FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Reconfigurable peripheral manager for embedded robotic systems

Filipe Miguel Monteiro Lopes

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: José Carlos dos Santos Alves

July 30, 2015

© Filipe Lopes, 2015

U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

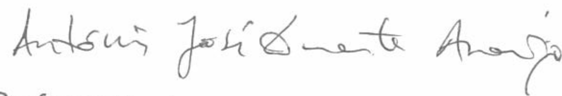
MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES **2014/2015**

A Dissertação intitulada

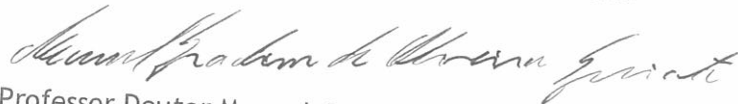
"Reconfigurable Peripheral Manager for Embedded Robotic Systems"

foi aprovada em provas realizadas em 13-07-2015

o júri



Presidente Professor Doutor António José Duarte Araújo
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto



Professor Doutor Manuel Gradim de Oliveira Gericota
Professor Adjunto do Departamento de Engenharia Eletrotécnica do Instituto Superior de Engenharia do Porto

Professor Doutor José Carlos dos Santos Alves
Professor Associado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Filipe Miguel Monteiro Lopes

Faculdade de Engenharia da Universidade do Porto

Resumo

Uma preocupação crescente em relação à poupança energética tem permitido o crescimento de sistemas computacionais de baixo consumo no mundo da robótica. Este facto combinado com o aumento da diversidade da robótica no dia a dia, torna essencial que os sistemas robóticos possuam a capacidade de serem facilmente reconfigurados.

No entanto, apesar do crescente interesse em volta do desenvolvimento de hardware por parte da comunidade tecnológica juntamente com ferramentas cada vez mais user-friendly ainda existe pouco interesse e conhecimento em relacão nesta área.

Este trabalho aproxima o utilizador comum do desenvolvimento de hardware e reconfigurabilidade usando um computador de baixo consumo, BeagleBone Black, Logi-Bone uma placa com FPGA, e uma ferramenta capaz de facilmente e sem necessidade de conhecimentos adicionais gerar um projecto de hardware completo.

A ferramenta desenvolvida e explicada em detalhe neste documento permite gerar um projeto de hardware completo para a Logi-Bone contendo os periféricos mais comuns da robótica, usando uma linguagem simples e intuitiva. E providência as APIs necessárias para uma fácil integração dos periféricos gerados na FPGA com o software presente na BeagleBone.

Finalmente, para atrair utilizadores com algum conhecimento em desenvolvimento de hardware a ferramenta permite a integração de blocos feitos à medida no projecto.

Abstract

Nowadays there is a huge concern with power consumption, opening the doors to small and low energy consumption computers in the world of robotics. With the advance of electronic technology and attempts to delegate further tasks to robots daily, a reconfigurable system is of utmost importance.

However, although the increase in interest regarding digital design and related tools user-friendlier than ever, the adoption of hardware development by the developer community is ratter low.

This work brings digital design and system reconfigurability to the common user, using the low energy consumption, powerful and small computer, BeagleBone Black, the Logi-Bone, a device that contains a FPGA, attached to it and a software tool used to easily generate the full hardware project for the FPGA, without needing any particular hardware design knowledge.

This tool, developed and explored in detail through the length of the document, is capable of generating full hardware design containing most of the common peripherals used in robotics for the Logi-Bone via simple and intuitive language. Furthermore, this tool also provides the APIs necessary to easily interface with the peripherals chosen by the user and implemented in the FPGA.

Finally, in an attempt to captivate hardware designers, the software tool also allows inclusion of one or more custom blocks in a project.

Agradecimentos

Em primeiro gostaria de agradecer ao meu orientador Professor José Carlos Alves, pelo o apoio, auxilio e conselhos dados ao longo desta dissertação é de louvar o entusiasmo deste professor pelo o seu trabalho que me permitiu continuar a trabalhar entusiasticamente mesmo nas fases menos boas da dissertação. Muitos outros professores também merecem o meu agradecimento pelo todo o apoio prestado durante o meu percurso académico.

Fica aqui um obrigado à minha família por me proporcionar tudo o que necessitei para concluir esta jornada. Ao meu gato, a princesa da minha vida, o meu Bart, por pareceres saber sempre como me sinto e estares à minha beira naqueles dias mais complicados.

Quero deixar um agradecimento também às palavras de apoio dadas pela dona Zeza e à Brigite, incrível como uma rapariga tão nova tem tanta sabedoria.

Ao Sr. Eng. Tiago Souto, deixo um obrigado e um abraço pelos gelados tomados depois do trabalho e por seres aquela pessoa que está comigo desde o inicio.

Ao pessoal do costume, Orelhinhas, Armandalho, Investimento, Cebolinha, Monstro, Mãe Susana, DiMarina, Batares, SraDeArtes, Cláudia, etc um obrigado por os almoços, jogos de cartas, piadas, companhia, etc.

E finalmente, o meu maior agradecimento vai para ti, por teres sido a melhor coisa que me aconteceu, por saber que me apoias apesar de tudo, por me teres levantado do chão, mostrado o mundo e feito sorrir, por seres um orgulho e admiração para mim, por tudo e por nada, pelas palavras e pelos silêncios. Obrigado por tudo, Rainha.

Filipe

"Multiply it by infinity, and take it to the depth of forever,
and you will still have barely a glimpse of what
I'm talking about."

William Parrish (Meet Joe Black)

Contents

1 Introduction **1**
1.1 Problem definition 2
1.2 Thesis organization 3

2 State of art **5**
2.1 CPU-FPGA embedded platform 5
2.1.1 CPU and FPGA in a SoC 5
2.1.2 FPGA and Softcore CPU 13
2.1.3 Discrete CPU and discrete FPGA 15
2.2 Development tools 18
2.2.1 Xilinx Software Development Kit (SDK) and Vivado Design Suite 18
2.2.2 Qsys Altera's System Integration Tool 19
2.2.3 Valentf(x) Skeleton 19
2.3 Conclusion 20

3 BeagleBone Black, Logi-Bone and Tools **21**
3.1 BeagleBone Black and Logi-Bone 21
3.2 Valentf(x) Tools 23
3.2.1 Skeleton 24
3.3 Xilinx ISE 25
3.4 Communication between BeagleBone Black and Logi-Bone 26
3.5 Conclusion 28

4 Hardware Project **29**
4.1 BeagleBone and FPGA Synchronization 29
4.2 Wishbone 30
4.3 Registers and Memories 32
4.3.1 Registers 33
4.3.2 Memory 37
4.4 Generation of block, parameters and techniques 39
4.5 I/O Ports 41
4.6 Peripherals 42
4.6.1 Digital I/O 42
4.6.2 PWM Unidirectional 43
4.6.3 PWM H-Bridge 43
4.6.4 UART 44
4.6.5 Servo Controller 44
4.6.6 SPI 45

4.6.7 Custom blocks 45
4.7 Conclusion 48

5 Project Configuration and implementation **49**
5.1 Language 49
5.2 Tool 51
5.3 APIs 54
5.4 Work Flow 55
5.4.1 Complete project 55
5.4.2 Complete Project with custom blocks 57
5.5 Conclusion 59

6 Conclusion **61**
6.1 Future work 61

List of Figures

2.1 Virtex-II Pro Generic Architecture Overview 6
2.2 CoreConnect Block Diagram 7
2.3 Xilinx Zynq-7000 diagram 9
2.4 Altera Cyclone V SoC block diagram 10
2.5 Altera Arria V SoC block diagram 11
2.6 SmartFusion block diagram 12
2.7 SmartFusion 2 block diagram 12
2.8 LatticeMico32 block diagram 14
2.9 Leon 3 block diagram 15
2.10 Armadeus APF51 memory bus 16
2.11 Logi Bone board 17
2.12 Logi Pi board 17
2.13 Logi Stack Overview 18
2.14 Skeleton editor 19

3.1 BeagleBone Black board 22
3.2 Logi Bone board 23
3.3 Skeleton interface with a simple project configuration 24
3.4 Blocks and parameters generated 25
3.5 Xilinx ISE program 26
3.6 Read operation in GPMC bus 27
3.7 Write operation in GPMC bus 27
3.8 Layers of communication between the BeagleBone Black and the FPGA 28

4.1 Two flip-flop synchronizer diagram 29
4.2 GPMC and Whisbone read signals 31
4.3 GPMC and Whisbone write signals 31
4.4 GPMC to Whisbone block diagram 32
4.5 Map of the 16-bit GPMC address space 33
4.6 Block W_RO illustrating the method used to have a parameterizable number of registers 37
4.7 Block RW_RI with a single register 38
4.8 Distribution of the memory address space 39
4.9 H-Bridge 43

5.1 Error port unknown 51
5.2 Error port already in use 52
5.3 Error frequency unknown 52
5.4 Peripherals and parameters generated 53

5.5 Include file, used to interact with the peripherals 54
5.6 Block.txt file with the peripherals written 55
5.7 ./start script running in terminal 56
5.8 End result of the script 56
5.9 Files in the BeagleBone Black 56
5.10 Application for running in the BeagleBone Black 57
5.11 Blocks.txt file, with custom block 57
5.12 ./start_custom script running in terminal 58
5.13 Adding the template to the logibot_top.v file 58
5.14 Generating the programming file 59

List of Tables

2.1 Characteristics of high end Virtex boards with PowerPC 7
2.2 Characteristics of low and high end SoC's of Zynq family 8
2.3 Features for the AXI4 9
2.4 Principal characteristics of Altera Cyclone V and Arriva V high-end SoCs 10
2.5 Principal characteristics of Microsemi SmartFusion and SmartFusion2 high-end SoCs 12
2.6 Principal characteristics of Armadeus boards 15
2.7 Principal characteristics of Xilinx Spartan 6 LX9 16
2.8 Principal characteristics of Beaglebone Black and Raspberry Pi 17

4.1 Distribution and size of memories depending on his quantity 38
4.2 Parameters and compiler directives for registers and memories 40
4.3 Digital I/O FPGA occupation 42
4.4 PWM Unidirectional FPGA occupation 43
4.5 PWM H-Bridge FPGA occupation 44
4.6 UART FPGA occupation 44
4.7 Servo FPGA occupation 45
4.8 SPI FPGA occupation 45
4.9 Parameters for custom_r blocks 46
4.10 Parameters for custom_m blocks 47

5.1 Language and parameters for generate peripheralas c 49

Abbreviations

AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
APU	Auxiliary Processor Unit
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
AXI	Advanced Extensible Interface
BBB	BeagleBone Black
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
GPIO	General Purpose Input/Output
GPMC	General Purpose Memory Controller
HDL	Hardware Description Language
I/O	Input/Output
I2C	Inter-Integrated Circuit
IP	Intellectual Property
LUT	Look Up Table
RISC	Reduced Instruction Set Computer
SDRAM	Synchronous dynamic random access memory
SoC	System on Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter

Chapter 1

Introduction

The high rate of advances in the embedded computing, electronics, communications and battery technologies has been a great motivation for the growth of small, affordable but powerful and sophisticated robotic systems. This has been noticed mostly in the field of low cost remote controlled autonomous flying drones increasingly used in several fields like photography, filmmaking, deliveries, surveillance and pure entertainment. The needs for highly integrated sensors for providing the vehicle position (GPS, IMU's and magnetometers), the integration of this with the system computer and control and, at the same time, maintaining low energy consumption is of utmost importance.

Other domain were autonomous robotics systems are gaining importance is in the access to marine environment, both underwater and at the surface. This increase of importance results from the need to exploit a diversity of valuable ocean resources and monitor the state of the planet, promoting the development of several projects concerning autonomous and remotely operated marine vehicles and its equipment like sensors, mechanical manipulators and sampling devices.

The marine environment presents several difficulties to developers. For instance, the radio waves suffer a drastic attenuation in water and in practice the communication underwater through radio frequency signals is only use for very short ranges. The best way to exchange information underwater is using sound waves. However the slow speed of propagation and the complex mechanisms associated to the underwater acoustic channel presents a series of difficulties for the development of autonomous robotics systems for underwater environment.

Besides, the problems previously mentioned to the communication and navigation in the underwater environment, the tasks that can be accomplished in marine applications have fostered the development and invention of a diversity of underwater and surface marine vehicles.

Usually marine robots have a conventional CPU running software on some kind of operating system that performs the control, navigation tasks and communication. To deal with the electronic part like sensors, communication or actuators, the computing system has to implement appropriate interfaces. These interfaces depend on the actual vehicle configuration and mission to accomplish, and in order to provide adaptability the electronics and the computing system should be capable of providing flexibility and easy reconfiguration capability.

Although todays embedded computers and microcontrollers already include the most common interfaces (e.g. I2C, SPI, UART, PWM) this mixed interface is fixed and cannot be easily expanded to fulfil the system needs. As consequence of this, for adding new interfaces there is a need usually requires attaching physical boards, that naturally impact on time of development and debug, system size and power consumption.

Attaching a FPGA (Field-Programmable Gate Array)[1] to a conventional CPU is an effective solution to overcome the problems referred above. In fact, all of the electronics related to the interfacing with the outside world can be moved into the FPGA, taking advantage of the reconfigurability of these devices to implement customized interface mechanisms, including specialized pre-processors for data acquired from external devices to alleviate the load of the main processor (e.g. digital filters, FFT calculators, etc.) and also implement custom computing blocks to accelerate critical parts of the software applications running in the main processor.

This solution has been adopted in various products including FPGA-based devices and single-board embedded computers as presented in chapter 2.

1.1 Problem definition

Although FPGA devices have many advantages for the type of work described above most of the developers don't use it because this approach requires digital design skills and detailed knowledge of FPGA technology. In fact, system developers are more proficient on software programming than in digital hardware design, in spite of the more and more user friendly and easy to use FPGA design tools.

The objective of this work is to develop an easy to use reconfigurable system, based on an embedded computer and a FPGA to implement highly customized interfaces with external devices, including dedicated pre-processors to alleviate the load of the main processor (e.g. digital filters or FFT calculators) and also specialized computing modules to accelerate critical parts of the software applications running on main processor.

From the FPGA point of view, this work will develop a software tool to automatically generate a complete HDL design for the FPGA device, based on a high-level description of the set of interfacing blocks and protocols configured for the project, from a library of commonly used interfaces in robotic systems (e.g. UART, PWM, SPI, etc). The objective is to hide as most as possible from the system developer all the details of the digital system that implements the external hardware interface. The tool will also allow the integration of other custom designed blocks developed by the user and support the inclusion of new modules in the FPGA design framework.

In addition, the tool will also create the software application programming interfaces (API) to facilitate the software development and the communication with the interfacing resources implemented in the FPGA.

The hardware system to be used is a commercial popular low cost embedded computer (the BeagleBone Black) and an FPGA-based daughter board (Logi-Bone) built on the XILINX Spartan6 FPGA.

1.2 Thesis organization

This report is divided in 6 chapters. Chapter 1 address the thesis problem, context and motivation. Chapter 2 presents the state of art divided in 2 parts, first the CPU and FPGA interaction products and second the tools provided to make a project in those platforms. Chapter 3 describes the hardware used, as well as the tools used in this project. Chapter 4 and 5 talk about this thesis work, the first the hardware development part and the second the software developed. Finally, chapter 6 presents the conclusions and future work for this project.

Chapter 2

State of art

This chapter presents technologies and tools for hardware design. The chapter is divided in two main parts. Section 2.1 presents the products that involve a FPGA and a CPU organized in three different categories: CPU and FPGA in a SoC, FPGA integrating one or more softcore CPU and boards with a discrete CPU and FPGA chips. Section 2.2 presents the development tools used to implement and to work with some of the products mentioned in the first section.

2.1 CPU-FPGA embedded platform

As mentioned in chapter 1 this works aims to interface a FPGA and a CPU or embedded processor. This section presents some technologies that fit in this category. The section is divided in three parts:

- CPU and FPGA in a SoC
- FPGA and Softcore CPU
- Discrete CPU and discrete FPGA

2.1.1 CPU and FPGA in a SoC

Semiconductor devices with hardened CPU subsystem integrated in FPGA offer a whole world of options for system designs. This type of devices came to help and optimize the integration of hard CPUs and FPGAs, making a single SoC with powerful performance and low power consumption.

Some of the best known SoCs with hard CPU and FPGA are the Xilinx Virtex II, 4 and 5 resulting in a partnership between Xilinx and IBM, more recent Xilinx change their CPUs to ARM processors in the Zynq family. Altera also provide their own SoCs, the Cyclone V and Arria V integrating an ARM just like the Zynq, Microsemi also enter the race for SoCs with hard CPU and FPGA launching the Smartfusion series.

2.1.1.1 Xilinx Virtex with PowerPC

Introduced in 2002[2], the Xilinx Virtex-II Pro was the first FPGA SoC developed by Xilinx to include a physical processor core, in this case an IBM PowerPC 405[3, 4, 5] running at 300MHz. All 700 I/O ports of the processor can be accessed by the FPGA[2]. This is done by immersing the processor in the first four layers of metal and having 5 layers of routing above, allowing to access the entire CPU from the FPGA fabric.

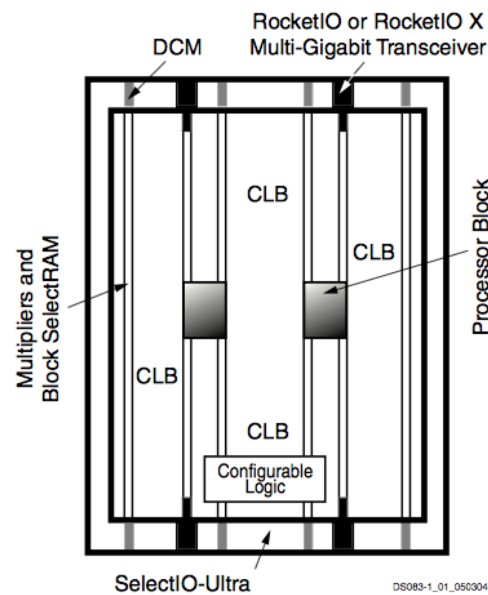


Figure 2.1: Virtex-II Pro Generic Architecture Overview[6]

Figure 2.1 presents an overview of the Virtex-II Pro architecture. As seen in the figure, the CPU is embedded in the FPGA fabric (CLB), and the connection to the FPGA is done through an IBM Core-Connect on-chip bus capable of running at 133MHz. This bus can be seen in detail in figure 2.2.

The CoreConnect bus is composed by:

1. Processor Local Bus (PLB)

2. On-Chip Peripheral Bus (OPB)

3. Device Control Register (DCR)

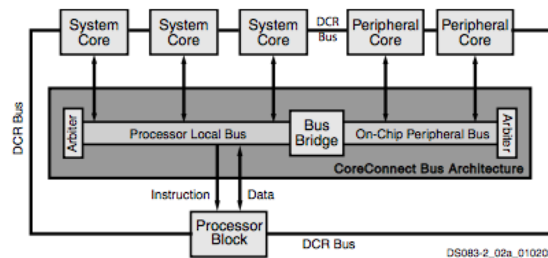


Figure 2.2: CoreConnect Block Diagram[6]

The high-performance peripherals are connected to the high-bandwidth and low-latency PLB while slower peripheral connect to OPB to reduce the traffic on the PLB, and improve the overall performance[6].

Later, in 2005, Xilinx began to ship its new hybrid CPU and FPGA SoC, the Xilinx Virtex-4 FX[7] with a new PowerPC 405 running up to 450MHz. In this case the interface between the CPU and FPGA is through APU (Auxiliary Processor Unit) capable of: [8]

- Running at different clock rates
- Supporting autonomous instructions: no pipeline stalls
- 32-bit instruction and 64-bit data
- 4-cycle cache line transfer

Virtex-5 FXT launched in 2008[9] with a new PowerPC 440 at 550MHz[10], was the last FPGA with a PowerPC CPU launched by Xilinx. The communication between the FPGA and CPU like in the Virtex-4 FX is done through an APU with the difference of 128-bit wide pipelined APU load/store capability[10].

Table 2.1 presents the characteristics of high-end Virtex devices mentioned above.

Table 2.1: Characteristics of high end Virtex boards with PowerPC [6, 8, 10]

Xilinx Virtex Pro	Virtex-II Pro	Virtex-4 FX	Virtex-5 FXT
PowerPC Processor Blocks	2	2	2
Maximum Frequency	300MHz	450MHz	550MHz
L1 Cache	16 KB Instruction 16 KB Data per processor		32 KB Instruction 32 KB Data per processor
Programmable Logic Cells	99,216	142,128	30,720[1]
Extensible Block RAM (KB)	7,992	9,936	16,416

[1] Virtex-5 FPGA slices are organized differently from previous generations. Each Virtex-5 FPGA slice contains four LUTs and four flip-flops (previously it was two LUTs and two flip-flops.)

2.1.1.2 Xilinx Zynq family

Xilinx Zynq[11] is the new generation of hybrid CPU and FPGA developed by Xilinx. The products in this family use an ARM[12] instead of a PowerPC, the ARM power and low consumption make it a better choice. The FPGA is equivalent to a Xilinx Artix-7[13] or a Xilinx Kintex-7[14], depending of the models, becoming an excellent example of a FPGA and CPU SoC integration. Announced in 2011[15, 16] Zynq become the first FPGA and Dual ARM CPU on a single SoC[17, 15] with the first versions launched in 2012[18] in a 28nm technology.

The Zynq family SoC's has a Dual CPU ARM A9 running from 667Mhz to 1GHz and a considerable number of fixed peripherals like, 2xUART, 2xCAN, 2xI2C, 2xSPI, 2xEthernet ports, 2xUSB ports and 2xSD card port. Unlike the CPU that only has one version running with different clock speeds, there are two versions of the FPGA, Artix-7 and Kintex-7. Table 2.2 resumes the characteristics of the low and high model of Zynq family.

Table 2.2: Characteristics of low and high end SoC's of Zynq family

Xilinx Zynq	Z-7010	Z-7100
Processor Core	Dual ARM Cortex-A9 MPCore	
Maximum Frequency	866 MHz	Up to 1 GHz
L1 Cache	32 KB Instruction, 32 KB Data per processor	
FPGA	Artix-7 FPGA	Kintex-7 FPGA
Programmable Logic Cells	28K Logic Cells	444K Logic Cells
Look-Up Tables	17,600	277,400
Flip-Flops	35,200	554,800
Extensible Block RAM	240 KB	3,020 KB
Programmable DSP Slices	80	2,020
Peak DSP Performance	100 GMACs	2,622 GMACs
Processing System User I/Os	32	54

One of the advantages of this device, besides its superior characteristics is the possibility of being utilized without the need to program the FPGA part allowing using only the CPU, unlike older families with embedded CPU[16].

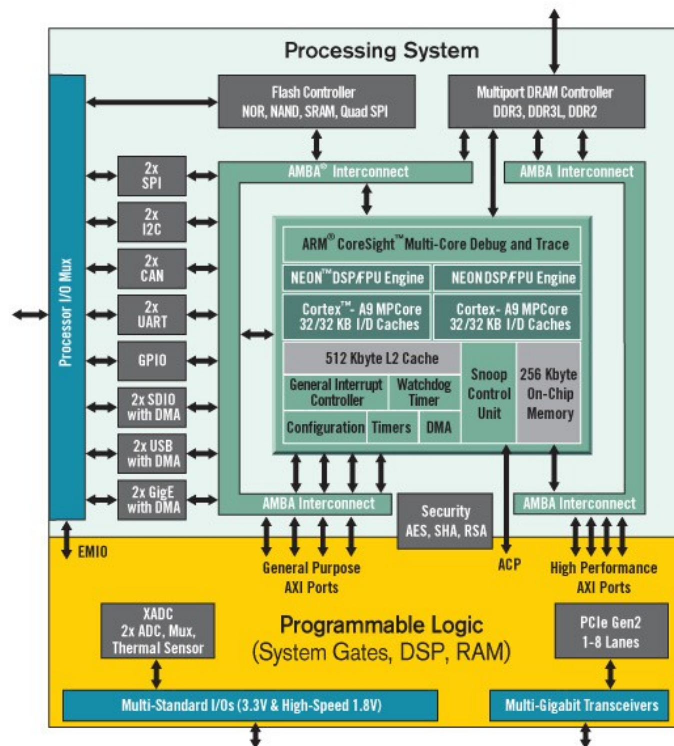


Figure 2.3: Xilinx Zynq-7000 diagram[11]

Figure 2.3 shows the main blocks of Zynq, one of the interesting parts for this work is the connection between the FPGA and CPU, done through a high performance AXI4 and a general purpose AXI4 connection[19].The AXI connection is part of AMBA protocol, a open standard for functional blocks in a SoC developed by ARM. AXI4 is divided in 3 parts, the AXI4 memory map, AXI4 stream and AXI4 lite with the features presented in table 2.3.

Table 2.3: Features for the AXI4[20]

Interface	Features
AXI4 Memory Map	Traditional Address/Data Burst (single address, multiple data)
AXI4 Streaming	Data-Only, Burst
AXI4 Lite	Traditional Address (single address, single data)

Although the AXI4 lite supports 32-bit and 64-bit data, the Xilinx IP only supports 32-bit. The AXI4 memory map and AXI4 streaming supports burst data being that the first is capable of burst up to 256 data beats[1] and the second don't have a burst limit but only works from master to slave.

[1] A 'beat' is an individual data transfer within an AXI burst[20]

2.1.1.3 Altera Cyclone V and Arria V

In the race for the hybrid CPU and FPGA SoC market, Altera[21], another FPGA manufacture, release the Cyclone V[22] and Arria V[23]. Launched in second half of 2012, using a technology of 28nm they were the Altera response to the Xilinx Zynq family. Both Cyclone V and Arria V have a Dual ARM Cortex-9 running at 925MHz and 1.05GHz respectively, 64KB of scratch RAM, 2xEthernet port, 2xUSB, 4xI2C controllers, 2xUART and 2xSPI.

Table 2.4 presents the principal characteristics of these two Altera SoCs and figure 2.4 and 2.5 shows an overview of the SoC's typology.

Table 2.4: Principal characteristics of Altera Cyclone V and Arriva V high-end SoCs [24, 25]

Altera	Cyclone V 5CSTD6	Arriva V 5ASTD5
Processor Core	Dual ARM Cortex-A9	
Maximum Frequency	915MHz	1.05GHz
L1 Cache	32 KB Instruction, 32 KB Data per processor	
Logic Elements	110K	462K
M10K memory	5,570Kb	22,820Kb
Maximum FPGA I/Os	288	504
Maximum CPU I/Os	181	208

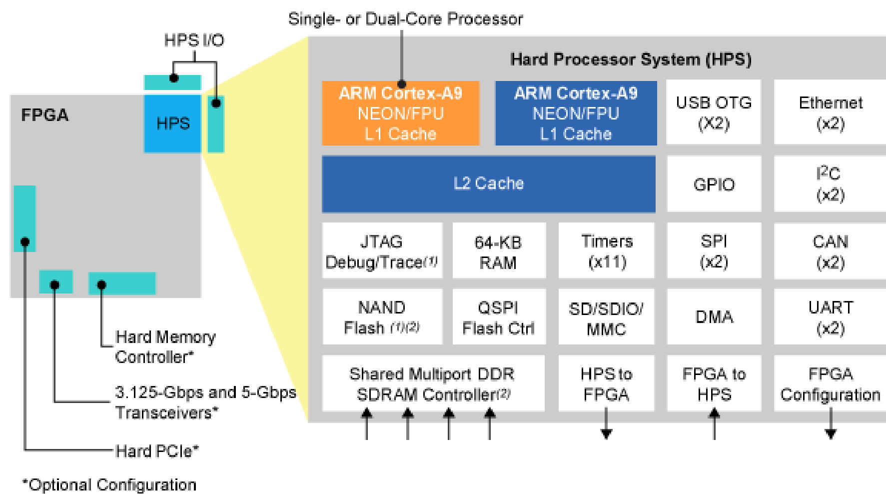


Figure 2.4: Altera Cyclone V SoC block diagram[22]

As showed in figure 2.4 and 2.5 there are 4 blocks for interfacing the CPU with the FPGA fabric, this interface like in the Xilinx Zynq family is based on the AMBA AXI, in this case the AXI-3. The main features are[23, 22]:

- 32, 64 or 128 bit interface with the FPGA (both directions)
- SDRAM controller: up to 6 masters, 4x64 bit read data ports and 4x64 bit write data ports

- 32 bit FPGA configuration manager (used to configure the FPGA)

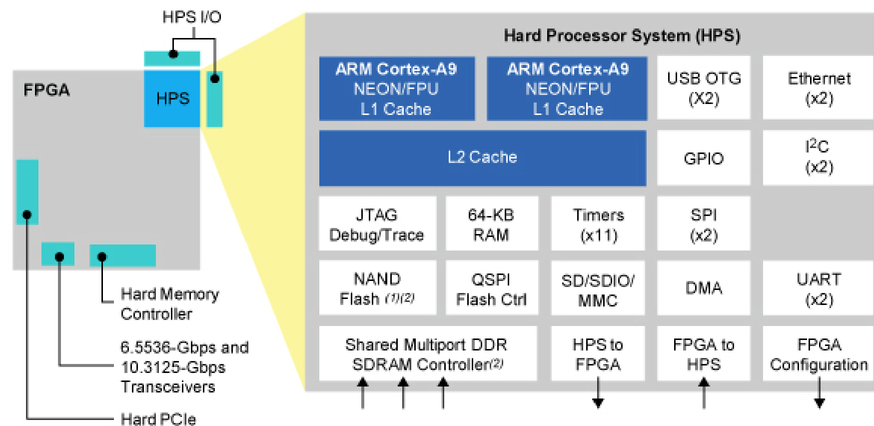


Figure 2.5: Altera Arria V SoC block diagram[23]

2.1.1.4 Microsemi SmartFusion and SmartFusion 2

Another FPGA manufacturer, the Microsemi, launched its own hybrid CPU and FPGA SoC, the SmartFusion[26] having upgraded it later to the SmartFusion 2[27]. SmartFusion was introduced in Spring of 2010[28], before the company was bought by Microsemi[29]. SmartFusion 2 was introduced in 2012[30]. This SoC was released to compete in the advanced security, high reliability and low power fields, important to the military, aviation, communications and medical industries.

Both SmartFusion and SmartFusion 2 come with an ARM Cortex-M3 CPU running at 100MHz and 133MHz respectively, 2xI2C, 2xSPI, 2xUARTs and a 10/100 Ethernet port[31, 32]. Figure 2.6 and 2.7 presents the block diagram of SmartFusion and Smartfusion 2 respectively and can be seen other physical peripherals connected to them.

Both figures 2.6 and 2.7 show the interface between the CPU and the FPGA fabric, done through an AHB bus. This is a protocol from AMBA similar to the AXI-4 and AXI-3 presented in the previous sections.

As the previous SoCs each one of these families have different models, table 2.5 presents the characteristics of the high-end SoC of each family.

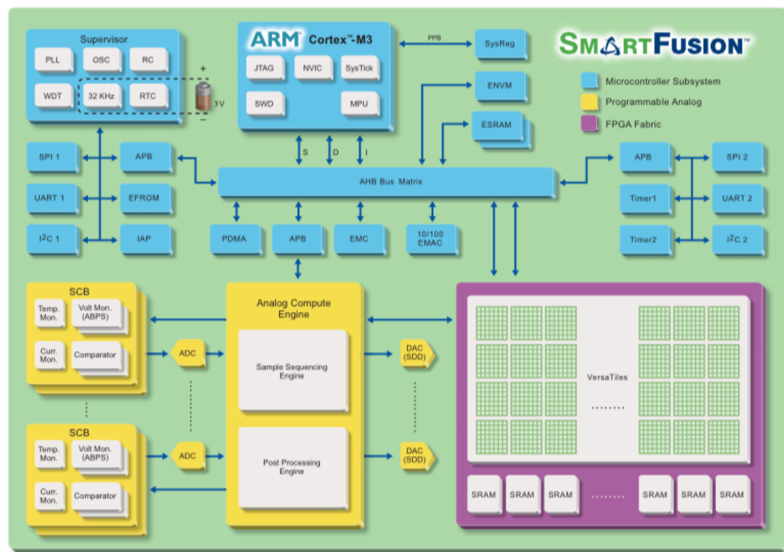


Figure 2.6: SmartFusion block diagram[31]

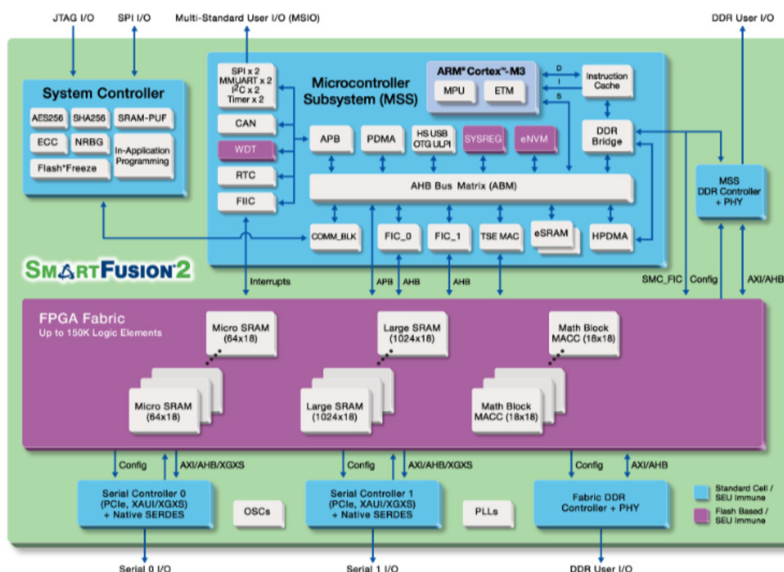


Figure 2.7: SmartFusion 2 block diagram[32]

Table 2.5: Principal characteristics of Microsemi SmartFusion and SmartFusion2 high-end SoCs [32, 31]

Microsemi	SmartFusion	SmartFusion2
Processor Core	Dual ARM Cortex-M3	
Maximum Frequency	100MHz	133MHz
L1 Cache	8Kb	
Logic Elements	5.5K	146K
RAM	110Kb	4,488Kb
Total I/Os	204	574

2.1.2 FPGA and Softcore CPU

One alternative to FPGA with physical CPU is the use of a softcore CPU. Some FPGA manufacturers develop an IP (Intellectual Property) of a microcontroller capable of interact with a design implemented in the FPGA. Softcore CPUs can be designed and adapted to the FPGA, usually they occupied less space than the physical ones and can be erase from the FPGA when not need it also the integration of CPU with the FPGA blocks is easy and in some cases can be done through registers.

This subsection presents some of the most common ones.

2.1.2.1 Xilinx Microblaze and Picoblaze

Picoblaze[33] and Microblaze[34], are softcore CPU capable of being implemented in Xilinx FPGA. These processors allow implementing a high-level programming algorithm to interact and/or control the FPGA blocks.

The Picoblaze is a fully embedded 8-bit RISC (Reduced Instruction Set Computer) microcontroller and optimized to Xilinx FPGA architectures. It's a free IP for Xilinx users, and is accompanied by an easy to use assembler code and several documentation[33]. The main features are:

- Small size (in some Xilinx boards only takes 26 slices)
- Up to 4K 18-bit instructions
- Up to 240MHz performance
- No external components needed
- Predictable fast interrupt response

The exchange of data between FPGA blocks and the Picoblaze is done through registers, so the communication speed depends on the system clock.

For more powerful CPU, Xilinx provides the Microblaze. This is a 32-bit RISC soft processor with some advantages over Picoblaze such as the capability of being optimized for an application unlike the Picoblaze, and the capability to install an OS (Operating System) like Linux[35].

The implementation of the Microblaze requires the use of one or more Xilinx tools, like the Vivado Design Edition, Vivado Webpack Edition or IDS Embedded Edition[34].

2.1.2.2 LatticeMico32

LatticeMico32[36] is a softcore CPU from Lattice under a free IP license, meaning that can be used with any FPGA, ASIC or other hardware. As his name suggests is a 32-bit RISC microprocessor, having an interface based on wishbone that allows easily adding external peripheral (SPI, I2C, UART, GPIO, etc.). Figure 2.8 presents the block diagram of a project with LatticeMico32 whose primary features are:

- 32 general purpose registers
- Up to 32 external interrupts
- Dual wishbone memory interfaces (Instruction and Data)

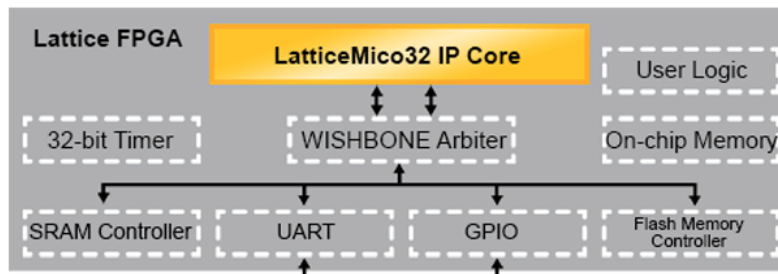


Figure 2.8: LatticeMico32 block diagram[36]

2.1.2.3 Altera Nios I and II

In the softcore CPU race, Altera competes with the Nios II[37], a second version of the first softcore CPU Nios I[38]. Nios II is a 32-bit processor very similar to others available in the market, having 32 external interrupts and accessed to up 2 GB of external address space. Like in others manufactures users have a large portfolio of peripheral IP cores to choose and one of three different cores for Nios II[39] according to their necessities:

- Nios II/f : For more speed
- Nios II/e : For more space economy
- Nios II/s : Standard

Altera Nios II softcore processor is available to ASIC designs through collaboration between Altera and Synopsys, a world class company in the field of IP development[40].

2.1.2.4 Microsemi Leon 3

The last softcore CPU to be presented is the Leon 3 from Microsemi, a 32- bit processor compliant with the SPARC (Scalable Processor ARChitecture) V8 architecture. It is highly configurable and suitable for a SoC design[41] and available under a low-cost commercial license however is code is available for research and education proposes for free. For critical systems, there is a version call LEON3-FT. The Leon 3 has an AMBA AHB 32-bit bus to interface with other hardware code. Figure 2.9 illustrates the block diagram of Leon 3.

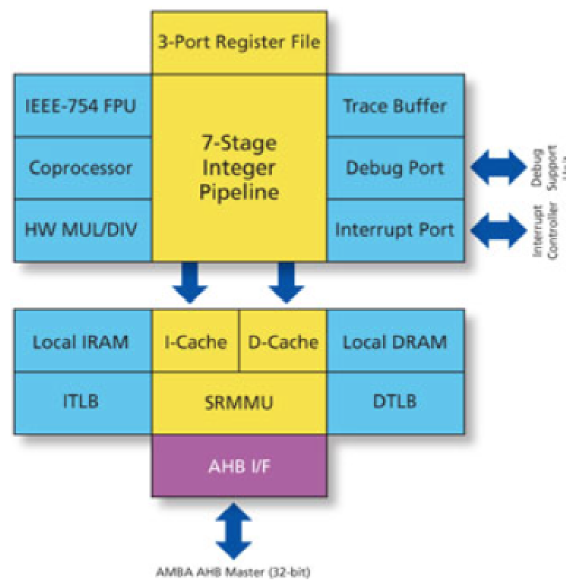


Figure 2.9: Leon 3 block diagram[41]

2.1.3 Discrete CPU and discrete FPGA

In the last part of this section is presented discrete CPU and FPGA integration and boards. This type of products has the advantage of being the best of both worlds without compromising the performance and being modular, losing in consumption, data transfer rate and integration between them. However, the adaptability from this type of interaction allows upgrading parts of the system without having to change all the hardware.

2.1.3.1 Armadeus APF51 and APF27

Armadeus is a non-profit association that produces development boards that combine a discrete CPU with a discrete FPGA, the APF27[42] and the APF51[43]. Both of have an ARM processor and a Xilinx FPGA. Table 2.6 presents a brief description of each board.

Table 2.6: Principal characteristics of Armadeus boards [43, 42]

Board	APF27	APF51
Processor	MX27 ARM926 400MHz	MX515 Cortex-A8 800MHz
Memory	64/128 or 256MB	256MB or 512MB
FPGA	Spartan 3A	Spartan 6 LX9
USB	2	2

The connection between the ARM processor and the FPGA is through a parallel 16-bit memory bus[44]. Figure 2.10 shows the memory bus for the APF51, very similar to the APF27 board.

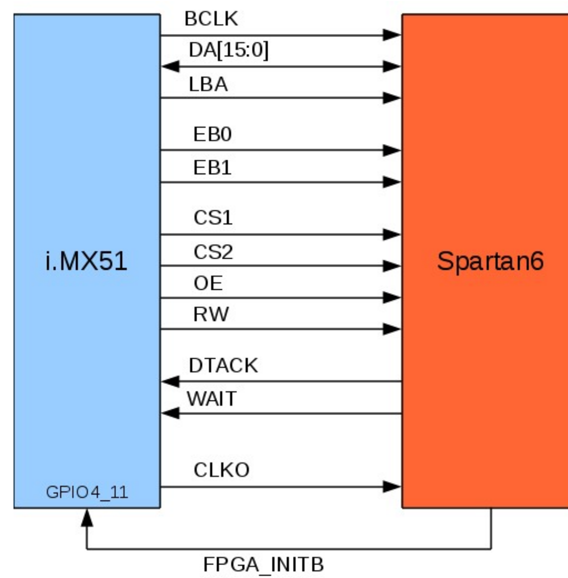


Figure 2.10: Armadeus APF51 memory bus[44]

Table 2.6 shows that the devices FPGAs are the Spartan 3A[45] and Spartan LX9[46]. Table 2.7 presents the characteristics of these FPGAs.

Table 2.7: Principal characteristics of Xilinx Spartan 6 LX9 [46, 45]

Xilinx	Spartan 3A	Spartan 6 LX9
Logic Cells	4,032	9,152
Max RAM	288Kb	576Kb
Max User I/O	248	200

2.1.3.2 ValentF(x) Logi Bone and Logi Pi

ValentF(x) launched their devices on kickstarter, a website to find investors to finance projects in exchange of discount in the products or gifts. Logi-Bone[47], figure 2.11, and Logi-Pi[48], figure 2.12, are devices containing a FPGA that attach Beaglebone Black and Raspberry Pi respectively, whose characteristics are presented in table 2.8.

The shield boards are very similar, both of them have a FPGA Spartan 6 LX9 from Xilinx, 32 FPGA GPIO, 1 sata connector and a 32MB SDRAM. Table 2.7, presented previously shows some of the characteristics of this FPGA.

Table 2.8: Principal characteristics of Beaglebone Black and Raspberry Pi [49, 50]

Board	BeagleBone Black	Raspberry Pi 2
Processor	1GHz ARM Cortex-A8	900MHz ARM Cortex-A7
Memory	512MB	1GB
User GPIOs	92	40
HDMI	Yes	Yes
USB	1	4

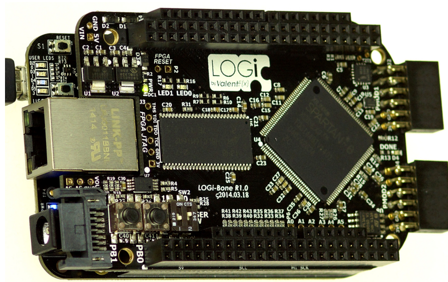


Figure 2.11: Logi Bone board[47]

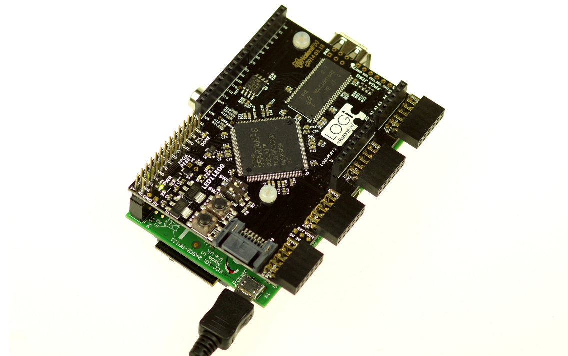


Figure 2.12: Logi Pi board[48]

ValentF(x) provides all the tools for easy interact with the FPGA from the CPU, and a tool to generate HDL code using pre made blocks and connecting them in a graphical interface.

In Logi-Pi the connection with the FPGA is through SPI bus at a speed of 32-48MHz, providing 4MB/s. The Logi-Bone uses a different connection, GPMC (General Purpose Memory Controller), this is a 16-bit data/address bus capable of 16.6MB/s. The use of this bus has the disadvantage of disabling the HDMI port of the Beaglebone Black, removing the possibility of having a graphical interface[51].

In figure 2.13 is a schematic of the communication layers between the Beaglebone Black or Raspberry PI and the FPGA.

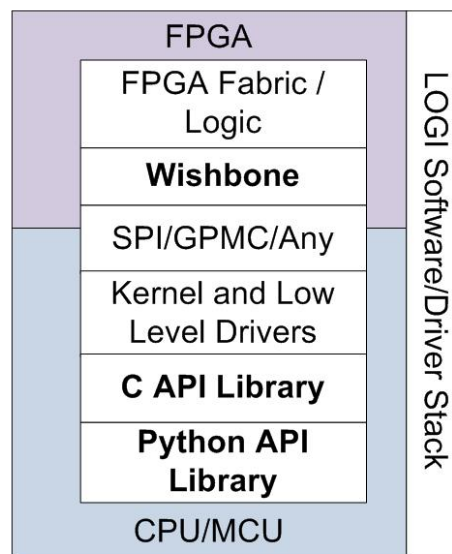


Figure 2.13: Logi Stack Overview[52]

2.2 Development tools

Most of the devices and SoCs presented above need development tools to program and configure the FPGAs, some of this tools improve the user experience and try to make digital design development more easy, aiming for users that don't have that kind of knowledge.

2.2.1 Xilinx Software Development Kit (SDK) and Vivado Design Suite

First section presents the Xilinx Zynq and Microblaze. Xilinx launched a development tool to help working with these systems, the Xilinx SDK[53]. It allows creating embedded applications for any board of Zynq family and for Microblaze, and includes:

- System Debugger
- Custom Design Aware
- Drivers and Libraries
- Software Profiling
- System Performance Analysis and Optimization

The Vivado Design Suite is used to develop the FPGA design, allowing the IP integration and implementation, verification and debug and design exploration and IP generation. Some of these features are available in the free version of the software[54].

Although these software have a high level of features allowing the user to implement a large number of IPs, configure several Microblazes and interface them with PowerPC from Virtex family

and ARM from Zynq family, working with them implies a detailed knowledge of the hardware design and hardware development languages.

2.2.2 Qsys Altera's System Integration Tool

Qsys[55] is the integration tool from Altera for saving time and effort by automatically generate and interconnect logic to other IP functions. The main features to this software are:

- Easy-to-use GUI
- Automatic generation of interconnect logic
- Support mixing of different industry-standard interfaces

Qsys can be used to generate the Nios II as well as few peripherals for it, but to integrate it in a FPGA is necessary to use the Quartus II software. Quartus II tool is similar to the Xilinx Vivado and has the same problem of needing hardware design knowledge to understand and interact with him.

2.2.3 Valentf(x) Skeleton

Valentf(x) designed a web-based software to provide an easy way to develop a project to the Logi Bone and Logi Pi. Skeleton[56] allows to generate VHDL code through a block programming mode like showed in figure 2.14.

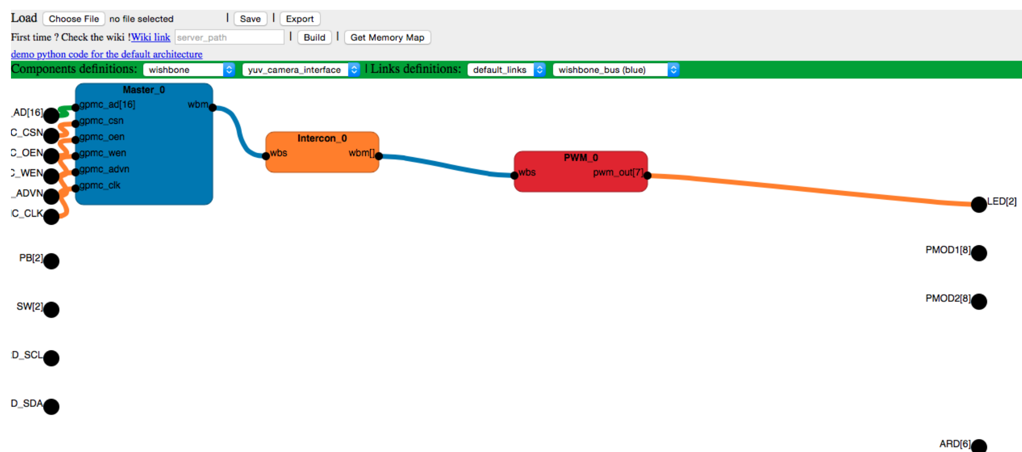


Figure 2.14: Skeleton editor[56]

After the project is created in Skeleton it's possible to save the VHDL code and synthesize it in Xilinx Vivado tool without having to interface with the hardware code. However it's necessary some knowledge of how the software works and the work flow of a hardware project. Beyond that, the Skeleton also provides a map memory for use in the BeagleBone Black to interface with the modules created in the Skeleton. Chapter 4 presents a more deep look in to this software.

2.3 Conclusion

As can be seen there is a huge variety of products in the market that gives some interaction between a CPU and a FPGA. However, all of them have advantages and disadvantages.

Products that have a CPU and FPGA on a SoC are more energy efficient and have a faster communication between CPU and FPGA but have less performance compare to those with discrete CPU and FPGA. The advantages of the softcore CPU is that they are very low power consumption SoC and don't have a physical CPU occupying FPGA fabric, but they are the less powerful.

The discrete CPU and FPGA can have the best of both the FPGA and embedded computing, but consumption is higher and the connection between the SoCs is slower.

The development tools have the problem of requiring detailed knowledge of the FPGA design and HDL code, something that most of software developers don't have. The only one that manages to be simpler to use is the Skeleton from Valentf(x), however the options to configure the FPGA are limited and the interface is confused and messy and requires the use of a Xilinx tool.

Concluding, there are several products that give the possibility of integrate the best of software with reconfigurable hardware, however most of them need previous knowledge in digital design and the ones that have tools to surpass this are very limited.

Chapter 3

BeagleBone Black, Logi-Bone and Tools

In this dissertation, the goal was set to develop a reconfigurable peripheral manager towards robotics, capable of being used by developers without or with little hardware design knowledge.

Develop a whole system from scratch in the time given for this dissertation is almost impossible, so there is the need to use tools already available in the market for some tasks. Valentf(x) software and hardware provide all the necessary tools to allow focusing in the most important objective of this dissertation. A configuration tool capable of:

- Using simple language to interact with it
- Allowing users to easy reconfigure the hardware according to their needs
- Be easy to upgrade and add peripherals and eventually of being used in another hardware.
- Be the most automatically possible, allowing the user to abstract for all the hardware development part
- Capable of being used by developers that don't have hardware design knowledge
- Capable of having customized peripherals

3.1 BeagleBone Black and Logi-Bone

A system with discrete CPU and FPGA provides more flexibility and adaptability and therefore more options for future work. Valentf(x) vision for their products is similar to this dissertation objectives, a reconfigurable system capable of being utilized by users without hardware design knowledge through a set of tools capable of automatize the whole process and development. For that reason and because of Valentf(x) open source tools used for this project the Logi-Bone and BeagleBone Black were the system chosen.

The Logi-Pi and Raspberry Pi system were rejected for this project primary for two reasons, the BeagleBone Black grow in the robotics world makes this computer more appellative for the work, and the communication of the BeagleBone Black with the Logi-Bone is four times faster

than Logi-Pi with Raspberry Pi, 16.6MBs over 4MBs. With the improvement of controllability of robots in todays world, a faster communication is increasingly essential.

BeagleBone Black is a small, affordable, low power computer with several applications in the robotics world. It possesses an ARM processor 1Ghz, 512Mb of memory and can run Linux OS. This board is being more and more adopted by the developers community, is performance and the amount of ports available make it a must have for most robotic applications.

Figure 3.1 presents the BeagleBone Black.

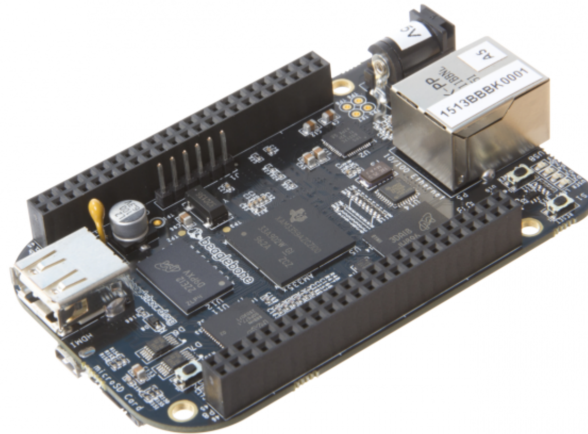


Figure 3.1: BeagleBone Black board[57]

Logi-Bone, presented in chapter 2, is an attachment board with a Xilinx Spartan 6 FPGA for the BeagleBone Black that allows it to be easily morphed in an innumerable amount of digital applications. The simple and straightforward combination of CPU and FPGA provided by the Logi-Bone and BeagleBone Black creates a powerful and versatility system that can be reconfigurable to match the needs of a project and users imagination.

Figure 3.2 shows the Logi-Bone board.

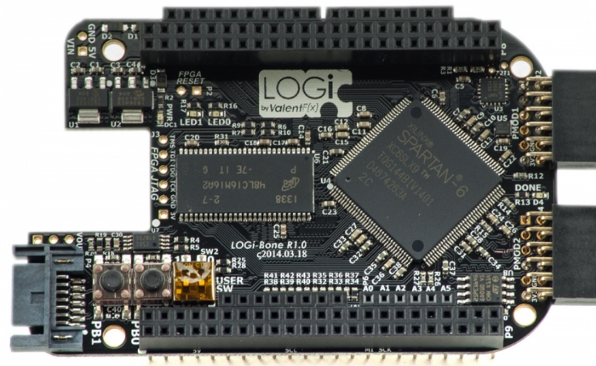


Figure 3.2: Logi Bone board[47]

3.2 Valentf(x) Tools

Most of the companies are aware of the developers paper in their tools development, so the recent trending is for companies to open the code to the public, allowing it to be read, learned, corrected and adapted to the needs of each user. Valentf(x) is no different, providing all of his tools as open-source and allowing them to be used in this project and tailored to his needs.

Valentf(x) provides an Ubuntu OS image containing all the necessary tools and configurations for interact with Logi-Bone and Logi-Pi, however also allows users to use their own Ubuntu or Debian image and install the tools posterior, if the kernel version of it is superior to bone60 [51]. One of the most important tools is the logi-loader, it's a script used to upload a bitstream file with the hardware project directly to the FPGA in Logi-Bone.

The APIs provided, permit a low level interaction with the FPGA from the BeagleBone Black. APIs for writing, reading, opening and closing the connection that interact with a memory file that is mapped to the GPMC. All of the APIs are available in the Valentf(x) repository and come in the pre-built image of Ubuntu provided by them.

The communication between the BeagleBone Black and Logi-Bone is done through a GPMC bus however this comes disable by default in the CPU. A modification of the device tree[58], a way to describe the hardware in the system, is need to enable GPMC, luckily Valentf(x) provides it to users with all the pins information as well as the GPMC signal timings.

One drawback of this system is the lost of HDMI port interface and therefore all the graphical interface, however in autonomous robotics there is rarely the need for it and for programming the BeagleBone, tools like the cloud9 IDE can be used.

3.2.1 Skeleton

As mentioned before, Valentf(x) company shares the ambitions and objectives of this dissertation. To ensure that users without hardware design knowledge are capable of using his devices, they provide the Skeleton editor, responsible of generating HDL code with a simple drag and drop blocks and connections interface, figure 3.3.

Although the effort of Valentf(x) work in bringing reconfigurable hardware towards software developers, Skeleton editor fails the propose of being a simple, intuitive and easy to use tool. For that reason, the primarily focus on this work was to develop a tool capable of generate a project for Logi-Bone as well as provide the necessary APIs focused on the peripherals chosen by the user.

When a project is tested in the field there is always the need of doing last minute alterations. In a situation where the user is testing a robotic application in a marine or in a desert field, skeleton as online tool forces the use of a internet connection to make small changes in the project, which most of the times isn't available.

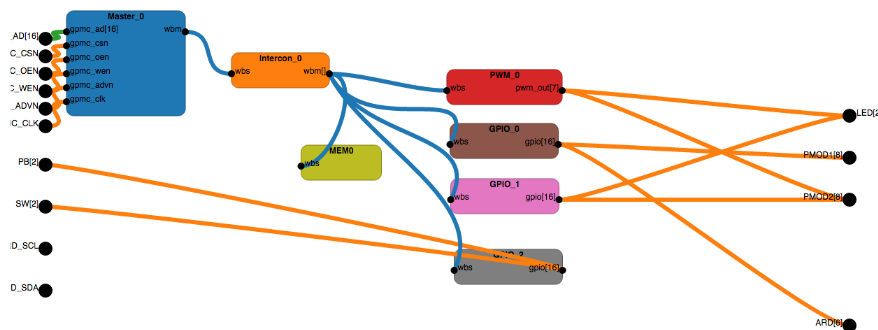


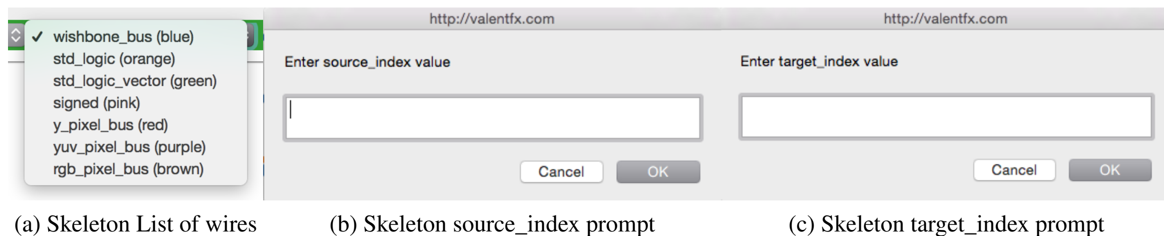
Figure 3.3: Skeleton interface with a simple project configuration

Figure 3.3 shows a simple project with three GPIOs, a PWM and a Memory block. The interface is very confusing and it's impossible to see if the GPIOs are inputs or outputs and to which ports are the blocks connected.

To complicate things more, Skeleton lets the user choose from several number of wires, what for experienced users facilitates, for others makes it more confuse and difficult to use. In addition, for connecting a block to a port the user has to select a wire from the menu, figure 3.4 a, connect the block and respond to 2 prompt messages, figures 3.4 b) and 3.4 c).

Connecting several blocks using single wires causes a very confusing design with multiple wires across the project. One alternative to reduce it is to use vector wires, they can connect multiple outputs of the same block to multiple ports of the same connector. This solution enables the need of responding to another prompt question, asking the size of the vector. Vector wires have the disadvantage of forcing users to use sequential ports as outputs of the same block.

Skeleton presents several blocks that can be used:



(a) Skeleton List of wires (b) Skeleton source_index prompt (c) Skeleton target_index prompt

Figure 3.4: Blocks and parameters generated

- Whisbone_register
- Whisbone_pwm
- Whisbone_servo
- Whisbone_gpio
- Whisbone_gps
- Whisbone_ping
- Whisbone_mem
- Whisbone_watchedog
- Whisbone_7seg4x
- Whisbone_fifo
- Whisbone_pmic
- Whisbone_to_xil_fifo

There is a considerable number of blocks for such an early project, however there is no explanation whatsoever of the blocks, how they work and how to interact with them, and although some of them have suggestive names, others are almost impossible to identify what are and their purpose.

An output port of a digital circuit can only have one source. Multiple sources on the same output port cause a design error derivation of being impossible to choose which signals to propagate to outside world. A error should be expected by Skeleton editor, when connecting two blocks to the same port yet, that doesn't happen, the user only finds out the error when the code is synthesize by Xilinx tool. Actually at the moment Skeleton appears to allow everything without giving error, include out of range index's and wire vectors bigger than the block allows.

It's typical to have the need of changing something in a already made project, Skeleton allows to save and load projects, however it doesn't gives the option to see the wire information making the user guess the wire corresponding to the alteration pretended.

Besides it, it's important to allow that more experience users with hardware design knowledge integrate their blocks in an easy way without having to understand the entire project or having to develop a project from scratch.

3.3 Xilinx ISE

FPGAs are programed through a bitstream file generated from one of the several tools, in the case of the Spartan 6 the Xilinx ISE (Integrated Synthesis Environment) 14.6, figure 3.5. Unfortunately the use of this software is mandatory for generating the bitstream, with present two problems:

- The user has to know how to create a project and generate the bitstream using ISE
- Forces the use of a computer to run ISE and generate the bitstream.

This dissertation brings a solution for the first problem by using a script to automatize the project creation and bitstream generation, the user only needs to install the software in his computer.

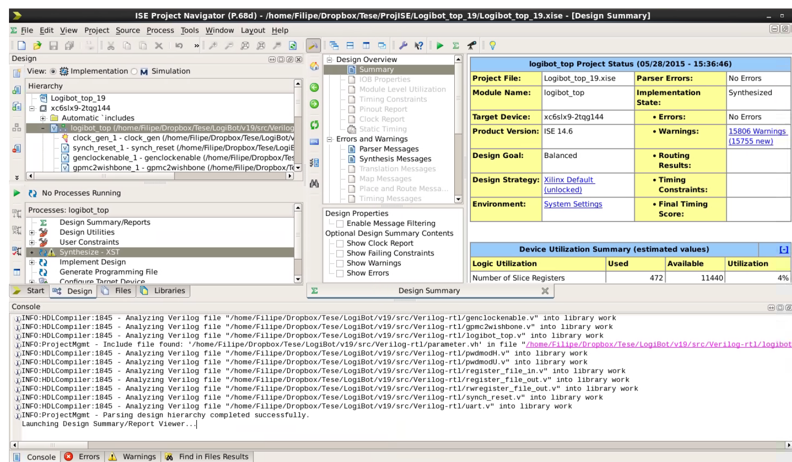


Figure 3.5: Xilinx ISE program

3.4 Communication between BeagleBone Black and Logi-Bone

BeagleBone and the Logi-Bone communicate through GPMC (General Purpose Memory Controller) bus, using the following signals:

- GPMC_CLK: Signal generate by BeagleBone during a read or write operation to synchronize the bus.
- GPMC_AD: Address and data bus (16-bit wide), it's an input and output bus.
- GPMC_ADVn: When asserted low the data in the AD is an address.
- GPMC_CSn: Chip Select signal.
- GPMC_OEn: Output enable signal, when asserted low indicates an on going read. The bus became in high impedance when OEn is low to allow the slave to take control (in read operations).
- GPMC_WEn: Write enable signal, indicates an on going write when asserted low.
- GPMC_BEn: A 2-bit signal to determinate the number of bytes of data used in communication.

GPMC in BeagleBone Black has various modes of operation, this work uses a synchronous without burst mode to write and read operations. In this mode, the bus has a speed of 16.6MB\s. Figures 3.6 and 3.7 present a read and write respectively from BeagleBone GPMC.

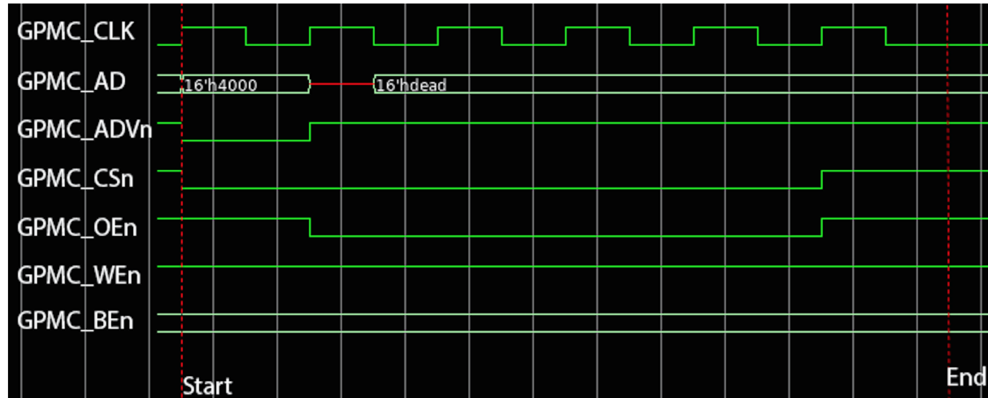


Figure 3.6: Read operation in GPMC bus

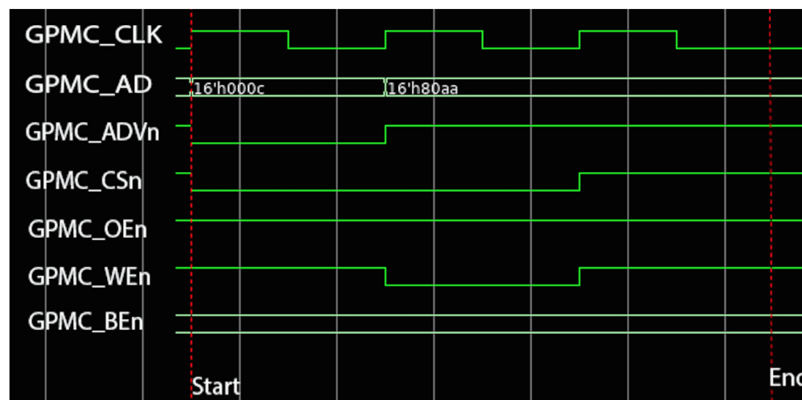


Figure 3.7: Write operation in GPMC bus

The write operation only takes three GPMC clocks unlike a read operation that takes six CPMC clocks.

It's to wonder why the read timing is so conservative, twice of the write time, when shorter timing would enable faster access and communication. GPMC clock is only active when a transfer occurs, meaning that this clock can't be used to clock the FPGA. For that reason the FPGA has to use is own clock, a 50Mhz oscillator (100Mhz generate through FPGA on-board PLL), making impossible to infer about the clocks synchronization in each transfer.

The extra time in a read operation allows the FPGA to synchronise the signals, compute the address and send the data back to the GPMC_AD bus.

The communication between the BeagleBone and the Logi-bone is done through several layers of software and hardware. Figure 3.8 presents a diagram of it.

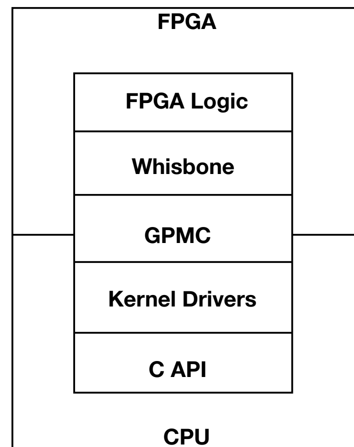


Figure 3.8: Layers of communication between the BeagleBone Black and the FPGA

3.5 Conclusion

Valentf(x) tools allows to save time and focus on the most important part of this work. Logi-Bone and BeagleBone integration provides an excellent platform for demonstrating the concept of this dissertation.

Skeleton editor provides an easy way to generate a hardware project, but presents several problems that difficult the use of it.

GPMC bus provides a reliable and fast way to make the BeagleBone Black and Logi-Bone communicate and exchange information.

Chapter 4

Hardware Project

The work accomplished in this dissertation can be divided in two parts, the hardware project and the configuration tool for it. This chapter presents the first, divided in two parts, the first involving the clock synchronization, the wishbone bus and the low level interface (registers and memories) and the second part containing the peripherals.

Although the hardware and software are presented in separated chapters, they were developed side by side and with their integration in mind. Reconfigure a project is easily done by making it based on parameters, compile directives and include files, however this makes his development more complicated. To accomplish that some strategies were used.

4.1 BeagleBone and FPGA Synchronization

As mentioned earlier, the Beaglebone Black and Logi-Bone use two different clocks, to ensure a proper communication between them there is a need to synchronize the signals between them. It was used a a technique called two flip-flop synchronizers in order to do that.

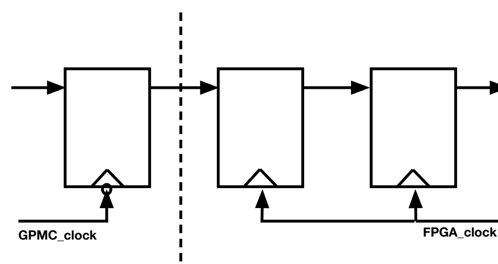


Figure 4.1: Two flip-flop synchronizer diagram

Figure 4.1 shows the two flip-flop synchronizer scheme, regardless his name, this technique actually uses three flip-flops. Bringing the probability of the signal being in a meta-stable after passing through the flip-flops close to zero.

Despite the possibility of acquire the data correctly after the first flip-flop only after the second one there is certain that his value is correct. A two clock waiting causes a delay in the communication between the BeagleBone Black and Logi-Bone making it slower.

4.2 Wishbone

GPMC_AD, as explained in previous chapter is a 16-bit bus responsible for transfer address and data from/and to the BeagleBone, making it an input and output bus with multiple purposes. The use of this bus inside the FPGA to interact with the blocks not only complicates the whole project but also enforces the use of more logic occupying more space inside the FPGA.

Wishbone comes from the need of creating a common interface between IP cores improving the portability of projects and reliability. It separates the data written, data read and address in distinctive buses, making it more easy and straightforward to use. For that reason was chose to decode the GPMC into wishbone signals.

Although wishbone has more signals, the only ones used in this project are:

- WB_ADDR: contains the address and is used to choose from which blocks the data is going to be read or written
- WB_WRDATA: contains the data from wishbone to send to other blocks
- WB_RDDATA: contains the data send it to the wishbone from other block
- WB_WREN: it's a single bit that goes high during a write operation to give the information that is for write the value of the write data in the address provided by the wishbone

GPMC to Wishbone block implements the signals decodification and synchronization between the BeagleBone Black and the FPGA using the two flip-flop technique mentioned above.

Figure 4.2 and 4.3 shows the wishbone signals converted after a read and write operation from GPMC respectively, for better understand it was used the same write and read operation presented in figure 3.6 and 3.7.

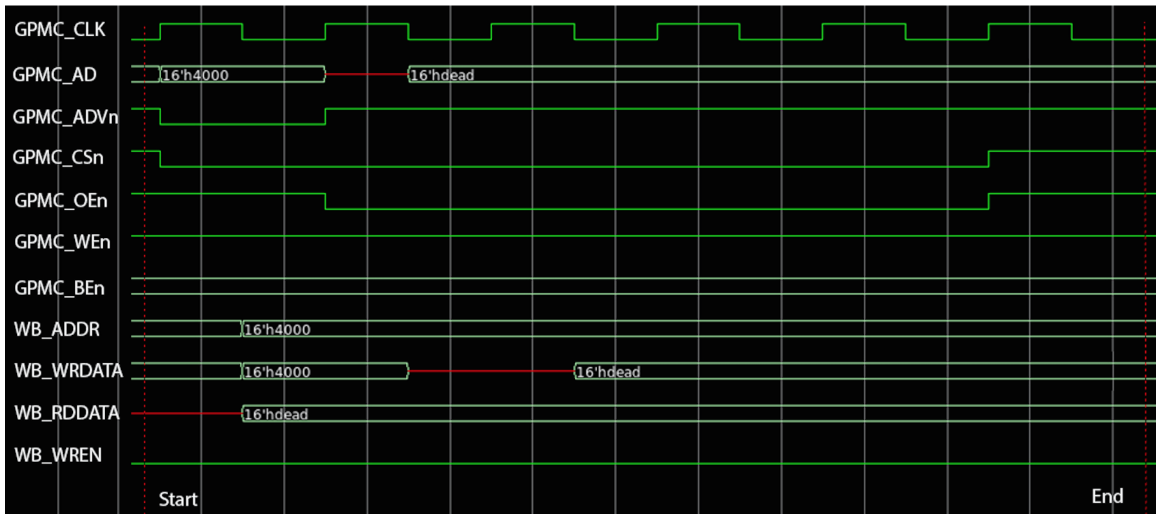


Figure 4.2: GPMC and Whisbone read signals

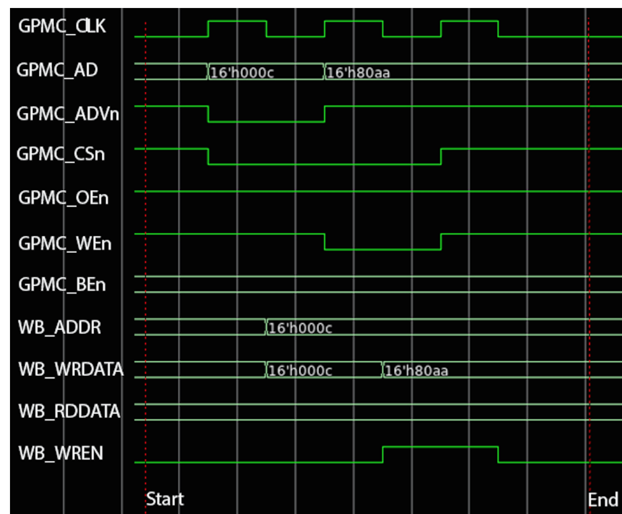


Figure 4.3: GPMC and Whisbone write signals

It's possible to see that even in read operations the write data signal of the wishbone has some value but it is only propagated to the rest of the project if the write enable signal is high.

Figure 4.4 presents the GPMC to Wishbone block diagram.

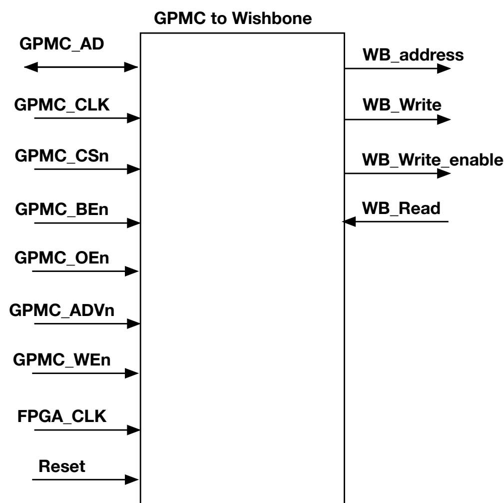


Figure 4.4: GPMC to Whisbone block diagram

4.3 Registers and Memories

Most of the peripherals have registers interfaces, including the ones used in this project, for that reason it was chosen not to use wishbone compatible peripherals and instead connect the wishbone to a bank of registers and memories. This puts another layer between the BeagleBone Black and the peripherals.

Five different blocks are connected to the wishbone bus:

- Register output write only (W_RO)
- Register output write and read (WR_RO)
- Register input read only (R_RI)
- Register input read and write (RW_RI)
- Memories (Up to 8 memories) (MEM)

Depending of the peripherals chosen by the user these blocks can be generated or not.

The 16-bit width WB_ADDR signal gives 64Kbits of address space to connect the registers and memories. The first approach to distribute the blocks by the address space was to connect all the blocks in a sequential manner. The first register of the second block would start right after the last register of the first block, and so on however using this approach, forces the comparison of address signals. Comparing 16-bit width signals implies a large amount of logical blocks occupying more space in the FPGA fabric.

The solution chosen was to connect the blocks to the same address space independently of their size. Using the three most significant bits of the address bus, a simple 3-bit equal operation is sufficient to compute the address, requiring less logic and less space.

Figure 4.5 shows the mapping of the 16-bit GPMC address space.

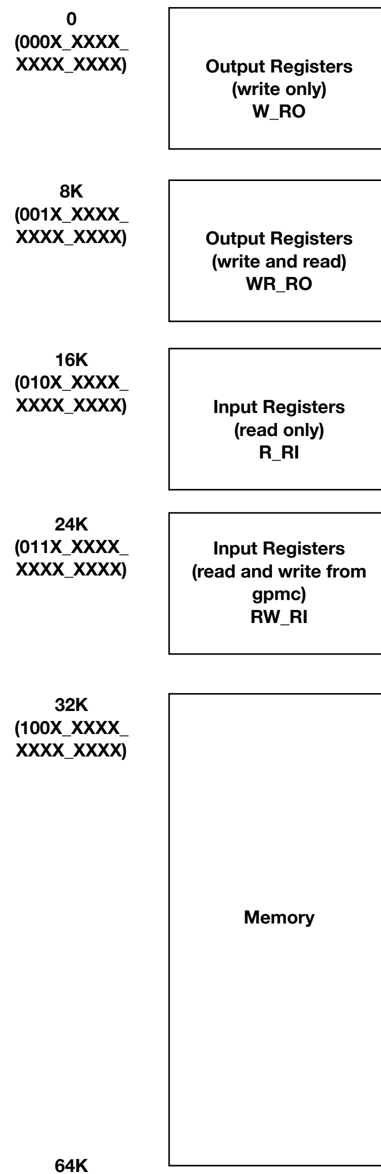


Figure 4.5: Map of the 16-bit GPMC address space

4.3.1 Registers

One of the matters of discussion was choosing a width for the registers. The ideal solution is to have each register with the width corresponding to the needs of the peripheral attached to it. However, doing that in a generic and easy to reconfigurable way is extremely difficult not to say almost impracticable, so a fixed width for the registers was decided to be the best for this work.

Every multiple of 2 (2, 4, 8, 16, 32, 64, etc.) are a possibility for the register width. Using 2, 4 or 8-bit allows writing multiple registers at once, 16-bit writing one each time, and 32, 64, etc. implies several writes to complete the same register.

In theory registers banks with 8-bit width is an accepted solution, but in practice that's not the case. Taking into account that in each GPMC interaction uses 16-bit words the value of a write operation would have to be saved in two registers. In a case where two different peripheral are connect to adjacent registers, a write operation for one will overwrite the other data, of course this situation can be surpass by first read the value of the registers, making an aggregation of that with the value wanted and then writing in the register bank. However, this implies making two GPMC operations slowing down the communication.

Using registers wider that 16-bit allows more data in the same register however, the most common peripherals don't take advantage of this number of bits. Naturally there is the possibility of using the same register for two or more different peripherals but that brings the problem of overwrite the data of one when try to write the other like mention above. Another problem is the need of more logical components because it takes at least two operations to completely write a register so there is the need to save all the data until all the bits of the register are received from the GPMC bus.

A 16-bit wide configuration was chosen for the registers blocks, not only because of the 16-bit wide GPMC bus but also because most of the peripheral need more than 8-bit but less than 32-bit.

4.3.1.1 Register Output

Most of the peripheral have a register interface, in other words, their data is obtained and/or store in a register. The register output blocks are stacks of registers written by the wishbone where the peripheral can be connected to collect data inputs.

In this project there are two different types of register output blocks, from the point of view of the wishbone, the one with write only capability (W_RO) and other with write and read capability (WR_RO).

There is no need for reading the value written in the register in most of the cases, for that reason most of the peripherals are connected to the W_RO block, reducing the FPGA occupation.

To make a hardware design easy to reconfigure and adapt to the user needs various strategies had to be used during this work. In a fixed design the number of registers need in each block is known and the blocks can be declared with that exactly number, code 4.1.

Due to the reconfigurability of the design, the number of registers in the block can change from one to a maximum of 8K registers, creating a problem as how to define a block with variable number of outputs. This problem could be easily solved if the Verilog language allowed to declare arrays of inputs or outputs, and it would be instantly solve by allowing to change the block number of outputs with a simple parameter.

However having that limitation, several solutions were taken in consideration, in this particular case, two of them were debated. One solution was to have a tool to generate the entire code of

```
1 module W_RO (
2         clock,
3         reset,
4         wb_rddata,
5         wb_wrdata,
6         wren,
7         RO0,
8         RO1,
9         RO2
10    );
```

Code 4.1: Module W_RO declaration for fixed size

the block and is declaration in the top level, but this implies a substantial increase in the software complexity and an increase of bugs probability.

The other solution, the adopted one, was instead of having several output registers (RO0, RO1, etc.) from the block have only one output bus of parameterizable size, that is 16 times the number of registers need it, and outside and inside the block separate that bus in 16-bit buses.

Using this method it's possible to configure the number of register in the blocks, using the generate option of Verilog. Code 4.2 presents this method.

R_O is an output of the W_RO block with parameterizable width using the parameter WIDTH_OUTPUT. Inside the block, an array of 16-bit registers is connected to the R_O bus merging the exits of the several registers in a single bus.

In the top level, the inverse is done by separating the R_O bus in an array of 16-bit wires so that can be more easily connected to the peripherals, code 4.3.

Figure 4.6 illustrates this method.

This strategy is also used in the WR_RO block as well as in the input register blocks, R_RI and RW_RI.

4.3.1.2 Register inputs

While in the register output blocks the registers are written by the wishbone and read by the peripheral the input blocks are the other way around, they are written by the peripherals and read by the wishbone.

4.3.1.2.1 R_RI

Although the register name in the category of R_RI block in reality there is no register, the block only decodes the address and does pass-through of the respective wires to the wishbone.

```
1 module W_RO(
2         clock,
3         reset,
4         wb_rddata,
5         wb_wrdata,
6         wren,
7         R_O
8     );
9
10 output        [(WIDTH_OUTPUT * 16) - 1 : 0]   R_O;
11 wire          [(WIDTH_OUTPUT * 16) - 1 : 0]   R_O;
12 reg           [15:0] RO [WIDTH_OUTPUT - 1:0];
13
14 genvar width;
15
16 generate
17  for(width = 0; width < WIDTH_OUTPUT; width = width + 1)
18  begin : Register_to_vector
19     assign R_O[(width*16) +: 16] = RO[width];
20  end
21 endgenerate
```

Code 4.2: Module W_RO declaration parameterizable

```
1 wire       [15:0] W_RO [WIDTH_OUTPUT - 1:0];
2 wire       [(WIDTH_OUTPUT * 16) - 1 : 0] R_O;
3
4 genvar w_width;
5
6 generate
7  for(w_width = 0; w_width < WIDTH_OUTPUT;w_width = w_width + 1)
8  begin : w_register_output
9     assign W_RO[w_width] = R_O[(w_width*16) +: 16] ;
10  end
11 endgenerate
```

Code 4.3: Separation of the number of registers in top level

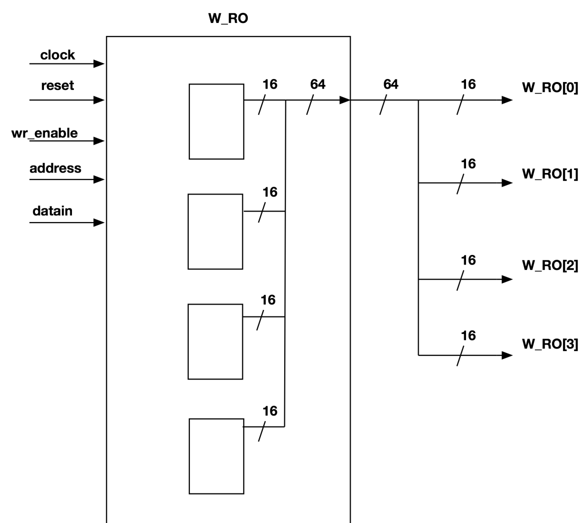


Figure 4.6: Block W_RO illustrating the method used to have a parameterizable number of registers

4.3.1.2.2 RW_RI

The addition of some peripheral in the project like the UART, opened the necessity of having another type of input register blocks, in this both the wishbone and the peripheral can write in his registers.

Taking the UART block as example, when data is received in the RX port is processed and put in an output register, and data ready bit goes to high during only one clock.

The probability of read the data ready bit is almost zero, that creates a problem of knowing when is new data or older. One method for the user to know that is to save the last read value from UART and compare with the one read, but even so this posses a problem in cases when the same value is received two or more times in a row.

Figure 4.7 shows the diagram of the block containing only a single register.

Connecting the data ready signal of the UART to the enable wire makes that only when this goes high the new data is saved in the register. After reading the register, the user can write in it a default value assuring the ability of identified a new received value.

Although the example given is for the UART port this block can be connected to other peripherals.

4.3.2 Memory

The project allows up to eight RAM memories with variable width and size. Although the memories occupy 32Kbits of the address space provided by the wishbone address. This space divided by eight allows a maximum size for each memory of 4Kbits but if only one is generated in the project that one can have a maximum size of 32Kbits .

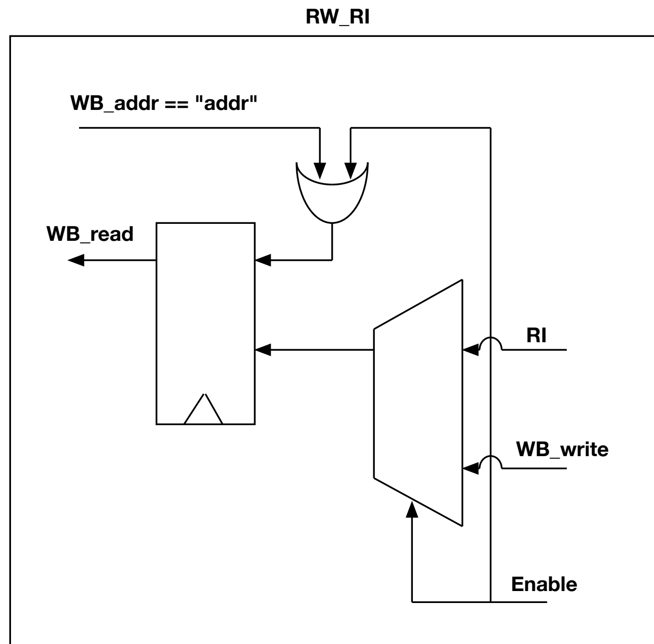


Figure 4.7: Block RW_RI with a single register

Table 4.1 indicates the maximum size of each memory depending of the quantity of them.

Table 4.1: Distribution and size of memories depending on his quantity

	Memory 1	Memory 2	Memory 3	Memory 4	Memory 5	Memory 6	Memory 7	Memory 8
1	32Kbits	x	x	x	x	x	x	x
2	16Kbits	16Kbits	x	x	x	x	x	x
3	16Kbits	8Kbits	8Kbits	x	x	x	x	x
4	8Kbits	8Kbits	8Kbits	8Kbits	x	x	x	x
5	4Kbits	8Kbits	8Kbits	8Kbits	4Kbits	x	x	x
6	4Kbits	8Kbits	8Kbits	4Kbits	4Kbits	4Kbits	x	x
7	4Kbits	4Kbits	8Kbits	4Kbits	4Kbits	4Kbits	4Kbits	x
8	4Kbits	4Kbits	4Kbits	4Kbits	4Kbits	4Kbits	4Kbits	4Kbits

Figure 4.8 shows the distribution in the address space of each memory.

Using this configuration, although enforces the maxim of 4Kbits if there is the need of eight memories also allows an even repartition of memories sizes as well as facilitates the memory access and decode from wishbone. The user can choose the memory width going from 8-bit to 16-bit and can be different for each memory in the project.

The memory itself it's a normal dual-port RAM, having one side connected to the wishbone bus and another to a peripheral.

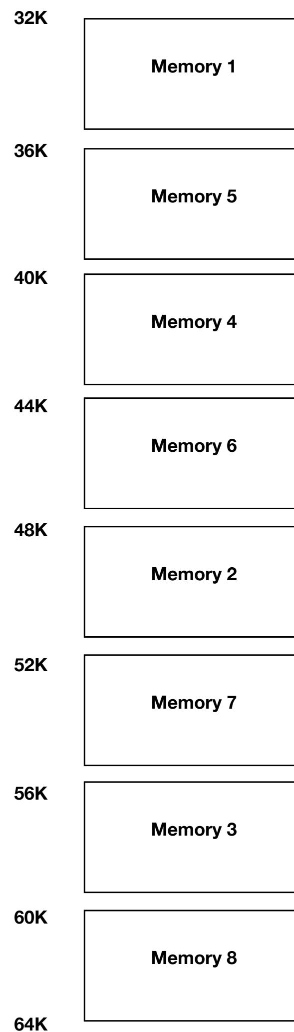


Figure 4.8: Distribution of the memory address space

4.4 Generation of block, parameters and techniques

The structure presented above is the basics for the project developed, but as mentioned before isn't permanent. The blocks are generated or not according to the users specifications and peripheral chosen. The only exception is the R_RI block that is always generated and has at least two registers with a fixed value of 16'hdead and 16'hbeef respectively that are used to debug and test the communication.

Using parameter and compiler directives such as *include* and *'defines* it's possible to generate the configuration need.Table 4.2 presents the parameters and compiler directives used for each of the mentioned blocks.

To generate the number of memories need by the project is used the generate function from Verilog showing in the following code. This allows generating multiple instances of the same

Table 4.2: Parameters and compiler directives for registers and memories

	Parameters	Observation	Compiler directives	Observation
W_RO	W_WIDTH_OUTPUT	Indicates the number of registers to be generate	REGISTER_OUTPUT_W	When defined generates the block
WR_RO	WR_WIDTH_OUTPUT	Indicates the number of registers to be generate	REGISTER_OUTPUT_WR	When defined generates the block
R_RI	WIDTH_INPUT	Indicates the number of registers to be generate	REGISTER_INPUT	When defined generates the block
RW_RI	WIDTH_INPUT_W	Indicates the number of registers to be generate	W_REGISTER_INPUT	When defined generates the block
Memory	MEM_NUMBER MEM_SIZE MEM_WIDTH MEM_ADDRSTART	Number of memories to be generate Size of each memory Width of each memory Address start of each memory	MEM	When defined generate one or more blocks of memories

block without having to instance it multiple times in the top level, code 4.4.

Once again, Verilog limitations make the generation of diferent memories more difficult by not allowing an array of parameters. To surpass that was used the same parameter for all the memories, and concatenate all the values in it with a number of fixed bits for each.

parameter MEM_SIZE = {16'b1024, 16'b512};

In this way in the generate function is possible to scroll inside the parameter from 16-bit to 16-bit using the genvar variable.

MEM_SIZE(MEM_SIZE[(mem_number*16) +: 16])

This solution increase the difficult of the configuration tool developed, however it's the only way to allow having the same block generated multiple times with different configurations.

```
1 genvar  mem_number;
2
3 generate
4  for(mem_number=0;mem_number<MEM_NUMBER;mem_number=mem_number+1)
5   begin : mem_generate
6    DPRAM_2    dpram0
7   (
8        ...
9   ...
10    );
11   end
12 endgenerate
```

Code 4.4: Generation of multiple memories

4.5 I/O Ports

Logi-Bone has 21 ports that can be used to connect sensors, actuators, adc's or others.

The user should control where to physical connect his devices in Logi-Bone. This presents several challenges as to permit to configure a port as input or output depending what the user choses. The first solution presented was to divided the ports as inputs and outputs, in this way the user had some flexibility being able to chose where to connect is peripheral, but respecting the outputs and inputs.

This approach besides not giving total control of the configuration to a user, limits to half the number of outputs and inputs, not allowing in a extreme case to have all the ports as output or all as input.

The second method tested was to declared all the ports as a vector and using compiler directives (*'defines*) and choose for each member of the vector if it's an input or output. However that is not allowed by the synthesis program, the vector has to be input or output can't have members as input and others as output.

The solution comes in the form of compiler directives, in each if the directive associated to a port is defined that port is an output else is an input.

```
1 `ifdef PMOD1_0
2     output pmod1_0;
3     assign pmod1_0 = out[0];
4 `else
5     input pmod1_0;
6     assign in[0] = pmod1_0;
7 `endif
```

Code 4.5: Declaration of ports as input or output

Integrate ports with different names in the peripherals and at the same time giving the possibility to the user of choosing the port for each peripheral creates a difficult problem. To solve this problem the *in* and *out* vectors are used. Using *in* and *out* vectors to connect the peripherals to the ports makes the connection of the ports and peripherals as easy as to put the correct index in the vector using a parameter for the index, code 4.5.

Of course, that if the port is declared as output the port position in the *in* vector is left unconnected, but the synthesis tool decodes that as being connected to GND.

4.6 Peripherals

The peripherals used in this work were made and tested outside of the dissertation environment. Some of them were lightly adapted but most of them were capable of being integrated quickly and easily. Because the primary objective of this work is to develop the complete workflow for a reconfigurable project and having limited time, was preferred to have a small amount of peripherals but a complete project and a solid based for easier upgrading the system.

However, the capability of adding custom blocks highly increases the capability of customizing and upgrading the project.

4.6.1 Digital I/O

The most basic interface in a digital circuit, the digital input and output ports are essential to any robotic project. A digital output port receives a 1 or 0 and outputs VCC or GND, digital input does the opposite, receives VCC or GND from the outside and outputs to the system 1 or 0.

Digital I/O allows to be set in any port of Logi-Bone, allowing user total control of where to connect his devices. This block works with only one bit as input or output and is connect to a bit of a register of WR_RO or R_R blocks respectively. In a single register is possible to have up to 16 digital inputs or outputs. Connecting the digital output to the WR_RO block allows changing a single bit, by reading the corresponding register and changing a single bit and writing the data back to the register is assure that the only change happens in the bit modified.

This whole operation is camouflaged from the user by using the API designed for interact with digital output ports.

This blocks is the ones that occupy less space in the FPGA fabric, table 4.3.

Table 4.3: Digital I/O FPGA occupation

Digital	Input		Output	
	Quantity	Occupation	Quantity	Occupation
Maximum Frequency	210.700MHz		114.626MHz	
Number of Flip Flops	19	1%	113	1%
Number of LUTs	16	1%	31	1%
Number of Slices	6	1%	33	2%

4.6.2 PWM Unidirectional

Pulse Width Modulation (PWM) is a periodic digital signal mostly used in robotics to control motors, leds and power controls. The combination of high time and low time of the PWM gives an average value that is interpreted almost as an analogue voltage. This value can be change modifying the high time and low time.

Although the most used frequency for PWM is 20Khz, in this work is possible to choose between 20Khz, 100Khz and 200Khz frequency as well as the corresponding Logi-Bone port to output the signal

To control the PWM, the block receives an 8-bit word with a signed number between -100 and 100, a 4-bit deadzone value and an enable signal. All of this data fits in an output register so it's connected to only one of them.

Generating the project only occupy a small part of the FPGA, table 4.4.

Table 4.4: PWM Unidirectional FPGA occupation

PWM Unidirectional	20Khz		100Khz		20Khz	
	Quantity	Occupation	Quantity	Occupation	Quantity	Occupation
Maximum Frequency	116.469MHz		103.093MHz		168.350MHz	
Number of Flip Flops	117	1%	117	1%	36	1%
Number of LUTs	89	1%	104	1%	13	1%
Number of Slices	50	3%	49	3%	16	1%

4.6.3 PWM H-Bridge

Several times DC motors are controlled using a H-bridge. H-Bridge, Figure 4.9, is an electronic device that permits that the polarity of a voltage applied to a motor can be changed, allowing a DC motor to rotate in both directions and at different speeds.

S1, S2, S3 and S4 represent the switches used to control the direction and speed of the motor. The PWM H-Bridge block receives a signed number between -100 and 100 and outputs the four necessary PWMs to control each of the H-Bridge switches individually.

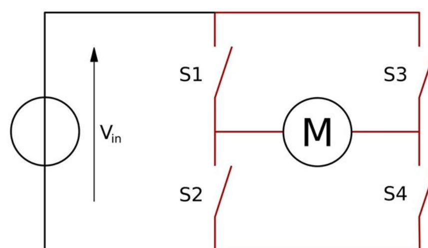


Figure 4.9: H-Bridge [59]

In terms of implementation, this block is very similar to the unidirectional PWM being the only difference the number of outputs, four instead of one.

PWM H-bridge occupation in FPGA is similar to PWM Unidirectional, table 4.5 shows it.

Table 4.5: PWM H-Bridge FPGA occupation

PWM H-Bridge	20Khz		100Khz		20Khz	
	Quantity	Occupation	Quantity	Occupation	Quantity	Occupation
Maximum Frequency	103.605MHz		115.260MHz		168.350MHz	
Number of Flip Flops	118	1%	121	1%	36	1%
Number of LUTs	103	1%	120	2%	13	1%
Number of Slices	57	3%	54	3%	16	1%

4.6.4 UART

There are several communication protocols used in robotics nowadays, maybe one of the most used is UART. UART (Universal Asynchronous Receiver Transmitter) is an asynchronous serial bus, capable of transmit 8-bit data per communication at several different speeds, in this work the maximum speed tested was 115200Kbps, but could be higher.

As mentioned above, this work uses already made blocks from other projects that are tested and with functionality assured, in this case the UART block is for 8-bit, 1-stop bit and no parity communication, for that reason this options are fix and can't be changed.

The Tx port, Rx port and baudrate can be chosen by the user when the block is generated. Uart is connected to the RW_RI block in this way it's possible to have more control over the data received.

For measuring UART occupation and frequency two baudrates were chosen, 9600kbps and 115200kbps, table 4.6.

Table 4.6: UART FPGA occupation

UART	9600kbps		115200kbps	
	Quantity	Occupation	Quantity	Occupation
Maximum Frequency	68.418MHz		68.418MHz	
Number of Flip Flops	208	1%	200	1%
Number of LUTs	129	2%	112	1%
Number of Slices	70	4%	62	4%

4.6.5 Servo Controller

Servo controller block outputs a signal to control servomotors that is similar to PWM but is not defined by the duty cycle (high and low time) but rather by the width of the high pulse.

The servo controller signal has a fixed 50Hz frequency but like all the peripherals, the port for outputting the signal can be assigned by the user.

Table 4.7 presents the occupation of the FPGA when the block Servo is implemented

Table 4.7: Servo FPGA occupation

Servo		
	Quantity	Occupation
Maximum Frequency	162.655MHz	
Number of Flip Flops	36	1%
Number of LUTs	16	1%
Number of Slices	15	1%

4.6.6 SPI

The Serial Peripheral Interface (SPI) is highly used in robotics as a synchronous serial communication. The typical applications include sensors, shift-registers and LCD screens. SPI uses a configuration master-slave, this project implements a master SPI. There are four signals composing the SPI:

- CLK – SPI clock signal
- SDO - Master Output, Slave Input
- SDI – Slave Output, Master Input
- CS – Chip Select

The users can chose the number of bits of the SPI between 8 and 16, and the ports for outputting and receiving the SPI signals. Table 4.8 presents a comparative of the occupation of the FPGA depending on the number of bits (8 or 16).

Table 4.8: SPI FPGA occupation

SPI	8-bits		16-bits	
	Quantity	Occupation	Quantity	Occupation
Maximum Frequency	127.000MHz		113.250MHz	
Number of Flip Flops	112	1%	183	1%
Number of LUTs	94	1%	129	2%
Number of Slices	51	3%	55	3%

4.6.7 Custom blocks

One of Skeleton faults is to not permit in an easy way to integrate custom blocks in a project. In this dissertation a solution that permits this is proposed, this solution allows a developer to integrate their own blocks for interfacing with the outside or simply to help processing or calculating something. Of course, this aims towards users with some knowledge in hardware development.

After some consideration of how to integrate the blocks in the project, the answer came in a form of a template. The users should follow the templates to ensure that the block is perfectly

```
1 custom_r   custom_1  (
2           .clock                      (clock100M),
3           .reset                      (reset),
4           .register_input         (W_RO[CUSTOM0_RO_OFFSET0]),
5           .register_output            (RI[CUSTOM0_RI_OFFSET0]),
6           .port_input          (in[CUSTOM0_IN_OFFSET0]),
7           .port_output         (out[CUSTOM0_OUT_OFFSET0])
8    );
```

Code 4.6: Template of the custom_r blocks

integrated in the whole project and works correctly. There are two types of blocks, custom register block (custom_r) and custom memory block (custom_m) the first connects to registers and the second to memories in the top level of the project. The code 4.6 presents the template for a custom_r block.

This template is from a custom block connected to registers (input and output). The connections to the top level have to be done as the template specifies for a successful integration. Table 4.9 presents the connections.

Table 4.9: Parameters for custom_r blocks

	Wire	Parameter
Clock	clock100M	
Reset	reset	
Input	To the block W_RO	CUSTOMx_RO_OFFSETy
Output	From block R_RI	CUSTOMx_RI_OFFSETy
Port input	in	CUSTOMx_IN_OFFSETy
Port output	out	CUSTOMx_OUT_OFFSETy

Code 4.7 and table 4.10 presents the template and parameters for custom blocks connected to memories.

In table 4.9 and 4.10 the parameters have x and y, the first represents the number of each custom blocks, starting at zero, and the other the number of the same type of connecting in the block. For example if there is two custom blocks connected to registers in the same project, the parameter for a third output register of the second block is CUSTOM1_RI_OFFSET2.

The user can make the number of custom blocks that he wants and can choose the any name for it. To integrate it's need to past the template filled between the respective defines in top level, in this case between '*ifdef* CUSTOM_R and '*endif* or '*ifdef* CUSTOM_M and '*endif* for custom block connected to register and memories respectively.

The software tool presented in next chapter is responsible to generate the correct parameters so that the integration becomes flawless. These blocks allow users much more flexibility and in the process saves time by given a stable platform and project.

```
1 custom_m  #(.CUSTOM0_MEM_BITS  (CUSTOM0_MEM_BITS),
2             .CUSTOM0_MEM_SIZE  (CUSTOM0_MEM_SIZE))
3            custom_m1  (
4               .clock               (clock100M),
5               .reset               (reset),
6               .enable              (mem_enable[CUSTOM0_MEM]),
7               .write_enable        (mem_w_ren[CUSTOM0_MEM]),
8               .data_in             (mem_dataout[CUSTOM0_MEM]),
9               .data_out            (mem_datain[CUSTOM0_MEM]),
10              .address             (mem_addr[CUSTOM0_MEM]);
11              .port_input          (in[CUSTOM0_MEM_IN0]),
12              .port_output         (out[CUSTOM0_MEM_OUT0])
13         );
```

Code 4.7: Template of the custom_m blocks

Table 4.10: Parameters for custom_m blocks

	Wire	Parameter
Clock	clock100M	
Reset	reset	
Memory enable	mem_enable	CUSTOMx_MEM
Memory Write enable	mem_w_ren	CUSTOMx_MEM
Data from memory in to the block	mem_dataout	CUSTOMx_MEM
Data from block to memory	mem_datain	CUSTOMx_MEM
Memory address	mem_addr	CUSTOMx_MEM
Port input	in	CUSTOMx_MEM_INy
Port output	out	CUSTOMx_MEM_OUTy

4.7 Conclusion

Coding hardware for generic and reconfigurable purposes is time consuming and very complex. Some limitations in Verilog language forced the use of some tricks to accomplish the goal.

The hardware part was developed a the same time as the software presented in next chapter. Because of this, some of the hardware development was complicated to make the software development more straightforward.

A complete reconfigurable project makes the development much more difficult and also increases slightly the FPGA occupation.

Chapter 5

Project Configuration and implementation

Previous chapter described the hardware developed. The reconfigurability of it was made using simple parameters and compiler directives, however there is the need of hiding all this technical details from users. A software tool was designed with that purpose. This tool is capable of generate all the parameters need for the project from a single file containing the peripherals description using a simple and straightforward language.

Such a simple language takes the hardware reconfigurability and the whole project more close to software developers and anyone without hardware design knowledge.

5.1 Language

The language was created to be simple, intuitive, easy to read and memorise and the most transparent possible allowing to be used and understood by everyone. Similar to a function in C code, the language created, contains the name of the peripherals, follow by the corresponding parameters separated by commas.

Table 5.1 presents the block names and is parameters, follow by an example to each peripheral.

Table 5.1: Language and parameters for generate peripheralas c

Blocks	Parameters				
PWMU	Frequency	Port	x	x	x
PWMH	Frequency	Port1	Port2	Port3	Port4
UART	Baudrate	Tx_port	Rx_port	x	x
SPI	Number_bits	SCLK_port	SDO_port	SDI_port	CS_port
DIGITAL_IN	Port	x	x	x	x
DIGITAL_OUT	Port	x	x	x	x
SERVO_CONTROL	Port	x	x	x	x

Besides these blocks, there is the custom ones that don't have a fixed number of parameters but depends of the user design. Nonetheless the language is in the same way easy and intuitive.

CUSTOM_R (Number of R_WO registers used in the block,
Number of RI registers used in the block,
Number of the output ports,
Output ports (this is one for each of the number of ports given in the parameter above),
Number of input ports,
Output ports (this is one for each of the number of ports given in the parameter above)
)

CUSTOM_M (Memory size,
Memory width,
Number of the output ports,
Output ports (this is one for each of the number of ports given in the parameter above),
Number of input ports,
Output ports (this is one for each of the number of ports given in the parameter above)
)

In the following list, there is an example for every block.

- PWMU (20K,ARD_0)
- DIGITAL_IN (PMOD2_5)
- UART (115200,PMOD1_2,PMOD2_6)
- DIGITAL_OUT (PMOD1_3)
- PWMH (20K,PMOD2_2,PMOD2_3,PMOD2_4,PMOD2_7)
- PWMU (20K,ARD_1)
- CUSTOM_R (2,2,2,PMOD1_5,PMOD1_7,1,ARD_2)
- CUSTOM_M (512,8,1,PMOD1_6,1,ARD_3)
- SPI (16,PMOD1_0,PMOD1_1,ARD_4,PMOD2_0)
- # This is a commentary and ignored by the tool
- SERVO_CONTROL (ARD_5)

In the list above there is also a commentary, for having a commentary is only need that the first character of the line be a #.

5.2 Tool

The reconfigurability of the project is depending on the *parameter.vh* file which contains the correct defines and parameters to configure the project according to the user needs. The configuration tool was developed so that can interpret the language described above and generate the correct file. The software reads a file describing the project line by line, so each peripheral should be declared in its own line for the correctly function of the tool.

The tool starts with the first line and goes down the document, it checks if the line is a commentary by seeing if the first character is a #. The commentaries are ignored existing only to help the user. If the line doesn't have a commentary then the tool search for the keywords, these keywords are the names of peripherals. In the case of a line don't having the keywords and not being a commentary the same is printed in the console as warning and ignored by the tool. When the keyword is found, the tool saves the peripheral information in a C++ object corresponding to it.

Although the tool isn't focused in checking the validity of the information provided by the user, some verifications are made. Verifying the existence of the port, as well as his repetition on the design is essential to avoid errors in the project. Another verification is in the PWMs frequency and in the SPI bit number.

Figure 5.1, 5.2 and 5.3 presents an error from an unknown port, an already use port and PWM unsupported frequency respectively.

```
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_u.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/digital_ports.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_h.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/memory.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/uart.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/spi.cpp
g++ -Wall -Werror -fpic -c main.cpp
g++ -Wall -Werror -fpic pwm_u.o digital_ports.o memory.o pwm_h.o uart.o spi.o ma
in.o -o main


 ERROR:: Port: ARD_6 don't exist
 ERROR in line 1 : PWMU (100K,ARD_6)


ERROR WITH PROGRAM
[Filipe@vlsilabb v27]$
```

Figure 5.1: Error port unknown

```
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_u.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/digital_ports.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_h.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/memory.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/uart.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/spi.cpp
g++ -Wall -Werror -fpic -c main.cpp
g++ -Wall -Werror -fpic pwm_u.o digital_ports.o memory.o pwm_h.o uart.o spi.o ma
in.o -o main


 ERROR:: Port: ARD_0 already in use
 ERROR in line 2 : DIGITAL_IN (ARD_0)


ERROR WITH PROGRAM
[Filipe@vlsilabb v27]$
```

Figure 5.2: Error port already in use

```
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_u.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/digital_ports.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_h.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/memory.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/uart.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/spi.cpp
g++ -Wall -Werror -fpic -c main.cpp
g++ -Wall -Werror -fpic pwm_u.o digital_ports.o memory.o pwm_h.o uart.o spi.o ma
in.o -o main


 ERROR:: PWMU FREQUENCY is : 35K can only be 20K, 100K or 200K
 ERROR in line 1 : PWMU (35K,ARD_0)


ERROR WITH PROGRAM
[Filipe@vlsilabb v27]$
```

Figure 5.3: Error frequency unknown

After analysing all of the user inputs, the program generates the *parameter.vh* file containing the appropriated parameters and directives for the Verilog project. Figure 5.4 show the output of the program (b,c) for the peripherals (a)

Besides this, the program also outputs an *include.h* file containing all of the addresses for communication with the Logi-Bone from BeagleBone Black. Using the defines of the *include.h* file simplifies the interaction by having suggestive names for the peripherals instead of address numbers.

Figure 5.5 shows the include file resulting from the peripherals generated in figure 5.4 a).

blocks.txt

```
PWMU (20K,ARD_1)
DIGITAL_IN (ARD_0)
DIGITAL_OUT (ARD_5)
UART (115200,PMOD1_2,PMOD2_6)
```

(a) Peripherals

parameter.vh

```
`define ARD_1
`define ARD_5
`define PMOD1_2


// UART PARAMETER AND DEFINES
`define UART

parameter UART_NUMBER = 1;
parameter UART_OUT_OFFSET = 0;
parameter UART_IN_OFFSET = 0;
parameter BAUDRATE = { 32'd115200 };
parameter UART_PORT_OUT_OFFSET = { 5'd2 };
parameter UART_PORT_IN_OFFSET = { 5'd14 };


// PWM UNIDERECTIONAL PARAMETER AND DEFINES //
`define PWM_UNI_20

parameter PWMU_NUMBER_20 = 1;
parameter PWMU_PORTS_20 = {5'd17 };
parameter PWMU_OFFSET_20 = 1;

// DEFINES AND PARAMETERS OF DIGITAL_INPUTS
`define DIGITAL_INPUT

parameter DIGITAL_INPUT_NUMBER = 1;
parameter DIGITAL_INPUT_OFFSET = 2;
parameter DIGITAL_INPUT_PORTS = { 5'd16};
parameter DIGITAL_INPUT_ROUND = 16;
```

(b) Parameters generated 1

parameter.vh

```
// DEFINES AND PARAMETERS OF DIGITAL_OUTPUTS
`define DIGITAL_OUTPUT

parameter DIGITAL_OUTPUT_NUMBER = 1;
parameter DIGITAL_OUTPUT_OFFSET = 0;
parameter DIGITAL_OUTPUT_PORTS = { 5'd21};

// DEFINES AND PARAMETERS OF REISTERS
`define REGISTER_INPUT

parameter WIDTH_INPUT = 3;

`define REGISTER_OUTPUT_WR

parameter WR_WIDTH_OUTPUT = 1;

`define REGISTER_OUTPUT_W

parameter W_WIDTH_OUTPUT = 2;

`define W_REGISTER_INPUT

parameter WIDTH_INPUT_W = 1;
```

(c) Parameters generated 2

Figure 5.4: Peripherals and parameters generated

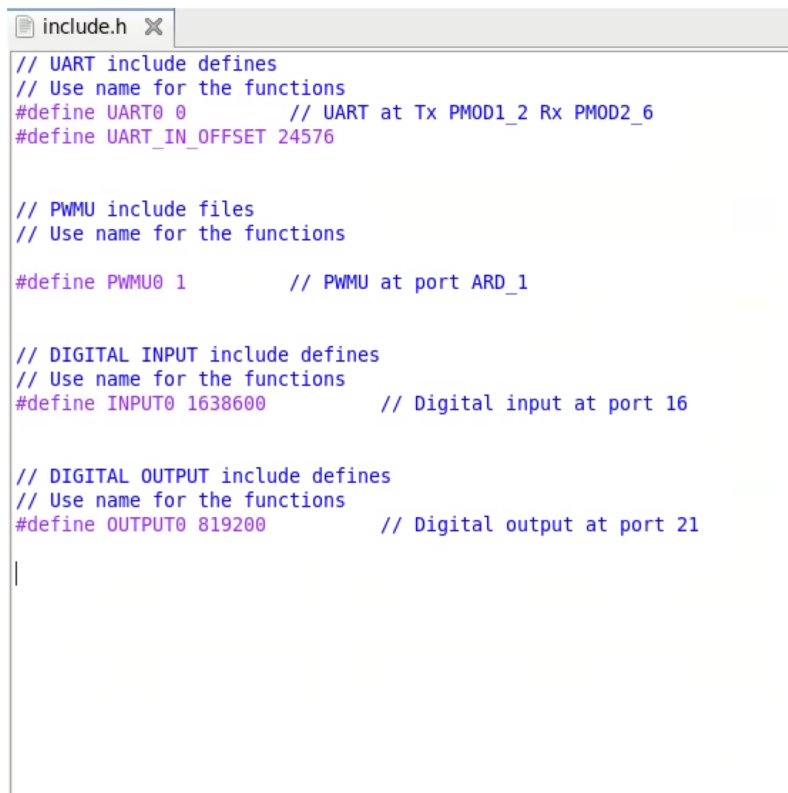


Figure 5.5: Include file, used to interact with the peripherals

5.3 APIs

Valentf(x) provides low level APIs to interface the BeagleBone Black with the Logi-Bone, these APIs allow to opening and closing the communication, write and read to/from a register. Based on that, several peripheral dedicated APIs were created in order to simplify the hardware and software integration and communication.

For example, the API for the PWM receives the his address using PWM0 (a define from the *include.h* file), the value wanted for the pwm and the deadzone value. The API makes the necessaries operations and writes the value in the correspondent register in order to successful generates the correct PWM value.

The list of APIs made:

- bool Serial_available (unsigned int uart);
- uint8_t Serial_read(unsigned int uart);
- int Serial_write (unsigned int uart, unsigned short * buffer);
- int pwm_enable (unsigned int address, int value, int deadzone);
- int pwm_disable (unsigned int address);

- bool digital_in (unsigned int digital);
- int digital_out (unsigned int digital, bool state);
- int memory_write (unsigned int memory, unsigned int address, int value);
- int memory_read (unsigned int memory, unsigned int address, unsigned short * buffer);
- int SPI_Read (unsigned int spi, unsigned short buffer);
- int SPI_Write (unsigned int spi, int data);

To simplify the development environment for the user the configuration tool only provides the APIs necessaries to interface with the peripherals chosen.

5.4 Work Flow

As mentioned early the code has to be synthetized in Xilinx ISE, although is not the most difficult program to work, for someone who don't have knowledge in digital design becomes confuse and tricky. A script was made with the purpose of automate the whole process. From the execution of the configuration tool, to move the parameter file to the correct directory, initiate the ISE and generate the bitstream file to place the finish result in a folder inside the program directory with the bitstream, APIs and include file, all of these is automatic using the script.

After that, the user only needs to move the folder to the BeagleBone, use the logi_loader to program the FPGA with the bitstream file provided and start writing his code using the APIs and the *include.h* file to interface with the peripherals.

5.4.1 Complete project

1. Write in the file blocks.txt the peripherals wanted using the language above, figure 5.6.

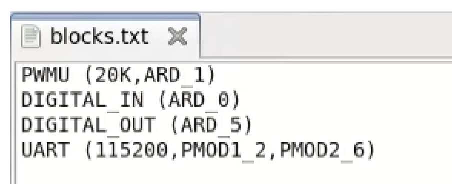


Figure 5.6: Block.txt file with the peripherals written

2. Run the ./start script and wait for it to finish, figure 5.7.

```
##########################################################################
Parameter.vh file generate with sucess
Generate 1 PWM Uniderecitional
Generate 0 PWM H-Bridge
Generate 1 Uart blocks
Generate 0 SPI blocks
Generate 1 digital input ports
Generate 1 digital output ports
Generate 0 Servo Control blocks
Generate 0 Servo Receiver blocks
Generate 0 custom register blocks
Generate 0 custom memory blocks

##########################################################################
A total of 61 lines read from file
A total of 11 comentary lines read from file
```

Figure 5.7: ./start script running in terminal

3. The bitstream and the APIs and include file appear in the Finish Project folder, figure 5.8.

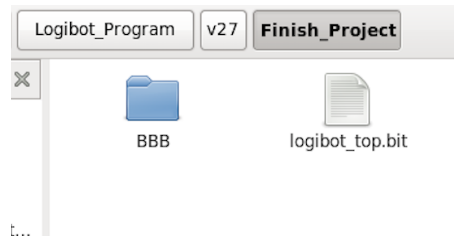


Figure 5.8: End result of the script

4. Move it to the BeagleBone Black and program the BeagleBone using the logi_loader, figure 5.9.

```
ubuntu@arm:~/Finish_Project$ ls
BBB  logibot_top.bit
ubuntu@arm:~/Finish_Project$
```

Figure 5.9: Files in the BeagleBone Black

5. Write the code using the APIs and execute, figure 5.10.

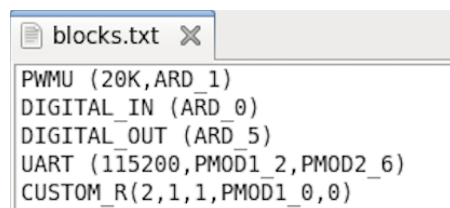
```
#include "include.h"
#include "function.h"
#include <iostream>
#include <stdio.h>
#include <unistd.h>


using namespace std;

int main() {
	int i;
	logi_open();
	while (1) {
		digital_out(OUTPUT0, digital_in(INPUT0));
		pwm_enable(PWMU0, i, 0);
		usleep(10000);
		i++;
		if(i > 100) {
			i= 0;
		}
	}
	logi_close();
	return 0;
}
```

Figure 5.10: Application for running in the BeagleBone Black

5.4.2 Complete Project with custom blocks

For the small amount of users with digital design skills, other script is available. This allows users to enter Xilinx ISE program to add their own custom blocks and have access to all definitions.

1. Write in the file blocks.txt the peripherals as well as the custom blocks, figure 5.11.



```
blocks.txt
PWMU (20K,ARD_1)
DIGITAL_IN (ARD_0)
DIGITAL_OUT (ARD_5)
UART (115200,PMOD1_2,PMOD2_6)
CUSTOM_R(2,1,1,PMOD1_0,0)
```

Figure 5.11: Blocks.txt file, with custom block

2. Run the ./start_custom script and wait for it to open ISE program, figure 5.12.

```
rm *.o main
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_u.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/digital_ports.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/pwm_h.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/memory.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/uart.cpp
g++ -Wall -Werror -fpic -c main_source/CPP/spi.cpp
g++ -Wall -Werror -fpic -c main.cpp
g++ -Wall -Werror -fpic pwm_u.o digital_ports.o memory.o pwm_h.o uart.o spi.o main.o -o
main



 #######################################################################
 Parameter.vh file generate with sucess
 Generate 1 PWM Unidirecitional
 Generate 0 PWM H-Bridge
 Generate 1 Uart blocks
 Generate 0 SPI blocks
 Generate 1 digital input ports
 Generate 1 digital output ports
 Generate 0 Servo Control blocks
 Generate 0 Servo Receiver blocks
 Generate 1 custom register blocks
 Generate 0 custom memory blocks


 #######################################################################
 A total of 61 lines read from file
 A total of 11 comentary lines read from file

[Filipe@vlsilabb v27]$
```

Figure 5.12: ./start_custom script running in terminal

3. Open the logibot_top.v and past the custom blocks declaration inside after the ‘ifdef CUSTOM_R or ‘ifdef ‘CUSTOM_M as mention in chapter 4. Give the file source for the code presented in these blocks, figure 5.13.

```
`ifdef CUSTOM_R


custom_r        custom_1 (
                    .clock              (clock100M),
                    .reset              (reset),
                    .register_input     (W_RO[CUSTOM0_RO_OFFSET0]),
                    .register_input1    (W_RO[CUSTOM0_RO_OFFSET1]),
                    .register_output      (RI[CUSTOM0_RI_OFFSET0]),
                    .register_output1     (RI[CUSTOM0_RI_OFFSET1]),
                    .port_input         (in[CUSTOM0_IN_OFFSET0]),
                    .port_output        (out[CUSTOM0_OUT_OFFSET0]),
                    .port_output1       (out[CUSTOM0_OUT_OFFSET1])
);

`endif



`ifdef CUSTOM_M
```

Figure 5.13: Adding the template to the logibot_top.v file

4. Execute the generation programming file to generate the bitstream file and wait it to finish, figure 5.14.

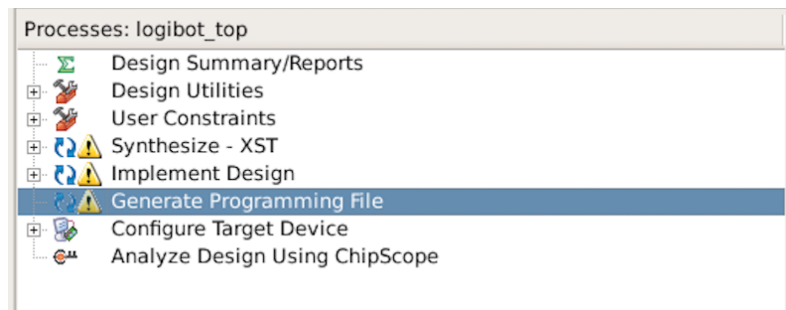


Figure 5.14: Generating the programming file

5. Go to main_folder/Project/Logi_bot and find the logibot_top.bit file. Move it to the Finish Project folder and execute the steps 4 and 5 mention in the example above.

5.5 Conclusion

The software was made with the intended of simplify the interaction of LogiBot with the user and hide all the digital design part of it from him. In this way, the final product is software capable of generate the correct parameters and directives using a simple and easy language and a script capable of hiding all the hardware design tools and configurations.

Learning from the mistakes of Skeleton from Valentf(x), this tool provides a simple interface, intuitive, offline and that don't require the contact with hardware development or hardware tools and at the same time gives a option to develop custom blocks and hardware for people that have the required knowledge of digital design.

The design of both software and hardware also makes it simpler of adding more blocks and functionalities for someone who is willing to edit the project and increment it.

Chapter 6

Conclusion

In light of the work presented in this dissertation, the following conclusions can be draw:

1. A reconfigurable design was successfully developed
2. The design is fully parameterizable, and can be generated by using simple parameters and compiler directives
3. The integration of peripherals in the design is straightforward making it easily updated
4. The capability of integrate custom blocks makes the project more customizable and takes it to another level of personalization
5. The language developed to configure the project is simple and easy to use allowing users without hardware design knowledge take advantage of the reconfigurability of a FPGA
6. The script automates the whole project including bitstream generation making it simpler to use
7. The API tools developed as well as the include.h file provided makes the interaction with the peripherals easy and elegant.
8. The hardware project can be used in another FPGA with simple changes.

6.1 Future work

Having develop the foundation for a reconfigurable peripheral manager, a number of improvements and adding's can be done to the project:

1. Evaluate the possibility of making the communication between the BeagleBone Black and the Logi-Bone faster by compressing the GPMC signals timings and increasing the FPGA clock.
2. Develop a board with a FPGA but without the SDRAM memory of the Logi-Bone, making more ports available to be used

3. Develop more peripherals
4. Make a graphical interface for the software with a list of the peripherals
5. Evaluate the possibility of personalise custom blocks to have another name and be passed from project to project
6. Develop an easy way to configure peripherals in the software capable of being used by everyone
7. Prepare the software tool to deal with more input errors

Bibliography

[1] S. Hauck and A. DeHon, *Reconfigurable computing - The Theory and Practice of FPGA-Based Computation*. Elsevier, 2008.

[2] A. Cataldo, "Xilinx enhances FPGAs with embedded PowerPCs," 2002. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1200084 [Accessed: 2015-02-01]

[3] IBM Microelectronics, "PowerPC 440 Embedded Core," 2009. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core [Accessed: 2015-02-01]

[4] ——, "PowerPC," 2006. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC [Accessed: 2015-02-01]

[5] ——, "PowerPC 405 Embedded Cores," 2006. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores [Accessed: 2002-05-20]

[6] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," 2011. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf [Accessed: 2015-02-01]

[7] I. Scouras, "Virtex-4 FX60 FPGA 'first' to feature multi-gigabit serial transceiver | EE Times," 2005. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1296499 [Accessed: 2015-02-01]

[8] Xilinx, "Virtex-4 Family Overview," 2010. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf [Accessed: 2015-02-01]

[9] C. Holland, "Xilinx completes Virtex-5 line-up, adds-in PowerPC 440 processors," *EEtimes*, 2008. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1275235

[10] Xilinx, "Virtex-5 Family Overview," 2009. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf [Accessed: 2015-02-01]

[11] ——, "Zynq-7000," 2014. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html [Accessed: 2014-12-01]

[12] ARM, “Cortex-A9 Processor,” 2011. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php [Accessed: 2014-12-01]

[13] Xilinx, “Artix-7.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/artix-7.html [Accessed: 2015-01-01]

[14] ——, “Kintex-7.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html [Accessed: 2015-01-01]

[15] M. Santarini, “Xilinx Architects ARM-Based Processor-First, Processor-Centric Device,” *Xcell journal*, no. 71, 2010.

[16] L. Getman, “Creating the Xilinx Zynq-7000 Extensible Processing Platform,” *EETimes*, 2011.

[17] M. Santarini, “Zynq-7000 EPP sets stage for new era of innovations,” *EETimes*, 2011.

[18] R. Wilson, “Xilinx goes SoC with ARM chip,” *Electronics Weekly*, 2011.

[19] Xilinx, “Zynq-7000 All Programmable SoC Overview,” 2014. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf [Accessed: 2015-01-01]

[20] ——, “Zynq Architecture,” 2012. [Online]. Available: http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf [Accessed: 2014-12-01]

[21] C. Holland, “Altera integrates ARM processor in FPGAs,” 2011. [Online]. Available: http://www.embedded.com/electronics-news/4229353/Altera-integrates-ARM-processor-in-FPGAs [Accessed: 2015-02-01]

[22] Altera, “Cyclone V FPGAs.” [Online]. Available: http://www.altera.com/devices/fpga/cyclone-v-fpgas/cyv-index.jsp [Accessed: 2015-02-01]

[23] ——, “Arria V SoCs,” 2014. [Online]. Available: http://www.altera.com/devices/processor/soc-fpga/arria-v-soc/arria-v-soc.html [Accessed: 2014-12-01]

[24] ——, “Arria V FPGA Family Overview.” [Online]. Available: http://www.altera.com/devices/fpga/arria-fpgas/arria-v/overview/arrv-overview.html [Accessed: 2015-02-11]

[25] ——, “Cyclone V FPGA Family Overview.” [Online]. Available: http://www.altera.com/devices/fpga/cyclone-v-fpgas/overview/cyv-overview.html [Accessed: 2015-02-11]

[26] Microsemi, “SmartFusion SoC FPGAs.” [Online]. Available: http://www.microsemi.com/products/fpga-soc/soc-fpga/smartfusion [Accessed: 2015-02-11]

[27] ——, “SmartFusion2 SoC FPGAs.” [Online]. Available: http://www.microsemi.com/products/fpga-soc/soc-fpga/smartfusion2 [Accessed: 2015-02-11]

[28] M. LaPedus, "Actel rolls mixed-signal FPGA with hard ARM core," *EETimes*, 2010. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1313764

[29] D. McGrath, "Microsemi buys Actel for $430 million," *EETimes*, 2010. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1257518

[30] E. Burns, "Microsemi release M2S050T SmartFusion2 system-on-chip FPGA family," 2012. [Online]. Available: http://www.electronicspecifier.com/fpgas/m2s050t-microsemi-next-gen-smartfusion2-soc-fpga [Accessed: 2015-02-11]

[31] Microsemi, "SmartFusion Customizable System-on-Chip (cSoC)," 2012. [Online]. Available: http://www.microsemi.com/document-portal/doc_download/131347-smartfusion-product-brief [Accessed: 2015-02-01]

[32] ——, "SmartFusion2 System-on-Chip FPGAs Product Brief," 2015. [Online]. Available: http://www.microsemi.com/document-portal/doc_download/132721-smartfusion2-product-brief [Accessed: 2002-05-20]

[33] Xilinx, "PicoBlaze 8-bit Microcontroller." [Online]. Available: http://www.xilinx.com/products/intellectual-property/picoblaze.html [Accessed: 2015-02-12]

[34] ——, "MicroBlaze Soft Processor." [Online]. Available: http://www.xilinx.com/tools/microblaze.htm [Accessed: 2015-02-12]

[35] Staff, "Linux added to Xilinx softcore processor," 2007. [Online]. Available: http://www.electronicsweekly.com/news/components/programmable-logic-and-asic/linux-added-to-xilinx-softcore-processor-2007-11/ [Accessed: 2015-02-12]

[36] Lattice Semiconductor, "LatticeMico32 Processor." [Online]. Available: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx [Accessed: 2015-02-12]

[37] Altera, "Nios II Processor: The World's Most Versatile Embedded Processor." [Online]. Available: http://www.altera.com/devices/processor/nios2/ni2-index.html [Accessed: 2015-02-13]

[38] ——, "Nios Embedded Processor." [Online]. Available: http://www.altera.com/products/ip/processors/nios/nio-index.html [Accessed: 2015-02-13]

[39] ——, "Nios II Processor Cores." [Online]. Available: http://www.altera.com/devices/processor/nios2/cores/ni2-processor_cores.html [Accessed: 2015-02-13]

[40] M. Plungy and S. Gulizia, "Altera and Synopsys Collaborate to Make Nios II Processor Core Available for ASIC Designs." [Online]. Available: http://news.synopsys.com/index.php?s=20295&item=122862 [Accessed: 2015-02-13]

[41] Microsemi, "IP Module - LEON3." [Online]. Available: http://soc.microsemi.com/products/ip/search/detail.aspx?id=635 [Accessed: 2015-02-13]

[42] Armadeus, "Armadeus APF27." [Online]. Available: http://www.armadeus.com/english/products-processor_boards-apf27.html [Accessed: 2015-02-15]

[43] ——, "Armadeus APF51." [Online]. Available: http://www.armadeus.com/english/products-processor_boards-apf51.html [Accessed: 2015-02-15]

[44] Armadeus Project, "IMX51-Spartan6 interface description." [Online]. Available: http://www.armadeus.com/wiki/index.php?title=IMX51-Spartan6_interface_description [Accessed: 2015-02-15]

[45] Xilinx, "Extended Spartan-3A Family Overview." [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds706.pdf [Accessed: 2015-02-15]

[46] ——, "Spartan-6 LX FPGAs." [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/spartan-6/lx.html [Accessed: 2015-02-15]

[47] ValentF(x), "Logi Bone," 2014. [Online]. Available: http://valentfx.com/logi-bone/ [Accessed: 2014-11-01]

[48] ——, "LOGI Pi," 2014. [Online]. Available: http://valentfx.com/logi-pi/ [Accessed: 2015-02-15]

[49] Beagleboard, "BeagleBone Black," 2014. [Online]. Available: http://beagleboard.org/BLACK [Accessed: 2014-12-01]

[50] Raspberry, "Raspberry Pi 2 Model B." [Online]. Available: http://www.raspberrypi.org/products/raspberry-pi-2-model-b/ [Accessed: 2015-02-15]

[51] ValentF(x), "LOGI Bone User Manual," 2014. [Online]. Available: http://valentfx.com/wiki/index.php?title=Logi-Bone_User_Guide#BBB_GPMC_to_FPGA_Interface [Accessed: 2015-02-15]

[52] ——, "LOGI Pi User Manual," 2014. [Online]. Available: http://valentfx.com/wiki/index.php?title=LOGI_Pi_User_Manual [Accessed: 2015-02-15]

[53] Xilinx, "Xilinx Software Development Kit (SDK)." [Online]. Available: http://www.xilinx.com/tools/sdk.htm [Accessed: 2015-02-15]

[54] ——, "Vivado Design Suite Evaluation and WebPACK." [Online]. Available: http://www.xilinx.com/products/design_tools/vivado/vivado-webpack.htm [Accessed: 2015-02-17]

[55] Altera, "Qsys - Altera's System Integration Tool." [Online]. Available: http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html [Accessed: 2015-02-15]

[56] ValentF(x), “Skeleton,” 2014. [Online]. Available: http://www.valentfx.com/skeleton/ [Accessed: 2014-11-01]

[57] A. Technica, “For your robot-building needs, $45 BeagleBone Linux PC goes on sale.” [Online]. Available: http://arstechnica.com/information-technology/2013/04/for-your-robot-building-needs-the-45-beaglebone-linux-pc-goes-on-sale/ [Accessed: 2015-06-25]

[58] A. L. System, “Introduction to the BeagleBone Black Device Tree.” [Online]. Available: https://learn.adafruit.com/introduction-to-the-beaglebone-black-device-tree/overview [Accessed: 2015-06-25]

[59] M. Science, “Robot Basics: Using an H Bridge to Move Your Bot Backwards.” [Online]. Available: http://mad-science.wonderhowto.com/how-to/robot-basics-using-h-bridge-move-your-bot-backwards-0137740/ [Accessed: 2015-06-26]