);
   when s18_lock_rsrc_mutex_last=>
      mutex_lock(kernel_call,kernel_response, CSYSMUTEX_WOFFSET);
   when s19_read_corners_last=>
      sysram_write_word_burst(kernel_call, kernel_response,index_write_q,corners_word_i);
      pop_crnrs_i<=kernel_response.block_task;
      inc_windex_i <= kernel_response.block_task;
when s20_unlock_rsrc_mutex_last=>
      mutex_unlock(kernel_call,kernel_response, CSYSMUTEX_WOFFSET);
   when s22_write_block_last=>
      transfer_data_to_dds_sysram(kernel_call, kernel_response,32768/4,
count_crnr_bytes_q);
      trfr_len_d <= cast_return_to_transfer_len(kernel_response);
   when s23_stop_fast_nms=>
      config_d(0)<=(others=>'0');
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s24_write_message=>
      safe_write_lram(kernel_call,kernel_response,fmessage,
std_logic_vector(to_unsigned(count_crnrs_q,32) & to_unsigned(pixel_block_counter,32)),0);
   when s90_print_stdio=>
      write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
   when s99_exit=>
      task_exit(kernel_call, kernel_response);
   when others=> null;
  end case;
end if;
end process EXTENEDED_FEATURES;
--------------------------------------------------------------------------
```

back to Figure 5.12.

Listing C.44: Extended features for the standalone synchronous Control unit of the HW accelerated feature detection using the SYSRAM(1/2).

```
---------------------------------------------------------------------------------------
EXTENEDED_FEATURES: process(task_state, resetn_i, kernel_response, line0_in_burst_q,
block_len, count_crnr_bytes_q,pixel_block_counter,pixels_target_q, index_read_q,
index_write_q,fifo_out_burst_last_q,fifo_out_burst_q, count_crnrs_q,
corners_word_i,trfr_len_q,data_in_i, ifile_q, ofile_q)
-----------------------------------------------------------------------------------------
variable RAM_DATA:STD_LOGIC_VECTOR(31 DOWNTO 0);
begin
RAM_DATA:=(OTHERS=>'0');pixels_target_d<= pixels_target_q;
config_d<= (others=>(others=>'0')); OFFSET_i<= (others=>'0');
pixels_word_d <= (OTHERS=>'0'); pop_crnrs_i <= '0';
ofile_len_d <= ofile_len_q;
inc_rindex_i<='0';write_pixel_word<='0';
ifile_d<=ifile_q; ofile_d<=ofile_q;
  if resetn_i = '0' then
   reset_sys_call(kernel_call);
  else
   hal_asos_link_to_kernel(kernel_response,kernel_call);
   case task_state is
      when s1_query_ifile=>
          pooled_fstream_query(kernel_call,kernel_response,ifile_q, ifile_d);
      when s2_query_ofile=>
          pooled_fstream_query(kernel_call,kernel_response,ofile_q, ofile_d);
      when s3_query_conf=>
          transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
      when s4_read_config=>
          kernel_call.enable_index <= '1';
          kernel_call.increment_index <= '1';
          config_d(0)<=data_in_i(0);
          config_d(1)<=data_in_i(1);
          config_d(2)<=data_in_i(2);
          OFFSET_i<= std_logic_vector(to_unsigned(kernel_response.index,3));
          if kernel_response.index = (C_CONF_LEN/4)-1 then
            kernel_call.increment_index <= '0';
          end if;
      when s5_config_run=>
          config_d(0)<= x"00000080";
          OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
      when s6_read_file=>
         pooled_fstream_read_sysram(kernel_call,kernel_response, ifile_q, block_len,0);
         pixels_target_d <= cast_return_to_transfer_len(kernel_response);
      when s11_push_pixels=>
         safe_safe_read_sysram_word32_burst(kernel_call,kernel_response, pixels_word_d,
fifo_in_burst_q,index_read_q);
         write_pixel_word <= cast_return_to_push_data(kernel_response);
         inc_rindex_i<=cast_return_to_push_data(kernel_response);
      when s14_write_block=>
         safe_write_sysram_word32_burst(kernel_call,kernel_response,corners_word_i,
fifo_out_burst_q,index_write_q);
          pop_crnrs_i <= cast_return_to_pop_data(kernel_response);
          inc_windex_i <= cast_return_to_pop_data(kernel_response);
      when s18_fstream_write=>
          pooled_fstream_write_sysram(kernel_call, kernel_response,ofile_q,
count_crnr_bytes_q,32768);
          ofile_len_d <= cast_return_to_transfer_len(kernel_response);
      when s21_ write_block _last=>
         safe_write_sysram_word32_burst (kernel_call,kernel_response, corners_word_i,
fifo_out_burst_last_q,index_write_q);
          pop_crnrs_i <= cast_return_to_pop_data(kernel_response);
          inc_windex_i <= cast_return_to_pop_data(kernel_response);
```

back to Figure 5.18.

Listing C.45: Extended features for the standalone synchronous Control unit of the HW accelerated feature detection using the SYSRAM(2/2).

```
      when s25_fstream_write_last=>
        pooled_fstream_write_sysram(kernel_call, kernel_response, ofile_q,
count_crnr_bytes_q,32768);
        ofile_len_d <= cast_return_to_transfer_len(kernel_response);
      when s26_stop_fast_nms =>
         config_d(0)<=(others=>'0');
         OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
      when s27_write_message=>
         safe_write_lram(kernel_call,kernel_response,fmessage,
std_logic_vector(to_unsigned(count_crnrs_q,32) & to_unsigned(pixel_block_counter,32)),0);
      when s90_print_stdio=>
          write_stdio(kernel_call, kernel_response, fmessage'high,m_len,0);
      when s99_exit=>
         task_exit(kernel_call, kernel_response);
      when others=> null;
    end case;
  end if;
 end process EXTENEDED_FEATURES;
---------------------------------------------------------------------------
```

back to Figure 5.18.

Listing C.46: Control FSM VHDL description for the FastSA HW-Task (1/2).

```
-----------------------------------------------------------------------------------------
CONTROL_FSM: process(fast_state,s00_kernel_run ,blen_param_q, kernel_response,
file_done_q, line0_space_Q, w_address_offset_q, space_available_q,pixels_target_q,
pixel_counter_q, hw_fast_done_q)
-------------------------------------------------------------------------------------------
begin
task_done_i<= '0'; task_state_next <= fast_state; WR_CE_i <= '0';
w_address_offset_d <= w_address_offset_q; hw_fast_done_d <= hw_fast_done_q;
inc_pixel_block_counter<='0';clr_pixel_counter_i <= '0'; load_rindex_i<='0';
case fast_state is
   when s0_ready=>
     hw_fast_done_d <= '0';
     task_state_next <= s0_ready;
     if s00_kernel_run = '1' then
       task_state_next <= s1_query_ifile;
     end if;
   when s1_query_ifile=>
     task_state_next <= s2_query_conf;
   when s2_query_conf=>
     task_state_next <= s3_read_config;
   when s3_read_config=>
     task_state_next <=s3_read_config;
     WR_CE_i <= '1';
     if kernel_response.index = C_CONF_LEN/4-1 then
       task_state_next <=s4_config_run;
     end if;
   when s4_config_run=>
     WR_CE_i <= '1';
     inc_pixel_block_counter<= '1';
     task_state_next <=s5_async_read_fstream_0;
   when s5_async_read_fstream_0=>
     w_address_offset_d <= blen_param_q;
     task_state_next <= s6_async_fstream_read;
   when s6_async_fstream_read=>
     task_state_next <= s7_fin_fstream_read;
   when s7_fin_fstream_read=>
     clr_pixel_counter_i <='1';
     task_state_next <= s8_eval_fread;
```

Listing C.47: Control FSM VHDL description for the FastSA HW-Task (2/2).

```vhdl
    when s8_eval_fread=>
      inc_pixel_block_counter<= '1';
      task_state_next <= s9_handshake_dpath;
      if (pixels_target_q = 0) then
        task_state_next <=s14_exhausted_file;
      end if;
    when s9_handshake_dpath=>
      task_state_next <= s12_wait_space;
      if space_available_q = '1' then
        task_state_next <= s10_write_pixels;
      end if;
    when s10_write_pixels=>
      task_state_next <= s11_check_pixel_target;
    when s11_check_pixel_target =>
      task_state_next <= s12_wait_space;
      if(pixel_counter_q >= pixels_target_q) then
        task_state_next <= s13_update_index;
      elsif  space_available_q = '1' then
        task_state_next <= s9_handshake_dpath;
      end if;
    when s12_wait_space=>
      task_state_next <=s11_push_pixels;
    when s13_update_index =>
      load_rindex_i<='1';
      task_state_next <= s6_async_fstream_read;
      if file_done_q = '1' then
        task_state_next <= s7_fin_fstream_read;
      end if;
    when s14_exhausted_size=>
      if(to_integer(unsigned(line0_space_Q)) = CRAM_WIDTH) then
        task_state_next <= s15_stop_fast_nms;
      end if;
    when s15_stop_fast_nms =>
      hw_fast_done_d <= '1';
      task_state_next <= s16_write_message;
    when s16_write_message=>
      task_state_next <= s90_print_stdio;
    when s90_print_stdio=>
      task_state_next <= s99_exit;
    when s99_exit=>
      task_done_i<= '1';
      task_state_next <=s99_exit;
    when others=> null;
  end case;
  end process CONTROL_FSM;
  ------------------------------------------------------------------------------
```

back to Figure 5.23.

Listing C.48: Extended features VHDL description for the MFastSA HW-Task.

```vhdl
-------------------------------------------------------------------------------------
EXTENEDED_FEATURES: process(fast_state,  data_in_i, blen_param_q, ifile_q,
kernel_response, w_address_offset_q, burst_target_q, space_available_q, kernel_call,
pixel_block_counter, pixels_target_q, index_read_q, total_pixel_counter_q)
-------------------------------------------------------------------------------------
begin
is_event_d <= false; pixels_target_d<= pixels_target_q;
config_d<= (others=>(others=>'0')); pixels_word_d <= (OTHERS=>'0');
OFFSET_i<= (others=>'0'); inc_rindex_i<='0';
write_pixel_word_i <= '0';ifile_d<=ifile_q;
if resetn_i = '0' then
 reset_sys_call(kernel_call);
else
 hal_asos_link_to_kernel(kernel_response,kernel_call);
case fast_state is
    when s1_query_ifile=>
      pooled_fstream_query(kernel_call,kernel_response,ifile_q, ifile_d);
    when s2_query_conf=>
      transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
    when s3_read_config=>
      kernel_call.enable_index <= '1';
      kernel_call.increment_index <= '1';
      config_d(0)<=data_in_i(0);
      config_d(1)<=data_in_i(1);
      config_d(2)<=data_in_i(2);
      OFFSET_i<= std_logic_vector(to_unsigned(kernel_response.index,3));
      if kernel_response.index = C_CONF_LEN/4-1 then
        kernel_call.increment_index <= '0';
      end if;
   when s4_config_run=>
      config_d(0)<= x"00000080";
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s5_async_read_fstream_0=>
     async_pooled_fstream_read_sysram(kernel_call,kernel_response, ifile_q,
blen_param_q,0);
   when s6_async_fstream_read=>
     async_pooled_fstream_read_sysram(kernel_call, kernel_response, ifile_q,
blen_param_q, w_address_offset_q);
   when s7_fin_fstream_read=>
     async_finalize_pooled_fstream_read_sysram(kernel_call,kernel_response);
     pixels_target_d <= cast_return_to_transfer_len(kernel_response);
   when s10_push_pixels=>
     unsafe_safe_read_sysram_word32_burst(kernel_call,kernel_response,pixels_word_d,
burst_target_q,index_read_q);
     write_pixel_word_i <= cast_return_to_push_data(kernel_response);
     inc_rindex_i<=cast_return_to_push_data(kernel_response);
   when s12_wait_space=>
     wait_signal_event(kernel_call,kernel_response,space_available_q,is_event_d,0);
   when s15_stop_fast_nms =>
      config_d(0)<=(others=>'0');
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s16_write_message=>
     safe_write_lram(kernel_call,kernel_response,fmessage,
        std_logic_vector(to_unsigned(total_pixel_counter_q,32) &
            to_unsigned(pixel_block_counter,32)),0);
   when s90_print_stdio=>
     write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
   when s99_exit=>
     task_exit(kernel_call, kernel_response);
   when others=> null;
end case;
end if;
end process EXTENEDED_FEATURES;
-----------------------------------------------------------------------
```

back to Figure 5.24.

Listing C.49: Extended features VHDL description for the FastSA HW-Task.

```
-----------------------------------------------------------------------------------------
EXTENEDED_FEATURES: process(fast_state,  data_in_i, blen_param_q, ifile_q,
kernel_response, w_address_offset_q, burst_target_q, space_available_q, kernel_call,
pixel_block_counter, pixels_target_q, index_read_q, total_pixel_counter_q)
-----------------------------------------------------------------------------------------
variable RAM_DATA:STD_LOGIC_VECTOR(31 DOWNTO 0);
begin
RAM_DATA:=(OTHERS=>'0');is_event_d <= false; pixels_target_d<= pixels_target_q;
config_d<= (others=>(others=>'0')); pixels_word_d <= (OTHERS=>'0');
OFFSET_i<= (others=>'0'); inc_rindex_i<='0';
write_pixel_word_i <= '0';ifile_d<=ifile_q;
if resetn_i = '0' then
 reset_sys_call(kernel_call);
else
 hal_asos_link_to_kernel(kernel_response,kernel_call);
case fast_state is
    when s1_query_ifile=>
      pooled_fstream_query(kernel_call,kernel_response,ifile_q, ifile_d);
    when s2_query_conf=>
      transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
    when s3_read_config=>
      kernel_call.enable_index <= '1';
      kernel_call.increment_index <= '1';
      config_d(0)<=data_in_i(0);
      config_d(1)<=data_in_i(1);
      config_d(2)<=data_in_i(2);
      OFFSET_i<= std_logic_vector(to_unsigned(kernel_response.index,3));
      if kernel_response.index = C_CONF_LEN/4-1 then
        kernel_call.increment_index <= '0';
      end if;
   when s4_config_run=>
      config_d(0)<= x"00000080";
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s5_async_read_fstream_0=>
     async_pooled_fstream_read(kernel_call,kernel_response, ifile_q,
blen_param_q,0);
   when s6_async_fstream_read=>
     async_pooled_fstream_read(kernel_call, kernel_response, ifile_q,
blen_param_q, w_address_offset_q);
   when s7_fin_fstream_read=>
     async_finalize_pooled_fstream_read(kernel_call,kernel_response);
     pixels_target_d <= cast_return_to_transfer_len(kernel_response);
   when s10_push_pixels=>
     lram_read_word_burst(kernel_call,kernel_response,RAM_DATA, burst_target_q,
index_read_q);
     pixels_word_d<= RAM_DATA;
     write_pixel_word_i <= cast_return_to_push_data(kernel_response);
     inc_rindex_i<=cast_return_to_push_data(kernel_response);
   when s12_wait_space=>
     wait_signal_event(kernel_call,kernel_response,space_available_q,is_event_d,0);
   when s15_stop_fast_nms =>
      config_d(0)<=(others=>'0');
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
   when s16_write_message=>
     safe_write_lram(kernel_call,kernel_response,fmessage,
        std_logic_vector(to_unsigned(total_pixel_counter_q,32) &
            to_unsigned(pixel_block_counter,32)),0);
   when s90_print_stdio=>
     write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
   when s99_exit=>
     task_exit(kernel_call, kernel_response);
   when others=> null;
end case;
end if;
end process EXTENEDED_FEATURES;
-------------------------------------------------------------------------
```

back to Figure 5.24.

Listing C.50: Asynchronous control unit for the NonmaxSA HW-Task(1/2).

```
-----------------------------------------------------------------------------------------
CONTROL_FSM: process(nms_state,s00_kernel_run ,Run_q,kernel_response, count_crnr_bytes_q,
fifo_size_block_q,fifo_out_burst_q, ofile_len_q,i_fast_done,pending_transfer_q,
wake_control_q)
------------------------------------------------------------------------------------------
begin
task_done_i<= '0';task_state_next <= nms_state;WR_CE_i <= '0';
pop_crnrs_i <= '0';load_windex_i<='0';Run_d <= Run_q;
pending_transfer_d <= pending_transfer_q;
case nms_state is
   when s0_ready=>
     task_state_next <= s0_ready;
     load_windex_i <= '1';
     if s00_kernel_run = '1' then
       task_state_next <= s2_query_ofile;
     end if;
   when s1_query_ofile=>
     task_state_next <=s2_query_conf;
   when s2_query_conf=>
     task_state_next <= s3_read_config;
   when s3_read_config=>
     WR_CE_i <= '1';
     if kernel_response.index = C_CONF_LEN/4-2 then
         task_state_next <=s4_config_run;
     end if;
   when s4_config_run=>
     Run_d <= '1';
     WR_CE_i <= '1';
     task_state_next <=s10_fifo_out_check_size;
   when s5_check_crnrs_size=>
     task_state_next <= s9_check_upld_tgt;
     if fifo_size_block_q >= fifo_out_burst_q  and  fifo_out_burst_q > 0 then
       task_state_next <= s7_write_corners;
     end if;
   when s7_write_corners=>
     task_state_next <= s15_check_corners;
   when s9_check_upld_tgt=>
     task_state_next <= s13_check_done;
     if count_crnr_bytes_q >= upload_crnr_bytes_q then
       task_state_next <= s10_fstream_async_write;
     end if;
   when s10_fstream_async_write=>
     load_windex_i <= '1';
     inc_pending <= '1';
     if pending_transfer_q = 1 then
       task_state_next <= s11_fstream_finalz_write;
     else
       task_state_next<= s13_check_done;
     end if;
   when s11_fstream_finalz_write=>
     task_state_next <= s12_eval_fstream_write;
   when s12_eval_fstream_write=>
     task_state_next <=  s13_check_done;
     dec_pending <= '1';
     if ofile_len_q = 0 then
       task_state_next <= s17_stop_fast_nms;
     end if;
   when s13_check_done=>
     if i_fast_done = '1' and fifo_size_block_q = 0 then
       task_state_next <= s15_check_corners_last;
     elsif wake_control_q = '1' then
       task_state_next <= s5_check_crnrs_size;
     else
       task_state_next <= s14_wait_corners;
     end if;
   when s14_wait_corners=>
     task_state_next <= s5_check_crnrs_size;
```

Listing C.51: Asynchronous control unit for the NonmaxSA HW-Task(2/2).

```
    when s15_fstream_finwrite=>
      dec_pending<='1';
      task_state_next <= s16_stop_fast_nms;
    when s16_stop_fast_nms =>
      WR_CE_i <= '1';
      task_state_next <= s17_write_message;
    when s17_write_message=>
      task_state_next <= s90_print_stdio;
    when s90_print_stdio=>
      task_state_next <= s99_exit;
    when s99_exit=>
      task_done_i<= '1';
    when others=> null;
end case;
end process CONTROL_FSM;
--------------------------------------------------------------------------
```

back to Figure 5.23

Listing C.52: Asynchronous Extended Features the NonmaxSA HW-Task(1/2).

```
-----------------------------------------------------------------------------------------
EXTENDED_FEATURES: process(nms_state, data_in_i,kernel_response, remaining_space_q,
count_crnr_bytes_q, fifo_out_burst_q, index_write_q, count_crnrs_q, ofile_q,
corners_word_i,wake_control_q, upload_crnr_bytes_q, kernel_call)
------------------------------------------------------------------------------------------
begin
config_d     <= (others=>(others=>'0'));pixels_word_d <= (OTHERS=>'0');
OFFSET_i<= (others=>'0');pop_crnrs_i <= '0';
ofile_d <= ofile_q;
if resetn_i = '0' then
  reset_sys_call(kernel_call);
else case nms_state is
    when s1_query_ofile=>
      pooled_fstream_query(kernel_call,kernel_response,ofile_q, ofile_d);
    when s2_query_conf=>
      transfer_from_host_swfifo(kernel_call,kernel_response,C_CONF_LEN);
    when s3_read_config=>
      kernel_call.enable_index <= '1';
      kernel_call.increment_index <= '1';
      config_d(0)<=data_in_i(0);
      config_d(1)<=data_in_i(1);
      config_d(2)<=data_in_i(2);
      OFFSET_i<= std_logic_vector(to_unsigned(kernel_response.index,3));
      if kernel_response.index = C_CONF_LEN/4-2 then
          kernel_call.increment_index <= '0';
      end if;
    when s4_config_run=>
      config_d(0)<= x"00000080";
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
    when s7_write_corners=>
      unsafe_write_sysram_word32_burst(kernel_call,kernel_response,corners_word_i,
fifo_out_burst_q,index_write_q);
      pop_crnrs_i <= cast_return_to_pop_data(kernel_response);
      inc_windex_i <= cast_return_to_pop_data(kernel_response);
    when s11_fstream_finalz_write=>
      async_finalize_pooled_fstream_write_sysram(kernel_call, kernel_response);
      ofile_len_d <= cast_return_to_transfer_len(kernel_response);
    when s14_wait_corners=>
      wait_signal_event(kernel_call,kernel_response,wake_control_q,is_event_d,0);
    when s15_fstream_finwrite=>
      async_finalize_pooled_fstream_write(kernel_call, kernel_response);
      ofile_len_d <= cast_return_to_transfer_len(kernel_response);
```

Listing C.53: Asynchronous Extended Features the NonmaxSA HW-Task(2/2).

```
    when s16_stop_fast_nms =>
      config_d(0)<=(others=>'0');
      OFFSET_i<= std_logic_vector(to_unsigned(CONTROL_OFFSET,OFFSET_i'length));
    when s17_write_message=>
      safe_write_lram(kernel_call,kernel_response,fmessage,
std_logic_vector(to_unsigned(count_crnrs_q,32)),0);;
    when s90_print_stdio=>
      write_stdio(kernel_call, kernel_response,fmessage'high,m_len,0);
    when s99_exit=>
      task_exit(kernel_call, kernel_response);
    when others=> null;
end case; end if;
end process EXTENDED_FEATURES;
--------------------------------------------------------------------------
```

back to Figure 5.24

Listing C.54: Network changes to asynchronous Extended Features the NonmaxSA.

```
--------------------------------------------------------------------------------------------
EXTENDED_FEATURES: process(nms_state, ..., trfr_len_q , psocket_q ,kernel_call)
--------------------------------------------------------------------------------------------
begin
...
psocket_d<=psocket_q;
trfr_len_d<= trfr_len_q;
...
   when s1_query_ofile=>
     pooled_socket_query (kernel_call,kernel_response, psocket_d, psocket_q);
...
   when s11_fstream_finalz_write=>
     async_finalize_pooled_socket_write_sysram (kernel_call, kernel_response);
     trfr_len_d <= cast_return_to_transfer_len(kernel_response);
...
   when s15_fstream_finwrite=>
     async_finalize_pooled_socket_write_sysram(kernel_call, kernel_response);
     ofile_len_d <= cast_return_to_transfer_len(kernel_response);
...
end process EXTENDED_FEATURES;
--------------------------------------------------------------------------
```

Listing C.55: Standalone Dual-Task network-based Feature Detection.

```cpp
void hal_asos_demo::feature_detector::
test_fast_detector_std_alone_dual_master_net(void){
    using namespace hal_asos;
    bool status; int ret; FrameControl iFrame;
    std::string s, header0, header1, header2, version, width, height;
    std::stringstream ss;
    hal_asos::networking::CSocket<hal_asos::networking::Client> Soc;
    Task<HwTask, MTFastSA,segment_len<(BLOCK_LEN << 1)>>T1;
    Task<HwTask, MTNonmaxNet>T2;
    Soc.set_ip_address(hal_asos_demo::feature_detector::image_ip);
    Soc.set_sock_type(SOCK_STREAM);
    Soc.set_sock_family(AF_INET);
    Soc.set_sock_port(IMAGE_PORT_NO);
    status = Soc.open_connection();
    if (!status) {
        Soc.get_error_message(hal_asos_demo::feature_detector::image_ip);
        Soc.close_connection();
        LOG_MSG << hal_asos_demo::feature_detector::image_ip << "\n";
        return;
    }
    std::shared_ptr<StreamData> Conf = std::make_shared<StreamData>(16);
    detector::config_words* p_config = (detector::config_words*) Conf.get();

    iFrame.top = TOPSYMBOL;
    std::copy_n(scene_img.c_str(),scene_img.length(), iFrame.filename);
    iFrame.filename[hal_asos_demo::feature_detector::scene_img.length()] = 0;
    iFrame.th = FEATURE_THRESHOLD;
    iFrame.block_len = CORNER_LEN;
    iFrame.delimitor = CONTROLSYMBOL;
    Soc.safe_write((char*)&iFrame, sizeof(struct FrameControl));

    CFstream<std::ifstream> Input_file(scene_img.c_str());
    Input_file.set_flags(std::ios::in | std::ifstream::binary);
    if (Input_file.open_file() < 0) {
        LOG_MSG << "[" << __FUNCTION__ << "]:error opening input file!\n";
        return;
    }// First line : version
    Input_file.get_line(header0);
    if (header0.compare("P5") != 0) {
        LOG_MSG << "[" << __FUNCTION__ << "]:Wrong file format or version\n";
        return;
    }
    Input_file.get_line(header1);
    ss.str(header1);
    ss >> p_config->image_width >> p_config->image_height;
    Input_file.get_line(header2);
    ss.clear();
    p_config->threshould = FEATURE_THRESHOLD;
    p_config->block_len = block_size;
    T1.submit_to_pool(Input_file);
    T2.submit_to_pool(Soc);
    T1.submit_data(Conf);
    T2.submit_data(Conf);
    T1.start();
    T2.start();
    T1.join();
    T2.join();
    Input_file.close_file();
    Soc.safe_read((char*)&ret, 4);
    Soc.close_connection();
}
```

back to Figure 5.33.

Listing C.56: Ed.Rosten-C network-based Feature Detection (1/2).

```
void hal_asos_demo::feature_detector::
test_feature_detector_edrosten_c_mframe_net(void) {
    edrosten::xy* nonmax;
    int nonmax_len, index_pos = 0, ret = 1, coord_i, Read_len, pixel_len = 0;
    std::string TaskName = "[EdRosten-C Network]";
    bool status;
    hal_asos::networking::CSocket<hal_asos::networking::Client> Soc;
    std::ifstream input_file;
    std::string s, header0,header1, header2, version;
    FrameControl iFrame;
    std::stringstream lines, ss;
    int width, height, threshould, imlen;
    int file_pos , count_corners=0;
    std::shared_ptr<uint16_t[]> p_coord;
    std::shared_ptr<char[]> p_buff;

    Soc.set_ip_address(hal_asos_demo::feature_detector::image_ip);
    Soc.set_sock_type(SOCK_STREAM);
    Soc.set_sock_family(AF_INET);
    Soc.set_sock_port(IMAGE_PORT_NO);
    status = Soc.open_connection();
    if (!status) {
        Soc.get_error_message(hal_asos_demo::feature_detector::image_ip);
        Soc.close_connection();
        LOG_MSG << hal_asos_demo::feature_detector::image_ip << "\n";
        return;
    }
    iFrame.top = TOPSYMBOL;
    std::copy_n(scene_img.c_str(),scene_img.length(), iFrame.filename);
    iFrame.filename[scene_img.length()] = 0;
    iFrame.th = FEATURE_THRESHOLD;
    iFrame.block_len = CORNER_LEN;
    iFrame.delimitor = CONTROLSYMBOL;
    Soc.safe_write((char*)&iFrame, sizeof(struct FrameControl));

    input_file.open(scene_img.c_str(), std::ios::in | std::ifstream::binary);
    if (!input_file.is_open()) {
        LOG_MSG << TaskName << ":error opening input file!\n";
        return;
    }

    getline(input_file, header0);
    if (header0.compare("P5") != 0) {
        LOG_MSG << TaskName << "Wrong file format or version\n";
        return;
    }
    getline(input_file, header1);
    ss.str(header1);
    ss >> width >> height;
    imlen = width * height;
    getline(input_file, header2);
    ss.clear();
    p_buff = std::shared_ptr<char[]>(new char[imlen]);
    Read_len = imlen;    index_pos = 0;
    file_pos = 0;
    input_file.read(p_buff.get(), imlen);
    Read_len = (int)input_file.gcount();
    file_pos = Read_len;
    while (Read_len > 0) {
     nonmax = fast9_detect_nonmax((const edrosten::byte*)p_buff.get(), width, height,
width, th, &nonmax_len);
     if (nonmax_len) {
     p_coord = std::shared_ptr<uint16_t[]>(new uint16_t[nonmax_len << 1]);
     for(index_pos = 0,coord_i = 0; index_pos < nonmax_len; index_pos++,  coord_i += 2) {
         p_coord[coord_i] = (uint16_t)nonmax[index_pos].x;
         p_coord[coord_i + 1] = (uint16_t)nonmax[index_pos].y;
         count_corners++;
      }
```

Listing C.57: Ed.Rosten-C network-based Feature Detection (1/2).

```
        Soc.safe_write((char*)p_coord.get(), nonmax_len << 2);
        free(nonmax);

      }
      input_file.read(p_buff.get(), imlen);
      Read_len = (int)input_file.gcount();
      file_pos += Read_len;
     }

     lines.str("");

     input_file.close();
     Soc.safe_read((char*)&ret, 4);
     Soc.close_connection();
     LOG_MSG << TaskName << ":finished!...(" << file_pos << ", " << count_corners << ")\n";
     return;
}
```

back to Figure 5.34.

Appendix D

Auxiliary Figures

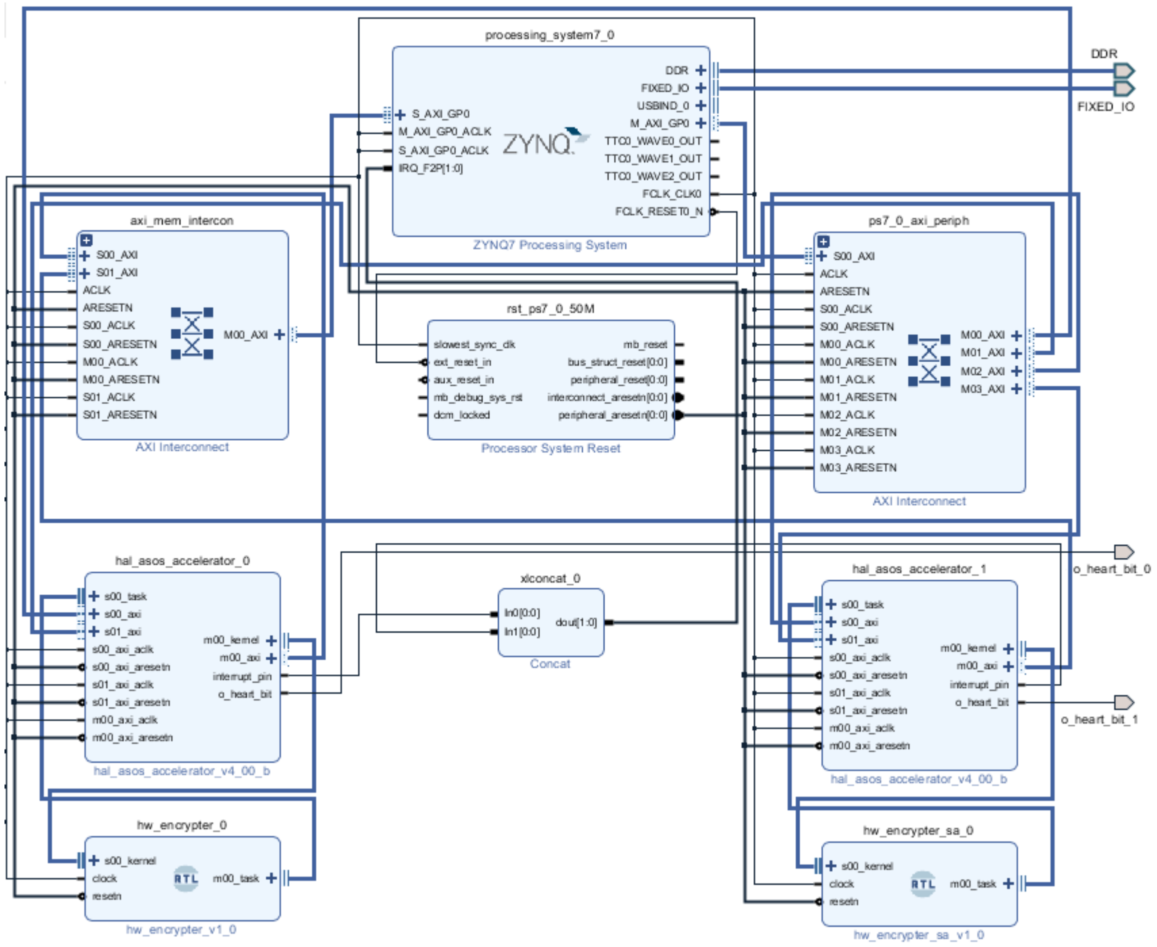
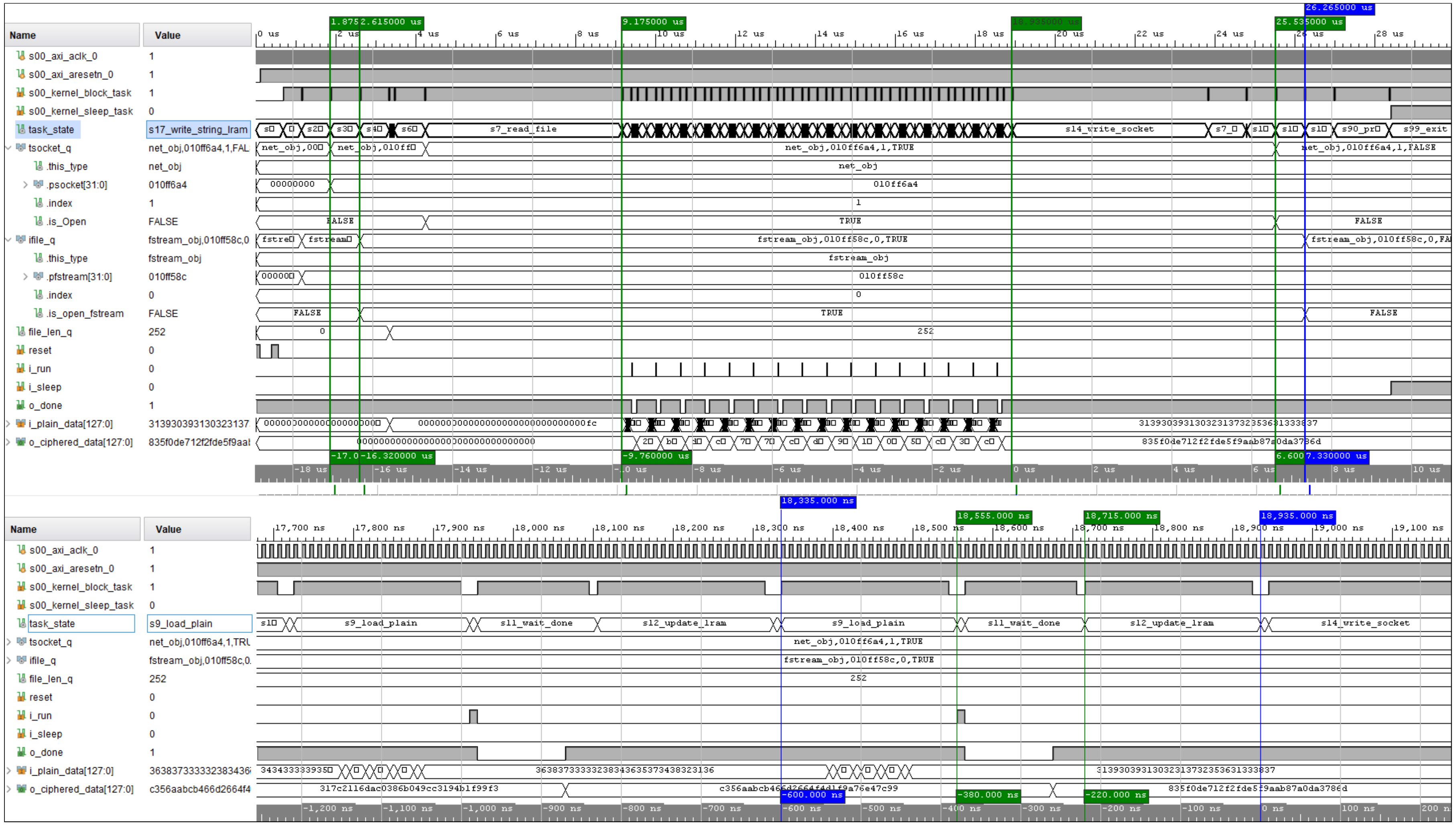


Figure D.1: Platform Deployment - Machine 1 block design using ZC702 board (Back to Figure 2.36).



Name	Value
s00_axi_aclk_0	1
s00_axi_aresetn_0	1
s00_kernel_block_task	1
s00_kernel_sleep_task	0
task_state	s17_write_string_lram
tsocket_q	net_obj,010ff6a4,1,FAL
.this_type	net_obj
.psocket[31:0]	010ff6a4
.index	1
.is_Open	FALSE
ifile_q	fstream_obj,010ff58c,0
.this_type	fstream_obj
.pfstream[31:0]	010ff58c
.index	0
.is_open_fstream	FALSE
file_len_q	252
reset	0
i_run	0
i_sleep	0
o_done	1
i_plain_data[127:0]	313930393130323137
o_ciphered_data[127:0]	835f0de712f2fde5f9aal

Name	Value
s00_axi_aclk_0	1
s00_axi_aresetn_0	1
s00_kernel_block_task	1
s00_kernel_sleep_task	0
task_state	s9_load_plain
tsocket_q	net_obj,010ff6a4,1,TRU
ifile_q	fstream_obj,010ff58c,0,
file_len_q	252
reset	0
i_run	0
i_sleep	0
o_done	1
i_plain_data[127:0]	363837333332383436
o_ciphered_data[127:0]	c356aabcb466d2664f4

Figure D.2: Co-Simulation - Standalone HW Encryptor signals using Vivado simulator (back to Figure 2.35)

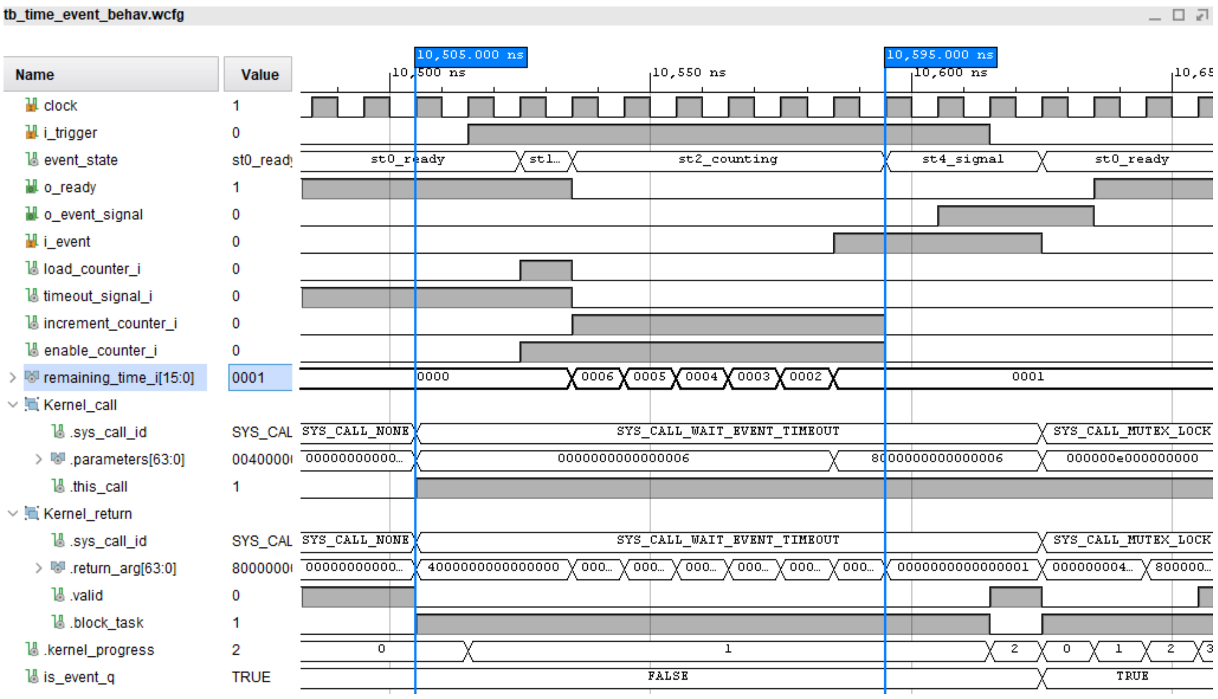


Figure D.3: Time Event unit - Wait for a signal event using 6 as timeout parameter, not exhausted.

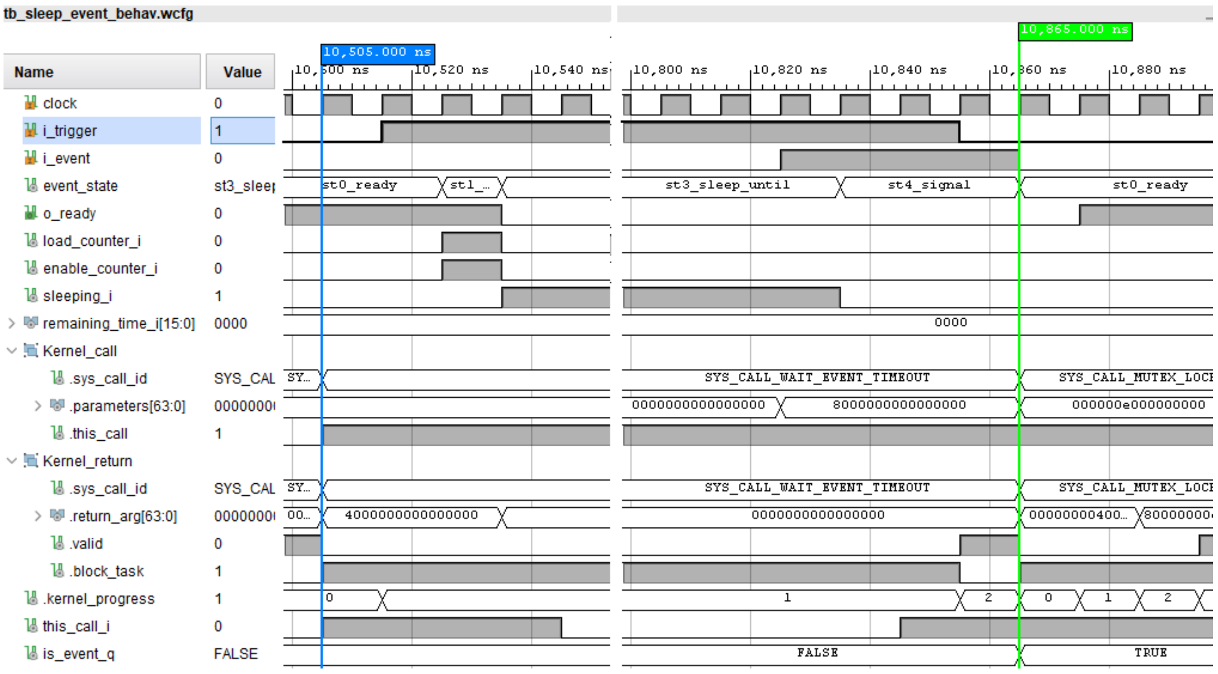


Figure D.4: Time Event unit - Wait for a signal event using no timeout.

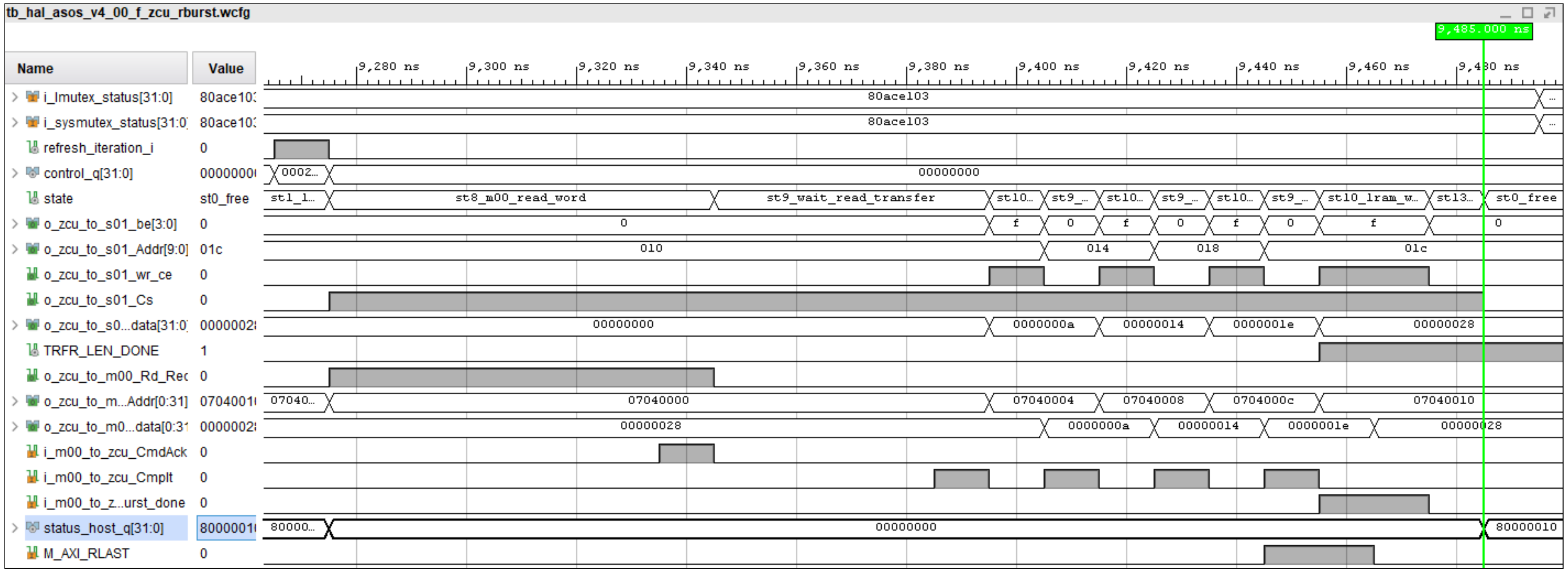


Figure D.5: ZeroCopy unit wave diagram - Read system memory and write S01 interface in burst format.

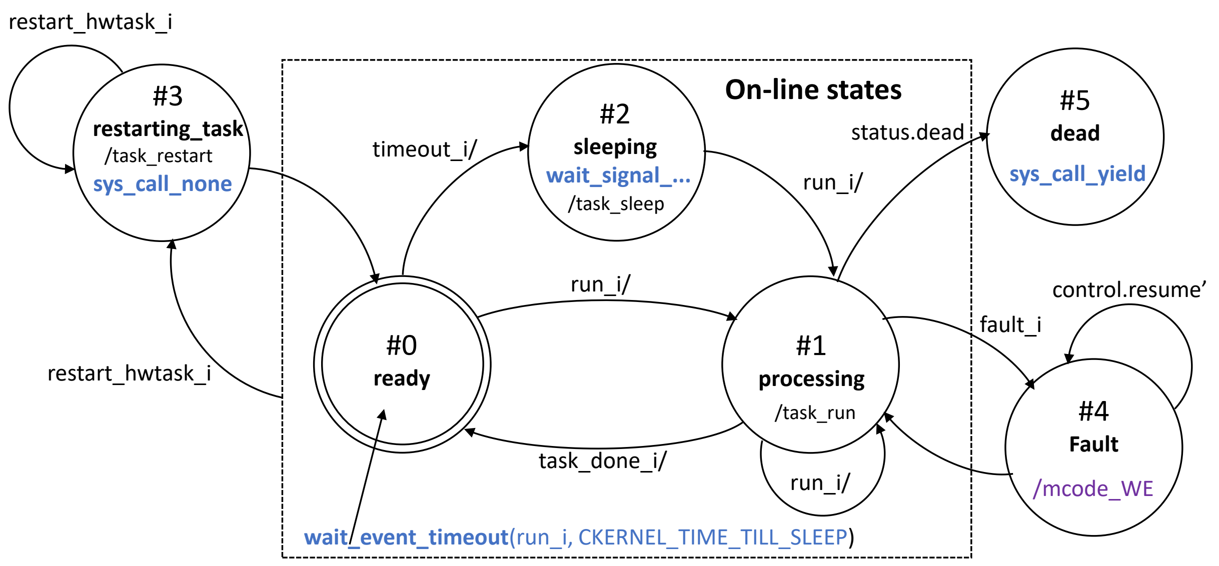


Figure D.6: Kernel Core - FSM state diagram for the RAM-based microprogram.

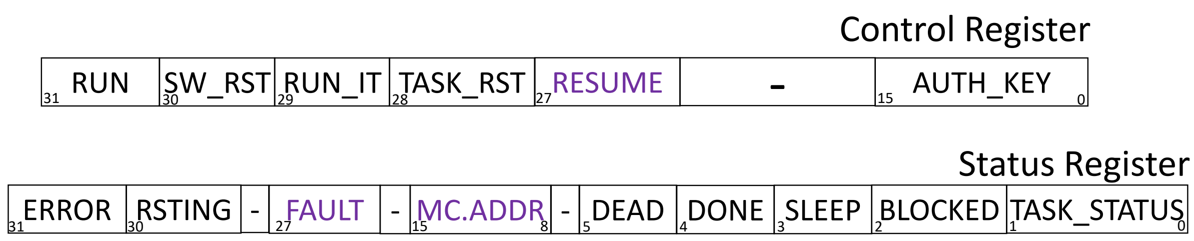


Figure D.7: Kernel Core - Control and status registers for the RAM-based microprogram.

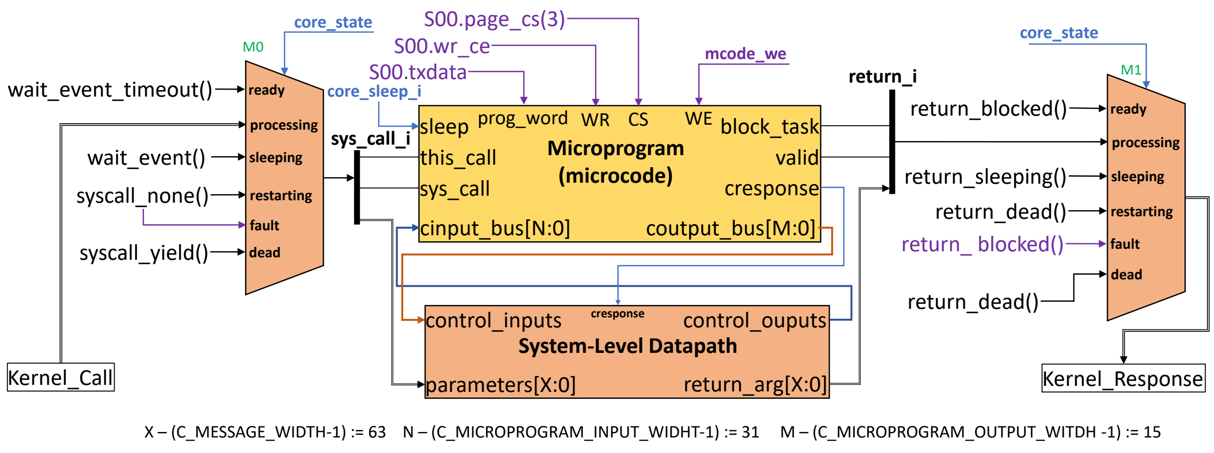


Figure D.8: Kernel Core - system-level datapath and microprogram interaction for the RAM-based microprogram.

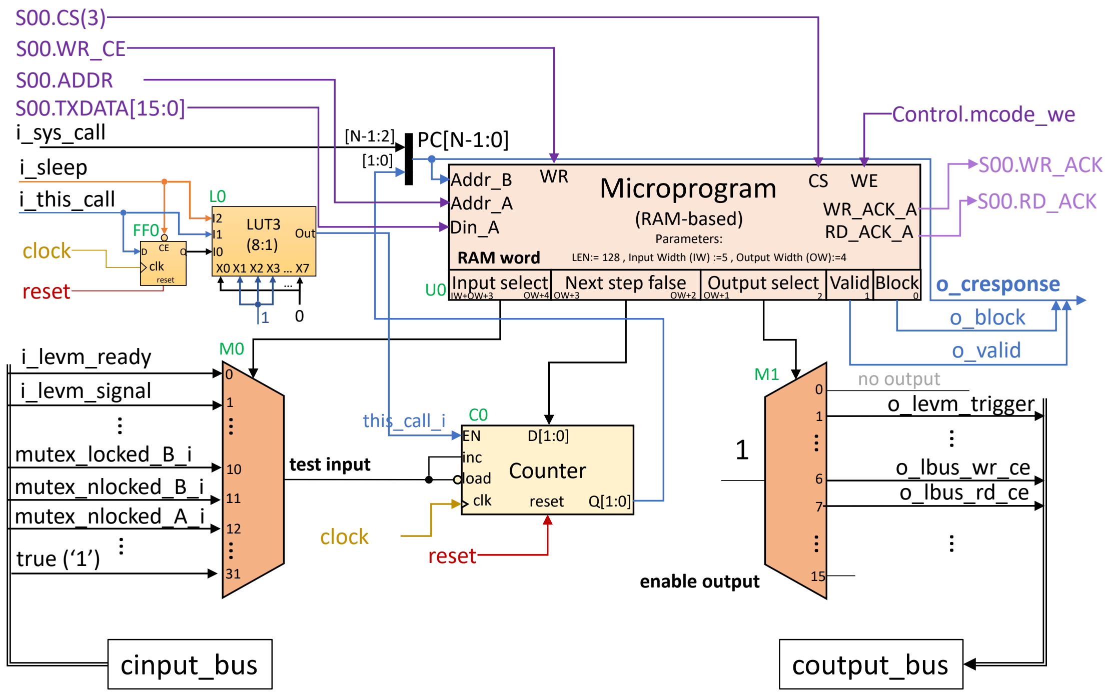
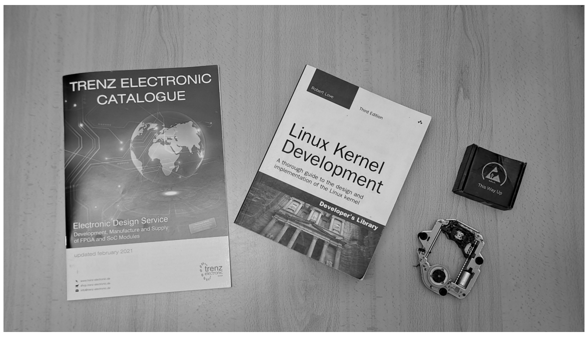
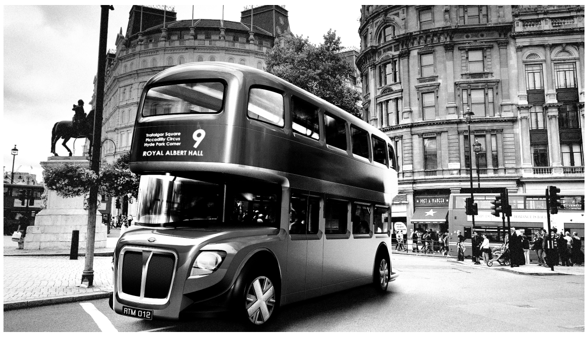


Figure D.9: Microprogram - RAM-based internal architecture.

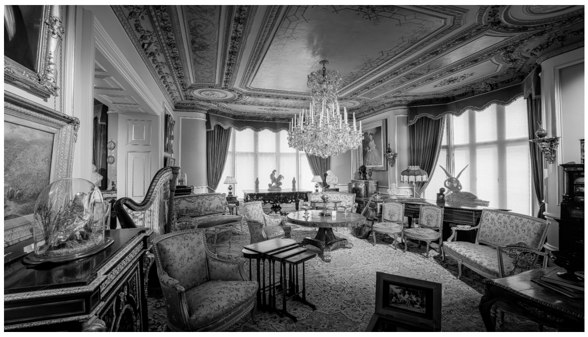


(a) filename:table.pgm 4600 corners.

(b) filename:turkey.pgm 33700 corners.



(c) filename:london.pgm 61700 corners.



(d) filename:jean.pgm 90000 corners.

Figure D.10: Feature detection test images dataset.

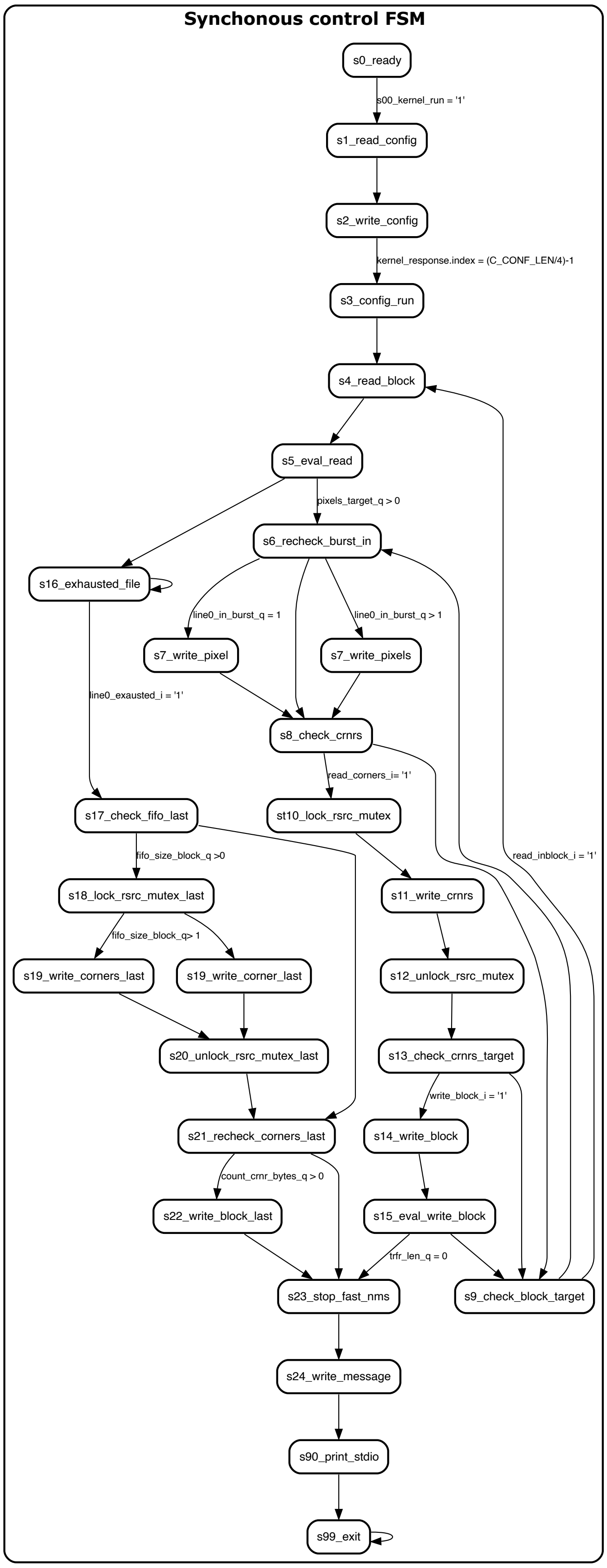


Figure D.11: Synchronous control unit for the multithread-based HW-Task design.

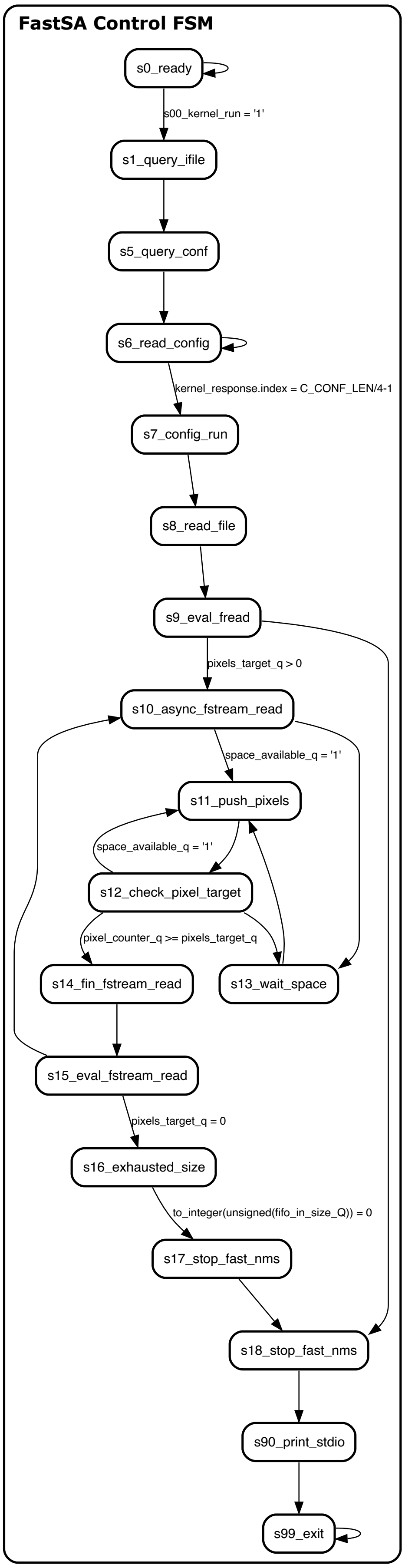


Figure D.12: Synchronous control unit for the FastSA HW-Task.

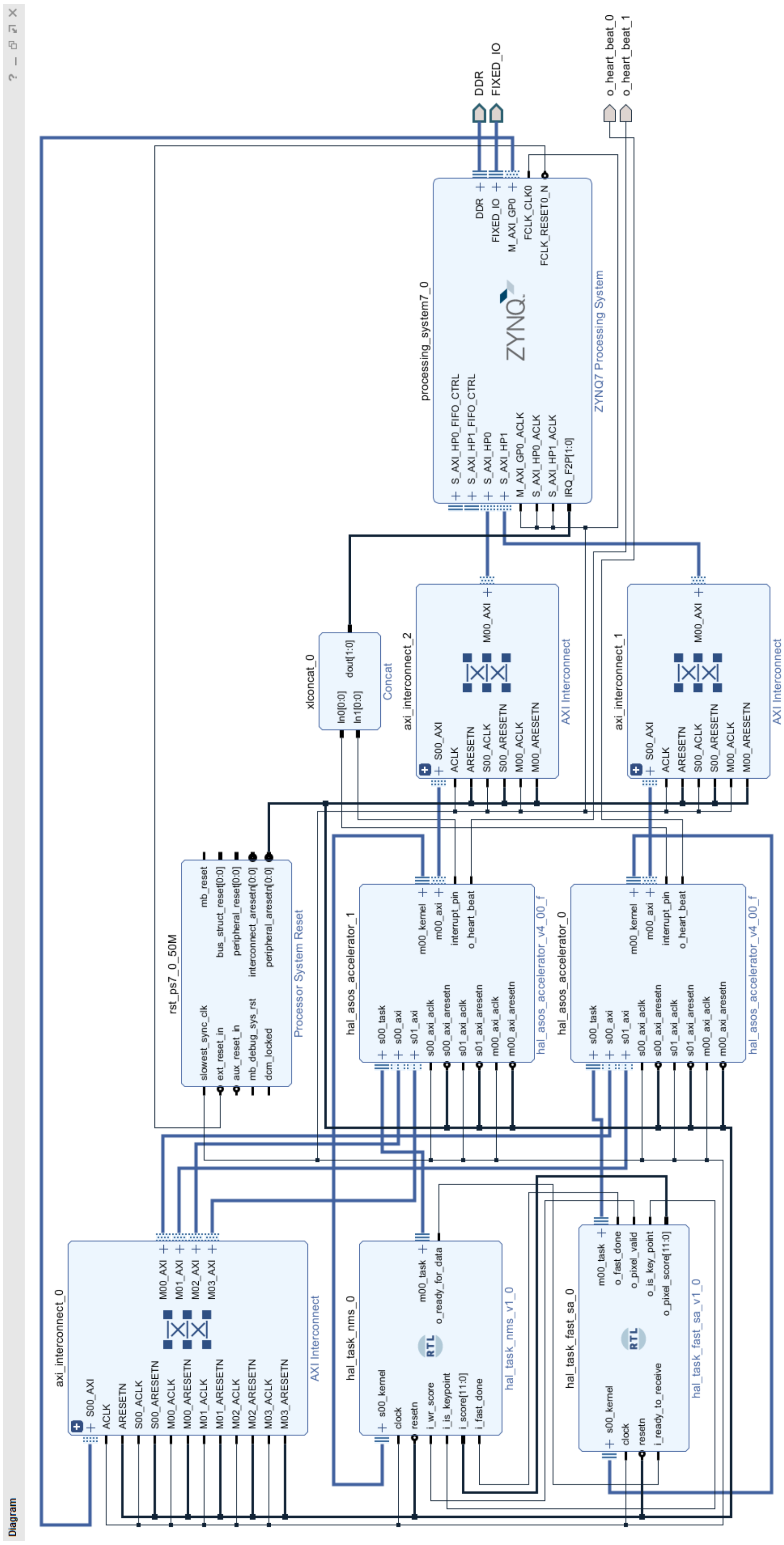


Figure D.13: Block design for the Dual-Task Asynchronous design targeting ZC702 platform.

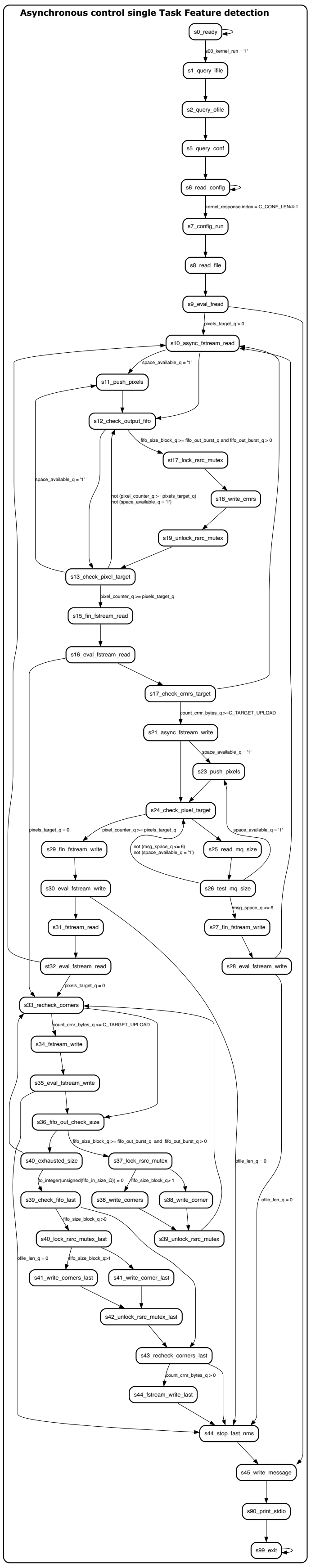


Figure D.14: Asynchronous control unit for the single Task feature detection.

Appendix E

AES Implementation

In this appendix, the Advanced Encryption Standard (AES) citeAES2011 is described together with discussion on implementation details of AES algorithm, as part of the applications used in Chapter 2. The chapter starts with a brief motivation and a moderate description introduces the AES structure and provides a simplified view about the several layers that implement the encryption and decryption processes. Among the different AES variants, the 128-bit using the Electronic Code Book (ECB) mode was selected, and proceed in the comprehensive description, while comparing it with details of this thesis. The description is supported with software in C/C++ language examples, for the encryption and decryption processes, and then complemented with the similar implementation for the hardware encryption using VHDL. The appendix concludes describing two implementation strategies for the hardware AES. The first is constrained by moderate use of logic resources, while the second is more throughput concerned and uses a full pipelined design strategy.

Up until 1977 the digital encryption was accomplished using the Digital Encryption Standard (DES) that was introduced as U.S. federal standard in 1976. Conceptually, DES is a block cipher that operates on 64-bit words and uses a 56-bit cipher which represents 2^{56} possible keys. Although a large number (72.057.594.037.927.936), a so-called DES challenges, with sufficient number of computational resources connected to the internet and exhaustively searching the key space, demonstrated this weakness dramatically. In 1997 the U.S. National Institute of Standards and Technology (NIST) created a public competition in order to find a replacement algorithm for the DES. The original requirements demanded for a block cipher supporting cipher key lengths of 128, 192 e 256 bits, and in August of 1999, 15 algorithms were accepted. In April 2000, the *Rijndael* algorithm was selected and six months later the NIST announced that the cipher was chosen as the Advanced Encryption Standard (AES). The NIST standard specifies five operation modes for the AES: Electronic CodeBook mode (ECB); Cipher Block Chaining mode (CBC); Cipher FeedBack mode (CFB); Output FeedBack mode (OFB); and Counter mode. Only two of these require the Inverse AES operation for decryption, the ECB and the CBC. The reminder modes use the same stream of pseudo-random sequence to encrypt and decrypt the plain data.

E.1 Computations in the AES

In this section we provide a mathematical overview of the numeric systems used in the AES. The description is limited to the computations that need to be implemented. In mathematics, the Finite Field or Galois Field (GF), is a field that contains a finite number of elements. Like any regular field (such as $\mathbb{R}$ or $\mathbb{C}$), finite or not, it must define the four arithmetical operations: addition, subtraction, multiplication and inverse (or division). By definition, the finite field $GF(p^m)$, is a field defined by p, a prime number, and m a positive integer. When m is 1, the $GF(p)$, is a prime field by definition, where the elements are integers numbers in the finite range $\{0, 1, ..., p-1\}$. The four arithmetic operations are defined by the following axioms. Let $a, b \in GF(p) = \{0, 1, ..., p-1\}$,

$$Addition: \; a + b = c \, mod \, p \tag{D.1}$$

$$Subtraction: \; a - b = d \, mod \, p \tag{D.2}$$

$$Multiplication: \; a.b = e \, mod \, p \tag{D.3}$$

$$Inverse: \; a.a^{-1} = 1 \, mod \, p \tag{D.4}$$

The final results $c,\ d$ and e, are numbers in the field, and result from the modulo reduction using p, the prime number. Example : $GF(7)$: $5 + 6 = 4 \; modulo \; 7$, where $4 = 11\%7$ (reminder in division).

The first three operations are easily computed in any microcontroller. The most complex is the inverse of a, or (a^{-1}), that can be computed using the Extended Euclidean algorithm.

In the AES the fields $p = 2$ are of particular interest, and one important field is the $GF(2)$ with $(m = 1)$, the prime field that contains the two elements $\{0, 1\}$. The particular case of prime field $GF(2)$ is that the addition and subtraction operations produce the same result:

$$\{1\} + \{1\} = \{0\} \text{ and } \{1\} - \{1\} = \{0\}$$

In reality $\{1\} + \{1\} = \{10\}$, but the result is not in the field of $GF(2)$. We then apply a modulo reduction using the prime 2 (in binary $\{10\}$) and the result is 0.

When compared, the above results are equivalent to the XOR operation, and so it can be used instead:

$$\{1\} + \{0\} = \{0\} - \{1\} = \{1\}mod\{10\} \;\Leftrightarrow\; \{1\} \oplus \{0\} = \{0\} \oplus \{1\} = \{1\}$$

In the multiplication, $\{0\}$ is the element zero and we multiply $\{1\}$ with simple rotate left, and apply the modulo reduction using the prime $\{10\}$. Similarly, the division can be implemented by a rotate right.

The extended finite fields occur when $m > 1$, where the elements are polynomials defined by the expression:

$$GF(2^m) = a_{(m-1)}.X^{(m-1)} + ... + a_1.X^1 + a_0, \;\; a_i \in GF(2) = \{0, 1\} \tag{D.5}$$

The AES uses the extended finite field set with $m = 8$, the $GF(2^8)$, and therefore 256 polynomials exist. In each of the polynomials, the coefficients a_i are the elements in the $GF(2)$ (e.g. $\{0, 1\}$), and so they are written using the following expression:

Let $A(x) \in GF(2^8)$,

$$A(x) = a_7X^7 + a_6X^6 + a_5X^5 + a_4X^4 + a_3X^3 + a_2X^2 + a_1X + a0 \tag{D.6}$$

$$\text{and } \{a_7, ..., a_0\} \in \{0, 1\}$$

To compute the four arithmetic operations described above, the AES uses the $(m + 1)$ irreducible polynomial $P(x)$:

$$P(x) = X^8 + X^4 + X^3 + X + 1 \tag{D.7}$$

Equivalent prime field number is $\{100011011\}$ in binary or $\{01\}\{1B\}$ in hexadecimal notations. This polynomial, among other constants, is part of the AES formal specifications ($m(x)$ in the AES specifications).

Let us now consider a smaller finite field, with $p = 2$ and $m = 3$, the $GF(2^3)$. This finite field is composed of 8 polynomials, significantly less than the ones used the AES, and contains some of the $GF(2^8)$ elements. Since the coefficients are in the GF(2), the finite field set is transcribed as:

$$GF(2^3) = \{0, 1, X, X+1, X^2, X^2+1, X^2+X, X^2+X+1\}$$

As an example, taking two elements from this set, where denoted as $A(x)$ and $B(x)$:

$$A(x) = X^2 + X + 1$$

$$B(x) = X^2 + 0X + 1$$

For every field $GF(2^m)$ there are several irreducible polynomials of degree $m+1$. One must be used in all arithmetic operations, and of course results will depend on the irreducible polynomial used. For the example we will consider $Q(x)$ as an irreducible polynomial in the $GF(2^3)$:

$$Q(x) = X^3 + X + 1$$

When performing the addition of $A(x)$ and $B(x)$:

$$C(x) = A(x) + B(x)\; mod\; Q(x)$$

$$C^{'}(x) = A(x) + B(x) = (1+1)X^2 + (1+0)X + (1+1) = X$$

$$C(x) = C^{'}(x)\; mod\; Q(x)$$

The $C^{'}(x)$ denotes the intermediate result of the addition, before we apply a modulo reduction. But since $C^{'}(x)$ is in the field of $GF(2^3)$, no reduction is needed and the final result $C(x)$ is equal to $C^{'}(x)$.

If considering the same elements and applying the multiplication:

$$D(x) = A(x).B(x)\; mod\; Q(x)$$

$$D^{'}(x) = A(x).B(x) = (X^2 + X + 1).(X^2 + 1)$$

$$D^{'}(x) = X^4 + X^3 + (1+1)X^2 + X + 1$$

$$D^{'}(x) = X^4 + X^3 + X + 1$$

Since $D^{'}(x)$ is not on the field of $GF(2^3)$, a modulo reduction is required.

$$D(x) = D^{'}(x)\; mod\; Q(x)$$

$(X^4 + X^3 + X + 1) \div (X^3 + X + 1) = X + 1$

$$\begin{array}{r} \underline{+X^4 + 0X^3 + X^2 + 0X + 1} \\ 0X^4 + X^3 + X^2 + 0X + 1 \\ \underline{+X^3 + 0X^2 + X + 1} \\ X^2 + X = D(x) \end{array}$$

In doing so the remainder of the polynomial division is the result of the multiplication modulo, and thus equal to $D(x)$.

As for the inverse multiplication, the inverse of $A(x)$ in the $GF(2^3)$ is the polynomial that verifies the axiom D.4. While the computation is more elaborated the verification is much simple:

$$A(x) = X^2 + X + 1$$

$$A^{-1}(x) = X^2 + X \qquad (= D(x))$$

$$A.A^{-1} = (1+1)X^2 + (1+1)X + 1 = 1$$

The result is 1, a polynomial in the set $GF(2^3)$ and so the $A^{-1}(x)$ is the inverse of the polynomial $A(x)$.

E.2 Overview of the AES

Generically, the AES is a symmetric encryption algorithm, meaning that the encryption and decryption processes are performed by essentially the same steps. It is a block cipher where data is manipulated in blocks of 128 bits, and each block is modified by several rounds of processing. In Figure E.1 is depicted a simplified block representation for the AES encryption. The plain data is submitted to the AES calculation block using the input x, and a cipher key that uses one of three supported lengths is set at the input k. Three different lengths are supported, namely: 128 bits, 192 bits or 256 bits and for each distinct length,

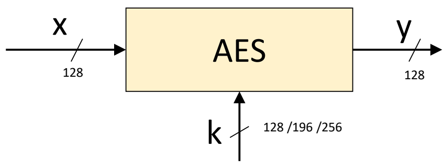


Figure E.1: The AES block interface.

a specific number of rounds, 10, 12, and 14 rounds respectively, is required to complete the AES. After the predetermined number rounds, the ciphered block can be found in the output y.

To conform with the purpose of this appendix, the following descriptions will be limited to the 128-bit key length and describe the design of the AES-128 ECB that uses 10 encryption rounds to compute the cipher block.

E.3 Structure of the AES

Each of the AES rounds is composed of four steps named: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. In Figure E.2 it can be seen the algorithmic structure of the AES when the encryption and decryption processes occur. The cipher rounds (xNr) are identical and each uses a distinct cipher key. In the last round, the final round, it skips the *MixColumns* in both processes. An equivalent inverse cipher

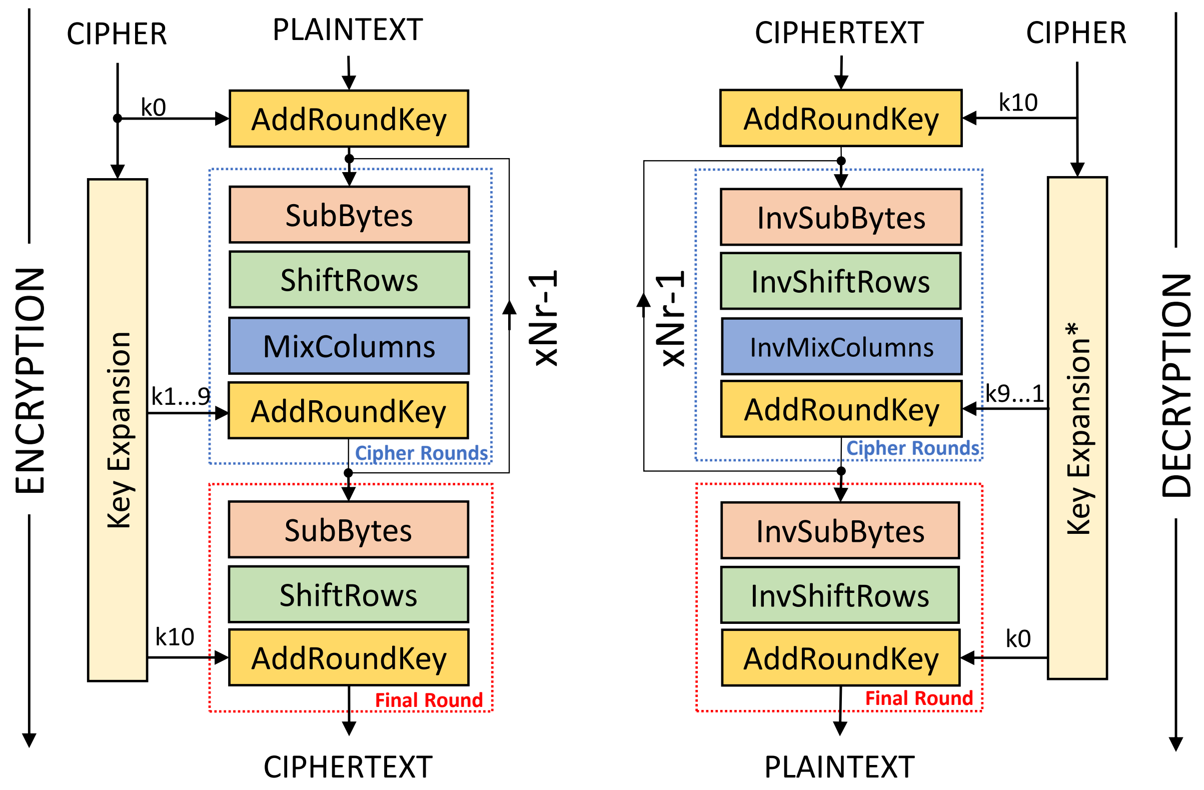


Figure E.2: The AES-128 algorithmic structure.

is used in the decryption process, where the mathematical inverse of each step takes place in the cipher rounds, with certain constants changed.

The *SubBytes* step is the source of non-linearity in the cipher. It provides confusion by replacing each byte in the input with the application of a *sbox* function. The *ShiftRows* and *MixColumns* together provide diffusion in the algorithm using linear operations. The *AddRoundKey* is the addition operation in the algorithm. It increases security in the cipher by combining the 128 bits with portions of the cipher key. It implements a key whitening technique in the most common form, XOR-Encrypt-XOR, and for this reason, an initial *AddRoundKey* is used prior to the AES cipher rounds. Since the input key is pseudo random, the *AddRoundKey* step can be considered source of confusion in the cipher. The confusion and diffusion levels are the strength of the AES against differential attacks.

In the AES internal structure, the 128-bit block is treated on a byte-level, and is stored using a *state* a matrix that consists of four rows and four columns ($S_{r,c}$). A four-byte column is considered one word, and in some cases, the *state* is considered an array of 4 words ($W_0, ..., W_3$). Figure E.3 shows the *state* matrix and highlights the byte- and word-oriented uses. The *state* matrix is the datapath of the AES

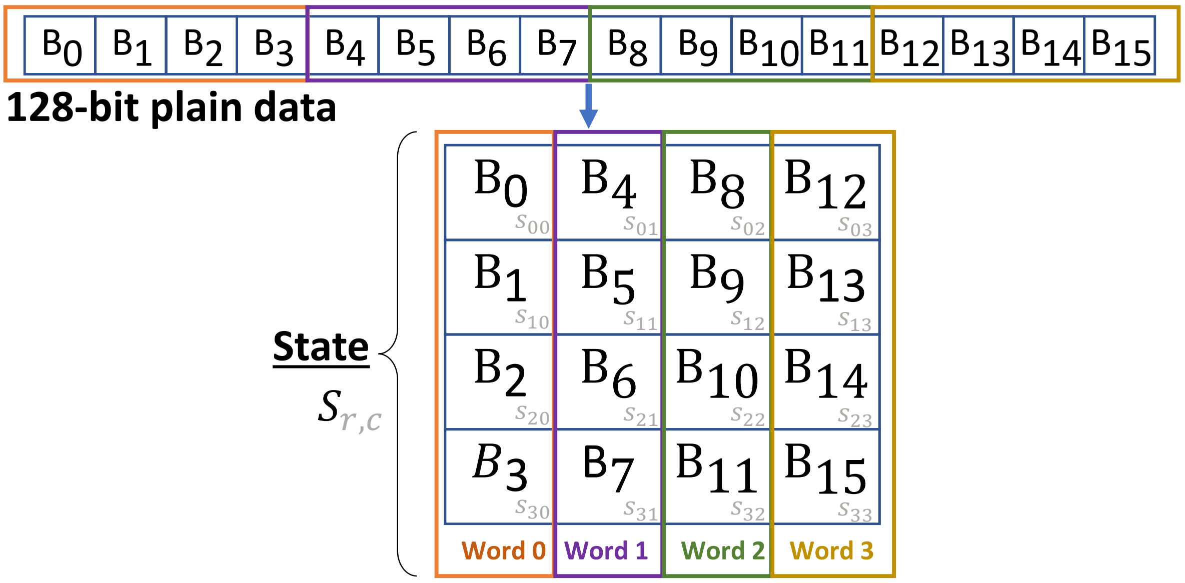

B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15
128-bit plain data
State
$S_{r,c}$
B0 s_{00} B4 s_{01} B8 s_{02} B12 s_{03}
B1 s_{10} B5 s_{11} B9 s_{12} B13 s_{13}
B2 s_{20} B6 s_{21} B10 s_{22} B14 s_{23}
B3 s_{30} B7 s_{31} B11 s_{32} B15 s_{33}
Word 0
Word 1
Word 2
Word 3

Figure E.3: The AES block cipher matrix - The state S matrix.

structure, and is used for storing the input plain data, exchange data between rounds, and after the final round, store the resulting ciphered data.

E.4 Cipher Round

An important aspect of the AES is that in one round, the algorithm operates on the entire 128 bits in the datapath, while other ciphers need more rounds to achieve this level of operation. Such strong encryption level is accomplished using the four steps that implement the AES round: *SubBytes*, *ShitRows*, *MixColumns* and *AddRoundKey*. In all the four steps, computations are performed using $GF(2^8)$ numeric system and use $P(x)$ (equation D.7) irreducible polynomial to apply modulo reduction.

SubBytes step

The *SubBytes*, from substitute bytes, is the first step in the AES round. It implements a non-linear substitution in which the individual 8-bit values in the input, are replaced by the result of applying the *sbox* function, using the value they form. This first layer implements confusion in the AES, since bit-flips in will occur in a non-linear sequence. Figure E.4 shows a parallel implementation block diagram for the *SubBytes* step.

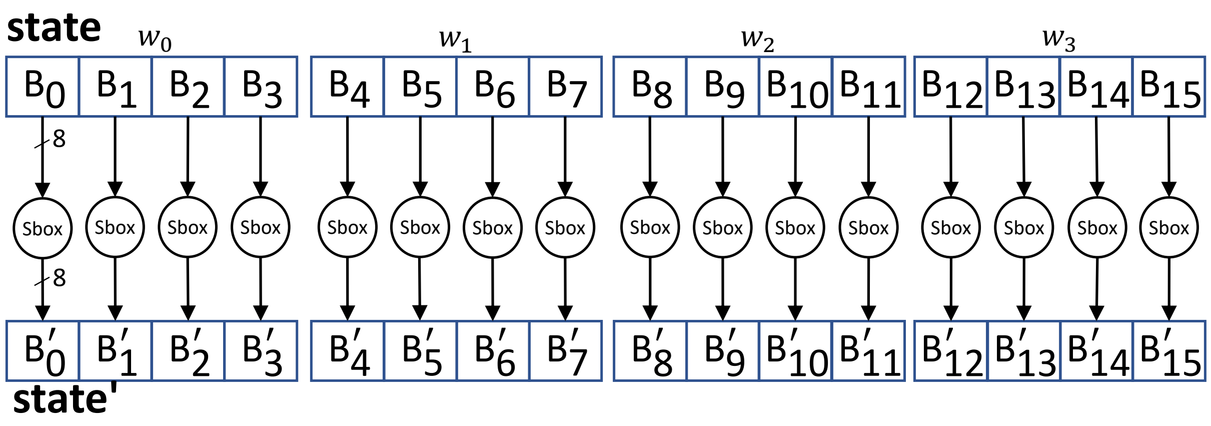

state
w_0
w_1
w_2
w_3
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}
8
Sbox
8
B'_0 B'_1 B'_2 B'_3 B'_4 B'_5 B'_6 B'_7 B'_8 B'_9 B'_{10} B'_{11} B'_{12} B'_{13} B'_{14} B'_{15}
state'

Figure E.4: The AES parallel *SubBytes* block diagram.

The *sbox* derives from the multiplicative inverse over $GF(2^8)$, which is known to have good non-linearity properties. It considers each byte in the input as a polynomial in in the field, and computes two separate transformations: the multiplicative inverse and the affine mapping. Figure E.5 shows the block diagram for the *sbox* function.

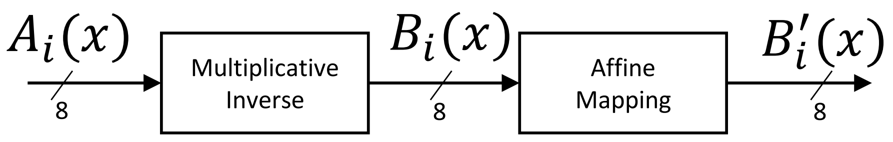


Figure E.5: The *sbox* function internal blocks.

First, the multiplicative inverse of each byte is computed using the Extended Euclidean algorithm, and then the affine transformation over $GF(2^8)$ is applied to the resulting byte. In the affine transformation, every bit in one byte is added (XOR) with bits from the same byte, and finally added with the C(x) polynomial. The bit XOR sequence can be expressed by the following axiom:

$$b'_{[i]} = b_{[i]} \oplus b_{[(i+4)\ mod\ 8]} \oplus b_{[(i+5)\ mod\ 8]} \oplus b_{[(i+6)\ mod\ 8]} \oplus b_{[(i+7)\ mod\ 8]} \oplus c_{[i]} \quad \text{(D.8)}$$

and $i = \{0, 1, ..., 7\}$

Where i, is the i^{th} bit in the input byte, and the $C(x)$, is the constant polynomial $X^6 + X^5 + X + 1$, or $\{63\}$ in hexadecimal representation, or in binary $\{01100011\}$. A formal representation is described using the following matrix multiplication:

$$\overset{B'_x}{\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \overset{B_x}{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}} + \overset{C_x}{\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}} \tag{D.9}$$

The *sbox* function is not a trivial calculation and, for this reason, it is common to use the pre-calculated values that result from the *sbox*. Such values assume a table form, where the input byte is used as index to fetch the corresponding result.

Figure E.6 outlines the source code in C language for the *SubBytes* step. As an example, let us assume that the state matrix, in the column 0 row 0, contains de byte 0x13. Such byte is used to fetch the 0x7d, as an integer index in the *SBox* array (line 72). A common way of reading this value is to split the 0x13 byte and select the row 1 and column 3 in the *SBox* array. The decryption process uses the *InvSubBytes* step in similar manner, but the inverse of the *sbox* function is selected (or *rsbox*).

Figure E.7 depicts the array that contains the 256 pre-calculated values in *rSBox* table. The *InvSubBytes* step is implemented using similar structure and can be seen in line 361. The main difference is the source of data that uses the *rSBox* array (lines 364 to 367), as opposed to the *SBox* array of the *SubBytes* step.

A similar implementation can be used in the hardware *SubBytes* step, and is depicted in Figure E.8. Such an *SBOX* table requires 256 bytes of storage, along with the circuitry to address and fetch the corresponding values. But larger tables translate into more complex implementations with longer propagation delays. Also, if one considers that the input bytes can go into the *sbox* independently, they should be processed in parallel. In doing so, the *SubBytes* step demands for the 16 *sbox* copies. To fully pipeline the AES,

```c
static const uint8_t SBox[256] = {
//0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, //0
0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0, //1
...
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16}; //F
...
#define sbox(x)  SBox[x]
...
void SubBytes(matrix_t* p_State) {
  int i;
  for (i = 0; i < 4; ++i) {
    p_State->column[i].row[0] = sbox(p_State->column[i].row[0]);
    p_State->column[i].row[1] = sbox(p_State->column[i].row[1]);
    p_State->column[i].row[2] = sbox(p_State->column[i].row[2]);
    p_State->column[i].row[3] = sbox(p_State->column[i].row[3]);
  }
}
```

Figure E.6: The software *SubBytes* function using C language.

```c
static const uint8_t rSBox[256] = {
//0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb, //0
0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb, //1
...
0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d}; //F
...
void InvSubBytes(matrix_t* p_State) {
  int i;
  for (i = 0; i < 4; ++i) {
    p_State->column[i].row[0] = rSBox[p_State->column[i].row[0]];
    p_State->column[i].row[1] = rSBox[p_State->column[i].row[1]];
    p_State->column[i].row[2] = rSBox[p_State->column[i].row[2]];
    p_State->column[i].row[3] = rSBox[p_State->column[i].row[3]];
  }
}
```

Figure E.7: The software *InvSubBytes* function using C language.

```vhdl
entity sbox is
  port ( i_data  : in  byte_t;
         o_data : out byte_t);
end entity sbox;

architecture lut of sbox is

    type byte_array_t is array (0 to 255) of std_logic_vector(7 downto 0);

    constant SBOX : byte_array_t := (
            0=>x"63,    1=>x"7c",   2=>x"77",   3=>x"7b",
            4=>x"f2",   5=>x"6b",   6=>x"6f",   7=>x"c5",
            8=>x"30",   9=>x"01",   10=>x"67",  11=>x"2b",
            12=>x"fe",  13=>x"d7",  14=>x"ab",  15=>x"76",
            16=>x"ca",  17=>x"82",  18=>x"c9",  19=>x"7d",
...
            248=>x"41", 249=>x"99", 250=>x"2d", 251=>x"0f",
            252=>x"b0", 253=>x"54", 254=>x"bb", 255=>x"16");

begin
  o_data <= SBOX(to_integer(unsigned(i_data)));
end architecture lut;
```

Figure E.8: The hardware *SubBytes* step using VHDL language.

each of the ten rounds requires one *SubBytes* step, meaning that 10 *SubBytes* are needed. Since and additional of 40 *sbox* functions will be needed in a pipelined key expansion step, the final design requires 200 *sbox* units, a significant allocation of hardware resources that usually dictates the performance of the hardware AES.

The alternative solution is to compute the *SubBytes* step. The most challenging part is the Extended Euclidean algorithm that is known to be "resource greedy". A viable solution is to apply arithmetic decomposition in the $GF(2^8)$, using $GF(2^4)$ and $GF(2^2)$ operations. D. Canright [37] implemented a software algorithm that maps seamlessly to hardware descriptions using VHDL and can be seen in the Listing E.8.

ShiftRows step

The *ShiftRows* step, processes the state matrix row-by-row, applying a rotation to the left in increasing rotate lengths. One can say that the first row uses a "rotation left by zero" positions, meaning that it remains unchanged, the second row uses rotation left of one position, the third row uses rotation left of two positions and the last row uses rotation left of three positions. Despite the data being treated in rows, in reality this is just a byte permutation between the column words. Such transformation provides the first form of diffusion, where the bit-flips produced by the *sbox* function are now propagated horizontally in the matrix. Figure E.9 depicts the byte positions at the input and output of the *ShiftRows* step.

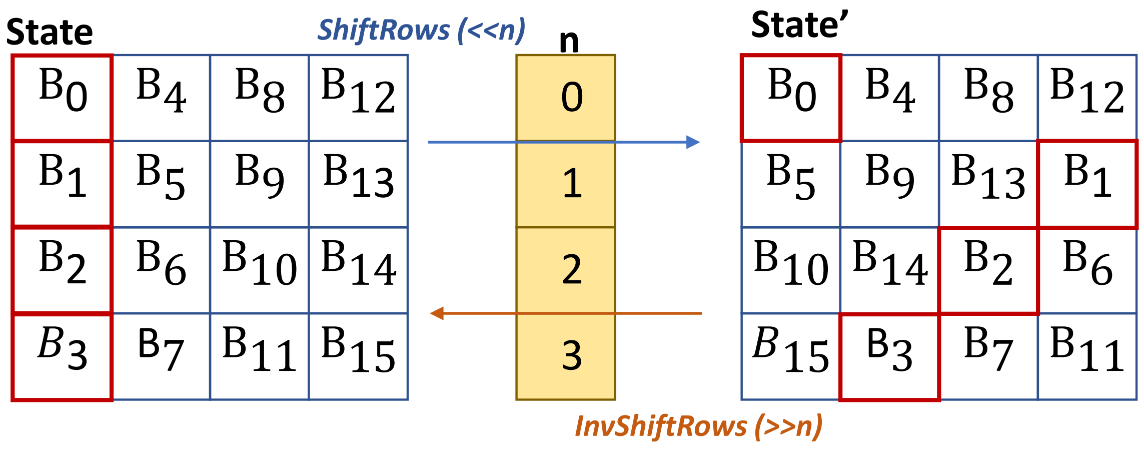


Figure E.9: The *ShiftRows* step in the AES.

At the left-side of the figure, stands the state matrix prior to the *ShiftRows* step, and at the right-side, it can be seen the results from such byte permutation. In the decryption process, the *InvShiftRows* is used, in the inverse order of rotation, reverting the matrix layout to its original form.

The software code to compute the *ShiftRows* can be seen in Figure E.10. The row zero remains unchanged, and a per-row left rotation is performed in sequence from row 1 to row 3.

```c
void ShiftRows (matrix_t* p_State) {
  uint8_t temp;

  temp = p_State->column[0].row[1];
  p_State->column[0].row[1] = p_State->column[1].row[1];
  p_State->column[1].row[1] = p_State->column[2].row[1];
  p_State->column[2].row[1] = p_State->column[3].row[1];
  p_State->column[3].row[1] = temp;

  temp = (p_State->column[0]).row[2];
  p_State->column[0].row[2] = p_State->column[2].row[2];
  p_State->column[2].row[2] = temp;
  temp = (p_State->column[1]).row[2];
  p_State->column[1].row[2] = p_State->column[3].row[2];
  p_State->column[3].row[2] = temp;

  temp = (p_State->column[0]).row[3];
  p_State->column[0].row[3] = p_State->column[3].row[3];
  p_State->column[3].row[3] = p_State->column[2].row[3];
  p_State->column[2].row[3] = p_State->column[1].row[3];
  p_State->column[1].row[3] = temp;
}
```

Figure E.10: The software *ShiftRows* function using C language.

The *InvShiftRows* is implemented similarly and is depicted in the Figure E.11. The step starts using rows 1 to 3 in the inverse column order, and assign the bytes to its original position.

```c
void InvShiftRows(matrix_t* p_State){
  uint8_t temp;

  temp = p_State->column[3].row[1];
  p_State->column[3].row[1] = p_State->column[2].row[1];
  p_State->column[2].row[1] = p_State->column[1].row[1];
  p_State->column[1].row[1] = p_State->column[0].row[1];
  p_State->column[0].row[1] = temp;

  temp = p_State->column[0].row[2];
  p_State->column[0].row[2] = p_State->column[2].row[2];
  p_State->column[2].row[2] = temp;
  temp = p_State->column[1].row[2];
  p_State->column[1].row[2] = p_State->column[3].row[2];
  p_State->column[3].row[2] = temp;

  temp = p_State->column[0].row[3];
  p_State->column[0].row[3] = p_State->column[1].row[3];
  p_State->column[1].row[3] = p_State->column[2].row[3];
  p_State->column[2].row[3] = p_State->column[3].row[3];
  p_State->column[3].row[3] = temp;
}
```

Figure E.11: The software inverse of *ShiftRows* function using C language.

A similar implementation is used in the hardware *ShiftRows* and can be seen in Figure E.12. Each byte in the datapath is rotated using assignments. Those translate into wirings, and so no logic elements are required. The implemented circuit is just a path that links the outputs of the *SubBytes* step, to the correct inputs of the next step, the *MixColumns*.

```
-------------------------------------------------------------------------
  function ShiftRows (
    data_i : matrix_t)
    return matrix_t is
------------------------------------------------------------------------------
    variable data_o : matrix_t;
  begin
    -- row 0                --row 1
    data_o(0)(0) := data_i(0)(0);   data_o(0)(1) := data_i(1)(1);
    data_o(1)(0) := data_i(1)(0);   data_o(1)(1) := data_i(2)(1);
    data_o(2)(0) := data_i(2)(0);   data_o(2)(1) := data_i(3)(1);
    data_o(3)(0) := data_i(3)(0);   data_o(3)(1) := data_i(0)(1);

  -- row 2                --row3
    data_o(0)(2) := data_i(2)(2);   data_o(0)(3) := data_i(3)(3);
    data_o(1)(2) := data_i(3)(2);   data_o(1)(3) := data_i(0)(3);
    data_o(2)(2) := data_i(0)(2);   data_o(2)(3) := data_i(1)(3);
    data_o(3)(2) := data_i(1)(2);   data_o(3)(3) := data_i(2)(3);

    return data_o;
  end function ShiftRows;
 ---------------------------------------------------------------------------------------
```

Figure E.12: The hardware *ShiftRows* function using VHDL.

MixColumns step

The *MixColumns* step operates on the state matrix vertically, or column-by-column. It provides a second form of diffusion, where the bit-flips that were spread horizontally, are now spread to the entire state matrix. The transformation treats each column as a four-term polynomial over the $GF(2^8)$, and multiplies modulo $M(x) = \{01\}X^4 + \{01\}$, with a fixed polynomial $A(x)$ given by:

$$A(x) = \{03\}X^3 + \{01\}X^2 + \{01\}X + \{02\} \tag{D.10}$$

$$S^{'}(x) = A(x) \otimes S(x) \tag{D.11}$$

The numbers in between the braces of D.10 represent the polynomial coefficients using hexadecimal notation. The multiplication in D.11, between the polynomial $A(x)$ and the *state* (S), can be written using a matrix notation and applied column-by-column, as it can be seen in D.12. For completeness, in D.13, it can also be seen the same multiplication using adequate polynomial notation in the $GF(2^8)$.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{D.12}$$

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} X & X+1 & 1 & 1 \\ 1 & X & X+1 & 1 \\ 1 & 1 & X & X+1 \\ X+1 & 1 & 1 & X \end{bmatrix} \cdot \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{D.13}$$

The multiplication can then be decomposed in byte terms, using the following expressions:

$$S'_{0,c} = (\{02\}.S_{0,c}) \oplus (\{03\}.S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \tag{D.14}$$

$$S'_{1,c} = S_{0,c} \oplus (\{02\}.S_{1,c}) \oplus (\{03\}.S_{2,c}) \oplus S_{3,c} \tag{D.15}$$

$$S'_{2,c} = S_{0,c} \oplus S_{1,c} \oplus (\{02\}.S_{2,c}) \oplus (\{03\}.S_{3,c}) \tag{D.16}$$

$$S'_{3,c} = (\{03\}.S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\}.S_{3,c}) \tag{D.17}$$

Once again, in the coefficient multiplication, the AES irreducible polynomial $M(x)$ is used to apply a modulo reduction. For the $\{02\}$ multiplication modulo, a *xtime* function can be applied, and the implementation is described by the following pseudo-code:

```
$if\ B(7) = \{1\}\ then$
    $B' \Leftarrow B(6\ downto\ 0)\&\{0\} \oplus \{1B\}$
$else$
    $B' \Leftarrow B(6\ downto\ 0)\&\{0\}$
```

Here B represents any byte in the input, and the numeric description ($6\ downto\ 0$) means the six right-most bits in the same byte. If the 7^{th} bit is zero, second entry, B' is replaced by the 6 left-most bits at the input concatenated with 0. If otherwise, first entry, the 6 left-most bits are *XORed* with $\{1B\}$ before the assignment to B'.

In similar manner, the multiplication by $\{03\}$ can be decomposed into the *xtime*(B) resulted added/XOR with B:

$$\{03\}.B = (\{02\} \oplus \{01\}).B \tag{D.18}$$

The following pseudo-code, describes an efficient way to implement D.14 to D.17. It considers one column in each iteration and is using the *xtime* function. These descriptions are part of the AES design recommendations.

```
$Tmp \Leftarrow b_0 \oplus b_1 \oplus b_2 \oplus b_3$
$Tm \Leftarrow b_0 \oplus b_1;\ Tm \Leftarrow xtime(Tm);\ b_0^{'} \Leftarrow b_0 \oplus Tm \oplus Tmp$
$Tm \Leftarrow b_1 \oplus b_2;\ Tm \Leftarrow xtime(Tm);\ b_1^{'} \Leftarrow b_1 \oplus Tm \oplus Tmp$
$Tm \Leftarrow b_2 \oplus b_3;\ Tm \Leftarrow xtime(Tm);\ b_2^{'} \Leftarrow b_2 \oplus Tm \oplus Tmp$
$Tm \Leftarrow b_3 \oplus b_0;\ Tm \Leftarrow xtime(Tm);\ b_3^{'} \Leftarrow b_3 \oplus Tm \oplus Tmp$
```

The intermediate value *Tmp* is the transformation that results from adding (XOR) the 4 bytes in one column. The intermediate value Tm is the transformation that results from adding and two consecutive bytes and multiplying *xtime* by *Tmp*. The final $b_n^{'}$ is the result from adding each input byte b_n width the Tm and the Tmp values in each column.

The block diagram of Figure E.13 illustrates the resulting datapath in implementing the above descriptions.

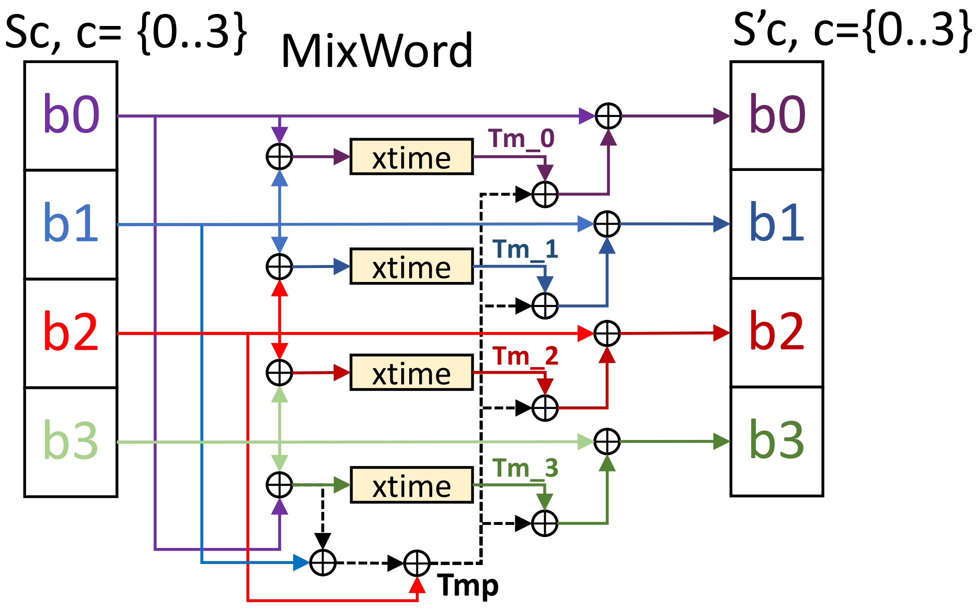


Figure E.13: The *MixWord* (or *MixColumn*) block in the AES *MixColumns* step.

The inverse of the *MixColumns* step is performed in similar manner, but the multiplication uses the polynomial of $A^{-1}(x)$:

$$A^{-1}(x) = \{0B\}X^3 + \{0D\}X^2 + \{09\}X + \{0E\} \tag{D.19}$$

Such computation demands for higher logic resources in multiplying the input word by the coefficients $\{0B, 0D, 09, 0E\}$. The multiplication can be implemented similarly, by using the *xtime* function repeatedly to obtain the higher powers of X.

The software implementation of the *MixColumns* step, follows a column-by-column loop and uses the algorithm depicted above. Figure E.14 outlines this source code using C language. Here the *xtime* function is a macro and is enforcing constant computation where both results are always computed, but only one is non-zero. Such computation strategy is in compliance to recommendations in defending the block cipher against side-channel attacks. Since these are executed in sequence, the byte 0 (i.e., the *p_word->row[0]*) is pre-stored before any computation, and is used in the last line. A temporary storage is used in the variable *Tm*, and the result *Tmp* is used column-wide. The four loops that result from the *NCOLUMS* symbol, use the *p_word* pointer in reapplying this computation to the four columns of the state matrix.

```c
#define xtime(x)  ((x << 1) ^ (((x >> 7) & 1) * 0x1b))
...
void MixColumns (matrix_t* p_State) {
  uint8_t i, Tmp, Tm, w_row0;
  word* p_word = &(p_State->column[0]);

  for (i = 0; i < NCOLUMNS; p_word = &(p_State->column[++i])) {

    w_row0 = p_word->row[0];
    Tmp = p_word->row[0] ^ p_word->row[1] ^ p_word->row[2] ^ p_word->row[3];

    Tm = p_word->row[0] ^ p_word->row[1]; Tm = xtime(Tm);  p_word->row[0] ^= Tm ^ Tmp;
    Tm = p_word->row[1] ^ p_word->row[2]; Tm = xtime(Tm);  p_word->row[1] ^= Tm ^ Tmp;
    Tm = p_word->row[2] ^ p_word->row[3]; Tm = xtime(Tm);  p_word->row[2] ^= Tm ^ Tmp;
    Tm = p_word->row[3] ^ w_row0;         Tm = xtime(Tm);  p_word->row[3] ^= Tm ^ Tmp;
  }
}
```

Figure E.14: The AES *MixColumns* - software implementation using C language.

Figure E.15 outlines the software source of the *InvMixComluns* step. A simplification was introduced that uses *mulGF16* instead of *mulG265*. Here x is the input byte from the state columns, and y is the polynomial coefficient (line 13). Since the four leftmost bits in the polynomial coefficients are zero, it is treated as four bit operation and so the use of the mulGF16. In doing so, the full 8-bit rotation is avoided and consequent computation due to the re-use of the *xtime* function.

```
#define mulGF16(x, y)                                     \
      (  ((y & 1) * x) ^                                  \
      ((y>>1 & 1) * xtime(x)) ^                           \
      ((y>>2 & 1) * xtime(xtime(x))) ^                    \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) )
...
void InvMixColumns(matrix_t* p_State) {
  int i;
  char a,b,c,d;
  unsigned char* p_row;
  for (i = 0; i < 4; ++i) {

    p_row = &p_State->column[i].row[0];

    a = p_row[0]; b = p_row[1]; c = p_row[2]; d = p_row[3];

    p_row[0]=mulGF16(a,0x0e)^mulGF16(b,0x0b)^mulGF16(c,0x0d)^mulGF16(d,0x09);
    p_row[1]=mulGF16(a,0x09)^mulGF16(b,0x0e)^mulGF16(c,0x0b)^mulGF16(d,0x0d);
    p_row[2]=mulGF16(a,0x0d)^mulGF16(b,0x09)^mulGF16(c,0x0e)^mulGF16(d,0x0b);
    p_row[3]=mulGF16(a,0x0b)^mulGF16(b,0x0d)^mulGF16(c,0x09)^mulGF16(d,0x0e);
  }
}
```

Figure E.15: The AES *InvMixWord* - software source using C language.

The hardware *MixColumns* step design is based on four *mixWord* units implementing the algorithm in Figure E.13. Figure E.16 outlines the *mixWord* entity using a VHDL description. The MixColumns architecture operates using parallel *mixWords* and can be consulted attached in Listing E.9.

```
entity mixWord is
  port (
    i_word : in  word_t;
    o_word : out word_t);
end entity mixWord;
architecture Structural of mixWord is

signal  byte0_d1, tm_0: std_logic_vector(7 downto 0);
signal  byte1_d1, tm_1: std_logic_vector(7 downto 0);
signal  byte2_d1, tm_2: std_logic_vector(7 downto 0);
signal  byte3_d1, tm_3: std_logic_vector(7 downto 0);
signal Tmp : std_logic_vector(7 downto 0);
begin

Tmp <= i_word(0) xor i_word(1) xor i_word(2) xor i_word(3);

byte0_d1 <= i_word(0) xor i_word(1);
byte1_d1 <= i_word(1) xor i_word(2);
byte2_d1 <= i_word(2) xor i_word(3);
byte3_d1 <= i_word(3) xor i_word(0);

tm_0 <=(byte0_d1(6downto0)&'0')xorx"1B" when byte0_d1(7)='1'else byte0_d1(6 downto 0) & '0';
tm_1 <=(byte1_d1(6downto0)&'0')xorx"1B" when byte1_d1(7)='1'else byte1_d1(6 downto 0) & '0';
tm_2 <=(byte2_d1(6downto0)&'0')xorx"1B" when byte2_d1(7)='1'else byte2_d1(6 downto 0) & '0';
tm_3 <=(byte3_d1(6downto0)&'0')xorx"1B" when byte3_d1(7)='1'else byte3_d1(6 downto 0) & '0';

o_word(0) <= i_word(0) xor Tm_0 xor Tmp;
o_word(1) <= i_word(1) xor Tm_1 xor Tmp;
o_word(2) <= i_word(2) xor Tm_2 xor Tmp;
o_word(3) <= i_word(3) xor Tm_3 xor Tmp;

end architecture Structural;
```

Figure E.16: The AES *MixWord* - hardware description using VHDL.

In line 23 the *Tmp* result is computed using a triple XOR that uses the four input bytes. The XOR operation between consecutive byte-pairs in the word is lines 25 to 26, and in lines 30 to 33 the *xtime* function is implemented using these results as input. After all independent terms are computed, the output is assigned using the XOR operations that consider the input bytes, the result form the *xtime* functions and the *Tmp* value (lines 35 to 38).

AddRoundKey step

The final step in the AES cipher round is the *AddRoundKey* step. The implementation is a XOR operation that uses the 16 bytes in the datapath and each distinct round key. This step also creates some form of confusion, since bit-flips will occur depending on the cipher key that is pseudo-random. Just like in the previous steps, all input bytes can be handled independently, and the *AddRoundKey* step can be implemented in a sequential or parallel byte XOR. Figure E.17 depicts this *AddRoundKey* step block diagram.

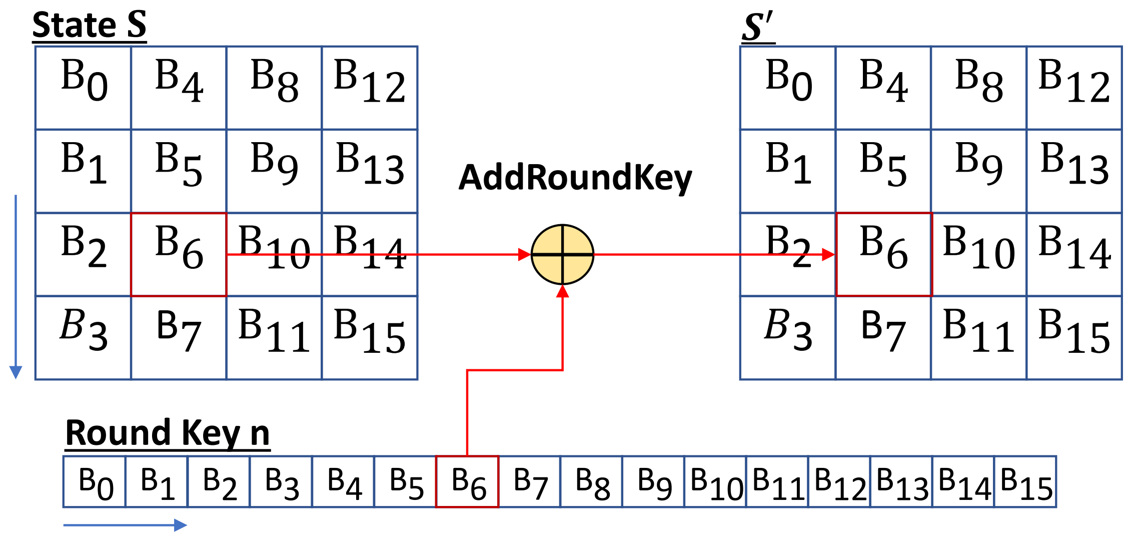


Figure E.17: The AES *AddRoundKey* step block diagram.

It can be seen that each of the input bytes in the state matrix, is XORed with the bytes in the same position in the n round key. The results are stored back to the original positions of the state matrix.

In Figure E.18 the software code that implements this step is listed. A word-based addition, or column-based addition, is using a state matrix pointer in the application scope. The XOR operator of the *word* type is being overloaded to perform 32-bit arithmetic (lines 25 and 32). In doing so, the *add_round_key* function is performing a 4-column sequential XOR using a round indexed key. The *RoundKey* variable is an array that contains the 10+1 keys that were generated from the original key, and it will be discussed in the next section. Since XOR is an involutory operation that results in its own inverse, the *AddRoundKey* step is used in both, the encryption and decryption processes.

```
word word::operator^(const word& operand_b){
  word rc;
  *((int*)&rc) = *((int*)(this)) ^ *((int*)&operand_b);
  return rc;
}

word& word::operator^=(const word& operand_b){
  *((int*)this->row) ^= *((int*)operand_b.row);
  return *this;
}
...
void AddRoundKey(matrix_t* p_State, int round){
  p_State->column[0] ^= RoundKey[round].column[0];
  p_State->column[1] ^= RoundKey[round].column[1];
  p_State->column[2] ^= RoundKey[round].column[2];
  p_State->column[3] ^= RoundKey[round].column[3];
}
```

Figure E.18: The AES *AddRoundKey* step - software implementation using C language.

The hardware *AddRoundKey* description is depicted in Figure E.19. It can be seen a VHDL function that implements a parallel byte-oriented XOR, between the state matrix and a round key. In the hardware implementation phase, such description is translated to the original 128 bits, and 128 XOR logic gates are selected using the available resources.

E.5 The Key Schedule

The ten keys used in the AES rounds result from the application of the key expansion algorithm to the input key, and for this reason they are often called subkeys. The AES takes the input key and computes an expansion routine that generates a key schedule. The same routine is computed by the inverse cipher, but the key schedule is used in the inverse order of keys. Such schedule consists in one array of 44 words(4 bytes each), here denoted by W_i. Figure E.20 depicts the block diagram of a 10 round expansion routine that uses the cipher key as input and outputs the eleven resulting keys.

The first four words are filled with the entry cipher key, that is the *RoundKey[0]*, and the expansion performs according to the following rules:

```
--------------------------------------------------------------------------------------
function AddRoundKey(
--------------------------------------------------------------------------------------
   state_i  : matrix_t;
   key_i : std_logic_vector)
   return matrix_t is
--------------------------------------------------------------------------------------
variable state_o : matrix_t;
begin

   state_o(0)(0) := state_i(0)(0) xor key_i(127 downto 120);
   state_o(0)(1) := state_i(0)(1) xor key_i(119 downto 112);
   state_o(0)(2) := state_i(0)(2) xor key_i(111 downto 104);
   state_o(0)(3) := state_i(0)(3) xor key_i(103 downto 96);

   state_o(1)(0) := state_i(1)(0) xor key_i(95 downto 88);
   state_o(1)(1) := state_i(1)(1) xor key_i(87 downto 80);
   state_o(1)(2) := state_i(1)(2) xor key_i(79 downto 72);
   state_o(1)(3) := state_i(1)(3) xor key_i(71 downto 64);

   state_o(2)(0) := state_i(2)(0) xor key_i(63 downto 56);
   state_o(2)(1) := state_i(2)(1) xor key_i(55 downto 48);
   state_o(2)(2) := state_i(2)(2) xor key_i(47 downto 40);
   state_o(2)(3) := state_i(2)(3) xor key_i(39 downto 32);

   state_o(3)(0) := state_i(3)(0) xor key_i(31 downto 24);
   state_o(3)(1) := state_i(3)(1) xor key_i(23 downto 16);
   state_o(3)(2) := state_i(3)(2) xor key_i(15 downto 8);
   state_o(3)(3) := state_i(3)(3) xor key_i(7 downto 0);

   return state_o;
end AddRoundKey;
--------------------------------------------------------------------------------------
```

Figure E.19: The AES *AddRoundKey* step - hardware implementation using VHDL.

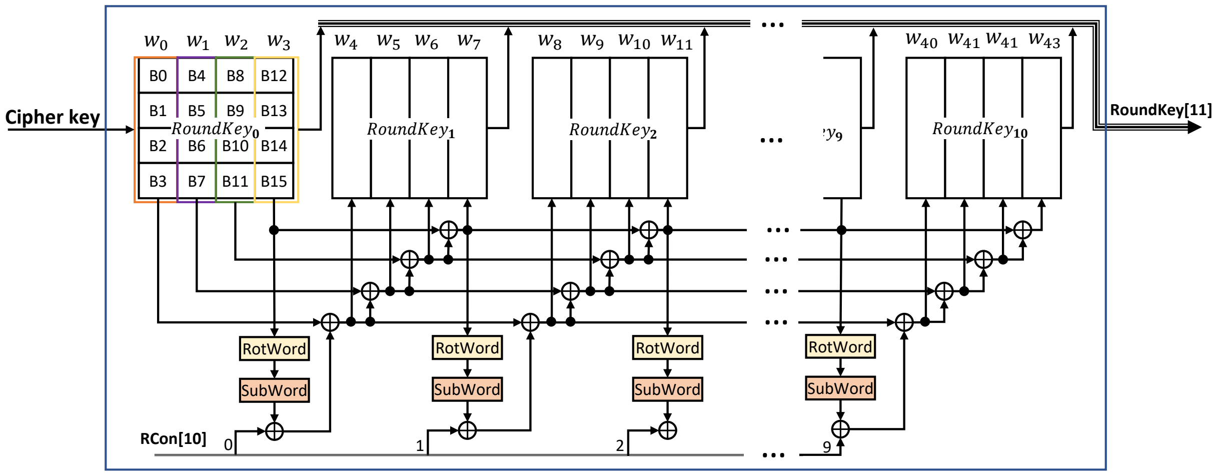


Figure E.20: The AES Key Schedule Algorithm- pipeline-based block diagram.

1. Every following W_i word is equal to XOR result between the previous word, $W_{(i-1)}$, and the word that is 4 positions earlier, $W_{(i-4)}$. This rule can be rewritten using the following equation:

$$W_i = W_{(i-1)} \oplus W_{(i-4)} \quad \text{(D.20)}$$

2. For the Words in the positions that are multiple of 4, the preceding word $W_{(i-1)}$ suffers a transformation, followed by a XOR with a round constant *RCon[k]*, before applying rule (1). The transformation consists of a word rotation, using a *RotWord* function, followed by a substitute bytes step, using a *SubWord* function. The *RotWord* function implements one rotation to the word, and the *SubWord* function is a substitute bytes step applied to 4 bytes in the word. The following equation describes the complete transformation, and includes rule (1):

$$W_{4k} = SubWord[RotWord(W_{(4k-1)})] \oplus Rcon[k-1] \oplus W_{(i-4)} \tag{D.21}$$

As an example, if $k = 1$ the above expression translates to:

$$W_4 = SubWord[RotWord(W_3)] \oplus Rcon[0] \oplus W_0$$

E.5.1 Sequential Key Expansion

The key expansion routine can be implemented using two design strategies: a pipeline-based design, composed by independent functional units as depicted in the Figure E.20; or a sequential-based design that computes each subkey inside a processing loop, and shares the computing resources between expansion rounds. Figure E.21 includes the software implementation of the sequential key expansion routine in C programming language.

In the listing of Figure E.21, the first four lines copy the 128-bit input key to the position zero in the schedule (k_0 in lines 29 to 32), and then, the *for* loop implements ten sequential expansion rounds. Once again, columns are same as words, and so the transformation starts using the 3^{rd} word (column) of the previous subkey ($k-1$), that is used in the 1^{st} word of the new subkey, (k). It applies the *rotWord* and *subWord* "functions", and completes the transformation by adding the *RoundConstant* to the resulting word (lines 36 to 46). Each word is then consecutively XORed using the $w_{(i-1)}$ word in the current subkey, and $w_{(i-4)}$ word from the previous subkey (lines 48 to 51). The cycle restarts with a new subkey and repeats the processing until all the 10+1 subkeys are in the schedule.

A similar implementation can be used for the sequential hardware key expansion. Figure E.22 outlines such implementation and the complete description can be consulted in Listing E.4.

A single FSM with three states starts in the *s0_Ready* state, uses the *s1_SchedRounds* 9 times, and completes the key schedule in the *s2_FinalRound* state (line 112). In doing so, it requires 1+9+1=11

```
void key_expansion(char* key;){
  state_t  word_t W3, * p_cipher = (state_t*) key;

  //k0
  RoundKey[0].column[0] = p_cipher->column[0];
  RoundKey[0].column[1] = p_cipher->column[1];
  RoundKey[0].column[2] = p_cipher->column[2];
  RoundKey[0].column[3] = p_cipher->column[3];

  for (int k = 1; k < 10 + 1; ++k) {
    //RotWord()
    W3.row[0] = RoundKey[k - 1].column[3].row[1];
    W3.row[1] = RoundKey[k - 1].column[3].row[2];
    W3.row[2] = RoundKey[k - 1].column[3].row[3];
    W3.row[3] = RoundKey[k - 1].column[3].row[0];
    //SubWord()
    W3.row[0] = sbox(W3.row[0]);
    W3.row[1] = sbox(W3.row[1]);
    W3.row[2] = sbox(W3.row[2]);
    W3.row[3] = sbox(W3.row[3]);
    //Add RoundConstant
    W3.row[0] ^= RCon[k - 1];

    RoundKey[k].column[0] = W3                    ^ RoundKey[k - 1].column[0];
    RoundKey[k].column[1] = RoundKey[k].column[0] ^ RoundKey[k - 1].column[1];
    RoundKey[k].column[2] = RoundKey[k].column[1] ^ RoundKey[k - 1].column[2];
    RoundKey[k].column[3] = RoundKey[k].column[2] ^ RoundKey[k - 1].column[3];
  }
  NeedsExpand = false;
}
```

Figure E.21: The AES Key Expansion - software C sequential implementation.

clocks to complete the key schedule. The implementation of such FSM can be seen in lines 128 to 148 of Listing E.4.

In Figure E.22, an input MUX is used to select the $k_{(i-1)}$ key in the schedule registry, from *sched_words_q*, and assigned to a *sub_key_i* array of words (lines 190 to 193). The *sub_key_i* is input to the single expand round in the datapath, and this round starts by applying the *subWord* computation to the third column of *sub_key_i* (line 197). The output is XORed with *RCON*(*key_sel*) and uses the *rotWord* positioned sequence $\{1, 2, 3, 0\}$ (lines 201 to 204). A cascade of consecutive XORs concludes the schedule round, leaving the resulting subkey in the *sub_key_o* word array (line 206 to 209). The FSM assigns such words to the next position in the schedule registry (line 214 to 217), and the next round will use this new words for expanding the next subkey.

E.5.2 Pipelined Key Expansion

When following a pipeline design strategy, the ten expansion rounds are implemented as independent functional units. Each round is similar in structure, as seen in the sequential design, which translates to a resource demand of 4x10 *sbox* functions, used in the each *subWord* unit, and 10x10 32-bit XOR

```
...
type StateType is (s0_Ready,s1_SchedRounds,s2_FinalRound);
...
type byte_array_t is array (0 to 9) of std_logic_vector(7 downto 0);
...
constant RCON:byte_array_t:=(x"01",x"02",x"04",x"08",x"10",x"20",x"40",x"80",x"1B", x"36");
...
----------------------------------------------------------------------------------------
-- Key Round
----------------------------------------------------------------------------------------
    --Input Mux
    subkey_i(0) <= sched_words_q(key_sel*4);
    subkey_i(1) <= sched_words_q(key_sel*4+1);
    subkey_i(2) <= sched_words_q(key_sel*4+2);
    subkey_i(3) <= sched_words_q(key_sel*4+3);
    -------------------------------------
    SubWord3: entity SubWord(Parallel)
    -------------------------------------
        port map ( i_word  => subkey_i(3),o_word => subword_o);
    -------------------------------------
    --rcon[k] xor rotWord()
    -------------------------------------
     rcon_i(0) <= subword_o(1) xor RCON(key_sel);
     rcon_i(1) <= subword_o(2);
     rcon_i(2) <= subword_o(3);
     rcon_i(3) <= subword_o(0);

  subkey_o(0) <= rcon_i xor subkey_i(0)                                                ;
  subkey_o(1) <= rcon_i xor subkey_i(0) xor subkey_i(1)                                ;
  subkey_o(2) <= rcon_i xor subkey_i(0) xor subkey_i(1) xor subkey_i(2)                ;
  subkey_o(3) <= rcon_i xor subkey_i(0) xor subkey_i(1) xor subkey_i(2) xor subkey_i(3);
 ----------------------------------------------------------------------------------------
 out_mux:process(key_next_sel,subkey_o)
----------------------------------------------------------------------------------------
 begin
     sched_words_d(key_next_sel*4)     <= subkey_o(0);
     sched_words_d(key_next_sel*4 + 1) <= subkey_o(1);
     sched_words_d(key_next_sel*4 + 2) <= subkey_o(2);
     sched_words_d(key_next_sel*4 + 3) <= subkey_o(3);
  end process out_mux;
----------------------------------------------------------------------------------------
  RoundKeys_o : for k in 0 to 10 generate
----------------------------------------------------------------------------------------
    o_round_keys(k) <= conv_std_logic_vector(sched_words_q(4*k),...
...
```

Figure E.22: The AES Key Expansion - hardware sequential-based description using VHDL.

logic operations. Such a design description can be consulted in the Listing E.3. In lines 34 to 40, the input key is assigned to the schedule words 0 to 3, and is also input to the expansion rounds pipeline using *sched_word_q(0)* register. A *for generate* description implements the 10 expansion rounds, that are connected with the schedule registry using *sched_words_q* as input, and *sched_words_d* as output, that is stored in the next clock rising. A similar description style is used in lines 43 to 50 to generate the 10 *subWord* units used in each round (lines 58 to 61). To enable the round operation, the input *i_run* signal is forwarded across the pipeline using the *round_enable* registry, in line 84, and is used in lines 64, 66, 68, and 70. Once in each round all operations are concurrent, and the consecutive XOR between words should not use the preceding word results. The result is a growing cascade XOR where the next word repeats the operations that compute the words behind (lines 63, 65, 66, 67 and 69). A *for generate* concludes the design description by connecting the output of the schedule registry to *o_rounk_keys* (line 102), an output at the top-level of the Key Expansion unit.

E.6 The Software AES

The software AES follows a sequential implementation of the algorithm depicted in the Figure E.2. The user provides a cipher key in the appropriate size, and a key schedule algorithm expands this key to generates a key schedule. The same key expansion can be used in the encrypt and decrypt processes, by use of the inverse key sequence. The first key in the schedule is added to the input block, and for a pre-determined number of rounds, the input block undergoes in four consecutive transformations that use one of the ten keys generated. Such rounds use the steps described in the above sections and are also depicted in the block diagrams of the same figure.

E.6.1 Encrypt process

In the 128-bit AES encryption, after 10 rounds the input block is fully encrypted and its decryption is pending for the key used. Figure E.23 outlines a software function using C programming language that implements the AES encryption process.

The *encrypt_block* function, receives a pointer that specifies the memory location of the input data to be encrypted. The same location is used throughout the encryption rounds for storing the intermediate values. The key schedule is generated once using the *KeyExpansion* function, and the resulting keys can be used in successive encryption iterations. Whenever a new key is defined in the system, the state variable *NeedsExpansion* is modified, and the next use incurs in a new scheduling operation before implementing the cipher rounds. The *InitialAdd* step adds the input key, round key 0, to the received block (in line 232). The nine "regular" cipher rounds are executed using a *for-loop* and use the consecutive key in the schedule (lines 238 to 241). The final round is executed before the function completes (lines 243 to 245). At this point, the *Mixcolumns* function is skipped and the last round key is used, replacing the input data with the final result for the encrypted block.

```
#define NROUNDS 10
int encrypt_block(char* p_block) {
  matrix_t* p_State = (matrix_t*)p_block;
  int round = 0;

  if (!p_State) return -EFAULT;

  AddRoundKey(p_State,0);

  if (NeedsExpansion)
    KeyExpansion();

  for (round = 1; round < NROUNDS; ++round) {
    SubBytes(p_State);
    ShiftRows(p_State);
    MixColumns(p_State);
    AddRoundKey(p_State, round);
  }
  SubBytes(p_State);
  ShiftRows(p_State);
  AddRoundKey(p_State, NROUNDS);

  return round;
}
```

Figure E.23: The AES software encrypt process - sequential implementation.

E.6.2 Decrypt process

The decryption process follows a similar sequence to the encryption, using the inverse operations and key schedule that was generated by the same input key in the inverse order. In the particular case of ECB mode, the decryption does not represent a pure symmetric sequence, since the *AddRounKey* step must be used before the *InvMixColumns* step. To cope with a pure symmetric sequence, the order can be reversed since both operations are linear. As a consequence, the *InvMixColmuns* step needs to be applied to both, the state matrix and the round key. The expressions below exemplify the required change.

$$\boldsymbol{InvMixColumns}(\ State \oplus RoundKey[k]\ ) \Leftrightarrow$$

$$\Leftrightarrow [\ InvMixColumns(State)\ ] \oplus [\ InvMixColumns(RoundKey[k])\ ]$$

Due to this change, it is common to perform the transformation on the key side in the key schedule step. The cipher round can then assume a pure symmetrical structure as depicted in Figure E.2. The consequence of this transformation is an increased processing that is often avoided, and the order of operations maintains the non-symmetric sequence.

In Figure E.24 the software C function that implements the AES decryption process, following the non-pure symmetric structure is depicted. Typically, the encryption and decryption processes are executed in different systems, meaning that each one performs its own key scheduling using the same cipher key value. To cope with the inverse key sequence, the *AddRoundKey* starts with the last key in the schedule, and the reverse sequence proceeds to the ten rounds that implement the AES decryption (line 475). The decryption rounds, where the *AddRoundKey* step precedes the *InvMixColumns* are implemented in lines 478 to 481. The final round is executed using the input key (lines 484 to 486) that can be found in the index 0 of the key schedule and the function concludes replacing the encrypted block with the corresponding plain data.

```
466 int decrypt_block(char* p_block){
467   int round = 0;
468   matrix_t* p_State = (matrix_t*)p_block;
469
470   if (!p_State)return -EFAULT;
471
472   if (NeedsExpansion)
473     KeyExpansion();
474
475   AddRoundKey(p_State, NROUNDS);
476
477   for (round = (NROUNDS - 1); round > 0; --round) {
478     InvSubBytes(p_State);
479     InvShiftRows(p_State);
480     AddRoundKey(p_State, round);
481     InvMixColumns(p_State);
482
483   }
484   InvSubBytes(p_State);
485   InvShiftRows(p_State);
486   AddRoundKey(p_State,0);
487
488   return NROUNDS;
489 }
```

Figure E.24: The AES software decrypt process - sequential implementation.

E.7 Hardware Architecture for AES

The hardware architecture for the AES can follow the two distinct design strategies: sequential- or pipelined-based designs. Since all bytes should be treated independently, the search for the computational performance can push designers into the adopt the pipeline-based designs. As discussed above, the demand for logic resource can be a limiting condition and a sequential design can use significantly less logic resources, at the expense of lowering the overall throughput.

E.7.1 Sequential Encryption

The sequential design strategy used in the hardware AES encryption is depicted in Figure E.25. In this approach, the four layers that implement the AES cipher round can be reused in all the 10+1 rounds. It can also use a sequential Key Expansion as described in the previous section.

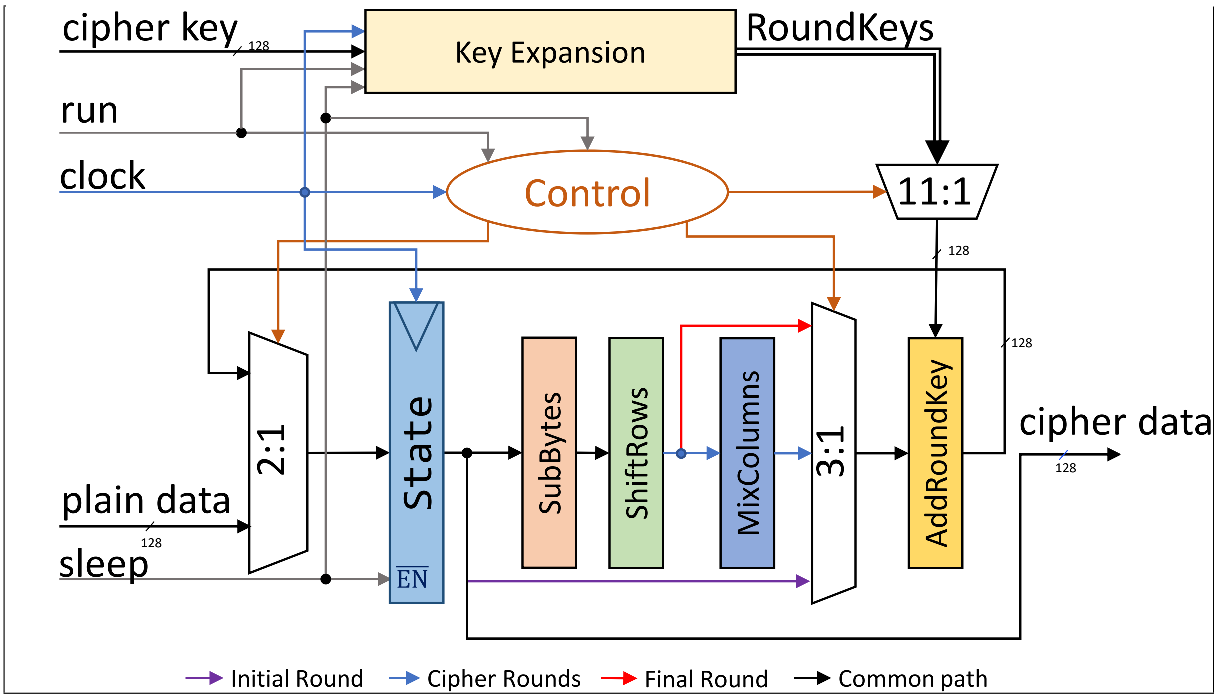


Figure E.25: The hardware AES - sequential design.

To analyze this block diagram, is better to begin with an initial clock pulse that stores the input plain data using the appropriate source in the 2:1 MUX. The purple line in the datapath will be followed to compute the initial *AddRoundKey*, placing the results at the second input of the same MUX. The next clock pulse stores this result and the blue path will be followed in the subsequent nine clocks pulses. In this path, the design completes nine regular cipher rounds and in the 11^{th} clock pulse, the red line will be followed to avoid the *MixColumns* step. In the 12^{th} clock cycle, the computed results will be stored in the state matrix that receives the encrypted data block. The Control unit is responsible for synchronizing the inputs in this datapath, and it must also select the appropriate subkey for each of the 10+1 rounds. Each of the AES round layers is implemented using the VHDL descriptions described above.

The sequential architecture descriptions that implement the block diagram of Figure E.25, can be consulted attached in the Listing E.2. In this Listing, in line 117 the states for the Control FSM are defined, and the description of this unit can be seen in lines 127 to 149. Four states are used, namely: *s_ready*; *s_init_round*; *s_cipher_rounds*; *s_final_round*; and the FSM repeats the third state for nine clock pulses. The Control stays in state *s_ready* until it receives the *i_run* signal, at which it stores the input plain data (lines 117 and 221) and proceeds to the *s_init_round* state. During these two states, the *Add_Mux* description (MUX 3:1) in lines 195 to 208, keeps the *cipher_state_q* register selected, where the input block is stored. The output from this MUX is received as an argument on the *AddRoundKey* function on line 212, which uses *round_counter* to select the appropriate key in the *round_keys_o* array. This condition computes the initial round of the AES, using the input state and the key 0 in the schedule. For that, the *round_counter* value is kept in 0 by means of the condition in lines 231 and 232. The return from *AddRoundKey* function is sent via the *cipher_round_o* signal to the MUX 2:1 on line 177, and the result is stored in the next clock pulse on line 221.

For the next nine clock pulses, the Control Unit implements the nine regular rounds of the AES using the *s_cipher_rounds* state. During this state, the datapath forwards the value of the *cipher_state_q* register to the *SubBytes* entity in line 182, and the output of this unit links with the *ShiftRows* function in line 187. The return from the *ShitRows* is received at the input of the *MixColumns* unit in line 192, and the successive computing steps end once again in the line 212, using the *AddRoundKey* function. For that, during the *s_cipher_rounds* state, the *Add_Mux* maintains selected the signal from the *MixColumns* (line 204).

After 9 clock pulses, the Control switches to the *s_final_round* state and *Add_Mux* selects the input from *ShiftRows*, thus removing the *MixColumns* unit from the processing chain. Upon reaching this final state, the matrix value is stored in the *final_cipher_q* register, allowing the Control Unit to start a new block encryption before this value has been read (lines 226 and 227). The contents of this register links with the top-level *o_ciphered_data* output on line 241 and its reading must be triggered after the *o_done* signal rises.

After this last clock pulse, the control is once again in the *s_ready* state and the MUX 2:1 already selected the input *i_plain_data*. Twelve clock cycles were used by the Control FSM to complete the 10+1 AES encryption Rounds.

E.7.2 Pipelined Encryption

The pipelined AES architecture is based on the replication of the hardware resources involved in the sequential description. The state matrix is replicated in 1+10+1 units, interspersed with 9 units of regular cipher rounds, and 1 final round unit that skips the *MixColumns* step. The last matrix closes the pipeline's processing chain and makes the encrypted data stable at the output of the AES architecture. It will hold this value for at least one clock pulse when operating continuously, or until a new received block reaches the final stage. Figure E.26 describes this pipeline architecture using a simplified block diagram.

Simultaneously, the key schedule is triggered in the *Key Expansion* unit that is also implemented using a pipeline-based architecture. It will provide the sub-keys used in each of the 10 + 1 *AddRoundKey* steps. In the same figure, it can also be seen a feed-forward of the *run* signal that combined with the input *sleep*

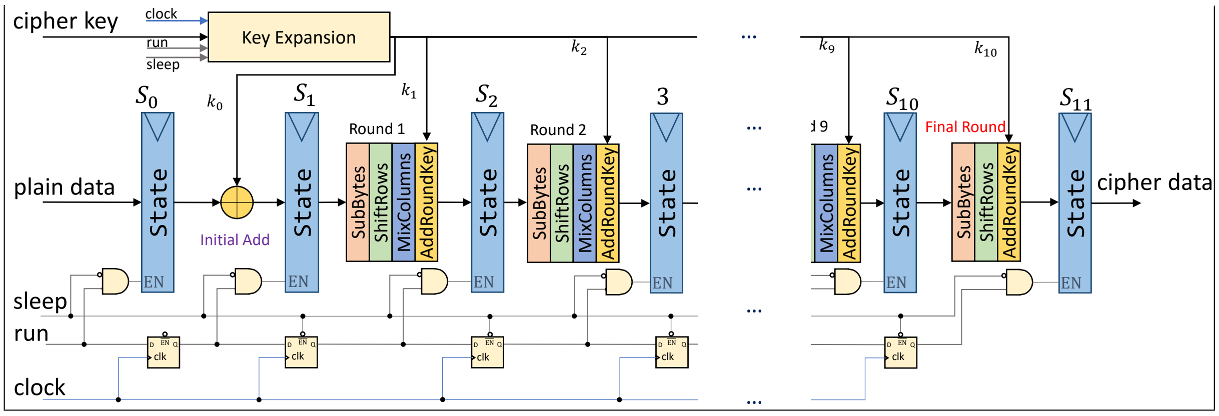


Figure E.26: The hardware AES - pipeline design.

signal, enables or suspends the propagation of the encryption block throughout the eleven stages of the pipeline. At least twelve clock pulses are needed to fully encrypt the input plain data. A steady high *sleep* signal suspends the processing for a necessary number of clock pulses, or it can be used periodically to lower the overall throughput.

The pipeline-based architecture descriptions for the AES can be seen in the attached Listing E.1. Lines 40 to 47, implement the connections between the AES pipeline and the key Expansion unit, and it is possible to observe that the entity was selected to bind with the pipeline-based architecture. In line 50, the description of the *Initial Round* uses the round key 0 and *plain_data_q* register to compute the first *AddRoundKey* step. This register stores the value of the input block if it receives a clock rise pulse and the *sleep* signal is not active (line 94). The return of this function feeds the first element in the array of state matrices, *cipher_state_q*, which divides the *cipher_rounds* in the stages of the pipeline. These rounds are implemented using the *for generate* descriptions, on lines 52 to 60, where each unit uses the input matrix index one position behind of the output it produces, so as to produce interlacing. The internal composition of these regular rounds include the four steps described in the AES structures and can be consulted in the Listing E.5.

The output of the 9^{th} regular round, uses the index 9 in the array of matrices, to feed the final round of the pipeline (line 65). This final round completes the AES encryption process on line 68, by computing the *AddRoundKey* step. This function is using the results from the *SubBytes* step (line 66), in the appropriate positions linked by the *ShiftRows* step, and selects the subkey 10 in the key schedule. The last state matrix in the array, stores the return value from this function on line 106, where it remains steady for at least one clock pulse. In line 112, the value of this register links to the output *o_ciphered_data*, that belongs to the top-level to AES unit. In Similarity width the sequential architecture, the reading of this data must be made after the rising edge of the *o_done* signal.

Twelve clock pulses are needed to encrypt the input block using the pipeline hardware AES. After the first encryption is completed, the one block encryption per clock pulse rate can be maintained continuously.

The remainder of this appendix includes the listings that describe the hardware AES sequential- and pipeline-based architectures.

Listing E.1: The hardware AES - Pipeline architecture (back to Figure E.26).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.all;
use aes_128_v1_00_a.KeyExpansion;
use aes_128_v1_00_a.SubBytes;
use aes_128_v1_00_a.MixColumns;
use aes_128_v1_00_a.CipherRound;

entity aes128 is
  port ( clock : in std_logic;
         reset : in std_logic;
         i_run : in std_logic;
         i_sleep : in std_logic;
         i_key_expand : in std_logic;
         o_done : out std_logic;
         i_plain_data  : in  std_logic_vector(127 downto 0);
         i_cipher_key  : in  std_logic_vector(127 downto 0);
         o_ciphered_data : out std_logic_vector(127 downto 0));
end entity aes128;

architecture Pipeline of aes128 is

    type state_array_t is array (0 to 9) of matrix_t;
    signal plain_data_q     : std_logic_vector(127 downto 0);
    signal key_expand_i     : std_logic;
    signal sleep_i          : std_logic;
    signal cipher_states_d, cipher_states_q   : state_array_t;
    signal ciphered_data_d, ciphered_data_q   : matrix_t;
    signal stage_en_d, stage_en_q             : std_logic_vector(0 to 10);
    signal round_keys_o      : roudkey_array_t;
    signal subBytes_r10_o    : matrix_t;

begin

 key_expand_i <= i_run and i_key_expand;
 ----------------------------------------------------------------------------------------
 k0 : entity keyExpansion(Pipeline)
 ----------------------------------------------------------------------------------------
    port map (clock       => clock,
            reset         => reset,
            i_sleep       => i_sleep,
            i_run         => key_expand_i,
            i_cipher_key  => i_cipher_key,
            o_round_keys  => round_keys_o);
 ----------------------------------------------------------------------------------------
 --Initial Round
 cipher_states_d(0) <= conv_state(round_keys_o(0) xor plain_data_q);
 ----------------------------------------------------------------------------------------
  Rounds : for I in 1 to 9 generate
 ----------------------------------------------------------------------------------------
  begin
   Round_i :  entity cipherRound
      port map (
        i_state  => cipher_states_q(I-1),
        i_round_key => round_keys_o(I),
        o_state => cipher_states_d(I));
  end generate Rounds;
 ----------------------------------------------------------------------------------------
 --Last Round
 s10_sub :  entity SubBytes(Parallel)
    port map (
      i_state  => cipher_states_q(9),
      o_state => subBytes_r10_o);

ciphered_data_d <= AddRoundKey(ShiftRows(subBytes_r10_o), round_keys_o(10));
----------------------------------------------------------------------------------------
```

```
sleep_logic : process (stage_en_q) is
-----------------------------------------------------------------------------------------
 variable tmp : std_logic;
 begin
    tmp := stage_en_q(0);
    for i in 1 to 10 loop
      tmp := tmp or stage_en_q(i);
    end loop;
    sleep_i <=  (not tmp);
end process sleep_logic;
-----------------------------------------------------------------------------------------
stage_en_d <=  i_run  &  stage_en_q(0 to 9);
-----------------------------------------------------------------------------------------
 sync_datapath : process (clock)
-----------------------------------------------------------------------------------------
  begin
  if rising_edge(clock) then
    if reset = '1' then
      plain_data_q      <= (others =>'0');
      stage_en_q        <= (others =>'0');
      cipher_states_q   <= (others =>(others=>(others =>'0')));
      ciphered_data_q   <= (others =>(others=>(others =>(others =>'0'))));
    else
        if i_sleep = '0' then   stage_en_q <= stage_en_d; end if;
        if i_sleep = '0' then   plain_data_q <= inverted(i_plain_data); end if;

        if stage_en_q(0) = '1' then cipher_states_q(0) <= cipher_states_d(0); end if;
        if stage_en_q(1) = '1' then cipher_states_q(1) <= cipher_states_d(1); end if;
        if stage_en_q(2) = '1' then cipher_states_q(2) <= cipher_states_d(2); end if;
        if stage_en_q(3) = '1' then cipher_states_q(3) <= cipher_states_d(3); end if;
        if stage_en_q(4) = '1' then cipher_states_q(4) <= cipher_states_d(4); end if;
        if stage_en_q(5) = '1' then cipher_states_q(5) <= cipher_states_d(5); end if;
        if stage_en_q(6) = '1' then cipher_states_q(6) <= cipher_states_d(6); end if;
        if stage_en_q(7) = '1' then cipher_states_q(7) <= cipher_states_d(7); end if;
        if stage_en_q(8) = '1' then cipher_states_q(8) <= cipher_states_d(8); end if;
        if stage_en_q(9) = '1' then cipher_states_q(9) <= cipher_states_d(9); end if;
        if stage_en_q(10) = '1' then ciphered_data_q <= ciphered_data_d; end if;

    end if;
 end if;
end process sync_datapath;
-----------------------------------------------------------------------------------------
o_ciphered_data <= conv_std_logic_vector_inverted(ciphered_data_q);
o_done <= sleep_i;
end architecture Pipeline;
```

Listing E.2: The hardware AES - Sequential architecture (back to Figure E.25).

```
architecture Sequential of aes128 is
    type control_state_t is (s_ready, s_init_round, s_cipher_rounds,s_final_round);
    signal state, next_state: control_state_t;
    signal init_i,key_expand_i,store_i: std_logic;
    signal round_counter: integer range 0 to 10;
    signal roundkeys_o : roudkey_array_t;
    signal cipher_state_d,cipher_state_q: matrix_t;
    signal sub_matrix_o,shift_matrix_o,mix_matrix_o,add_matrix_i,cipher_round_o: matrix_t;
    signal final_cipher_q: std_logic_vector(127 downto 0);
begin
-----------------------------------------------------------------------
CONTROL_FSM:process(state,i_run,round_counter)
-----------------------------------------------------------------------
begin
init_i <= '0';
store_i<= '0';
next_state <= state;
    case state is
        when s_ready=>
```

```
            init_i <= '1';
            if(i_run = '1') then
                next_state <= s_init_round;
            end if;
        when s_init_round=>
             next_state <= s_cipher_rounds;
        when s_cipher_rounds=>
            if(round_counter = 9) then
                 next_state<= s_final_round;
            end if;
        when s_final_round=>
            store_i<='1';
            next_state<= s_ready;
    end case;
end process CONTROL_FSM;
----------------------------------------------------------------------------
----------------------------------------------------------------------------
fsm_regs:process(clock)
----------------------------------------------------------------------------
begin
    if rising_edge(clock) then
        if reset = '1' then
            state <= s_ready;
        elsif i_sleep = '0' then
            state <= next_state;
        end if;
    end if;
end process fsm_regs;
----------------------------------------------------------------------------------------
key_expand_i <= i_run and i_key_expand;
----------------------------------------------------------------------------------------
k0: entity keyExpansion(Sequential)
----------------------------------------------------------------------------------------
    Port map( clock => clock,
           reset => reset,
           i_sleep => i_sleep,
           i_run=> key_expand_i,
           i_cipher_key=> i_cipher_key,
           o_round_keys => roundkeys_o);
----------------------------------------------------------------------------------------
--Inp_Mux:
----------------------------------------------------------------------------------------
cipher_state_d<=conv_state_inverted(i_plain_data) when init_i = '1' else cipher_round_o;
----------------------------------------------------------------------------------------
SubBytes0 :  entity SubBytes(Parallel)
----------------------------------------------------------------------------------------
    port map (
      i_state  => cipher_state_q,
      o_state => sub_matrix_o);
----------------------------------------------------------------------------------------
--ShifRows
----------------------------------------------------------------------------------------
shift_matrix_o <= ShiftRows(sub_matrix_o);
----------------------------------------------------------------------------------------
  MixColumns0 :  entity MixColumns
----------------------------------------------------------------------------------------
    port map (
      i_state => shift_matrix_o,
      o_state => mix_matrix_o);
----------------------------------------------------------------------------------------
Add_mux:process(state,cipher_state_q,mix_matrix_o,shift_matrix_o)
----------------------------------------------------------------------------------------
begin
case state is
    when s_ready=>
        add_matrix_i <= cipher_state_q;
    when s_init_round=>
        add_matrix_i <= cipher_state_q;
   when s_cipher_rounds=>
        add_matrix_i <= mix_matrix_o;
    when s_final_round=>
        add_matrix_i <= shift_matrix_o;
   end case;
end process Add_mux;
-------------------------------------------------------------------------------------
-- AddRoundKey
-------------------------------------------------------------------------------------
cipher_round_o<= AddRoundKey(add_matrix_i,roundkeys_o(round_counter));
-------------------------------------------------------------------------------------
DPATH_REGS:process(clock)
-------------------------------------------------------------------------------------
begin
    if rising_edge(clock) then
        if reset = '1' then
            cipher_state_q <= (others=>(others=>(others=>'0')));
        elsif i_sleep = '0' then
            cipher_state_q <= cipher_state_d;
        end if;

        if reset = '1' then
            final_cipher_q <= (others=>'0');
        elsif store_i = '1' and i_sleep = '0' then
            final_cipher_q <= conv_std_logic_vector_inverted(cipher_round_o);
        end if;


        if reset = '1' or init_i = '1' or store_i = '1'then
            round_counter <= 0;
        elsif  i_sleep = '0' then
            round_counter <= round_counter + 1;
        end if;

    end if;
end process DPATH_REGS;
-------------------------------------------------------------------------------------
o_ciphered_data <= final_cipher_q;
o_done          <= init_i;
end Sequential;
```

Listing E.3: AES Key Expansion - Pipeline architecture (back to E.5.2).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.all;
use aes_128_v1_00_a.SubWord;

entity keyExpansion is
    port (clock    : in std_logic;
      reset        : in std_logic;
      i_sleep      : in std_logic;
      i_run        : in std_logic;
      i_cipher_key : in std_logic_vector(127 downto 0);
      o_round_keys : out roudkey_array_t);
end entity keyExpansion;

architecture Pipeline of keyExpansion is

type byte_array_t  is array (0 to 9)  of std_logic_vector(7 downto 0);
type word_array_t  is array (0 to 9)  of word_t;
type rcon_array_t  is array (0 to 9)  of word_t;
type key_array_t   is array (0 to 43)   of word_t;
signal subWord_i, subWord_o         : word_array_t;
signal rcon_i                       : rcon_array_t;
signal sched_words_d, sched_words_q : key_array_t;
```

```vhdl
signal sleep_i                 : std_logic;
signal round_en_d, round_en_q:  std_logic_vector(0 to 9);
constant RCON:byte_array_t:=(x"01",x"02",x"04",x"08",x"10",x"20",x"40",x"80",x"1B", x"36");

begin
----------------------------------------------------------------------------------------
--Input cipher key
----------------------------------------------------------------------------------------
  sched_words_d(0) <= conv_word(i_cipher_key(127 downto 96))
                       when i_run = '1' else sched_words_q(0);
  sched_words_d(1) <= conv_word(i_cipher_key(95 downto 64))
                       when i_run = '1' else sched_words_q(1);
  sched_words_d(2) <= conv_word(i_cipher_key(63 downto 32))
                       when i_run = '1' else sched_words_q(2);
  sched_words_d(3) <= conv_word(i_cipher_key(31 downto 0))
                       when i_run = '1' else sched_words_q(3);
----------------------------------------------------------------------------------------
 SubWords : for w4k in 0 to 9 generate
----------------------------------------------------------------------------------------
  begin
    SubWord_4k : entity SubWord(Parallel)
      port map (
        i_word  => subWord_i(w4k),
        o_word => subWord_o(w4k));
  end generate SubWords;
----------------------------------------------------------------------------------------
  ScheduleRounds : for k in 0 to 9 generate
----------------------------------------------------------------------------------------
--SubWord()
subWord_i(k) <= sched_words_q(4*k+3);

--Rcon[k] xor rotWord(1)
rcon_i(k)(0) <= subWord_o(k)(1) xor RCON(k);
rcon_i(k)(1) <= subWord_o(k)(2);
rcon_i(k)(2) <= subWord_o(k)(3);
rcon_i(k)(3) <= subWord_o(k)(0);

sched_words_d(4*(k+1)+0)<=rcon_i(k) xor sched_words_q(4*k)
                        when round_en_q(k) = '1' else sched_words_q(4*(k+1)+0);
sched_words_d(4*(k+1)+1)<=rcon_i(k) xor sched_words_q(4*k) xor sched_words_q(4*k+1)
                        when round_en_q(k) = '1' else sched_words_q(4*(k+1)+1);
sched_words_d(4*(k+1)+2)<=rcon_i(k) xor sched_words_q(4*k) xor sched_words_q(4*k+1) xor
 sched_words_q(4*k+2)
                        when round_en_q(k) = '1' else sched_words_q(4*(k+1)+2);
sched_words_d(4*(k+1)+3)<= rcon_i(k) xor sched_words_q(4*k) xor sched_words_q(4*k+1) xor
 sched_words_q(4*k+2) xor sched_words_q(4*k+3)
                        when round_en_q(k) = '1' else sched_words_q(4*(k+1)+3);
end generate ScheduleRounds;
----------------------------------------------------------------------------------------
  sleep_logic : process (round_en_q) is
----------------------------------------------------------------------------------------
  variable tmp : std_logic;
  begin
    tmp := round_en_q(0);
    for I in 1 to 9 loop
      tmp := tmp or round_en_q(I);
    end loop;
    sleep_i <= (not tmp);
  end process sleep_logic;
----------------------------------------------------------------------------------------
  round_en_d <= i_run & round_en_q(0 to 8);
----------------------------------------------------------------------------------------
  DPATH_REGS : process (clock, reset) is
----------------------------------------------------------------------------------------
  begin
    if rising_edge(clock) then
        if reset = '1' then
          sched_words_q    <= (others =>(others=>(others=>'0')));
          round_en_q       <= (others => '0');
        else
          sched_words_q    <= sched_words_d;
          round_en_q       <= round_en_d;
        end if;
    end if;
  end process DPATH_REGS;
-----------------------------------------------------------------------------------------
  RoundKeys_o : for k in 0 to 10 generate
-----------------------------------------------------------------------------------------
    o_round_keys(k) <= conv_std_logic_vector(sched_words_q(4*k),
                                             sched_words_q(4*k+1),
                                             sched_words_q(4*k+2),
                                             sched_words_q(4*k+3));
  end generate RoundKeys_o;
-----------------------------------------------------------------------------------------
end architecture Pipeline;
-----------------------------------------------------------------------------------------
```

Listing E.4: AES Key Expansion - Sequential architecture (back to E.5.1).

```vhdl
architecture Sequential of keyExpansion is
type StateType is (s0_Ready,s1_SchedRounds,s2_FinalRound);
type sub_key_t is array (0 to 3) of word_t;
type byte_array_t is array (0 to 9) of std_logic_vector(7 downto 0);
type key_schedule_t  is array (0 to 43)   of word_t;

signal state, nextstate: StateType;
signal key_sel        : integer range 0 to 9;
signal key_next_sel   : integer range 1 to 10;
signal count_i,stoped_i  : std_logic;
signal subkey_i,subkey_o:sub_key_t;
signal subword_o,rcon_i :word_t;
signal sched_words_d,sched_words_q : key_schedule_t;
constant RCON:byte_array_t:=(x"01",x"02",x"04",x"08",x"10",x"20",x"40",x"80",x"1B", x"36");
begin
-----------------------------------------------------------------------------------------
FSM:process(state,i_run,key_sel)
-----------------------------------------------------------------------------------------
begin
 stoped_i <= '0';
 nextstate <= state;
 count_i <= '1';
    case state is
        when s0_Ready=>
            stoped_i <= '1';
            if(i_run = '1') then
                nextstate <= s1_SchedRounds;
            end if;
        when s1_SchedRounds=>
            if(key_sel = 8) then
                nextstate <= s2_FinalRound;
            end if;
        when s2_FinalRound=>
            count_i <= '0';
            nextstate <= s0_Ready;
    end case;
end process FSM;
-----------------------------------------------------------------------------------------
DPATH_REGS:process(clock)
-----------------------------------------------------------------------------------------
begin
    if rising_edge(clock) then

        if reset= '1' then
            state <= s0_Ready;
        elsif i_sleep = '0' then
            state <= nextstate;
        end if;

        if reset= '1' then
```

```vhdl
            sched_words_q <= (others=>(ZERO_WORD));
        else
            if i_sleep = '0' and i_run = '1' then
                sched_words_q(0) <= conv_word(i_cipher_key(127 downto 96));
                sched_words_q(1) <= conv_word(i_cipher_key(95 downto 64));
                sched_words_q(2) <= conv_word(i_cipher_key(63 downto 32));
                sched_words_q(3) <= conv_word(i_cipher_key(31 downto 0));
            end if;

            if i_sleep = '0' and stoped_i = '0' then
                sched_words_q(4 to 43) <= sched_words_d(4 to 43);
            end if;

        end if;

         if reset= '1' or stoped_i = '1' then
            key_sel         <= 0;
            key_next_sel    <= 1;
        elsif i_sleep = '0' and count_i = '1' then
            key_sel         <= key_sel +1;
            key_next_sel    <= key_next_sel +1;
        end if;
    end if;
end process DPATH_REGS;
---------------------------------------------------------------------------------------
-- Key Round
---------------------------------------------------------------------------------------
    --Input Mux
    subkey_i(0) <= sched_words_q(key_sel*4);
    subkey_i(1) <= sched_words_q(key_sel*4+1);
    subkey_i(2) <= sched_words_q(key_sel*4+2);
    subkey_i(3) <= sched_words_q(key_sel*4+3);
    -------------------------------------
    SubWord3: entity SubWord(Parallel)
    -------------------------------------
        port map ( i_word  => subkey_i(3),o_word => subword_o);
    -------------------------------------
    --rcon[k] xor rotWord()
    -------------------------------------
     rcon_i(0) <= subword_o(1) xor RCON(key_sel);
     rcon_i(1) <= subword_o(2);
     rcon_i(2) <= subword_o(3);
     rcon_i(3) <= subword_o(0);

  subkey_o(0) <= rcon_i xor subkey_i(0)                                                ;
  subkey_o(1) <= rcon_i xor subkey_i(0) xor subkey_i(1)                                ;
  subkey_o(2) <= rcon_i xor subkey_i(0) xor subkey_i(1) xor subkey_i(2)                ;
  subkey_o(3) <= rcon_i xor subkey_i(0) xor subkey_i(1) xor subkey_i(2) xor subkey_i(3);
 ---------------------------------------------------------------------------------------
 out_mux:process(key_next_sel,subkey_o)
---------------------------------------------------------------------------------------
 begin
     sched_words_d(key_next_sel*4)     <= subkey_o(0);
     sched_words_d(key_next_sel*4 + 1) <= subkey_o(1);
     sched_words_d(key_next_sel*4 + 2) <= subkey_o(2);
     sched_words_d(key_next_sel*4 + 3) <= subkey_o(3);
  end process out_mux;
---------------------------------------------------------------------------------------
  RoundKeys_o : for k in 0 to 10 generate
---------------------------------------------------------------------------------------
    o_round_keys(k) <= conv_std_logic_vector(sched_words_q(4*k),
                                          sched_words_q(4*k+1),
                                          sched_words_q(4*k+2),
                                          sched_words_q(4*k+3));
  end generate RoundKeys_o;
---------------------------------------------------------------------------------------
end Sequential;
```

Listing E.5: AES Cipher Round - architecture description using VHDL.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library ieee;
use ieee.std_logic_1164.all;
library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.all;
use aes_128_v1_00_a.SubBytes;
use aes_128_v1_00_a.MixColumns;

entity CipherRound is
  port ( i_state     : in  matrix_t;
        i_round_key  : in  std_logic_vector(127 downto 0);
        o_state      : out matrix_t);
end entity CipherRound;

architecture Pipeline of CipherRound is
  signal sub_bytes_o    : matrix_t;
  signal shift_rows_o   : matrix_t;
  signal mix_columns_o  : matrix_t;
begin

SubBytes_i : entity SubBytes(Parallel)
    port map (
      i_state  => i_state,
      o_state => sub_bytes_o);

 shift_rows_o <= ShiftRows(sub_bytes_o);

MixColumns_i : entity MixColumns
    port map (
      i_state  => shift_rows_o,
      o_state => mix_columns_o);

o_state    <= AddRoundKey(mix_columns_o, i_round_key);

end architecture Pipeline;
```

Listing E.6: AES SubBytes - architecture description using VHDL.

```
library ieee;
use ieee.std_logic_1164.all;
library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.all;
use aes_128_v1_00_a.SubWord;

entity SubBytes is
  port (
    i_state : in  matrix_t;
    o_state : out matrix_t);
end entity SubBytes;

architecture Parallel of SubBytes is
begin
SubWord0 :  entity SubWord(Parallel)
      port map (
        i_word  => i_state(0),
        o_word => o_state(0));
SubWord1 :  entity SubWord(Parallel)
      port map (
        i_word  => i_state(1),
        o_word => o_state(1));
```

```
SubWord2 :  entity SubWord(Parallel)
      port map (
        i_word  => i_state(2),
        o_word => o_state(2));
SubWord3 :  entity SubWord(Parallel)
      port map (
        i_word  => i_state(3),
        o_word => o_state(3));
end architecture Parallel;
```

Listing E.7: AES SubWord - architecture description using VHDL.

```
library IEEE;
use IEEE.std_logic_1164.all;
library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.all;
use aes_128_v1_00_a.sbox;

entity SubWord is
  port (
    i_word : in  word_t;
    o_word : out word_t);
end entity SubWord;

architecture Parallel of SubWord is
begin
  sbox_b0 :  entity sbox(DCanright)
    port map (
      i_data  => i_word(0),
      o_data => o_word(0));
  sbox_b1 :  entity sbox(DCanright)
    port map (
      i_data  => i_word(1),
      o_data => o_word(1));
  sbox_b2 :  entity sbox(DCanright)
    port map (
      i_data  => i_word(2),
      o_data => o_word(2));
  sbox_b3 :  entity sbox(DCanright)
    port map (
      i_data  => i_word(3),
      o_data => o_word(3));
end architecture Parallel;
```

Listing E.8: AES Sbox - architecture description using VHDL (back to Figure E.8).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.byte_t;

entity sbox is
  port ( i_data  : in  byte_t;
         o_data : out byte_t);

end entity sbox;

architecture DCanright of SBox is
  type byte_array_t is array (0 to 7) of byte_t;

-- to convert between polynomial (A^7...1) basis A & normal basis X
-- or to basis S which incorporates bit matrix of Sbox
```

```vhdl
  constant A2X : byte_array_t := (x"98", x"F3", x"F2", x"48", x"09", x"81", x"A9", x"FF");
  constant X2S : byte_array_t := (x"58", x"2D", x"9E", x"0B", x"DC", x"04", x"03", x"24");
 --multiply in GF(2^2), using normal basis (Omega^2,Omega)
  function G4_mul (
    x : std_logic_vector(1 downto 0);
    y : std_logic_vector(1 downto 0))
    return std_logic_vector is
    variable a, b, c, d, e, p, q : std_logic;
  begin
    a := x(1); b := x(0);
    c := y(1); d := y(0);
    e := (a xor b) and (c xor d);
    p := (a and c) xor e;
    q := (b and d) xor e;
    return p & q;
  end function G4_mul;

 --scale by N = Omega^2 in GF(2^2), using normal basis (Omega^2,Omega)
  function G4_scl_N (
    x : std_logic_vector(1 downto 0))
    return std_logic_vector is
  begin
    return (x(0) & (x(0) xor x(1)));
  end function G4_scl_N;

  --scale by N^2 = Omega in GF(2^2), using normal basis (Omega^2,Omega)
  function G4_scl_N2 (
    x : std_logic_vector(1 downto 0))
    return std_logic_vector is
  begin
    return ((x(0) xor x(1)) & x(1));
  end function G4_scl_N2;

-- square in GF(2^2), using normal basis (Omega^2,Omega)
-- NOTE: inverse is identical
  function G4_sq (
    x : std_logic_vector(1 downto 0))
    return std_logic_vector is
  begin
    return (x(0) & x(1));
  end function G4_sq;

--multiply in GF(2^4), using normal basis (alpha^8,alpha^2)
  function G16_mul (
    x : std_logic_vector(3 downto 0);
    y : std_logic_vector(3 downto 0))
    return std_logic_vector is
    variable a, b, c, d, e, p, q : std_logic_vector(1 downto 0);
  begin
    a := x(3 downto 2); b := x(1 downto 0);
    c := y(3 downto 2); d := y(1 downto 0);
    e := G4_mul(a xor b, c xor d);
    e := G4_scl_N(e);
    p := (G4_mul(a, c) xor e);
    q := (G4_mul(b, d) xor e);
    return p & q;
  end function G16_mul;

--square & scale by nu in GF(2^4)/GF(2^2), normal basis (alpha^8,alpha^2)
--nu = beta^8 = N^2*alpha^2, N = w^2
  function G16_sq_scl (
    x : std_logic_vector(3 downto 0))
    return std_logic_vector is
    variable p, q : std_logic_vector(1 downto 0);
  begin
    p := G4_sq(x(3 downto 2) xor x(1 downto 0));
    q := G4_scl_N2(G4_sq(x(1 downto 0)));
    return p & q;
  end function G16_sq_scl;

```

```vhdl
--inverse in GF(2^4), using normal basis (alpha^8,alpha^2)
  function G16_inv (
    x : std_logic_vector(3 downto 0))
    return std_logic_vector is
    variable a,b,c,d,e,p,q : std_logic_vector(1 downto 0);
  begin
    a := x(3 downto 2); b := x(1 downto 0);
    c := G4_scl_N(G4_sq(a xor b));
    d := G4_mul(a, b);
    e := G4_sq(c xor d);
    p := G4_mul(e, b);
    q := G4_mul(e, a);
    return p & q;
  end function G16_inv;

 --inverse in GF(2^8), using normal basis (d^16,d)
  function G256_inv (
    x : std_logic_vector(7 downto 0))
    return std_logic_vector is
    variable a,b,c,d,e,p,q : std_logic_vector(3 downto 0);
  begin
    a := x(7 downto 4); b:= x(3 downto 0);
    c := G16_sq_scl(a xor b);
    d := G16_mul(a, b);
    e := G16_inv(c xor d);
    p := G16_mul(e, b);
    q := G16_mul(e, a);
    return p & q;
  end function G256_inv;

--convert to new basis in GF(2^8), i.e., bit matrix multiply
  function G256_newbasis (
    x : byte_t;
    b : byte_array_t)
    return byte_t is
    variable y : byte_t;
  begin
    y := (others=>'0');
    for i in 7 downto 0 loop
      if x(7-i) = '1' then
        y := y xor b(i);
      end if;
    end loop;
    return y;
  end function G256_newbasis;

  signal t0 : byte_t;
  signal t1 : byte_t;
  signal t2 : byte_t;

begin

  t0 <= G256_newbasis(i_data, A2X);
  t1 <= G256_inv(t0);
  t2 <= G256_newbasis(t1, X2S);

  o_data <= t2 xor x"63";
end architecture DCanright;
```

Listing E.9: AES MixColumns description using VHDL (Figure E.16).

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library ieee;
use ieee.std_logic_1164.all;
library aes_128_v1_00_a;
use aes_128_v1_00_a.aes128Pkg.all;
use aes_128_v1_00_a.mixWord;

entity MixColumns is
  port (
    i_state  : in  matrix_t;
    o_state : out matrix_t);
end entity MixColumns;

architecture Structural of MixColumns is
begin
Column0: entity mixWord(Structural)
      port map (i_word  => i_state(0),
                 o_word => o_state(0));

Column1: entity mixWord(Structural)
      port map (i_word  => i_state(1),
                 o_word => o_state(1));

Column2: entity mixWord(Structural)
      port map (i_word  => i_state(2),
                 o_word => o_state(2));

Column3: entity mixWord(Structural)
      port map (i_word  => i_state(3),
                 o_word => o_state(3));
end architecture Structural;
```

Appendix F

Features from Segment Test

In this appendix, we describe some of the hardware accelerated FAST datapath design considerations, while applying the concept of asynchronous-synchronous design. Such datapath is fundamentally composed of two IPS, which implement the detection of key points, here referred as FAST for simplicity, and the corresponding non-maximum suppression here referred as NMS. Storing of the output results of this datapath uses a 32-bit HW-FIFO, and for this reason, it's not discussed in this Appendix, as it was already discussed in Chapter 4. To allow operating over distinct image sizes and experimental conditions, the developed IPs provide a configuration interface composed of three parameter registers, and a device control register. Figure F.1 shows a logic diagram that describes this interface.

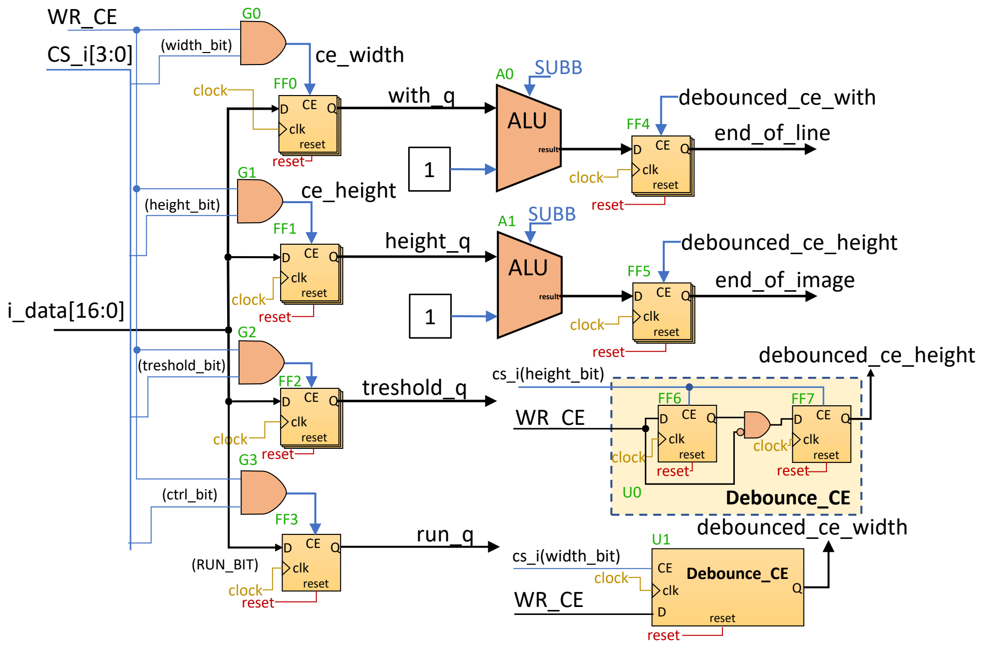


Figure F.1: HW-Fast or HW-NMS parameter registers.

To write in the register area, the chip select bus (CS_i) selects one of the four existing registers defining: the image resolution, through the *with_q* (FF0) and *height_q* (FF1) registers; the threshold value (FF2) used as a parameter in FAST; and finally, the control register (FF3) that allows starting and stopping the IP operation by asserting the *run_q* bit. When the write enable clock signal (WR_CE) is active for at least one clock pulse, the combination of G0 to G3, selects the CE input in the corresponding register, and therefore stores the value of the *i_data* input. Writing to FF0 and FF1 establishes the image parameters, and in response, using U0 and U1, it establishes the read and write address limits for the 6 lines (lines 1 to 6) of the IP FAST local storage. Similar operation is observed in the NMS but in this time used only in lines 1 and 2 of the local storage.

F.1 Feature Detection

Feature Detection implements the FAST algorithm up to the scoring and contiguity checking stages. Such design operates on seven image lines provided locally which are replaced by new lines as the Line zero pixels advance in the datapath. For this purpose, it implements seven true dual-port RAMs, U0 to U6, as it can be seen in the logic diagram of Figure F.2. To control the flow of image lines, the design implements the address controller, U7 in the figure, which generates the read and write addresses according to the handshakes it performs between: the operations of writing pixels on the line 0; and the advancement the data in the Bresenham to the *classifilter* unit.

To form a Bresenham circle, the FAST's datapath implements a matrix of registers with seven rows and seven columns, which operates as a shift register of seven positions per row. The pixel in the center of the matrix, marked in green, is the target of the FAST algorithm and will be compared with the sixteen pixels on the circle periphery, marked in orange, to determine whether the center is darker or brighter than each of the pixels in orange.

The comparison step is implemented by the *classifilter* unit, U8 in Figure F.2. Beyond this point data flow is divided between the light and dark comparisons, being distributed concurrently by the contiguity checking and scoring units that implement the two final steps of the feature detection. The output result of each unit converges to a single data stream that determines the final score of the central pixel, using U11 for handshake, and A0 and M0 for choosing one of the two flows, and FF49 for storing the chosen score.

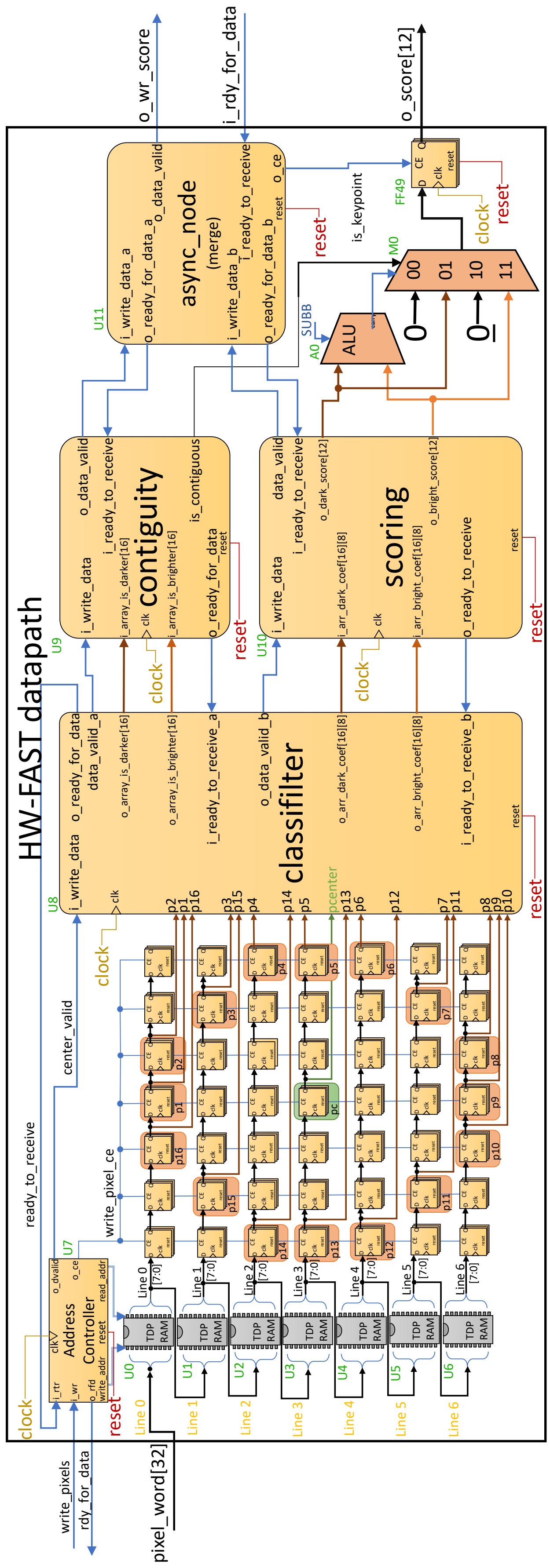


Figure F.2: Hardware Feature detection datapath.

F.1.1 Address Controller

The address controller synchronizes the FAST input pixel stream with the image size, by receiving 4-pixel words at the input (32-bit) and copying the received pixels across the seven lines of local storage. This component is also used in the NMS IP to synchronize the input scores and distribute the received values over the two lines of local storage. For better understanding, the descriptions about the address controller refer to its use in the FAST datapath, and a similar operation can be observed in the NMS datapath. Figure F.3 shopws a simplified diagram of the address controller.

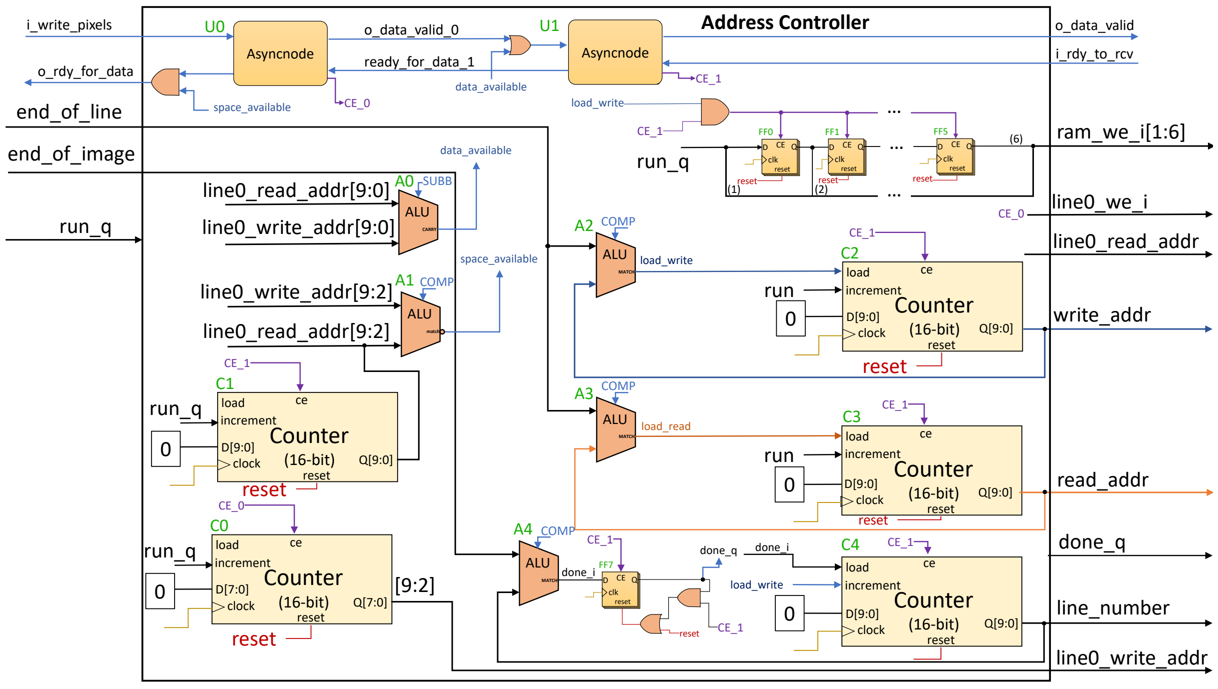


Figure F.3: Address controller hardware block diagram.

While writing to line zero, the ready for data flag is mandatory, and with the logic value ’1’ in the write pixels input, U0 enables the CE_O signal at the input of the C0 counter, for at least one clock pulse. The same signal is used to enable writing to RAM U0 of the FAST datapath (be means of line0_we_i), and in the next clock cycle, four pixels are written to line zero and C0 advances one unit. The output of this counter is then shifted two positions to the left to generate the next write address that advances at the rate of four bytes, in response to the writing of four pixels.

At the same time, the data valid output of U0 is asserted, that triggers the handshake with U1, when the flag ready for data is also asserted. In response, U1 activates the CE_1 signal that increments the read address (i.e., line0_read_addr) in C1. This address is used to send data from the line L0 to the datapath matrix, thus reducing the stream of data from 4 bytes to 1 byte. For this reason, the absolute read address in L0 is divided into bits [9:2] to form the word address, and at bits [1:0] to select byte 0 to 3 at the output (not represented). When read address reaches the value of the write address counter, the data available flag in A0 is deasserted, thus stopping pixel read operations in U1. If otherwise the writing address reaches the reading address with one line in advance, the space available flag in A1 is deasserted, to prevent writing in addresses that where not yet read.

Similarly, counters C2 and C3 are used to copy 1-byte pixels from line 'i' to line 'i+1' before a new pixel takes the line 'i' space at the current read address. In this way, the read address is always one position ahead of the write address, and to avoid propagating pixels that where not yet written, the *ram_we_i* signal uses the six-position shift register, (FF0 to FF5), which enables writing to line 'i+1' whenever the previous line 'i' reaches the value of e*nd_of_line* register. These registers keep write enabled in all RAMs after the first seven lines of the image. At each increment in counters C2 and C3 , A2 and A3 compare the output values with the end of line register, activating the load signal of the respective counter, when reaching the end of the line. Whenever the writintg address reaches the end of a line, the *load_write* signal is also used to increment C4 which counts the number of lines processed. This counter is reset to zero whenever the count value reaches the end of image register in A4 and the done output is asserted. Ultimately, the read address and line number counters are used to generate the x and y coordinates of the center pixel.

F.1.2 Classifier Filter

The *classifilter* block operates on data from the Bresenham circle in the FAST datapath and tests whether the central pixel is brighter or darker than each of the sixteen pixels. For this, it implements a parallel of bright and dark comparisons using sixteen *classifilter_i* modules. Figure F.4 shows the block diagram of this component. The comparison between the central pixel and the adjacent pixel is obtained by subtracting their intensities, in A0 and A1, for darker or brighter respectively. To eliminate proximity cases, the difference value is further subtracted from the threshold parameter, in A2 and A3. Finally, the resulting values are compared with zero in A4 and A5, being nulled in the output whenever the value is less than zero.

Three active clock cycles are required to complete the computation between the 16 pixels and the central pixel, and at the output of this component it can be seen a data stream that is distinguished by the values of brighter and darker. Each of the streams includes an array of coefficients composed by the differences between center and circle pixels, and an array of flags that indicate whether the center is brighter or darker than each pixel in the circle. To control the data flow, three asynchronous nodes are used, U0 to U2. At the classifilter output, the data flow is divided between contiguity testing and score calculation, and as such, U2 implements a split architecture that provides two distinct handshakes. In these, the data advance is only signaled at the output, when the handshake of A and B channels has been performed.

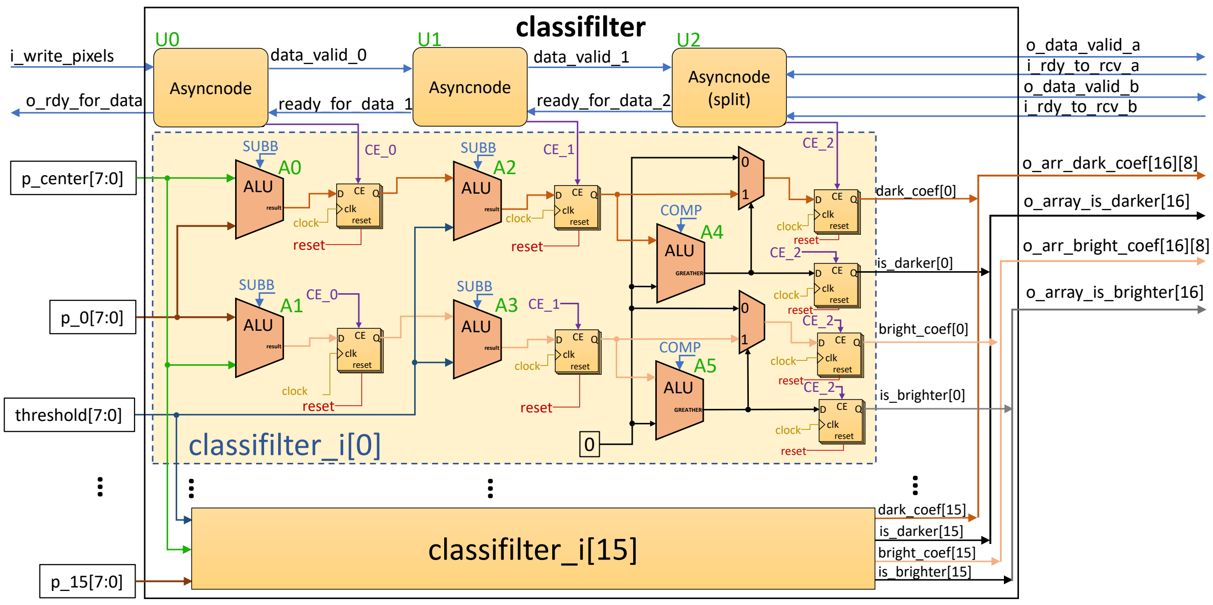


Figure F.4: Classifier filter hardware block diagram.

F.1.3 Contiguity Checking

The block contiguity check marks whether the central pixel is a key point in the image, in a continuity of 9 in 16 pixels (i.e., using FAST9_16). This calculation is performed simultaneously on both bright and dark streams, using sixteen comparisons per stream, composed of nine consecutive elements extracted from each flag array. It starts at the subset of elements 0 to 8 and advances one position for each comparison, ending with flag subset 15 concatenated with subset 0 to 7. Figure F.5 describes the architecture of this component using a diagram of logical elements.

The result of the sixteen comparisons, A0 to A15, is stored in register FF0, and in the next cycle, this value is compared with zeros. If there is at least one comparison flag active, which signals a contiguity of 9 out of 16 pixels, the *is_contiguous* signal is asserted at the output two clock cycles later. The two delay cycles implement data alignment with the scoring unit, which operates concurrently on the data stream, and requires four active clock cycles.

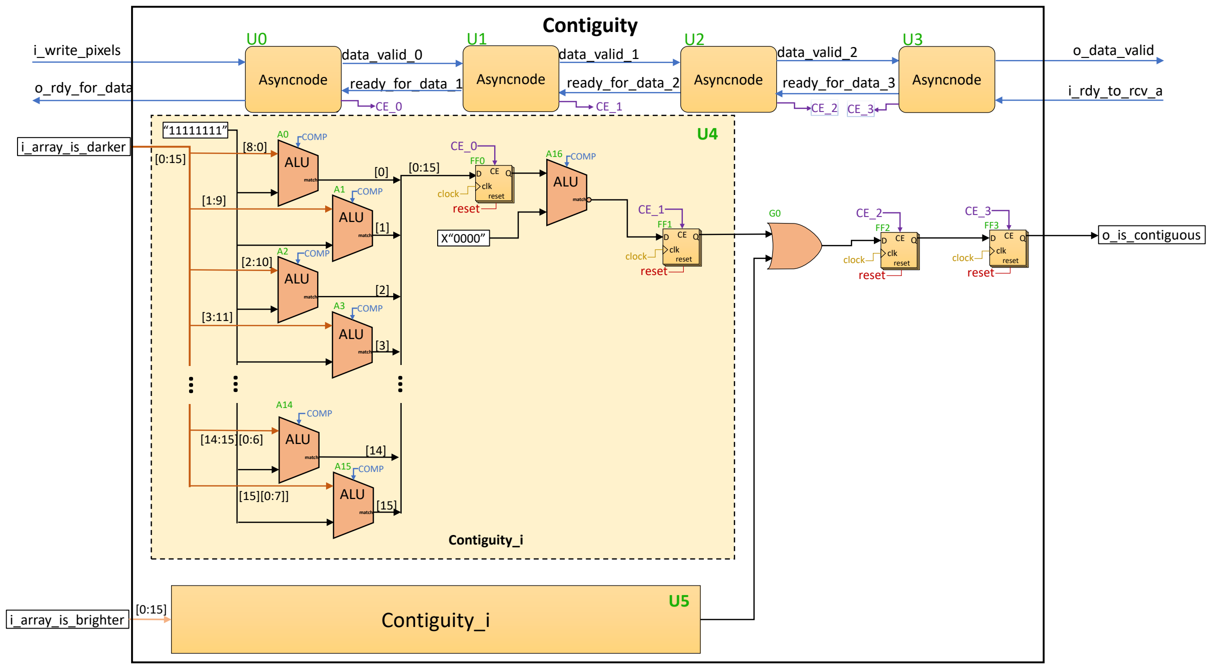


Figure F.5: Contiguity check hardware block diagram.

F.1.4 Scoring

The scoring unit operates on the set of differences between center and pixel in the circle to implement the sum of the elements for the bright and dark data streams. Figure F.6 describes the internal implementation of this unit through a diagram of logic elements. Each data stream is composed of sixteen elements of 8-bit, which in the first clock cycle, require eight sums that are stored in 9-bit registers FF0 to FF7. In the next clock cycle, four sums are required to receive the previous data, and results are stored in 10-bit registers FF9 to FF11. In the third cycle only two sums are required, with results stored in 11-bit registers FF12 and FF13, and lastly, the final sum is stored in the fourth cycle in the 12-bit register FF14. Components U0 to U3 handshake between the four phases, and two *scoring_i* units, U4 and U5, compute the coefficients of the bright and dark data streams.

F.1.5 Final Score

The last processing phase converges the two data streams to a single output, producing the final score for the pixel in the center of the bresenham circle. Figure F.7 shows the logical diagram of this final computing phase. To receive the two data streams from the contiguity and scoring units, this phase handshakes using a merge architecture in U0, which only confirms the reception of data on channels A and B after both inputs are simultaneously available. The final score will be set to the highest value between the bright and

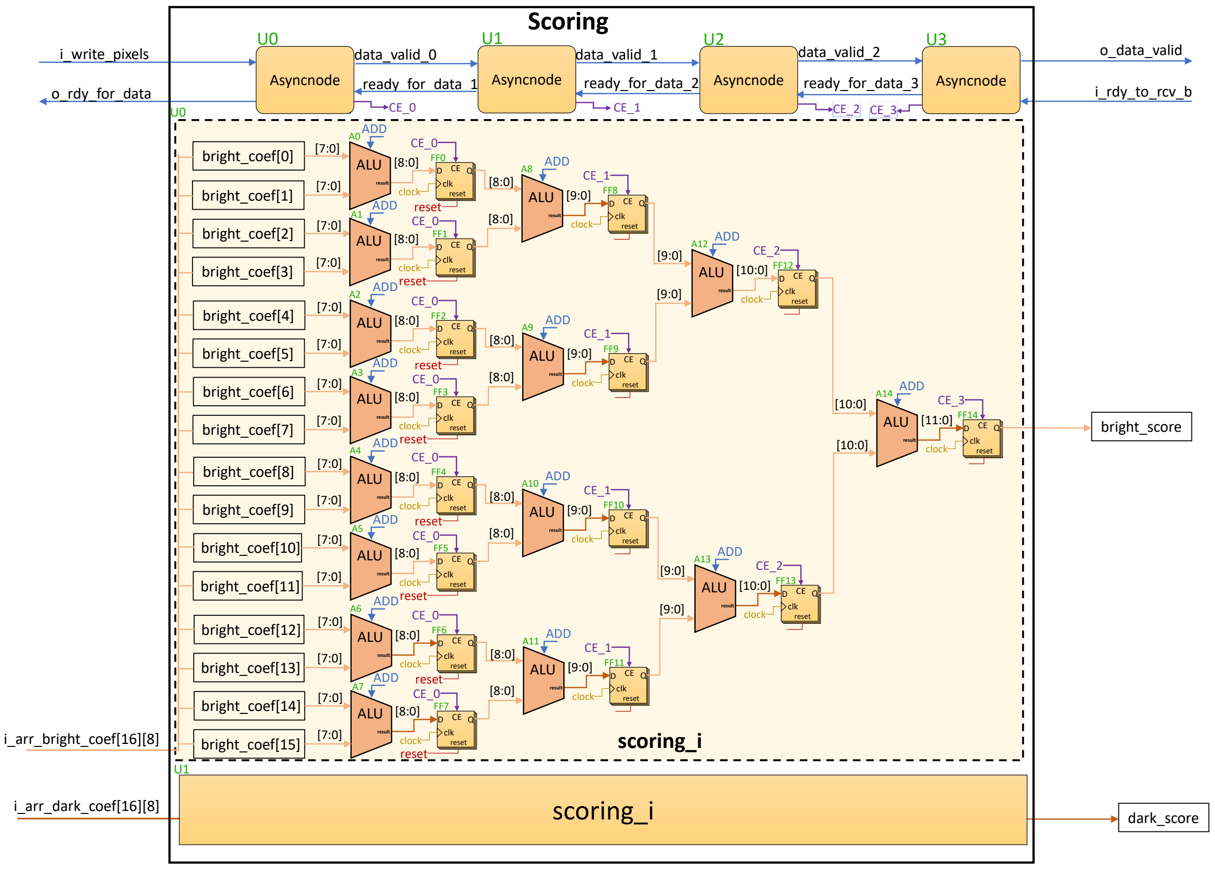


Figure F.6: Scoring hardware block diagram.

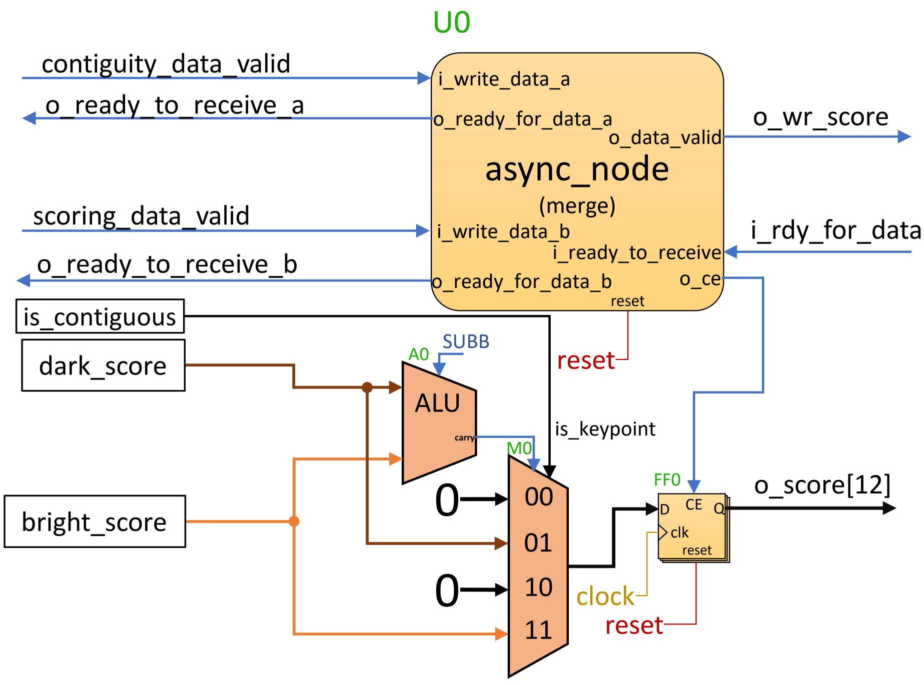


Figure F.7: Feature detection final score.

dark scores if the center pixel is considered a key point, otherwise the output will be set to null. For this, the M0 is used for the selection of four inputs to one output, and A0 compares which of the two scores is the highest absolute value.

F.2 Non-maximum Suppression

The NMS datapath shows similarities with the FAST design, by reusing the same local storage model, here only considering three lines, and the matrix of registers that aligns the central score to the adjacent values, before submitting to the suppression computation. The address controller U3, is generally used with the same image parameters as the FAST datapath, and in this case, the value of the read and line addresses are used to produce the x and y coordinates, respectively. Figure F.8 shows the logical element diagram that describes the NMS datapath.

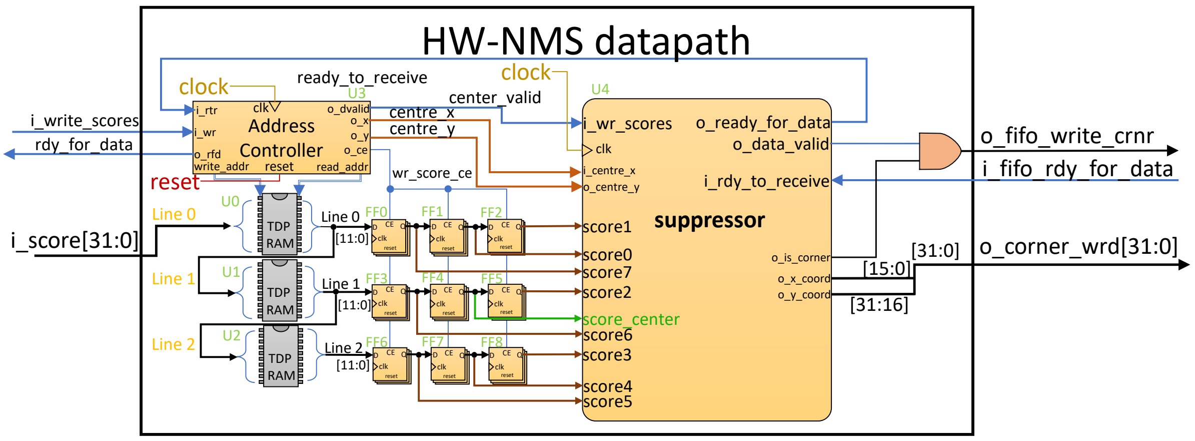


Figure F.8: Non-Maximum suppression datapath.

In that figure, it can be seen the three lines of local storage, U0 to U2, that feed the matrix of registers composed by FF0 to FF8. When the source of scores is connected to the FAST IP, as opposed to the system bus, the L0 line can be suppressed and the input *i_scores* is reduced to 12-bit range. In such case, it can be forwarded directly to the matrix of registers without passing by L0. Upon reaching the center position in the matrix of scores, the *center_valid* signal is asserted at the input of U4, and the suppressor initiates the computation of the first set of scores.

F.2.1 Suppressor

In Figure F.9 it can be seen the logic diagram that describes the suppressor used in the NMS datapath. Here, the input of scores is forwarded to the ALUs A0 to A7, to subtract each adjacent score with the value of the central score. Upon receiving the write_scores signal at the input of U0, the CE_0 signal activates the writing of the carry flags in the FF0 to FF7 registers.

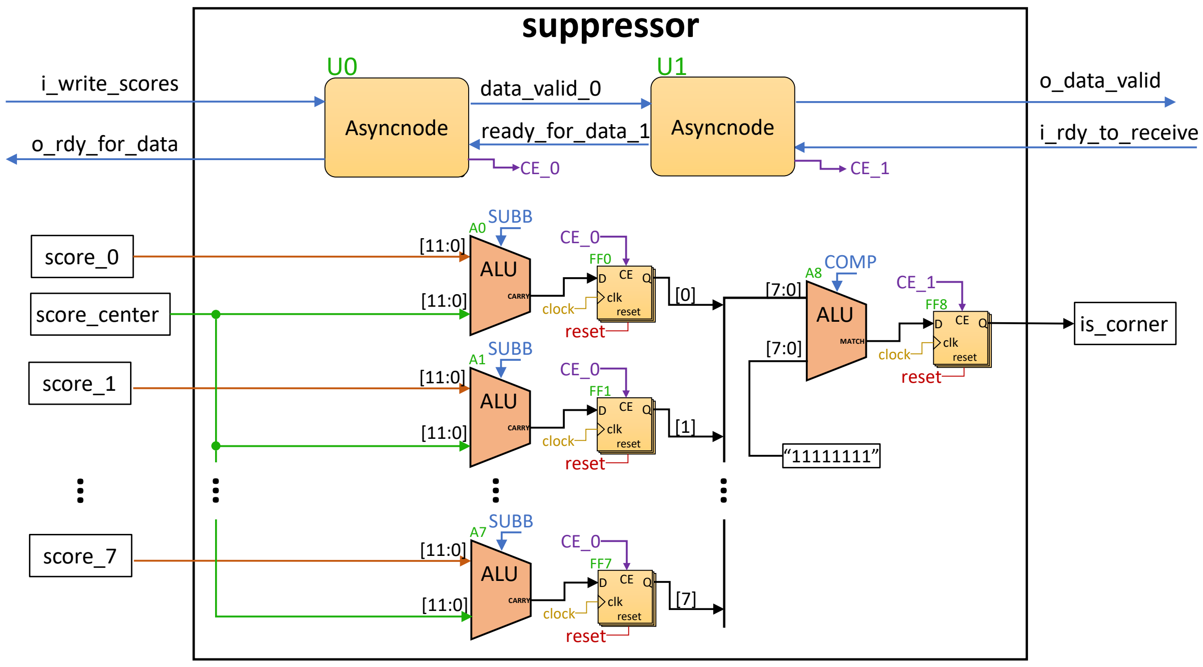


Figure F.9: Suppression hardware block diagram.

When the score of the central pixel is higher than the adjacent scores, in each ALU the carry flag that results from the subtraction becomes asserted. If all flags are set at the input of A8, the match flag is asserted and with the logical value '1' of CE_1, the is_corner output is set in the next clock cycle. In such case, the central score is considered a corner in the image. If otherwise, a single carry flag is deasserted as result of a higher adjacent score, the comparison in A8 will fail, and the center score is not considered a corner.

F.3 Design Co-simulation

In the first design phase, the HAL-ASOS methodology proposes a functional simulation, to anticipate the first contact with the application to be developed, before commitment to the underlying hardware. During this phase, decisions can be made with less risk of compromising qualitative results, since it is in the best position to implement the necessary changes. Simulations performed where carried out within the development phases of the application discussed in Chapter 5. On the software side, the Task classes where parameterized with appropriate qualifiers for the co-simulation and the application was compiled to execute in the host development system.

On the synthesis tool, a block design was created using the required accelerator for the co-simulation environment, while using the Vivado tool (i.e., V4_00_C_v). Figure F.10 shows this block design, where it can be seen two HW-Tasks connected to each other, and in its turn each attached to an HAL-ASOS accelerator

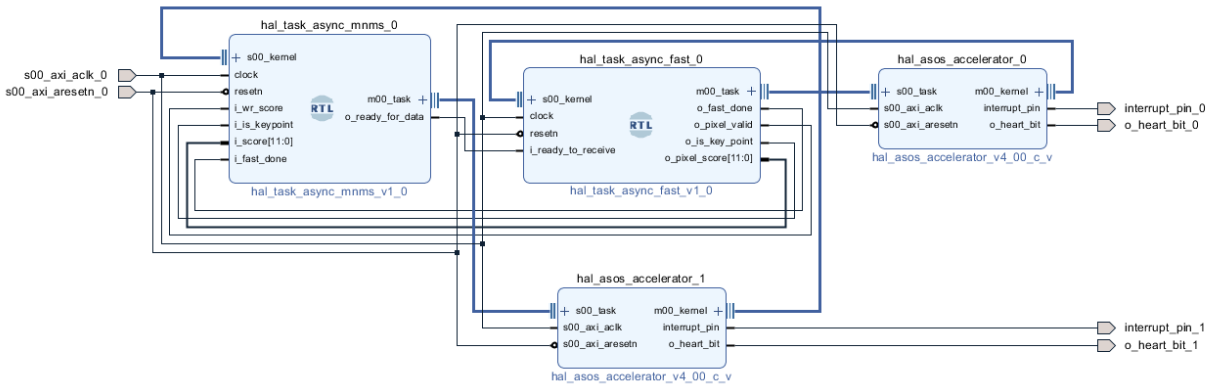
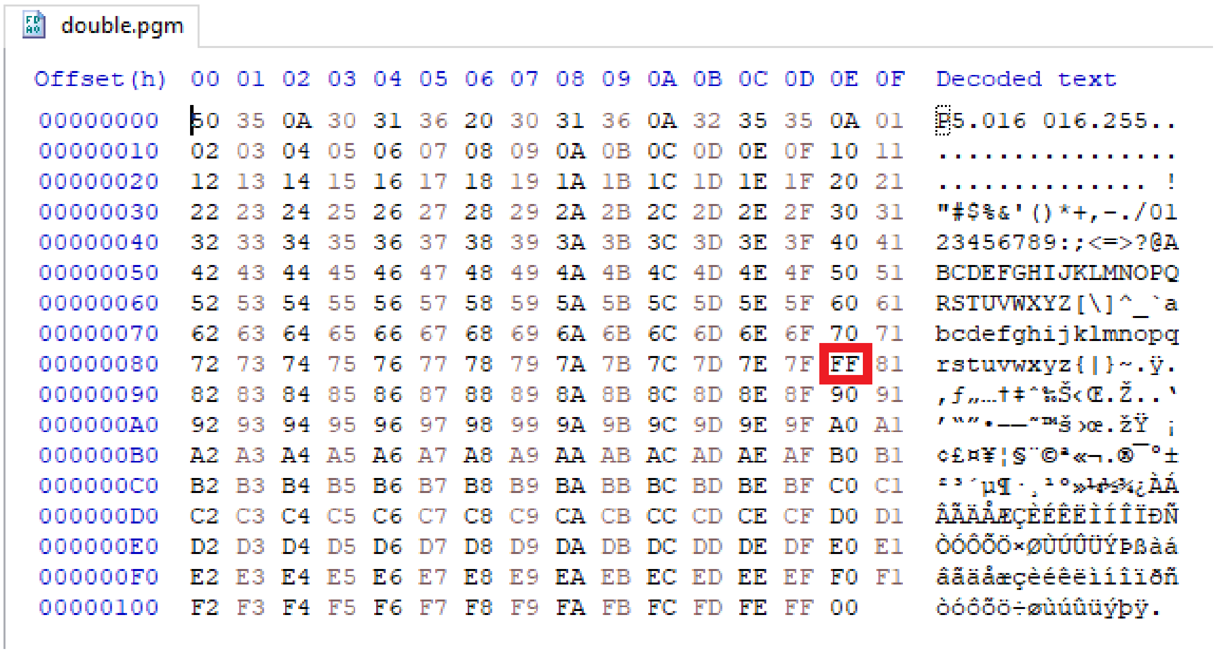


Figure F.10: Feature Detection co-simulation - Block design using Vivado.

In the top-level of the HW-Tasks, one can notice additional interconnect signals that allow the FAST task to write calculated scores in the NMS task, while implementing bidirectional data handshake according to the asynchronous model. For a better comprehension of the simulation results, a simplified image file was created using a 16x16 resolution and containing only one corner. The contents of such file can be seen in Figure F.11.

The first 15 bytes compose the 'pgm' file header, which includes the version (i.e., P5), the image resolution (i.e., 016x016), and the maximum pixel intensity (i.e., 255). A new line character (0x0A) precedes the first image pixel that occupies the position 0x0F in the file. The image contained in the file represents a grayscale square that darkens by an intensity level for each pixel. To insert a corner, the value 0x80 has been replaced by the maximum value '0xFF' (squared red in the figure). The coordinate of this pixel in the image is given by X:0x0F and Y:0x07, and differs from the file by the space required for the header. If removing the first line, the pixel Y coordinate is updated to 0x07 instead of 0x08, and by advancing the first pixel in line 0 ('0x01') by one position to start in 0 at the X coordinate, the '0xFF' pixel also advances one position, moving to the 0x0F coordinate.


double.pgm

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  50 35 0A 30 31 36 20 30 31 36 0A 32 35 35 0A 01  P5.016 016.255..
00000010  02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11  ................
00000020  12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21  .............. !
00000030  22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31  "#$%&'()*+,-./01
00000040  32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41  23456789:;<=>?@A
00000050  42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51  BCDEFGHIJKLMNOPQ
00000060  52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61  RSTUVWXYZ[\]^_`a
00000070  62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71  bcdefghijklmnopq
00000080  72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F FF 81  rstuvwxyz{|}~.ÿ.
00000090  82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91  ‚ƒ„…†‡ˆ‰Š‹Œ.Ž..‘
000000A0  92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1  ’“”•–—˜™š›œ.žŸ ¡
000000B0  A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1  ¢£¤¥¦§¨©ª«¬.®¯°±
000000C0  B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1  ²³´µ¶·¸¹º»¼½¾¿ÀÁ
000000D0  C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1  ÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑ
000000E0  D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1  ÒÓÔÕÖ×ØÙÚÛÜÝÞßàá
000000F0  E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1  âãäåæçèéêëìíîïðñ
00000100  F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 00     òóôõö÷øùúûüýþÿ.
```

Figure F.11: Feature Detection co-simulation - Single corner input file.

Figure F.12 shows the wave plot that results from the co-simulation of the application while using the block design of Figure F.10. In this simulation we have considered a clock period of 10 nanoseconds, and it can be observed that the input achieves a pixel rate of four pixels every seven clock cycles. In this way, it is possible to observe that while no new pixels enter the datapath, those that have already been accepted advance in the processing until they reach the output.

In the sixth line of the signal column, it is possible to observe the arrangement that makes up the matrix of registers from which the Bresenham circle is obtained, has highlighted in red. At the 14,275.00 nanosecond marker, the pixel with 0xFF intensity reaches the center line in the matrix. At the 14,335.00 nanoseconds marker, the same pixel reached the center of the matrix, and is sent to the *classifilter* unit together with the pixels that are on the periphery of the circle. Three clock cycles later, at the 14,365.00 nanoseconds of simulation time , despite only one pixel entered the datapath, the *array_class_is_dark* shows all flags marked as active (i.e., 0xFFFF). Four clock cycles later, the value 0x610 appears in the output of the score unit (i.e., *dark_score[11:0]*), accompanied by the flag *is_contiguous*. In the next clock cycle, the highest score is presented in the output together with the *is_keypoint* flag asserted.

From the instant in time when the pixel reached the center position in the matrix (i.e., at the 14,335.00 nanoseconds marker), eight clock cycles were required to compute the FAST algorithm using the current circle and present the final score in the output (i.e., at the 14,415.00 nanoseconds marker).

A wave plot that results from the same simulation and includes additional signals from the NMS datapath can be seen in Figure F.13. The blue and black markers denote the same events of the FAST datapath that were discussed above, and the final marker in green shows the instant in time when the 0x613 score was selected maximum and thus the pixel was considered a corner. The signals column shows the *score_matrix* contents, and it can be observed that while no other score was received, the current values were preserved for the next four clocks, but processing beyond the matrix kept on going. At the same time, processing beyond the matrix kept on going in the NMS datapath and two clock cycles later, the *is_corner* flag is asserted at the output. The outputs *y_coord* and *x_coord*, provide the corner location in the image that corresponds with the coordinates deduced from the input file contents i.e., Y:0x0007 and X:0x000F.

From the instant in time that such score value was presented at the output of the FAST datapath, thirty two clock cycles were required until it reached the output of the NMS suppression. Although only sixteen memory positions (Line L1) exist between these two places, other scores are required to build the adjacent values in the 3x3 matrix, and so it was delayed in the local storage waiting for input pixels and correspondent scores produced by the previous IP.

For completeness, Figure F.14 shows the log of the software application running on the host system, and the console log of the Vivado simulator. In Figure F.14a it can be seen the command issued to launch the application in the Linux terminal console. It receives the character 'c' as first argument, to select the application with co-simulation qualifiers, and also receives the string 'double.pgm' as argument, to specify the name of the input file. The next five messages notify about the progress in the handshake performed between the tools involved, and the next two messages indicate the processing results of each Task class. The MFastSA0 message indicates that a 256-byte block of data was transferred. In its turn, the message from MNonmaxSA0 indicates that a corner was found.

In Figure F.14 it can be see then parameters of each HW-Task in the generic list. These include the accelerator tag, and the IP and port used to handshake with the software application. Following these, some messages are emitted that relate to the accelerator setup, where it receives some parameters such as memory addresses in the system. The last set of messages indicate the instants in time when each accelerator launched interrupts to the host system, and the simulation concludes with the exit of each HW-Task that closes the connectivity.

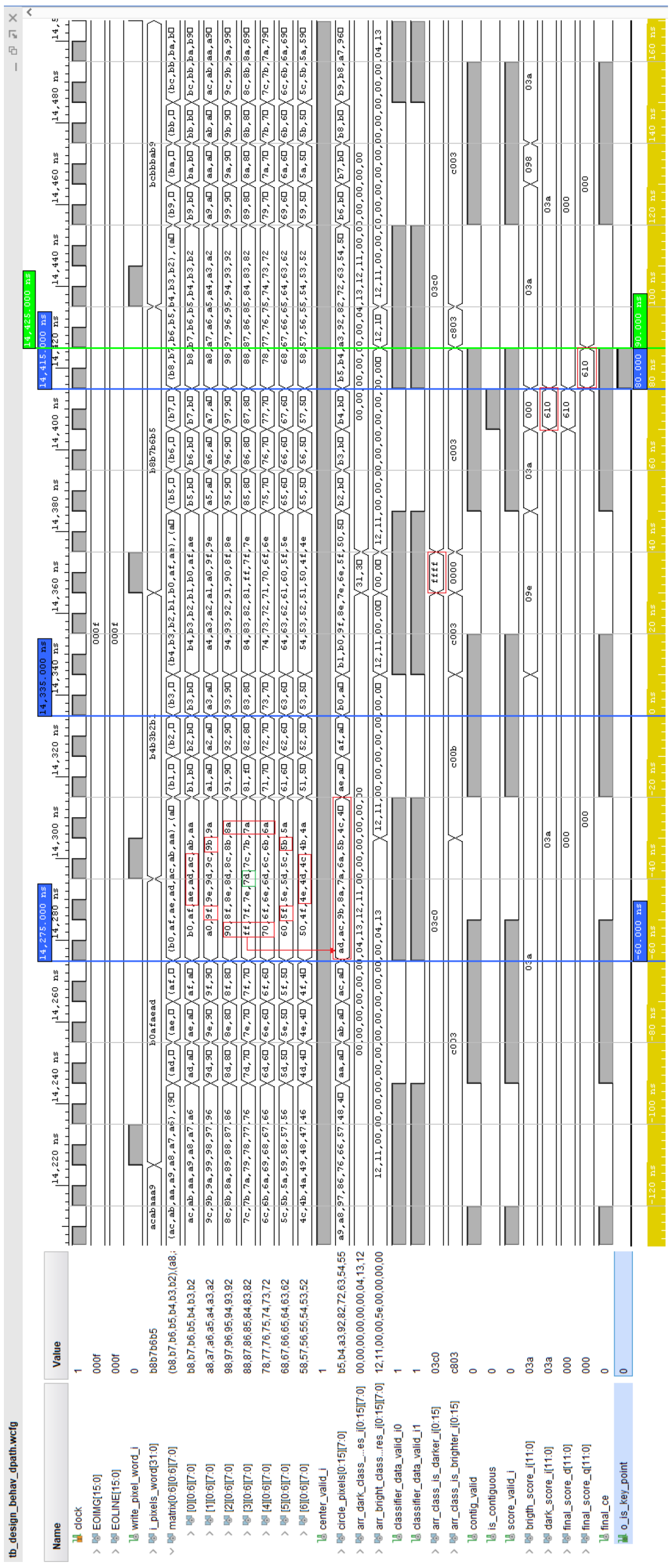
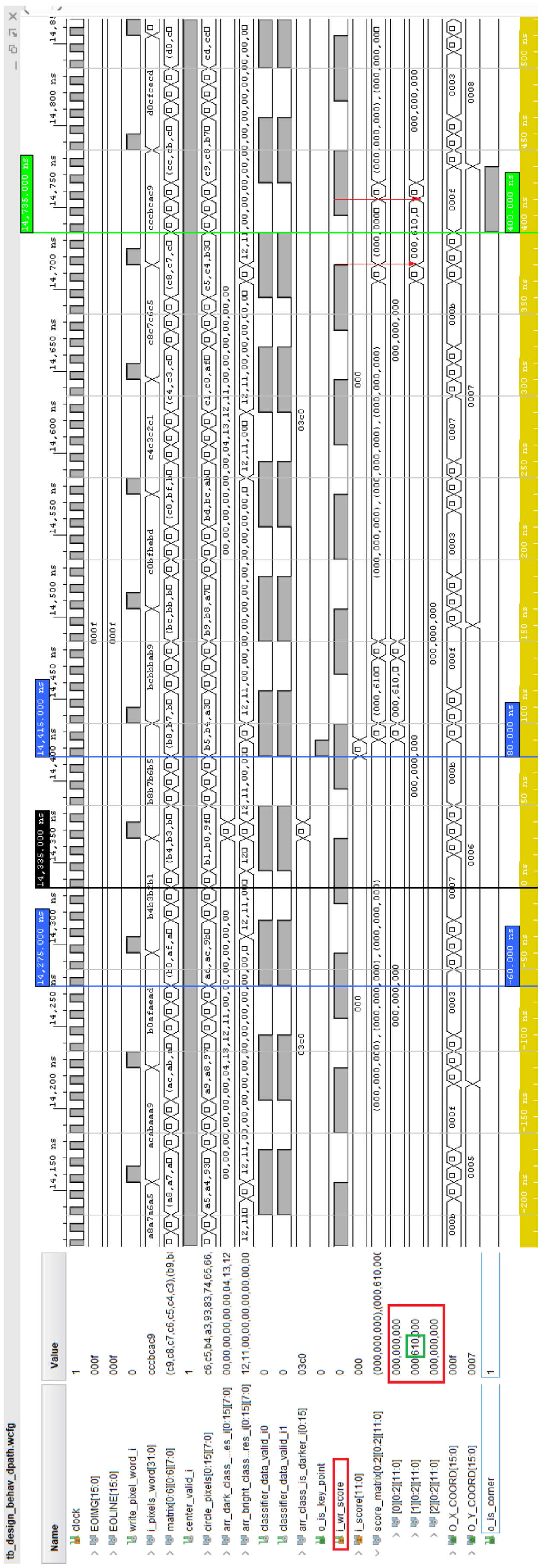


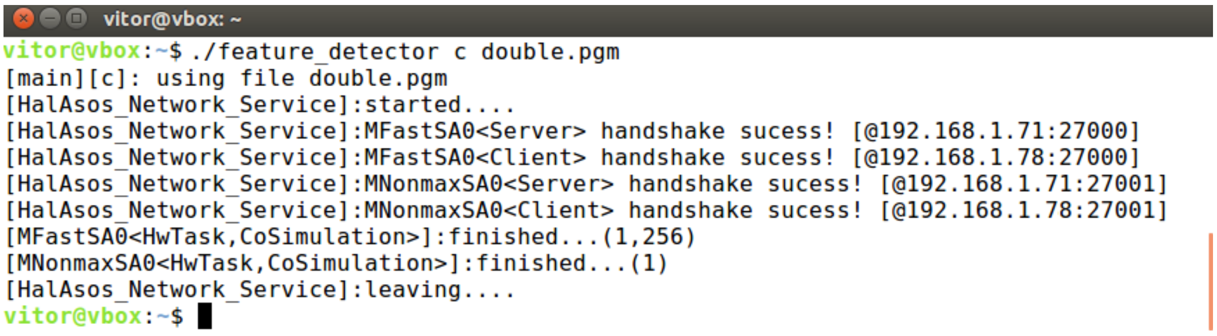
Figure F.12: Feature Detection co-simulation - Wave plot of the FAST datapath.



tb_design_behav_dpath.wcfg

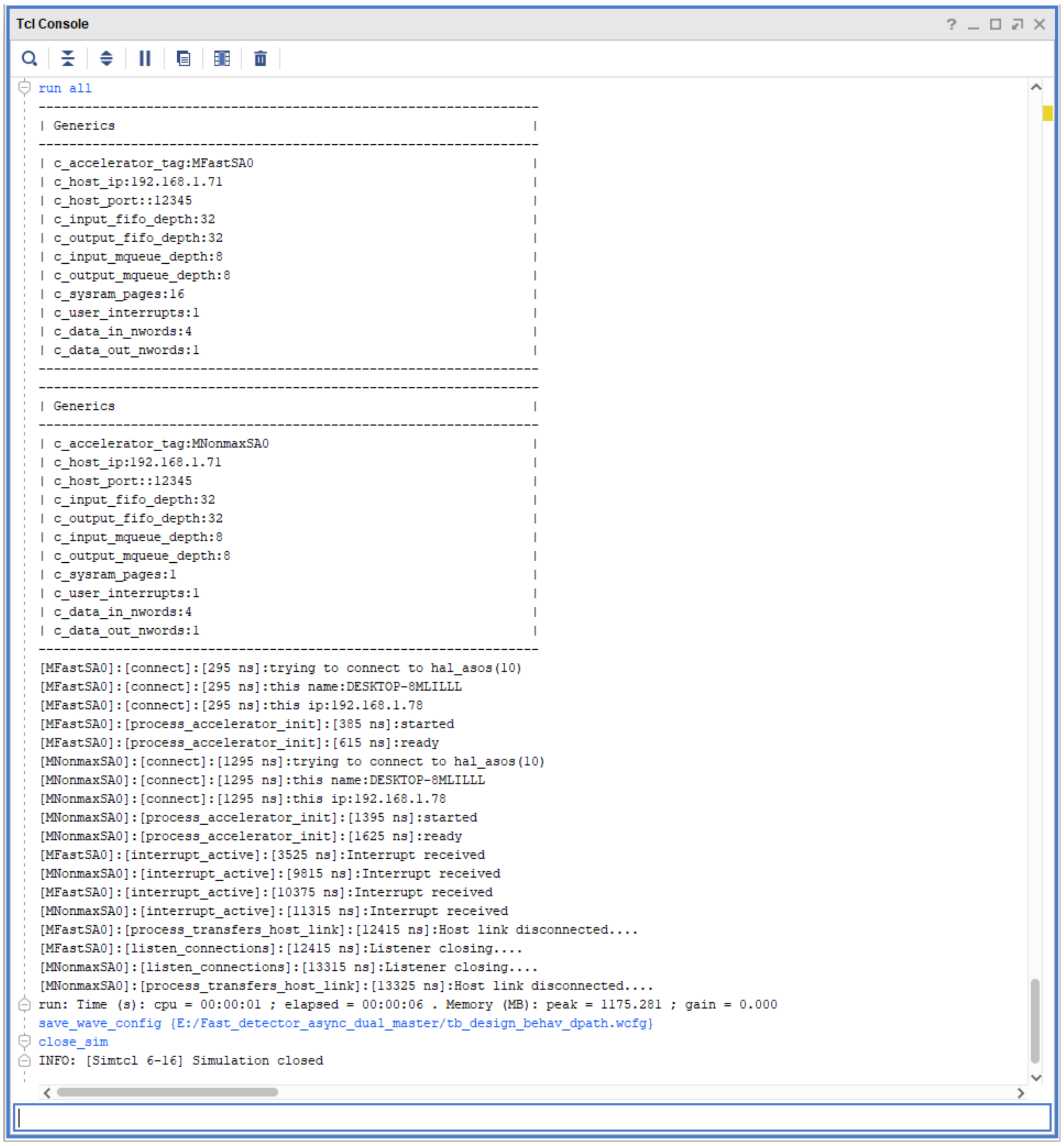
Name	Value
clock	1
EOIMG[15:0]	000f
EOLINE[15:0]	000f
write_pixel_word_i	0
i_pixels_word[31:0]	cccbcac9
matrix[0:6][0:6][7:0]	(c9,c8,c7,c6,c5,c4,c3),(b9,b
center_valid_i	1
circle_pixels[0:15][7:0]	c6,c5,b4,a3,93,83,74,65,66,
arr_dark_class_...es_i[0:15][7:0]	00,00,00,00,00,00,00,04,13,12
arr_bright_class...res_i[0:15][7:0]	12,11,00,00,00,00,00,00,00,00
classifier_data_valid_i0	0
classifier_data_valid_i1	0
arr_class_is_darker_i[0:15]	03c0
o_is_key_point	0
i_wr_score	0
i_score[11:0]	000
score_matrix[0:2][0:2][11:0]	(000,000,000),(000,610,000
[0][0:2][11:0]	000,000,000
[1][0:2][11:0]	000,610,000
[2][0:2][11:0]	000,000,000
O_X_COORD[15:0]	000f
O_Y_COORD[15:0]	0007
o_is_corner	1

Figure F.13: Feature Detection co-simulation - Wave plot of the FAST+NMS datapaths.



```
vitor@vbox:~$ ./feature_detector c double.pgm
[main][c]: using file double.pgm
[HalAsos_Network_Service]:started....
[HalAsos_Network_Service]:MFastSA0<Server> handshake sucess! [@192.168.1.71:27000]
[HalAsos_Network_Service]:MFastSA0<Client> handshake sucess! [@192.168.1.78:27000]
[HalAsos_Network_Service]:MNonmaxSA0<Server> handshake sucess! [@192.168.1.71:27001]
[HalAsos_Network_Service]:MNonmaxSA0<Client> handshake sucess! [@192.168.1.78:27001]
[MFastSA0<HwTask,CoSimulation>]:finished...(1,256)
[MNonmaxSA0<HwTask,CoSimulation>]:finished...(1)
[HalAsos_Network_Service]:leaving....
vitor@vbox:~$
```

(a) Host application output.



```
run all
----------------------------------------------------------------
| Generics                                                     |
----------------------------------------------------------------
| c_accelerator_tag:MFastSA0                                   |
| c_host_ip:192.168.1.71                                       |
| c_host_port::12345                                           |
| c_input_fifo_depth:32                                        |
| c_output_fifo_depth:32                                       |
| c_input_mqueue_depth:8                                       |
| c_output_mqueue_depth:8                                      |
| c_sysram_pages:16                                            |
| c_user_interrupts:1                                          |
| c_data_in_nwords:4                                           |
| c_data_out_nwords:1                                          |
----------------------------------------------------------------
----------------------------------------------------------------
| Generics                                                     |
----------------------------------------------------------------
| c_accelerator_tag:MNonmaxSA0                                 |
| c_host_ip:192.168.1.71                                       |
| c_host_port::12345                                           |
| c_input_fifo_depth:32                                        |
| c_output_fifo_depth:32                                       |
| c_input_mqueue_depth:8                                       |
| c_output_mqueue_depth:8                                      |
| c_sysram_pages:1                                             |
| c_user_interrupts:1                                          |
| c_data_in_nwords:4                                           |
| c_data_out_nwords:1                                          |
----------------------------------------------------------------
[MFastSA0]:[connect]:[295 ns]:trying to connect to hal_asos(10)
[MFastSA0]:[connect]:[295 ns]:this name:DESKTOP-8MLILLL
[MFastSA0]:[connect]:[295 ns]:this ip:192.168.1.78
[MFastSA0]:[process_accelerator_init]:[385 ns]:started
[MFastSA0]:[process_accelerator_init]:[615 ns]:ready
[MNonmaxSA0]:[connect]:[1295 ns]:trying to connect to hal_asos(10)
[MNonmaxSA0]:[connect]:[1295 ns]:this name:DESKTOP-8MLILLL
[MNonmaxSA0]:[connect]:[1295 ns]:this ip:192.168.1.78
[MNonmaxSA0]:[process_accelerator_init]:[1395 ns]:started
[MNonmaxSA0]:[process_accelerator_init]:[1625 ns]:ready
[MFastSA0]:[interrupt_active]:[3525 ns]:Interrupt received
[MNonmaxSA0]:[interrupt_active]:[9815 ns]:Interrupt received
[MFastSA0]:[interrupt_active]:[10375 ns]:Interrupt received
[MNonmaxSA0]:[interrupt_active]:[11315 ns]:Interrupt received
[MFastSA0]:[process_transfers_host_link]:[12415 ns]:Host link disconnected....
[MFastSA0]:[listen_connections]:[12415 ns]:Listener closing....
[MNonmaxSA0]:[listen_connections]:[13315 ns]:Listener closing....
[MNonmaxSA0]:[process_transfers_host_link]:[13325 ns]:Host link disconnected....
run: Time (s): cpu = 00:00:01 ; elapsed = 00:00:06 . Memory (MB): peak = 1175.281 ; gain = 0.000
save_wave_config {E:/Fast_detector_async_dual_master/tb_design_behav_dpath.wcfg}
close_sim
INFO: [Simtcl 6-16] Simulation closed
```

(b) Vivado simulator output log.

Figure F.14: Feature Detection co-simulation output logs.

References

[1] G. E. Moore, "Progress in digital integrated electronics [technical literature, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.]," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 36–37, Sep. 1975.

[2] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[3] IEEE. (2015) IEEE/IEC 62014-4-2015 - IEEE/IEC International Standard - IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. [Online]. Available: https://standards.ieee.org/standard/62014-4-2015.html

[4] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming models for hybrid FPGA-cpu computational components: a missing link," *IEEE Micro*, vol. 24, no. 4, pp. 42–53, 2004.

[5] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø, "Using dynamic partial reconfiguration of FPGAs in real-time systems," *Microprocessors and Microsystems*, vol. 61, pp. 198–206, 2018.

[6] H.-G. Vu, T. Nakada, and Y. Nakashima, "Efficient hardware task migration for heterogeneous FPGA computing using HDL-based checkpointing," *Integration*, vol. 77, pp. 180–192, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167926020302984

[7] H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embedded Comput. Syst.*, vol. 7, 02 2008.

[8] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 170–177.

[9] L. Gantel, A. Duc, L. Steiner, F. Vannel, A. Upegui, and F. Gluck, “A FPGA-Based Post-Processing and Validation Platform for Random Number Generators,” in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 123–126.

[10] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, “SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, pp. 2179–2192, 12 2013.

[11] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, “ReconOS: An Operating System Approach for Reconfigurable Computing,” *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[12] Z. Zhu, J. Zhang, J. Zhao, J. Cao, D. Zhao, G. Jia, and Q. Meng, “A Hardware and Software Task-Scheduling Framework Based on CPU+FPGA Heterogeneous Architecture in Edge Computing,” *IEEE Access*, vol. 7, pp. 148 975–148 988, 2019.

[13] A. Vaishnav, K. Pham, J. Powell, and D. Koch, “FOS: A Modular FPGA Operating System for Dynamic Workloads,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, pp. 20:1–20:28, 2020.

[14] D. Korolija, T. Roscoe, and G. Alonso, “Do OS abstractions make sense on FPGAs?” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 991–1010. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/roscoe

[15] P. Yuan, Y. Guo, L. Zhang, X. Chen, and H. Mei, “Building application-specific operating systems: a profile-guided approach,” *Science China Information Sciences*, vol. 61, pp. 1–17, 2017.

[16] B. Kollenda, P. Koppe, M. Fyrbiak, C. Kison, C. Paar, and T. Holz, “An Exploratory Analysis of Microcode as a Building Block for System Defenses,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1649–1666. [Online]. Available: https://doi.org/10.1145/3243734.3243861

[17] R. Sharifi and A. Venkat, *CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities*. IEEE Press, 2020, p. 762–775. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00068

[18] K. Nam, B. Fort, and S. Brown, "FISH: Linux system calls for FPGA accelerators," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[19] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–8.

[20] C. Vatsolakis and D. Pnevmatikatos, "RACOS: Transparent access and virtualization of reconfigurable hardware accelerators," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2017, pp. 11–19.

[21] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez, "R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs," *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1542–1556, 2013.

[22] S. E. Ong, S. C. Lee, N. B. Z. Ali, and F. A. B. Hussin, "SEOS: Hardware Implementation of Real-Time Operating System for Adaptability," in *2013 First International Symposium on Computing and Networking*, 2013, pp. 612–616.

[23] Q. community. (2017) QEMU a generic and open source machine emulator and virtualizer. [Online]. Available: https://www.qemu.org/

[24] B. team. (2021) Buildroot Making Embedded Linux Easy. [Online]. Available: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

[25] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, Roback, and J. F. D. Jr, "Advanced Encryption Standard (AES)," Nov 2001. [Online]. Available: https://www.nist.gov/publications/advanced-encryption-standard-aes

[26] OProfile, "OProfile - A System Profiler for Linux." Jul 2017. [Online]. Available: http://oprofile.sourceforge.net/news/

[27] GNU, "GNU gprof," Jul 2015. [Online]. Available: https://sourceware.org/binutils/docs/gprof/

[28] O. team. (2021) OpenEmbedded. [Online]. Available: http://www.openembedded.org/wiki/Main_Page

[29] ——. (2021) O]pen Source Computer Vision Library. [Online]. Available: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

[30] ——. (2021) OpenCV feature matching. [Online]. Available: https://opencv.org/

[31] O. Team. (2021) FAST algorithm for corner detection. [Online]. Available: https://docs.opencv.org/4.x/df/d0c/tutorial_py_fast.html

[32] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 2, 2005, pp. 1508–1515 Vol. 2.

[33] STMicroeletronics. (2017) Digital camera interface (DCMI) on STM32 MCUs. [Online]. Available: https://www.qemu.org/

[34] M. Kraft, A. Schmidt, and A. Kasi□ski, "High-Speed Image Feature Detection Using FPGA Implementation of Fast Algorithm." in *Third International Conference on Computer Vision Theory and Applications.*, vol. 1, 01 2008, pp. 174–179.

[35] J. Huang, G. Zhou, X. Zhou, and R. Zhang, "A new FPGA architecture of FAST and BRIEF algorithm for on-board corner detection and matching," *Sensors*, vol. 18, p. 1014, 03 2018.

[36] H. Heo, J.-y. Lee, K.-y. Lee, and C.-h. Lee, "FPGA based implementation of FAST and BRIEF algorithm for object recognition," in *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*, 2013, pp. 1–4.

[37] D. Canright, "A Very Compact S-Box for AES," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 441–455.