

**A PROJECT REPORT
ON
“HARDWARE RISC-V PROCESSOR LEARNING
TOOL”**

**Submitted to
National University of Ireland Galway**

**In Partial Fulfilment of the Requirement for the Award of

BACHELOR’S DEGREE IN
Electronic and Computer Engineering**

BY

Joseph Clancy 15410758

**UNDER THE GUIDANCE OF
Dr. Fearghal Morgan**

**DEPARTMENT OF ELECTRONIC AND ELECTRICAL
ENGINEERING
COLLEGE OF ENGINEERING AND INFORMATICS
LOCATION IN GALWAY CITY, CO. GALWAY - H91 HX31
2018-2019**

**AFFILIATED TO
National University of Ireland Galway**

**A PROJECT REPORT
ON
“HARDWARE RISC-V PROCESSOR LEARNING TOOL”**

**Submitted to
National University of Ireland Galway**

In Partial Fulfilment of the Requirement for the Award of

**BACHELOR’S DEGREE IN
COMPUTER ENGINEERING**

BY

Joseph Clancy 15410758

**UNDER THE GUIDANCE OF
Dr. Fearghal Morgan**



**DEPARTMENT OF ELECTRONIC AND ELECTRICAL
ENGINEERING
COLLEGE OF ENGINEERING AND INFORMATICS
LOCATION IN GALWAY CITY, CO. GALWAY - H91 HX31
2018-2019**

AFFILIATED TO



National University of Ireland Galway

ABSTRACT

This report details a hardware RISC-V processor core implemented in VHDL for FPGAs, an embedded application design flow, a series of online lessons introducing RISC-V on a Xilinx ZYNQ-7020 FPGA SoC in the cloud, and the design of a browser based IDE for RISC-V. The aim of this project is to provide a facility for training and assessment in the areas of processor design, digital systems, assembly programming, compiler design, application development. As well is the aim to contribute to the adoption of RISC-V globally. An existing RISC-V architecture was reviewed and an updated RISC-V processor was developed in VHDL using design methodologies that render the processor easy to understand and available for modification in the interest of exploration of computer architecture design and performance. Existing RISC-V application development tools (C Compilers) were reviewed and application development flow was created. Online lesson prototypes were created via viciLogic's course creator tool suite. The design of an online RISC-V IDE was performed. The project was conceived to enhance existing course material available at the university in the areas of digital design and computer architecture by taking advantage of a newly available open source resource the RISC-V ISA and building on an existing platform in use in the university, vicilogic.

Keywords: RISC-V, Computer Architecture, Digital Systems, Online Lesson, VHDL, vicilogic

I declare that this thesis is my original work except where stated Date

Signature

This declaration constitutes an assertion that full and accurate references and citations have been included for all material, directly included and indirectly contributing to, the thesis.

Acknowledgements

I would like to express my profound appreciation to **Dr. Fearghal Morgan** for his valuable and constructive suggestions during the planning and development of this project. His willingness to give his time and resources so generously and unceasing faith in my capabilities have been very much appreciated.

I would like to thank **Mr. Arthur Beretta** for his excellent work with the RISC-V core he developed during his time at N.U.I. Galway.

I would like to offer my special thanks to **Mr. Frank Callaly** for his guidance through his work on Vicilogic, giving me an insight on how to marry my vision with his reality.

I would like to offer my special thanks to **Mr. Jim Wilson** for his support with the RISC-V C Compilers used in this project

I would like to offer my special thanks to **Mr. David McInerney** for sharing his 10yr+ experience in software engineering and web development, regardless of the time of day or night.

At last I must express my sincere heartfelt gratitude to all the staff members of Electronic and Electrical Engineering Department who helped me directly or indirectly during this course of work.

Joseph Clancy

Contents

List of Figures	iii
Glossary	vii
1 Introduction	2
1.1 Project Introduction	2
1.2 Project Planning	4
2 Background	7
2.1 RISC-V	7
2.1.1 RISC-V Architecture	8
2.1.2 Programmer's Model/ Base Integer ISA	10
2.2 viciLogic	12
2.2.1 viciLearn	12
2.2.2 viciLab	14
2.3 VHDL	15
2.3.1 History	15
2.3.2 Design	16
2.3.3 Simulation	17
2.3.4 Synthesis	18
2.4 Previous work	20
2.4.1 Single Cycle Computer (SCC)	20
2.4.2 SCC RISC-V	23
3 RISC-V Processor Core	25
3.1 Design	25
3.1.1 Purpose/ Design Specification	25

3.1.2	Instruction Set Architecture	29
3.1.3	Processor Architecture	31
3.2	Implementation and Testing	51
3.2.1	Implementation	52
3.2.2	Testing	58
4	Learning Tools	62
4.1	Compiler/ Vivado Project	62
4.2	ViciLearn Lessons	67
4.3	RISC-V IDE Design	72
5	Health and Safety	78
5.1	Risk to Life and Limb	78
5.2	Broader Societal Impact	78
6	Conclusion and Future Scope	80
6.1	Conclusion	80
6.2	Future Scope	82
6.2.1	RISC-V Core	82
6.2.2	viciLogic Integration	84
	References	85
	Appendices	89
	Appendix A.1 GitHub Repository	90
	Appendix A.2 Standard Operating Procedures	91
	Appendix A.3 Risk Assessment	94

List of Figures

1.1	Final Year Project Context Diagram	3
1.2	Project Gantt Chart	5
2.1	RISC-V Opcode Table	9
2.2	RISC-V Registers	10
2.3	Base Integer Instruction Set	11
2.4	viciLearn lesson on a 1-bit register device	13
2.5	viciLab project - A 2-to-1 multiplexer	14
2.6	Example of a D-Flip-Flop described in VHDL	16
2.7	Example of a waveform produced by a simulation tool	18
2.8	Section of the RISC-V processor core's synthesised schematic	19
2.9	SCC Context Diagram	20
2.10	SFR map of the SCC	22
2.11	Programmer's model of the SCC	23
2.12	SCC inspired RISC-V processor architecture diagram	24
3.1	3-piece model of a processor	26
3.2	Practical model of a processor	27
3.3	Von-Neumann vs. Harvard Architectures	32
3.4	Classic RISC 5-stage pipeline	35
3.5	Architecture used for the processor design during this project	36
3.6	Instruction Fetch Module Design	38
3.7	Instruction Decode Module Design	40
3.8	RISC-V 32-bit instruction formats	41
3.9	Execution Module Design	43
3.10	Memory Management Module Design	47
3.11	Write Back Module Design	49
3.12	Register Bank Module Design	50

3.13	Screenshot of IF and ID instantiations in RISC-V_Top . . .	52
3.14	Screenshot of the program counter the Instruction Fetch module	53
3.15	Screenshot of decode logic in the Instruction Decode module	54
3.16	Screenshot of the ALU in the Execution Unit	55
3.17	Screenshot of the Branch Detection Unit in the Execution Unit	55
3.18	Screenshot of a section of the Memory Management Module	56
3.19	Screenshot of the "wb_sel" generation logic implementation	57
3.20	Screenshot of the Write Back multiplexer implementation .	57
3.21	Screenshot of the implementation of the registers	58
3.22	Screenshot of the implementation of the write logic for the registers	58
3.23	Screenshot of the implementation of the read logic of the registers	58
3.24	Screenshot of the testbench for the Instruction Decoder . .	59
3.25	Screenshot of the waveform generated by the simulation of the testbench in figure 3.24	60
3.26	Screenshot of the VHDL formatted hexadecimal representation of the program in the testbench	60
3.27	Screenshot of the top level testbench code that runs the processor	61
4.1	Screenshot of a simple C program	63
4.2	Screenshot of the assembly program produced by the RISC-V C Compilers	64
4.3	Screenshot of the assembly program produced by the RISC-V Assembler and Linker	65
4.4	Screenshot of the VHDL formatted hexadecimal representation of the program	66
4.5	Screenshot of the VHDL formatted hexadecimal representation of the program in the testbench	67
4.6	Screenshot of the introduction to the RISC-V prototype course	68
4.7	Screenshot of the 2-instruction program lesson	69

4.8	Screenshot of the step that provides a sandbox for users . .	70
4.9	Screenshot of the step that explains the architecture of the processor	70
4.10	Screenshot of the step that explains the design of the Control and Data Path	71
4.11	Screenshot of the step that explains the design of the Execution Unit	72
4.12	Mock-up of the RISC-V IDE Editor Tab	74
4.13	Mock-up of the RISC-V IDE Programmer's Model Tab . .	75
4.14	Mock-up of the RISC-V IDE Data Memory Tab	76
4.15	Mock-up of the RISC-V IDE Processor View Tab	77

Glossary

- **Instruction Set** - A list of the operations a processor can perform e.g. addition, subtraction, multiplication, comparisons etc.
- **RISC** - Reduced Instruction Set Computer, a processor designed with an instruction set where each instruction completes a simple operation, allowing for simple hardware implementation and higher operation speeds. Also characterised by a load/store architecture, meaning memory can only be accessed via special instructions i.e., load and store, and consequently the operands to instructions are always the internal registers featured in the system
- **IC** - Integrated Circuit, digital or analogue circuits implemented on silicon chips.
- **SoC** - System on a Chip, this is the acronym given to an IC that contains a processor core but also many other functions, essentially rendering it equivalent to an entire conventional computer which is smaller and contained on one silicon chip.
- **BRAM** - Block Random Access Memory, blocks of high-speed configurable memory featured in Xilinx FPGAs.
- **FPGA** - Field Programmable Gate Array, an IC featuring configurable blocks of circuitry and specialised digital systems such as; digital signal processing blocks, BRAM etc. This device allows users to implement any design on hardware without having to manufacture the circuit.
- **ISA** - Instruction Set Architecture, the abstract model of the processor, it contains all the instructions that a computer can process and all of the

conventions that software must follow. It's the connection between the software and the hardware.

- **CISC** - Complex Instruction Set Computer, the counterpart of RISC, these machines feature instructions that can do very complex operations in one instruction, such as an if else statement that accesses main memory, in RISC such an operation would require many instructions.
- **SCC** - Single Cycle Computer, an existing single cycle processor in the university.
- **HDL** - Hardware Description Language, a specialised computer language for designing and implementing digital or mixed signal integrated circuits.
- **VHDL** - VHSIC Hardware Description Language, a programming language designed to be used for the prototyping and synthesis of digital systems.
- **VHSIC** - Very High Speed Integrated Circuit, a term given to ICs of severe size or complexity or speed.
- **ALU** - Arithmetic Logic Unit, the circuitry in a processor that handles arithmetic and logic operations.
- **ASIC** - Application Specific Integrated Circuit, an IC that has been created for a specific purpose.
- **Turing Complete** - The phrase used to describe a machine that can simulate a Turing Machine and is therefore itself a Turing Machine
- **Turing Machine** - The phrase used to describe a machine that can calculate anything that is calculable and compute anything that is computable.

Chapter 1

Introduction

1.1 Project Introduction

The final year project that this report details (see figure 1.1), was spawned from the possibility of expanding the university's ability to educate and assess in the areas of digital system design, computer architecture, processor design and embedded application development by taking advantage of the new open-source RISC-V [14] instruction set architecture (ISA), and an existing learning platform in the university, viciLogic [32]. This project is the design and implementation of a 32-bit RISC-V processor core learning tool, the generation of an application development flow by reviewing the RISC-V GNU C compilers [15], the presentation of this learning tool by creating online lessons via viciLogic's online course creation tool and the design of an online RISC-V IDE for viciLogic. The RISC-V processor core designed during this project is a single cycle implementation (instructions execute in one clock cycle), similar to existing processors in the university. However the architecture being used differs by following the RISC-V methodologies. RISC-V is designed to naturally complement a 5-stage architecture [22], [41], though it is not mandatory for a RISC-V processor to have a 5-stage architecture. This architecture is useful as it lends itself to easy modification to a pipe-lined processor [23], [44] (a more complicated but performant processor design), and an intuitive understanding of the execution of an instruction. The 5-stage architecture explains the life cycle of an instruction in a very intuitive way. It naturally answers questions such as: How does the processor retrieve instructions? How are these decoded? How are they executed? How does the processor interact with data memory? The

modularity present in this dissection of the instruction life cycle is a fantastic learning tool, as each stage of the architecture can be treated as a module. The implementation of these modules can be hot swapped by alternate user designs. As an example, the ALU in the processor could be implemented in many ways, giving varying levels of performance when executing most if not all instructions. This modular facet allows users to explore, experiment with and analyse different implementations of the components contained within a processor.

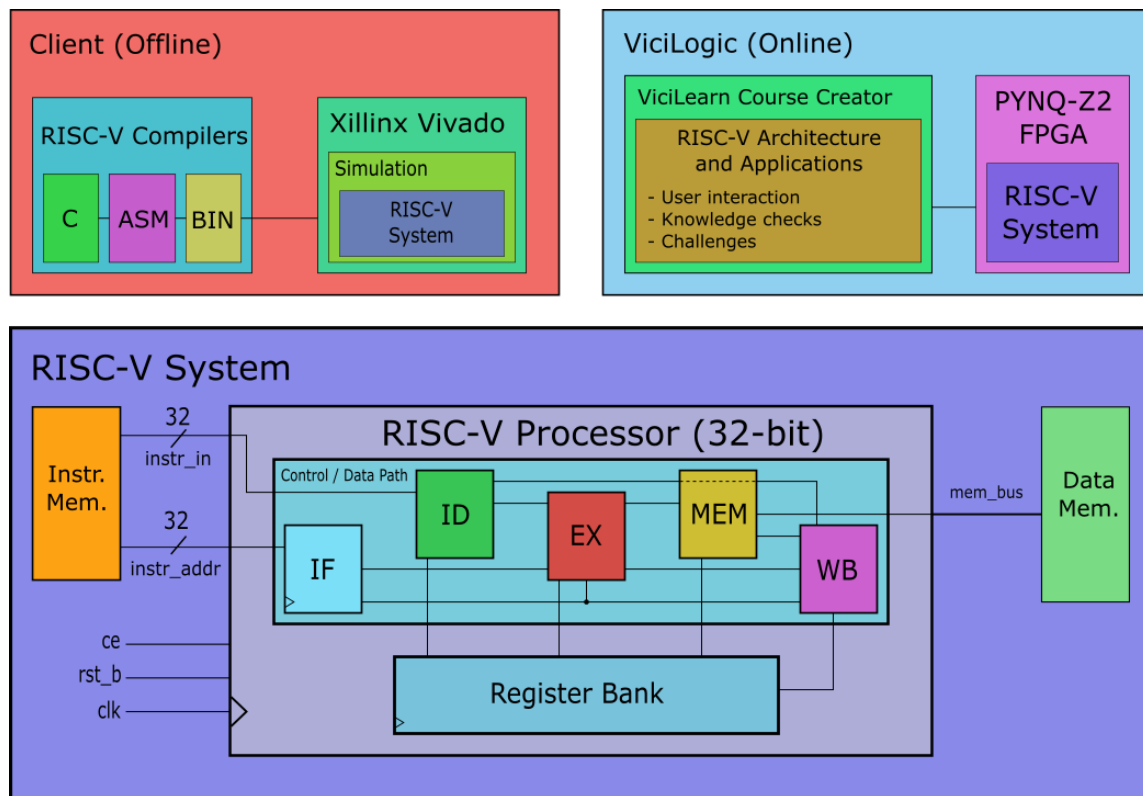


Figure 1.1: Final Year Project Context Diagram

The application development flow will require both the Vivado Design Suite [52] and RISC-V C compilers to be installed on a user's local machine (see upper left corner of figure 1.1). However, documentation on how to install the Vivado Design Suite is available online and compiler installation and use guides were created. These guides enable users to alter and simulate the RISC-V processor core which in turn allows users to fully investigate and understand the processor. With the RISC-V Compiler users can compile their C or RISC-V assembly [13] code into RISC-V binary code and initialise this into the processor's instruction memory in a simulation. Users also have the option of

uploading the design to their own FPGA and initialising the on-chip memories with their programs. A lesson on viciLogic is currently available [33](see upper right corner of figure 1.1), connecting the user to this RISC-V processor which is implemented in hardware on a Xilinx ZYNQ-7020 FPGA SoC [53] on a PYNQ-Z2 [51] board in the cloud as an example of how this learning tool could be used to train and assess. These lessons are also an example of how RISC-V and the processor will be integrated with viciLogic via a RISC-V course [33], which will train users on: the RISC-V instruction set architecture, processor design and application development. The design of an online IDE has also been performed to illustrate how the capabilities of this system can be expanded and improved in the future.

1.2 Project Planning

The plan for this project follows that of a typical digital IC design project. Therefore, it has typical phases like conception, planning, research, design, documentation, implementation, test and evaluation. This project was heavily researched based as to implement a RISC-V processor for learning applications a suitable architecture had to be researched and the RISC-V instruction set architecture had to be fully investigated before any hardware design or implementation could begin.

The associated deliverable for this project are:

- A set of VHDL models and testbenches i.e. the RISC-V processor
- An application development flow i.e. RISC-V compilers and their use
- An online lesson on viciLearn presenting the RISC-V processor

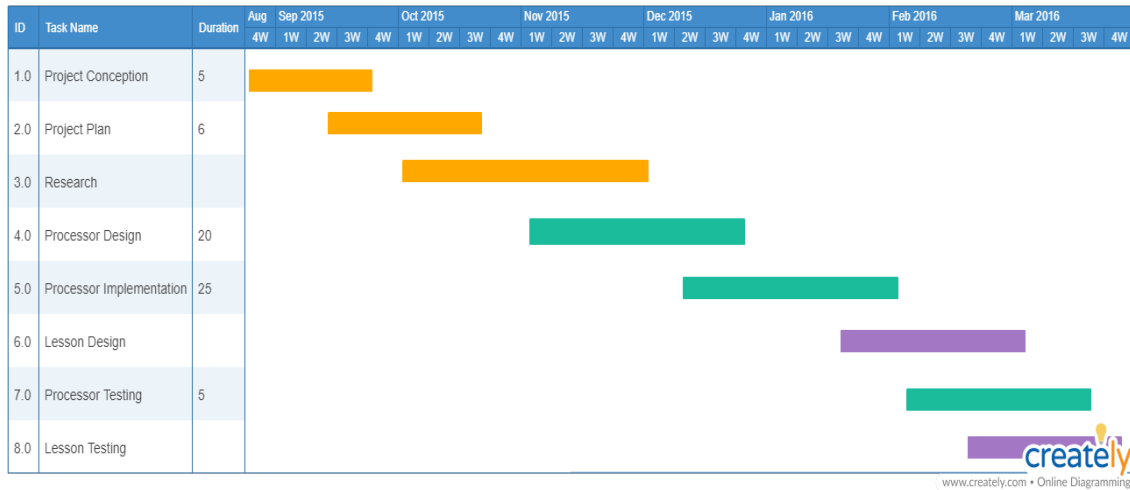


Figure 1.2: Project Gantt Chart

Figure 1.2 presents the plan for this project. Each of the main phases of the project are listed down the left side of the image. The calendar progresses to the right. The coloured bars represent the time spent on particular phase. The initial conception of the project mainly entailed feasibility research i.e. would this project be useful? Could it be completed in the time given? Has it already been done before? If so, how could it be done differently? The plan phase entailed detailing the resources needed to complete this project i.e. FPGAs, software tools etc. and synthesising a gantt chart similar to figure 1.2 to ensure that progress was being made throughout the course of the project. The research phase began during the planning, where-in RISC-V was researched in it's entirety and research into computer architectures and processor design and how it is taught was completed to pinpoint an architecture that would suit this application most efficiently. Once the project plan was finalised and a decent amount of research was completed, the processor design phase began, in this phase the logistics of creating the chosen architecture in VHDL was tackled. As the design began to mature the implementation of the processor began, once the processor reached a stage of maturity that meant programs were running successfully, the design of the lesson began. As the implementation of the processor came to an end, more formal tests were ran to ensure corner cases in the VHDL code were covered. Finally after the processor and lesson were tested the project was completed.

The remainder of this report will detail the background review of the project, the work completed throughout the course of this project, health and safety considerations and the future scope of the project. Chapter 2 details the background review of this project, including the RISC-V instruction set architecture (ISA), viciLogic, VHDL [2], [49] and the existing previous work in this area. Chapters 3 & 4 detail the work completed throughout the course of the project. Chapter 3 includes the design/ implementation/ testing on the RISC-V processor itself. Chapter 4 includes the creation of the offline and online learning tools i.e. the compilers and the viciLearn lesson. Chapter 5 & 6 detail the health and safety and societal impacts of the project and the future work that can be spawned from this project.

Chapter 2

Background

This chapter details the background review of the project: the RISC-V instruction set architecture, viciLogic, VHDL and previous work completed in this area.

2.1 RISC-V

RISC-V [14] (pronounced 'risk-five'), is an open-source RISC [46] ISA [8], [43] created in academia and research in the hopes of avoiding the problems associated with teaching computer architecture [22], [54] with the case studies of industry level processors. To access the instruction sets of proprietary industry level processors, large fees have to be paid and the permissions to acquire these instruction sets is difficult to attain as processor design companies wish to keep certain aspects of their designs secret. RISC-V was designed for universities and companies all over the globe to have a free, open-source but practical resource for teaching and producing processors.



2.1.1 RISC-V Architecture

The RISC-V ISA [16] comes in three flavours: 32-bit, 64-bit and in future 128-bit. RISC-V's main strength comes about in how it tackles future-proofing itself. It does this in two ways: by making itself applicable in many domains, and by using a modular architecture.

From the markets present to date, it can be seen that commercial ISAs are only popular in certain domains. It is well known that Intel and AMD dominate the desktop computer market with their CISC [42] based processors. These complex instructions are well fitted for such general-purpose applications like home computing, office work, film and media, and CAD (Computer Aided Design) design. Whereas ARM dominate the mobile and server markets, their RISC based processors are excellent at low power, low area applications. However, these commercial ISAs, even ARM's Thumb/Thumb-2 [47] RISC ISAs, are very complex. These companies have developed their ISAs by adding new features and expanding around their current ISA, to ensure that legacy processors support newer programs. This patch work type of development on the ISA is not sustainable, eventually the ISA becomes too large and complex to fix design flaws present from earlier design decisions. Historically its also well known that commercial ISAs tend to come and go, this is because as a commercial ISA loses its popularity, its parent company will reduce the amount of capital invested in this ISA. Due to the proprietary nature of these ISAs, interested third parties then cannot continue development.

RISC-V tackles all these problems by being open-source and by its modular design. RISC-V is in fact a set of instruction sets, referred to as extensions [16, pg.3]. However, one of the instruction sets is mandatory for all RISC-V processors to implement; this is the base integer instruction set, referenced as "I" [16, chap. 2]. The most basic RISC-V processors are described as RVI (RISC-V "I"), a 32-bit RISC-V processor would be described as RV32I. This base instruction set contains the bare minimum instructions required for a processor to be Turing Complete [17], [48]. This includes the most basic arithmetic and logic operations: addition, subtraction, AND, OR, XOR and shifting. It

also includes memory accessing operations: load and store, and operations to allow basic decisions to be made i.e., jumps/branches. The 64-bit variation offers 64-bit versions of these operations plus the existing 32-bit operations. All other types of instructions, such as, multiplication, atomic instructions [27], single/double/quad precision floating point (Note: RISC-V supports the IEEE Floating Point standard, [5]), compressed instructions, bit manipulation, vectors etc. are all examples of instruction sets classified as optional extensions. This means a particular RISC-V processor must implement "I", the base integer instruction set, however it does not need to but can implement the other instruction sets. Each extension is referenced as a letter i.e., the multiplication/division extension is referenced as "M" [16, chap. 6], the atomic extension is referenced as "A" [16, chap. 7], the single precision floating-point extension is referenced as "F" [16, chap. 7]. The optional extensions mentioned are examples of standard extensions, this means that these are instruction sets designed and reviewed by the RISC-V foundation. These extensions are guaranteed to not conflict with each other; however, the RISC-V opcode encoding space (see figure 2.1), leaves room for application specific or custom instruction sets for users to implement, these may conflict.

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

Figure 2.1: RISC-V Opcode Table

Figure 2.1 presents the RISC-V opcode space. The RISC-V ISA features a 7-bit opcode. Bits 0 and 1 of the opcode are always 1's for all instructions that are not compressed. Opcodes with bits 2,3,4 set are reserved for instruction lengths greater than 32 bits.

2.1.2 Programmer's Model/ Base Integer ISA

Programmer's Model

Figure 2.2 presents the registers available in RISC-V and their calling conventions. RISC-V defines 31 (x1-x31) general purpose XLEN registers, where XLEN is the data width of the processor i.e., 32-bit, 64-bit, 128-bit, and a 32nd register (x0) which is hardwired to zero/ground. Writing to this register does nothing and reading from it yields zero. Registers x1-x4 are used as pointers, registers x5-x7, x28-31 are used as temporary registers, registers x8-x9, x18-27 are used as saved registers and registers x10-17 are used as function arguments or return values. The programmer's model for RISC-V also includes a program counter of XLEN in length.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Figure 2.2: RISC-V Registers

Base Integer ISA

The base integer instruction set i.e "I" contains 37 instructions [16, chap. 2]. As previously mentioned this instruction set contains the minimum amount of instructions required for a processor to be Turing complete [17], [48]. These include: basic arithmetic and logic instructions, load and store instructions and control flow instructions i.e., jumps and conditional branches. See Figure 2.3 for the list of the instructions in the base integer instruction set.

RISC-V defines the familiar and expected instructions. The first of these are arithmetic and logic functions such as: addition, subtract, AND, OR, XOR and shifting [16, pg. 13-15]. These instructions apply their function to two values (Two registers or a register and an immediate value) and store their result in a destination register. It also defines control flow functions [16, pg. 15-18] such as jumping and conditional branches: branch if equal/ not equal/ less than etc., which all change the flow of the program either absolutely or based on a comparison. The last of the expected instructions are load and store functions [16, pg. 18-19], these are defined due to the RISC aspect of the architecture. However, the instructions the RISC-V base integer instruction set defines that are different are the "set less than" [16, pg. 13-15] and AUIPC/LUI [16, pg. 14] instructions. The "set less than" instructions compare two values, if the first value is less than the second value, "1" is written to a destination register, otherwise "0" is written to the destination register. The AUIPC/LUI instructions are used to insert immediate values into registers. AUIPC adds a 12-bit immediate value to the current program counter and stores the result to a register. LUI adds a 20-bit immediate value to the upper 20 bits of a register and sets the lower 12 bits to "0".

Category	Name	Fmt	RV32I Base
Loads	Load Byte	I	LB rd,rs1,imm
	Load Halfword	I	LH rd,rs1,imm
	Load Word	I	LW rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm
	Load Half Unsigned	I	LHU rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm
	Store Halfword	S	SH rs1,rs2,imm
	Store Word	S	SW rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm
	Add Upper Imm to PC	U	AUIPC rd,imm
Logical	XOR	R	XOR rd,rs1,rs2
	XOR Immediate	I	XORI rd,rs1,imm
	OR	R	OR rd,rs1,rs2
	OR Immediate	I	ORI rd,rs1,imm
	AND	R	AND rd,rs1,rs2
Compare	AND Immediate	I	ANDI rd,rs1,imm
	Set <	R	SLT rd,rs1,rs2
	Set < Immediate	I	SLTI rd,rs1,imm
	Set < Unsigned	R	SLTU rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm
Branches	Branch =	SB	BEQ rs1,rs2,imm
	Branch ≠	SB	BNE rs1,rs2,imm
	Branch <	SB	BLT rs1,rs2,imm
	Branch ≥	SB	BGE rs1,rs2,imm
	Branch < Unsigned	SB	BLTU rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm
Jump & Link	J&L	UJ	JAL rd,imm
	Jump & Link Register	UJ	JALR rd,rs1,imm

Figure 2.3: Base Integer Instruction Set

2.2 viciLogic

Currently in the university, there are efforts to provide an online learning platform that features course material on digital IC design and prototyping. The aim of this platform is to make industry level IC design accessible to everyone. This platform is called viciLogic [32]. It is composed of two main parts; viciLearn [31] and viciLab [39]. viciLearn contains the course material to be taught i.e., digital design [50], VHDL [2], [49] and use of development tools. viciLearn is a set of self-paced courses detailing the previously mentioned topics with animated views of digital circuits which are running on an FPGA SoC [51], [53] in the cloud [40]. viciLab is a prototyping tool that runs on the users local machine, written in Python, it allows the user to animate, control and demonstrate the users own hardware designs from a pre-existing Xilinx Vivado project in the cloud or using a local FPGA connected to the users machine.



2.2.1 viciLearn

viciLearn [31] currently provides self-paced lessons on; fundamental digital logic functions, Xilinx design tools, viciLab and digital system design. Figure 2.4 shows an example of one such lessons. viciLearn brings users closer to hardware by running the designs present in viciLearn on a Xilinx FPGA SoC in the cloud i.e., when the user sees the widgets on screen display some behaviour e.g., an LED toggles, the hardware running on the FGPA in the cloud generates this behaviour. viciLearn allows users to interact with this hardware via the widgets displayed.

HOME

COURSES

DIGITAL SYSTEMS DESIGN AND FPGA PROTOTYPING: FUNDAMENTALS, HDL AND EDA TOOLS

Contents

27%

Ask a Question

FD1C DIRECTED LESSON (CONTROL/OBSERVE FD1C BEHAVIOUR)

FD1C 1-bit register (D flip flop), with asynchronous rst

Component symbol

Row	rst	clk	D	Q
0	1	X	X	0 (reset)
1	0	1	0	0
2	0	0 or 1	X	Q (no change)

Function Table

clk

 System clock strobe, rising edge active

rst

 Asynchronous reset signal. Assertion clears register

D

 Flip Flop input data

Q

 Flip Flop output data

Signal Data Dictionary

Timing Diagram

STEP TIMING DIAGRAM | CLEAR | PRINT

FD1C directed lesson

Restart directed lesson

- This step directs you to
 - control the FD1C component input signals
 - observe the FD1C behaviour
- Click [Restart directed lesson](#) to repeat the directed lesson steps

CONTINUE

Figure 2.4: viciLearn lesson on a 1-bit register device

In figure 2.4, there is a lesson demonstrating a 1-bit register i.e. a D-Flip-Flop. This is a memory circuit. When the "clk" signal is toggled, the "Q" output will take the value of the "D" input. If "rst" is asserted, the "Q" output will return to "0". In this lesson, users are prompted to assert/de-assert/toggle the: "D", "clk", "rst" inputs signals. When users change the input signals, they are actually altering the inputs to physical circuitry contained on the FPGA SoCs. Users can then observe the behaviour exhibited by the circuitry via the output widgets i.e. the LEDs and the timing waveform which is generated as the user "plays" with the device.

In addition to allowing the user to directly access the circuitry they are viewing, viciLearn also provides "ask a question" and knowledge check facilities for users. If users are unclear on some aspect of the course material, they can submit a question and the viciLogic team will respond to the question via the user's email. Knowledge checks are provided to aid users in the retaining information presented in a lesson. Knowledge checks come in the form of predicting the output of the circuitry presented given certain inputs, this will require users to input binary or hexadecimal values to progress in the lesson.

2.2.2 viciLab

viciLab provides a facility for users to take their digital system designs and bring them to life via a graphical user interface (GUI) creator. viciLab connects to an FPGA in the cloud or to an FPGA directly connected to the user's local machine and then uploads the user's design to the FPGA. From here viciLab behaves as the bridge between the user's GUI and the design on the FPGA. This facility requires user designs to have been written in a hardware description language (HDL) e.g. VHDL, Verilog[3], System Verilog [4]. These HDL code models must then be compiled together into a Xilinx Vivado project. Vivado contains the facilities to compile the HDL code and prepare it for Xilinx FPGA devices.

Similar to viciLearn, when the user interacts with the widgets that represent inputs to the design i.e., switches, toggles, input boxes, viciLab will alter the physical inputs of the design in the FPGA. Conversely, when the design's outputs exhibit behaviour, viciLab will interpret this and display this behaviour to the user via the widgets that represent outputs i.e. LEDs, etc. This results in users having the ability to take their designs and demonstrate or test them in a way HDL model simulations cannot achieve. Figure 2.5 presents the example of a 2-to-1 multiplexer device that has had a viciLab GUI created. A multiplexer is a selection device. There are two inputs, "muxIn1" and "muxIn0". "sel" selects which of the two inputs "muxOut" will take the value of. Here the user can assert/de-assert the input switches and observe the behaviour of the design as it runs in hardware via the "muxOut" LED.

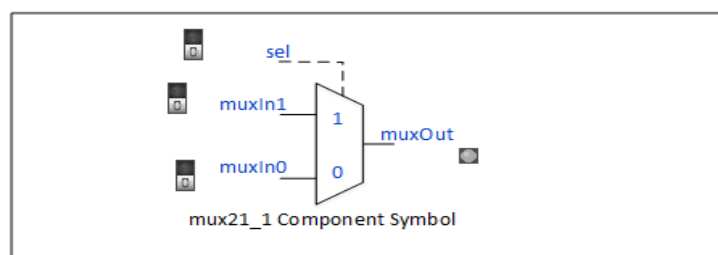


Figure 2.5: viciLab project - A 2-to-1 multiplexer

2.3 VHDL

VHDL (VHSIC Hardware Description Language) [2], [49] is a specialised programming language known as a HDL, which is used to describe digital and mixed signal circuits. VHDL is also a general purpose parallel programming language but it is rarely used as such.

2.3.1 History

In 1983, VHDL was developed at the request of the United States Department of Defence as a mechanism to ensure that the ASICs being used by the Department of Defence were of sufficient quality i.e. military grade. VHDL provides a facility to ensure designs are of sufficient quality by allowing designers to first simulate the ASICs being produced. The practical operation of a device could be tested without having to spend capital on the expensive process of producing an ASIC.

VHDL is very heavily based on a programming language known as Ada [29]. At the time of VHDL's development, Ada had been well tested by the Department of Defence and in order to not re-invent the wheel, VHDL heavily borrowed its underlying concepts and syntax from Ada. This characteristic of VHDL plays in its favour. Ada is known to be a very verbose programming language, this means that the programmer/designer is required to write more code to complete the same functionality as other programming languages. In VHDL this verbosity requirement forces better design creation and fewer mistakes. This is due to the designer being forced to consider a lot more when writing their models. For example, when using any HDL to describe a digital circuits function, the designer should define how the outputs of the circuit behave in response to every possible input. VHDL is designed such that the code is not considered valid and EDA (Electronic Design Automation) tools will not compile the design unless this requirement is met. Whereas in Verilog, another HDL, it is not enforced to do this, this can result in pieces of circuitry required in a design not being created at all.

VHDL is also an IEEE standard [2], this defines a published document that details specifications and procedures that describe VHDL and its uses to maximise its reliability. In being an IEEE standard, this defines an acceptance suite [1] for VHDL, meaning if any EDA tool vendors wish to provide VHDL support, they must first meet the acceptance suite requirements before they can market any products that claim to support VHDL.

2.3.2 Design

As with any HDL, VHDL is used to write text models of digital and mixed signal circuitry. To tackle the dilemma of modelling physical circuit behaviour, VHDL, and HDLs in general, utilise the concept of parallel or concurrent execution. This is a result of fact that conventionally computers execute code sequentially, one instruction after another. This however is not how physical circuitry behaves, electronic components do not wait for other components to finish what they are doing, electronic components behave in real time. VHDL provides a programming construct known as a "process" to deal with this. The circuitry a designer wants to model is described within a "process" construct. VHDL demands that designers use separate "process" constructs to define pieces of circuitry that operate in parallel or independently i.e., when a VHDL model is simulated, all of the processes defined execute at the same time, not sequentially. This allows designers to create models of physical circuits while retaining the correct behaviour. Figure 2.6 presents a VHDL model of the D-Flip Flop device in figure 2.5.

```
1  D_Flip_Flop : process(clk, rst) is
2  begin
3      if rst = '1' then
4          Q <= '0';
5      elsif rising_edge(clk) then
6          Q <= D;
7      end if;
8  end process D_Flip_Flop;
```

Figure 2.6: Example of a D-Flip-Flop described in VHDL

Lines 1, 2 & 8 define the process construct in figure 2.6. The process has been given the name "D_Flip_Flop", however any alias that the designer wishes can be used here. "process(clk, rst)" is the syntax used by VHDL to communicate to the EDA tools what input signals should activate the circuits behaviour. In this example, the output "Q" of the D-Flip-Flop will not change unless the "clk" or "rst" signals have been asserted, so therefore these inputs must be placed within the parenthesis of the "process()" syntax. On line 3, it can be seen that the "rst" signal takes priority in this device, meaning that if "rst" is asserted and the "clk" signal is asserted, the device will respond to the "rst" signal only and the output "Q" will be set to "0". Note: "<=" is the VHDL syntax for "the signal on the left will take the value on the right". On line 5, it can be seen that if the "rst" signal is not asserted, then this device checks if "clk" is being asserted i.e. is "clk" rising from "0" to "1". If so, then "Q" will take the value of the input "D". If neither "rst" nor "clk" are asserted, then the process will not execute.

2.3.3 Simulation

These text models of the circuitry can be dealt with in two ways, they can be processed via a simulation program or a synthesis program. Design convention dictates that designers should simulate to locate any errors or bugs in the design before they synthesise but it is not mandatory.

To simulate designs, a designer must create a special type of HDL model called a *testbench* [6]. This is a HDL model that contains the design to be tested and simulation models of the circuits that the design would be connected to i.e. to test the possible types of inputs the design might receive. The testbench is then passed to the simulation program. This program is event-based. This means that the behaviour of the circuitry is treated as a series of events, where each event is placed onto a queue. For example, if a signal needs to be asserted after 1 millisecond, then it will be placed onto the queue for +1ms time. This simulation tool will produce an output waveform similar to figure 2.7. This waveform will display the behaviour of a signal for the entire duration of the simulation. Therefore the designer can view the behaviour of all signals

in the design throughout the course of an entire simulation. Figure 2.7 displays a section of the waveform produced for the RISC-V processor core designed during this project. It can be seen that the name of the signals that were desired to be viewed are listed along the left column. The time axis progresses to the right as can be seen by the incrementing time values along the top of the figure. The values of a particular signal can then be seen varying as time progresses.

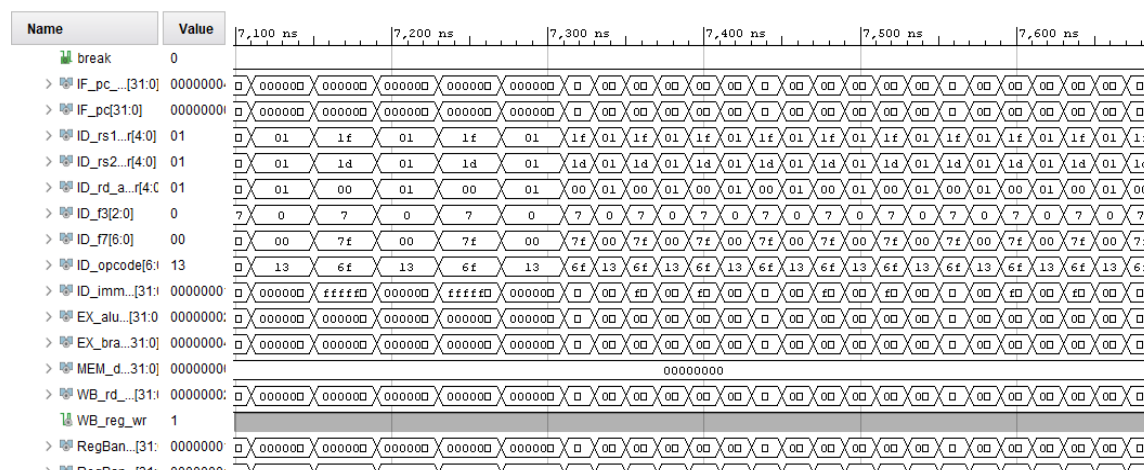


Figure 2.7: Example of a waveform produced by a simulation tool

Designers can use these waveform to verify that their HDL designs are functioning correctly by ensuring that all signals in the design are behaving as expected throughout the course of a simulation. This technique is vital in producing better ASICs as it saves monumental amounts of capital to handle errors and bugs at this stage rather than when the ASIC has been manufactured. It should be noted that VHDL also has file input and output capabilities so designers can input stimulus to the simulation or record simulation results to a file for later use.

2.3.4 Synthesis

When designers are sure that their design functions correctly, the HDL designs are then passed to a synthesis program [11]. This synthesis program processes the HDL design which is mapped into actual digital circuitry i.e. for a particular FPGA or a particular ASIC process node such as 14nm. This synthesis program will produce a schematic for the design in the form of the particular

technology being used. For example, if the design is being implemented on an FPGA, the design schematic will contain the circuit elements in an FPGA, usually LUTs (Look Up Tables) , multiplexers and D-Flip-Flops. If the design is being implemented on an ASIC, the design schematic will present more familiar circuitry such as logic gates and D-flip-flops. Figure 2.8 presents a section of the synthesised schematic for the RISC-V processor core designed during this project. A series of D-Flip-flops (FDCE) and LUTs can be seen.

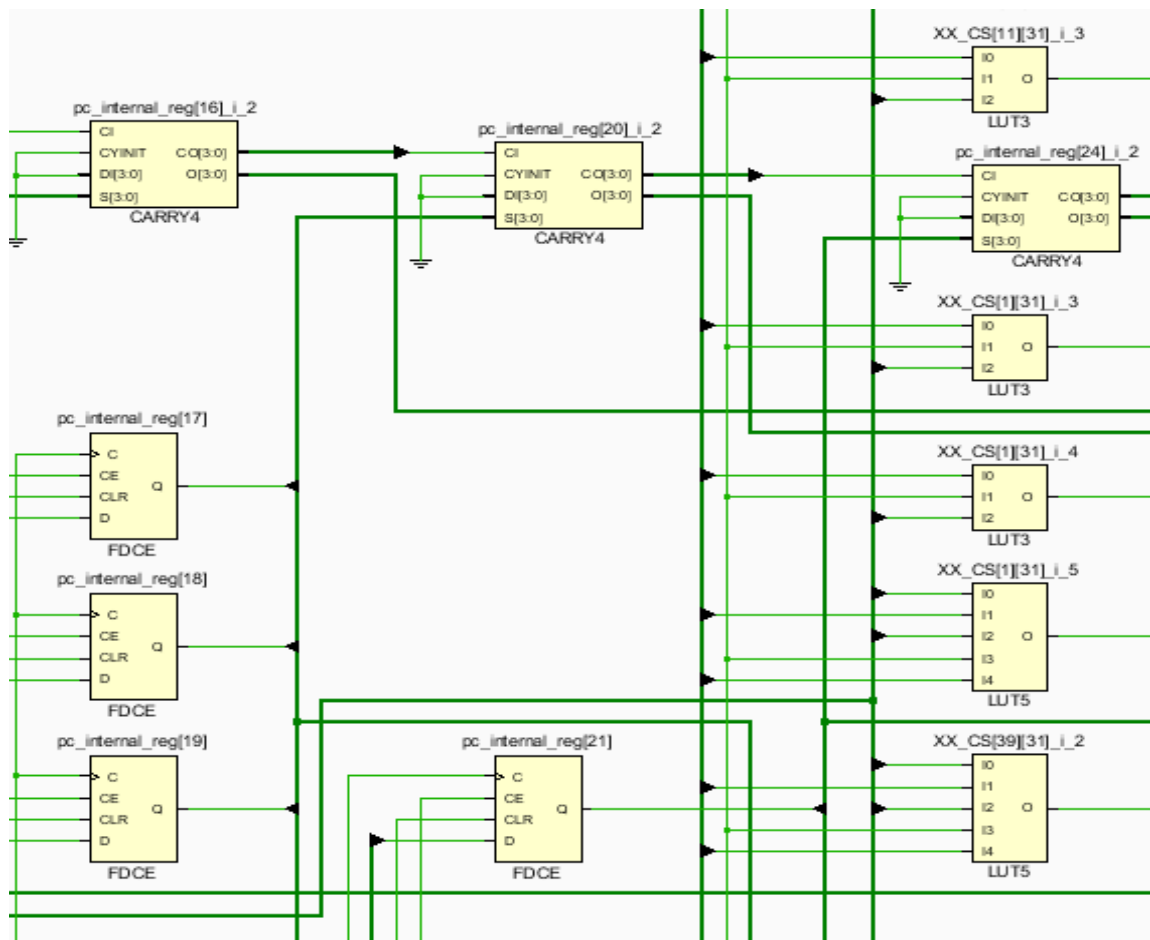


Figure 2.8: Section of the RISC-V processor core's synthesised schematic

2.4 Previous work

2.4.1 Single Cycle Computer (SCC)

Overview

There exists in the university a custom 16-bit RISC single cycle processor [36], [37] as part of the current course material at the university in digital design and computer architecture. This system features 8 general purpose 16-bit registers and 16 special purpose 16-bit registers. It also features an instruction set [38] with 47 instructions that encapsulates; all instructions needed to be Turing complete (arithmetic, logic, load/store, control flow), multiplication instructions, stack manipulation instructions and interrupt handling instructions. The instruction set listing is available online. This system features extensive documentation, compilers and supporting applications to develop on and alter the single cycle processor. Figure 2.9 presents the top level context diagram of the SCC. The SCC can be seen with it's associated register banks exposed. The SCC connects to two memories. One stores user programs i.e. instruction memory, and the other stores the data to be operated on during the execution of a program i.e. data memory.

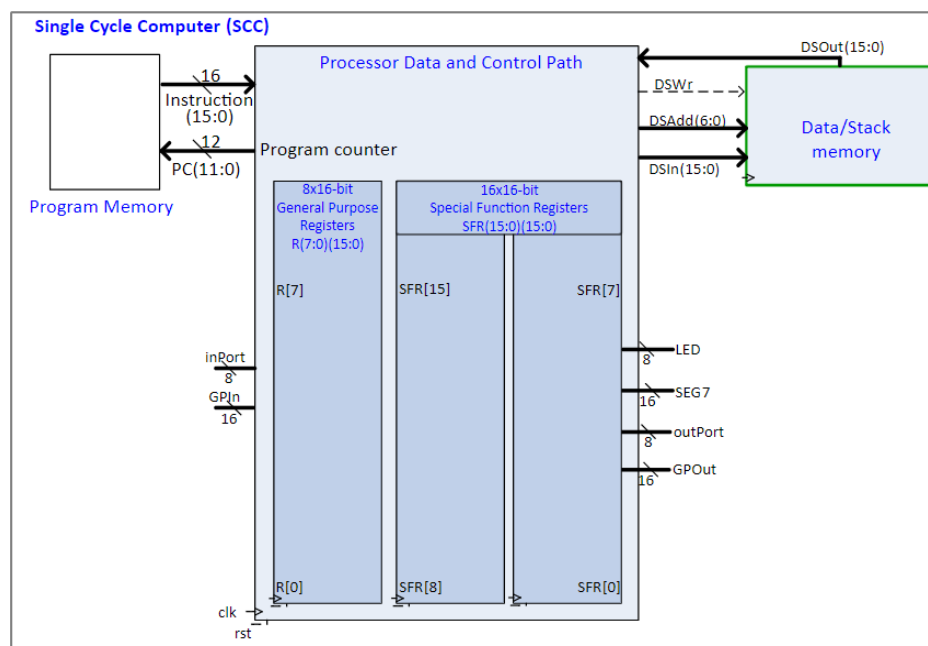


Figure 2.9: SCC Context Diagram

Special Function Registers (SFRs)

The SCC features 16 16-bit SFRs. These registers are used to provide extra functionality that is convenient to a programmer. Figure 2.10 displays the SFR map for the SCC. The registers in this map are as follows:

- **SFR0** - This SFR contains all of the flags the SCC contains. these include carry flags, done flags and enable bits for the SCC's interrupts and interrupt timer.
- **SFR1-2** - These SFRs contain the lower and upper 16-bits of a 32-bit timer that SCC programmers can use to trigger periodic interrupts.
- **SFR3** - This SFR contains the data and stack pointers which the SCC's stack related instructions rely on.
- **SFR4-5** - These SFRs are connected to 4 seven segment displays (SFR4) and 8 LEDs (SFR3). Whatever values are placed in these registers will appear on the output of the displays and LEDs i.e. if "BEEF" in hexadecimal or "1011111011101111" in binary is place in SFR4, then "BEEF" will appear on the seven segment displays.
- **SFR6-7** - These SFRs are connected to the general purpose 16-bit input and output ports featured on the SCC. Programmers can use these as generic ways of inputting and outputting data to and from the SCC.
- **SFR8-9** - These SFRs are used as re-load values for the timers in SFR1-2. If SFR0 is configured to reload the timers, each time the value in SFR1-2 reaches 0, the values of SFR8-9 will be stored in SFR1-2.
- **SFR10-11** - These SFRs are used to store the multiplier and multiplicand of any multiplication operation the SCC executes. If a programmer wishes to multiply two numbers, they must first be placed in these SFRs before the multiply instruction is executed.
- **SFR12** - This SFR is connected to the 8-bit "inPort" (upper 8-bits of the SFR) and to the 8-bit "outPort2 (lower 8-bits of the SFR) of the SCC. This SFR allows programmers to take user input for smaller input requirements i.e. 4-bits of input for directions: up, down, left, right.

- **SFR13** - This SFR contains the maximum stack pointer value to date, this value is used by programmers to keep track of the size of the stack. If the stack becomes too large, it may encroach on other data in the data memory.
- **SFR14** - This SFR is unused for special functions and can be considered as a 9th general purpose register.
- **SFR15** - This SFR is used to store immediate values in the general purpose registers i.e. if the program wants to store a 12-bit value such as "BAD" in hexadecimal or "1011101011101" in binary to a register. There is a special instruction in the SCC's instruction set that places a 12-bit value into SFR15, from here that value can be placed or added to other general purpose registers.

SFR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
15	U (Unused)					DPTR(11:0)												
14	U																	
13	U					maxSP(11:0)												
12	inPort(7:0) [Rd only, register on clk] fL										outPort(7:0)							
11	multiplicand (MD)																	
10	multiplier (MR)																	
9	TMRH_LDVAL																	
8	TMRL_LDVAL																	
7	GPIn(15:0) [Read only, register on clk] fL																	
6	GPOut(15:0)																	
5	U								LED(7)	LED(6)	LED(5)	LED(4)	LED(3)	LED(2)	LED(1)	LED(0)		
4	SEG7(15:12)					SEG7(11:8)					SEG7(7:4)				SEG7(3:0)			
3	DP(3:0)					SP(11:0)												
2	TMRH																	
1	TMRL																	
0							rotThruCca rry	mult Done	carry Flag	down TMR	Auto Reload TMR	En TMR	TMR IntEn	ISR1 IntEn	ISR0 IntEn	Global IntEn		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

Figure 2.10: SFR map of the SCC

SCC IDE

The SCC also features a viciLab project known as the SCC IDE. This IDE provides users with an IDE for programming applications in SCC assembly. This IDE features syntax checking, compilation and a run function which uploads the compiled SCC assembly code to the SCC's instruction memory.

Figure 2.11 presents the SCC IDE's programmer's model of the SCC. This view of the SCC presents the programmer with all of the necessary information

to develop applications. The largest section of the view presents the current contents of instruction memory. In this example there is a simple program which writes "FF7E" in hexadecimal or "111111101111110" in binary to the seven segments display, which will in turn display "FF7E" in the view and turns on all of the LEDs next to the seven segment displays. The instruction memory window also allows programmers to use break-pointing for debugging their applications. Break-pointing is the act of stopping executing at a certain point i.e. the break point. The current values of all register can be seen to the left of the instruction memory, beneath this a window into the SCC's stack can be seen, this window will shift up and down the stack to always display the top of the stack. Along the bottom of the programmer's model the flags and enable bits contained in SFR0 can be seen.

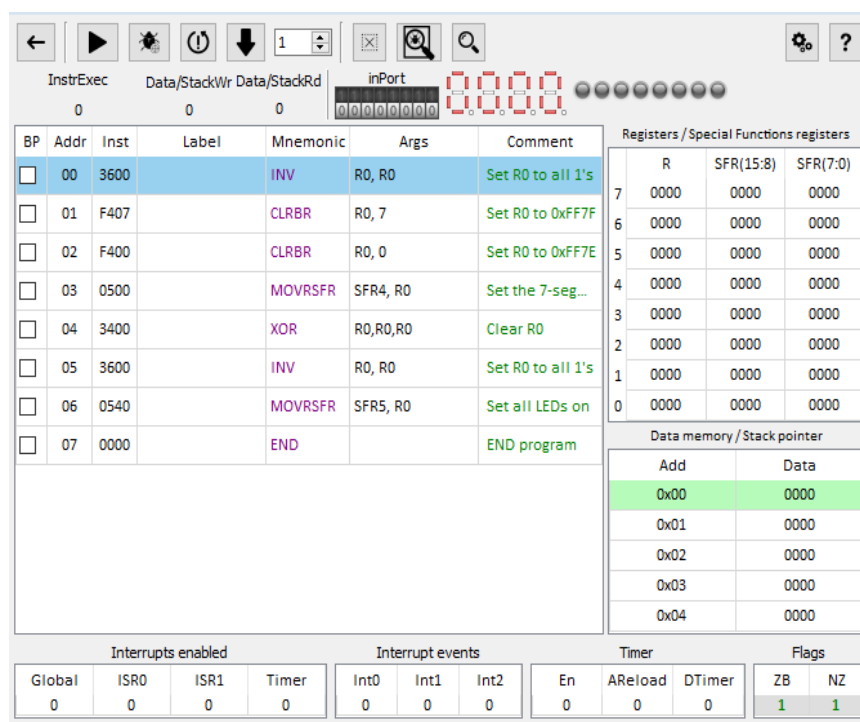


Figure 2.11: Programmer's model of the SCC

2.4.2 SCC RISC-V

There also exists in the university an SCC inspired RISC-V processor core that implements almost all of the RISC-V base integer instruction set instructions. The instructions not implemented by this processor are the load/store instructions that load 8-bit and 16-bit values to and from data memory rather

than 32-bit values. This SCC based RISC-V core almost follows the same architecture and naming conventions as the SCC bar the fact that SCC does not have a central controller module and the instructions executed on this processor are RISC-V ones. The RISC-V aspects of the VHDL code of this SCC-based processor were used as a basis for the RISC-V processor in this project.

Where the architecture of this SCC-RISC-V processor differs from the SCC architecture is in the controller module implemented in this design. This controller module replaces most of the functionality in the processor that is concerned with decoding and interpreting instructions. It behaves like the conductor in an orchestra, directing the other modules in the processor on how and when to behave. Architectures that utilise a main controller module tend to be easier for the processor designer to implement but this comes at the price of extensibility and performance. This controller module architecture does not allow for easy modification as the controller may need to be re-designed if certain RISC-V instruction sets such as multiplication or floating-point operations are to be implemented, this architecture also raises the difficulty of increasing the processors performance via pipe-lining, as the controller module's function requires keeping the context of the current instruction. In a pipe-lined processor it would not be efficient to use one controller module to keep the context of many instructions at once.

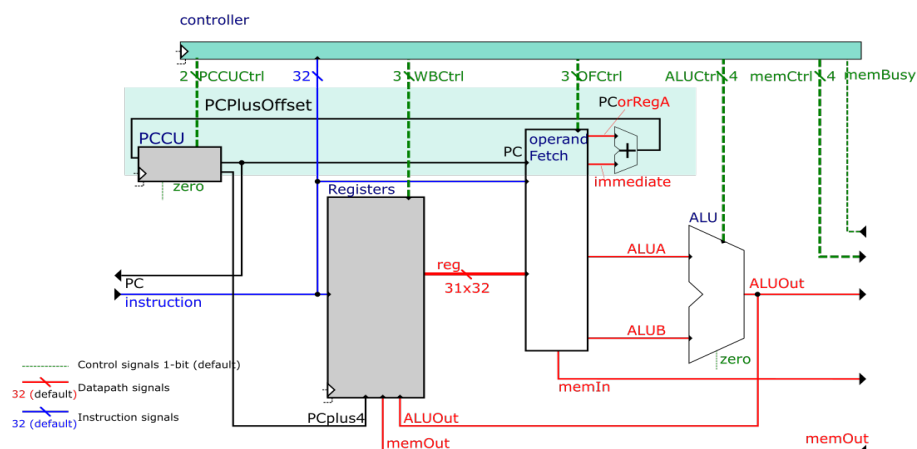


Figure 2.12: SCC inspired RISC-V processor architecture diagram

Chapter 3

RISC-V Processor Core

This chapter details the development of the RISC-V processor. The design section details how the processor was designed and how was it was researched i.e., topics like computer architectures, processor design, RISC-V and alternate design considerations. The implementation section details the process of creating the RISC-V processor in VHDL using the Vivado Design Suite. Finally, the testing section details how the RISC-V processor was evaluated and tested using VHDL testbenches and other verification techniques.

3.1 Design

3.1.1 Purpose/ Design Specification

The first task in designing a processor is to first understand what a processor is and what it is required to do i.e., it's purpose and use cases. Based on these requirements a design specification can be generated. This specification details what instructions or operations this processor can execute to meet these use cases and how the hardware of the processor implements these instructions or operations.

What is a processor?

As a concept, a processor is a Turing Machine [30]. A Turing machine is a mathematical model of computation that describes a hypothetical machine that can calculate anything calculable and compute anything computable, given sufficiently large memory and a sufficiently large amount of time. The Turing Machine was invented by Alan Turing [17] in 1936. Turing Machines are

automatic machines, meaning that Turing Machines will operate automatically when they are provided with a series of instructions to execute. Processors are Turing Machines constructed from digital circuitry. To implement a Turing machine in this fashion, generally three main pieces of circuitry are required (see figure 3.1):

- **Memory circuitry** - To store the information that the machine will operate on
- **Calculation circuitry** - To perform calculations on the information stored in the memory circuitry
- **Control circuitry** - To interpret the instructions passed to the machine and orchestrate the behaviour of the Memory and Calculation circuitry

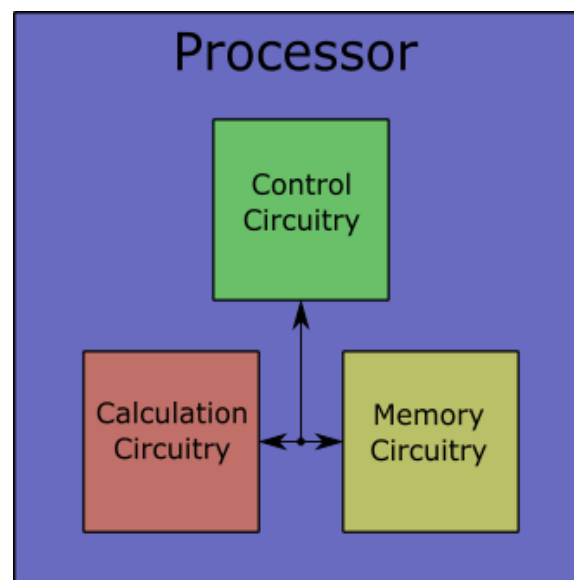


Figure 3.1: 3-piece model of a processor

The major pieces of circuitry in a processor in reality are slightly more complex than this 3-piece model due to the practicalities of implementing this machine on physical circuits. The largest limitation on modern processors that renders this 3-piece model inaccurate is memory. In practice two major pieces of Memory circuitry are required, one to store information, like the one mentioned above, often referred to as data memory. But another to store the instructions that are to be executed on the processor, often referred to as instruction memory (see figure 3.2). In addition to this, practical Memory circuitry tends to be quite

large and therefore much slower than the Calculation and Control circuitry. This results in the Calculation and Control circuitry waiting large amounts of time for the Memory circuitry to finish its work. Having the Calculation and Control circuitry "idle" for large periods of time is not ideal, this reduces the efficiency of the processor drastically. The solution to this is to provide the Calculation and Control circuitry with a smaller amount of Memory circuitry that operates as fast as the Calculation and Control circuitry. This smaller Memory circuitry is known as "Registers" [26], [45]. Where one "Register" can contain one piece of information. Typically, a processor will contain anywhere from 8-32 registers. Registers are viewed as the bridge between the Calculation and Control circuitry, and the larger Memory circuitry or data memory. Processors will include special instructions that instruct the two types of memory, data memory and registers, to exchange data between them. Typically, the information the processor immediately needs to operate on is transferred from the data memory to the registers. From here the Calculation and Control circuitry operate as fast as they can on the data. The data is then transferred from the registers back to the data memory for more long term storage. This system of memory "hierarchy" can be equated to an artist at their desk. The artist only places the materials and tools they need for the particular piece of art they are working with on the desk. They do not store all of their art supplies on the desk as they work. The rest of their materials and tools are stored in their cupboards and shelves around but not on the desk.

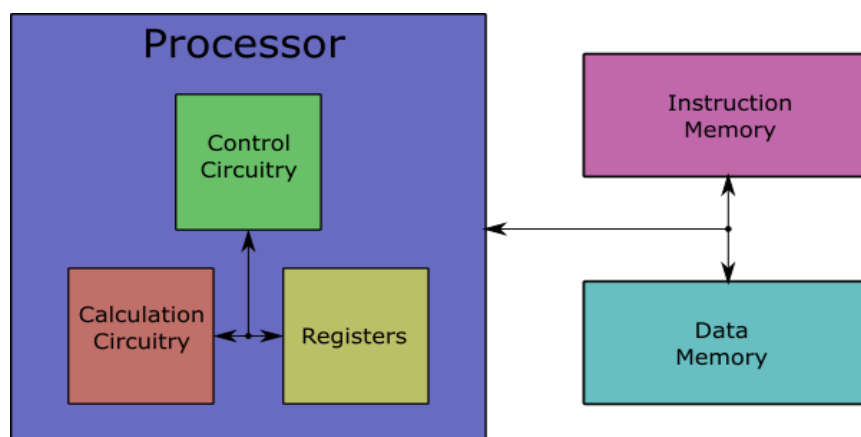


Figure 3.2: Practical model of a processor

What will this processor be used for?

The purpose of the processor created during this project is to be a learning tool. The processor is not required to be high performance or to be low power, it does not have to meet any of the constraints that a modern processor typically has to conform to. This processor will be presented to students as a teaching tool to aid them in learning about computer architecture, processor design and developing embedded applications. To effectively utilise this processor, students should be able to:

- Write their own applications and execute them on the processor
- Explore the hardware implementation of the processor as it executes instructions
- Create new or modify existing aspects of the processor

These use cases highlight the freedoms and constraints placed on this processor. To allow students to write their own applications for the processor, a suitable ISA [8], [43] must be chosen that has: accessible compilation tools for popular high level programming languages such as C or C++, and instructions that are useful in real world contexts i.e. to educate students on how to develop embedded applications for real world problems in both high level programming languages like C or C++ but also in the assembly language (the instructions) of the processor.

To allow students to explore the hardware implementation or architecture [22], [54] of the processor i.e. how the hardware components are organised in the processor, must be simple and intuitive to understand. The processor's architecture should implicitly explain how an instruction is executed within the machine. If this is achieved, students will then be able to execute their own applications on the processor and view the behaviour of the processor at any stage of its execution and at any level of detail required. To truly gain an appreciation for the machine, students should have the ability to create new or modify existing aspects of the processor. With this resource students can investigate how different methods of implementing the same functionality can

affect the behaviour and performance of the processor. Students may also be able to add to or modify the processor's behaviour and or performance in ways that were not previously known before. The architecture and implementation of the processor should be modular in nature to achieve this. This way students can alter the processor as easily as possible i.e. if component "A" of the processor performs function "X", the other components in the processor do not care about how "A" performs "X" but only that it does so.

3.1.2 Instruction Set Architecture

The instruction set architecture [8], [43] of a processor is the bridge between the software and the hardware. The ISA of a processor defines all of the instructions a processor must be able to execute. This in turns defines what software compilation tools must target i.e. translating from high level programming languages like C to the instruction set of the processor. The ISA also defines how many registers the processor must have and the conventions programmers and software must follow such as; how to use these registers and how to partition the data memory i.e. a memory map containing elements such as stacks, user data etc.

RISC vs. CISC

Typically there are two types of ISAs. These are known as RISC (Reduced Instruction Set Computer) [46] and CISC (Complex Instruction Set Computer) [42]. A RISC ISA is an ISA that defines a small set of general use instructions that perform simple tasks. RISC ISAs tend to define a larger amount of registers, typically 16-32. Due to the large amount of registers, RISC ISAs tend to follow a load/store architecture. Load/store architectures dictate that data memory can only be accessed via special instructions i.e. load from memory, store to memory. These special instructions transfer data between the registers and data memory. Conversely, a CISC ISA is an ISA that defines a large set of specific use instructions that perform complex tasks i.e. transfer data from memory then perform calculations on that data and transfer it back to memory. CISC ISAs tend to define a smaller amount of registers, typically 8, as CISC architectures allow many instructions to access data memory, not just special

instructions.

Retrieving two numbers from memory, adding them and storing the result back into memory would require 4 RISC instructions i.e., load the first number from memory, load the second number from memory, add both numbers together, store the result in memory. This exact same task would only require 1 CISC instruction i.e., add these two numbers stored in memory, place the result back in memory. The main trade-offs associated with these ISA types are: who's job needs to be more difficult? The hardware designer or the programmer? RISC ISAs encourage simpler hardware implementation by offloading the complexity to the compilation tools and the programmers. Whereas, CISC ISAs try to make the compilation tool's and programmer's lives easier by offloading the complexity to the hardware implementation of the processor.

For the requirements that are demanded of the processor designed during this project, a RISC ISA was chosen. A simpler hardware implementation is more likely to be beneficial to students than more convenient application development. The tedium present in developing applications that are inevitably longer in a RISC instruction set, rather than a CISC instruction set, can be avoided by utilising compilation tools for higher level programming languages like C.

Chosen ISA - RISC-V

There are two options when choosing an ISA, either a custom ISA can be designed or an existing ISA can be used. Given that this processor must give students practical skills for real world contexts, designing a custom ISA was deemed needless. It would be more beneficial for students to have skills and experience associated with products currently being used in industry today. This argument against a custom ISA is coupled with the fact that existing ISAs have been created by professional designers with many years experience. Therefore the decision was made to search for a suitable existing ISA. Many RISC ISAs were considered as potential candidates for the ISA of the processor designed during this project. The most attractive options are ISAs that are open-source. Gaining access to and using these ISAs requires no permissions to be granted

or royalties to be paid.

The ISAs that fit these requirements are: Mico32 [28], MIPS [25], OpenRISC [7], RISC-V [14] and SPARC [21]. Mico32 was ruled out because while it is currently being used, it is only used by Lattice Semiconductors in their FPGA devices. SPARC used to be a very popular ISA in older Sun Microsystems machines, however this ISA fallen out of popularity as the industry progressed. OpenRISC is a more modern RISC ISA, but however there are no companies producing OpenRISC processors outside of implementing them on FPGAs. These factors of popularity reduced the options down to MIPS and RISC-V. MIPS is a veteran ISA which has been used for many years but unfortunately its design does not mesh well with modern processor requirements. It's contender, RISC-V, has been designed fresh with more modern design considerations, learning from the mistakes of older ISAs that MIPS still suffers from. This is also paired with fact that RISC-V is more versatile and expandable with it's modular structure (see section 2.1 in chapter 2). As of January 2019, RISC-V has received massive industry adoption, most notably, Western Digital has created their own open-source RISC-V processor core, named "SweRV" [9]. This RISC-V processor is aimed to be used in their products in the years to come. For these reasons, the RISC-V 32-bit ISA was chosen as the ISA for the processor designed during this project.

3.1.3 Processor Architecture

The architecture of a processor details how the hardware components of processor are organised. A processor can have one ISA i.e. RISC-V, but can have many different architectures or ways of implementing that ISA. Each of these will have varying levels of size, power and performance. There are two main architectural decisions that must first be made when designing a CPU. These are: what memory architecture will be used? Will this be a single cycle or multi-cycle processor? The decisions that were made to answer these questions will be discussed in this section.

Architectural Decisions

There are two types of memory architecture that can be used, these are known as; Von-Neumann and Harvard. A Von-Neumann architecture places instructions and data into the same memory circuitry (see the left side of 3.3). While a Harvard architecture places instructions and data into two separate pieces of memory circuitry (see figure right side of figure 3.3). The trade-offs present with these two architectures are complexity and performance. A Von-Neumann architecture is simpler than the Harvard architecture as the processor only has to communicate with one piece of memory circuitry. However, this means that the processor cannot access new instructions and transfer data at the same time, this introduces a performance bottleneck. The Harvard architecture separates the two types of memory, forcing the processor to become more complicated. However, having the ability to access new instructions and transfer data at the same time significantly improves performance.

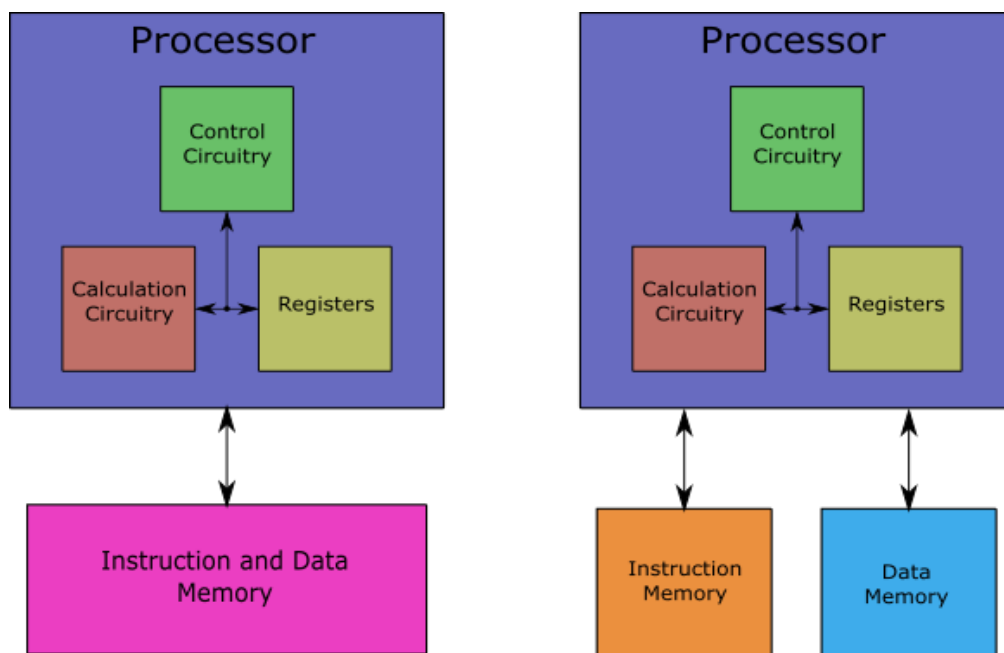


Figure 3.3: Von-Neumann vs. Harvard Architectures

A RISC-V processor can be designed in either way, however, given that this processor is meant to be future proofed, it was deemed better to utilise a Harvard architecture as it gives the processor more flexibility in terms of modification and extensibility i.e. for pipe-lining [23], [44]. It could also be argued

that a Harvard architecture is also conceptually and practically easier to understand as the processor designer can separate out the two different forms of information the processor must handle i.e. instructions and data.

The next architectural decision to be made is to design the processor such that it executes instructions in either one cycle or in multiple cycles. A cycle is one oscillation of the processors clock. The processor's clock is a signal that turns on and off at regular intervals. It can be equated to the heartbeat of the processor, coordinating all of the actions of the components within processor in a predictable manner. A processor that executes instructions in a single cycle tends to be simpler to implement but tends to be inefficient. This is because the length of time that constitutes one cycle is dependant on the slowest instruction i.e. a chain is only as strong as it's weakest link. A processor that executes instructions in multiple cycles tends to be more complicated to implement as it has to split up it's constituent components. But this results in better performance as the processor finishes executing an instruction regardless of how long the slowest instruction takes. As with the Von-Neumann and Harvard architectures, this another trade-off which balances simplicity and performance. The processor designed during this project must be simple so that it will be more intuitive to understand. Therefore a single cycle implementation was chosen. The ability to extend the processor i.e. increase it's performance, can be implemented by splitting up the processors functions within this single cycle. This will allow the processor to be pipe-lined in the future.

Architecture Design

With the main architectural decisions for the processor made. The next step is to design the Control and Calculation circuitry described in section 3.1.1 to implement the RISC-V ISA. The Control and Calculation circuitry together is known as the "Control and Data Path" as it is the "path" which the data takes through the processor. The Control circuitry then orchestrates how the data flows through this path.

The architecture of this processor must be simple so that it is easy to understand and as previously mentioned, it must also be easily extended to improve the performance of the processor. To meet these requirements it was decided upon to design an architecture that splits up the functionality of the processor in such a way that it's still simple and intuitive to understand, but that it also lends itself to pipe-lining in the future. The classic RISC pipeline [22], [41] was chosen as the architecture for the Control and Data Path (see figure 3.4). It was chosen because of how intuitively it describes how a processor executes an instruction, also known as the instruction life cycle. This architecture also leads well into a 5-stage pipeline. Each of the modules will be introduced briefly and then their designs will be described in greater detail. Note: the Control circuitry described in section 3.1.1 is distributed throughout to the modules. In figure 3.4 the modules are as follows:

- **IF** - Instruction Fetch module, the module that interfaces with instruction memory and keeps track of what instruction is currently being executed
- **ID** - Instruction Decode module, the module that receives the current instruction, decodes the instruction and passes extracted information to the remaining modules i.e., instruction type, function codes, register addresses etc.
- **EX** - Execution module, the module that contains the Calculation circuitry described in section 3.1.1. It executes all of the calculations and comparisons the processor can perform
- **MEM** - Memory Management module, the module that interfaces with data memory during load and store instructions
- **WB** - Write Back module, the module that arbitrates which piece of information i.e. results from calculations or data from memory, is stored in the registers.
- **Registers** - Register Bank module, the module that contains the 32 registers defines by the RISC-V ISA (see section 2.1 of chapter 2).

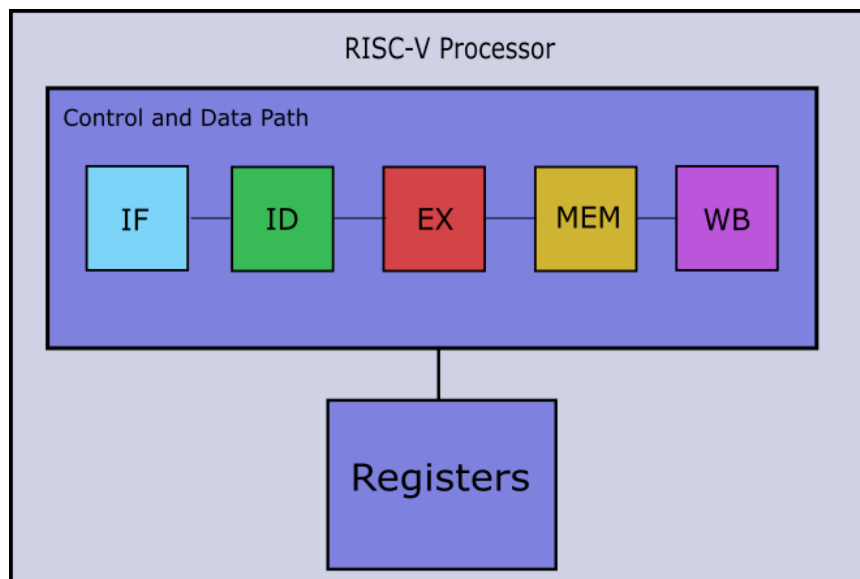


Figure 3.4: Classic RISC 5-stage pipeline

Figure 3.5 presents the 5-stage architecture in the context of the entire system. How the processor interfaces with the instruction and data memories can be seen in the figure. The processor is intended to be as simple as possible, so therefore no high-security handshake based interfaces are used. There are three other connections to the processor in the figure:

- **ce** - Chip enable, the system will not do anything unless this signal is asserted or "1"
- **rst_b** - Reset, when this signal is de-asserted or "0", the entire system will reset. This entails setting all memory circuits inside the processor back to their default state, typically "0". the "_b" notation is used to describe a signal that performs it's action when it is de-asserted or "0".
- **clk** - Clock, the clock signal of the processor, as described earlier.

The interface with instruction memory can be seen on left side of the figure. There are only a few signals needed to transfer the required information between the processor and instruction memory, these signals are:

- **instr_addr** - The address of the instruction i.e. where in the instruction memory is the instruction stored
- **instr_in** - The instruction itself contained in memory

The interface with data memory can be seen on the right side of the figure. It has considerably more signals than the instruction memory interface, but all these signals are required none the less to correctly communicate with the data memory. These signals are:

- **data_in** - The signal that carries data from memory to the processor
- **data_out** - The signal that carries data from the processor to memory
- **data_addr** - The signal that carries the address within memory that data is stored to or retrieved from
- **mem_rd_wr** - The signal that communicates to data memory to store the information we provide it into the address we provide i.e. write, or to retrieve the data at the address we provide i.e. read.
- **mem_valid** - The signal that communicates to data memory that the information on the other signals is valid
- **mem_size** - The signal that communicates to the memory what size of data word we wish to receive i.e. a byte (8-bits), a half-word (16-bits) or a word (32-bits). This signal is required by the RISC-V ISA as it defines load and store instructions of varying sizes.

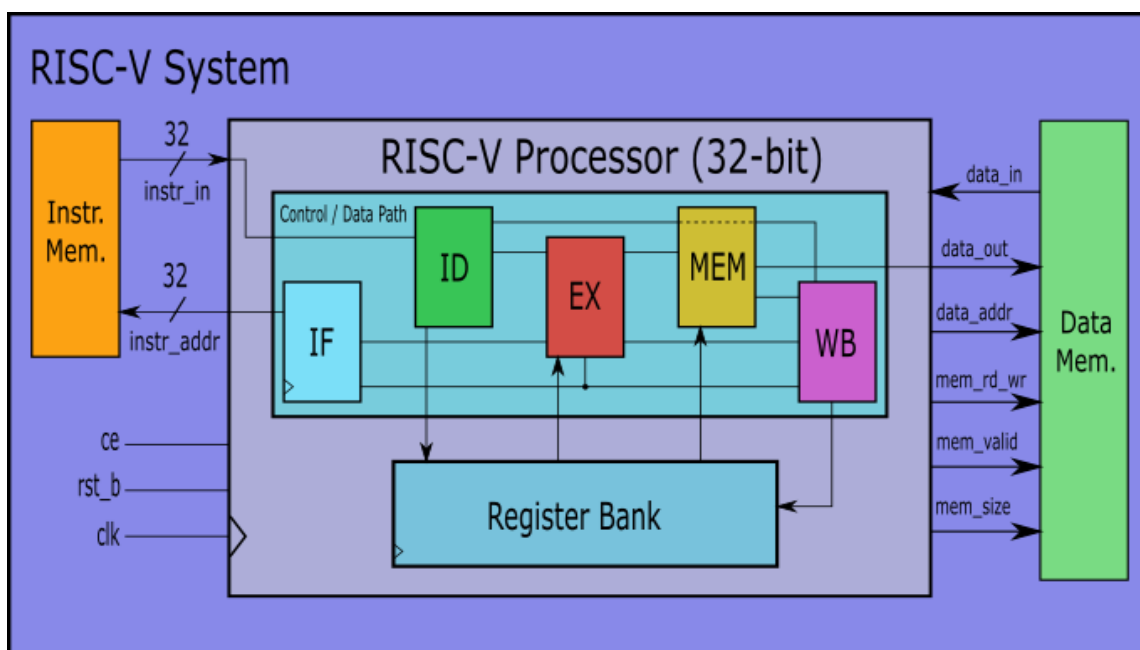


Figure 3.5: Architecture used for the processor design during this project

It can be seen in the figure that the instruction address signal originates from the Instruction Fetch (IF) module, the Instruction Fetch module is the beginning of the life cycle of an instruction. When an instruction is transferred from the instruction memory to the processor, it is received by the Instruction Decoder (ID) module. The Instruction Decoder extracts the information encoded in the instruction and passes it to the remaining modules. The Register Bank (RB) will return the data stored in any registers that are required for the instruction. The Execution (EX) module performs any calculations and comparisons required on the data in the registers based on the information given by the Instruction Decoder. The Execution module will output the address of the next instruction the processor needs to execute and transfer this to the Instruction Fetch module. If the instruction is a load or store instruction, the result of the calculations performed will be used as the data address and the Memory Management (MEM) module will communicate with the data memory by asserting the correct information on the data memory interface signals. In the case of a store, Memory Management will take data stored in the registers and direct data memory to store this data i.e. "mem_rd_wr" = "1". In the case of a load, Memory Management will direct the data memory to return the data stored at an address i.e. "mem_rd_wr" = "0", Memory Management will transfer the data received from data memory to the Write Back (WB) module. Write Back will arbitrate between the different data values generated during all this activity i.e., result of calculations, data from memory, and will select which data value will be written back to the registers. In the case of an arithmetic instruction, this will be the result of a calculation. In the case of a load instruction, this will be the data returned from memory. Once the register has received its new data, this is the end of the instruction life cycle.

Instruction Fetch Module (IF)

The Instruction Fetch module is the module that interfaces with the instruction memory. Figure 3.6 presents the design of the Instruction Fetch module. The Instruction Fetch module consists of a memory circuit known as a "program counter", this memory circuit keeps track of the address of the current instruction being executed by the processor. This module's inputs are on the

left and it's outputs are on the right, these are:

- **npc** - Next Program Counter, on the next clock cycle, this is the value the program counter will take i.e. the next instruction address.
- **ce** - Chip Enable, the Instruction Fetch will not accept a new instruction address if the processor is not enabled.
- **mem_busy** - Memory Busy, while the memories are working, this signal is asserted. The processor will not progress it's execution until both instruction and data memory are ready.
- **pc** - Program Counter, the current value of the program counter i.e. the current instruction address.
- **pc_plus_4** - Program Counter + 4, this is the next sequential instruction address. "4" is added to the program counter because RISC-V defines that memory is byte (8-bits) addressable. Meaning every location stores a byte (8-bits) rather than the width of the processor (32-bit).

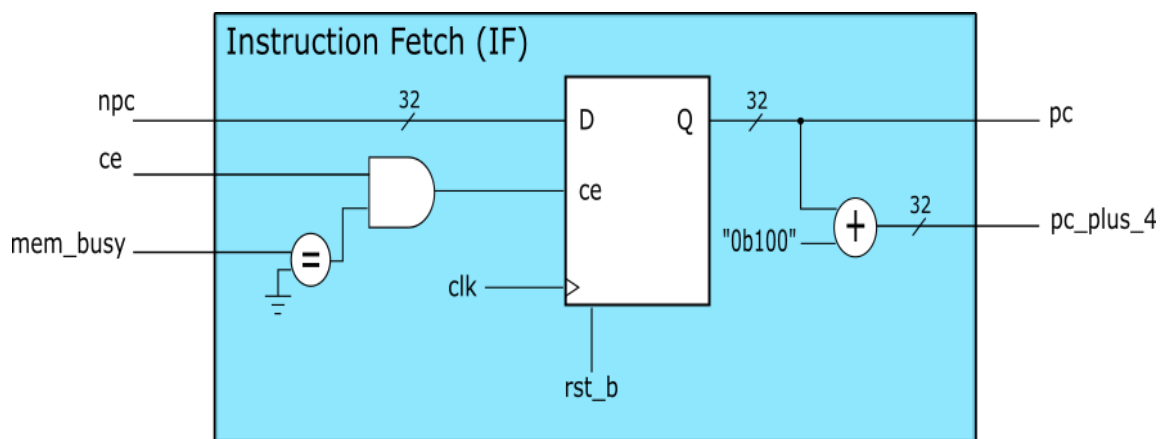


Figure 3.6: Instruction Fetch Module Design

The Instruction Fetch module will only allow the program counter to update it's value if "ce" is asserted and if "mem_busy" is de-asserted. This is denoted by the comparison with "0" for "mem_busy" and the AND gate connecting "ce" and "mem_busy" which in turn connects to the "ce" or chip enable input of the program counter itself.

Instruction Decode Module (ID)

The Instruction Decode module is the module that receives the current instruction from instruction memory and decodes the information encoded in the instruction required to execute it. Figure 3.7 presents the design of the Instruction Decode module. The information decoded by this module is all defined by the RISC-V ISA. This module's inputs are on the left and its outputs are on the right, these are:

- **instruction** - The 32-bit instruction coming from instruction memory.
- **opcode** - This signal is the unique identifier for the type of instruction being executed i.e. load, store, register-register, register-immediate, jump
- **f3** - This signal is a function code identifying which exact function to execute within an instruction type i.e. addition, subtraction, AND, OR etc.
- **f7** - This signal is an auxiliary function code used during addition/subtraction instructions to identify whether the instruction is addition or subtraction, as both share the same "f3" code.
- **rs1** - This signal is the address of the first source register, contained in the Register Bank, that will be used during calculations
- **rs2** - This signal is the address of the second source register, contained in the Register Bank, that will be used during calculations and store instructions
- **rd** - This signal is the address of the destination register, which will store the result of a calculation or data retrieved from memory.
- **immediate** - This signal is the immediate value encoded in some of the RISC-V instructions sign-extended to 32-bits.

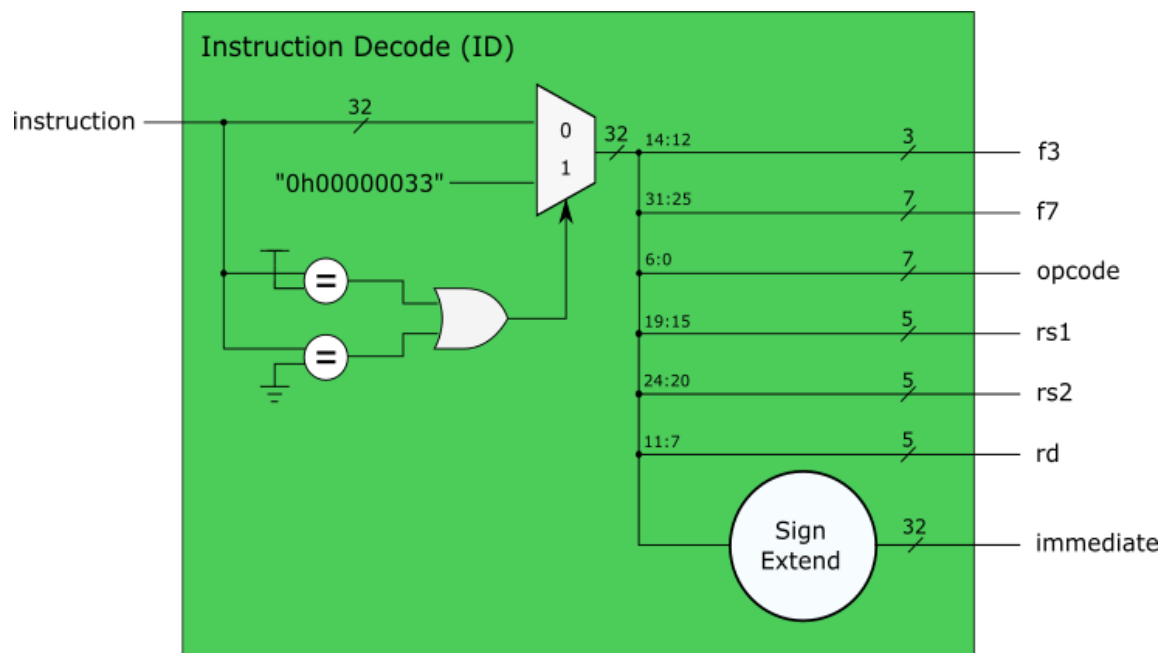


Figure 3.7: Instruction Decode Module Design

Before the Instruction Decode module actually begins to decode the instruction, the module first detects whether or not this instruction came from an uninitialised memory circuit. Typically, memory circuits that have not been programmed will contain either all "1"s or all "0"s. The Instruction Decoder detects this by comparing the incoming instruction to both "FFFFFFFF" and "00000000" in hexadecimal or all "1"s and all "0"s in binary. If either of these comparisons return as true, a NOP (No-Operation) instruction will be injected into the processor instead. This can be seen via the multiplexer present in figure 3.7. A NOP instruction in RISC-V is an addition instruction that adds the contents of register "x0" to itself and then stores the result into "x0". However, because register "x0" is hardwired to "0", this results in: $0 + 0 = 0$, which is then stored in register "x0", so nothing happens. NOP instructions are often used when a programmer wants the processor to literally do nothing.

Assuming that the instruction received from instruction memory is valid, the Instruction Decoder will then decode the instruction. Due to how RISC-V has been designed, most of the decoding is simply just tapping off certain sections of the 32-bit instruction i.e. "opcode" is just bits 6 down to 0 of the instruction. The only major piece of decoding the Instruction Decoder must do is generating the immediate values encoded in each instruction format. Figure

3.8 presents the 32-bit instruction formats. These are used to encode different types of instructions. Note how RISC-V has been designed so that the fields in these formats remain in the same position, this fact is what allows the Instruction Decoder to be so simple. The only deviation from this is the immediate value encoded in all the instructions formats bar "R". The purpose of each instruction format is as follows:

- **R-type** - This format is used for instructions that use two registers as the operands or sources of the data for calculations.
- **I-type** - This format is used for instructions that use one register and an immediate value as the operands or sources of data for calculations.
- **S-type** - This format is used for store instructions.
- **B-type** - This format is used for jump or branch instructions.
- **U-type** - This format is used for instructions that require a large immediate value
- **J-type** - This format is used for the JAL (Jump And Link) instruction. Which adds the large immediate value to the program counter to create the next instruction address and then stores the next sequential instruction address (Program Counter + 4) to the destination register.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]				rs2		rs1		funct3		imm[4:1:11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20:10:1:11:19:12]										rd		opcode		J-type

Figure 3.8: RISC-V 32-bit instruction formats

The sign-extend function of the Instruction Decoder extracts the immediate value encoded in the instruction and sign-extends it to 32-bits. For example, for the "I-type" format, the immediate value encoded is 12-bits i.e. imm[11:0]. This value is tapped from the instruction and placed into a 32-bit word. However the most significant bit i.e. bit 11, is copied into bits 31 down to 12 of this

new 32-bit word, thus sign-extending the immediate value.

Execution Module (EX)

The Execution module is the module that performs all the calculations and comparisons the processor can execute. Figure 3.9 presents the design of the Execution module. The Execution module contains the Calculation circuitry described in section 3.1.1. This module's inputs are on the left and its outputs are on the right, these are:

- **f3** - This signal is the function code generated by the Instruction Decode section
- **f7** - Auxiliary code generated by the Instruction Decode section
- **opcode** - This signal is the instruction type code generated by the Instruction Decode section
- **pc_plus_4** - This signal is the next instruction in memory i.e. current instruction address stored in the program counter + 4
- **rs2_addr** - This signal is used during shift instructions to determine how much to shift data by.
- **pc** - The current instruction address stored in the program counter
- **rs1_data** - This signal is the data contained in the first source register selected by the current instruction
- **immediate** - This signal is the immediate value generated by the Instruction Decode module
- **rs2_data** - This signal is the data contained in the second source register selected by the current instruction
- **branch_out** - This signal is the next instruction address. On the next clock cycle the program counter will take on this value.
- **alu_out** - This signal is the result of calculations performed by the Execution module

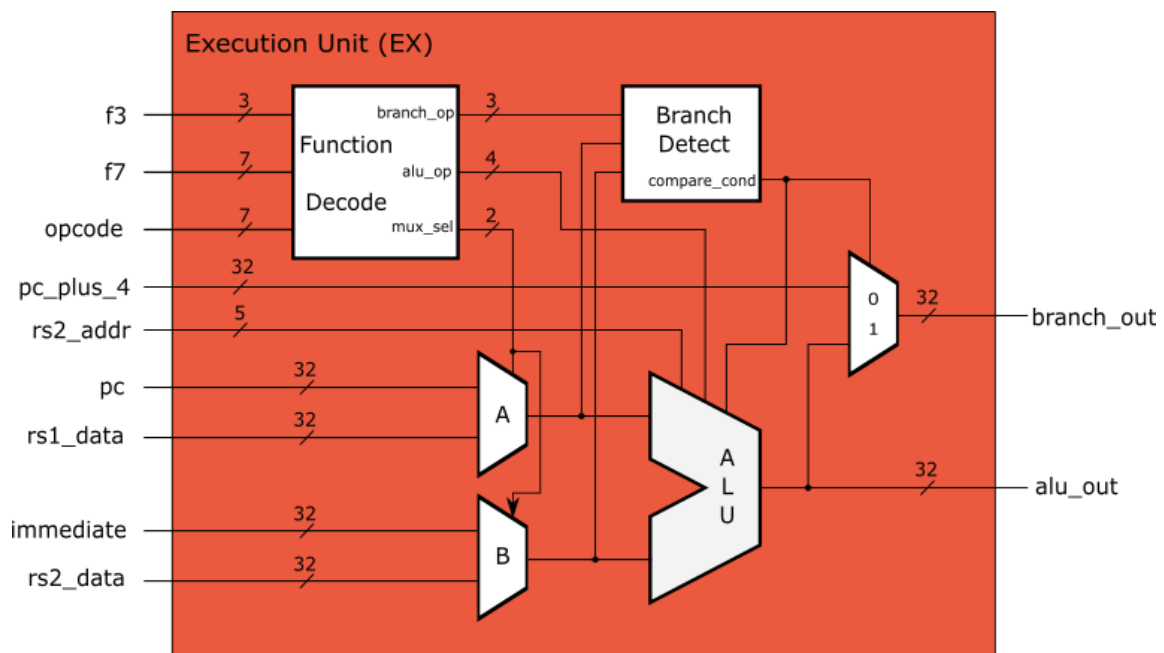


Figure 3.9: Execution Module Design

The Execution module was designed to take care of two major pieces of functionality that the processor requires. These are calculations and control flow adjustment. Control flow adjustment is the phrased used to describe when the processor makes decisions on which instruction to execute next based on some comparison result. This fact generates the requirement for two pieces of circuitry that need to be designed; an ALU (Arithmetic/Logic Unit) which performs calculations and a Branch Detection Unit which performs comparisons and therefore decisions based on these comparisons.

For the ALU and Branch Detection Units to know what calculations and comparisons to perform, the Function Decode module was designed to interpret the opcode, f3 and f7 codes generated by the Instruction Decoder. These 3 signals identify which type of instruction (opcode) and which specific function within that instruction type (f3 and f7) is being executed. The Function Decode module generates 3 signals based on the values of the function codes. These are:

- **branch_op** - This signal directs the Branch Detection Unit to either perform comparisons to determine what decision should be made, to change the flow of the program or to not do anything.

- **alu_op** - This signal directs the ALU to perform certain calculations i.e. addition, subtraction, AND etc.
- **mux_sel** - This signal directs the input multiplexers of the Execution module to select the appropriate data for the ALU to perform calculations on.

The Branch Detection Unit receives the "branch_op", "A" and "B" signals. Depending on the value of "branch_op", the Branch Detection Unit will perform different types of comparisons on "A" and "B", if the result of these comparisons is true, the Branch Detection Unit will output the signal "compare_cond". This signal is used to select what will be the next instruction address i.e. which instruction to execute next. There are two possible instruction addresses to use; the next sequential address i.e. "pc_plus_4", or the result of the ALU. The ALU is used by the processor during decision making or branch instruction to generate the next instruction address. The "branch_op" has 8 possible values, the Branch Detection Unit therefore has 8 possible functions:

- **000** - If "A" and "B" are equal, select the branch address calculated by the ALU and not the next instruction address i.e. "pc_plus_4"
- **001** - If "A" and "B" are not equal, select the branch address calculated by the ALU and not the next instruction address
- **010** - If "A" is less than "B" (signed comparison), select the branch address calculated by the ALU and not the next instruction address
- **011** - If "A" is greater than or equal to "B" (signed comparison), select the branch address calculated by the ALU and not the next instruction address
- **100** - If "A" is less than "B" (unsigned comparison), select the branch address calculated by the ALU and not the next instruction address
- **101** - If "A" is greater than or equal to "B" (unsigned comparison), select the branch address calculated by the ALU and not the next instruction address
- **110** - Don't perform any comparisons but select the branch address calculated by the ALU and not the next instruction address

- **111** - Don't perform any comparisons and select the next instruction address, not the branch address calculated by the ALU

The ALU receives the "alu_op", "compare_cond", "rs2_data", "A" and "B" signals. Depending on the value of "alu_op" the ALU will perform different calculations on "A" and "B". The ALU can perform 12 calculations, these are:

- **0000** - Add "A" and "B" together
- **0001** - Subtract "B" from "A"
- **0010** - AND "A" and "B" together
- **0011** - OR "A" and "B" together
- **0100** - XOR "A" and "B" together
- **0101** - Shift "A" left logically by the amount in signal "rs2_addr". This is because in RISC-V, "rs2_addr" is used to shift by an immediate value and not a value stored in a register
- **0110** - Shift "A" right logically by the amount in signal "rs2_addr"
- **0111** - Shift "A" right arithmetic by the amount in signal "rs2_addr"
- **1000** - Shift "A" left logically by the amount in "B"
- **1001** - Shift "A" right logically by the amount in "B"
- **1010** - Shift "A" right arithmetically by the amount in "B"
- **1011** - If "compare_cond" is asserted because a comparison resulted true, the ALU will output "1". If "compare_cond" is de-asserted, the ALU will output "0". This functionality is required for the "Set less than" instructions described in section 2.1.2 of chapter 2.

Memory Management Module (MEM)

The Memory Management module is the module that interfaces with data memory during load and store instructions. Figure 3.10 presents the design of the Memory Management module. To retain simplicity in processor to memory

interfaces, it is assumed that the processor will transfer data values i.e. the 8-bit, 16-bit and 32-bit data values, on a 32-bit bus. This module's inputs are on the left and its outputs are on the right, these are:

- **mem_data_in** - This signal contains the data coming from the data memory to the processor
- **f3** - This signal is the function code generated by the Instruction Decode section
- **rs2_data** - This signal is the data contained in the second source register selected by the current instruction. This is the data that is transferred from the processor to data memory
- **opcode** - This signal is the instruction type code generated by the Instruction Decode section
- **data_val** - This signal contains the data coming from the data memory to the processor sign-extended to 32-bits
- **mem_data_out** - This signal contains the data going from the processor to the data memory sign-extended to 32-bits
- **mem_rd_wr** - This signal communicates to the data memory whether perform a read or a write i.e. "0" or a "1".
- **mem_size** - This signal communicates to the data memory the size of the data being transferred. 00 - 8-bit, 01 - 16-bit, 10 - 32-bit.
- **mem_valid** - This signal communicates to the data memory that the information on the other signals is valid and can be used

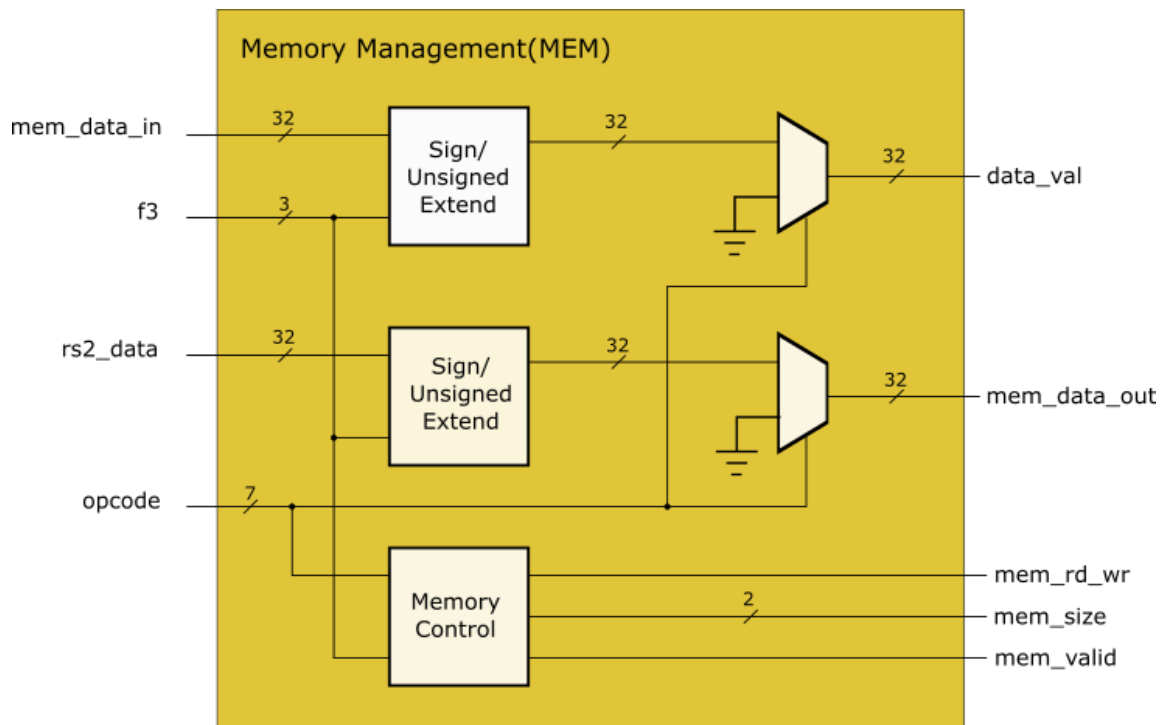


Figure 3.10: Memory Management Module Design

During load and store instructions i.e. "opcode" = load or store, the Memory Management module interprets the "f3" function code to know which type of load or store instruction is being executed. In the case of a load instruction, the Memory Management module will de-assert "mem_rd_wr" to communicate to the Data memory to return a data value at the address provided by "alu_out" signal. It will also; set "mem_size" to the value corresponding with the correct data size being transferred and assert "mem_valid" to direct the Data memory to return the data value. The Data memory will then provide the corresponding data value on "mem_data_in" and sign-extend this value to 32-bits. In the case of a store instruction, the Memory Management module will assert "mem_rd_wr" to communicate to the Data memory to store the data value provided by "mem_data_out", at the address provided by "alu_out" signal. It will also; set "mem_size" and "mem_valid" accordingly, as done in a load instruction. The Memory Management module will then take the data stored in "rs2_data", sign-extend it to 32-bits and transfer it to "mem_data_out". Note: the 2 multiplexers present in figure 3.10 are used so that unless a load or store instruction is being executed, no data will exit the Memory Management module.

Write Back Module (WB)

The Write Back module is the module that arbitrates which piece of information generates during the execution of the processor i.e. results from calculations or data from memory, is stored in the registers. Figure 3.11 presents the design of the Write Back module. This is the simplest module in the processor, however it is absolutely integral when creating a pipe-lined architecture. This module's inputs are on the left and its outputs are on the right, these are:

- **pc_plus_4** - This signal is the next instruction address i.e. current instruction address + 4, during jump instructions this is the return address and must be saved to a register
- **mem_data_val** - This signal is the data transferred from Data memory to the processor during a load instruction.
- **alu_out** - This signal is the output of the ALU in the Execution module
- **opcode** - This signal is the instruction type code generated by the Instruction Decode section
- **rd_data** - This signal is the data which will be written to the destination register at the end of this instructions execution
- **reg_wr** - This signal is used to communicate with the Register Bank as to whether or not this instruction will write to a destination register.

The Write Back module contains a small piece of control circuitry i.e. "WB Logic". This circuitry interprets the "opcode" signal to select based on the type of instruction being executed which data value will be written to the destination register. "WB Logic" will also assert the "reg_wr" signal if the destination register is being written to during this instruction.

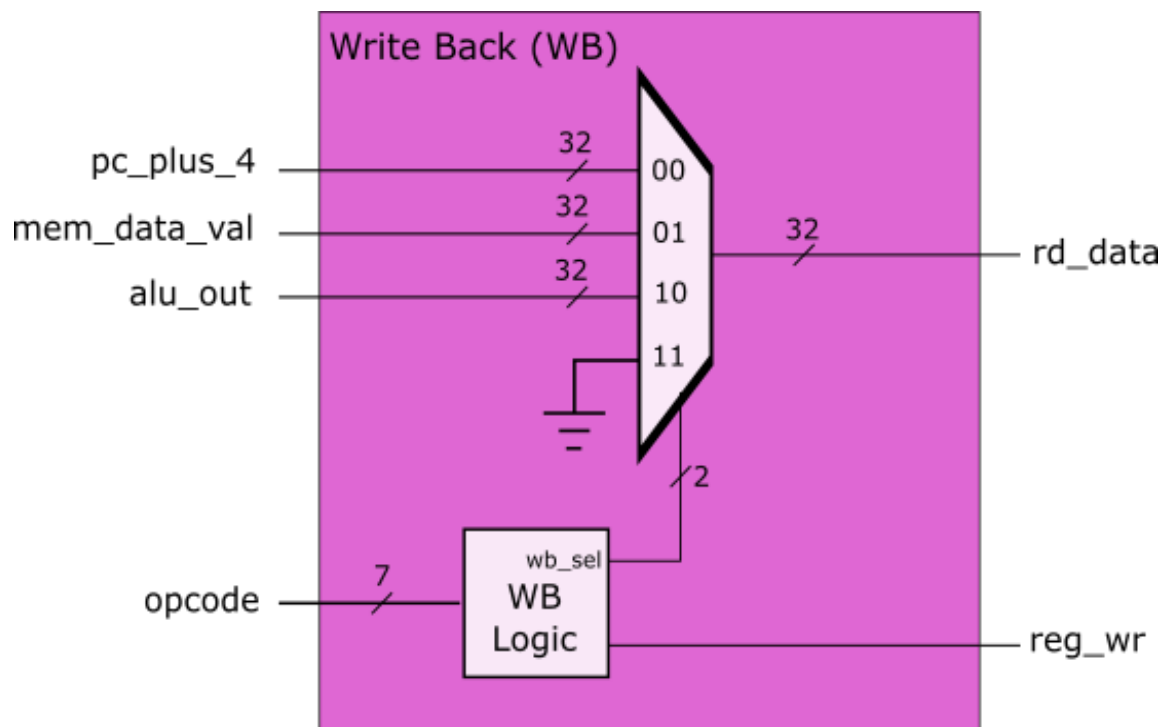


Figure 3.11: Write Back Module Design

Register Bank Module (RB)

The Register Bank module is the module that contains the 32 32-bit registers defined by RISC-V and the circuitry required to manage them. Figure 3.12 presents the design of the Register Bank module. Note: there is no physical register "x0" as defined by RISC-V as this register is always "0". The management circuitry in the Register Bank has been designed such that if register "x0" is selected, the Register Bank will output "0" and if the destination register is "x0", the Register Bank will do nothing at all. This module's inputs are on the left and its outputs are on the right, these are:

- **rs1_addr** - This signal is the address of the first source register generated by the Instruction Decoder
- **rs2_addr** - This signal is the address of the second source register generated by the Instruction Decoder
- **rd_data** - This signal is the data to be stored in the destination register, selected by the Write Back module
- **ce** - Chip enable, the system will not do anything unless this signal is asserted or "1"

- **rd_addr** - This signal is the address of the destination register generated by the Instruction Decoder
- **reg_wr** - This signal is the write signal for the destination register. It lets the Register Bank know whether or not to write the value of "rd_data" to the register located at "rd_addr".
- **rs1_data** - This signal is the data contained in the register selected by "rs1_addr"
- **rs2_data** - This signal is the data contained in the register selected by "rs2_addr"
- **reg_bank** - This signal is used as a tap for a future debugging modules that will be added to the processor

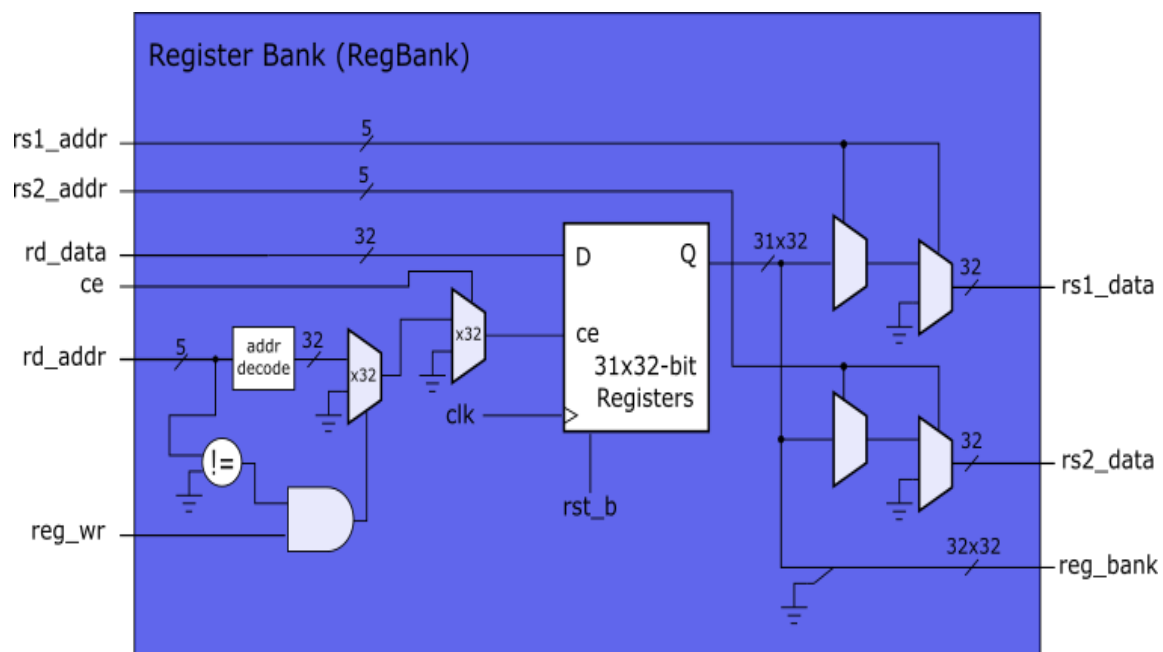


Figure 3.12: Register Bank Module Design

The Register Bank module must manage two aspects of the registers. It must handle returning the data in the registers selected by "rs1_addr" and "rs2_addr". It must also handle writing to the destination register selected by "rd_addr". To handle the source register selection, the Register Bank first isolates one of the 31 physical registers based on the addresses. The Register Bank then selects either the isolated register or "0" if the register address is "0". To handle writing

to the destination register, the Register Bank first checks if "rd_addr" is equal to "0", if it is, nothing will happen. If "rd_addr" is not equal to "0", the Register Bank will check if "reg_wr" is asserted. If it is not, nothing will happen. If it is asserted, the Register Bank decodes the "rd_addr" to assert 1 of 32 chip enable lines, these are then tied to the chip enable signals of the physical registers themselves.

3.2 Implementation and Testing

Taking the architecture designed in section 3.1.3 and making it a reality entails converting the design elements into VHDL modules and constructs. The implementation of the Instruction and Data memory was already produced for the SCC in section 2.4 of chapter 2. These Instruction and Data memories were re-used for this RISC-V processor. The processor was implemented in Xilinx's Vivado Tool Suite [51]–[53]. The VHDL implementation of this processor was organised such that it was as modular as possible. This will give students the ability to easily alter the implementations of each design element and the functions within these design elements. All of the VHDL files and the Xilinx Vivado project files are contained in the GitHub repository associated with this project (see appendix A.1). The VHDL files created and the implementation hierarchy is as follows:

- **RISC-V_Top** - This VHDL module represents the entire processor, this module contains all the other modules in the form of instantiations.
 - **RISC-V_IF** - This VHDL module represents the Instruction Fetch module, all of the circuitry in the Instruction Fetch module will be contained here
 - **RISC-V_ID** - This VHDL module represents the Instruction Decode module, all of the circuitry in the Instruction Decode module will be contained here
 - **RISC-V_EX** - This VHDL module represents the Execution module, all of the circuitry in the Execution module will be contained here

- **RISC-V_MEM** - This VHDL module represents the Memory Management module, all of the circuitry in the Memory Management module will be contained here
- **RISC-V_WB** - This VHDL module represents the Write Back module, all of the circuitry in the Write Back module will be contained here
- **RISC-V_RegBank** - This VHDL module represents the Register Bank module, all of the circuitry in the Register Bank module will be contained here

3.2.1 Implementation

RISC-V_Top

RISC-V_Top is known as a hierarchical component. In HDL convention, a hierarchical component is a module that only instantiates other modules, it does not contain any VHDL constructs that define circuitry. Hierarchical components are used as "wrappers" to wrap up all the components of a system into one "box" and are generally good practice as they encourage more modular designs.

```

93
94  RISC_V_IF_i: RISC_V_IF
95  Port map(
96      clk      => clk,
97      rst_b    => rst_b,
98      ce       => ce,
99      mem_busy => mem_busy,
100     npc      => EX_branch_out,
101     pc_plus_4 => IF_pc_plus_4,
102     pc       => IF_pc
103  );
104
105  RISC_V_ID_i: RISC_V_ID
106  Port map(
107     instruction => instr_in,
108     rs1         => ID_rs1_addr,
109     rs2         => ID_rs2_addr,
110     rd         => ID_rd_addr,
111     f3         => ID_f3,
112     f7         => ID_f7,
113     opcode     => ID_opcode,
114     immediate  => ID_immediate
115  );

```

Figure 3.13: Screenshot of IF and ID instantiations in RISC-V_Top

Figure 3.13 presents the instantiations of the Instruction Fetch and Instruction Decoder. These instantiations direct the VHDL compiler to "pull in" the circuitry of the modules into "RISC-V_Top". The "port map" syntax is used to direct the VHDL compiler on how to connect the inputs and outputs of the modules to the existing connections in "RISC-V_Top". The convention of "moduleName_signalName" is used to describe where a signal in "RISC-V_Top" has originated. In figure 3.13, the outputs of "RISC-V_IF" are named "IF_signalName" to denote that these signals are outputs of the Instruction Fetch module.

RISC-V_IF

The implementation of the Instruction Fetch module consists of two types of VHDL constructs; a "process" and an "assign" statement. Figure 3.14 presents the implementation of the Instruction Fetch module. The "process" construct is used to model the program counter. The "process" contains a 32-bit D-Flip-Flop with "ce" and "mem_busy" as chip enable signals. The "assign" statements are used to generate the two outputs of the Instruction Fetch module. The output of the program counter is passed to "pc" and has 4 added to it to produce "pc_plus_4".

```

36
37   current_pc_Assign: pc      <= pc_internal;
38   next_pc_Assign:  pc_plus_4 <= std_logic_vector(unsigned(pc_internal) + 4);
39
40   program_counter_Reg: process(clk, rst_b) -- Register for the program counter
41   begin
42       if rst_b = '0' then
43           pc_internal <= (others => '0');
44       elsif rising_edge(clk) then
45           if ce = '1' and mem_busy = '0' then
46               pc_internal <= npc;
47           end if;
48       end if;
49   end process;

```

Figure 3.14: Screenshot of the program counter the Instruction Fetch module

RISC-V_ID

The implementation of the Instruction Decode module consists of two types of VHDL constructs similar to the Instruction Fetch module. Figure 3.15 presents a section of the implementation of the Instruction Decode module. The "assign" statements provide the simple "tapping" of the instruction by assigning segments of the instruction to the output signals. The figure presents a section of the immediate generation logic. It can be seen how the "I-type" immediate is sign-extended. This sign-extension only requires tapping certain sections of the instruction and placing it into certain sections of the immediate instruction.

```

43
44   rs1_Assign:    rs1    <= instr(19 downto 15); -- Decoding rs1 operand
45   rs2_Assign:    rs2    <= instr(24 downto 20); -- Decoding rs2 operand
46   rd_Assign:     rd     <= instr(11 downto 7);  -- Decoding rd operand
47   f3_Assign:     f3     <= instr(14 downto 12); -- Decoding funct3 operand
48   f7_Assign:     f7     <= instr(31 downto 25); -- Decoding funct7 operand
49   opcode_Assign: opcode <= instr(6 downto 0);  -- Decode opcode operand
50
51
52   --immediate_generator_Logical: process(instr)
53   begin
54       immediate <= (others => '0'); --default assignment
55
56       case instr(6 downto 2) is
57       when "11001" | "00000" | "00100" => -- I-immediate
58           immediate(31 downto 11) <= (others => instr(31));
59           immediate(10 downto 5)  <= instr(30 downto 25);
60           immediate(4 downto 1)   <= instr(24 downto 21);
61           immediate(0)            <= instr(20);
62

```

Figure 3.15: Screenshot of decode logic in the Instruction Decode module

RISC-V_EX

The implementation of the Execution module has the most substantial amount of VHDL code out of all the modules in the processor. There is a corresponding "process" construct for each of the design elements present in the Execution module design, section 3.1.3. figures 3.16 and 3.17 present the implementation of the ALU and Branch Detection Unit. The implementation of the Function Decode and multiplexers are just a series of case statements used to implement multiplexers, their implementation is short and not as interesting as the ALU and Branch Detection Unit.

The implementation of both the ALU and the Branch Detection Unit consist of one large case statement that is dependant on "alu_op" and "branch_op" respectively. Depending on the value of "alu_op" or "branch_op" a different line of VHDL code will be executed in the ALU or Branch Detection Unit. Each of the 12 functions the ALU can perform and the 7 function the Branch Detect Unit can perform are listed within their respective case statements.

```

50 alu_Logics: process(A, B, alu_op, compare_cond, rs2_addr) -- Logic containing all ALU functions
51 begin
52     result <= (others => '0'); -- Default assignment
53
54     case alu_op is
55     when "0000" => result <= std_logic_vector(signed(A) + signed(B)); -- Addition
56     when "0001" => result <= std_logic_vector(signed(A) - signed(B)); -- Subtraction
57     when "0010" => result <= A and B; -- AND
58     when "0011" => result <= A or B; -- OR
59     when "0100" => result <= A xor B; -- XOR
60     when "0101" => result <= std_logic_vector(shift_left(unsigned(A), to_integer(unsigned(rs2_addr))));
61     when "0110" => result <= std_logic_vector(shift_right(unsigned(A), to_integer(unsigned(rs2_addr))));
62     when "0111" => result <= std_logic_vector(shift_right(signed(A), to_integer(unsigned(rs2_addr))));
63     when "1000" => result <= std_logic_vector(shift_left(unsigned(A), to_integer(unsigned(B))));
64     when "1001" => result <= std_logic_vector(shift_right(unsigned(A), to_integer(unsigned(B))));
65     when "1010" => result <= std_logic_vector(shift_right(signed(A), to_integer(unsigned(B))));
66     when "1011" => -- Set less than
67         case compare_cond is
68         when '0' => null;
69         when '1' => result <= X"00000001";
70         when others => null;
71         end case;
72     when others => null;
73     end case;
74 end process;
75

```

Figure 3.16: Screenshot of the ALU in the Execution Unit

```

77 branch_detection_Logics: process(A, B, branch_op)
78 begin
79     compare_cond <= '0'; -- Default assignment
80
81     case branch_op is
82     when "000" => if A = B then -- BEQ
83         compare_cond <= '1';
84     end if;
85
86     when "001" => if A /= B then -- BNE
87         compare_cond <= '1';
88     end if;
89
90     when "010" => if signed(A) < signed(B) then -- BLT
91         compare_cond <= '1';
92     end if;
93
94     when "011" => if signed(A) >= signed(B) then -- BGE
95         compare_cond <= '1';
96     end if;
97
98     when "100" => if unsigned(A) < unsigned(B) then -- BLTU
99         compare_cond <= '1';
100     end if;
101
102     when "101" => if unsigned(A) >= unsigned(B) then -- BGEU
103         compare_cond <= '1';
104     end if;
105
106     when "110" => compare_cond <= '1'; -- JAL and JALR
107
108     when others => null;
109     end case;
110 end process;

```

Figure 3.17: Screenshot of the Branch Detection Unit in the Execution Unit

RISC-V_MEM

The implementation of the Memory Management module is relatively short in terms of functionality. However due to the different number of load and store instructions, the case statements used to decode the "opcode" and "f3" codes are quite large. Figure 3.18 presents a section of the Memory Management module's implementation. It can be seen that when bits 6 down to 2 of "opcode" are "0" a load instruction is executed. When "f3" is equal to "0" this is a load byte instruction. To implement the load byte instruction, the most significant bit of the byte coming in on "mem_data_in" is placed into bits 31 down to 8 of "data_val". The byte itself is then placed into the least significant 8 bits of "data_val". The Memory Management module also sets "mem_size" to "0" as this is the size code for bytes. The rest of the Memory Management module is very similar to this one section.

```

34
35 mem_ctrl_logic: process(opcode, f3, mem_data_in, rs2_data)
36 begin
37     data_val <= (others => '0'); -- Default assignment
38     mem_data_out <= (others => '0');
39     mem_rd_wr <= '0';
40     mem_valid <= '0';
41     mem_size <= "00";
42
43     case opcode(6 downto 2) is
44     when "00000" => -- Load operation
45         mem_valid <= '1';
46
47         case f3 is
48         when "000" => -- LB
49             data_val(31 downto 8) <= (others => mem_data_in(7));
50             data_val(7 downto 0) <= mem_data_in(7 downto 0);
51             mem_size <= "00";
52

```

Figure 3.18: Screenshot of a section of the Memory Management Module

RISC-V_WB

The implementation of the Write Back module consists of two parts; decoding the "opcode" and the write back multiplexer. Figures 3.19 and 3.20 present the implementation of each of these parts. The logic that decodes the "opcode" signal simply contains a case statement that asserts certain values of "wb_sel" for certain instructions. These values can be seen in the figure. The write back

multiplexer itself is then a smaller case statement dependant on "wb_sel", where "rd_data" will take on the value of the data value selected.

```

34 wb_sel_reg_wr_Logic: process(opcode) -- Generate the mux selection and register write signal
35 begin
36     wb_sel <= "11"; -- Default assignments, don't write back to registers
37     reg_wr <= '0';
38
39     case opcode(6 downto 2) is
40         when "01101"|"00101"|"00100"|"01100" => -- LUI, AUIPC, Arithmetic, Logical
41             wb_sel <= "00";
42             reg_wr <= '1';
43
44         when "00000" => -- Load
45             wb_sel <= "01";
46             reg_wr <= '1';
47
48         when "11011"|"11001" => -- JAL, JALR
49             wb_sel <= "10";
50             reg_wr <= '1';
51
52         when others => null;
53     end case;
54 end process;

```

Figure 3.19: Screenshot of the "wb_sel" generation logic implementation

```

56 rd_data_Assign: process(wb_sel, data_mem_in, alu_out, pc_plus_4) -- Write back mux
57 begin
58     case wb_sel is
59         when "00" => rd_data <= alu_out;
60         when "01" => rd_data <= data_mem_in;
61         when "10" => rd_data <= pc_plus_4;
62         when "11" => rd_data <= X"00000000";
63         when others => null;
64     end case;
65 end process;

```

Figure 3.20: Screenshot of the Write Back multiplexer implementation

RISC-V_RegBank

The implementation of the Register Bank module consists of three parts; the registers, the write logic and read logic. Figures 3.21, 3.22 and 3.23 present the implementations of the registers, write logic and read logic for source register 1. The physical registers themselves are implemented as a 2-dimensional array of D-Flip-Flop which is 32 in height and 32 in width. The write logic for the registers is implemented as a simple combinational logic block which will only update the value of the selected register if "reg_wr" is asserted and "rd_addr" is not equal to "0". The read logic for the first source register is simply a piece of

combinational logic which checks if "rs1_addr" is zero, if so, "rs1_data" is set to "0", otherwise the value of the register at "rs1_addr" is placed on "rs1_data". The same logic is used for the second source register.

```

42 state_Reg: process(clk, rst_b) -- State registers for our 32 32-bit registers
43 begin
44     if rst_b = '0' then
45         int_reg_bank <= (others => (others => '0'));
46     elsif rising_edge(clk) then
47         if ce = '1' then
48             int_reg_bank <= cmb_reg_bank;
49         end if;
50     end if;
51 end process;

```

Figure 3.21: Screenshot of the implementation of the registers

```

53 wr_Logic: process(reg_wr, rd_addr, rd_data, int_reg_bank) -- The write logic for our registers
54 begin
55     cmb_reg_bank <= int_reg_bank;
56
57     if reg_wr = '1' and rd_addr /= "00000" then
58         cmb_reg_bank(to_integer(unsigned(rd_addr))) <= rd_data;
59     end if;
60 end process;

```

Figure 3.22: Screenshot of the implementation of the write logic for the registers

```

62 rs1_data_Assign: process(rs1_addr, int_reg_bank) -- First operand read logic
63 begin
64     if rs1_addr = "00000" then
65         rs1_data <= (others => '0');
66     else
67         rs1_data <= int_reg_bank(to_integer(unsigned(rs1_addr)));
68     end if;
69 end process;
70

```

Figure 3.23: Screenshot of the implementation of the read logic of the registers

3.2.2 Testing

Submodule Testbenches

Each of the sub-modules of this RISC-V processor were tested with their own VHDL testbenches. Due to the fact that each module only contained combinational logic, complicated formal verification methodologies were not necessary. The testbenches created simple emulated the signals that the module in question would receive in practice during the execution of the processor. The sub-module testbenches can be found on the GitHub repository associated with

this project (see appendix [A.1]). To create a testbench to test a module, the module must first be instantiated in the testbench. From here the module is connected to stimulus signals that the testbench will place values onto to simulate the typical execution of the processor. Figure 3.24 illustrates the testbench for the Instruction Decoder module. Lines 42-50 illustrate the instantiation of the Instruction Decoder. This syntax is similar to that in the implementation of the "RISC-V_Top" module in section 3.2.1. The "port map" syntax in the figure connects the inputs and outputs of the Instruction Decoder to the stimulus signals of the testbench. Line 52-62 illustrate the stimulus generation of the testbench. This figure illustrates the "instruction" input being set to "0", this is to simulate the processor being connected to an uninitialised memory. 20 nanoseconds later the "instruction" input is set to "00A07013", this is a "ANDI x0, x0, A" instruction. Figure 3.25 illustrates the waveform generated by this testbench. It can be seen that the Instruction Decoder successfully decodes the "ANDI x0, x0, A" instruction into it's constituent fields i.e. opcode = "13", f3 = "7", rd = "x0", rs1 = "x0" and immediate = "0000000A".

```

42      dut : RISC_V_ID
43      port map (instruction => instruction,
44                rs1        => rs1,
45                rs2        => rs2,
46                rd         => rd,
47                f3         => f3,
48                f7         => f7,
49                opcode     => opcode,
50                immediate  => immediate);
51
52      stimuli : process
53      begin
54
55          instruction <= (others => '0');
56
57          wait for 20ns;
58
59          instruction <= X"00A07013";
60
61          wait;
62      end process;

```

Figure 3.24: Screenshot of the testbench for the Instruction Decoder

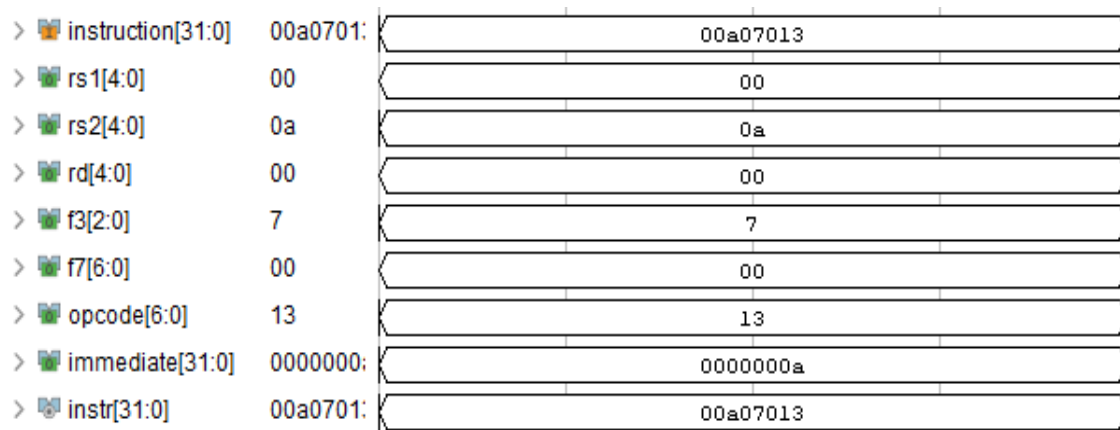


Figure 3.25: Screenshot of the waveform generated by the simulation of the testbench in figure 3.24

Top Level Testbench

A top level testbench was re-used from the SCC (see section 2.4.1 of chapter 2), for the processor as a one-stop shop test facility. The testbench for the entire processor instantiates the processor with the memories re-used from the SCC. The testbench allows users to test any program wished to be executed on the processor. The contents of the instruction memory need only be altered to contain the new program. This testbench can be found in the GitHub repository associated with this project (see appendix [A.1]). Figure 3.26 illustrates the array that is used to initialise the instruction memory. There is currently a simple compiled from C that adds two variables "a" and "b" and stores their result in "c".

```

signal instrArray: array128x32 :=
(
  X"ff010113", X"00100793", X"00f12623", X"00200793", X"00f12423", X"00012223", X"00c12703", X"00812783",
  X"00f707b3", X"00f12223", X"00412783", X"00078513", X"01010113", X"00008067", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000",
  X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000", X"00000000"
);

```

Figure 3.26: Screenshot of the VHDL formatted hexadecimal representation of the program in the testbench

When the simulation of the processor is run this array is taken and loaded in via testbench to the instruction memory. When this information has successfully been stored in the instruction memory. Figure 3.27 illustrates the top level testbench code that runs the processor itself once the instruction memory has been loaded. Lines 351-354 reset the processor to place it into a known state i.e. ready to begin execution of a new program. Lines 356-361 step the program every clock period for 50 clock periods to illustrate how the processor reacts to instruction by instruction execution. Finally the processor is let use using the "runAll" signal which permanently asserts the "ce" signal that governs the operation of most of the sequential elements in the processor.

```

349 --reset doesn't affect instr mme, breakpoint mem or data & stack mem
350 -- assert rst to clear PC and reg
351 rst      <= '1';
352 wait for clkPeriod;
353 rst      <= '0';
354 wait for clkPeriod;
355
356 for j in 1 to 50 loop
357     step <= '1'; -- single step assert
358     wait for clkPeriod;
359     step <= '0'; -- single step deassert
360     wait for clkPeriod;
361 end loop;
362
363
364 runAll <= '1'; -- single step assert
365 wait for 500*clkPeriod;
366
367 report "simulation done"; -- o/p msg to sim transcript
368 endOfSim <= true;
369 wait; -- will wait forever
370 END PROCESS;

```

Figure 3.27: Screenshot of the top level testbench code that runs the processor

Chapter 4

Learning Tools

This chapter details the learning tools developed alongside the RISC-V processor: the application development flow using the RISC-V GNU C Compiler, the online lessons on viciLogic and the design of an online RISC-V IDE to be implemented on viciLogic.

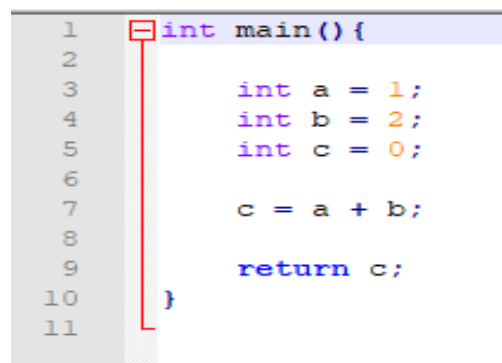
4.1 Compiler/ Vivado Project

To allow users to utilise the processor designed effectively, an application development flow was created by taking advantage of the RISC-V GNU C Compilers [15], these compilers were designed by the RISC-V Foundation to target any possible combination of RISC-V ISAs. These compilers feature separate compile and assembly functions which results in users having the ability to:

- Compile user C code to RISC-V assembly to learn about C compilers
- Compiler user C code directly to RISC-V binary to place into the instruction memory of the processor to be executed
- Assemble user RISC-V assembly code into RISC-V binary to place into the instruction memory of the processor to be executed
- Assemble user RISC-V assembly code to learn about RISC-V assembler and linker programs

To simplify this process for users an installation and use guide was created. These installation and use guides can be found on the GitHub repository associated with this project [A.1]. The RISC-V GNU C Compilers were created to

be used on Linux machines. The installation on Linux is very straight forward requiring very few alternate program and libraries to be installed bar GCC itself. However, because these compilers were designed for Linux, some extra installation is required for Windows machines. Windows machines must "trick" the RISC-V C Compilers into thinking they are being installed by using a fake Linux environment. This fake Linux environment is MSYS2 [10]. How to install MSYS2 and how to install the RISC-V C Compilers is all featured in the GitHub repository [A.1]. Via the MSYS2 Linux environment the RISC-V compilers can be used to compile any user C or RISC-V assembly programs. Figure 4.1 illustrates an example C program, figure 4.2 illustrates the compiled RISC-V assembly code. The scripts present in the use guide were re-used from the SCC RISC-V processor described in section 2.4.2 of chapter 2. These scripts allow users to do a full compilation from C down to binary or from assembly down to binary. These scripts automatically call the RISC-V Compilers in the correct fashion. Each of the files generated by these scripts is detailed in the use guide.



```
1 int main() {
2
3     int a = 1;
4     int b = 2;
5     int c = 0;
6
7     c = a + b;
8
9     return c;
10 }
11
```

Figure 4.1: Screenshot of a simple C program

```

1  .file "add_var.c"
2  .option nopic
3  .text
4  .align 2
5  .globl _start
6  .type _start, @function
7  _start:
8      addi    sp, sp, -16
9      li     a5, 1
10     sw     a5, 12(sp)
11     li     a5, 2
12     sw     a5, 8(sp)
13     sw     zero, 4(sp)
14     lw     a4, 12(sp)
15     lw     a5, 8(sp)
16     add    a5, a4, a5
17     sw     a5, 4(sp)
18     lw     a5, 4(sp)
19     mv     a0, a5
20     addi    sp, sp, 16
21     jr     ra
22     .size   _start, .-_start
23     .ident  "GCC: (GNU) 8.2.0"
24

```

Figure 4.2: Screenshot of the assembly program produced by the RISC-V C Compilers

The simple C program in figure 4.1 creates 3 variables; $a = 1$, $b = 2$ and $c = 0$. The program then adds "a" and "b" and stores the result in "c". The function then returns "c". The RISC-V assembly equivalent or compilation result of this C program in figure 4.2 demonstrates what the RISC-V C Compilers produce. Lines 1 through 6 of figure 4.2 illustrate some pre-processor directives such as: what C file did this program come from, what byte alignment does the instruction memory have, where is the start of the program? Line 7 denotes the beginning of the program. Lines 8 through 21 are the assembly instructions that equate to this C program. The RISC-V convention is to utilise a stack to store any variables present in the program and to utilise registers $a0, a4, a5$ as function arguments, these are in-fact registers $x10, x14$ and $x15$. All of these software conventions can be found on [16, pg. 109]. This stack is located at the very top of memory and grows down into the memory space. Line 8 illustrates the stack being setup by assigning the current maximum stack value into "sp" or "stack pointer" which is the RISC-V convention for using register $x2$. Lines 9 through 13 illustrate our variables being created, just like in the C program, each of the variables is created with an "li" or "load immediate instruction" and is put into the stack. Lines 14 through 19 pull back in the values for "a" and "b" from the stack, add them and place them back onto the stack in the place

of "c". The result is also passed to register a0 as the return value. Line 21 then jumps to the instruction address stored in register ra or "return address", this register has not been altered during this program and so it still contains "0". There program therefore jumps back to line 7 and executes the program all over again. Lines 22 and 23 are post-processor directives that are not relevant for application development.

This RISC-V assembly program must then be run through the RISC-V Assembler program and the RISC-V Linker program. The Assembler program first takes these pseudo-code RISC-V instructions and translates them to literal RISC-V instruction. One pseudo-code instruction may translate to sequence of literal RISC-V instructions. The Linker program then calculates any branch addresses and re-configures any register allocations after the Assembler program has run. Figure 4.3 illustrates the same program after it has been passed through the Assembler and Linker programs.

```

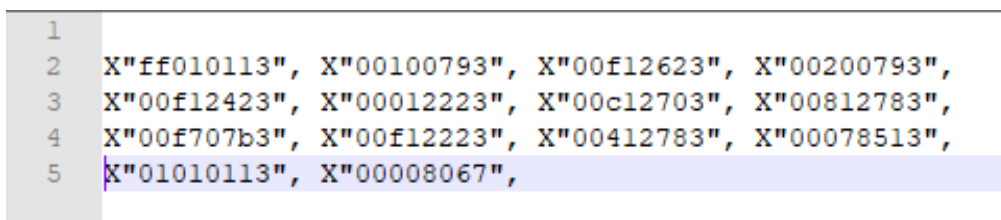
1
2  add_var.out:      file format elf32-littleriscv
3
4
5  Disassembly of section .text:
6
7  00000000 <_start>:
8      0:  ff010113      addi    sp,sp,-16
9      4:  00100093      li     ra,1
10     8:  00112623      sw     ra,12(sp)
11     c:  00200093      li     ra,2
12    10:  00112423      sw     ra,8(sp)
13    14:  00012223      sw     zero,4(sp)
14    18:  00c12703      lw     gp,12(sp)
15    1c:  00812083      lw     ra,8(sp)
16    20:  001700b3      add    ra,gp,ra
17    24:  00112223      sw     ra,4(sp)
18    28:  00412083      lw     ra,4(sp)
19    2c:  00008513      mv     a0,ra
20    30:  01010113      addi    sp,sp,16
21    34:  00008067      ret
22

```

Figure 4.3: Screenshot of the assembly program produced by the RISC-V Assembler and Linker

It can be seen in figure 4.3 that there is a lot more useful information about the program now given. The location of each instruction in instruction memory is provided along the left side of the figure. The start address is always "0" for the scope of this project but this can be altered by provided function arguments to the assembler and linker scripts. Each of the pseudo-instruction from figure 4.2 and the equivalent hexadecimal value for each of these is present. The Assembler and Linker programs tend not to directly translate the pseudo-instructions as they are written. In this example, "li ra, 1" is in-fact an "addi ra, zero, 1" instruction.

The scripts that were re-used from the previous SCC RISC-V processor then take these hexadecimal values and prepare them into a format that can be copied and pasted into the testbench described in section 3.2.2 of chapter 3. Figure 4.4 illustrates the program ready to be placed in the testbench. The VHDL syntax for defining a 32-bit hexadecimal value is "X"number"" where number is however many hexadecimal digits are present in the value, in this case it's 8.



```

1
2  X"ff010113", X"00100793", X"00f12623", X"00200793",
3  X"00f12423", X"00012223", X"00c12703", X"00812783",
4  X"00f707b3", X"00f12223", X"00412783", X"00078513",
5  X"01010113", X"00008067",

```

Figure 4.4: Screenshot of the VHDL formatted hexadecimal representation of the program

These lines are then copied and pasted into an array used as to pre-load the instruction memory at the beginning of a simulation run. Figure 4.5 illustrates this array in the testbench. From here the execution of this program can be explored by running a simulation of the processor and investigating the resulting waveform.

RISC-V Processor Introduction

The introduction to this prototype course presents the top level view of the RISC-V processor and its features, introducing the user to what these lessons contain and what is being explained. The introduction also details what vi-ciLearn is and how its features will be used in these lessons. Figure 4.6 illustrates the first step of the step of the introduction.

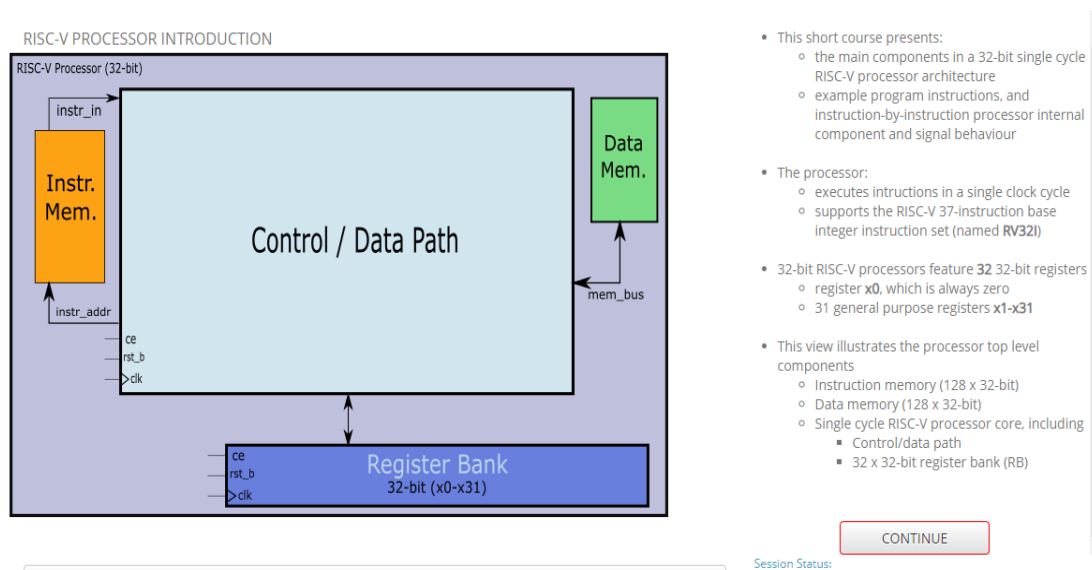


Figure 4.6: Screenshot of the introduction to the RISC-V prototype course

RISC-V 2-instruction Program (ADDI/JAL, add/jump)

This lesson introduces a simple 2-instruction program consisting of an "ADDI" and "JAL" instruction. Where the "ADDI" instruction adds 1 to register x1 and stores the result in x1. The "JAL" instruction then re-directs the program back to the "ADDI" instruction. This results in the "ADDI" instruction being ran continuously until the processor is reset. Figure 4.7 illustrates a step from this lesson. This lesson walks through the program step by step, prompting the user to interact with the program by toggling the clock and resetting the processor. The lesson then provides detailed descriptions of each instruction, explaining the syntax of the assembly instruction and how this information is encoded into the binary representation of the instruction.

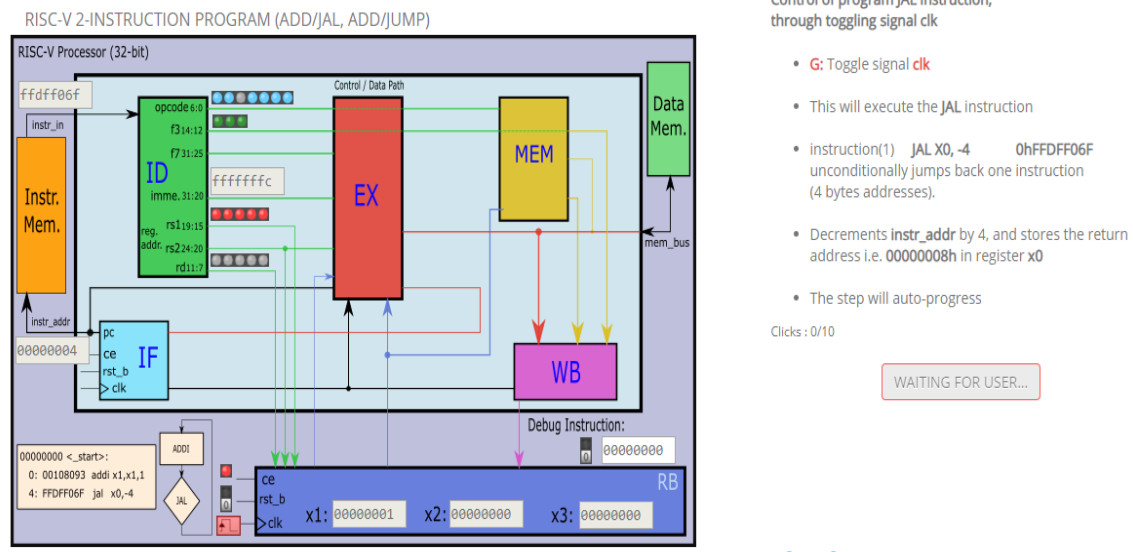


Figure 4.7: Screenshot of the 2-instruction program lesson

RISC-V Sandbox and Challenge

This section of the course presents users with a sandbox step. This sandbox encourages users to input their own instructions into the processor and observe the behaviour of the processor. 4 instructions are provided as suggestions, but the users are also provided with a link to the RISC-V specification which outlines all of the instruction formats from which they can generate their own instructions to run on the processor. Figure 4.8 illustrates this sandbox aspect of the course. When the user is finished with the sandbox they are presented with a challenge to progress. Users must use the instructions provided or their own instructions to place the value "0h5" in register x3.

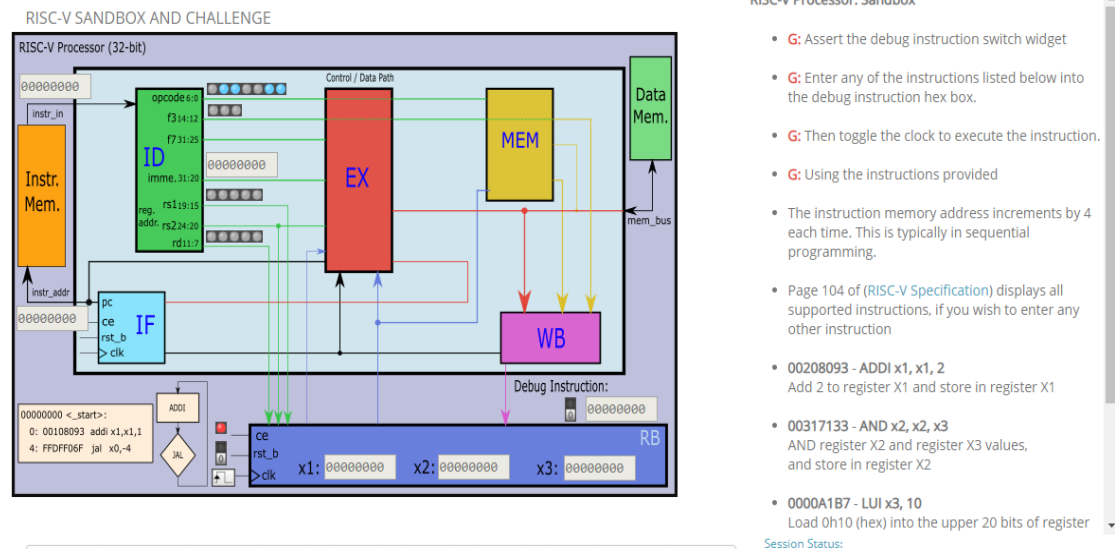


Figure 4.8: Screenshot of the step that provides a sandbox for users

RISC-V Processor Architecture

This lesson introduces the architecture of the processor by briefly explaining the life cycle of an instruction by briefly explain the operations of each of the components of the architecture in the context of the instruction. Figure 4.9 illustrates the first step in the RISC-V Processor Architecture lesson.

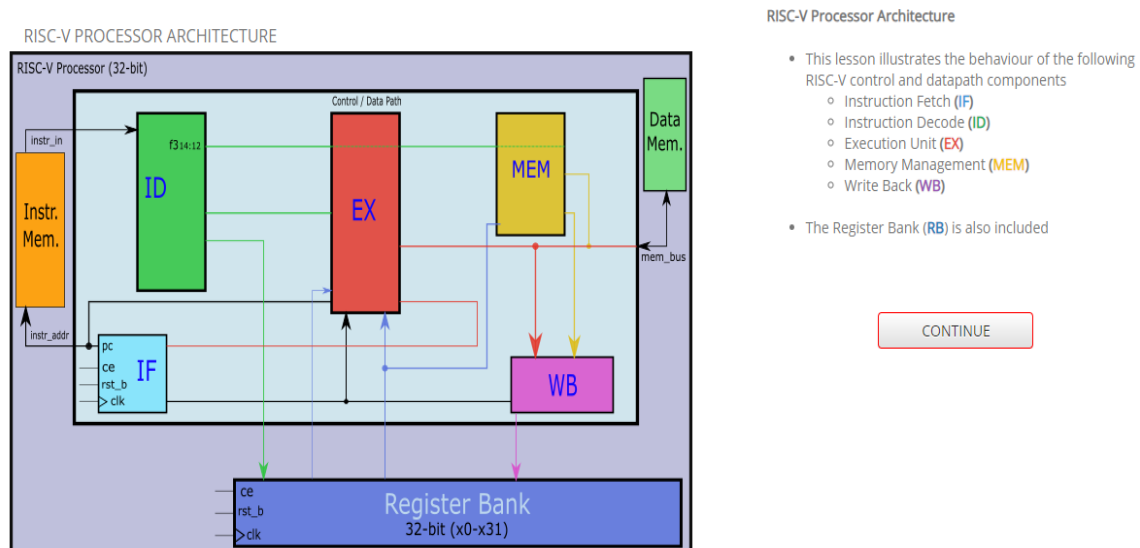


Figure 4.9: Screenshot of the step that explains the architecture of the processor

Control and Data Path Design (ADDI x1, x1, 3 instruction)

This lesson leads on from the previous one by explaining in further detail the design of the Control and Data Path. This is achieved by the combination of an "ADDI x1, x1, 3" instruction and user interaction. Figure 4.10 illustrates the first step in this lesson. This lesson begins with introducing present in the image illustrated in the figure. The lesson directs the user to toggle the clock to observe the behaviour of these new signals as the processor executes the familiar "ADDI x1, x1, 3" instruction. After introducing the behaviour of these new signals to the user, the lesson walks through the context of each set of signals as they propagate into and out of each module, explaining the behaviour of each of the modules in the Control and Data Path. To review this explanation the lesson then prompts the user once again to toggle the clock and observe the processor execute the "ADDI x1, x1, 3" instruction.

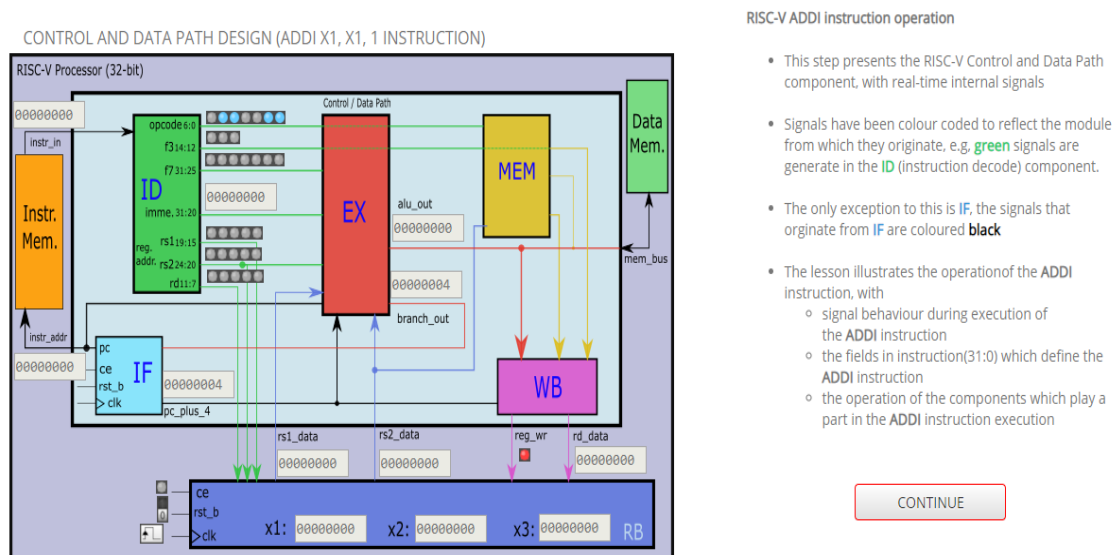


Figure 4.10: Screenshot of the step that explains the design of the Control and Data Path

EX Component Design (ADDI x1, x1, 3 instruction)

This lesson then builds on the previous lesson by detailing the design of the Execution module. The Execution module was chosen as the design to present in this prototype course as it is the most interesting of all the modules present in the Control and Data path. The lesson first introduces the components of the Execution module as can be seen in figure 4.11. The lesson continues on to

prompt the user to toggle the clock to observe the behaviour of the Execution module as the "ADDI x1, x1, 3" instruction is being executed once again. The lesson then describes the design of each of the components and internal signals in the Execution module. The wrap up the lesson the user is finally prompted to toggle the clock to observe the behaviour of the Execution module after receiving this new information.

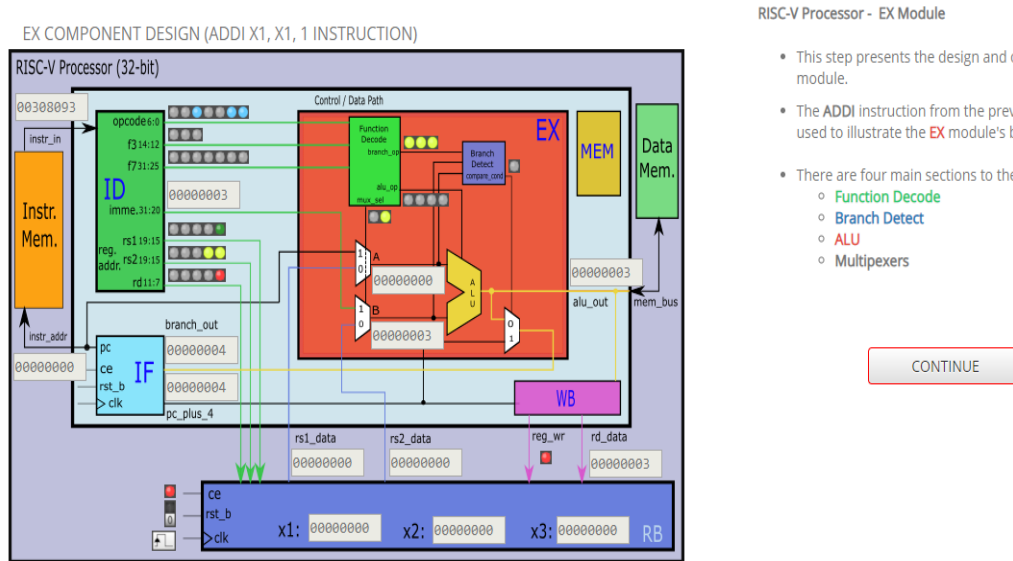


Figure 4.11: Screenshot of the step that explains the design of the Execution Unit

4.3 RISC-V IDE Design

During the initial conception of the project one of the aims of the project was to design and implement an online RISC-V IDE on the viciLogic platform. However, through the research conducted on the viciLogic platform and web-development. It was deemed not feasible to implement this RISC-V IDE alongside the development of the processor, application development flow and the online lessons in the time provided for a final year project. It was decided that the research completed would be used to design the IDE which could then be implemented at later date. The hypothesised RISC-V IDE consists of 4 tabs; the Editor tab, the Programmer's Model tab, the Data Memory tab and the Processor View Tab.

The ViciLogic uses the Django [12] framework to implement its online platform. Django is a python-based web development framework, it provides convention for organising a web-development project, such as handling different applications on a website, organising static assets such as files, CSS style-sheets, JavaScript files, AJAX and jQuery integration and extension of HTML with its built-in template language. Django is a very powerful and well tested framework, ultimately it takes care of the repetitive mundane setup and organisation that comes with setting up a fully-fledged website and allows developers to focus on the development of the application on the website rather than the website itself. Each of the tabs in the RISC-V IDE were designed with this framework in mind.

Editor Tab

The Editor tab would be the entry point of the RISC-V IDE, it would allow the user to upload their C or RISC-V assembly code to the viciLogic server, in the case of the C code, this would be compiled into RISC-V assembly and RISC-V binary code. The uploaded RISC-V assembly code would be compiled into RISC-V binary code. The user would then be able to switch between the different representations of their code, however the ability to edit and re-compile the C and assembly code would be allowed, not the binary code. The user would then be able to click an upload button and a Xilinx ZYNQ-7000 [53] FPGA on a PYNQ-Z2 [51] board in the cloud would then be loaded with the RISC-V processor and the on-board memories would be uploaded with the users code (see figure 4.12 as an example).

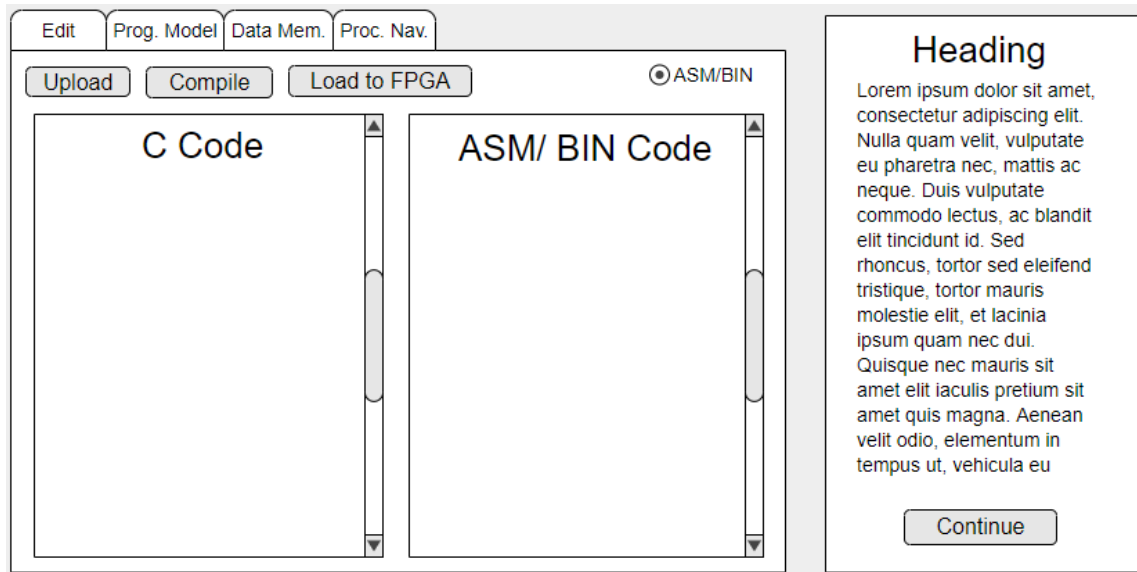


Figure 4.12: Mock-up of the RISC-V IDE Editor Tab

This functionality would be implemented within the viciLearn application on the Django project, the user interface could be implemented in the HTML file used the viciLearn Django template, where the editor itself would be implemented with the open source IDE known as Ace [24], this is an IDE designed to be integrated into websites and web applications. Django has the ability to install support libraries like the one Ace provides. The RISC-V GNU compiler will be installed on the ViciLogic server and would be invoked by a script written to handle the files that are uploaded by the user. This application can then utilize the existing functionality present in viciLogic to interact with the PYNQ-Z2 FPGAs, where-in the user pressing a button would initialise the processor and the user program.

Programmer's Model Tab

The Programmer's Model tab would be the next port of call for a user using RISC-V IDE application. It will allow the user to view and interact with the processor that is now initialised on the FPGA. It would give the ability run the processor at varying clock speeds, reset the processor or step the processor through its operation cycle by cycle. It would also give the user a live view of the register values within the processor, it should also give the user the ability to change the values of the registers on the fly. For convenience the user should be able to change the radix of the displayed values, options such as decimal, hex-

adecimal and binary would be available. The programmers model would also show the user the instruction memory as a list, displaying where in memory the instruction is and highlighting the current instruction. The instruction memory viewer should preferably give the user the ability to apply break points. Editing the register values and applying break points would require edits to the register bank of the processor and the addition of a debug module.

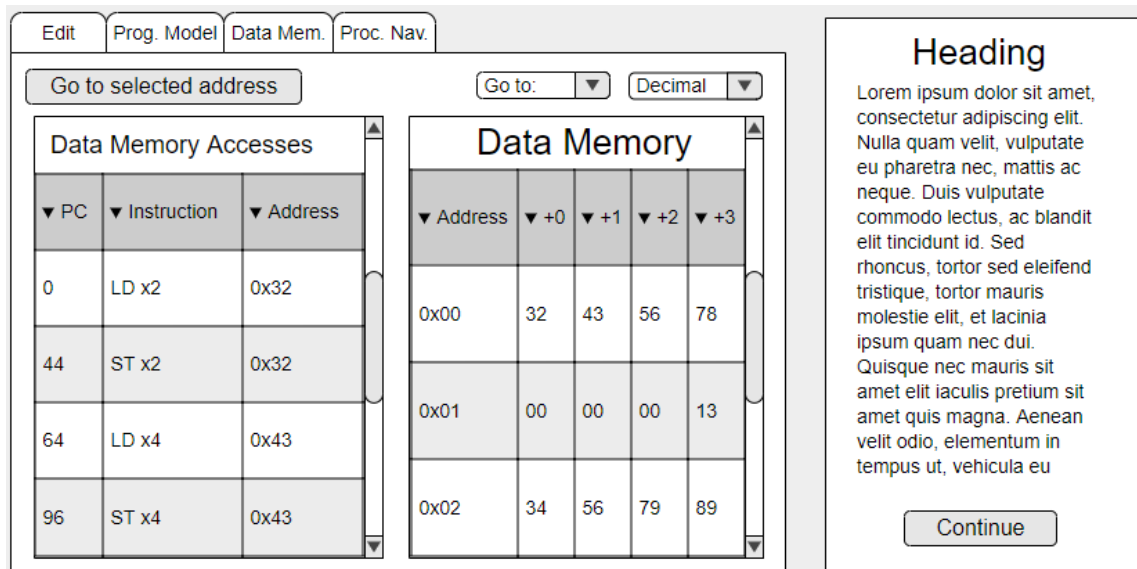


Figure 4.13: Mock-up of the RISC-V IDE Programmer's Model Tab

The Programmers Model tab would be implemented as a user interface like the Editor tab. The programmers model would be accessed by clicking on its respective tab. Pulling the information from the instruction memory and register banks would have to be implemented on the back of the existing functionality already in place to communicate data to and from the FPGA. The implementation of detecting the ticked break points and values of the registers being altered would be JavaScript HTML data object models (DOMs). DOMs are object tree representations (which are directly linked) to the web page HTML. With JavaScript the HTML and CSS elements can be created/removed/altered dynamically.

Data Memory Tab

The Data Memory tab would allow the user to view the RISC-V cores data memory, displaying the data memory, in a byte format, with addresses in multiples of four along the side as rows and +0, +1, +2, +3 columns to represent the respective bytes within that 32-bit region. The tab would also provide a transaction history alongside the data memory to show what accesses occurred (load/store) to what address, what value and at what program counter value did it occur. It would also provide the ability like the programmers tab to change the radix of the displayed values to; decimal, hexadecimal and binary. Considering memories can be quite massive, it should also allow the user to go to specific addresses in the memory in the data memory viewer list.

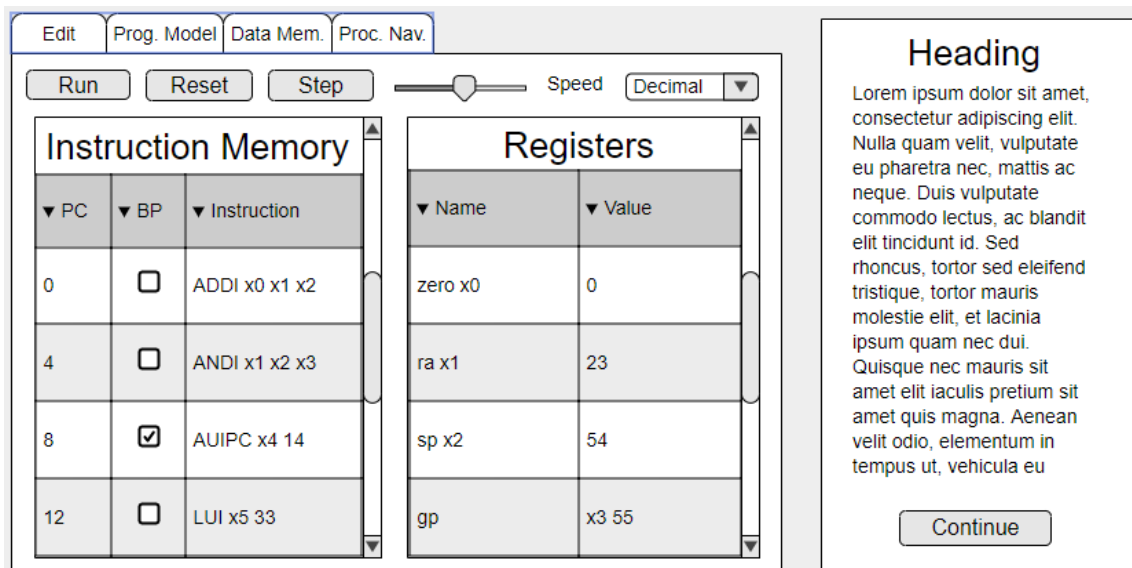


Figure 4.14: Mock-up of the RISC-V IDE Data Memory Tab

This tab like previous tabs would be a HTML element in the viciLearn application and it will also utilise the existing communication system set up between ViciLogic and the FPGAs. This tab would in fact be far simpler to implement than the programmers model as it does not take user input except from drop down menus and specific forms for entering in addresses, no JavaScript HTML DOM would be required for this.

Processor View Tab

The Processor View tab, this is arguably the most useful aspect of the RISC-V IDE application as this user interface will provide an animated view of the RISC-V architecture being implemented in real time. viciLogic already achieves this with the animated views of designs running on the FGPAs in the viciLearn courses. This tab will implement the running of the processor at variable frequencies and the manual stepping of the processors clock and the resetting of the system that the programmers model would implement. This tab will allow the user to explore the overall architecture of the core, zoom in, zoom out, zoom to fit, display signal values. If the user double clicks on any one element, they would be able to look inside that element. As an example, if they double clicked on the Instruction Fetch module, they would then be presented with an animated view of the Instruction Fetch internal design.

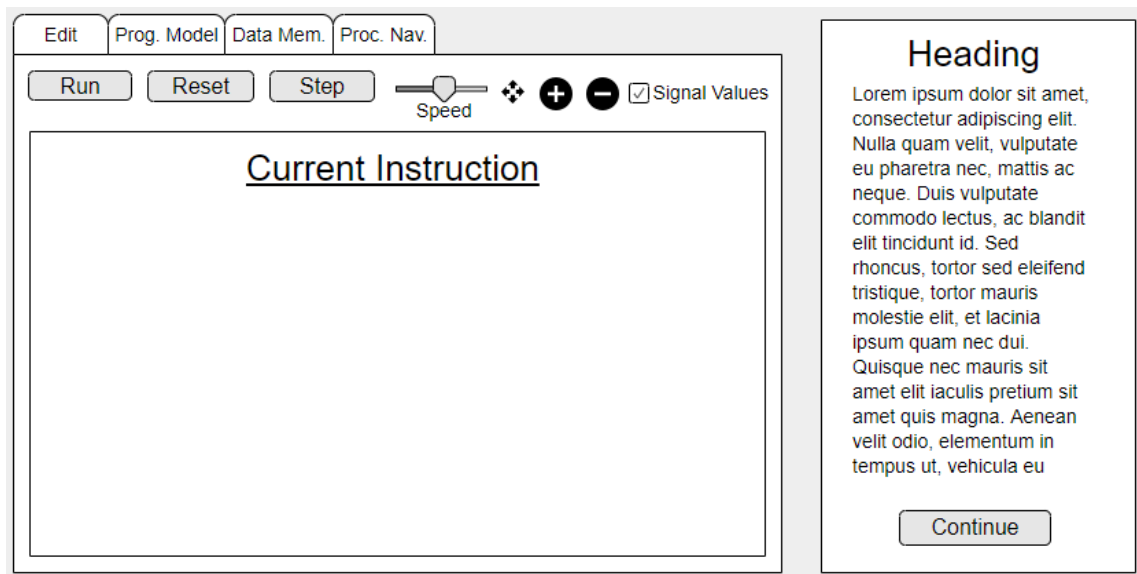


Figure 4.15: Mock-up of the RISC-V IDE Processor View Tab

This tab would be able to re-use the implementation from the Programmers Model and then would implement the animated views from viciLearn. The navigation throughout the modules would be achieved thanks to the fact that the diagrams would be static in the view and so therefore buttons styled to be invisible could be placed overhead but remain on top of the module. JavaScript HTML DOMs could be used to change and alter the diagrams, similar to how viciLearn operates.

Chapter 5

Health and Safety

5.1 Risk to Life and Limb

After conducting a risk assessment (RA)[A.3], generating a standard operating procedure (SOP) [A.2] document and given that the intended use of this project is as an online learning tool and given it is being developed in software, it is deemed that the risk of damage or injury to persons or property is very low.

The risks involved with using this project as regards damage or injury to persons are those associated with prolonged use of LCD screens, keyboards and chairs. If the standard operating procedures [A.2] document and good common practice is followed i.e. taking regular breaks while using a computer for prolonged periods of time, there should be a negligible risk of damage or injury to user's eyes, wrists or spine from viewing LCD screens, using keyboards and sitting in chairs.

5.2 Broader Societal Impact

The broader societal impact of this project will be contained within the demographic of the viciLogic platform and the university as a whole. The only way that this project could have come to disadvantage viciLogic or the university or its persons would be by being copied and sold without the consent of viciLogic or the university. However given that the design of the processor developed during this project is not aimed for any specialized practice i.e. low power, low area or high performance. It is extremely unlikely any malicious

parties will attempt to steal the design as it would provide no benefit.

The only perceived societal impact of this project is a positive one. The aim of the project is to utilize the RISC-V ISA on the viciLogic platform to contribute to the awareness and adoption of RISC-V and digital integrated circuit design as a whole globally.

Chapter 6

Conclusion and Future Scope

This chapter details the conclusion and discussion of the project and the possible future work and scope of the project.

6.1 Conclusion

From the initial conception of this project until its completion, the personal and objective aims of the project were to:

- Expand the awareness and adoption of the RISC-V ISA
- Raise the awareness and interest in digital systems and digital integrated circuit design
- Enhance the existing course material present in the university
- Expand the capabilities of the online learning platform, viciLogic
- Learn about computer architecture and processor design
- Become familiar with and understand in detail the RISC-V ISA
- Learn about how to teach complex topics such as processor design and digital integrated circuit design

The personal and objective aims of this project were achieved by:

- Reviewing an existing RISC-V architecture and developing an updated RISC-V processor under the design constraint of creating a learning tool.

- Generating an application development flow for Windows and Linux
- Generating prototype online lessons on viciLearn to illustrate how to teach the aforementioned topics
- Designing a RISC-V IDE for viciLogic to provide the ultimate learning tool for processor design and application development

The initial scope of the project was to infact implement the RISC-V IDE alongside the RISC-V processor, it's offline application development flow and the online lessons. However, after research was conducted on how to implement the RISC-V IDE and a design was generated. It was deemed that the RISC-V IDE in itself was an entire final year project. Attempting to implement this alongside the RISC-V processor, it's application development flow and the online lessons would not be feasible in the context of a final year project.

However, the research and design of the RISC-V IDE yielded many benefits both for future expansion of viciLogic and personal development. The research completed introduced the areas of web-development and software design. Subjects like backend and frontend development had to be explored to understand how a platform like viciLogic was designed and implemented. Django, HTML, CSS and JavaScript all had to be learned to gain an appreciation for the scale of the viciLogic platform and the work required to implement the RISC-V IDE.

Alongside what was learned about web-development and software design. A great deal was learned and understandings were solidified on the topics of computer architecture, processor design and on how to educate users on these topics. The research conducted on how to design the processor created during this project resulted in understanding all of the design considerations required to create a processor under certain constraints i.e. which ISA should be used, what architectural decisions need to be made etc. The generation of the offline application development flow resulted in a deeper understanding of the complexity of the software systems involved in compiling high-level programming languages like C and C++. A great deal was also learned about how assembler and linker programs handle compiled code to translate a program into a form

that a processor might understand. Creating the online lessons nailed home how difficult it can be to effectively communicate very complex and technical information, giving a greater appreciation for those who spearhead the improvement of education in these areas.

6.2 Future Scope

6.2.1 RISC-V Core

The RISC-V processor core designed during this project has the potential for a plethora of further expansion. This is mainly accredited to RISC-V itself. The following subsections will outline some of the potential expansions that can be made.

Implementing other RISC-V ISAs

The RISC-V ISA features many different useful instruction sets to implement. To truly give students and users a practical and thorough appreciation for the RISC-V ISA, it would be wise to implement further RISC-V ISAs alongside the base integer ISA. At the time of writing this report, only the multiplication/division "M", the atomic "A", the single-precision floating-point "F", the double-precision floating-point "D", the quad-precision floating-point and the compressed instruction sets have been frozen and or finalized [16, pg. 3] and are therefore the only options that are advisable to implement until further instruction sets have been frozen.

It is therefore recommended that if in future further instruction sets are to be implemented that the "M", "C" and "F" instruction sets are implemented. These instructions sets provide both extra functionality and performance over the base integer instruction set, but are also not so complicated like the "A", "D" and "Q" instruction sets that the complexity of understanding these instruction sets and their implementations begins to reduce the amount of useful learning completed.

Implementing the RISC-V CSRs

At the time of writing this report, the base integer instruction set [16] is under ratification to remove the control status register (CSR) instruction and to place these instructions into their own instruction set. It is therefore recommended that when this ratification is completed, that the CSR instructions are implemented.

The CSRs provide useful performance metrics in the form of the RISC-V timers that are defined by the CSR instructions. These timers track: the number of cycles since power on, the number of instructions executed since power on and the time since power on. The CSR address space allows for 4,096 CSRs in total, this in turn allows for many custom CSRs to be implemented. These custom CSRs can come in the form of interrupt vectors, connections to external resources i.e. output LEDs or seven segments displays, debugging or any other types of functionality required not already defined by the RISC-V ISA.

Implementing the RISC-V Debug Specification

As of 2019, SiFive [18], the company who currently spear heads RISC-V processor production, consisting of the inventors of RISC-V itself, generated an open-source debug specification [19] tailored for RISC-V processors. This debug specification details a "debug module" that can be implemented into a RISC-V processor. This debug module has complete control over the RISC-V processors execution. The debug module features:

- Complete control over all registers both general and CSRS,
- In a multi-core implementation any one of the cores can be independently debugged
- The debug module learns everything it needs to know about itself by monitoring the processor it is attached to
- Software breakpointing and hardware single-step execution
- Injection of code into the processor being debugged at any point during execution

Implementing the SiFive TileLink Specification

SiFive [18] also offer an open-source communication protocol specification known as "TileLink" [20]. This communication protocol is designed for communication between RISC-V processors and their memories. However, this specification could be implemented on other ISAs. This specification outlines several conformance levels of varying complexity depending on how complex the memory system of the processor in question is.

TileLink conformance levels feature support for general read/write operations, atomic operations, burst accesses and support for cached memories. The system that this specification would produce would result in a very practical and efficient implementation of a RISC-V processor that could be implemented as an ASIC.

6.2.2 viciLogic Integration

There is a lot of room for expansion following this project on the viciLogic platform. Separate RISC-V courses or lessons can be created on viciLearn for each of the pieces of future work on the processor that were detailed previously. The RISC-V IDE designed in section 4.3 in chapter 4 would be of enormous benefit to users alongside the courses and lessons that are currently and will be available on viciLearn.

viciLogic RISC-V Macro

Two pieces of functionality that could be intergrated with viciLogic in the future before the addition of the RISC-V IDE would be to give users the ability to compile their C or RISC-V assembly in the cloud and to then have this placed into the RISC-V processor's instruction memory. Then via the sandbox [35] and detailed architecture [34] steps that are part of the current RISC-V course detailed in section 4.2 of chapter 4, could be used to run these user programs to allow users to "play" with their own applications and perform their own application development in-browser.

References

- [1] J. Armstrong, C. Cho, S. Shah, and C. Kosaraju, *The VHDL Validation Suite*. ACM, 1990, ISBN: 0-89791-363-9. [Online]. Available: <http://doi.acm.org/10.1145/123186.123190>.
- [2] I. S. Association. (2008). Ieee 1076-2008 - ieee standard vhdl language reference manual, [Online]. Available: <https://standards.ieee.org/standard/1076-2008.html>.
- [3] ———, (2005). Ieee 1364-2005 - ieee standard for verilog hardware description language, [Online]. Available: <https://standards.ieee.org/standard/1364-2005.html>.
- [4] ———, (2017). Ieee 1800-2017 - ieee standard for systemverilog–unified hardware design, specification, and verification language, [Online]. Available: <https://standards.ieee.org/standard/1800-2017.html>.
- [5] ———, (2008). Ieee 754-2008 - ieee standard for floating-point arithmetic, [Online]. Available: <https://standards.ieee.org/standard/754-2008.html>.
- [6] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Springer, 2003.
- [7] O. Community. (2019). Openrisc, [Online]. Available: <https://openrisc.io/>.
- [8] K. S. Crystal Chen Greg Novick. (2000). Risc vs. cisc, [Online]. Available: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscisc/>.
- [9] W. Digital. (2019). Swerv risc-v coretm from western digital, [Online]. Available: https://github.com/westerndigitalcorporation/swerv_eh1.
- [10] A. P. R. Donnelly. (2019). Msys2, [Online]. Available: <https://www.msys2.org/>.
- [11] D. V. L. Eduardo Augusto Bezerra, *Synthesizable VHDL Design for FPGAs*. Springer Science & Business Media, 2013.
- [12] D. S. Foundation. (2019). The web framework for perfectionists with deadlines, [Online]. Available: <https://www.djangoproject.com/>.
- [13] R.-V. Foundation. (2019). Risc-v assembly programmer’s manual, [Online]. Available: <https://github.com/riscv/riscv-asm-manual>.
- [14] ———, (2019). Risc-v foundation, [Online]. Available: <https://riscv.org/>.
- [15] ———, (2019). Risc-v gnu compiler toolchain - risc-v foundation, [Online]. Available: <https://riscv.org/software-tools/risc-v-gnu-compiler-toolchain/>.

- [16] —, (2019). Risc-v instruction set manual, [Online]. Available: <https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.
- [17] A. Hodges, *Alan Turing: The Enigma*. Princeton Univeristy Press, 2014.
- [18] S. Inc. (2019). Home - sifive, [Online]. Available: <https://www.sifive.com/>.
- [19] —, (2019). Risc-v external debug support, [Online]. Available: <https://riscv.org/specifications/debug-specification/>.
- [20] —, (2019). Sifive tilelink specification, [Online]. Available: <https://www.sifive.com/documentation>.
- [21] S. I. Inc. (1992). The sparc architecture manual, [Online]. Available: <https://www.gaisler.com/doc/sparcv8.pdf>.
- [22] D. A. P. John L. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1981, ISBN: 978-0123838728.
- [23] M. H. L. John Shen, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Professional, 2004.
- [24] M. Labs. (2019). Ace - high performance code editor for the web, [Online]. Available: <https://ace.c9.io/>.
- [25] MIPS. (2019). Mips32 architecture, [Online]. Available: <https://www.mips.com/products/architectures/mips32-2/>.
- [26] S. Mittal, *A survey of techniques for designing and managing CPU register file*. Wiley, 2016.
- [27] Preshing. (2019). Atomic vs. non-atomic operations, [Online]. Available: <https://preshing.com/20130618/atomic-vs-non-atomic-operations/>.
- [28] L. Semiconductors. (2019). Latticemico32, [Online]. Available: <http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32>.
- [29] I. O. for Standardization. (2012). Information technology – programming languages – ada, [Online]. Available: <https://www.iso.org/standard/61507.html>.
- [30] A. M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 1936.
- [31] viciLogic. (2019). Course, [Online]. Available: <https://www.vicilogic.com/vicilearn/course/>.
- [32] —, (2019). Home, [Online]. Available: <https://www.vicilogic.com/>.
- [33] —, (2019). Risc-v architecture and applications, [Online]. Available: https://www.vicilogic.com/vicilearn/run_step/?c_id=27.
- [34] —, (2019). Risc-v control and data path design, [Online]. Available: https://www.vicilogic.com/vicilearn/run_step/?c_id=27&c_pos=5.

- [35] —, (2019). Risc-v sandbox and challenge, [Online]. Available: https://www.vicilogic.com/vicilearn/run_step/?c_id=27&c_pos=3.
- [36] —, (2019). Single cycle computer (scc), [Online]. Available: https://www.vicilogic.com/vicilearn/run_step/?c_id=24.
- [37] —, (2019). Single cycle computer (scc) functional partition, [Online]. Available: https://www.vicilogic.com/static/ext/SCC/spec/SCC_FP_A4.pdf.
- [38] —, (2019). Single cycle computer (scc) instruction set listing, [Online]. Available: <https://www.vicilogic.com/static/ext/SCC/spec/SCCInstrSet.pdf>.
- [39] —, (2019). Vicilab, [Online]. Available: <https://www.vicilogic.com/pages/vicilab/>.
- [40] —, (2019). Vicilogic server livestream, [Online]. Available: <https://www.vicilogic.com/static/webcam/stream.html>.
- [41] Wikipedia. (2019). Classic risc pipeline, [Online]. Available: https://en.wikipedia.org/wiki/Classic_RISC_pipeline.
- [42] —, (2019). Complex instruction set computer, [Online]. Available: https://en.wikipedia.org/wiki/Complex_instruction_set_computer.
- [43] —, (2019). Instruction set architecture, [Online]. Available: https://en.wikipedia.org/wiki/Instruction_set_architecture.
- [44] —, (2019). Pipeline (computing), [Online]. Available: [https://en.wikipedia.org/wiki/Pipeline_\(computing\)](https://en.wikipedia.org/wiki/Pipeline_(computing)).
- [45] —, (2019). Processor registers, [Online]. Available: https://en.wikipedia.org/wiki/Processor_register.
- [46] —, (2019). Reduced instruction set computer, [Online]. Available: https://en.wikipedia.org/wiki/Reduced_instruction_set_computer.
- [47] —, (2019). The thumb instruction set, [Online]. Available: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html>.
- [48] —, (2019). Turing completeness, [Online]. Available: https://en.wikipedia.org/wiki/Turing_completeness.
- [49] —, (2019). Vhdl, [Online]. Available: <https://en.wikipedia.org/wiki/VHDL>.
- [50] J. W. P. William J. Dally, *Digital Systems Engineering*. Cambridge University Press, 2008.
- [51] Xilinx. (2019). Pynq: Python productivity on zynq, [Online]. Available: <http://www.pynq.io/board>.
- [52] —, (2019). Vivado design suite, [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html>.
- [53] —, (2019). Zynq-7000 soc, [Online]. Available: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>.

- [54] V. C. H. Z. G. V. Š. G. Zaky, *Computer Organization*. McGraw-Hill Professional, 2001, ISBN: 0-07-232086-9.

Appendices


Appendix A.1


GitHub Repository

All of the VHDL modules, Xilinx Vivado project files, diagrams and documents generated during this final year project can be found at this GitHub link:
(<https://github.com/J-CLANCY/RISC-V-FYP>)

Appendix A.2

Standard Operating Procedures

 O'É Gaillimh NUI Galway	College of Engineering and Informatics	
Subject 4BP121, 4BLE121 projects	Title: 4BP121 and 4BLE121 final year projects Masters projects Research projects	Lab Number: Rm 3001-3003, Rm 3007
<u>Objective</u>	Development of embedded systems and web applications, using PC/laptop, network, FPGA/System on Chip (SoC) modules. Low voltage DC power supplies, desktop test equipment such as voltmeters and oscilloscopes.	
Health & safety Documents	Safety Statement for Laboratories Rm 3001, 3002, 3003 Safety Statement for Laboratories Rm 3007 Safety statement for workshop Rm3004 where project work is conducted in Rm 3004.	
New Safety Hazards	None	
Additional PPE (Personal Protective Equipment) Required	None	

Additional Engineering Controls	<p>For example:</p> <p>Special Handling: Lifting of objects related to project. All are small desktop items, e.g. laptop, FPGA/SoC modules, low voltage DC power supplies, desktop test equipment such as voltmeters and oscilloscopes.</p> <p>Storage Requirements: locker-based storage (if required)</p> <p>Accident & Injury Procedures. Refer to University procedures</p>		
Overall Risk Level Low/Med/High	See Risk Assessment Level 1		
Report all accidents to:-	<p>Lab Champion:- <u>Mr Myles Meehan</u> / <u>Mr Martin Burke</u></p> <p>Safety Representative:- <u>Mr Myles Meehan</u> / <u>Mr Martin Burke</u></p>		
SOP (standard operating procedure) prepared by:	Contact Details: Dr Fearghal Morgan	Approved By: 	Date: 8 th Jan 2019

#1	<p>Scope of Work/Activity: State the process/operation/equipment that will be used.</p> <p>Laptop / PC</p> <p>Development of embedded systems and web applications, using PC/laptop, network, FPGA/System on Chip (SoC) modules. Low voltage DC power supplies, desktop test equipment such as voltmeters and oscilloscopes.</p> <p>Working hours (max): 9am – 6pm. No evening or weekend.</p>
#2	<p>Reference Documents: Include any relevant documents</p> <p>N/A</p>
#3	<p>Environmental or Waste Management Impacts and procedures:</p> <p>N/A</p>
#4	<p>Other Equipment: List all items needed to run a process, E.g. T&M equipment, tweezers, vials, drill, lathe, etc.</p> <p>Module assembly using tools in Rm 3004 to follow Rm 3004 safety procedures.</p>
#5	<p>Materials: List all source material, gases, or chemicals used to operate the system.</p> <p>N/A</p>

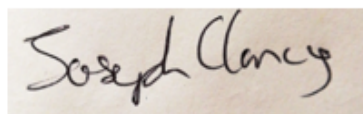
#6	<p>Maintenance:</p> <p>N/A</p>
#7	<p>Procedure: Include or reference instructions on programming and/or operating all equipment</p> <p>1. Safety Briefing</p> <ul style="list-style-type: none"> • Be aware of emergency exits and evacuation notice. • Read the prepared Risk Assessments. • Be aware that the departmental safety statement is available for viewing in the laboratory. • If for some reason any Erasmus intern wishes to perform a function that differs from the procedural methods summarized in the protocol below, they must consult with their supervisor initially and subsequently with members of the technical staff. <p>2. Apparatus</p> <p>Describe in detail the apparatus required. Include block diagrams, photographs and any other aids as required.</p> <p>Laptop / PC, connected to</p> <ul style="list-style-type: none"> - PYNQ-Z2 FPGA - Network: local router or college network (wired Ethernet or wireless) <p>3. Experimental Procedure</p> <p>Connect PC to network using wired Ethernet or wireless</p> <p>Connect PC to PYNQ-Z2 FPGA</p> <p>Develop models for digital systems and upload them to the FPGA</p> <p>Seek formal permission from subject(s) prior to capturing, storing, processing or publishing any personal data.</p>

Prepared By: Dr Fearghal Morgan

Approved by (Research/Project Supervisor(s):

Dr Fearghal Morgan _____

Signed by Student(s):



Joseph Clancy__

Date: 08/01/2019

(To confirm that they have read and understand the contents of this document)

Appendix A.3

Risk Assessment



College of Engineering and Informatics, Risk Assessment.

Locations: <u>List ALL</u> Lab Location(s): Rm 3001, 3002, 3003, 3004, 3007	Project Title: Final Year projects	Duration (Dates): Sept 18 - May 19
Person(s) at risk: (names and positions) Title: Final Year Projects Participants: <ul style="list-style-type: none"> • Joseph Clancy 		

Hazard(s)	Likelihood (1-4) (With Controls in place)	Severity (1-3) (With Controls in place)	Controls (State if normal Laboratory controls are sufficient per Safety Statement.)
Slip, trip or fall	1	1	Cables, personal items and objects eliminated from access routes by all users. Store items on benchtop or external lockers during class session.
Equipment malfunction	1	1	Report defects to laboratory supervisor (all users). Faulty units must not be used until they have been examined and declared usable by a competent person.
1.1 Electrical exposure	1	2	Report defects to laboratory supervisor (all users). Faulty units must not be used until they have been examined and declared usable by a competent person.
1.1 Cutting and stripping wires	1	1	Wear safety glasses. Use only wire cutting and wire stripping tools provided by the discipline. Ensure only cutters with safety clips are used.

Likelihood:

1= Very Unlikely/Yearly

2= Unlikely/During a Semester

Severity 1 2 3

Comments:

- Place an "X" in the matrix for each hazard listed above

Likelihood

XXXXXXXX	X	

Severity:

1= Slight Harm

2= Moderate Harm

Risk Assessment with controls (Taken as the highest colour in matrix)	High <input type="checkbox"/>	Medium <input type="checkbox"/>	Low <input checked="" type="checkbox"/>
---	-------------------------------	---------------------------------	---

Hierarchy of Controls: (Refer to Lab Safety Statement for basic controls)

Details of Administrative/Procedural Controls required: (e.g.)


- Standard Operating Procedures in place for experiments which use equipment other than T&M equipment, PCs/laptops, embedded systems (e.g. FPGA/System on Chip (SoC) modules).
- Relevant Material Safety Data Sheets (MSDS) available. N/A
- Any specific Emergency Procedures are in place and approved.

Any Specific additional PPE required (other than that listed below):

Safety glasses if using wire-stripper. Soldering, cutting, etc should be performed in Rm3004 (workshop), following Rm3004 safety procedures.

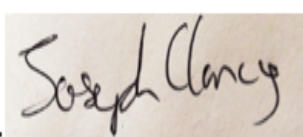
Eye Protection

Prepared By: Dr Fearghal Morgan



Approved by (Research/Project Supervisor(s):

Dr Fearghal Morgan _____



Signed by student (s): Joseph Clancy ____ Date: 08/01/2019

(To confirm that they have read and understand the contents of this document)