IMPLEMENTING A RISC-V PIPELINED PROCESSOR

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Roan James Richmond

Department of Computer Science

Contents

Word Count: 10242

Abstract

IMPLEMENTING A RISC-V PIPELINED PROCESSOR
Roan James Richmond
A report submitted to The University of Manchester
for the degree of Bachelor of Science, 2023

RISC-V is a new Instruction Set Architecture which uses reduced instruction set computer principles. It has been growing in popularity since its introduction in 2015, partly due to it being provided under a royalty-free open-source licence. Pipelining is a technique for increasing the performance of a processor and is often applied to reduced instruction set computers. This project first designs and then implements methods for applying pipelining techniques to a subset of the RISC-V instruction set. After the implementation is shown to work in simulations, the design is then synthesised onto a field programmable gate array to obtain estimates for performance figures, which are then evaluated.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Chapter 1

Introduction

Advances in Integrated Circuits (IC) and related technologies have led to the phenomenon seen today, where it is rare to find an electronic product that doesn't include a processor core. These devices form a network of interconnected hardware, each with its own onboard computational power, commonly referred to as the Internet of Things (IoT) [1]. As the IoT industry expands [2], there is a growing need for the unification of computer architecture as many processors had, and still have, differences in the instructions and registers they use. This inconsistency in the Instruction Set Architectures (ISA) makes it increasingly difficult to develop software that works efficiently across multiple processors, especially within the constraints of IoT devices which are usually limited by size and power. The RISC-V International Association [3] aims to resolve this issue by providing an open-source ISA called RISC-V [4]. The introduction of an open-source, royalty-free ISA allows companies to use this as a common architecture, meaning software that is compatible with hardware from different sources can be more easily written. The rising popularity of RISC-V has shown the success of this, with some processors already being commercially available and more in development [5], as well as popular compilers such as the GNU toolchain[1] now having RISC-V options [6]. RISC-V also has other advantages: as a new instruction set, there are no limitations or requirements for it to provide features like backward compatibility, in contrast to competitors like ARM [7], allowing RISC-V to support a more compact set of instructions. Other projects [8] show how the architecture produces processors applicable to IoT with a small area, low power consumption and high performance.

To increase performance, most modern processors use a pipelined architecture [9],

[1]The GNU toolchain is a set of tools for LINUX systems which allows the compilation of code into programs or libraries

where sequential instructions are split into smaller operations that can be executed in parallel. A pipelined architecture allows for significantly higher clock speeds as the signal has to pass through fewer gates during each clock cycle, resulting in a shorter critical path with a lower propagation delay. If there are significant disparities in how many sub-operations are in each instruction, it can result in unbalanced and therefore wasteful pipelines. Complex Instruction Set Computers (CISC) [10] combined simple instructions, resulting in irregular numbers of sub-operations. The case for the Reduced Instruction Set Computers (RISC) [11] is that they allow regular, single-cycle instructions, which can be more easily pipelined [12]. The introduction of RISC led to higher possible clock rates and is the principle many processors use today, with the notable exception of the Intel x86 [13].

1.1 Aims and Objectives

This project aims to develop a pipelined processor implementing RV32I, which is the base integer instruction set [14] from the RISC-V ISA. The objectives for the project:

- Develop a processor which implements all instructions from RV32I
- Design and Implement a pipeline that will increase the maximum clock speed of the processor
- Verify the implementation works correctly by simulating the design with a test program

This project will be evaluated and verified by simulating the implementation of then using multiple programs written in RISC-V. These test programs should test all instructions that the program claims to have implemented, as well as extreme values, such as maximums and minimums. This will allow for verification and proof that the project completed all the aims. Further evaluation may be beneficial if hardware requirements and clock speed estimates can be produced. The verification strategy:

- Simulate the implementation using a test file that contains at least one occurrence of each instruction defined in RV32I. The values used in these instructions should be ‘expected’ values
- Simulate the implementation with a test file that checks erroneous or unexpected values for each input, these could be maximum or minimum values for registers

Chapter 2

Background

2.1 RISC-V

2.1.1 Modular Approach

RISC-V is organised into modules, each containing sets of instructions designed for a specific task, allowing the designer to decide what functionality is needed. This organisation means processors can be smaller and more specialised to the application, with minimal RISC-V processors being roughly half the size of equivalent ARM units [15]. This structure allows RISC-V to unite the architecture of different types of cores under a single ISA. This architecture allows a system that contains different processors, like a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU), to use the same base ISA, each implementing a different composition of modules.

RISC-V aims to provide broad software support for an extensive array of applications, which is a challenging task. To provide this, it has a guaranteed base integer instruction set, RV32I, which has options for 32-, 64-, or 128-bit implementations and an array of optional extensions shown in Table 2.1. In the base instruction set, the encoding is regular with simple load and store instructions to access memory, resulting in smaller processors. However, as they become more complex with super-scalar operations and out-of-order execution, then RISC-V implementations become closer in size to their x86 and ARM counterparts.

As Table 2.1 shows, RISC-V has three base ISAs and a selection of extensions. A base ISA combined with M-A-D-F extensions, sometimes called the G-extensions,

form a general-purpose ISA able to handle floating point numbers and scalar operations. The extensions use reserved portions of the instruction space with further reserved sections for future development.

Base	Instructions	Description
RV32I	47	32-bit Integer Instructions
RV64I	59	64-bit Integer Instructions along with several 32-bit Integer Instructions
RV128I	71	128-bit Integer Instructions, along with several 32 and 64-bit Integer Instructions
RV32E	47	Subset of RV32I, restricted to 16 registers
Extension	**Instructions**	**Description**
Zifencei	1	Fence provides synchronisation between writes/reads to an instruction memory on the same hardware thread
Zicsr	6	defines a separate address space of control and status registers for each hardware thread
M	8	Integer multiply and divide
A	11	Atomic memory operations for load link, store conditional synchronisation techniques
F	26	32-bit floating point
D	26	64-bit floating point, requires F extension
Q	26	128-bit floating point, requires F and D extensions
C	46	Compressed integer instructions, reduced to 16-bits

Table 2.1: RISC-V base instruction sets and extensions

2.1.2 The Base ISA

The programmer's model for the RV32I is simple: it has 32 registers, referred to as x0 to x31 [14], with x0 being hardwired to zero. There is one additional architectural register visible to the programmer, which is the Program Counter (PC), which holds the current instruction's address. There is no dedicated stack pointer or link register, but the convention is for x1 to be the link register, with x5 as an alternate and x2 as a stack pointer.

As shown in Table 2.2, RISC-V uses a format with consistent regular alignment, meaning function codes and register addresses are placed consistently throughout the different instruction types to simplify the decoder. The designers have placed the opcode and operands in consistent locations, resulting in a compromise where the immediate values have had to be divided across multiple sections of the 32-bit instructions. The reasoning for this is that operand selection and accessing the opcode are usually on

the critical path and are therefore more important than the continuity of the immediate values [16]. All other base ISAs are slight variations on the RV32I base ISA.

The RISC-V address space is byte addressable and uses only simple base and offset addressing, which means a 12-bit signed immediate is added to the base register, rs1, which is then used as the address for loads and stores. This memory addressing method is a more simple memory structure than competitors such as ARM use [17]. Loads use the I-type format, and Stores use the S-type, see Table 2.2, where rs2 is the register containing the data to be written to memory. Another feature of the memory model used in RISC-V is that load values can be sign-extended, which is the default, or not, when not loading an entire word; this means that sign extension has to be done after the data is fetched.

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[31:12]				rd	opcode	U-Type

Table 2.2: RISC-V basic instruction format, source: [15]

2.1.3 Control and Status Registers

The Control and Status Registers (CSRs) are a group of registers that store various information about the processor. There is a defined 12-bit address space for these, meaning a resulting 4096 possible registers, which are unique to each hardware thread (hart). RISC-V doesn't define all of them, allowing possible implementations containing custom CSR usage in the undefined addresses; for example, Tehrani et al. [18] proposed using a selection of the CSRs to speed up cryptography. The primary usage for these registers is trap handling and exception handling and keeping track of the processor's current operating status.

The CSRs are mainly used in privileged mode [19]; however, some features such as counters, timers and floating point status are available in unprivileged modes. The instructions the Zicsr extension, see Table 2.1, contains are I-Type and atomically access a single CSR, where the 12-bit immediate field specifies which register to access with the CSR address space.

Operation	Arguments	Description
CSRRW	rd, rs1, 12-imm	Atomic Read/Write CSR, store CSR value in rd, write rs1 value to CSR
CSRRS	rd, rs1, 12-imm	Atomic Read and Set Bits CSR, store the CSR value in rd and perform bitwise AND on CSR with rs1 value
CSRRC	rd, rs1, 12-imm	Atomic Read and Clear Bits, store CSR value to rd, perform bitwise clear on CSR on the bits specified by rs1 value
CSRRWI, CSRRSI, CSRRCI	rd, 5-imm, 12-imm	Same as CSRR(W/S/C), except using a 5-bit immediate value encoded instead of rs1, which is zero-extended to required length

Table 2.3: RISC-V CSR Instructions

2.1.4 Conditional Execution

The RISC-V foundation decided not to include status flags (sometimes known as Condition Codes, CC) commonly present in an ISA. Status flags are single-bit flags that can be combined into a single status register and set as a result of an operation. An example of a status flag would be the zero flag which can be set when a result is zero. Most ISAs contain status flags that are contained within a status register; for example, ARMv7-A has five flags contained within the Current Processor Status Register (CPSR) [20]. These allow for the conditional execution of instructions based on the result of a previous operation.

Instead of including these, RISC-V uses conditional branch instructions, which include arithmetic comparison operations between two registers [14]. This substitution is motivated by the observation that compare-and-branch instructions can fit into a regular pipeline and reduce dynamic instruction fetch traffic and static code size. There is a benefit lost when multiple branch instructions depend on a single status flag, and therefore RISC-V would have to repeatedly make a comparison. However, the RISC-V foundation considers this to be a relatively rare case.

This decision has caused some backlash among developers [21], with complaints that the workarounds with conditional branches are more effort than the benefits are worth, especially for resource-constrained users who might not see the benefits that a superscalar implementation would provide. What RISC-V has achieved by removing flags is pushing the complexity from the hardware onto the user. This change can be a good thing as checks, like overflow checks, are usually not on the critical path and

instead are mainly used for error handling. Therefore removing them reduces delay on the critical path [22]. It also results in removing implicit dependencies generated by conditional execution that happens with flags and instead leaves only the dependencies between named registers. Removing this complexity within the hardware allows for RISC-V to have smaller cores than competitors, possibly leaving more room for other optimisations [15].

2.1.5 Immediate structure

The structuring of the immediate values within the base ISA initially looks convoluted; however, it has been designed to reduce the number of required multiplexers and always keep the most significant bit (the sign bit) in the same location. For RV32I, the signed bit for immediate values is always held in bit 31, which allows sign extension to happen in parallel with instruction decoding. The overall immediate arrangement implemented allows for reducing hardware multiplexers by around a factor of 2 [14] and reducing signal fan-out.

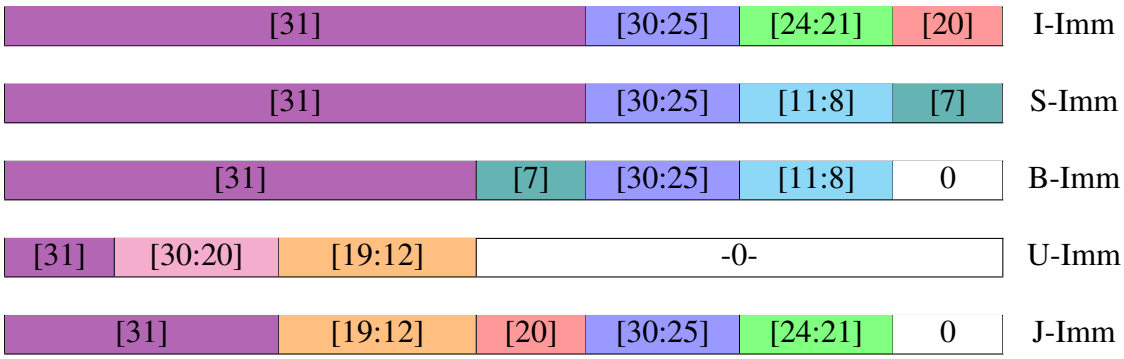

Table 2.4: Types of immediate encoding produced by RISC-V instructions. The fields are labelled with the instruction bits used to construct their value. Sign extension always uses instr[31]. Source: [14]

2.2 Pipelining

2.2.1 Five-Stage Pipeline

The challenge for all pipelines is ensuring the stages are as balanced in propagation delay as possible; this means trying to distribute processing equally throughout the pipeline. A typical design for pipelining in processor micro-architecture is the

five-stage pipeline, shown in Table 2.1, which consists of Fetch (FE), Decode (DC), Execute (EX), Memory (ME) and Write-back (WB). Each stage performs a different sub-task: FE fetches an instruction from memory and then the DC stage decodes this, getting any required operand registers and performing necessary sign extensions on immediate values. The operands are passed to the EX stage, where the Arithmetic Logic Unit (ALU) is; this stage performs calculations and comparisons on the operands. The ME stage then carries out interactions with non-register memory for load and store operations before the WB stage writes back the result to the destination register if required.

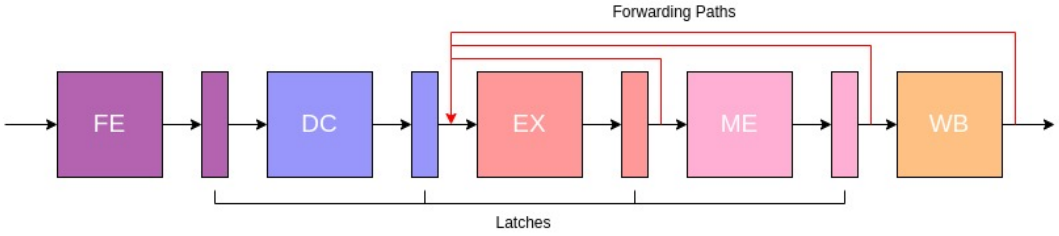

Figure 2.1: Five-stage Pipeline with Forwarding Paths

As shown in Figure 2.1, after each stage in the pipeline the results need to be latched; this enforces synchronisation with the clock. However, these extra registers, which act as latches, are an additional hardware overhead not present in a non-pipelined implementation. The latches store the outputs of the previous stage. Noticeably, there is no need for the results from the WB stage to be latched as these are directly written back to the processor's register bank. Another important feature is that an input to the fetch stage is the PC that points to the address, in memory, of the next instruction to be fetched.

The benefit of a pipelined architecture is that, within a clock cycle, the signal only passes through a single stage, reducing propagation delay and allowing the processor to have a potentially higher clock frequency. The processor should execute close to one instruction per cycle, similar to a non-pipelined processor; the only difference should be that now faster clock speeds are possible. There is a limit to this, where increasing the number of stages starts to reduce performance [23]. An optimum number of stages depends on the ISA being used and hardware constraints, like power and space.

2.2.2 Dependencies and Hazards

There are three types of dependencies, structural, control and data [24]. Structural dependencies occur because of a resource conflict within the pipeline, where the same hardware is accessed from multiple different places in the pipeline in the same clock cycle. In the initial pipeline, shown in Figure 2.1, there is a structural hazard as both the FE and the MEM stages could try to operate on the same location in memory simultaneously. Structural hazards like this can lead to unpredictable results if both stages try to access the same address within memory. A solution for this is to split the memory into Instruction Memory (IMEM) and Data Memory (DMEM). A possible drawback is that it stops the processor from modifying instructions dynamically at execution time; however, this functionality is rarely used.

Control dependencies, sometimes called branch hazards, occur because of branch instructions. When a branch instruction is fetched, the processor does not know it is a branch until after the DC stage, and then it cannot tell whether the branch is to be taken until the EX stage. However, the fetch unit needs to fetch an instruction every cycle, resulting in sometimes the wrong instructions being fetched. Depending on the implementation and assuming a five-stage pipeline, this can result in up to three incorrect instructions being fetched and subsequently executed.

Three types of data hazards can cause issues within a pipelined architecture [25]: Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). However, the architecture described by Figure 2.1, means that only RAW data hazards can occur. This is due to the architecture only updating the registers in the last stage of the pipeline. RAW hazards, sometimes called ‘true’ data dependencies, occur when an instruction relies upon the result of a preceding instruction. The second instruction depends upon the first instruction’s result and therefore requires this to be available when it goes through the DC stage, and the operands are fetched from the register bank. If it isn’t available, the instruction could use incorrect data, and therefore its result will be invalid. This dependency is explicit in Table 2.5, where the first instruction updates x1, and all the following instructions rely upon the value of x1. From Table 2.5, it can be seen that the following two instructions could use an incorrect value for x1 and therefore produce invalid results.

	Clock Cycle							
Instruction	1	2	3	4	5	6	7	8
ADD x1, x2, x3	FE	DC	EX	ME	WB			
AND x4, x1, x2		FE	DC	EX	ME	WB		
SUB x5, x1, x2			FE	DC	EX	ME	WB	
Value in x1 updated from ADD instruction								
XOR x6, x1, x2				FE	DC	EX	ME	WB
OR x7, x1, x2					FE	DC	EX	ME

Table 2.5: Data Dependencies in five-stage pipeline

2.2.3 Stalling and Flushing

Stalling is when the processor delays the output from stages in the pipeline, consequently delaying the output from the preceding stages to stop instructions from being overwritten and lost. Stalling introduces 'bubbles' into the pipeline; these are essentially empty stages, sometimes referred to as "No Operations (NOPs)". Stalling is a method for dealing with data dependencies, as it allows the register read to stall until the data required by the next instruction is available. This can be seen in Table 2.6, as the BEQ instruction stalls while waiting for the previous ADD to enter the WB so that the value in x1 is updated. This stall introduces a bubble, as seen in clock cycles 3 and 4, but means that the result of the BEQ operation is valid.

	Clock Cycle										
Instruction	1	2	3	4	5	6	7	8	9	10	11
ADD x1, x2, x3	FE	DC	EX	ME	WB						
BEQ x1, x2, 0x4		FE	-	-	DC	EX	ME	WB			
ADD x3, x1, x2					FE	DC	EX	-			
SUB x4, x1, x2						FE	DC	-			
XOR x3, x1, x2							FE	-			
AND x3, x1, x2								FE	DC	EX	ME

Table 2.6: Stalling and Flushing in five-stage pipeline

Flushing the pipeline means removing some instructions that have become invalid from the pipeline before they finish the WB stage. This method can be used to counteract control hazards introduced by branch instructions. If a branch is taken, the three instructions behind it in the pipeline are invalid as they were fetched speculatively, so they should be flushed from the pipeline. This can be seen in Table 2.6 where, on clock cycle 8, the invalid instructions are flushed from the pipeline to stop them from completing and writing the results back to the register bank.

Both of these methods protect against different hazards which pipelining introduces; however, they impose performance repercussions as now there is a possibility that the pipeline will be stalled or flushed, increasing Clocks Per Instruction (CPI) [26].

2.2.4 Operand Forwarding

Operand forwarding, as shown in Figure 2.1, is an optimisation method that can reduce the need for pipeline stalls and therefore decrease CPI [27]. It is where operands from later stages are made available as inputs to the execute stage. Except in specific conditions, it removes the need to stall because of a RAW data hazard. It does add extra hardware, space, and control complexity [28], but these drawbacks are outweighed by the possible speedup it provides. There is still a specific scenario where stalling is still necessary: this occurs on a load operation, where the result of an operand isn't available until after the memory stage; therefore, the pipeline would still have to stall for a single clock cycle at least, while the memory operation is performed.

2.3 Field Programmable Gate Arrays

This project is intended for synthesis onto Field Programmable Gate Arrays (FPGA), which are semiconductor devices constructed from a matrix of Programmable Logic Blocks (CLBs). These can be reprogrammed after manufacturing to fit the required application functionality. This project will be implemented onto an FPGA because it is cheaper and simpler to experiment and test functionality and performance. This is because FPGAs can be reconfigured numerous times, making developing them cheaper [29]. They also allow the use of techniques that wouldn't be available on an Application Specific Integrated Circuit (ASIC); For example, assigning values to registers at the start of execution.

Chapter 3

Design

3.1 Data Pipeline

In this Section, the design of the pipeline will be explored stage by stage, with possible future sources of error identified and specific design decisions detailed.

3.1.1 Fetch Stage

The FE stage consists of deciding on the correct PC value to use when fetching the next instruction and then performing the instruction fetch. The fetch stage works autonomously, loading the next instruction, which is most often the instruction at the 'PC + 4' address. If the pipeline is stalled, then this instruction isn't latched and therefore, on the next clock cycle, the current PC value is used, re-fetching the same instruction. This operation is occasionally overwritten by arriving branch instructions, where the address of the next instruction to be fetched is instead the value forwarded back from the EX stage. These three choices for the next PC value are shown in Figure 3.1, along with the 'condition' input, which is the output from the COMP unit in the EX stage indicating if a branch instruction should be taken.

At the end of the FE stage, an Instruction Register (IR) holds the instruction fetched from memory, and the PC contains the associated address. As detailed in Section 2.2.2, to avoid structural hazards, the memory is divided into Instruction Memory (IMEM) and Data Memory (DMEM), and the fetch stage will be the only stage with access to the IMEM. Since RISC-V has byte-addressable memory and the ISA in use has 32-bit (4-bytes) instructions, the PC will be incremented by four each cycle. This implementation doesn't need to account for varying instruction sizes, which are present

in RV64I, as all instructions are 32-bits long.

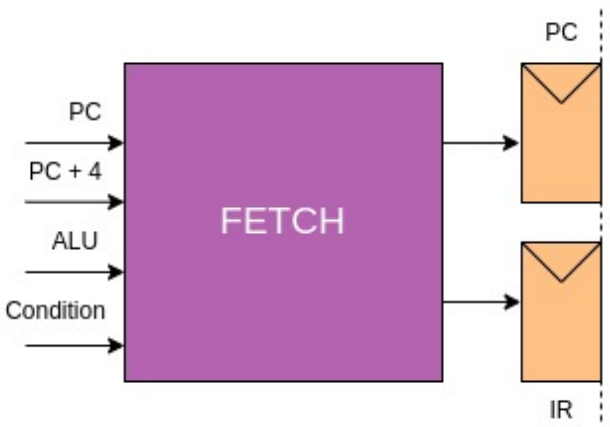

Figure 3.1: Block diagram showing Fetch stage

3.1.2 Decode Stage

The DE stage is responsible for accessing the register bank, returning the correct operands, and extending and concatenating immediate values. The regular structure of instructions means that register addresses are always aligned; therefore, the implementation of this stage can be simplified [16]. This allows for fetching the operands while decoding the instruction, which can be done in parallel, reducing the length of the critical path. Doing these operations in parallel means that sometimes registers that aren't needed are fetched, particularly the operand resulting from the rs2 address, the least used operand in RV32I. These unnecessary fetches waste resources such as power but allow the possible speedup provided by parallelism. Since the project aims to produce a high-performance pipelined processor, in this situation, the slight increase in power waste is less significant than the possible increase in performance. Therefore register operands rs1 and rs2, shown in Table 2.2, will always be latched, pushing the complexity of deciding if they are needed onto the EX stage.

The processor should carry out sign extension and decoding in parallel when doing immediate concatenation and extension. All immediate values in RISC-V are sign extended as standard; the only difference between unsigned specific instructions is that the COMP unit considers the operands and immediates as unsigned when doing comparison operations in the EX stage. There are five types of immediate encoding,

as shown in Table 2.4, and if the instruction doesn't contain an immediate value, then it won't matter what value is latched as it will be ignored in the EX stage.

The WB stage should pass results back to the DE stage so they can be saved into the register bank if needed. This adds a complication for register operands as there is a possibility that the WB stage is returning an updated version of the register that is being fetched. This means that after fetching the register operands, a multiplexer should compare the address for this and the address of the data from the WB stage. This technique adds more hardware, but the alternative is stalling for an extra cycle to allow the register write to take place.

3.1.3 Execute Stage

The EX stage of the pipeline contains the ALU and is responsible for carrying out any comparisons or operations on the operands. As shown in Figure 3.2, there are five inputs from the preceding DE stage and a further three inputs, which are forwarded values from EX, ME, and WB stages. This results in a possibly long propagation delay with multiplexers selecting the correct value; however, it removes the need to stall on data dependencies as detailed in Section 2.2.4.

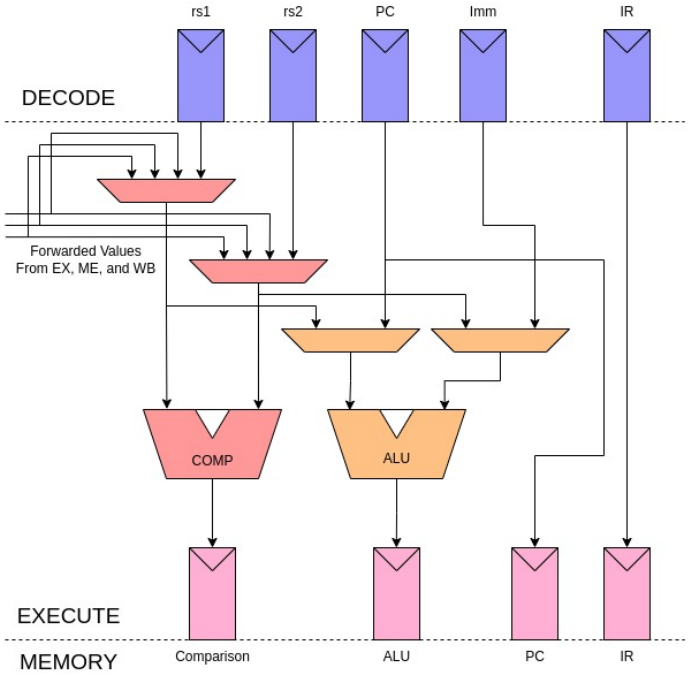

Figure 3.2: Register Transfer Level (RTL) diagram showing Execute stage

For the EX stage to handle conditional branches, two operations need to be carried out: the condition evaluation and the offset address calculation. Both of these operations need to be carried out in the execute stage, as this is the earliest stage where the operands are available, and branches need to be evaluated as early as possible. As shown in Figure 3.2, an extra piece of hardware, the comparison (COMP) unit, has been added, allowing both operations to be done in parallel. The cost of this is extra hardware and possible wasted energy of calculating the offset address when the branch isn't taken. This method is preferred to other options, such as evaluating the condition and then stalling if the branch is to be taken to calculate the address.

3.1.4 Memory and Writeback Stages

The function of the ME stage is to provide access to the DMEM through load and store operations. In RISC-V, offset addressing is used, which means that the addresses for load and store operations have to be calculated; this is done in the EX stage, and the output of the ALU becomes the address for the ME stage. Figure 3.3 shows how the DMEM is accessed within the ME stage; this structure means that within the ME stage, there can't be any processing on the data fetched in a load instruction before it is latched. This restriction imposes consequences on the pipeline, as it means that sign extension of data fetched can't be done in the ME stage itself. Therefore one possible solution, shown by Figure 3.3, is to move the complexity of the sign extension of these values into the WB stage. The consequence of doing this is imposing a stall, as now the value from a load operation won't be available for forwarding until the WB stage rather than during the ME stage.

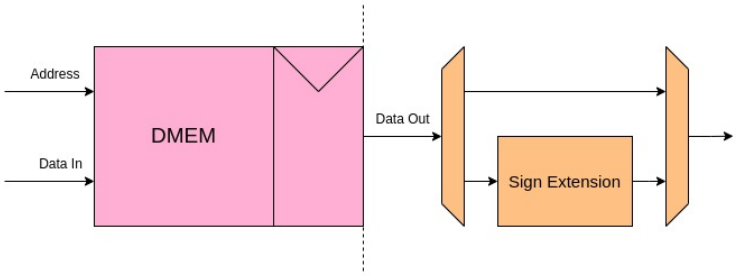

Figure 3.3: Register Transfer Level (RTL) diagram showing subsection of ME and WB stage

3.1.5 The CSRs

The implementation in this report does not include the CSRs; however, the placement of these is still considered to allow for possible expansion. The CSRs aren't part of the register bank or accessed through load and store operations like DMEM; instead, they are accessed through a series of complex swap instructions, see Table 2.3, which means placing them within the pipeline is more complex than placing other components. It is possible to treat them as a separate register bank and access them through the DE stage of the pipeline, which would be a simple solution to implement. However, this would cause stalls when multiple CSR instructions occurred in series and referred to the same register within the CSR bank, as access to each CSR register is atomic. Another option is to use them as inputs and outputs to the Execute stage; instead of latching the results, the results could be written directly back to the CSRs at the end of the stage. This implementation would remove the aforementioned single-cycle penalty but has the possibility to increase the length of the critical path and slow the execute stage and possibly the processor clock rate.

3.1.6 Final Netpath

The different sections of the pipeline, detailed above, need to be connected in an overall netpath, where consecutive stages and forwarding paths can be joined, as shown in Figure 3.4. In order to simplify the implementation of the overall netpath, each stage should be self-contained and consist of a series of input wires and output registers assigned at the end of each clock cycle. Therefore, each pipeline stage forms its own Mealy machine [30].

This implementation can be further simplified by standardising the interfaces between the stages, which incurs extra hardware costs. Excluding the WB stage at the end of each clock cycle, the IR will contain the 32-bit instruction relating to the outputs from that stage. This implementation isn't the most hardware-efficient design, as certain parts aren't needed after specific stages. For example, after the EX stage, there is no need to pass the section which contains the rs1 and rs2 addresses and any immediate values. However, to standardise the interfaces, the full IR will be passed through the entire pipeline, as shown in Figure 3.4.

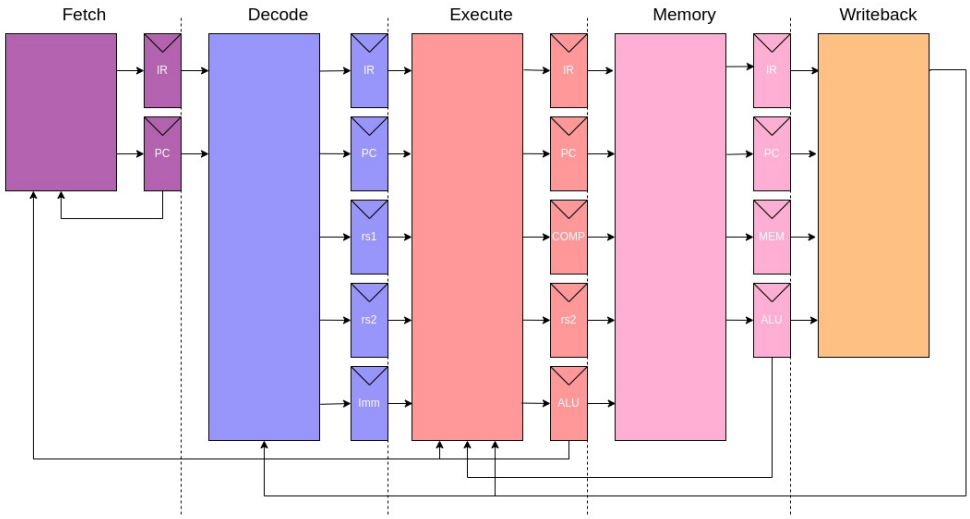

Figure 3.4: Block diagram showing pipeline

3.2 Control Signals

3.2.1 Forwarding

The three forwarding paths to implement are from the output of the EX, ME and WB stages to the input of the EX stage, as shown in Figure 3.4. Therefore for the EX stage to decide if these values are relevant, it also needs the addresses that the values relate to. This means there is an extra hardware cost of three 5-bit registers to hold the relevant register addresses, which are needed instead of just splitting the rd from the IR in case of S-Type instructions, see Table 2.2. These address registers should always contain a value, creating an issue when there isn't a value to be forwarded. In the simulation, one solution would be setting the address registers to undefined values (X-values), indicating that there isn't anything contained in the forwarded registers. However, this wouldn't work in a hardware implementation. Another solution is setting these registers to 0-values and adding a check for this in the EX stage.

Also considered in the design for forwarding is the priority of forwarded values. If all the forwarded addresses match an operand, then there needs to be an order of priority to decide which value is used. The most recent value should be prioritised, meaning the order is: EX, ME, WB, and then the value from the DE stage. Assuming it is valid, the value forwarded from the output of the EX will always be the most recent version of the register value.

3.2.2 Stalling

Stalling should happen purposely in a small number of scenarios and accidentally as a consequence of failures, such as cache misses on memory. The only purposeful stall for this design is when a load is followed by an operation that uses the load destination register as an input operand. In order to resolve this situation, the EX stage should stall. Therefore the EX or ME needs to be able to detect these dependencies. One method for this detection would be adding a register to the EX containing addresses that still need to be forwarded back to the EX. Consequently, there needs to be a 5-bit register in the EX containing the target return register of load instructions that have passed. This register can be updated when load instructions pass through the EX stage and can be reset to zero when the result from these load operations is forwarded back from the WB.

The pipeline will also stall on cache misses, which won't happen in the simulated version but would in hardware. Therefore there needs to be a method for verifying the pipeline design stalls correctly in these scenarios in simulation. In order to verify this, external stall signals should be included in each stage, primarily the ME stage. These can then be set at random intervals to simulate the unpredictable stalls.

The pipeline also needs some control mechanisms for signalling a stall within a particular stage. One method of doing this is with a 'ready' signal, as shown by Figure 3.5, passed from the stage in front to the one behind. The signal is a promise from the stage in front, indicating that it is ready to accept data on the next clock edge. When the stage behind receives this, it can promise the stage preceding it the same, meaning the signal propagates down the pipeline. This method means that on a stall, pipeline stages in front of the stalling stage will continue to operate normally while the trailing pipeline sections will stop.

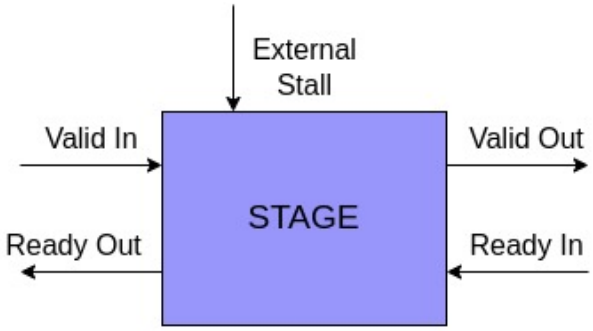

Figure 3.5: Figure showing control inputs and outputs from a pipeline stage

3.2.3 Flushing

Flushing happens after a branch is taken when the FE and DE stages need to be cleared of invalid data. One method for this is setting the output registers for these stages to NOPs or zero values. However, a better method is to add a 'valid' output signal indicating whether the output from each pipeline stage is valid or invalid, as shown in Figure 3.5. This approach would be similar to the 'ready' signal but instead, flowing in the opposite direction through the pipeline, allowing the 'bubble' created by branching to pass, as shown in Figure 3.6. Then if invalid data reaches the WB stage, it can be discarded at no extra cost by not writing to the register bank.

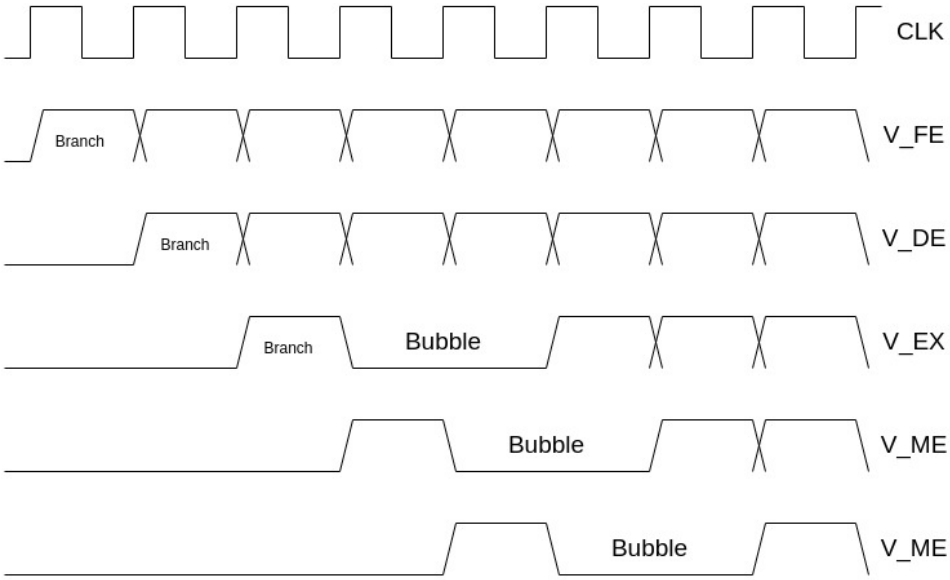

Figure 3.6: Showing 'bubble' in 'valid' signals after a branch is taken

Chapter 4

Implementation

The Implementation is done using Verilog [31], a Hardware Description Language (HDL) used to design and verify digital circuits at the RTL level. A subset of the statements provided in Verilog are synthesisable [32], and modules that contain only these can be implemented into hardware by synthesis software.

The specific FPGA device that the design is to be synthesised for is a Xilinx Spartan-3 'XC3S200' [33]. This device has 480 CLBs, twelve 18-Kbit block RAMs, and can support up to an estimated 200,000 logic gates. Block RAMs are synchronous memories where data output has to be latched, as shown by the interface in Figure 3.3. The implementation that is produced should be device independent, but some code structure choices may allow for speed, size, or power optimisations. Therefore, attention to how the synthesis software interprets the implementation on this specific device is significant throughout development.

4.1 Code Structure

4.1.1 Modules and Functions

The implementation is organised into a hierarchy of modules and functions to make structural features more explicit and consequently easier for a future project to develop; for example, one of the benefits of containing the complete ALU in a function is that it protects the functionality of this unit, allowing access only through inputs and the return value. Each pipeline stage forms its own module and subsections of these stages, such as the ALU and COMP unit in the EX stage, are contained in Verilog functions.

This structure allows individual stages to be modified separately without affecting

the others; for example, the EX stage could be replaced with a new version that implemented the M-Expansion shown in Table 2.1, allowing multiplication and division to be added without changing the rest of the pipeline.

4.1.2 Variables and Assignment

Mnemonics are available in Verilog and can be used for defining constants. They are used within the module to make the code more readable and are defined for the possible values of 'funct3' and the 'opcode', see Table 2.2. There isn't a need to define a mnemonic for all values for the 7-bits, as only some are relevant instructions with the rest being 'NOPs'. The value of 'funct3' has a different meaning depending on the instruction, and therefore multiple mnemonics are defined for each value. For example, BNE and SLL instructions have the same funct3 value. In a BNE instruction, this value indicates which comparison should occur in the COMP unit; in contrast, in an SLL instruction this relates to the ALU operation.

Verilog allow has two types of assignments: blocking and non-blocking. The use of non-blocking assignments allows for synchronisation and parallelism. Within a section of code using non-blocking, all the results of the expressions (the right-hand side) are calculated and then simultaneously assigned to the left-hand sides. The benefit of non-blocking methods is allowing the definition of a state machine [34] without using temporary variables. Blocking assignments allow a standard flow of instructions where lines of code are executed serially. However, this will not prevent the execution of separate blocks using blocking assignments to run in parallel, unless enforced by the programmer.

The two types of assignments are used in different parts of the implementation. Non-blocking assignments were used to assign values to the control signals and registers at the end of each clock cycle. Blocking assignment was used to pass signals through parts within an individual pipeline stage. In the EX stage, blocking was used in the multiplexers and ALU and non-blocking was used to capture these values and latch them in registers at the end of the clock cycle.

In the code, these assignments are structured into different types of blocks; non-blocking is used in combinatorial 'always' blocks, and blocking is used in sequential functions. Functions contain blocking assignments to and from wires based on the control signals, allowing temporary variables to change, but unless these are latched in a register, they won't retain these values. Clock-dependent 'always' blocks then control access to the registers, as shown by Figure 4.1. These registers are implemented

to latch the outputs, 'saving the values' from functions, allowing them to be passed to the next stage in the pipeline.

```
1
2     always @ (posedge clk) begin
3         if (r_out & v_in) I_out <= calculate_i(IR);
4         if (r_out & v_in) PC_out <= PC;
5         if (r_out & v_in) IR_out <= IR;
6     end
7
```

Figure 4.1: Figure showing output from DE stage being synchronously latched

4.1.3 Byte Addressable Memory

As mentioned in Section 3.1.1, the memory for both IMEM and DMEM is byte-addressable. This requirement has little impact on the IMEM as accesses to this memory are always fetches of four bytes on word-aligned contiguous locations. However, for the DMEM, each byte can be fetched and written individually through SB, LB, and LBU. The DMEM also has to allow up to four bytes to be read from or written by the same instruction with the only restriction on these bytes being that they are naturally aligned, as shown by Table 4.1. Therefore, for the pipeline to operate without needing to stall on a ME operation, this access must occur in a single clock cycle in the ME stage of the pipeline.

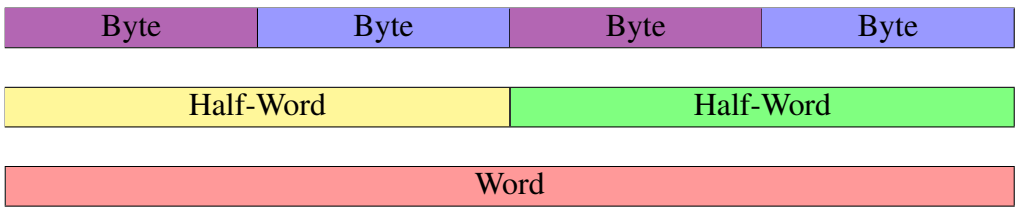

Table 4.1: Showing how bytes within memory can be accessed only on naturally aligning addresses [15]

When trying to code the DMEM as a single block of word-long locations in RAM, shown Figure 4.2 option one, the DMEM is represented by the synthesis software using a large block of 32-bit registers rather than the intended single RAM block. The required behaviour to make this implementation work can't be synthesised into an individual block of RAM. The RAM in use doesn't allow individual bytes within a word

to be written to without affecting the rest of the word. Therefore, the synthesis software represents it as a large block of word-long registers connected with a multiplexer in order to maintain the structure described and provide the required functionality.

The next option explored was to represent the DMEM as a single block of byte-long locations, shown by Figure 4.2 option two, which resulted in the synthesis software instead representing it as a block of byte-long registers connected with a multiplexer. This substitution by the software is because the RAM provided is dual port access. However, on LW and SW instructions, four bytes are accessed together, which can't be done in the same clock cycle meaning the DMEM can't be structured like this and represented using RAM without stalling on word-long accesses.

The solution implemented was to split the DMEM into four individual blocks of RAM, each containing byte-long memory locations, see Figure 4.2 option three. The bottom two bits of the memory address are used to determine which RAM block to start with. This method can be simulated by the software as a block of RAM instead of registers because, on LW and SW instructions, only one byte from each RAM block would be accessed, adhering to the hardware restrictions.

```
1
2     //Where i is the number of bytes in DMEM required in
     implementation
3
4     //Option 1:
5     reg [31:0] RAM_dmem [0:(0.25i)];
6
7     //Option 2:
8     reg [7:0] RAM_dmem [0:4i];
9
10    //Option 3:
11    reg [7:0] RAM_dmem_0 [0:i];
12    reg [7:0] RAM_dmem_1 [0:i];
13    reg [7:0] RAM_dmem_2 [0:i];
14    reg [7:0] RAM_dmem_3 [0:i];
15
```

Figure 4.2: Figure showing different options for DMEM implementation

4.1.4 Sign Extension

In Section 2.1.5, the reasoning for the immediate operand structure is detailed: to reduce the number of multiplexers and allow sign extension to happen in parallel with other sections of the decode stage. There are multiple options for implementing this

sign extension in Verilog. A possible method could be to declare it explicitly, detailing how the concatenation should occur for each of the different immediate structures, as shown by Table 2.4. However, this method isn't very readable and may impede future development, such as adding on the F-Extension, detailed in Table 2.1. A comprise was implemented, where the order for concatenation was explicitly defined; however, sign extension itself was left to an inbuilt function in Verilog. This allows the readability of the code to be maintained and pushes the complexity of optimising the operation onto the synthesis software instead.

4.2 Pipeline Implementation

4.2.1 Execute Stage

The EX stage is the most complex stage, with multiple calculations happening in parallel, as shown in Figure 3.2. The structuring of this stage is crucial to simplify the development and make it more readable. Figure 3.2 shows there need to be four multiplexers and two computational blocks (COMP and ALU) and, as described in Section 4.1.1, these will each be in a separate function using blocking assignments. In the implementation, the number of functions was the same as the design; however, the ALUs multiplexers' reliance on the COMP multiplexers was removed, fully separating the COMP and ALU functionality. There were two main reasons for this: it simplified the code and reduced the critical path length. The simplification of the code is the separation of the functionality, allowing the inputs to the ALU and COMP unit each to equal the output of a multiplexer function. The critical path is also shorter as now the ALU inputs aren't reliant on other functions, this change does add hardware overhead, as the ALU multiplexers are now five input multiplexers instead of two, but this is a negligible increase.

The ALU and COMP units themselves are simple, with case statements to decide on the operation based on 'funct3' see Table 2.2, and mnemonics defining the options for the binary. RISC-V uses signed operations as default, with unsigned operations being explicitly signalled. This design is advantageous for compilation from some low languages such as C [35], which use signed operations as default. However, as Verilog uses unsigned operations as default this made implementing this more complex. Two options for implementation in Verilog would allow for signed operations, explicitly defined signed wires, or the unsigned wires could be temporarily cast to signed for the

duration of an operation. The COMP unit and the ALU have been constructed using different techniques for dealing with signed operations; this was done for comparison purposes to see if it had an effect when the module was synthesised.

4.2.2 Control Signals

The pipeline control signals are 'Valid' and 'Ready', indicating if a pipeline stage is ready to accept data and whether the data out is valid. These allow for the stalling of the pipeline and the flushing of invalid data. When forwarding, the 'Valid' signal should also be passed; otherwise, data from invalid instructions could be used by the EX stage. As well as data forwarded to the EX stage, data forwarding from the EX to the FE stage should also be accompanied by 'valid' signals in case two consecutive branch instructions follow each other. As what could happen is that the first branch is taken, and then the second branch is then also taken because it wasn't accompanied by an unset 'Valid' signal to indicate it was junk.

Flushing has multiple possible implementations: the first method explored was an additional register in the EX stage to store the target of a branch instruction. Therefore on a branch instruction being taken, the target address would be stored in a separate register in the EX until an instruction with this address reached the EX stage. While this address is stored, all output from this stage would be marked as invalid. This implementation adds a hardware overhead of a 32-bit register used to store this address, and an extra comparison operation would need to be taken on every instruction after a branch.

Since the pipeline is a known length, the implementation could be simplified by marking the following two instructions through the EX stage as invalid. This could be done by adding a flush signal, which can be passed back to the DE stage and set the valid signal, as shown by Figure 4.3. This method would remove the need for adding the additional register and the complexity of the comparison on the incoming PC value.

4.2.3 Final Netpath Implementation

The different stages of the pipeline need to be joined together into an overall module, which contains the wires to pass signals between the individual modules. This netpath should also allow a test file to insert stalls in the pipeline to imitate failures such as cache misses, this is done through inputs to the netpath which are then passed as stall

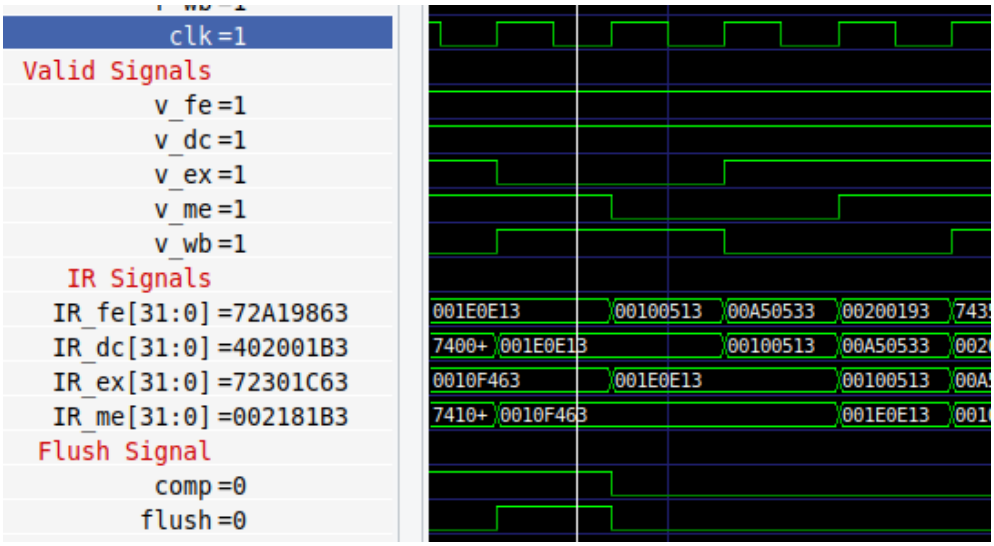

Figure 4.3: Waveform output showing 'bubble' in the pipeline from flushing after a branch instruction

signals to the pipeline stages. This file will be the final module or netpath for the implementation. As detailed in Section 3.1.6, this module solely connects the stages, therefore allowing all of the processing of signals to happen within the modules for the pipeline stages.

The challenge for this module was enforcing a naming convention, as most of the wires have the same or similar names. Therefore the naming was done using the format 'XXX_YY' where XXX is the name of the wire or register, and YY is the notation used for the stage the signal originates from. The only exception to this format is the clock, a signal passed to all stages and referred to as 'clk'. The netpath implemented allows for stall signals as inputs, one for each section of the pipeline, to provide the functionality to simulate stalls with an external stimulus file.

4.3 Unit testing and Debugging

The testing strategy for the project entailed unit tests, where each stage of the pipeline was tested individually and then integration testing, where the entire pipelined processor was tested on a set of test programs written in RISC-V assembly. The unit testing was mainly black box testing, focusing on the values stored in the registers at the end of each clock cycle; however, some additional white box testing occurred using the waveform viewer to spot redundancy.

4.3.1 Unit Test Structure

The stimulus files used to test each pipeline stage are different but conform to the same regular structure to make errors more noticeable. For each pipeline stage, all the defined behaviours are tested multiple times with random values. Some random values are tested with undefined behaviours to ensure that each pipeline stage treats these as 'NOPs'. For example, in the EX stage, all the instructions apart from load and store instructions are tested individually as they each perform a different operation and are tested with random values. In the stimulus file, this is structured using individual functions, which are then called by an 'initial' block. There is an enforced wait between each function call, allowing a pause in processing to be visible in the waveform to help debug the program. The values stored in the registers are read at the end of the clock cycle, and the inputs are changed on the negative edge of the clock, allowing them to be latched at the start of the next cycle.

The omission from the stimulus files is the ability to test the control signals. This omission from the unit tests is deliberate because, during development, the control signals were tested altogether but separately from the rest of the pipeline. The independent testing happened because all the pipeline stages, excluding the EX stage, have similar functionalities for the control signals. Therefore during development, an empty pipeline was created to test the control signals and simulate stalling before adding these to actual pipeline stages.

4.3.2 Waveform and Output files

Debugging for this project mainly occurred by simulating each stage, testing running a stimulus file, using the waveform viewer to detect some functional errors and redundancies and then using an output file to detect other functional errors. The advantage of using the waveform viewer for error detection is that it shows signal changes and therefore shows when output is available during each clock cycle. This was used in developing more specific targeted tests as the waveform viewer highlighted wires and signals that remained unchanged, therefore highlighting these as either untested signals or unnecessary additional hardware. When simulating the control signals in the 'fake pipeline', the waveform viewer was mainly used, as it could visually represent different aspects, such as bubbles moving down the pipeline and stalls stopping processing. The waveform viewer software used was GTKWave [36].

In conjunction with viewing the waveform output using GTKWave, Verilog also

allows writing to files from a stimulus file. By doing this periodically, register values can be stored, which was used to check that the processor was carrying out the correct operations. By printing these values in different forms, it demonstrated if the behaviour was correct. For example, errors in the SRL instruction are shown best when the output and input values are represented as binary. Between these two different debugging methods, most units were tested to a good standard. However, even for unit tests, this can't be an exhaustive process; therefore, this doesn't remove the chance of errors, it just reduces the likelihood.

Chapter 5

Evaluation and Verification

5.1 Verification

For the verification stage, the tests were written in RISC-V assembly rather than the Verilog stimulus files seen in Section 4.3. These were then assembled Verilog memory files, '.mem' files, containing machine code. These files were then loaded as the IMEM in an 'initial' block with a start index of zero.

5.1.1 Test Structure

After the netpath for the pipeline and forwarding paths had been implemented, integration testing was used to check the functionality. For the integration tests, the philosophy was to write test programs that initially validated all instructions and then checked on the structure and control mechanisms like stalling and forwarding.

Integration tests were also built for regression testing, which is a method of testing used to ensure previously tested software still functions as expected after changes have been made. The use of regression testing confirmed that the pipeline still worked as expected while the program was tuned to allow synthesis software to adapt the design for an FPGA. To enable the integration tests to be used in regression testing, they had to be structured as a single file that tested the limits of the pipeline, allowing for repeatability and making errors obvious within the waveform.

The ordering of tests was significant in the design of the test file, with branch instructions being tested first; this allowed the processor to build confidence in branches so that when an error was detected later, the program could branch to show this. Therefore, errors were easier to debug as they could be indicated by sudden changes in the

PC value. Figure 5.1 shows an error being detected and then a branch operation happening, this causes the PC of the FE stage to jump to 3000 indicating an error to the programmer. As branch instructions are used to detect errors, and branches being taken cause the 'valid' signal at the EX stage to drop, as shown by Figure 5.1, to easily tell where errors occur 'v_ex' can be watched. Initially, this was adequate; however, as more instructions were being tested and the test programs became increasingly complex, it became harder to tell precisely where errors were occurring. To increase the clarity of the waveform and help make the waveform more comprehensible, the branch instruction targets were spaced out; this made debugging easier as it meant that the exact lines where an error occurred could be more reliably identified.

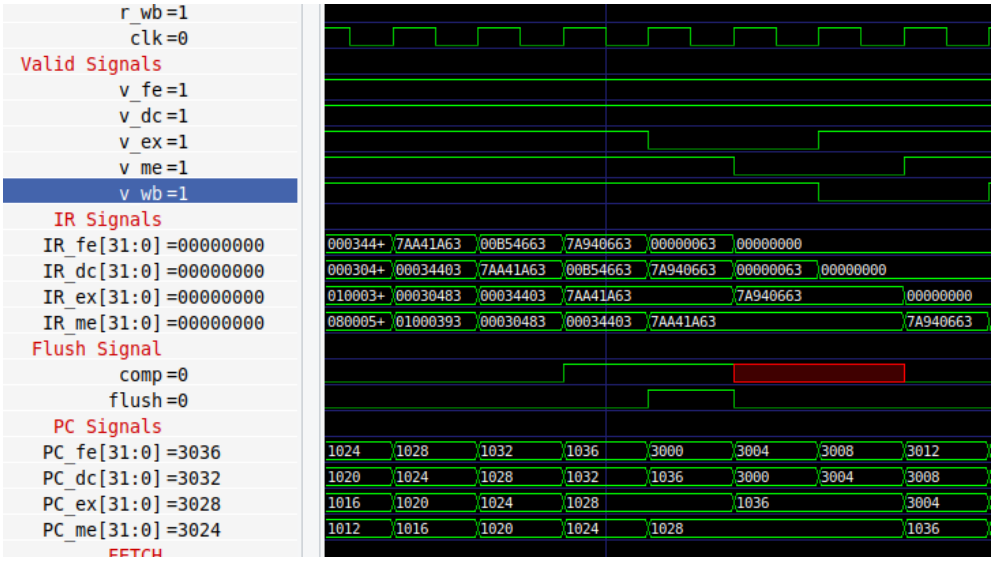

Figure 5.1: An Error in the Pipeline indicated by a branch operation

The tests were organised into functions, each designed to stress specific areas; for example, there was a separate function for testing shift instructions. Within these functions, the tests mainly consisted of using the pseudo-instruction[1] Load Immediate (LI) to start with an immediate and then carry out various operations. After these operations, branch instructions could be used to test if the results of these operations were correct. The branch instructions used to test each of these operations would branch to a different location to demonstrate in the waveform where an error occurred. As well as spacing out the error branch targets, the functions were also spread out, allowing for a series of NOPs between each function. This structure made it easier to indicate the

[1] Pseudo-Instructions are instructions for the processor expressed in a non-canonical way; these can indicate sequences of instructions, for example LI

processor's testing phase and simplified the waveform.

Some of the more complex tests to implement were memory tests, which required both load and store operations to be tested in tandem. Due to the implementation, the DMEM started initially full of zero values and then store operations could store register values. Then, using Load instructions to retrieve these values, both memory operations could be tested together. This structure also allowed for testing some stalling mechanisms, as results from Load instructions aren't available until the WB stage.

5.1.2 Discovered Errors

The first error encountered was in shift operations; this is because they don't conform to the instruction formats described in Table 2.2, and although they do involve an immediate called 'shift amount' (shamt), the structuring of this value doesn't conform to any of the encoding described in Table 2.4. Table 5.1 shows the actual shift instruction format, which is a variation of the R-type instruction where the 'rs2' value has been replaced with the 'shamt' value. This 'shamt' value doesn't need to be a large value as there is no need to shift more than 32 places, as after this, the values either become zero values or the maximum negative value of -2,147,483,648, depending on the shift type and the sign bit. The reason this wasn't initially detected in the unit testing phase was that this error sits between two stages, the DE and EX stages. The EX unit tests passed the shift instructions because they were already given the correct shift distance to run tests on. The DE unit tests passed because shift instructions were interpreted as R-type instructions rather than a separate encoding. Another reason this error wasn't spotted during the Unit tests was that the value for 'funct7' was a zero value for logical shifts, meaning if this value were to be included in the immediate extension, it wouldn't affect the value latched at the end of the DE stage. The simple alteration needed to fix this was adding extra encoding to the sign extension unit to detect and cope with these types of instructions.

funct7	shamt[4:0]	rs1	funct3	rd	opcode	Shift Instr

Table 5.1: Shift instruction format, source: [14]

One error that is present in the design takes place due to the flushing mechanism used to mark instructions as invalid after a branch is taken. The implementation for flushing is that the processor invalidates the following two instructions that pass through the EX stage after a branch instruction. This method was chosen as it can be

done by adding a one-bit register. However, it resulted in a flaw being introduced into the design. This flaw occurs when a branch instruction is taken, and at the same time, there is a stall in either the FE or DE stages. This stall means a NOP is introduced into the pipeline. The EX stage marks this NOP as invalid instead of the actual invalid instruction, meaning the stalled instruction is still executed. This situation doesn't happen in simulation as stalls are manually added in, however, it could happen if there is a cache miss in the FE stage and a branch is taken. This is an error in the design, that wasn't identified till the project had been completed; however, the severity of it is reduced as cache misses in the IMEM are perceived as rare events [37], and these would have to coincide with a branch instruction.

5.2 FPGA Simulation

5.2.1 Clock Speed

As detailed in Section 2.2.1, the challenge for the pipeline is to distribute the processing evenly throughout the different stages. As shown by Figure 5.2, this has been achieved successfully except for the FE stage, which is significantly shorter than the rest of the pipeline allowing for the faster clock frequency shown. The increased possible clock speed achieved by the FE is because all of the calculation for this stage happens in parallel, with the next instruction being fetched in parallel to the PC being incremented. Another reason for this speedup by the FE stage is that in the implementation, both the DMEM and IMEM were reduced in size so they could be synthesised into block RAM. This reduction in the IMEM means that the FE stage has to search through a smaller RAM than other implementations. The ME stage is similar in functionality to the FE stage; however, the ME stage can't achieve similar performance because the DMEM had to be split into four separate blocks of RAM. Whereas the FE stage only fetches instructions from the same block of RAM, however the DMEM must first select which block or blocks of RAM to use. This requirement means that in the ME stage, the address must first go through multiple multiplexers before a memory operation can be started.

The limiting stage for the clock frequency is the EX stage, as highlighted by Figure 5.2. One of the main reasons for this is that the synthesis software uses 'resource sharing' to reduce device utilisation, which means that some arithmetic and logic operations share the same hardware, imposing delays. The synthesis software shows that

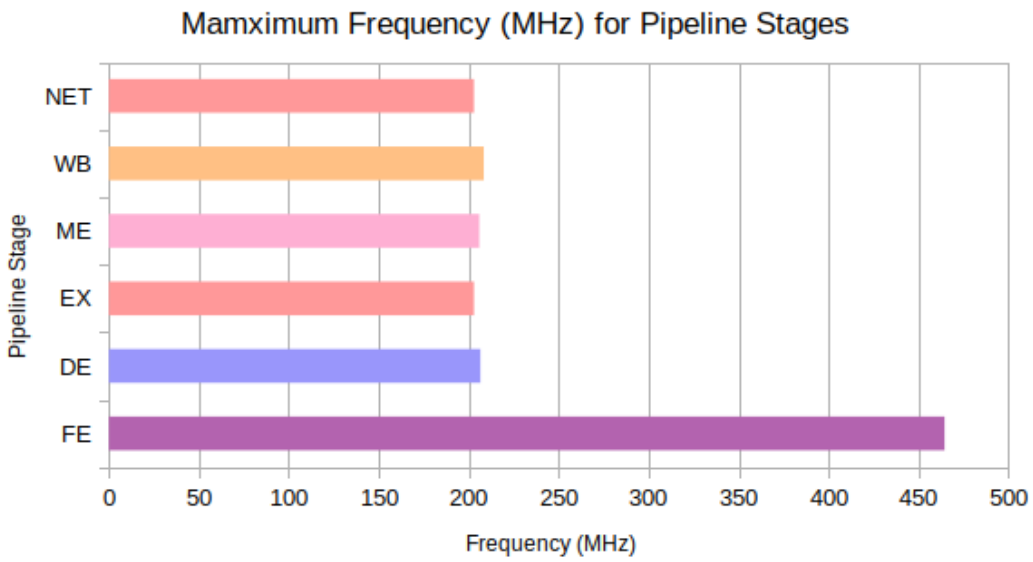

Figure 5.2: Graph showing Maximum Frequency for each Pipeline stage and the Final Netpath

the inputs to the ALU and the COMP unit use 32-bit 8-to-1 multiplexers; this suggests that the software has grouped the inputs to the ALU and COMP unit into dependent multiplexers, as shown in Figure 3.2. This was meant to be avoided; however, the synthesis software still managed to group these multiplexers, reducing hardware requirements but resulting in additional and possibly unnecessary delays.

5.2.2 Hardware Requirements

Section **??** details the limitations of the device being used for the implementation. Figure 5.3 shows that the implementation uses under half of the available device resources, as slices are the primary indication of device utilisation. Slices are subsections of a CLB, with a CLB containing four slices. This utilisation indicates that there is more space for performance improvements, as most of the device CLBs remain unused.

Figure 5.4 shows how the resources are divided between the different stages. The EX stage is the most resource-demanding stage due to the requirements for the ALU and COMP units. Figure 5.4 uses the individual reports produced after synthesising each stage separately; this is important to note as it affects the DE stages synthesis. When the DE stage is synthesised independently from the rest of the pipeline, the register bank is interpreted as a RAM block. This substitution is because the register bank

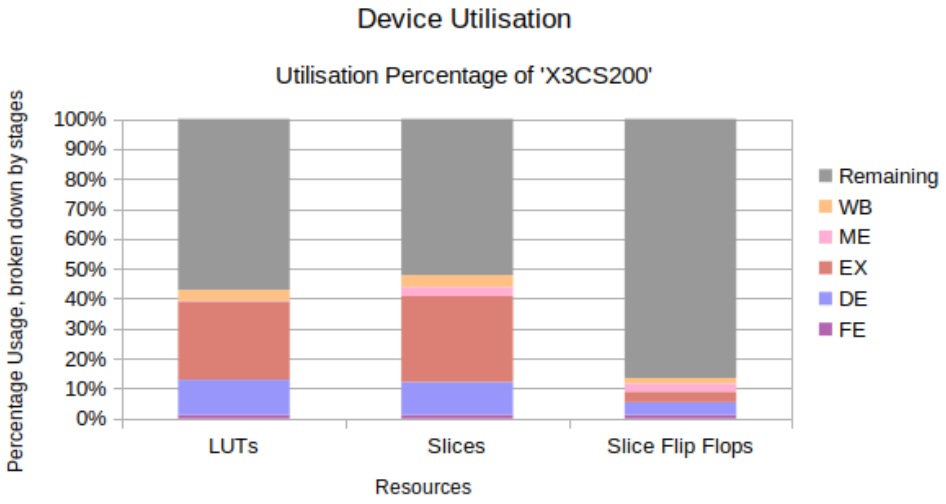

Figure 5.3: Graph showing resources used by each pipeline stage compared to overall device resources

in the design is accessed similarly to a RAM block, synchronously with a dedicated data input register from the WB stage. The use of RAM for a register bank allows the number of slice flip-flops used for the DE stage to be reduced without significant performance consequences. Removing the register bank means that the only registers being used are latches and the registers between pipeline stages. Since the EX, DE and ME stages have the most output registers and latches, these are the stages that make up most of the slice flip-flops.

The majority of Look-Up Tables (LUTs) used are within the EX stage; this is due to the ALU and COMP units using numerous LUTs to implement the logic functions required. The WB and DE stages combined use around 37% of the overall LUTs, this is because both of these stages carry out sign extension and concatenation, the DE on an immediate value and WB on the results from a Load operation.

One issue with the FPGA device used is that it doesn't provide enough RAM blocks to be able to implement a fraction of the address space available, meaning that the ME and FE stages are significantly smaller than they might otherwise be. This means that although the device highlights some inefficiencies with the EX stage, it is hard to evaluate these with respect to the possible overhead of having a much larger implemented address space. Meaning that in a design with a larger RAM the functionality distribution of the pipeline might be split slightly differently.

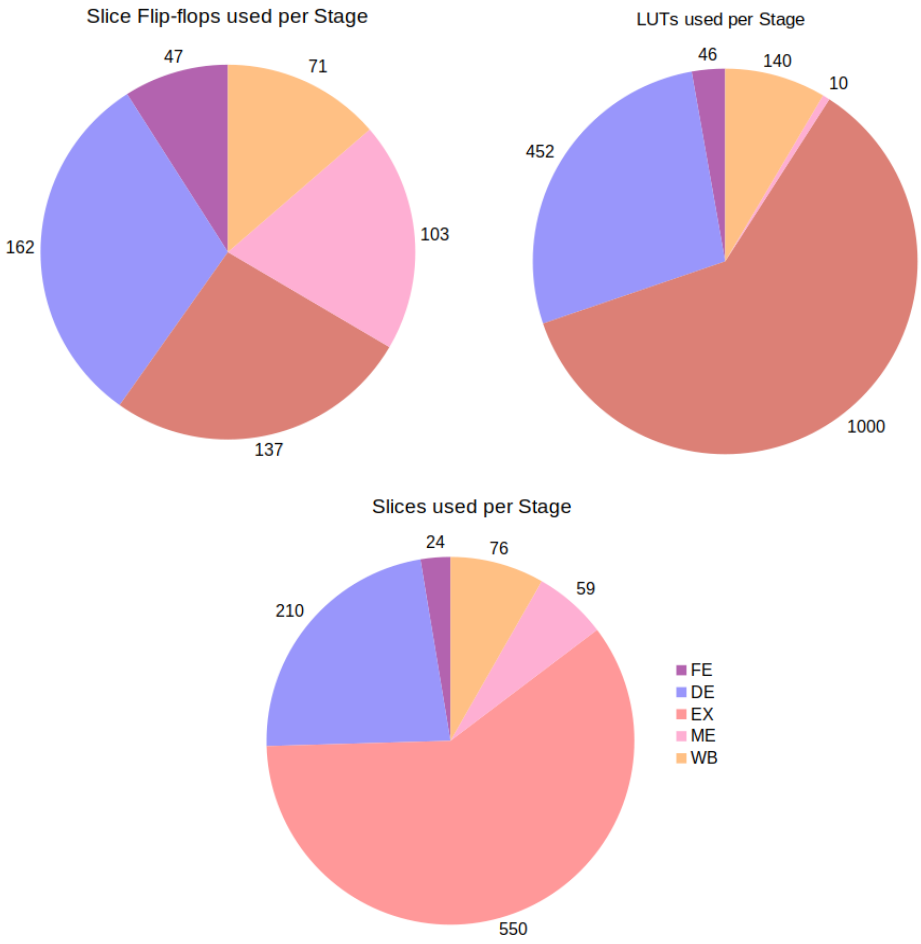

Figure 5.4: Charts Showing Resources used by each pipeline stage

Chapter 6

Conclusion

6.1 Summary of Objectives

The first objective of this project was to 'Develop a processor which implements all instructions from RV32I'. This goal has been achieved fully, with the synthesised processor able to correctly carry out all the instructions from the RV32I ISA, as evidenced by the integration and regression tests. The only cases that the implementation can't deal with are the 'ECALL, EBREAK and FENCE' instructions; these were excluded deliberately as they were unnecessary in a this single privilege mode implementation. These instructions provide functionality such as trap handling, device I/O access, and memory ordering [14]. Overall the project meets this objective in a satisfactory manner.

The second objective listed in Section 1.1: 'Design and Implement a pipeline that will increase the maximum clock speed of the processor'. This objective is harder to evidence than the previous one as the project didn't produce a non-pipelined version of the processor. This lack of comparison makes it harder to prove this objective empirically. However, assuming that a non-pipelined implementation would need to access both IMEM and DMEM in a single cycle, this operation takes at least 8 ns to carry out sequentially; this caps the performance of a non-pipelined implementation to 100 MHz, already half of the performance of the pipelined processor designed and implemented. The implementation, therefore, must meet this objective.

The final objective set was 'Verify the implementation works correctly by simulating the design with a test program'. This is evidenced by the integration and regression tests, showing that the implementation works correctly. As all tests were passed, excluding the specific scenario detailed in Section 5.1.2 where deliberate stalls have to

be introduced into the pipeline. This means that depending on the data, the processor design wouldn't work for an actual hardware implementation; however, since this objective only refers to software implementations, this can be classified as complete.

6.2 Future Work

Due to the modular structure of the RISC-V ISA, described by Table 2.1, and the large percentage of the FPGA device which is unused, shown by Figure 5.3, expansion on this report could fall into two categories, adding functionality or improving performance. Since there is still a large amount of space on the current device with unutilised multiplier blocks and over 200 CLBs left, further work would be to implement an additional module. This module should be the M-expansion as this would give the processor significantly more utility; it also would make the best use of the device's unutilised resources while providing a good way to test longer intentional stores and increase parallelism in the EX stage.

An alternative would be to explore methods of increasing performance and using the remaining CLBs to achieve higher frequencies and instructions per clock. This could be done by exploring out-of-order execution [38], initially by adding a bypass in the ME stage allowing instructions that aren't Load or Store operations to pass this stage when there is a cache miss and the memory operation being executed is stalled. As well as this a branch prediction unit [39] could be added to the FE stage storing the targets of branch instructions reducing the need for flushing on a branch instruction.

6.3 Reflection

This project has demonstrated that the open-source RISC-V ISA allows for small processor implementations suitable to IoT devices that have hardware limitations. This is evidenced in the remaining hardware resources left unused when implementing on the 'Spartan3 X3CS200 device' [33] shown in Figure 5.3. While a high-performance device by modern standards was unable to be implemented, the processor still did take advantage of pipelining methods and forwarding paths to create an implementation that could be synthesised onto hardware. Other implementations such as The Berkeley out-of-order machine [40] demonstrate that producing industry-competitive processors is possible with RISC-V, with this and similar work starting to indicate a shift towards more competitive open-source solutions in the future, as shown by Figure 6.1.

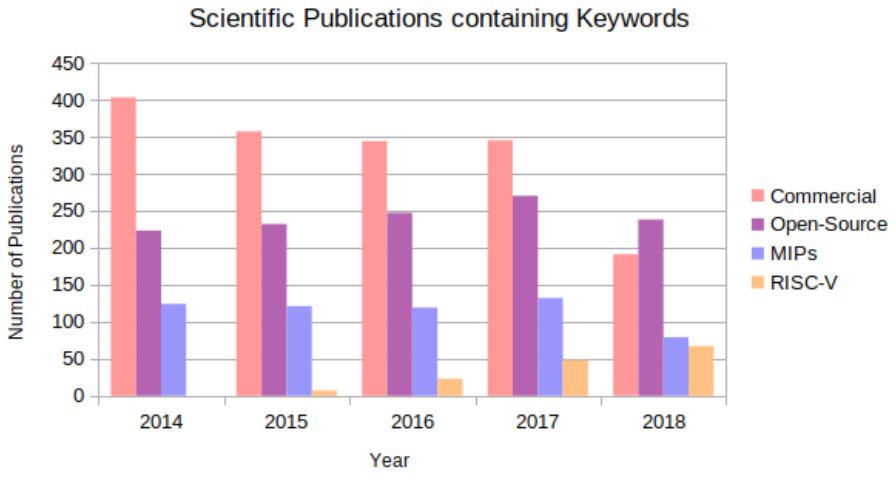

Figure 6.1: Comparison of the annually added number of scientific publications including keywords, Source: [41]

One of the main points that this project highlighted was the challenges in developing for FPGAs, with the synthesis software implementations containing vastly different hardware for similar code with slight variations. Section 4.1.3, demonstrates this effectively showing how slight variations allow the software to be able to detect and implement RAM. This suggests that, although FPGAs have been growing in popularity and therefore development tools are being improved upon, there is still lots of space for improvement and standardisation of implementations, especially surrounding synthesis software [42]. This presents a challenge for RISC-V, as choices in the design of the ISA such as the immediate structure, were specifically driven by a desire to simplify the hardware implementation and if synthesis software is unable to detect these optimisations then these choices could end up producing less efficient designs.

Bibliography

[1] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information systems frontiers*, 17:243–259, 2015.

[2] Yousaf Bin Zikria, Rashid Ali, Muhammad Khalil Afzal, and Sung Won Kim. Next-generation internet of things (IoT): Opportunities, challenges, and solutions. *Sensors*, 21(4):1174, 2021.

[3] RISC-V Foundation. RISC-V international, 2021. Accessed on 23rd September 2022.

[4] RISC-V Foundation. About the RISC-V foundation, 2021. Accessed on 23rd September 2022.

[5] Camille Kokozaki. The revolution evolution continues - sifive risc-v technology symposium, 2019.

[6] Richard M. Stallman et al. Gcc 12.2 manual, 2022.

[7] ARM. Arm a-profile a64 isa, 2022. Accessed on 3rd February 2023.

[8] Shuo Qi, Shao-peng Jin, Yi-hu Xu, and Yi-lin Dai. A five-stage pipeline processor using the RISC-V instruction set architecture designed by system verilog. In *International Conference on Electronic Information Technology (EIT 2022)*, volume 12254, pages 142–148. SPIE, 2022.

[9] Chittoor V Ramamoorthy and Hon Fung Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, 1977.

[10] David A. Patterson and Carlo H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, page 443–457, Washington, DC, USA, 1981. IEEE Computer Society Press.

[11] David A Patterson and David R Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, 1980.

[12] Paul Wallich. Toward simpler, faster computers: By omitting unnecessary functions, designers of reduced-instruction-set computers increase system speed and hold down equipment costs. *IEEE Spectrum*, 1985.

[13] Intel. Intel 64 and ia-32 architectures software developer's manual, 2022. Accessed on 10th February 2023.

[14] SiFive Inc. The RISC-V instruction set manual, volume i: Unprivileged ISA, 2019.

[15] David Kanter. RISC-V offers simple, modular ISA. *Microprocessor Report*, pages 4–5, 2016.

[16] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[17] ARM. ARM cortex-a series programmers guide for ARMv7-a, 2014. Accessed on 10th February 2023.

[18] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. RISC-V extension for lightweight cryptography. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 222–228. IEEE, 2020.

[19] Andrew Waterman and Kriste Asanovic. The RISC-V instruction set manual; volume ii: Privileged architecture, sifive inc. and cs division, EECS department. *University of California, Berkeley*, 1, 2017.

[20] ARM. ARM architecture reference manual armv7-a, 2008. Accessed on 10th February 2023.

[21] Various. RISC-V: What's missing and who's competing? comments, 2020. Accessed on 2nd October 2022.

[22] Johannes Knödtel, Sebastian Rachuj, and Marc Reichenbach. Suitability of ISAs for data paths based on redundant number systems: Is RISC-V the best? In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 247–253. IEEE, 2022.

[23] Allan Hartstein and Thomas R Puzak. The optimum pipeline depth for a microprocessor. *ACM Sigarch Computer Architecture News*, 30(2):7–13, 2002.

[24] Peter M Kogge. *The architecture of pipelined computers*. CRC press, 1981.

[25] Ian Finlayson, Gang-Ryung Uh, David B Whalley, and Gary Tyson. An overview of static pipelining. *IEEE Computer Architecture Letters*, 11(1):17–20, 2011.

[26] Emma and Davidson. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers*, C-36(7):859–875, 1987.

[27] Hoang Anh Tuan, Katsuhiro Yamazaki, and Shigeru Oyanagi. Three-stage pipeline implementation for SHA2 using data forwarding. In *2008 International Conference on Field Programmable Logic and Applications*, pages 29–34. IEEE, 2008.

[28] Arthur Abnous and Nader Bagherzadeh. Pipelining and bypassing in a VLIW processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):658–664, 1994.

[29] Mohammed Elnawawy, Abid Farhan, Ahmad Al Nabulsi, Abdul-Rahman Al-Ali, and Assim Sagahyroon. Role of FPGA in internet of things applications. In *2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 1–6. IEEE, 2019.

[30] Alexander Barkalov, Larysa Titarenko, and Kamil Mielcarek. Improving characteristics of LUT-based mealy FSMs. *International Journal of Applied Mathematics and Computer Science*, 30(4):745–759, 2020.

[31] IEEE standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[32] Xilinx. Vivado design suite user guide: Synthesis; ug901 (v2019. 1). 2022.

[33] DS099 Xilinx. Spartan-3 FPGA family: Complete data sheet. *Product Documentation,(November 2005)*, 2008.

[34] Timothy Sherwood and Brad Calder. Automated design of finite state machine predictors for customized processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, page 86–97, New York, NY, USA, 2001. Association for Computing Machinery.

[35] Brian W Kernighan and Dennis M Ritchie. The c programming language. 2002.

[36] U Finkelstein. GTKwave 3.3 wave analyzer user's guide, 2011.

[37] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, sep 1982.

[38] Stephen A Zekany, Jielun Tan, and James A Connolly. Teaching out-of-order processor design with the RISC-V ISA. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, pages 1–8. IEEE, 2021.

[39] C Arul Rathi, G Rajakumar, T Ananth Kumar, and TS Arun Samuel. Design and development of an efficient branch predictor for an in-order RISC-V processor. 2020.

[40] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized RISC-V processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[41] Roland Höller, Dominic Haselberger, Dominik Ballek, Peter Rössler, Markus Krapfenbauer, and Martin Linauer. Open-source RISC-V processor ip cores for FPGAs — overview and evaluation. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2019.

[42] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.