MASTER OF SCIENCE IN EMBEDDED SYSTEMS DESIGN

MASTER THESIS REPORT

RESEARCH AND IMPLEMENTATION OF TIMING MEASURE AND TIMING CONTROL INSTRUCTIONS FOR RISC-V CORES

A thesis submitted in partial fulfillment for the degree of Master of Science (M.Sc.) in Embedded Systems Design at Hochschule Bremerhaven

Submitted by

Pradeep Krishnamurthy (38065)

Under the Guidance of

Prof. Dr. Ing. Kai Mueller
Dipl. Inform. Frank Poppen

March 2022

ACKNOWLEDGEMENT

I am taking this opportunity to express my deepest appreciation to everyone who provided me with the help necessary to complete the thesis. A special thanks to my supervisor Dipl.- Inform. Frank Poppen at OFFIS - Institut für Informatik, now, DLR Institute of Systems Engineering for Future Mobility, for providing me with the constant suggestion throughout the period of work. I am thankful to him for his guidance, motivation and enthusiasm and in general for being an incredible mentor. I am very thankful to Prof. Dr.-Ing. Kai Mueller at Hochschule Bremerhaven for the supervision of this master thesis.

A special thanks to my parents who have always been an immense support and inspiration for me. Lastly, I would like to thank all my friends especially Sai Shashikaa, Deepak Murthy, Deepak Ravibabu and Avinaash for always being supportive and encouraging throughout my journey.

Declaration of Authorship

I confirm that this Master’s thesis, titled "Research and Implementation of Timing Measure and Timing Control Instructions for RISC-V Cores" is my own work. All the literature and sources referred in this thesis have been documented under Bibliography. This thesis was not previously presented to another examination board and has not been published.

SIGNED:.................................... DATE:....................................

ABSTRACT

Real Time Systems are integrated with physical environment. Real Time Systems are time critical and hence they must be able to handle upper time bounds along with the correct functional behavior. The correct implementation of the functionality specified in the software is done using the Instruction Set Architecture (ISA) by the processor. But, neither the software nor ISA have a time controlling role here and if time properties have to be guaranteed designers are required to reach beneath the abstraction layers which increases design complexity and effort.

This thesis proposes a solution to bring control over time to the software by examining the Instruction Set Architecture (ISA) layer. The ISA defines the contract between software instructions and hardware implementations. But ISAs usually have timing measure properties such as cycle counter and instruction counter but do not have timing control properties imbibed in them. Hence, this work investigates the instructions extension feature offered by RISC-V platform to allow programs to specify execution time properties in software. These include the ability to specify a minimum execution time for code blocks, and the ability to detect and handle missed deadlines from code blocks that exhibit variable execution times [1].

This thesis investigates the RISC-V CPU named “VexRiscV” [2] built by Charles Papon on the library SpinalHDL written in Scala language where the Custom Instructions are added into the RISC-V ISA through the instruction extension featured allowed by the RISC-V platform. This is implemented and tested on an Arty A7 – 100 FPGA with the help of Xilinx tools.

Contents

LIST OF FIGURES

LIST OF TABLES

Acronyms

1. Introduction

This chapter includes the motivation for choosing this particular topic with a brief explanation about the problem statement followed by the objective and organization of the thesis.

Real Time Systems are integrated with the physical environment. Real Time Systems are subject to high criticality in regard with time, meaning the time at which the result is delivered is as important as the result itself. For example, an autonomous driving system has to exactly calculate the distance between two cars and respond the corrective result in this case acceleration or speed in a specified time. Failing to do this will have a catastrophic effect. The timing compliance on the target processor is usually implemented using programmable timers and Interrupt Service Routines (ISR). But, this is a disadvantage in resource constrained embedded systems, especially if the software part for checking the timing characteristics is very large and occupies a lot of computing memory or time. If this application is a safety relevant system that depends on the correct operation of timing and control infrastructure at all times, it results in a safety problem and a software error here as a catastrophic effect on the system safety. For this reason, an architecture that can guarantee the timing properties is required and RISC-V architecture with its custom instruction extension feature allows us to determine the timing properties which can be taken up to the software level to guarantee the timing properties. Hence, this thesis investigates to what extent it is technically justifiable to implement the necessary time measurement and time control instruction for checking execution time intervals or waiting a certain period of time as part of an ISA extension provided by the RISC-V architecture.

The correct implementation of the functionality specified in the software is done using the Instruction Set Architecture (ISA) by the processor. But, neither the software nor the ISA usually have a timing control or timing measure role to deliver the result in a specified time. In fact, if the timing properties are to be guaranteed, to avoid delay in computations, designers must reach beneath the abstraction layers which makes the system complex and overdesigned [1]. Processors such as those based on the RISC-V ISA often already implement hardware structures for time measurement (cycle counter, instruction counter) and in a certain sense also for influencing execution times (interrupts from timer counter, watchdog). In the work from [3] , a combined timing specification and concept for timing annotation and control blocks in C++ was proposed, based solely on software with the existing limited hardware structures of an ARM processor. However, in this way, the timing and timing control algorithms themselves become a part of the software to be monitored and thus influence and distort the system to be monitored. In the context of this work it is analyzed which hardware and instruction set extensions are particularly suitable, in order to minimize undesired influences on the software system and to increase the measuring as well as control accuracy. The RISC-V ISA standard offers the possibility to extend the instruction set architecture with user-specified instructions. For this purpose, the Scala-based project VexRiscV [2] offers via its plugin architecture an easy to use possibility to extend a RISC-V SoC called Murax and to execute it on an FPGA.

In this thesis topic, the primary objective is to create a plugin in VexRiscV which is based on SpinalHDL library written in Scala Language, to add the custom instruction to the RISC-V architecture. A Verilog file is generated which contains the plugin and overall VexRiscV CPU. This Verilog file when deployed through Xilinx Vivado IDE programs the FPGA ARTY A7 – 100. A software program written in C Language uses the newly implemented user specified extended Instruction Set Architecture to measure and control time.

The thesis can be roughly categorized into three tasks. The first task identifies a concept on how the time measure and time control instructions can be implemented. Next, the defined instructions are implemented and tested inside the plugin as well as at the software level. The final step, was to evaluate and compare between the hardware implementation and software implementation and finally a solution proposal to integrate the hardware instructions with the already existing libbla library in C++ [3].

The formal structure of the thesis is as follows : The Technical Background chapter gives an overview of the technical terms, platform and languages used. The Related Work chapter, gives an overview about the research papers and reports referred and how these concepts are used in the thesis. The Concept chapter gives an idea about the initial concept used for implementation in this thesis. The Implementation chapter is about the implementation and testing of the timing measure and timing control instructions in the VexRiscV CPU which is based on RISC-V architecture. The Evaluation and Comparison chapter discusses the implementation between the hardware instructions and software implementation - advantages and disadvantages if the same concept was implemented purely in software. The Conclusion and Future Work chapter in this thesis concludes, gives a summary of the obtained outcomes and an outlook about possible future extensions of this work.

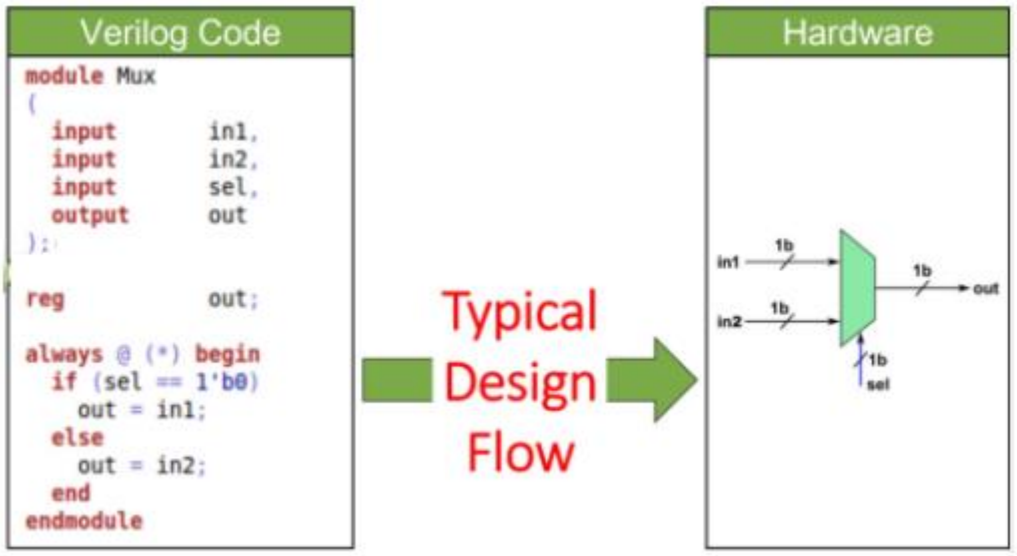
2. Technical Background

This chapter includes few technical terms explained in detail such as Instruction Set Architecture (ISA), Reduced Instruction Set Computer (RISC), RISC-V Architecture and its CPU VexRiscV and SoC Murax, Spinal HDL library. These technical terms are introduced in order to understand the technicalities behind them such as their features and their typical uses and for a clear understanding when these terms are used in the upcoming chapters in this thesis. Also a brief explanation of [3] is given in this chapter, for a clear understanding of the timing control blocks in C++ LIBBLA library, so that a better evaluation between the hardware implementation and software implementation can be done. To start with, explanation of two important commonly used terms in this thesis report are required, which are Hardware Description Language (HDL) and Field Programmable Gate Array (FPGA) because this work uses a software library called SpinalHDL which makes the program written in Scala Language to be used as HDL and a FPGA to experiment the code written in HDL.

2.1 Hardware Description Language

Interconnected transistors make up the digital circuits. A hierarchical structure to design and analyze these circuits is used. In theory, a central processing unit (CPU) could be interpreted as a vast sea of transistors, but it's much easier to organize transistors into logic gates, logic gates into adders or registers or timing modules, registers into memory banks, and so on. A developer can efficiently illustrate a digital circuit using interconnected diagrams thanks to this hierarchical structure. This is referred to as a schematic. This visual representation of a digital circuit is intuitive, but it becomes impractical as the circuit becomes more sophisticated. Another technique to describe digital circuits is to utilize a textual language that is specifically designed to convey the defining aspects of digital design in a clear and simple manner. As seen in Figure 1, on the left side, it is seen that a Verilog code which is more like a textual language is written that defines a hardware as seen on the right side of the Figure 1. Such languages exist, and they are called hardware description languages (HDLs).

Verilog and VHDL are the most widely used hardware description languages. They are frequently used in conjunction with FPGAs, which are digital devices meant to make it easier to build customized digital circuits. Hardware description languages allow you to describe a circuit using words and symbols, which development tools may subsequently turn into configuration data that is loaded into the FPGA to achieve the necessary functionality.



```
module Mux
(
  input       in1,
  input       in2,
  input       sel,
  output      out
);

reg           out;

always @ (*) begin
  if (sel == 1'b0)
    out = in1;
  else
    out = in2;
  end
endmodule
```

Figure 1 : Verilog Code to Hardware Conversion [4]

2.2 Field Programmable Gate Array

FPGAs (Field Programmable Gate Arrays) are semiconductor devices that consist of a matrix of customizable logic blocks (CLBs) linked by programmable interconnects [5]. After production, FPGAs can be reprogrammed to meet specific application or feature needs. FPGAs are distinguished by this feature from Application Specific Integrated Circuits (ASICs), which are custom-made for specific design requirements.

To begin, a developer must create the circuit's architecture. The prototype is then built and tested using an FPGA. Errors can be corrected. Once the prototype has proven to be functional, an ASIC project based on the FPGA design is produced and fabricated. Because manufacturing an integrated circuit is a complex and time-consuming procedure, this helps a developer to save time. It also saves money because a single FPGA can handle multiple versions of the same project.

As seen in Figure 2, FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects allowing blocks to be wired together. Logic blocks can be configured to perform complex combinational functions, or act as simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many FPGAs can be reprogrammed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software [6].

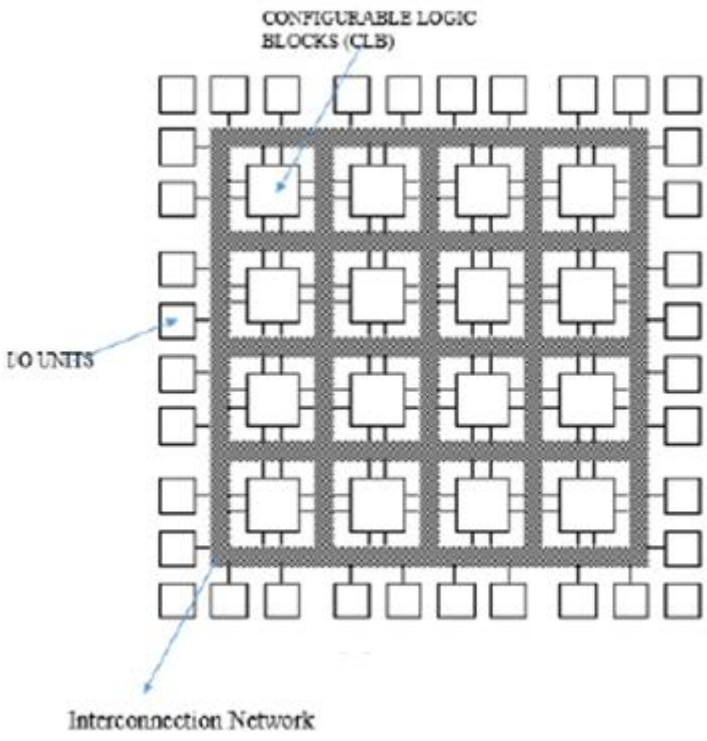


Figure 2 : General Structure of FPGA [6]

2.3 Instruction Set Architecture

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software [7]. It is important to understand ISA because understanding what the instruction set can do and how the compiler makes use of those instructions can help developers write more efficient code. It can also help developers understand the output of the compiler which can be useful for debugging.

As seen in Figure 3, the ISA serves as a bridge between hardware and software, defining what the processor is capable of doing as well as how it does it. The ISA is the only means by which a user can interact with the hardware. It can be thought of as a programmer's manual because it is the part of the machine that the assembly language programmer, compiler writer, and application programmer can see and interact with the hardware.

The ISA defines the supported data types, the registers, how the hardware manages main memory, which instructions a microprocessor can execute, and the input/output model of multiple ISA implementations. A developer can extend the ISA by adding instructions or other capabilities, or by adding support for larger addresses and data values [7].

The majority of the base ISA computational instructions have a three-operand syntax, with the destination register as the first operand, a source register as the second operand, and either a second source register or an immediate value as the third operand. An example of a three-operand instruction is : add x1, x2, x3. This instruction stores the result in register x1 by adding register x2 and x3. Many instructions take on extra functions that are performed by dedicated instructions in other CPU architectures to avoid introducing instructions that are not strictly necessary.

The instructions are given to the computer's hardware in a language that the machine understands. The words that make up the computer language are called instructions, and the vocabulary is referred to as an instruction set. The function of each instruction and how it is stored in memory are described by instruction sets (encoding). The phrase architecture refers to a processor's functional specifications. It specifies what capabilities the software can expect from the hardware. The architecture of a processor does not reveal how it is constructed. It describes the capabilities of a CPU.

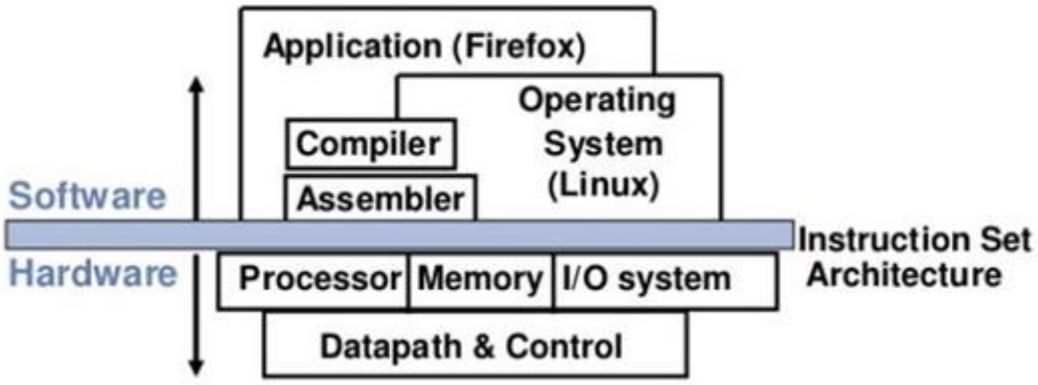


Figure 3 : Instruction Set Architecture (ISA) [8]

2.4 Reduced Instruction Set Computer (RISC)

A Reduced Instruction Set Computer (RISC) is a computer that simplifies the individual instructions supplied to it in order for it to complete a task. Because the individual instructions are written in simpler code on a RISC computer, a task may require more instructions (code) to complete. The goal is to reduce the need to process more instructions by increasing the speed of each one. With simpler instructions, implementing an instruction pipeline may be easier [9].

Higher microprocessor efficiency is critical for devices that run on batteries because it takes more electrical power to run a more demanding processor. Smartphones, laptops, and tablets are among the devices that benefit the most from RISC CPUs. Any electronic device, such as cellphones, servers, or professional desktop workstations, is powered by a microprocessor. A device's functions are carried out by the central processing unit (CPU), which performs rapid repetitions of mathematical operations. The speed of a processor is measured in gigahertz, abbreviated GHz, because modern processors perform operations in billions of cycles per second. Because RISC design only does one action per cycle, each processing cycle has less work to do. The total execution time of the system can be lowered by keeping to one cycle per instruction, resulting in faster and more efficient overall performance. The set length of instructions each cycle makes it easier for the system to pipeline data to and from the processor, as well as providing additional benefits such as faster data transfer to and from computer memory.

2.5 RISC-V Architecture

Every processor has an ISA, which is almost all proprietary and some of which can be licensed. Microchip company makes devices with 8-bit and 16-bit Peripheral Interface Controller CPUs, and there is an ISA to define them. These are Microchip company’s proprietary CPUs, which the company sells in their microcontrollers. These proprietary CPUs are licensable as intellectual property (IP). The company demands a price to use them because the firms behind them perform all the work to translate the ISA into a good processor design, develop tools to support them, and build other relevant infrastructure. When these solutions don't quite accomplish the desire the developer wants, the developers or the companies buying the ISA are faced with a problem.

For example, a new application may be required to do a single operation, such as encryption, extremely quickly while drawing very little power. The work could be completed in 100 instructions by a future licensable processor IP. If the power usage needs to be cut down, there will be a need to find a low-power silicon fabrication facility. That could be more expensive than a general-purpose fabrication procedure, making the product out of reach for the intended market. However, some bright engineers could speed up the execution of the code by writing additional instructions for the processor, but one can't change the ISA because it's proprietary. So, the engineers are stuck with a processor performance issue that requires a manufacturing solution. This is where RISC-V ISA comes into picture as a one stop solution to address all the above mentioned problems.

RISC-V (pronounced as risk five) is an open Instruction set Architecture (ISA) that began in 2010 as a summer project at the University of California, Berkeley. Unlike many other ISA designs, RISC-V is provided under open source licenses that do not require fees to use [10]. The name RISC-V was selected to name the fifth major RISC ISA design from UC Berkeley. The roman numeral “V” signifies “variations” and “vectors” to support a range of computer architecture research.

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISAs are very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings [11].

In the following paragraph some features of the RISC-V Architecture will be discussed. The RISC-V architecture is a load-store architecture, which means three things: Only load and store instructions move data to and from memory, and data must first be loaded into a register before it can be operated on. RISC-V is suited for all computing uses since it is not over-architectured or over-optimized for any particular implementation, micro-architectural pattern, or deployment target. Because its ISA is divided into two sections, the base ISA and optional extensions, it is possible to accomplish this. The base ISA is limited to a small number of instructions that are sufficient to build a compiler target and satisfy modern operating systems. Depending on the implementation, more ISA extensions can be added to the underlying ISA. As a result, the RISC-V can accommodate a lot of customization and specialization. It is the smallest ISA for 32-bit and 64-bit addresses, and the memory system uses little-endian byte ordering. The least important byte of multi-byte data is placed at the lowest memory location in little-endian byte ordering. RISC-V employs the RVC (RISC-V code compression) technique to minimize program code size while also reducing CPU cycles per instruction at the expense of increasing the number of instructions per program. To simplify the implementation circuits, it compromises code density. RISC results in large code sizes that are not optimum, particularly for embedded systems because they have a limited instruction memory capacities. The RVC extension of RISC-V is used to minimize code size. RVC uses 16-bit instruction encodings instead of 32-bit instruction encodings. There is also no branch delay slot. There are 47 instructions in the base RISC-V instruction set. System calls and performance counters are performed by eight system instructions. Computational instructions, control flow instructions, and memory access instructions make up the remaining 39 instructions.

Unlike normal ISAs as explained in section 2.3, a RISC-V addition instruction on the other hand adds a source register and an immediate value of zero, then stores the result in a destination register, yielding the same result, add x1, x2, 0, transferring the value (x2 + 0) to x1 is the command to transfer register x2 to x1.

The RISC-V architecture has a very limited instruction set, omitting various types of instructions found in other CPU architecture instruction sets. Many of the more recognizable instruction functions may be performed with RISC-V instructions, though perhaps not in the same straightforward way. The RISC-V assembler offers a variety of pseudo-instructions as depicted in Table 1, each of which maps to one or more RISC-V instructions that provide functionality similar to that found in a general-purpose processor instruction set. The RISC-V assembly language includes a number of pseudo-instructions that use vocabulary that assembly language programmers are more likely to recognize. As seen in Table 1, the assembler converts the mv rd, rs pseudo-instruction to a literal addi rd, rs, 0, which is a command to copy content from source register to destination register.

Pseudo Instruction	Base Instruction(s)	Meaning
nop	addi x0, x0, 0	No operation
li rd, immediate	Myriad sequences	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One’s complement
neg rd, rs	sub rd, x0, rs	Two’s complement
negw rd, rs	subw rd, x0, rs	Two’s complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if ≠zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero

Table 1 : Few RISC-V pseudo instructions [11]

2.5.1.1 RISC-V ISA Extension

RISC-V architecture goal is to provide a foundation for more specialized instruction-set extensions or customized accelerators, in addition to supporting regular general-purpose software development. RISC-V was created to allow for a great deal of customization and specialization. Each base integer ISA can be extended with one or more optional instruction-set extensions, and the RISC-V instruction-set encoding space is divided into three distinct categories: standard, reserved, and custom. The RISC-V Foundation defines standard encodings, which must not conflict with other standard extensions for the same underlying

ISA. Reserved encodings are not defined at this time, but they will be saved for future standard additions. Custom encodings are only available for vendor-specific non-standard extensions and should never be used for standard extensions.

RISC-V processor implementors can choose which extensions to incorporate in a processor design to optimize tradeoffs between chip size, system capabilities, and performance. This is a significant advantage, enabling one to reuse the community software stack and accelerate only the crucial parts.

2.6 VexRiscV

According to [12], there are about 111 RISC-V CPUs currently available, out of which VexRiscV is one among the RISC-V CPUs written in an RTL language named SpinalHDL, a hardware description library in Scala (see section 2.7) and licensed under Massachusetts Institute of Technology. VexRiscV is built upon the 32 bit Instruction Set Architecture of the RISC-V.

2.6.1 VexRiscV CPU

[2] defines some of the important features of VexRiscV as follows : VexRiscV is a RV32IM instruction set which is pipelined on 5 stages (Fetch, Decode, Execute, Memory, WriteBack) as seen in Figure 4. The performance of VexRiscV is 1.44 DMIPS/MHz when all features are enabled. It is optimized for FPGA. It has an optional debug extension allowing eclipse debugging via an GDB - OpenOCD - JTAG connection. It has an optional interrupts and exception handling capability.

VexRiscV CPU's hardware description is done in a very software-oriented manner (without any overhead in the generated hardware). There are relatively few things that are unchangeable. Plugins are used for almost anything. The PC manager, the register file, the hazard controller, and so on are all plugins. There is an automatic tool that allows plugins to enter data into the pipeline at a specific stage and read it at a later stage using automatic pipelining. A service system exists that provides a very dynamic framework. For example, a plugin might provide an exception service that other plugins can utilize to emit exceptions from the pipeline. There is no doubt that the VexRiscV won the first prize of the RISC-V Summit Softcore contest [13], with these features.

VexRiscV is designed to be very modular. All complimentary and optional components are built as plugins, allowing them to be readily merged and customized into various processor settings. Memory, caches, Input/output peripherals, and buses are all available in the VexRiscV ecosystem and can be combined as needed.

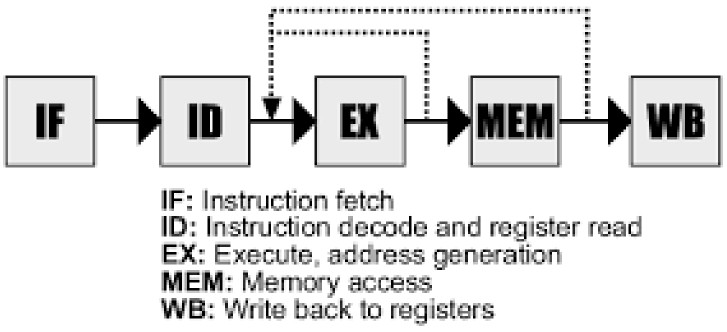

IF
ID
EX
MEM
WB
IF: Instruction fetch
ID: Instruction decode and register read
EX: Execute, address generation
MEM: Memory access
WB: Write back to registers

Figure 4 : VexRiscV - 5 stage pipeline [14]

2.6.2 VexRiscV Architecture

What makes the VexRiscV unique is that it is made up of a huge number of plugins that split up the design horizontally per feature rather than the traditional vertical pipeline-stage oriented method. It allows a developer to implement all parts of a new instruction, for example, in a single file rather than multiple files.

VexRiscV is based on a five-stage in-order pipeline, on which a variety of optional and complementary plugins add functionality to create a working RISC-V CPU. This method is absolutely novel and only possible with meta hardware description languages (in this case, SpinalHDL which is explained in section 2.7), but it has its benefits in the VexRiscV implementation - The plugin system allows a developer to swap/turn on/off portions of the CPU directly. A developer can add new features/instructions to the CPU without changing any of its sources. It enables the CPU setup to span a wide range of implementations. It enables the codebase to generate a fully parametrized CPU architecture. However, all of the above functions cannot be performed during runtime of the CPU. Each function or feature addition requires a generation of a new CPU and should be synthesized and programmed into FPGA.

The CPU contains only the specification of the 5 pipeline stages and their basic arbitration if it is generated without any plugins, and everything else, including the program counter, is added to the CPU via plugins.

2.6.3 Murax SoC

The VexRiscV ecosystem also offers a whole preconfigured processor setup called Murax SoC, which is compact and basic in design and focuses on low resource consumption. As seen in Figure 5, the VexRiscV CPU is combined with an on-chip instruction and data memory, an Advanced Peripheral Bus (APB), a JTAG programming interface, and a UART interface in the Murax SoC. Murax SoC just uses a small amount of resources and can run without any further external components.

Murax is a very light SoC which can work without any external components: It is built with VexRiscV RV32 architecture with JTAG debugger (Eclipse/GDB/OpenOCD). The Murax SoC has 8 kB of on-chip ram which can be extended as it is a softcore SoC. It also comes with an interrupt support and an APB bus for peripherals. Murax SoC has 32 GPIO pins with one 16 bits prescaler, two 16 bits timers and one UART with Tx/Rx FIFO.

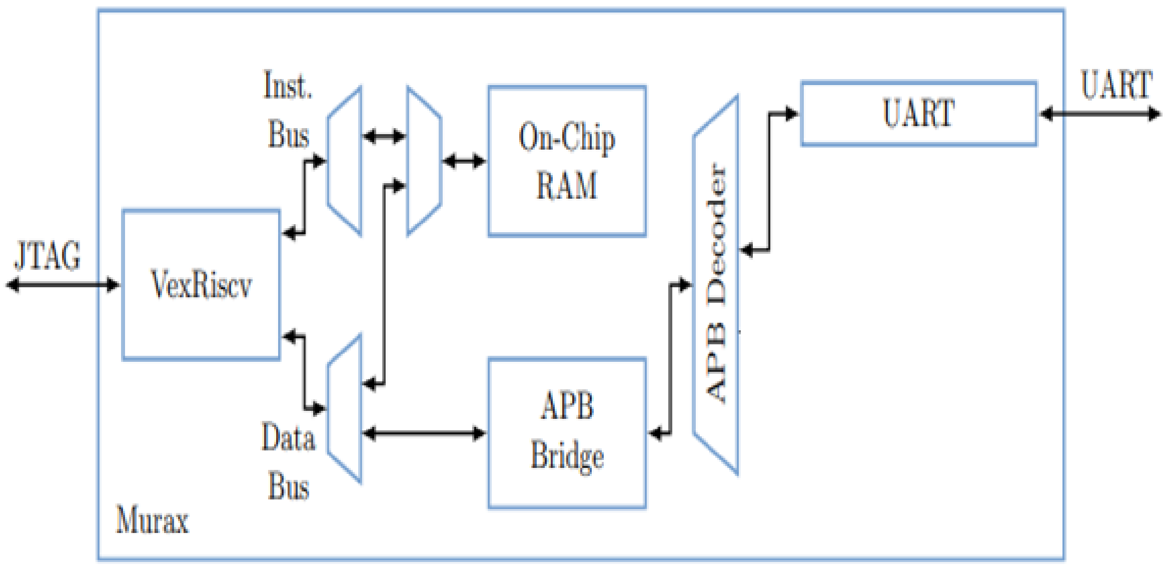


Figure 5 : Schematic of the Murax SoC [15]

2.7 Spinal HDL

SpinalHDL is a hardware description language written in Scala, a static-type functional language using the Java virtual machine (JVM). It is an open source hardware description language started in December 2014 by a person named Charles Papon, who by the way is also the creator of VexRiscV.

[16] explains some of the features of SpinalHDL - SpinalHDL is a language to describe digital hardware which is compatible with EDA tools, as it generates VHDL/Verilog files. It is much more powerful than VHDL, Verilog, and SystemVerilog in its syntax and features. SpinalHDL is much less verbose than VHDL, Verilog, and SystemVerilog. It is not an HLS, nor based on the event-driven paradigm and only generates what a developer asks it in a one-to-one way. It allows a user to use Object-Oriented Programming and Functional Programming to elaborate the hardware and verify it. It is completely free and can be used in the industry without any license.

SpinalHDL has a number of advantages over VHDL and Verilog. Because SpinalHDL is based on a high-level language, it offers a number of benefits that can help enhance hardware coding: It creates and connects complicated buses like AXI on a single line, eliminating the need for infinite wire. A user can create his own bus specification and abstraction layer to evolve the capabilities. SpinalHDL reduces the size of the code notably for wiring by a significant amount. This allows the user to gain a better understanding of the code base. Type conversions that are both powerful and simple - Any data type and bits can be translated in both directions. When loading a complex data structure from a CPU interface, this function comes in handy.

2.8 Timing control blocks in LIBBLA

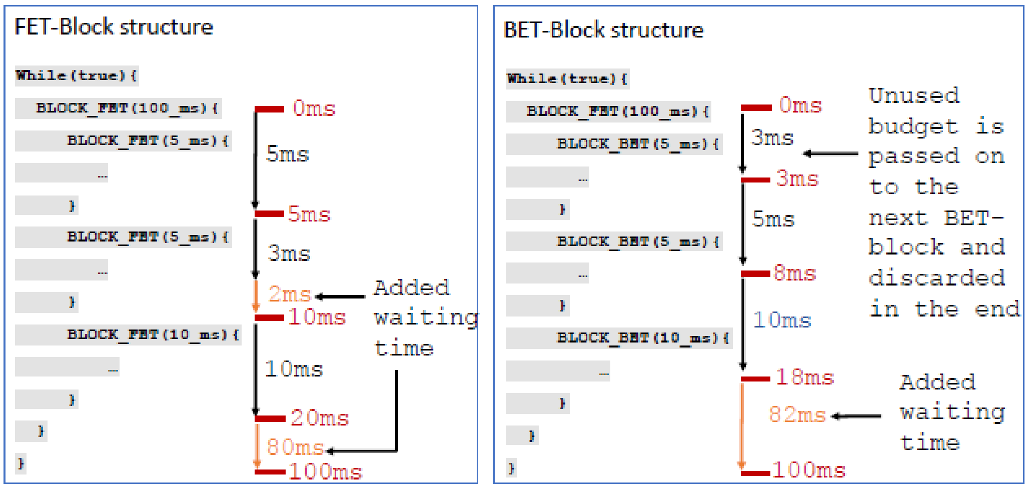
LIBBLA is a C++ extension realised as a C++ library which supports a software developer in adding block-based time annotations into bare-metal embedded C++ applications. With the timing blocks, a developer can specify an execution time limit for a user-defined block in the source code. At run-time, this block executes never longer than the specified time limit and reports a time violation (expressed by a user-defined error handling) if the execution time of the enclosed source code should exceed the given time limit. The time block concept in a C++ library was implemented and tested on an ARM Cortex A9 bare-metal platform.

[3] assumes a cyclic software execution model in which we can distinguish between the following block types:

(1) The specified block duration is kept precisely. As observed in Figure 6, after the execution of the contained code, the block waits until the block's duration has elapsed. This block is named as Forced Execution Time (FET) Block.

(2) A block is left directly after the execution of the contained source code has been finished. The remaining time until the specified end of this block is passed over to the following block as seen in Figure 6, i.e. the execution of the next block can be advanced. This block has been named as Budgeted Execution Time (BET) Block.

(3) In addition, there is another block named Estimated Execution Time (EET) Block that has been defined to support the time measurement in the profiling phase.



```
While(true){
  BLOCK_FET(100_ms){
     BLOCK_FET(5_ms){
          …
     }
     BLOCK_FET(5_ms){
          …
     }
     BLOCK_FET(10_ms){
          …
     }
  }
}
```

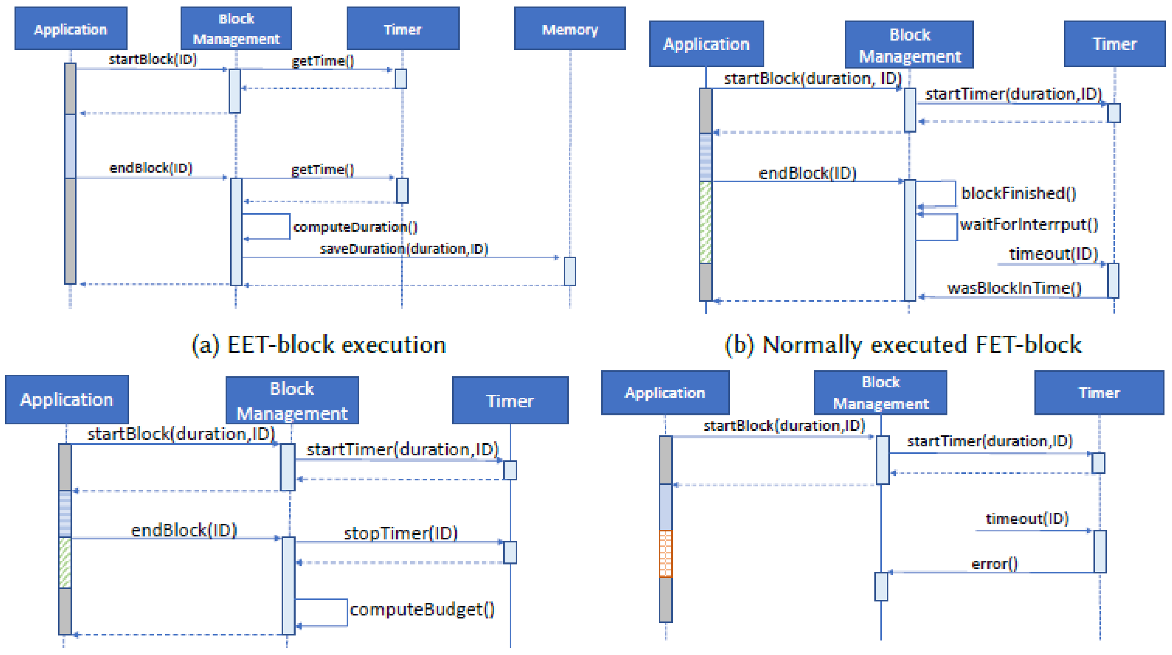
```
While(true){
  BLOCK_FET(100_ms){
     BLOCK_BET(5_ms){
          …
     }
     BLOCK_BET(5_ms){
          …
     }
     BLOCK_BET(10_ms){
          …
     }
  }
}
```

Figure 6 : FET and EET block structure [3]

Figure 7 shows the sequence diagrams for time measurement and control blocks. At run time, EET blocks save the start and end time instants and hence measure the execution time (or duration) of these blocks. The generated data is saved under the block-specific ID for each block.

To maintain strict alignment in a cyclic execution, the stated time period in FET must be kept exactly. If necessary, a specified waiting time (delay) is introduced before the following block is executed, as shown in Figure 7. If the execution time exceeds the time provided in the FETs, the user-defined error handling is used. FET-blocks can be nested to provide a new context in which the time behaviour can be specified without affecting the overlaying program structure's time behaviour.

When a BET-block finishes earlier than expected, the budget's leftover time is passed to an adjacent BET-block. A direct parent FET-block is required for each BET-block to control its time budget. As a result, BETs are always a refinement of a FET-block. Because FET-blocks have set execution periods, they are not permitted to consume the underlying BET-blocks remaining time limit. If a BET does not use its entire budget, it is passed on to the next BET-block, which can then begin earlier and possibly spend the extra time.



(a) EET-block execution (b) Normally executed FET-block

(c) Normally executed BET-block (d) Control blocks in case of exceeded deadline

Figure 7 : Sequence Diagram of time control blocks [3]

3. Related Work

This chapter gives an overview of the state of the art research papers referred for the implementation of deadline instructions and to differentiate the work in this thesis in relation to others.

Abstract PRET machines [17] implement an ISA with a deterministic model of temporal behaviour that permits delivering a deterministic timing without loss of performance. The focus lies on designing micro architectures where real-time problems consist of sporadic event streams with deadlines equal to periods. PRET machines examine the changes required at the lower levels of abstraction in order to support temporal isolation and effective software synthesis. PRET machines can provide guarantees concerning timing predictably and determinism wherein the approach in this thesis is also similar as it implements an ISA based on RISC-V that brings control over timing to the software level.

In order to address the lack of temporal semantics in the ISA, [1] proposes instruction extensions to the ISA that give temporal meaning to the program. In addition, [1] presents Precision Timed ARM Architecture (PTARM) that provides timing predictability and composability without sacrificing performance. They extend the ISA with the addition of four instructions to the ISA : user specified time to control code blocks and allowing an exception to occur if the use specified time is not met. In this thesis, the extension of the instructions resembles very closely to the concept idea given out in [1]. However, the thesis takes into account RISC-V architecture rather than the PTARM architecture.

An extension to an Instruction Set Architecture (ISA) with temporal semantics was given in [18]. They identified four desirable capabilities of an ISA. Specifying a minimum and/or a maximum duration for the execution of a code block and allowing a conditionally branch or a branch to an exception handler if the code block exceeds a deadline. By introducing semantics at the lower layers, they ensure that the behaviour of the deployed parallel and concurrent systems match their specifications. The problem addressed by this research paper is that even when tasks that are executing concurrently do not communicate, they may interfere by affecting each other's timing. In accordance to this thesis, the conceptual idea of the extension of ISA is very much similar, however the problem addressed is not taken into consideration in this thesis.

In order to expose the timing control to software, [19] augmented the traditional SPARC ISA with deadline instructions which allow the programmer to set and access cycle accurate timers. These deadline instructions offer precise timing control by explicitly specifying the number of clock cycles that should pass before the next instruction is completed. If the code block executes before the timer has expired it will be stalled using replay mechanism else an exception is thrown and appropriate trap handler code takes over. In this thesis, as it is known RISC-V ISA is used, and the idea to stall the instruction until the timer is expired is taken into account. However, the replay mechanism is not considered to stall the instruction.

To provide precise timing control to software, [20] proposes deadline instructions that allow the programmer to set and access cycle accurate timers. They have provided two deadline timers that can be accessed by the instruction : one deadline timer counts according to the main clock, the other deadline timer counts according to a clock generated by a programmable phase-locked loop (PLL). These timers appear as registers and can be accessed only through the deadline instruction. When a deadline instruction attempts to set a deadline register, it blocks until the deadline register reaches zero, at which point it reloads the register and passes control to the next instruction. Unlike [20], in this thesis the main cycle counter is utilized directly to add the current cycle counter value along with the user input which is stored in a register and then main cycle counter value is compared against the value in the register to determine whether the instruction needs to be halted or not.

To achieve high - precision timing in software, [21] proposes an instruction level access to cycle accurate timers. They add an instruction that waits for a timer to expire then reloads it synchronously. This provides a way to exactly specify the period of a loop and they validated their approach by implementing on a simple RISC processor with the extension on an FPGA and programmed it to behave like a video controller and an asynchronous serial receiver. Their technique was placing a deadline instruction at the beginning of a block of code that ends with another deadline, therefore, guarantees that the block runs in at least the amount of time given by the argument to deadline. On the contrary in this thesis, the deadline instruction, on the other hand, gives the precise time control required for software implementation. This thesis extends on this concept of controlling software execution time by introducing a set of timing instructions that allow a developer to manage not only the minimum execution time, but also in circumstances where the execution time exceeds a predetermined deadline. In this thesis, the deadline instruction is put at the end of the block whereas the instruction that takes the sum of user input and current cycle counter value is put at the start of the block, so when the deadline instruction is reached it waits there until the specified value is reached which also guarantees that the block runs for the exact duration specified by the user.

4. Concept

The Instruction Set Architecture (ISA) serves as the contract between the software and the hardware. The programmer understands the semantics of each instruction and uses it to construct programs. Computer architects ensure that the implementation of each instruction abides by the semantics specified in the ISA [1]. In recent ISAs, the semantics of the instructions frequently do not specify the instructions temporal properties. As a result, in order to reason about a program's temporal features, we must leave the ISA semantics and delve into the architectural details. Because ISAs do not give any ways of exposing or controlling software's timing behaviour, their implementations are not obligated to have predictable and repeatable timing behaviour. This makes deciphering program temporal behaviour much more challenging. Hence, in this section, an initial concept to extend the ISA with timing properties is presented.

A simple technique to add timing properties to the ISA would be to assign a constant execution time to each instruction. However, because instruction execution time must comply to the constant time stated in the ISA, the constant time ISA prohibits performance gains at the micro-architectural level. The concept idea is to modify the ISA in such a way that a developer can control the timing behaviour of programs rather than associating a constant execution time with all instructions.

In most modern embedded platforms, manipulating external timers and setting interrupts is already possible. The technique for configuring timing interrupts, on the other hand, differs greatly depending on platform implementation. The external timer is frequently treated as if it were just another I/O component, and access to it is frequently accomplished via memory mapped registers. As a result, the program's temporal behaviour is linked to the implementation platform. However, unification of the semantics of time across all programs, implemented using the ISA by specifying timing instructions as part of the instruction set can be done. Hence, instead than being a consequence of the underlying architecture implementation, software now has control over timing. In this section, the concept of adding the new timing instruction set, which allows a developer to experiment the timing behaviour is dealt.

In order to achieve the correct functionality of timing measure and timing control instructions, four instructions are needed. Chapter 3 gives a basis for three concepts which is evaluated in detail in the next paragraph.

In concept 1 [1], the timing control instructions on an ARM ISA is implemented in this fashion: First, a timestamp of the current platform time is taken (*get_time)* and then an offset is specified by the user in the next step. This offset is the minimum execution time needed. The time stamp and the offset are added in the second step and this value is the deadline value. In the third step, after writing the block of code to be executed, an instruction which delays the execution time until the current platform time exceeds the deadline value is given (*delay_until)*. During the third step, if the platform time has already passed the deadline value (*delay_until)*, then this instruction does not do anything and the program execution continues. To overcome this, another instruction (*exception_on_expire)* is implemented to throw an exception as soon as the deadline is missed.

In concept 2 [18], the implementation of the instruction is almost similar except for the fact that the offset needed to specify the minimum execution time is done in one instruction rather than an add instruction after getting the timestamp.

In concept 3 [19], first the instruction (*deadload*) loads the deadline value into a timer register and at the end of the block of code needed to be executed , an instruction (*dead*) stalls the execution until the timer value reaches zero and then loads a new value in the timer register. In the case, if the deadline for the block is not met, the instruction (*deadbranch)* raises an exception, the functionality which the instruction (*dead*) does not possess.

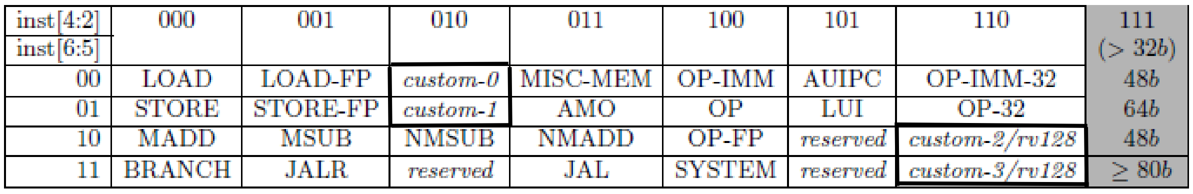
By taking into account, concept 1, concept 2 and concept 3, the idea of four instructions for this thesis is planned in RISC-V ISA as shown in Table 2. The reason behind this concept is that, in concept 1, the user needs to manually add another line of code to set an offset, which is not the case in the *deadload* instruction which is implemented in this thesis. In these three concepts, the idea of an instruction which calculates the execution time of the block of code in just one cycle or one instruction is not found, due to which the idea of the *measure* instruction is taken up. The base idea for the concept of the instructions (*dead* and *deadexcep*) in this thesis is taken from concept 1, concept 2 and concept 3, except for the way on how the time delay is achieved.

Instruction	Description
Measure	This instruction is used to measure the execution time of a program.
Deadload	This instruction obtains a value from the user and adds it with the current cycle counter value to give the deadline value.
Dead	This instruction delays the execution of the program until deadline value is reached.
Deadexcep	This instruction delays the execution of the program until deadline value is reached and throws an exception if the deadline value is missed.

Table 2 : List of Timing instructions

Table 2 shows the timing instructions and a brief description of their functionality. The concept idea given here extends the RISC-V Instruction set, so here the custom timing instructions extensions are presented in the context of RISC-V ISA. The concepts and extensions could be easily applied to other ISAs if custom extensions are allowed. The RISC-V ISA sets aside a custom instruction encoding space to allow additions to the architecture as seen in Figure 8.

Future standard extensions will avoid major opcodes labelled as Custom-0 and Custom-1 as seen in Figure 8, but they are recommended for usage by custom instruction-set extensions within the base 32-bit instruction format. Custom-2/rv128 and Custom-3/rv128 opcodes are reserved for future use by RV128, but will otherwise be avoided for standard extensions. Therefore they can also be used for custom instruction-set extensions in RV32 and RV64.



inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

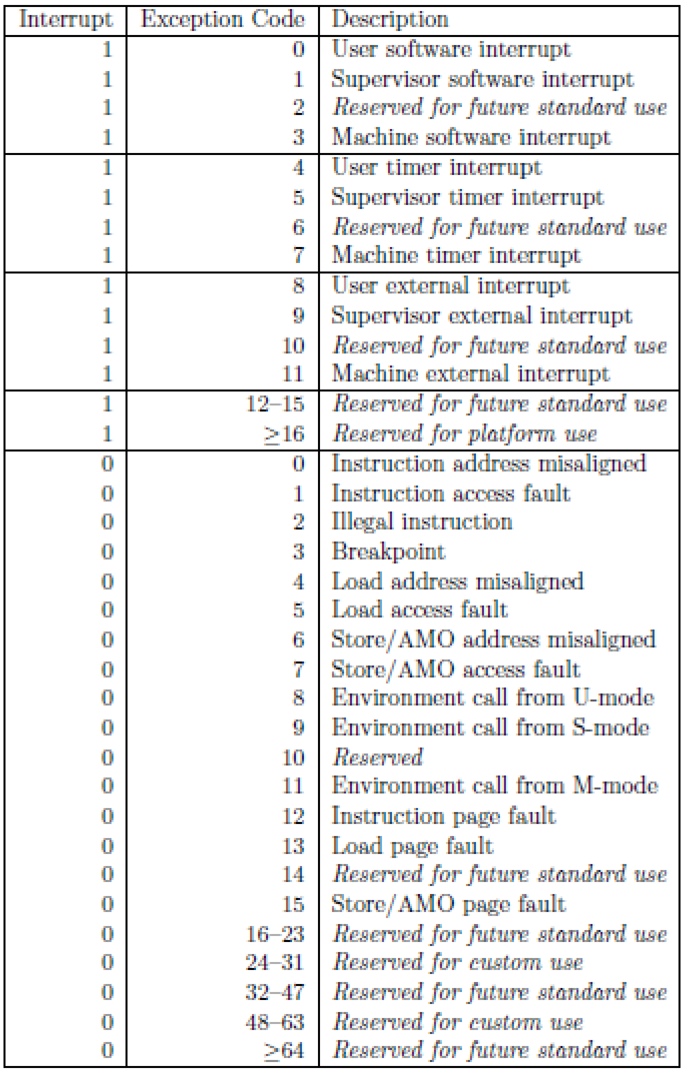
Figure 8 : RISC-V base opcode map [11]

The *measure* instruction is mainly used to obtain the execution time of any piece or block of code that we intend to measure in the program. To measure the execution time, the *measure* instruction has to be executed two times. When the instruction is executed for the first time, the current Cycle Counter value can be obtained. This value is retained in a register so that when the *measure* instruction is called for the second time, the difference between current Cycle Counter value and the previously retained value can be calculated to obtain the exact execution time. The value obtained will be in the form of prescaled values which then should be calculated according to the set resolution. The output can be observed on the dedicated UART controlled by hardware, after execution of each *measure* instruction.

The *deadload* instruction is used mainly to get the input from the user and perform an addition with the current cycle counter value. The input from the user here is the amount of time the user wants to delay a block of code. This user input value is then added to the prescaled current cycle counter value which is then stored in a register named deadline. The deadline value is used as input to other timing instructions which will be introduced below. This instruction has to be used in combination with *dead* or *deadexcep* instruction, if only *deadload* instruction is used then it will not perform any operation.

The *dead* instruction is used to delay program execution until a specified time. The *dead* instruction takes the deadline register which was computed using *deadload* instruction and compares it to current Cycle Counter value to determine whether delays are needed. So, when *dead* instruction is decoded, it will compare the deadline register value and the current cycle counter value and halts the pipeline until current cycle counter value is greater than the value in deadline register. Once the current cycle counter value is greater than the deadline register value, the code simply continues to execute the further lines of code. If current cycle counter value has already passed the deadline value, then the *dead* instruction will not do anything and the program will continue to execute. As it was explained earlier, *deadload* and *dead* instruction have to be always used together. If the *dead* instruction is used alone by itself, then the initialised value 0 for the deadline register will be taken into account and the execution continues with this initialised value.

Before going on to the concept of next instruction (*deadexcep)*, the concept of exception handling in RISC-V is to be known. In RISC-V, there are certain set of registers which help in the process of exception handling. Out of them, mtvec (Machine Trap-Vector Base-Address Register) and mcause (Machine cause register) are particularly important. The mtvec register is a read/write register that holds trap vector configuration and mcause register is a read/write register that contains the exception code indicating the event that causes the exception. Figure 9 shows the different exception codes and their descriptions. So, when an exception occurs the CPU loads the value stored in the mtvec register to a program counter and by this it jumps to the code at that address which is the exception handler. The exception handler is defined by the user in order to perform the necessary operation when an exception occurs. In this thesis, the exception codes which are reserved for custom usage, are used as seen in Figure 9, to raise a custom timer expired exception.



Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Figure 9 : Machine cause register (mcause) values after trap [22]

The *deadexcep* instruction is similar to that of *dead* instruction where in which it is used to delay program execution until a specified time and also to emit an exception if the deadline is missed. When the *deadexcep* instruction is decoded, if current cycle counter value is lesser than or equal to deadline register value, then it halts the pipeline in turn delaying the execution of the program. If the deadline is missed, meaning if current cycle counter value is greater than deadline register value, an exception is raised and the program flow goes into an exception handler. In the exception handler the required action to be performed can be defined. Similar to *dead* instruction, *deadexcep* instruction also has to be in conjunction with the *deadload* instruction.

Now, if taken a close look at the C++ LIBBLA blocks which were explained in Chapter 2 and the deadline instructions explained in this Chapter, *measure* instruction has the same functionality as EET block and *deadload* along with *deadexcep* instruction has the same functionality as the FET block but without the nesting feature. In other words, EET block measures the execution time taken by a block of code and so does the *measure* instruction which measures the execution time taken by a block of code. FET block sets a specific time for a block of code and waits until the set time has been passed or throws an error if the set time has already been passed and so does the *deadload* and *deadexcep* instruction, which assign a specified time to a block of code and wait until the deadline value is reached or else throw an exception if the deadline value is passed.

5. Implementation

With the initial concept in mind as explained in Chapter 4, the timing instructions are implemented in the VexRiscV CPU plugin architecture. In order to do this, an FPGA, Xilinx Vivado IDE and Linux platform are required. The basic idea is to write a plugin with the required specifications, for the timing instructions needed to be implemented. This implementation generates a Verilog file with the plugin, Murax SoC and the VexRiscV CPU which can be programmed on FPGA with the help of Xilinx Vivado IDE. Later, a software program will be loaded into the memory of the FPGA with the instructions to test and evaluate the specifications.

Figure 10 (personal communication, F. Poppen, [Mindmap], February 7th, 2022) shows the Overall flow of the implementation. The instructions are added to a plugin and are connected to the Murax SoC in the Murax.scala file. The next step is to obtain the Verilog, which is generated after compiling the plugin. This Verilog file is attached to the toplevel file in Xilinx Vivado IDE and with the help of IDE , a bitstream of the Verilog file is generated which is then uploaded onto the FPGA. A C program containing the instructions is written and compiled. After compiling, .elf, .asm, .hex and .v files are generated. The .elf file is loaded into the memory of the Murax SoC, by which we can see the results of the implemented concept either on UART or on the LEDs. OpenOCD - GDB helps us in debugging the software.

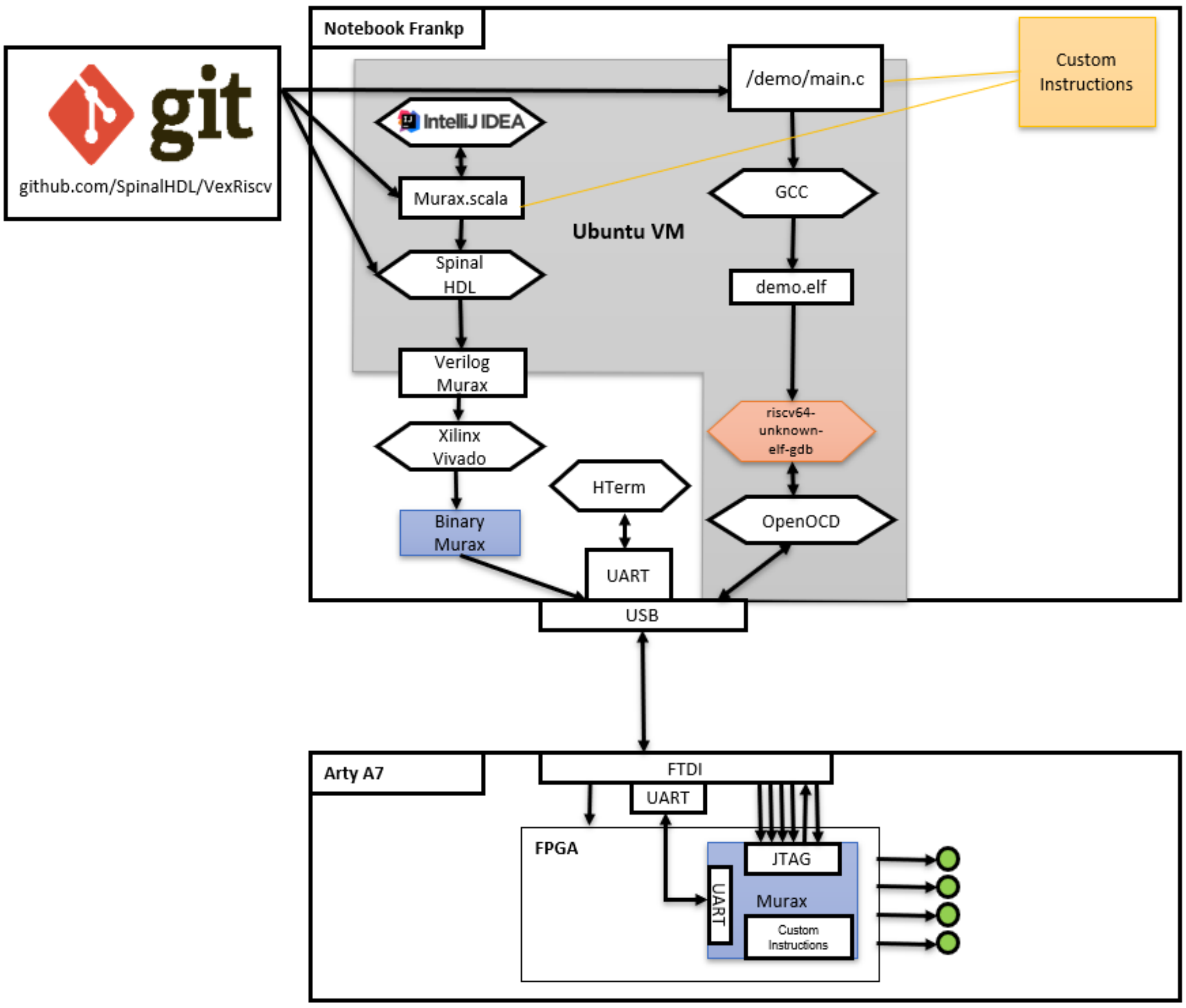


Figure 10 : Overall Flow of Implementation

The implementation of the custom extension of the ISA assumes a Cycle Counter which is running at a speed of 100 MHz or in other words has a resolution of 10 ns which is present in a plugin. This Cycle Counter is used by all the timing instructions to specify and manipulate the execution time of code blocks. This Cycle Counter is of 32 bits in size, therefore giving us approximately 43 s of time to perform the experiments before it resets to zero. This results in a shorter duration of time for the experiments to run. To overcome this issue, a prescaler is implemented (see Appendix 2), in such a way that the Cycle counter value increments when the prescaler value reaches zero. The prescaler is of 7 bits in size, thereby by giving us a resolution of approximately 1 μs.

The implementation of the instructions can be done in two ways in a plugin. The first way is to create a new plugin for each of the four custom instructions along with the already existing plugin from where the Cycle Counter values come from and connect the Cycle Counter plugin to all the other four plugins and also connect the *deadload* instruction plugin with the *dead* instruction and *deadexcep* instruction since the *deadload* instruction plugin deadline register value is required for *dead* and *deadexcep* instruction. The second way is to create one plugin with all the four instructions in it in addition to the already existing plugin from where the Cycle Counter values come from and to connect both the plugins so that all instructions can use the Cycle Counter value in the same plugin itself. The second way of implementation is taken forward from here because everything can be accommodated in one plugin itself. The pros and cons of choosing between these two ways of implementation is dealt in Chapter 6.

Table 3 shows the instructions name, their 32 bit instruction and their opcodes. All instructions have the instruction encoding as seen in Figure 11 with opcode differentiating the instruction type. In software, C language in this case, the 32 bit instructions are defined in a macro as seen in Listing 5.1 and the specific names and opcodes are assigned to each instruction, so that it becomes simpler for the user to use the instruction instead of typing the 32 bit instructions all the time.

Listing 5.1: Definition of 32 bit instruction in header file

```
asm(".set CUSTOM0  , 0x0B"); //opcode of first custom instruction
asm(".set CUSTOM1  , 0x2B"); //opcode of second custom instruction
asm(".set CUSTOM2  , 0x5B"); //opcode of third custom instruction
asm(".set CUSTOM3  , 0x7B"); //opcode of fourth custom instruction

#define opcode_R(opcode, func3, func7, rs1, rs2)
({
   register unsigned long __v;
   asm volatile(
    ".word ((" #opcode ") | (regnum_%0 << 7) | (regnum_%1 << 15) | (regnum_%2 << 20) | ((" #func3
") << 12) | ((" #func7 ") << 25));"
    : [rd] "=r" (__v)
    : "r" (rs1), "r" (rs2)
   );
   __v;
})
```

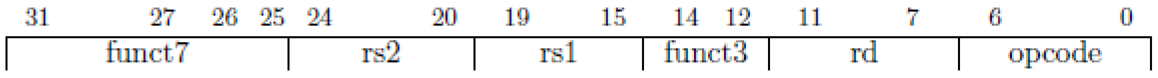


Figure 11 : Instruction encoding of RISC-V ISA [11]

Name	Instruction	Opcode [6:0]	Opcode in hexadecimal
Measure	00000000000000000000000000001011	0001011	0x0B
Deadload	000000000000-----000000000101011	0101011	0x2B
Dead	00000000000000000000000001011011	1011011	0x5B
Deadexcep	00000000000000000000000001111011	1111011	0x7B

Table 3 : List of Opcodes for Timing instructions

5.1 Measure Instruction

The pipeline design flow of *measure* instruction is straightforward and is shown in Figure 12. Before going into the pipeline, the design decisions taken to design the plugin are needed to be addressed (see Appendix). The instruction is assigned with a Boolean signal (IS_MEA_INST) which specifies if the current instruction is destined for the plugin we are using and a default value Boolean False will be assigned to the instruction signal. A key pattern is written in the plugin, which resembles the 32 bit *measure* instruction. When this key pattern is recognised, a list of decoding specifications are list, for example to notify something to hazard management etc., For the *measure* instruction, the specification would be to just drive the IS_MEA_INST signal to True, when the instruction matches the key pattern provided. Now, coming to the execution in pipeline, the *measure* instruction is fetched from memory mapped registers during instruction fetch stage. During the decode stage, the instruction is compared with the key pattern provided and IS_MEA_INST signal is driven to True. During the execute stage, if the input of the execution stage is the *measure* instruction, the calculations are computed, and finally values are written back to the registers.

As shown in Listing 5.2, the *measure* instruction has to be executed two times. When the first *measure* instruction is executed (line 1) it gives the current prescaled cycle counter value and when second *measure* instruction is executed (line 6) the difference between the current cycle counter value and the previous cycle counter value is given out as the output which can be observed on the UART. In this case, the resolution is exactly 1.28 μs, hence, for example if the computed value from the *measure* instruction gives out 5, then the time taken by the block of code is 1.28 μs multiplied by 5.

Listing 5.2: C code of measure instruction

```
//                          Opcode   FUNC3  FUNC7  RS1   RS2
#define measure opcode_R(CUSTOM0, 0x01,  0x00,  0x00,  0x00)

measure;                               //First measure call
cube(64);                              //task ──┐
square(56);                            //task   │ Block of code to be measured
int z[] = {23, 55, 22, 3, 40, 18};     //task   │
sumofarray(z, 6);                      //task ──┘
measure;                               //Second measure call
```

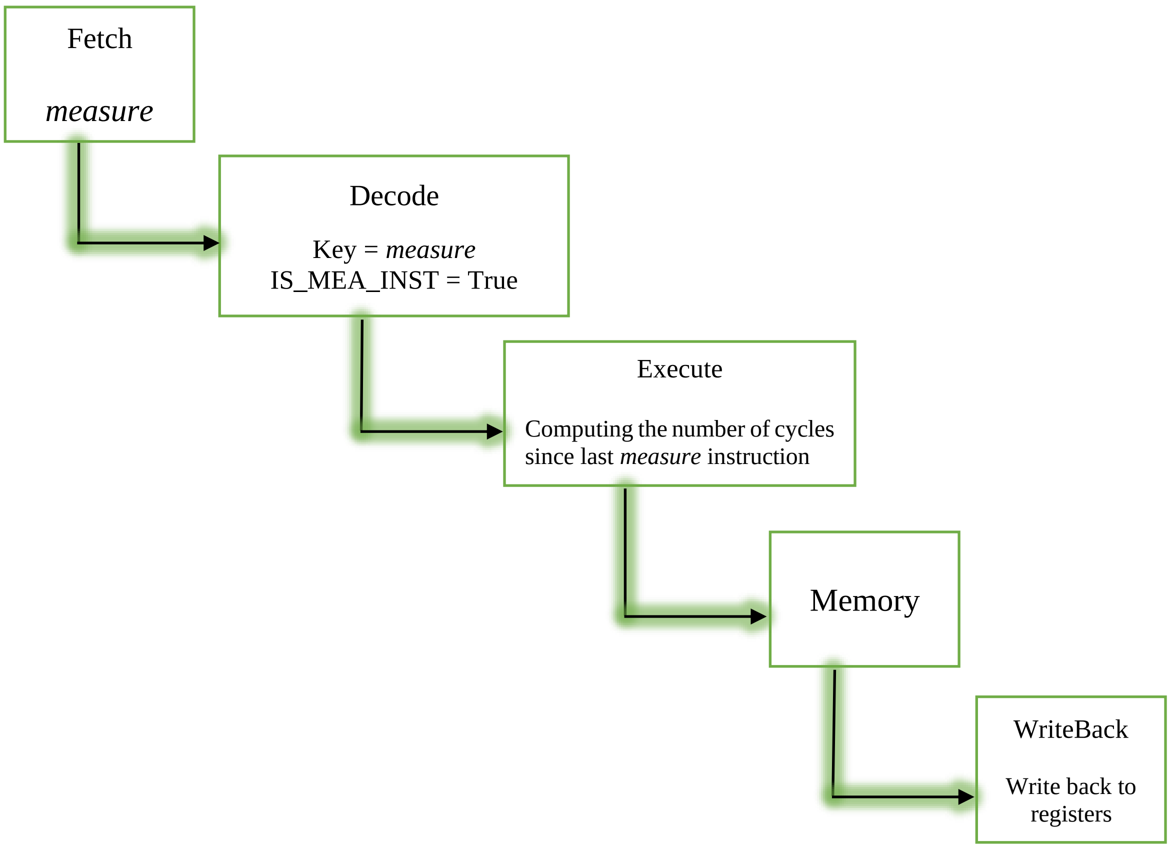


Figure 12 : Pipeline of measure Instruction

To verify the output of the *measure* instruction, an UART is implemented which is internal to the plugin where the instructions are implemented (see Appendix). In other words, the UART is controlled by hardware and not software. There are two reasons behind implementing plugins own UART. Firstly, to communicate the outputs of the *measure* instruction. Secondly, to reduce overall software overhead which is the main motto of this thesis. The similar UART can be controlled in software but the overhead increases drastically. However, with the UART being controlled in the hardware i.e. in the plugin, the overhead is reduced and this way the overall execution time of the entire program is reduced. In chapter 6, a comparison between software and hardware implementation is explained in detail.

5.2 Deadload Instruction

The pipeline design flow of the *deadload* instruction happens in a different manner and is shown in Figure 13. The plugin is designed in the same way as explained in the section of *measure* instruction. The only difference is that when key pattern is recognised in the decoding specification list, we have to specify to the hazard management unit that we are going to use source register rs1 for the purpose of getting input from the user (see Appendix 1). The rs1 bits as shown in Table 3 are commented. During execution, the instruction along with the user input stored in source register rs1 is fetched during instruction fetch stage. In the decode stage, when the key pattern is matched signal IS_DLD_INST is driven to True. During the execute stage, if the input of the execute stage is the *deadload* instruction the user input is added along with the current cycle counter value and finally values are written back to registers.

Listing 5.3, shows a code sample usage of the *deadload* instruction. The user gives a prescaled cycle counter value as input according to the requirement in line 1. In this implementation the prescaled cycle counter value is given as 100, so it would be 1.28 μs times 100, which is equal to around 128 μs. This value is then added with the current cycle counter value and is stored in the register named deadline.

Listing 5.3: C code of deadload and dead instruction

```
//                             Opcode  FUNC3 FUNC7 RS1   RS2
#define deadload(rs1) opcode_R(CUSTOM1, 0x00, 0x00, rs1, 0x00)

#define dead opcode_R(CUSTOM2, 0x00, 0x00, 0x00, 0x00)


1    deadload(100);                    //load deadline value from user ( 1 = 1.28 µs)
2    cube(64);                         //task
3    square(56);                       //task
4    int z[] = {23, 55, 22, 3, 40, 18};  //task
5    sumofarray(z, 6);                 //task
6    dead;                             // halt until deadline value is reached
```

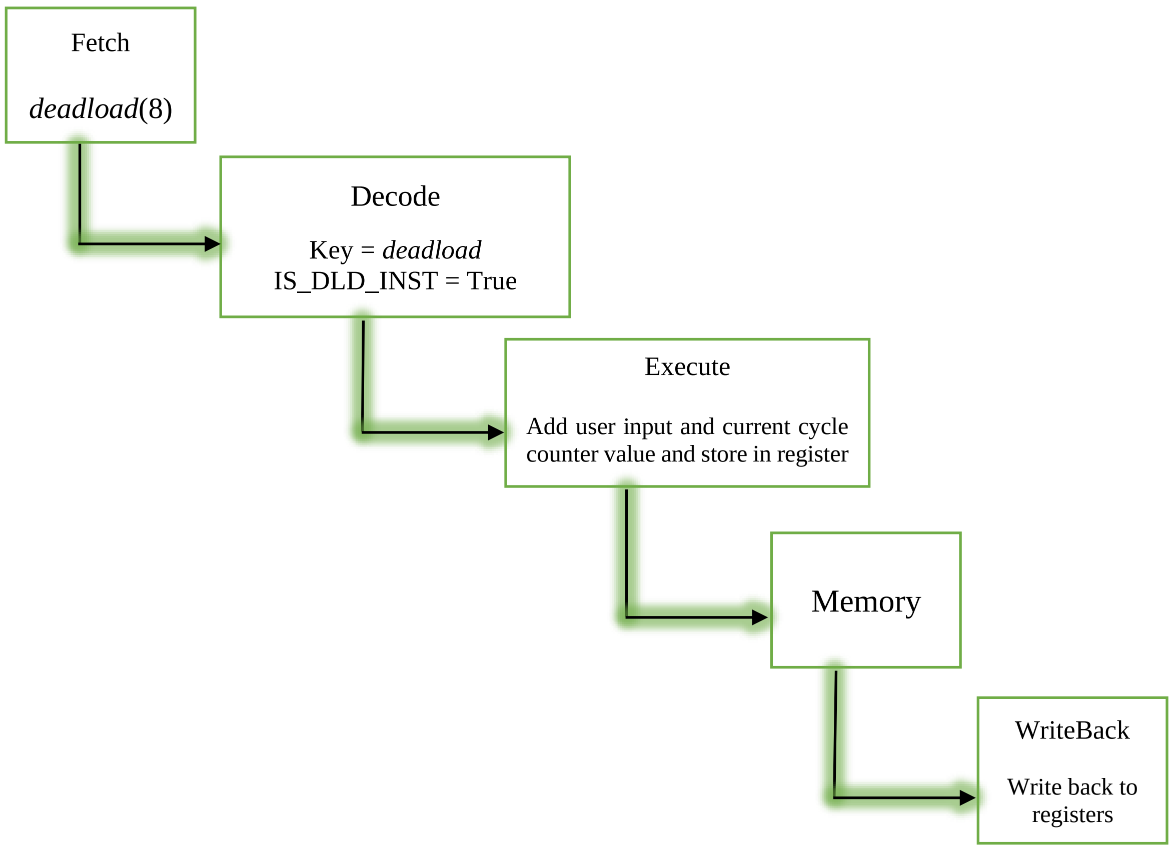


Figure 13 : Pipeline of deadload instruction

5.3 Dead Instruction

The pipeline design flow of the *dead* instruction is similar to that of *measure* instruction and the plugin is also designed in a similar fashion. Figure 14 shows the pipeline flow of the *dead* instruction. The instruction is fetched from the memory in the instruction fetch stage. During decode stage when the key pattern matches the *dead* instruction pattern, the signal IS_DED_INST is set to be True. In execute stage, when the input is *dead* instruction, the deadline register and current cycle counter are compared. When the current cycle counter is less than or equal to the value in the deadline register, a signal named haltItself is driven to be True, which halts the pipeline and a register named verify is set to decimal 5, to verify the output of this instruction on the LEDs of the FPGA. So, until the pipeline is halted the LEDs show the binary representation of decimal 5 - “0101” on them, and as soon as when the current cycle counter value is not less than or equal to the value in the deadline register the signal haltItself is set to False and a verify register is set to decimal 6. This LED implementation is only for visual debug purpose and should be removed after a successful implementation.

As shown in Listing 5.3, in line 1, the *deadload* instruction takes in the user input and the result is stored in a register. So, in line 6, when *dead* instruction is decoded, the program waits there because the pipeline is halted until the deadline value is reached by the cycle counter.

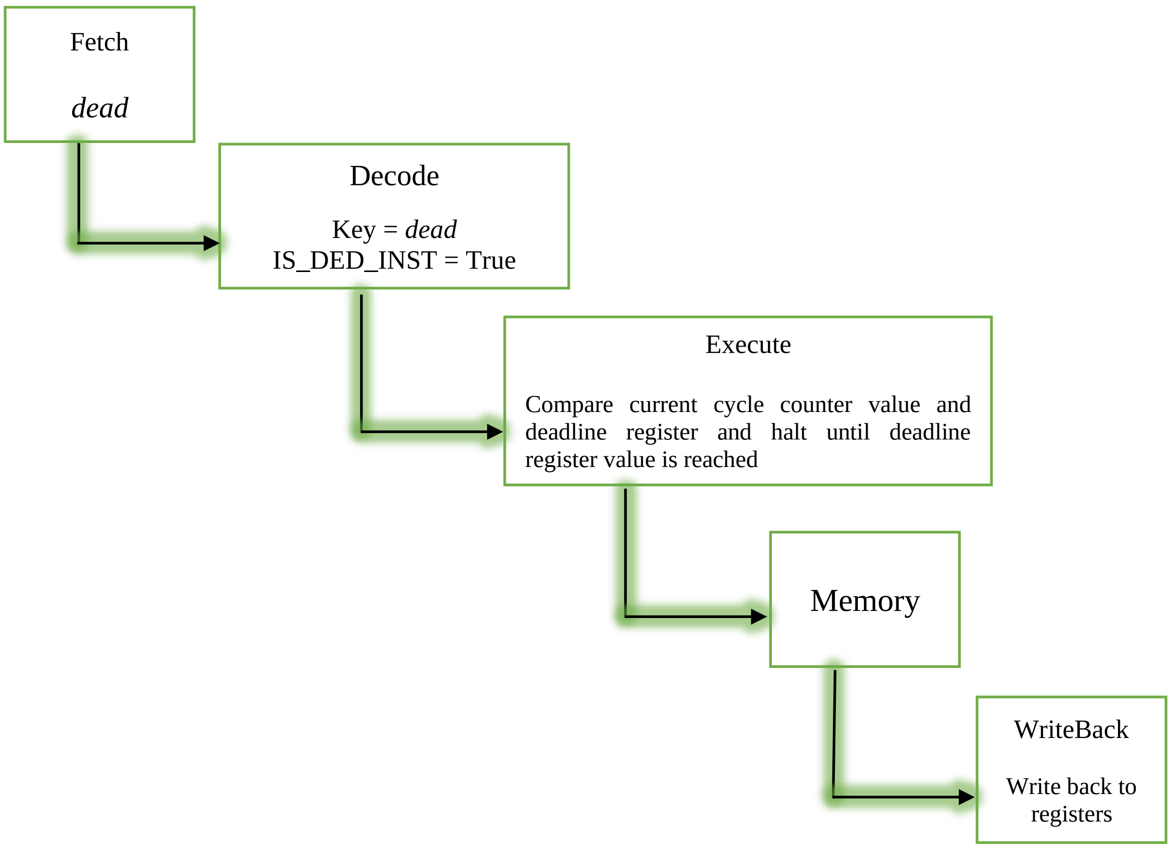

Fetch
dead
Decode
Key = dead
IS_DED_INST = True
Execute
Compare current cycle counter value and deadline register and halt until deadline register value is reached
Memory
WriteBack
Write back to registers

Figure 14 : Pipeline of dead Instruction

5.4 Deadexcep Instruction

The pipeline design flow of *deadexcep* is shown in Figure 15. In the plugin, initially an exception service for an exception to be emitted during execution stage has to be initiated. The instruction is fetched from the memory in the instruction fetch stage. During the decode stage, the key pattern is matched with the exception and the signal IS_DEX_INST is driven to True. During the execute stage, when current cycle counter value is less than or equal to deadline register value, then the pipeline will halt showing us visible result on the LED with the binary representation 0101. But when current cycle counter value is greater than the deadline register value, the exception port's valid signal must be set to True by the condition that current cycle counter value is greater than the deadline register value. The exception port which is defined in the setup part of the plugin (see Appendix 1) contains an input parameter where the exception code is written, this exception code which is nothing but the code which goes into mcause register (see Chapter 4). This code can be read in software, to take according measures to handle the exception.

As seen in Listing 5.4, *deadload* instruction at line 1, takes the input from the user and the *deadexcep* instruction at line 6, halts the pipeline until the current cycle counter value passes the deadline value else triggers an exception and the code in the exception handler is executed.

Listing 5.4: C code of deadload and dead instruction

```
//                              Opcode   FUNC3 FUNC7 RS1   RS2
#define deadload(rs1) opcode_R(CUSTOM1, 0x00, 0x00, rs1, 0x00)

#define deadexcep opcode_R(CUSTOM3, 0x00, 0x00, 0x00, 0x00)

1   deadload(50);                   //load deadline value from user ( 1 = 1.28 μs)
2   cube(64);                       //task
3   square(56);                     //task
4   int z[] = {23, 55, 22, 3, 40, 18};   //task
5   sumofarray(z, 6);               //task
6   deadexcep;                      // halt until deadline value is reached and emit an exception
                                    // if deadline is missed
```

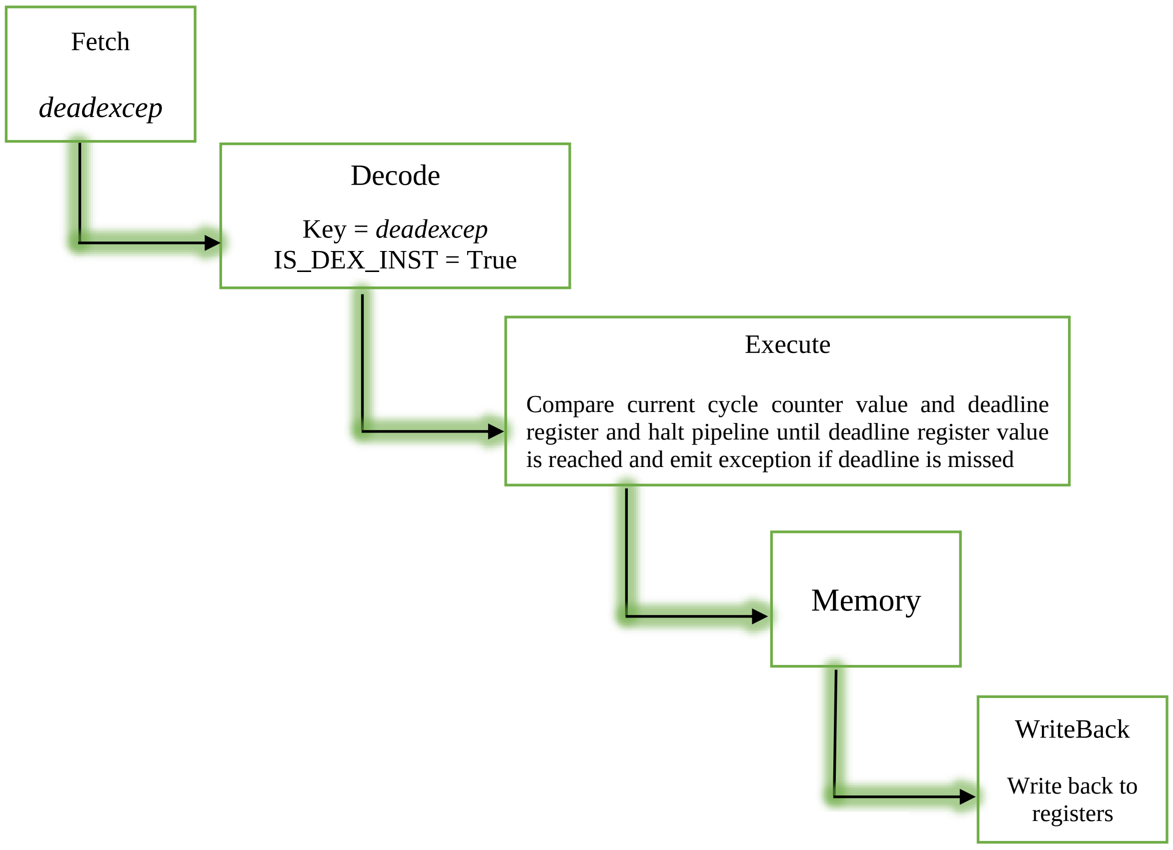

Fetch
deadexcep
Decode
Key = *deadexcep*
IS_DEX_INST = True
Execute
Compare current cycle counter value and deadline register and halt pipeline until deadline register value is reached and emit exception if deadline is missed
Memory
WriteBack
Write back to registers

Figure 15 : Pipeline of deadexcep instruction

5.5 LIBBLA with deadline instructions

The timing control blocks concept in C++ library LIBBLA was implemented on an ARM Cortex A9 bare metal platform. In order to use the deadline instructions, the concept of LIBBLA has to be adapted to the VexRiscV and Murax SoC architecture. Hence, the concept of EET block and FET block without block management is implemented according to the specifications of Murax SoC and was tested. From Chapter 4, it is understood that the deadline instructions and time control blocks in C++ LIBBLA library have a similar concept. Hence, the EET blocks are replaced by *measure* instruction and FET blocks are replaced by *deadload* and *deadexcep* instructions and the results between both C++ library time blocks and time blocks replaced by deadline instructions are obtained.

6. Evaluation and Comparison

The main motto of the thesis boils down to the fact that hardware implementation should most probably be much faster than the software implementation of the same concept. So, an evaluation and comparison between hardware implementation and software implementation is done, where in the aspects like number of lines of C code, number of lines of assembler code, software overhead, memory, accuracy etc, are compared to give a detailed overview about the advantages and disadvantages of both the implementations and also a comparison on hardware overhead is discussed. This chapter is about the evaluation of the results, which are associated with the implementation described in Chapter 5, and also comparison between hardware implementation and software implementation, when the similar concepts as explained in Chapter 5 are implemented on the software level without the use of the hardware instructions that are introduced.

The evaluation is done on the basis of accuracy for all the four instructions with the help of oscilloscope, meaning the results obtained by executing the instructions is cross verified to conclude if there is any loss in accuracy of measurement in case of *measure* instruction and in case of deadload and dead instructions which are used together. The delay time allotted to the block of code is evaluated to be accurate or not. Furthermore, the hardware overhead which occurs due to the addition of the implemented plugin is compared against the hardware overhead which occurs without the use of the plugin. Also, as explained in Chapter 5, the hardware overhead which occurs if four instructions are implemented in four different plugins against implementation of all instructions in one plugin is compared. The software overhead and increase in the execution time which occurs, when the similar concept is implemented purely in software without the use of the four instructions is evaluated against the implemented concept with the use of the four instructions. The size of the memory and number of lines of C code and assembler code generated in both the cases, with and without using instructions, is also taken into account and comparison is done accordingly.

6.1 Accuracy of Measure Instruction

The *measure* instruction as it is known is used to measure the execution time of a block of code. When measuring comes into picture, the accuracy of the results play an important role. Hence, to evaluate the accuracy, a reference point is needed. To obtain this reference point, a GPIO output signal of the Murax SoC is placed. The signal is placed when the block of code starts and one when the block of code ends. The signal at the start of the block of code gives out a value 1, while the signal at the end of the block of code gives out a value 0.

These GPIO output signals can be led out straight to the PMOD headers on the FPGA as seen in Figure 16, by defining the requirement on the toplevel file in Xilinx Vivado IDE. Now, the output coming out from the PMOD headers can be connected to an oscilloscope, and the wave pattern obtained there can be analysed to calculate the exact time taken by our chosen block of code to execute. Reading GPIO signals does not produce any overhead in the system as the program does not stop to read any memory and hence making it an efficient way to measure the accuracy here.

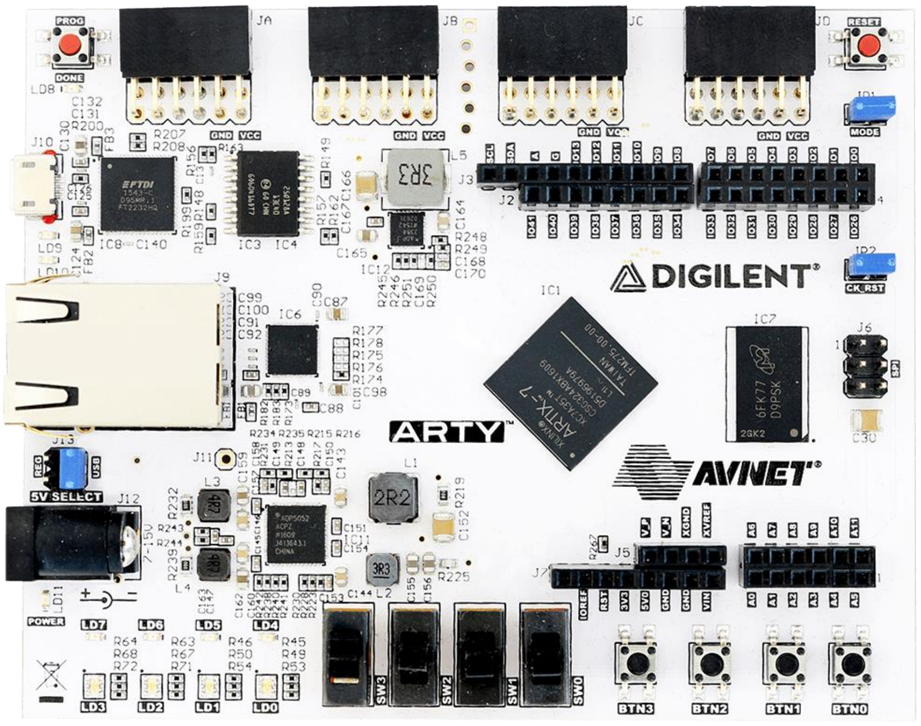


Figure 16 : Arty A7 FPGA [23]

The wave pattern is shown in Figure 17. By analysing the wave pattern, the execution time of the block of code (see Listing 5.2) is found to be 70.4 μs. To be exactly sure, the block of code is executed twice to be sure if it would take twice the time as seen in Figure 18, and the execution time was analysed to be 140 μs, concluding the first measurement was correct.

Now, since the reference point is set, the execution time measured from the *measure* instruction is observed. The output as known is obtained on the UART through the inbuilt UART in the plugin. After execution of the block of code, the prescaled cycle counter value are obtained as the output, with the value 55 showing on the UART, which means 55 multiplied by 1.28 μs is equal to 70.4 μs, thereby concluding that the *measure* instruction measures the execution time of the block of code accurately.

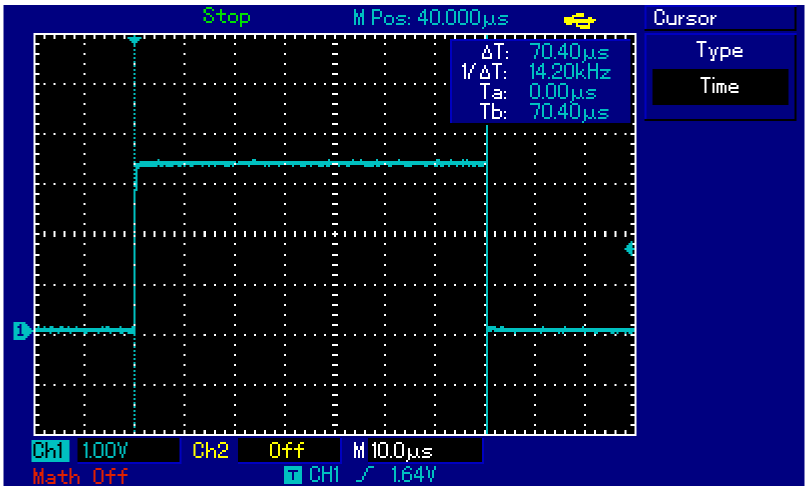


Figure 17 : Oscilloscope output of execution time taken by block of code

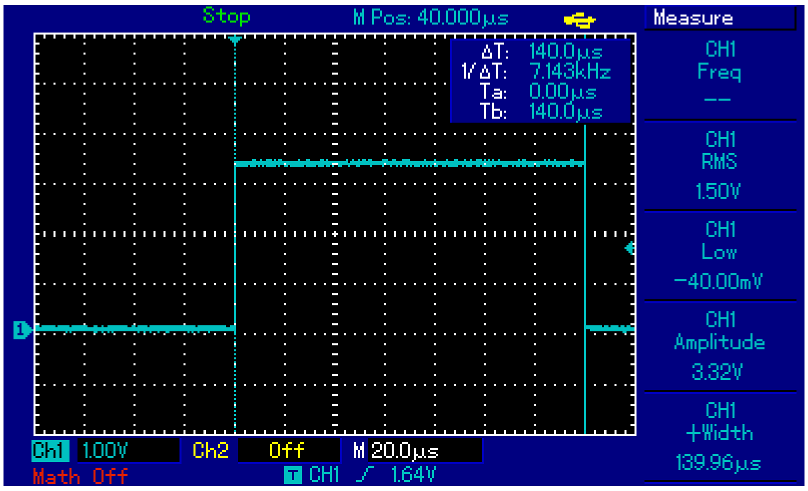


Figure 18 : Oscilloscope output of execution time taken by block of code when executed twice

6.2 Accuracy of Deadload and Dead Instruction

The *deadload* and *dead* instruction are always used in combination to set a minimum execution time for a block of code. To evaluate, whether the block of code is taking the minimum execution time prescribed by the user, the GPIO output signal is placed, one before the start of the code, but in this case above the deadload instruction because the deadload instruction will be setting the deadline value and one after the end of block of code (see Listing 5.3), but after dead instruction, where the program halts until the deadline value is passed.

The wave pattern is shown in Figure 19. For this experiment, a minimum execution time of 128 μs is assigned to the code block. The block of code used here is the one for which the execution time is already known, which is 70.4 μs. From the wave pattern, it can be observed that the minimum execution time given to the block of code is found to be 128.84 μs, with an overhead of 0.84 μs.

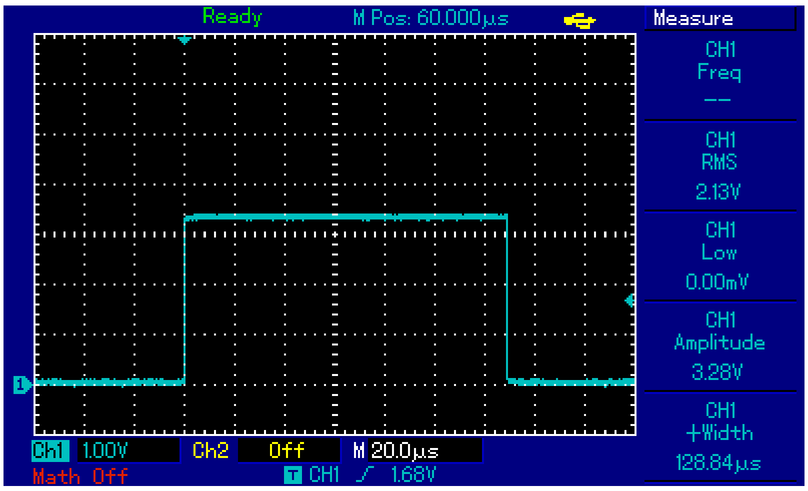


Figure 19 : Oscilloscope output of the test to observe if the block of code is exactly delayed to set the time

6.3 Comparison of Measure Instruction

The notion that anything implemented on hardware will be faster than pure software implementation persists. In order to make these statements it has to be backed up with a practical proof of implementation. Hence, the decision to compare the *measure* instruction implemented in the plugin against the same concept implemented at a pure software level. In order to perform this, in software a set of functions are implemented which replicate the actions performed by the *measure* instruction. For example, in the plugin, the *measure* instructions subtracts the current cycle counter value with the previous cycle counter and sends the result to the UART. This same action is implemented at the software level and the results are obtained in a similar way by placing the GPIO signal on either ends of the block of code as the accuracy experiment.

The GPIO output signals are placed in the program with *measure* instruction and in the program where the pure software implementation is present with the functions. As we already know, the program with the *measure* instruction with the same block of code that is used before, takes 70.4 μs including UART transmission as seen in Figure 20. But the pure software implementation of the same concept including UART transmission took 676 μs as seen in Figure 21, an overhead of about 600 μs is introduced if implemented in pure software excluding the execution time of the code block, which has a drastic increase in the overall execution of the program, thereby reducing the overall performance and efficiency of the program. This overhead is mainly due to the software controlled UART, whereas with the *measure* instruction an UART controlled by hardware is implemented which drastically reduces the overhead.

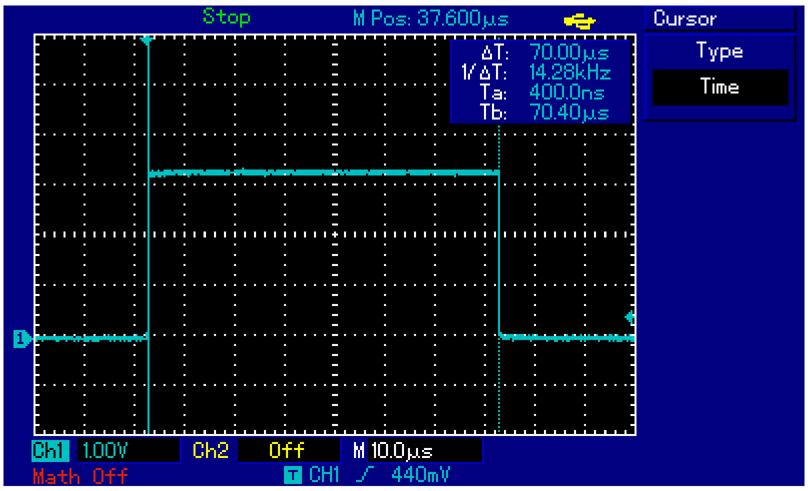


Figure 20 : Oscilloscope output to observe overhead of time produced by hardware measure instruction

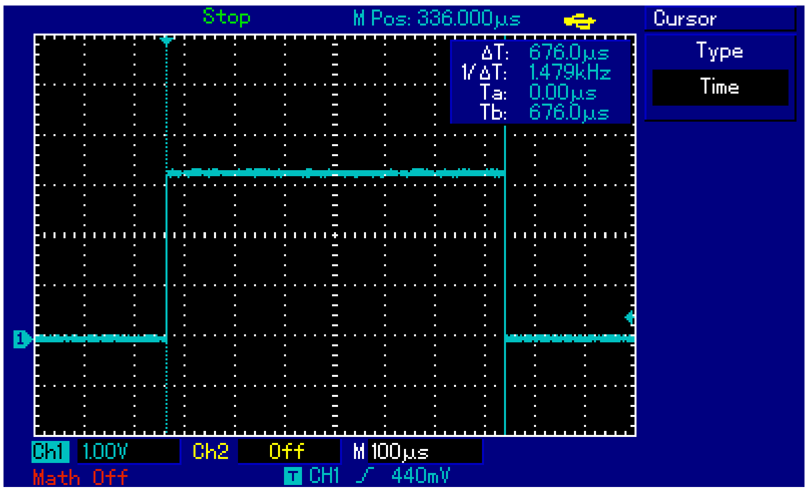


Figure 21 : Oscilloscope output to observe overhead of time produced when the concept of measure instruction is implemented in pure software

6.4 Comparison of Deadload and Dead instruction

Similar to the comparison of *measure* instruction with its concept implementation in pure software, the *deadload* and *dead* instructions concept is also implemented at the software level to compare the overhead produced. The exact procedure of the user providing the input for the minimum execution time of block of code in case of *deadload* instruction and delaying the execution of the program until the timer value is reached in case of *dead* instruction is implemented using pure software with a set of functions. The procedure which is incorporated in software is that the user gives an input for the minimum execution time of block of code, this input is added with the current cycle counter value and stored in a variable and then the block of code is executed. After the execution of block of code is complete, the value in the stored variable is subtracted from current cycle counter value. If the difference is a positive value, then the one of the Murax SoC timer is configured and is compared against the difference value and waits until timer is greater than the difference value obtained, thereby creating a delay until the specified time.

To compare the hardware and pure software implementations, the GPIO output signal is placed above *deadload* instruction and below *dead* instruction. Between those two instructions the block of code needed to be executed is present and the output obtained is observed in Figure 18. Similarly, the GPIO output pins were placed above deadload and dead functions in software and the execution time was obtained. From Figure 19 above, it is observed that when this concept is implemented at hardware level, there was overhead of 0.84 μs. But at the software level, as observed in Figure 22, the execution time is 133 μs , for the specified delay time 128 μs. Hence, there is overhead of 5 μs in pure software implementation as compared to hardware level implementation where the overhead is minimal. This affects the final result when trying to control a block of code only with software implementation without the help of hardware implementation.

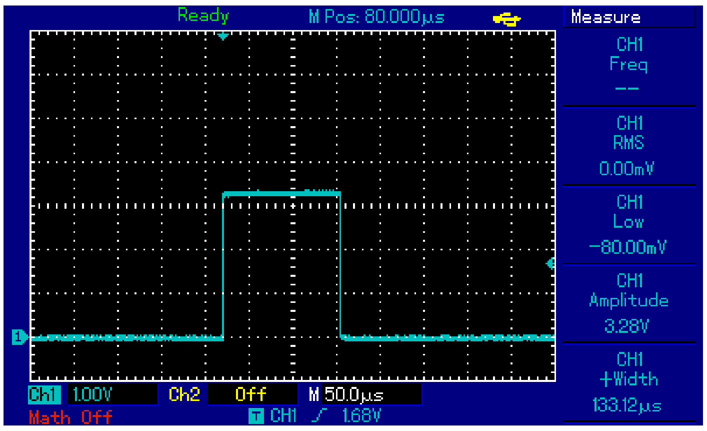


Figure 22 : Oscilloscope output to observe overhead of time produced when the concept of deadload and dead instruction is implemented in pure software

6.5 Comparison of Deadload and Deadexcep Instruction

The *deadexcep* instruction is compared to its software replication in a slightly different manner in pure software. The *deadexcep* instruction as it is known emits an exception if the current cycle value is greater than the deadline value set by the user. To replicate the same behaviour in software, the timer interrupt is used for this experiment.

So, for a timer interrupt to trigger at the specific time, the user has to set the timer according to the requirement and when this timer reaches zero, the interrupt handler does the respective operation defined by the user. This means the user has to set the timer every time, when the minimum execution time of the block of code changes, whereas in the case hardware implementation, there is no need of this extra setting by the user since the exception can be emitted from the plugin itself, thereby reducing the overhead and improving the overall performance and efficiency of the code. The procedure followed is similar as explained in previous section. The only addition here is that the timer is disabled when the block of code takes the specified execution time else the if the block of code takes more time than the specified time, the timer interrupt is triggered. The comparison of the overhead between hardware implementation and software implementation of emitting exception is shown in the next section.

6.6 Comparison of Assembler Lines of Code and Memory

The makefile and linker script is written in such a way that, where when the software is compiled every time, a (.asm) file will be generated which contains the assembly level code and a statistic regarding the memory usage of the on chip RAM of Murax SoC will also be printed. Since, Murax SoC is a softcore SoC, the On Chip RAM size is increased from its original 8KB to 128KB for the purpose of experiments.

Hence, the information is represented in the form a column graph for a better comparison of hardware and software implementation. Figure 23 shows the comparison between hardware and pure software implementation for the *measure* instruction. It can be observed that both the assembler lines of code and memory size have increased by 100.5% and 82.5% respectively, in pure software implementation as compared to the hardware implementation. This is purely due to the various functions defined in pure software implementation which takes up a lot of memory.

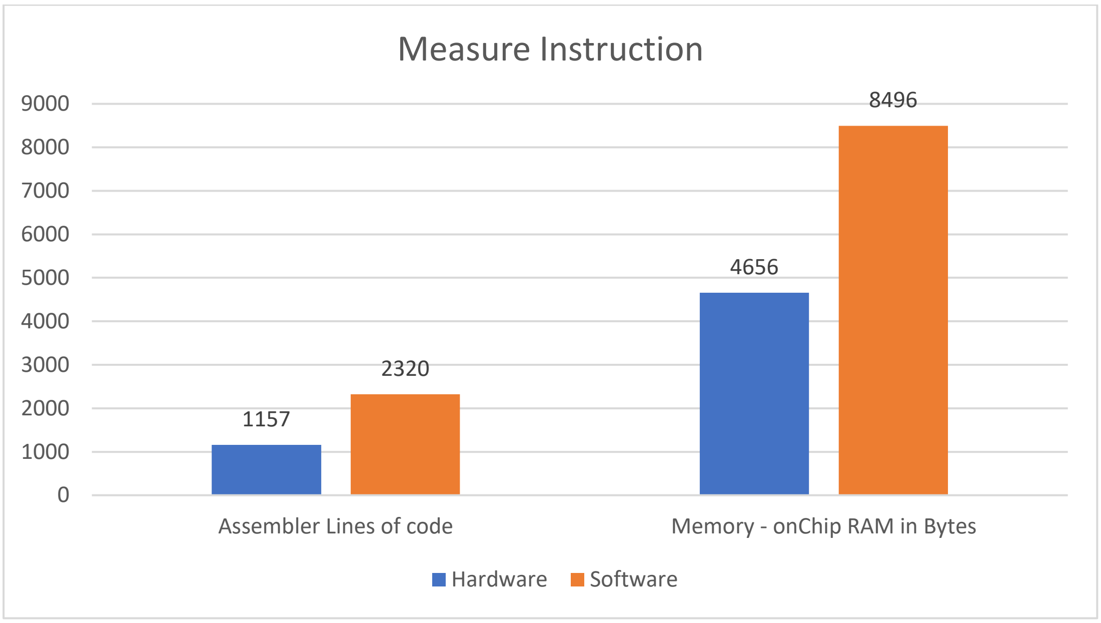


Figure 23 : Chart comparing the Assembler lines of code and memory of hardware measure instruction and its equivalent software implementation

Figure 24 shows the comparison between hardware and pure software implementation for the *deadload* and *dead* instruction. As expected, the assembler lines of code and the on chip memory size of RAM have increased by 26.4% and 21.3% respectively, in pure software implementation compared to the hardware implementation due to increase in code size because of the various functions defined in pure software implementation.

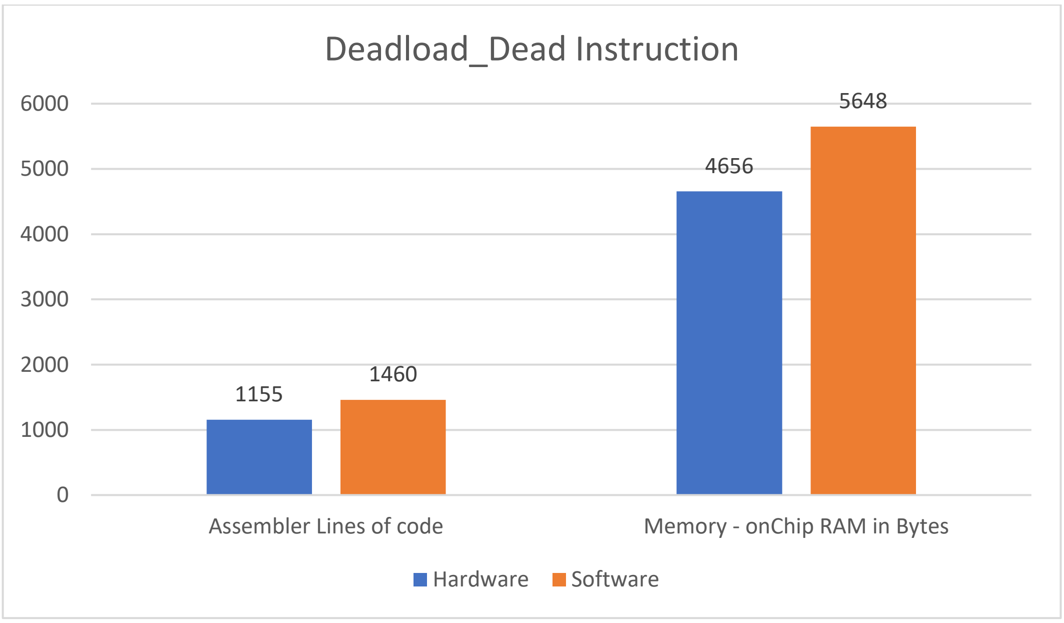


Figure 24 : Chart comparing the Assembler lines of code and memory of hardware deadload and dead instruction and its equivalent software implementation

Figure 25 shows the comparison between hardware and pure software implementation for the *deadload* and *deadexcep* instruction. As expected, the assembler lines of code and the on chip memory size of RAM have increased by 6.9% and 7.8% respectively in pure software implementation compared to the hardware implementation due to increase in code size because of the various functions defined in pure software implementation.

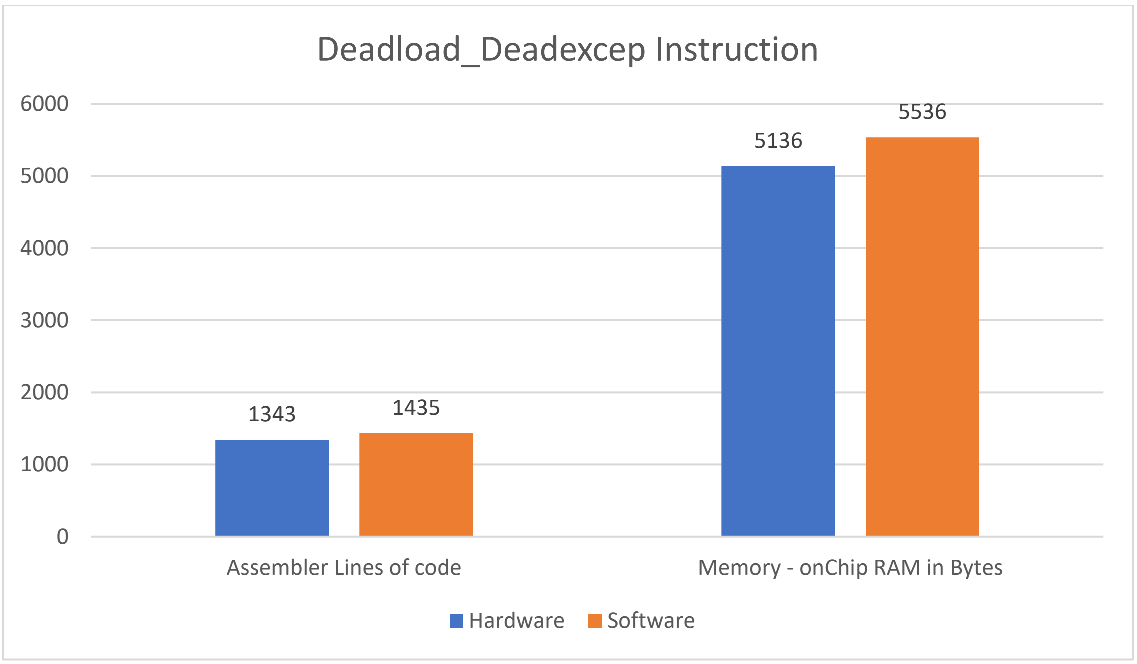


Figure 25 : Chart comparing the Assembler lines of code and memory of hardware deadload and deadexcep instruction and its equivalent software implementation

6.7 Comparison of Hardware Overhead

The most important aspect while dealing with FPGAs is to have a close look into the number of LUTs used, because these have a direct impact on the price of the FPGAs. Hence, an evaluation and comparison on how much hardware overhead is introduced due to the extra logic is required. To achieve the same, Xilinx Vivado IDE helps us in understanding the number of LUTs and Flip-Flops used and in more advanced cases, an even detailed breakdown of the hardware can be obtained from this tool. So, using Xilinx Vivado IDE, the information needed for the comparison of hardware overhead that the plugin is introducing after implementation is collected.

Figure 26 shows the comparison between the hardware produced with and without the plugin, after synthesising the RTL logic which is the generated Verilog file, VexRiscV CPU produces. It can be observed that without plugin 1606 LUTs are being used along with 1670 Flip-Flops. With plugin, the count of LUTs increases to 1738 and count of Flip-Flops increases to 1790, which is an increase of about 8.2% of LUTs and 7.2% of Flip-Flops.

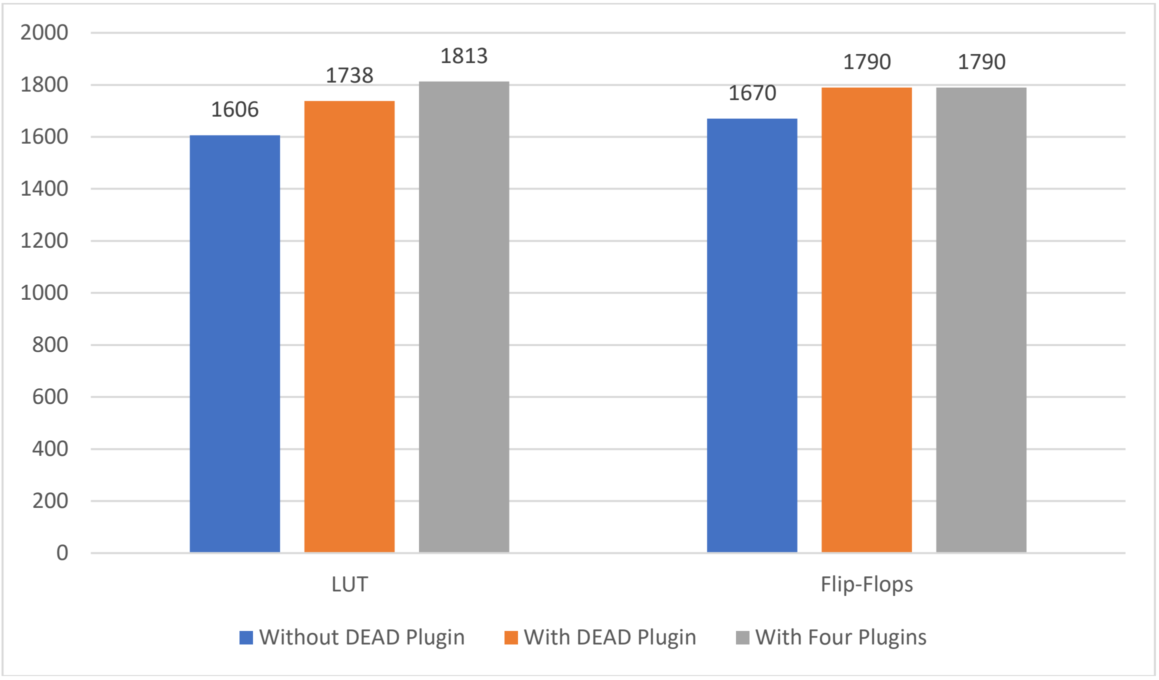


Figure 26 : Chart comparing the number of LUTs and Flip Flops used with DEAD Plugin, without DEAD Plugin and when instructions are implemented in four different plugins

Figure 26 also shows the hardware overhead produced, when instructions are written in one plugin and when instructions are written across four different plugins. According to the chart, as expected when instructions are spread across four different plugins the number of LUTs used are 1813 while if the instructions are written in one plugin, the LUTs produced are 1738, with a difference about 75 LUTs which is an increase of about 4.3% of LUTs, whereas the Flip-Flops generated remain the same in both the cases. This can be due to the more amount of logic required to implement four different plugins. Hence, during production phase one can adopt to the use of a single plugin for all the deadline instructions in order to reduce the overall LUT count, but at the same time the overall logic a chip can do reduces as the number of LUTs decreases, so a decision, matching the end requirements have to be taken.

6.8 Comparison of LIBBLA with deadline instructions

As explained in this Chapter, with the help of makefile and linker script, the number of assembler lines of code required and the memory occupancy of the code on the Murax SoCs On Chip RAM. The original C++ LIBBLA implementation results on Murax SoC and VexRiscV CPU and the results obtained by replacing the time blocks in C++ LIBLLA library by deadline instructions are compared and evaluated in detail in this subsection.

Figure 27 shows the comparison between the EET Block and *measure* instruction. As observed in the graph, it is clear that the using hardware instructions can drastically reduce the overhead produced compared to its pure software implementation. It can be seen that the number of assembler lines of code needed almost increased by 229.5% and the memory needed almost increased by 194.4% in case of pure software implementation due to the various functions defined. So, clearly use of the deadline instructions in libbla gives more advantage than using the libbla as it is.

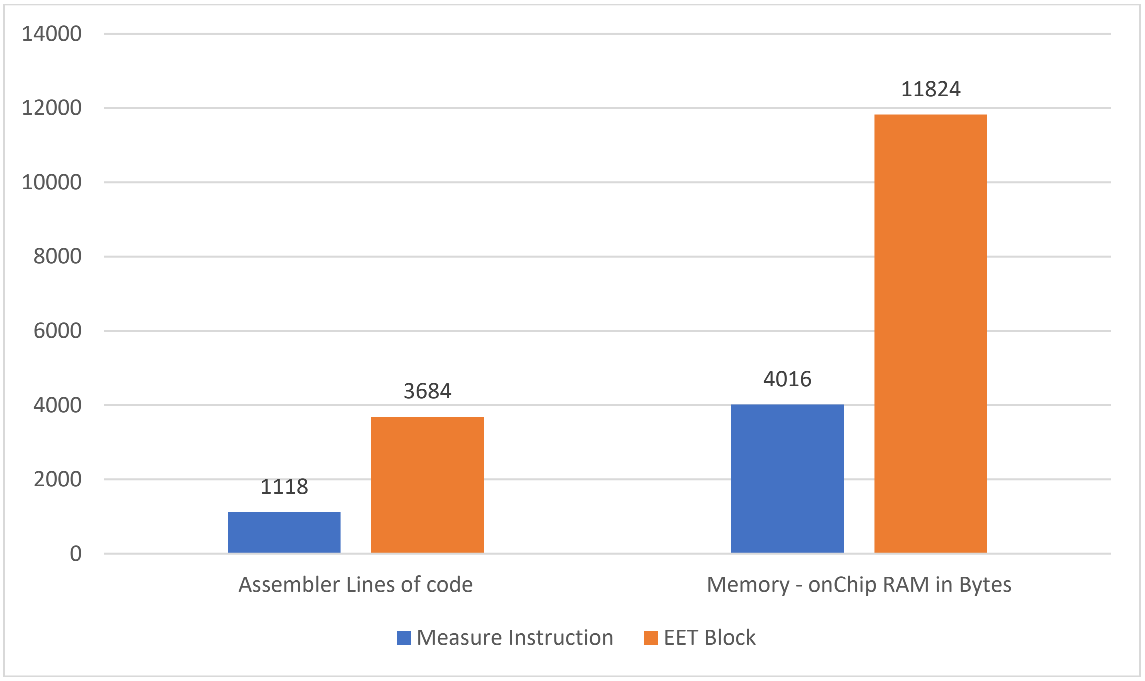


Figure 27 : Chart comparing the assembler lines of code and memory between libbla EET block and measure instruction

Figure 28 shows the graph of comparison between FET Block and the *deadload* and *deadexcep* instructions. Once again, it is clear that the pure software implementation requires double the assembler lines of code and triple the memory on RAM, when compared to the hardware implementation. The Assembler lines of code increased by 187.2% and memory increased by 168.5% in the pure software implementation.

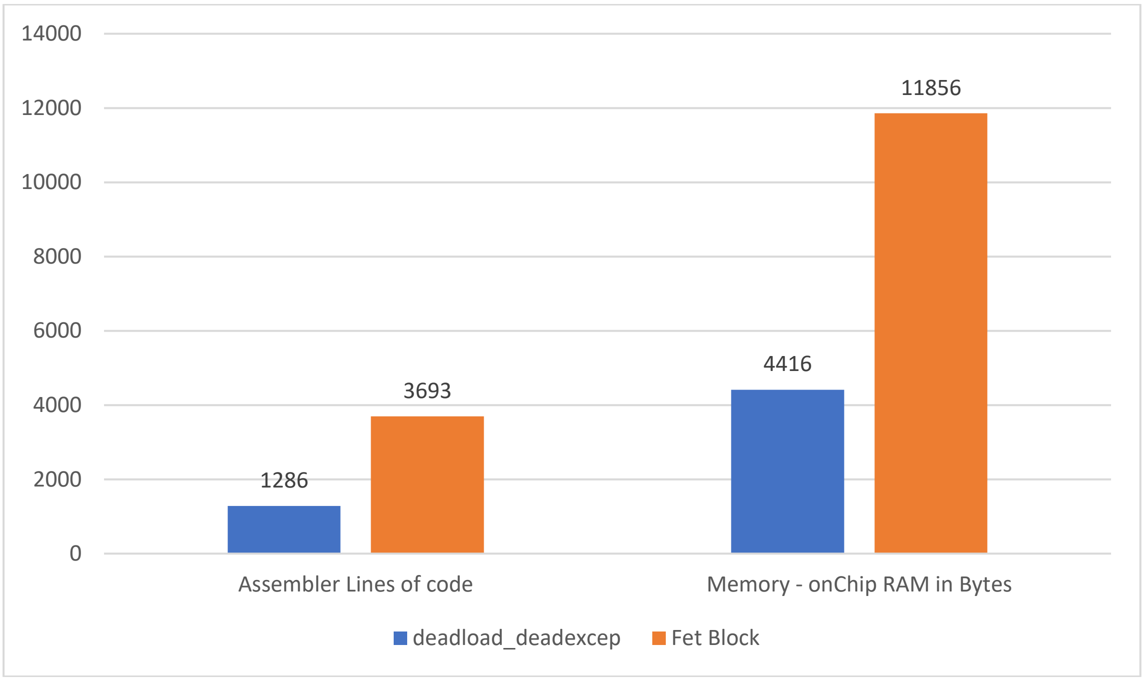


Figure 28 : Chart comparing the assembler lines of code and memory between libbla FET block and deadload and deadexcep instruction

7. Conclusion and Future Work

In order to integrate the Real Time systems with the environment and achieve timing compliance, it is contended that the designers must reach beneath the abstraction layers and also traditional methods of using programmable timers and interrupt service routines to imbibe temporal semantics were argued upon citing their disadvantages in resource constrained embedded systems. In this thesis, the focus is on targeting the instruction set architecture and keeping the software requirement to a minimal. The custom instruction extension feature of the RISC-V architecture is explored to bring timing control over to the software level, so that temporal semantics can be guaranteed.

To achieve these timing properties, this thesis presented four instructions to the existing instruction set. The instruction set extensions allows developers to apply a minimum execution time for a block of code and throw hardware exceptions when timing specifications are not met. The thesis started with a conceptual idea of implementation the custom instructions after referring a few state of the art research works. The next stage was the implementation of the decided conceptual idea in the chosen VexRiscV CPU in SpinalHDL Language. The implementation was tested on an FPGA named Arty A7 and expected results were obtained. The next stage was evaluation of the results obtained for its accuracy to conclude that the results produced were accurate. The evaluation proved that the accuracy was being met. Next, a comparison of various parameters were performed between hardware implementation and software implementation, where the hardware implementation proved to have many advantages over pure software implementation. In the next stage, an attempt to integrate the deadline instructions to a C++ library named LIBBLA was made, where it implements time control blocks in software. After migrating the concept which was implemented on ARM Cortex A9 to VexRiscV CPU softcore Murax SoC, the deadline instructions were integrated after finding the similarities in the functionalities between the deadline instructions and time control blocks in the C++ library. After a simple evaluation it was found that the hardware implementation was two to three time more advantageous in terms of lines of code required and memory occupied respectively.

Thus, it could be concluded that taking the help of hardware where processes happen concurrently rather than sequentially, the required specifications can be met by using the available resources to a minimal.

The implemented custom instructions have various possibilities for improvement and addition of new features. One such improvement is to implement the concept of nested instructions to integrate with the libbla library, as the time blocks in libbla have this nesting feature where FET block can be called inside an FET block, similarly the concept can be applied to the custom instructions *deadload* and *dead* , where in which we can set multiple *deadload* and *dead* instructions inside one another. Another improvement can be to pass on the leftover time to the next block if the previous block finishes early than the specified time.

8. Bibliography

[1] I. Liu, "Precision Timed Machines," *PhD dissertation UC Berkley,* 2012.

[2] C. Papon, "VexRiscv git repository," [Online]. Available: https://github.com/SpinalHDL/VexRiscv. [Accessed 20 October 2021].

[3] F. Bruns, I. Yarza, P. Ittershagen and K. Grüttner, "Time Measurement and Control Blocks for Bare-Metal C++ Applications," *ACM Trans. Embed. Comput. Syst. 20, 4, Article 34*, May 2021.

[4] A. Myers, "SecVerilog: Verilog + Information Flow," [Online]. Available: http://www.cs.cornell.edu/projects/secverilog/. [Accessed 29 January 2022].

[5] "Field Programmable Gate Array (FPGA)," Xilinx, [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed 03 February 2022].

[6] T. V. Subbareddy, M. R. Bommepalli, H. N. Upadhyay and E. Vellaiappan, "Low power look-up table topologies for FPGAs," July, 2014.

[7] "INSTRUCTION SET ARCHITECTURE (ISA)," [Online]. Available: https://www.arm.com/glossary/isa. [Accessed 30 January 2022].

[8] J. Blyler, "RISC-V, Open Source Hardware and CHIPS Trends," 28 January 2020. [Online]. Available: https://www.chipestimate.com/RISC-V-Open-Source-Hardware-and-CHIPS-Trends/blogs/3292. [Accessed 31 January 2022].

[9] "Reduced instruction set computer," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Reduced_instruction_set_computer. [Accessed 27 January 2022].

[10] "RISC-V," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/RISC-V. [Accessed 28 January 2022].

[11] A. Waterman and K. Asanovi´c, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version, RISC-V Foundation, December 2019.

[12] "RISC-V International," [Online]. Available: https://riscv.org/exchange/cores-socs/. [Accessed 05 February 2022].

[13] M. Sinclair, "RISC-V SoftCPU Contest Winners," 6 December 2018. [Online]. Available: https://riscv.org/announcements/2018/12/risc-v-softcpu-contest-winners-demonstrate-cutting-edge-risc-v-implementations-for-fpgas/. [Accessed 02 March 2022].

[14] C. Berg, J. Engblom and R. Wilhelm, "Requirements for and Design of a Processor with Predictable Timing," January 2004.

[15] W. Wang, B. Jungk, J. Wälde, S. Deng, N. Gupta, J. Szefer and R. Niederhagen, "Selected

Areas in Cryptography – SAC 2019 - 26th International Conference, Revised Selected Papers," in *XMSS and Embedded Systems: XMSS Hardware Accelerators for RISC-V*, Waterloo, Springer, 2020, pp. 523-550.

[16] C. Papon, "SpinlaHDL git repository," [Online]. Available: https://github.com/SpinalHDL/SpinalHDL. [Accessed 29 January 2022].

[17] E. A. Lee, J. Reineke and M. Zimmer, "Abstract PRET Machines," *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[18] D. Bui, E. Lee, I. Liu, H. Patel and J. Reineke, "Temporal Isolation on Multiprocessing Architectures," *Proceedings of the 48th Design Automation Conference*, 2011.

[19] I. Liu, B. Lickly, H. D. Patel and E. A. Lee, "Poster Abstract : Timing Instructions - ISA Extensions for Timing Guarantees," *Real-Time Embedded Technology and Applications Symposium (RTAS)*, 2009.

[20] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards and E. A. Lee, "Predictable Programming on a Precison Timed Architecture," *Conferences on Compiles, Architectures, and Synthesis of Embdedded Systems (CASES)*, 2008.

[21] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," *Embedded and Ubiquitous Computing*, vol. 4096, pp. 449-458, Aug. 2006.

[22] A. Waterman and K. Asanovi´c, The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified, RISC_V Foundation, June 2019.

[23] "SEGGER Evaluation Software for RISC-V Digilent ARTY," [Online]. Available: https://www.segger.com/evaluate-our-software/risc-v/digilent-arty/. [Accessed 02 March 2022].

[24] "WIkipedia - Field Programmable gate array," [Online]. Available: https://en.wikipedia.org/wiki/Field-programmable_gate_array. [Accessed 22 February 2022].

[25] J. Ledin, "The RISC-V Architecture," 05 June 2020. [Online]. Available: https://dzone.com/articles/introduction-to-the-risc-v-architecture. [Accessed 27 January 2022].

Appendices

1. DEAD_Plugin :

```
package vexriscv.demo

import vexriscv._
import spinal.core._
import vexriscv.demo._
import vexriscv.plugin.Plugin
import vexriscv.{Stageable, DecoderService, ExceptionService, VexRiscv}
import spinal.lib.io.TriStateArray
import spinal.lib.{Flow, master}
import vexriscv.plugin.{CsrInterface, Plugin}
import spinal.lib.com.uart._
import spinal.lib._

class DEAD_Plugin extends Plugin[VexRiscv]{

  //Define the concept of IS_SIMD_ADD signals, which specify if the current
instruction is destined for this plugin
  object IS_MEA_INST  extends Stageable(Bool)
  object IS_DLD_INST  extends Stageable(Bool)
  object IS_DED_INST  extends Stageable(Bool)
  object IS_DEX_INST  extends Stageable(Bool)

  var MEA                : UInt = _
  var uart               : Uart = _
  var deadExceptionPort : Flow[ExceptionCause] = null

  //Callback to setup the plugin and ask for different services
  override def setup(pipeline: VexRiscv): Unit = {
  import pipeline.config._

  //Retrieve the DecoderService instance
  val decoderService = pipeline.service(classOf[DecoderService])

    //Specify the instruction default value when instruction are decoded
    decoderService.addDefault(IS_MEA_INST, False)
    decoderService.addDefault(IS_DLD_INST, False)
    decoderService.addDefault(IS_DED_INST, False)
    decoderService.addDefault(IS_DEX_INST, False)

 //Specify the instruction decoding which should be applied when the
instruction match the 'key' pattern

    //Instruction decoding for measure instruction
    decoderService.add(
      key = M"00000000000000000000000000001011",
      // Decoding specification when the 'key' pattern is recognized in the
instruction
      List(
        IS_MEA_INST            -> True
       )
    )
    //Instruction decoding for deadload instruction
    decoderService.add(
      key = M"000000000000-----000000000101011",
      List(
        IS_DLD_INST             -> True,
        REGFILE_WRITE_VALID     -> True, //Enable the register file write
        RS1_USE                 -> True  //Notify the hazard management unit
that this instruction uses the RS1 value
      )
    )

    //Instruction decoding for dead instruction
    decoderService.add(
      key = M"00000000000000000000000001011011",
      List(
        IS_DED_INST             -> True
      )
    )

    //Instruction decoding for deadexcep instruction
    decoderService.add(
      key = M"00000000000000000000000001111011",
      List(
        IS_DEX_INST             -> True
      )
    )

//Intiating Exception Service for an exception to be emitted during
execution
    val exceptionService = pipeline.service(classOf[ExceptionService])
    deadExceptionPort = exceptionService.newExceptionPort(pipeline.execute)
  }

    override def build(pipeline: VexRiscv): Unit = {
    import pipeline._
    import pipeline.config._
    import spinal.lib.com.uart._

    MEA  = out UInt(32 bits) setName("MEA")
    uart = master(Uart())    setName("DEAD")

    // Stream that is being used as interface to send data
    // to plugin's own UART. Of special importance here are
    // the fields "payload" and "valid" to transmit data.
    val write = Stream(Bits(8 bits))

    //Initiating registers which are used during execution
    val Cycle_old   = Reg(UInt(32 bits)) init(0)
    val Clks        = Reg(UInt(32 bits)) init(0)
    val deadline    = Reg(UInt(32 bits)) init(0)
    val verify      = Reg(UInt(32 bits)) init(0)

    //Cycle Counter value which comes from CustomCsrDemoPlugin.scala file
val Cycle_new  =
pipeline.service(classOf[CustomCsrDemoPlugin]).cycleCounter
```

```
    //Add a new scope on the execute stage (used to give a name to signals)
    execute plug new Area {
      // UART is not sending any values unless MEA_INST es executed.
      // The default outside a MEA_INST call is to do nothing
      write.valid := False
      write.payload := 0

      when(execute.input(IS_MEA_INST) && execute.arbitration.isFiring) {
        //Executing a measure instruction. Computing number of cycles since
        // last call and transferring the result through UART by accessing
        // the connected Stream "write"

        // Computing the number of cycles since last MEA_INST
        Clks := Cycle_new - Cycle_old

         // Remember the current time for next call
        Cycle_old := Cycle_new

        // Produce some debug information to be visible at user LED
        verify := verify + 1

        // Send the least significant 8 bit of counter to DEAD UART
        write.valid := True
        write.payload :=  Clks.asBits(7 downto 0)
      }

  //Executing a deadload instruction.
 //Adding the current clock cycle value with the desired clock cycles delay
 //received from the user and storing in deadline variable
      val rs1 = execute.input(RS1).asUInt
      when(execute.input(IS_DLD_INST) && execute.arbitration.isFiring) {
        deadline := rs1 + Cycle_new
      }

 //Executing a dead instruction
 //Halting the instruction and doing nothing if current cycle counter value
 //is less than or equal to the deadline variable computed in deadload
instruction
      when(execute.input(IS_DED_INST) && execute.arbitration.isValid) {
        when(Cycle_new <= deadline) {
          execute.arbitration.haltItself := True
          verify := 5
        } otherwise {
          execute.arbitration.haltItself := False
          verify := 6
        }
      }

  //Executing a deadexcep instruction
 //Halting the instruction and doing nothing if current cycle counter value
 //is less than or equal to the deadline variable computed in deadload
instruction and
//emitting exception if current cycle counter value is more than the
deadline variable.
      when(execute.input(IS_DEX_INST) && execute.arbitration.isValid){
        when(Cycle_new <= deadline) {
          execute.arbitration.haltItself := True
          verify := 5
        } otherwise {
          execute.arbitration.haltItself := False
          verify := 6
        }
      }

   //Define the valid signal of exception port with reason for exception
        deadExceptionPort.valid   := (execute.input(IS_DEX_INST) &&
execute.arbitration.isValid && (Cycle_new > deadline))

        //Write the exception code into mcause register
        deadExceptionPort.code    := 24

//Write the instruction responsible for exception into mbadAddr register in
decimal form.
//Here, the instruction responsible is 00000000000000000000000001111011
        deadExceptionPort.badAddr := 123

//Produce some debug information to be visible at user LED to verify the
halt
      MEA := verify
    }

    // Instantiating and configuring a plugin dedicated UART
    val uartCtrl = new UartCtrl()
    uartCtrl.io.config.setClockDivider(115.2 kHz)
    uartCtrl.io.config.frame.dataLength := 7  //8 bits
    uartCtrl.io.config.frame.parity := UartParityType.NONE
    uartCtrl.io.config.frame.stop := UartStopType.ONE
    uartCtrl.io.writeBreak := False
    // Connect UART to outside of plugin
    uartCtrl.io.uart <> uart
    // Connect UART to inside of plugin
    // Write is driven by the MEA_INST instruction, only.
    // Adding a buffer to write: write >-> uartCtrl.io.write
    write.queue(16) >> uartCtrl.io.write
  }
}
```

2. Cycle Counter_Plugin :

```
package vexriscv.demo

import spinal.core._
import spinal.lib.io.TriStateArray
import spinal.lib.{Flow, master}
import vexriscv.plugin.{CsrInterface, Plugin}
import vexriscv.{DecoderService, Stageable, VexRiscv}

class CustomCsrDemoPlugin extends Plugin[VexRiscv]{

    var cycleCounter : UInt = null

  override def setup(pipeline: VexRiscv): Unit = {
    cycleCounter = Reg(UInt(32 bits))
  }

  override def build(pipeline: VexRiscv): Unit = {
    import pipeline._
    import pipeline.config._

    pipeline plug new Area{
      val prescaler = Reg(UInt(7 bits))

      prescaler := prescaler + 1
      when (prescaler === 99){
      prescaler := 0
      cycleCounter := cycleCounter + 1
      }
    }
  }
}
```