Multi-core RISC Processor Design and Implementation (Rev. 1.00)

ELEC5881M - Interim Report

Ben David Lancaster
Student ID: 201280376

Submitted in accordance with the requirements for the degree of
Master of Science (MSc)
in Embedded Systems Engineering

Supervisor: Dr. David Cowell
Assessor: Mr David Moore

University of Leeds
School of Electrical and Electronic Engineering

April 29, 2019

Abstract

This interim report details the 4-month progress on a project to design, implement, and verify, a multi-core FPGA RISC processor. The project has been split into two stages: firstly to build a functional single-core RISC processor, and then secondly to add multi-core principles and functionality to it.

Current multi-core and network-on-chip communication methods have been discussed and how they could be included in this multi-core RISC design. To-date, a 16-bit instruction set architecture has been designed featuring common load/store instructions, comparison, and bitwise operations. A single-core processor has been implemented in Verilog and verified using simulations/test benches running various simple programs.

Future tasks have been planned and will focus on the second stage of the project. Work will start on designing multi-core communication interfaces and bringing them to the single-core processor.

Revision History

Date	Version	Changes
10/04/2019	2.01	Add introduction.
05/04/2019	2.01	Fix processor RTL diagram.
04/04/2019	2.00	Initial processor RTL diagram.
01/04/2019	1.00	Initial section outline.

Table 1: Document revisions.

Declaration of Academic Integrity

The candidate confirms that the work submitted is his/her own, except where work which has formed part of jointly-authored publications has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated in the report. The candidate confirms that appropriate credit has been given within the report where reference has been made to the work of others.

This copy has been supplied on the understanding that no quotation from the report may be published without proper acknowledgement. The candidate, however, confirms his/her consent to the University of Leeds copying and distributing all or part of this work in any forms and using third parties, who might be outside the University, to monitor breaches of regulations, to verify whether this work contains plagiarised material, and for quality assurance purposes.

The candidate confirms that the details of any mitigating circumstances have been submitted to the Student Support Office at the School of Electronic and Electrical Engineering, at the University of Leeds.

Name: Ben David Lancaster
Date: April 29, 2019

Table of Contents

Chapter 1

Introduction

This project will detail the design, implementation, and verification, of a new multi-core RISC processor aimed at FPGA devices. This project was chosen due to my interest in processor design, in which I have only previously designed single-core RISC processors and wish to extend this knowledge to gain a basic understanding of multi-core communication, design considerations, and the limitations of parallelism first hand.

I will use this opportunity to further develop my knowledge of FPGA and processor design by implementing, designing, and verifying, a multi-core RISC processor from scratch, including the design of a communication interface between multiple cores.

1.1 Why Multi-core?

Moore's Law states that the number of transistors in a chip will double every 2 years []. CPU designers would utilize the additional transistors to add more pipeline stages in the processor to reduce the propagation delay [] which would allow for higher clock frequencies.

The size of transistors have been decreasing [] and today can be manufactured in sub-10 nanometer range. However, the extremely small transistor size increases electrical leakage and other negative effects resulting in unreliability and potential damage to the transistor []. The high transistor count produces large amounts of heat and requires increasing power to supply the chip. These trade-offs are currently managed by reducing the input voltage, utilising complex cooling techniques, and reducing clock frequency. These factors limit the performance of the chip significantly. These are contributing factors to Moore's Law *slowing* down. The capacity limit of the current-generation planar transistors is approaching and so in order for performance increases to continue, other approaches such as alternate transistor technologies like Multigate transistors [1], software and hardware optimisations, and multi-processor architectures are employed.

This report will focus on the latter: to produce a small multi-core processor that can utilise software-based parallelism to gain performance benefits, compared to a larger single-core design.

1.2 Why RISC?

RISC architectures feature simpler and fewer instructions compared to CISC, which emphasises instructions that perform larger tasks. A single CISC instruction might be performed with multiple RISC instructions. Because of the fewer and simpler instructions, RISC machines rely heavily on software optimisations for performance. RISC instruction sets are based on load/store architectures, where most instructions are either register-to-register or memory reading and writing [2]. This constraint greatly reduces complexity.

RISC architectures are easier to design implement, especially for beginners, due to their simpler instructions that share the same pipeline, compared to CISC where there may be different pipeline for each instruction, which would greatly consume FPGA resources.

1.3 Why FPGA?

Field programmable gate arrays (FPGA) are a great choice for prototyping digital logic designs due to their programmable nature and quick development times.

FPGAs, however, may not be suitable for prototyping all register-transistor logic (RTL) projects. Larger RTL projects, such as large commercial processors, may greatly exceed the logic cell resources available in today's high-end FPGA devices and may only be prototyped through silicon fabrication, which can be expensive. This restriction will not be problem

Chapter 2

Background

2.1 Parallelism and Amdahl's Law

In serial software programs there may exist many potential opportunities for parallelism.

2.2 Single core vs. Multi-core vs. Many-core

2.3 Network-on-chip

Network-on-chip (NoC) architectures implement on-chip communication mechanisms that are based on network communication principles, such as routing, switching, and massive scalability.

2.3.1 OpenPiton

2.3.2 Concurrency Problems

The "critical path" in an algorithm is described as the longest sequence of instructions that must be performed sequentially due to data dependencies within the instruction sequence [**?**].

Additional problems include access to shared resources and memory coherency in distributed systems. These will be discussed.

Data Dependencies

Data dependencies are a critical obstacle in parallelising serial programs.

```
1 function has_dependency(a, b):
2     x = a + b
3     y = x + b
```

Figure 2.1: Sequence of instructions where a data dependency exists between instructions 2 and 3 – Instruction 3 must wait for variable x to be ready.

```
1 function no_dependency(a, b):
2     x = a + b
3     y = b + 2
```

Figure 2.2: Sequence of instructions with no data dependencies. This sequence can be ran in parallel using multiple processing cores.

Access to shared memory

Memory Coherency

2.4 Summary

Chapter 3

Project Overview

This chapter discusses the the project's requirements, goals, risks, and structure.

3.1 Project Deliverables

The project's deliverables are split into two sections: core deliverables (CD) – each deliverable must be satisfied for the project to be a minimum viable product (MVP), and extended deliverables (ED) – deliverables that are not required for a MVP – features that only improve upon an existing feature.

3.1.1 Core Deliverables (CD)

The project's core deliverables are described below.

CD1 Design a compact 16-bit RISC instruction set architecture.

CD2 Design and implement a Verilog RISC core that implements the ISA in **CD1**.

CD3 Design and implement an on-chip interconnect for multi-core processing (2 to 32 cores) using the RISC core from **CD2**.

CD4 Analyse performance of serial and parallel software algorithms, such as parallel DFT [**?**], on the processor.

CD5 Allow the RISC core to be easily compiled to multiple FPGA vendors (Xilinx, Altera).

3.1.2 Extended Deliverables (ED)

The project's extended deliverables are described below.

ED1 Design a RISC core with an instructions-per-clock (IPC) rating of at least 1.0 (a single-cycle CPU).

ED2 Design a RISC core with a pipe-lined data path to increase the design's clock speed.

ED3 Design a scalable multi-core interconnect supporting arbitrary (more than 32) RISC core instances (manycore) using Network-on-Chip (NoC) architecture.

ED4 Design a compiler-backend for the PRCO304 [**?**] compiler to support the ISA from1 **CD1**. This will make it easier to build complex multi-core software for the processor.

ED5 The RISC core can communicate to peripherals via a memory-mapped addresses using the Wishbone [**?**] bus.

ED6 Implement various memory-mapped peripherals such as UART, GPIO, LCD, to aid visual representation of the processor during the demonstration viva.

ED7 Store instruction memory in SPI flash.

ED8 Reprogram instruction memory at runtime from host computer.

ED9 Processor external debugger using host-processor link.

3.2 Project Timeline

3.2.1 Project Stages

The project is split up into many stages to aid planning and management of the project. There are 8 unique stage areas: 1. Inital project conception; 2 Basic RISC core development; 3. Extended RISC core development; 4. Multi-core development; 5. Processor quality-of-life (QoL) improvements; 6. Compiler development; 7. Demo preparation, and 8. Final report.

The project stages are shown in Table 3.1.

3.2.2 Timeline

The project stages from Table 3.1 are displayed below in a Gantt chart.

3.3 Resources

This section describes the hardware and software resources required to design and implement the project.

Stage	Title	Start Date	Days	Core	Applicable Deliverables
1.0	Research	Feb 04	7	x	
1.1	Requirement gathering/review	Feb 11	14	x	
1.1	Processor specification, architecture, ISA	Feb 18	100	x	**CD1**
1.2	Stage/Time Allocation Planning	Feb 25	7	x	
2.1	Decoder, Register Set, impl & integration	Feb 25	14	x	**CD2**
2.2	Register set impl & integration	Mar 04	14	x	**CD2**
2.3	Local memory impl & integration	Mar 11	14	x	**CD2**
3.1	Memory mapped register layout & impl	Apr 01	21		**ED5**
3.2	Wishbone peripheral bus connected to MMU	Apr 08	21		**ED5**
3.3	Pipelined implementation and verification	Apr 15	21		**ED2**
3.4	Cache memory design & impl	Apr 22	28		**ED2**
4.1	Multi-core communication interface	TBD	TBD	x	**CD3**
4.2	Shared-memory controller	TBD	TBD	x	**CD3**
4.3	Scalable multi-core interface (10s of cores)	TBD	TBD	x	**CD3**
4.4	Multi-core example program (reduction)	TBD	TBD	x	**CD4**
5.1	SPI-FPGA interface for OTG programming	TBD	TBD		**ED7**
5.2	FPGA-PC interfacing	TBD	TBD		**ED9**
5.3	FPGA-PC debugging (instruction breakpoints)	TBD	TBD		**ED9**
6.1	Compiler backend for vmicro16	TBD	TBD		**ED4**
6.2	Compiler support for multi-core codegen	TBD	TBD		**ED4**
7.1	Wishbone peripherals for demo	TBD	TBD	x	**CD4**
8.1	Final Report	TBD	TBD	x	

Table 3.1: Project stages throughout the life cycle of the project.

3.3.1 Hardware Resources

Core deliverable **CD5** requires the designed RISC core to be implemented and demonstrated on multiple FPGA devices. Although my design should synthesise for physical IC implementation, due to high costs and lengthy production times, it is not a primary development target.

Due to having past experience with Xilinx FPGAs from my placement work and experience with Altera from university modules it was decided to target the Xilinx Spartan 6 XC6SLX9 and the Altera Cyclone V.

Terasic DE1-SoC Development Board

The Terasic DE1-SoC development board features a large Cyclone V FPGA and many peripherals, such as seven-segment displays, 64 MB SDRAM, ADCs, and buttons and switches, which will aid demonstration of the project. The development board is available through the university so the cost is negligible. Figure 3.2 shows the peripherals (green) available to the

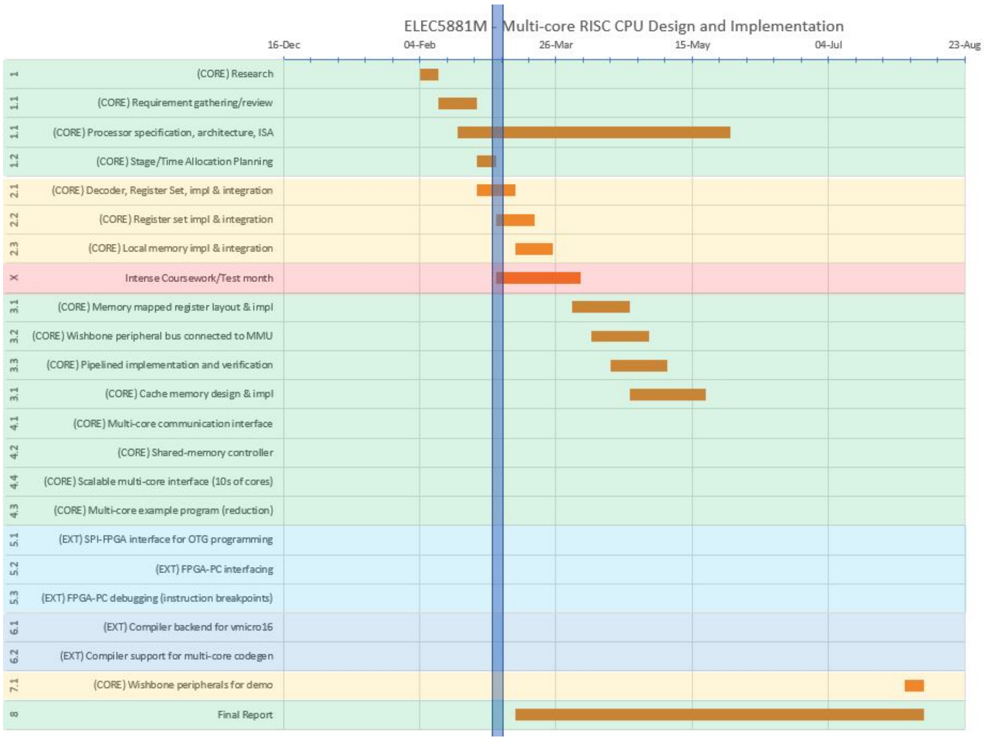

Figure 3.1: Project stages in a Gantt chart.

FPGA.

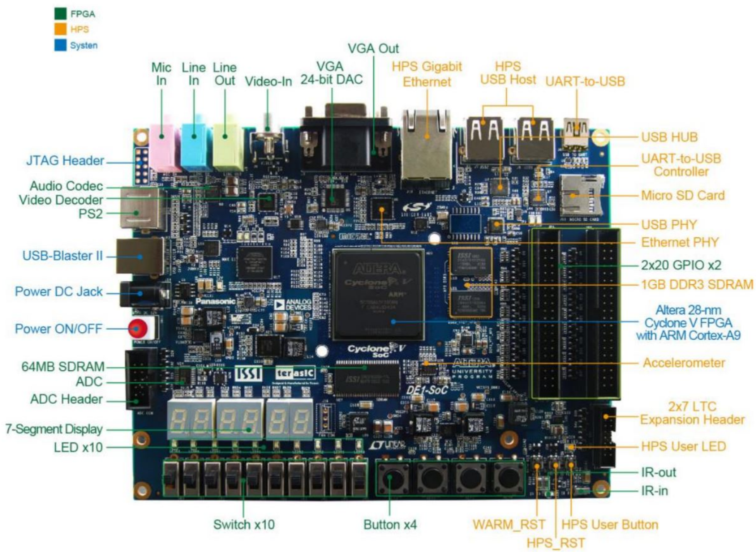

Figure 3.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [3].

Minispartan 6+ FPGA Development Board

The Minispartan 6+ is a hobbyist FGPA development board with fewer peripherals than the DE1-SoC. The board's simplicity will ease debugging. The board features a Xilinx Spartan 6 XC6LX9 however it's simplicity and Xilinx's software suite will speed up development.

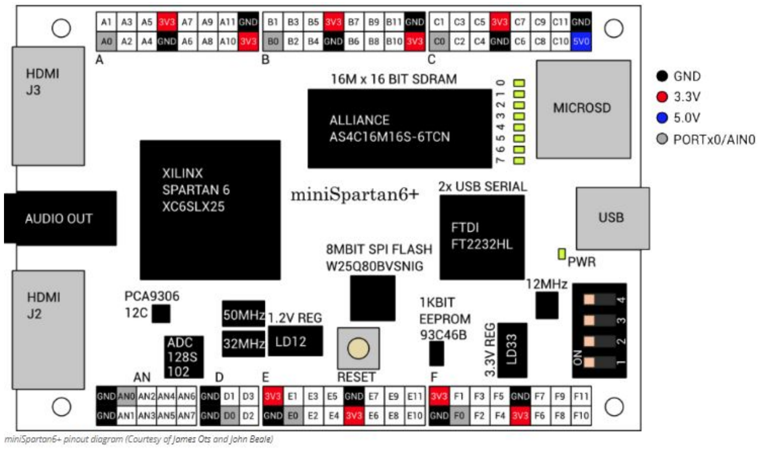

Figure 3.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Image source: [4].

3.3.2 Software Resources

Intel Quartus

Intel Quartus Prime is a paid-for SoC, CPLD, and FPGA software suite targeting Intel's Stratix, Arria, and Cyclone based FPGAs. The university provides student licences which will be used via VPN.

Xilinx ISE Webpack

Xilinx ISE Webkpack is Xilinx's free software suite for FPGA development for Spartan 6 based FPGAs. Due to ISE's intuitive and fast work flow, most of the initial simulation and verification processes will be performed using ISE. This will greatly improve development times.

Verilator

Verilator is an open-source Verilog to C++ transpiler which provides a C++ interface to simulate Verilog modules and read/write values similar to a test bench. Verilator will be used for specific modules within the RISC core such as the ALU and decoder as Verilator is useful when performing exhaustive verification.

3.4 Legal and Ethical Considerations

The RISC core is designed to be used as an academic research and educational tool to aid learning and understanding of RISC and multi-core machines. It should not be use for roles where mission critical or safety is a factor.

The processor does not provide any memory protection features and any software running on the processor has full access to all memory.

The processor does not store/track/predict software instructions. The processor uses pipelining techniques to improve performance which results in future instructions entering the pipeline even if the software's logical sequence does not include said instructions. This could result in security vulnerabilities similar to Intel's Spectre vulnerability [5].

Chapter 4

Current Progress

This chapter discusses the current progress made towards the project, including designs, implementation, and current results.

4.1 RISC Core

Following the project time line described in section 3.2, the first couple months have been dedicated to the design and implementation of the instruction set architecture and RISC core with stages 1-3. Good progress has been made in both deliverables, the ISA and the RISC core, and the progress is on-time with the initial project time line.

4.1.1 Instruction Set Architecture

A 16-bit instruction set architecture (ISA) has been designed using an iterative approach. There currently exists 32 unique instructions covering most generic RISC operations (add, load/store, branch, compare, etc.) and atleast 16 opcodes available to be provide multi-core communication and functionality. This number should be adequate to support these features when the work begins on the multi-core project stages (stages 4-7).

Design Goals

Having past experience designing and implementing ISAs for previous projects, I wanted to use that knowledge to design an even more efficient and compact instruction set that could provide much greater functionality. The technical design goals of the ISA are described below:

ISA1 Use a fixed width of 16-bits for all instructions.
This will significantly reduce RTL resources and encourage efficiency by not wasting spare bits. In addition, many SPI flash and RAMs support 16-bit wide data reads which will allow each instruction fetch to only require one clock cycle, thus increasing processor performance.

ISA2 Be able to select at least two registers for common instructions.
This will reduce the number of required instructions to manipulate register data. A disadvantage of using two instead of three reigster selects is that instructions are always destructive – they always *destroy* existing data in the destination register (e.g. R0 = ADD R0 R1) unlike constructive instructions that provide a unique register select for the destination (e.g. R2 = ADD R0 R1).

ISA3 Reduce bit-space for frequently used instructions (MOV, MOVI, ADD).
Due to the 16-bit limit, two register selects, and immediate values, the opcode bits are reduced resulting in fewer unique instructions. To overcome this constraint, spare bits in other instructions will be appended to the opcode bits to extend the opcode range. This however, will require a more complex decoder that must first switch the opcode, then switch any spare bits to determine the final opcode. This method will significantly increase the number of unique instructions provided by the instruction set.

ISA4 Provide frequently used actions as options for existing instructions.
In software, frequently used actions include incrementing/decrementing by 1 and performing logical comparisons which usually take more than one instruction on some RISC architectures. As they are common actions, the instruction overhead and time may be significant and can affect performance. To provide a solution to this problem, in addition to using spare bits to extend the opcode range, spare bits will be used to signify a frequently used action action to be performed by the ALU.

As shown in Figure 4.1, frequently used commands such as incrementing/decrementing and logical comparions are provided by setting spare bits to special values. For example, the instructions `ARITH_UADDI` and `ARITH_SSUBI` extend the `ARITH_U` and `ARITH_S` opcodes by filling the spare bit, 4. If this bit is not set (0), the instruction allows for a 4-bit immediate value to be added in addition to the two register selects. The 4-bit immediate allows adding a small number to the ALU which is useful in the case of software for loops where an increment/decrement of more than 1 is required.

Another example is the `SETC` instruction. Inspired by Intel's x86 `SETCC`, the instructions sets the destination register to zero or one depending on the result of the `CMP` instruction's flags. Without this instruction, multiple branches would be required to convert the comparion's flags to logical zeros and ones.

ISA5 Provide instructions for performing bitwise manipulations.
RISC processors are commonly used for microprocessing and microcontroller actions which typically includes bit manipulation. The ISA provides bitwise OR, XOR, AND, NOT, and shifting instructions under a single opcode to fill this need.

ISA6 Provide instructions for explicitly performing signed and unsigned arithmetic.
Performing signed and unsigned arithmetic is a key requirement for RISC applications and so it was decided to provide such instructions. Software programmers can easily switch between signed and unsigned arithmetic by setting bit 11 in the `ARITH` instruction family. Being able to change between signed and unsigned arithmetic instructions by changing a single bit will make the RISC processor's decoder module smaller and less complex.

Without explicit unsigned and signed instructions, extra instructions would be required to perform addition and subtraction. In addition, due to two's complement representation of signed numbers, the highest immediate operand value would be halved, resulting in more instructions to reach the desired value.

	15-11	**10-8**	**7-5**	**4-0**	**rd ra simm5**
	15-11	**10-8**	**7-0**		**rd imm8**
	15-11	**10-0**			**nop**
	15	**14:12**	**11:0**		**extended immediate**
NOP	00000	X			
LW	00001	Rd	Ra	s5	Rd <= RAM[Ra+s5]
SW	00010	Rd	Ra	s5	RAM[Ra+s5] <= Rd
BIT	**00011**	**Rd**	**Ra**	**s5**	**bitwise operations**
BIT_OR	00011	Rd	Ra	00000	Rd <= Rd | Ra
BIT_XOR	00011	Rd	Ra	00001	Rd <= Rd ^ Ra
BIT_AND	00011	Rd	Ra	00010	Rd <= Rd & Ra
BIT_NOT	00011	Rd	Ra	00011	Rd <= ~Ra
BIT_LSHFT	00011	Rd	Ra	00100	Rd <= Rd << Ra
BIT_RSHFT	00011	Rd	Ra	00101	Rd <= Rd >> Ra
MOV	00100	Rd	Ra	X	Rd <= Ra
MOVI	00101	Rd	i8		Rd <= i8
ARITH_U	**00110**	**Rd**	**Ra**	**s5**	**unsigned arithmetic**
ARITH_UADD	00110	Rd	Ra	11111	Rd <= uRd + uRa
ARITH_USUB	00110	Rd	Ra	10000	Rd <= uRd - uRa
ARITH_UADDI	00110	Rd	Ra	0AAAA	Rd <= uRd + Ra + AAAA
ARITH_S	**00111**	**Rd**	**Ra**	**s5**	**signed arithmetic**
ARITH_SADD	00111	Rd	Ra	11111	Rd <= sRd + sRa
ARITH_SSUB	00111	Rd	Ra	10000	Rd <= sRd - sRa
ARITH_SSUBI	00111	Rd	Ra	0AAAA	Rd <= sRd - sRa + AAAA
BR	**01000**	**Rd**	**i8**		**conditional branch**
BR_U	01000	Rd	0000 0000		Any
BR_E	01000	Rd	0000 0001		Z=1
BR_NE	01000	Rd	0000 0010		Z=0
BR_G	01000	Rd	0000 0011		Z=0 and S=O
BR_GE	01000	Rd	0000 0100		S=O
BR_L	01000	Rd	0000 0101		S != O
BR_LE	01000	Rd	0000 0110		Z=1 or (S != O)
BR_S	01000	Rd	0000 0111		S=1
BR_NS	01000	Rd	0000 1000		S=0
CMP	01001	Rd	Ra	X	SZO <= CMP(Rd, Ra)
SETC	01010	Rd	Ra	X	Rd <= Imm8 == SZO ? 1 : 0
MOVI_LARGE	1	Rd	i12		Rd <= i12

Figure 4.1: Initial Vmicro16 16-bit instruction set architecture. Coloured regions represent instruction families (bitwise, branching, arithmetic, etc.).

The ISA table is shown in Figure 4.1. The top 5 bits (15-11) are dedicated to the opcode resulting in 32 unique values. Currently only the bits 14-11 are used (`NOP` to `SETC`) leaving the top bit spare. Initially, this bit was reserved to indicate an extended immediate instruction, `MOVI12`, supporting a large 12-bit immediate value, however later in the design it was decided that the top bit would indicate special instructions dedicated for multi-core operation. This leaves 16 spare unique opcodes for this purpose.

4.1.2 Design and Implementation

The RISC core design is a traditional 5-stage processor (fetch, decode, execute, memory, write-back).

To satisfy **CD5**, the Verilog code will be self-contained in a single file. This reduces the hierarchical complexity and eases cross-vendor project set-up as only a single file is required to be included. A disadvantage with this single file approach is that some external Verilog verification tools that I plan to use, such as Verilator, do not currently support multiple Verilog modules (due to an unfixed bug) within a single file.

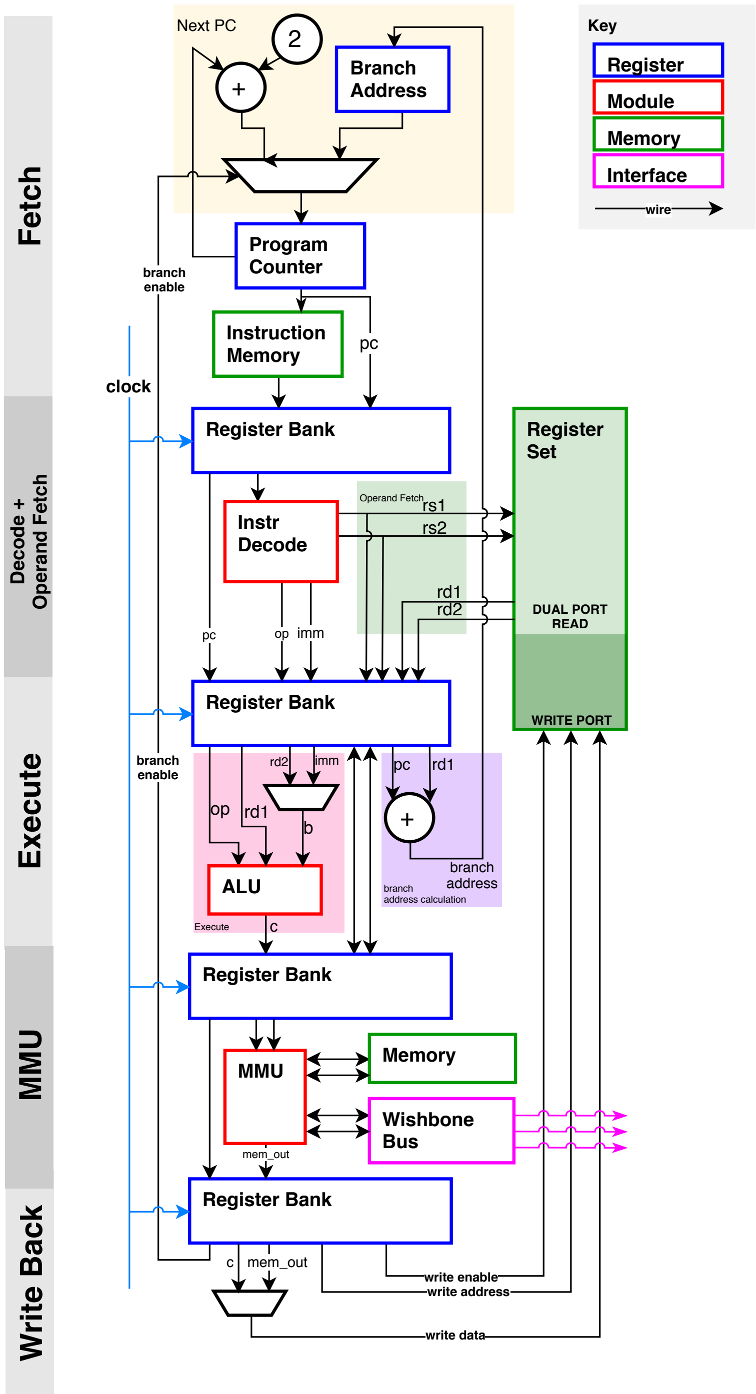

Figure 4.2: Vmicro16 RISC 5-stage RTL diagram.

Instruction and Data Memory

The design uses separate instruction and data memories similar to a Harvard architecture computer. This architecture was chosen due because I find it easier to implement.

Register File

To support design goal **ISA2**, the register set features a dual-port read and single-port write. This allows instructions to read 2 registers simultaneously for any instruction. The single-port write allows the instruction output to be written to the register file.

Pipelining

The extended deliverable **ED1**, to provide atleast 1 instructions per clock. Previous processor designs of mine have all required multiple clocks per instruction as it is a lot easier to implement. Modern processors today can output 1 or more instructions per clock through the use of instruction pipelining. This technique increases throughput of the processor by performing each stage in parallel. In this pipeline, instructions still travel through each stage in the same order, the difference is that the fetch stage does not wait for the final stage to complete and so fetches a new instruction every clock cycle, resulting in each stage operating on new data every clock cycle. To extend my knowledge in CPU pipelining, extended deliverable **ED1** is proposed.

Instruction pipelining is harder to implement as data and control hazards can occur. Data hazards occur when instructions are dependant on the output of a previous instruction that has not left the pipeline, for example a register dependency. Methods to detect this hazard include checking if the register selects in the decode stage are present in future stages of the pipeline. If this check is true, then the current instruction depends on an instruction in the pipeline, and the processor can either wait until the dependant instruction has left the pipeline (i.e. has been written back to registers) or insert a `NOP` that will produce a *bubble* in the pipeline allowing the final stage to execute before the dependant instruction continues.

Control hazards occur when conditional or interrupt branching instructions are in the pipeline and their result has not been calculated yet. This results in preceding instructions entering the pipeline when they should not be executed due to the conditional branch. To detect this hazard, for instructions that perform branching or conditional execution, a global flag is set. When the outcome of the conditional check is performed, stages after decode are allowed to commit their results. Fortunately this technique is fairly simple implement.

This project's RISC processor implements these two hazard detectors and solutions to resolve them. The data hazard resolver implements a `valid` signal that is passed forward from stage to stage. This signal is low when a hazard has occured and indicates that receiving stage should not operate on the previous stage's data. Each stage's `valid` signal is dependant on the previous stages `valid` signal. This allows future stages to stall when a hazard is detected in previous stages. A diagram of the implementation of these hazards in the processor is shown in Figure 4.3.

Memory Management Unit

It was decided to use a memory management unit (MMU) to make it easier and extensible to communicate with external peripherals or additional registers. This method would trans-

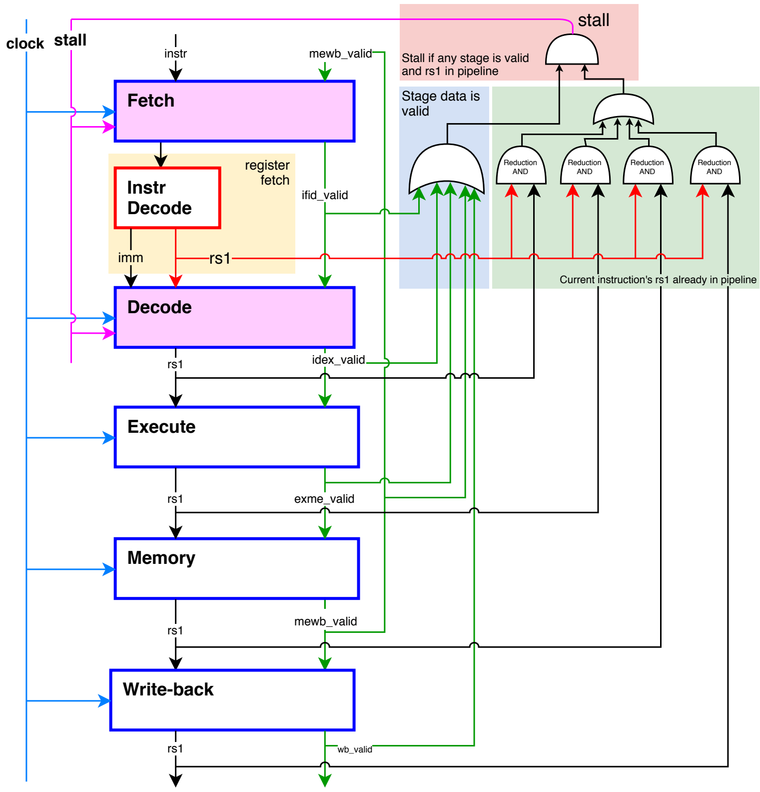

Figure 4.3: Pipeline stall detection logic.

parently use the existing `LW/SW` instructions which removes the requirement for a unique instruction for each peripheral.

Proposed Memory Mapped Addresses

The proposed memory mapped addresses for each system and peripheral are listed below.

Address (16-bit aligned)	Peripheral Name
`0x0000`	NOP (reads returns 0, writes do nothing)
`0x00ZZ`	Per-core scratch RAM (ZZ = 8-bit RAM address)
`0x0100`	Extended Core Registers 1
`0x0200`	Extended Core Registers 2
`0x03ZZ`	Wishbone Master controller select (ZZ contains 8-bit wishbone slave address)
`0x1XYZ`	Master core controller (X = slave select, Y = instruction, Z = data)

Table 4.1: Project stages throughout the life cycle of the project.

ALU Design

The Vmicro16's ALU is an asynchronous module that has 3 inputs: data a; data b; and opcode op, and outputs data value c. The ALU is able to operand on both register data (`rd1` and `rd2`) and immediate values. A switch is used to set the `b` input to either the `rd2` or `imm` value from the previous stage.

Currently, the ALU does not store flags to indicate overflow, equality, or zero values in the module itself. Instead the ALU outputs the result of the `CMP`, which calculates such flags, to be written back to the register set in the write-back stage. This means that in order to perform a conditional operation, such as a branch, the register containing the `CMP` flags must be included in the instruction.

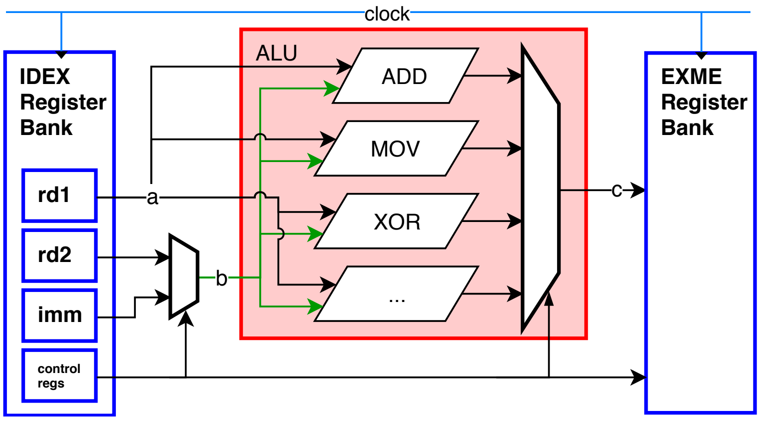

Figure 4.4: Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

The Verilog implementation of the ALU is shown in Figure 4.5. The ALU's asynchronous output is clocked with other registers, such as destination register `rs1` and other control signals, in the `EXME` register bank.

```
        always @(*) case (op)
                // branch/nop, output nothing
                `VMICRO16_ALU_BR,
                `VMICRO16_ALU_NOP:          c = 0;
                // load/store addresses (use value in rd2)
                `VMICRO16_ALU_LW,
                `VMICRO16_ALU_SW:           c = b;
                // bitwise operations
                `VMICRO16_ALU_BIT_OR:       c = a | b;
                `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
                `VMICRO16_ALU_BIT_AND:      c = a & b;
                `VMICRO16_ALU_BIT_NOT:      c = ~(b);
                `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
                `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;
```

Figure 4.5: Vmicro16's ALU implementation named vmicro16_alu. vmicro16.v

Decoder Design

Instruction decoding occurs in the between the IFID and IDEX stages. The decoder extracts register selects and operands from the input instruction. The decoder outputs are asynchronous which allows the register selects to be passed to the register set and register data to be read asynchronously. The register selects and register read data is then clocked into the IDEX register bank.

```
        always @(*) case (opcode)
                `VMICRO16_OP_HALT, // TODO: stop ifid
                `VMICRO16_OP_NOP:              alu_op = `VMICRO16_ALU_NOP;

                `VMICRO16_OP_LW:               alu_op = `VMICRO16_ALU_LW;
                `VMICRO16_OP_SW:               alu_op = `VMICRO16_ALU_SW;

                `VMICRO16_OP_MOV:              alu_op = `VMICRO16_ALU_MOV;
                `VMICRO16_OP_MOVI:             alu_op = `VMICRO16_ALU_MOVI;
                `VMICRO16_OP_MOVI_L:           alu_op = `VMICRO16_ALU_MOVI_L;

                `VMICRO16_OP_BR:               alu_op = `VMICRO16_ALU_BR;

                `VMICRO16_OP_BIT:          casez (simm5)
                        `VMICRO16_OP_BIT_OR:      alu_op = `VMICRO16_ALU_BIT_OR;
                        `VMICRO16_OP_BIT_XOR:     alu_op = `VMICRO16_ALU_BIT_XOR;
                        `VMICRO16_OP_BIT_AND:     alu_op = `VMICRO16_ALU_BIT_AND;
                        `VMICRO16_OP_BIT_NOT:     alu_op = `VMICRO16_ALU_BIT_NOT;
                        `VMICRO16_OP_BIT_LSHFT:   alu_op = `VMICRO16_ALU_BIT_LSHFT;
                        `VMICRO16_OP_BIT_RSHFT:   alu_op = `VMICRO16_ALU_BIT_RSHFT;
                        default:                  alu_op = `VMICRO16_ALU_BAD; endcase
```

Figure 4.6: Vmicro16's decoder module code showing nested bit switches to determine the intended opcode. vmicro16.v

In Figure 4.6, it can be seen that the first 8 opcode cases are represented using the same 15-11 bits, however the `VMICRO16_OP_BIT` instructions require another bit range to be compared to determine the output opcode.

4.1.3 Verification

Currently, the only verification method used is manual inspection of the output waveforms of a test bench.

Known Bugs

Several known bugs exist within the RISC core however none are critical as they can be easily avoided in software.

BUG1 Stall detection does not consider load/store instructions.

Due to pipelining techniques used by the processor and lack of address checking in the `EXME` and `MEWB` stages, `LW` instructions immediately after `SW` instructions:

```
SW R0 (R2+16)
LW R1 (R2+16)
```

will not return the previously stored value. In addition, because of the target address is calculated by the ALU (e.g. `R2+16`), detecting matching addresses at IFID and IDEX stage is not trivial, and because of this, a hardware fix is not planned for the final version. It is possible to overcome this problem in software by placing at least 5 `NOP` instructions after each `SW`.

Chapter 5

Future Progress

This chapter discusses planned future work

5.1 Project Status

Four months have passed since the start of the project.

5.1.1 Updated Project Time Line

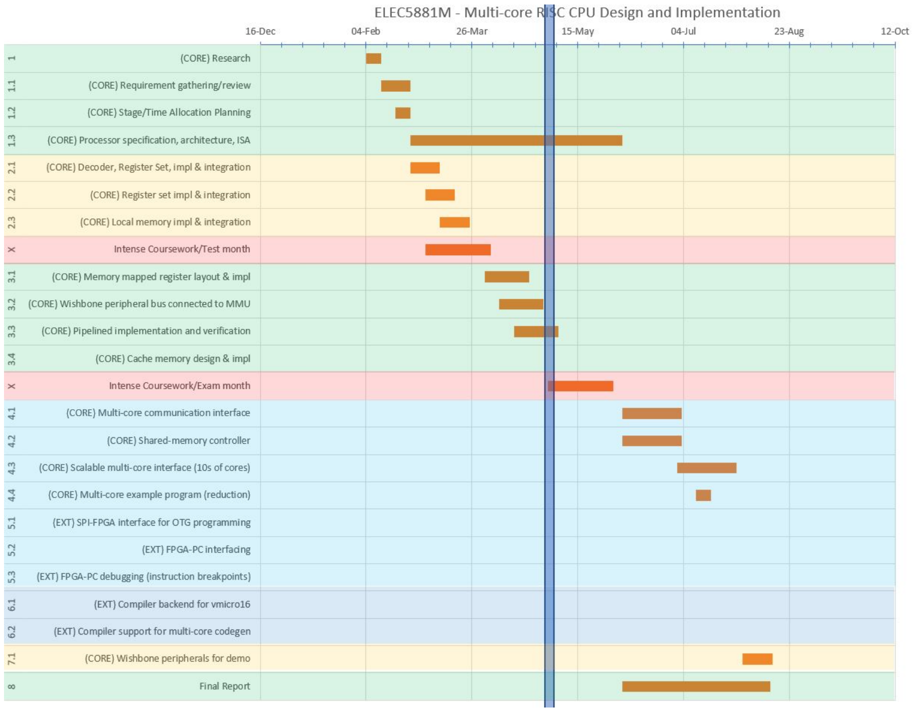

Figure 5.1: Updated project time gantt chart showing time allocations for stage 4.

[6] [7] [8]

Stage	Title	Start Date	Core	Status
1.0	Research	Feb 04	x	Completed
1.1	Requirement gathering/review	Feb 11	x	Completed
1.1	Processor specification, architecture, ISA	Feb 18	x	Completed
1.2	Stage/Time Allocation Planning	Feb 25	x	Completed
2.1	Decoder, Register Set, impl & integration	Feb 25	x	Completed
2.2	Register set impl & integration	Mar 04	x	Completed
2.3	Local memory impl & integration	Mar 11	x	Completed
3.1	Memory mapped register layout & impl	Apr 01		On-going
3.2	Wishbone peripheral bus connected to MMU	Apr 081		On-going
3.3	Pipelined implementation and verification	Apr 15		On-going
3.4	Cache memory design & impl	Apr 22		Not planned
4.1	Multi-core communication interface	TBD	x	Planned
4.2	Shared-memory controller	TBD	x	Planned
4.3	Scalable multi-core interface (10s of cores)	TBD	x	Planned
4.4	Multi-core example program (reduction)	TBD	x	Planned
5.1	SPI-FPGA interface for OTG programming	TBD		Unknown
5.2	FPGA-PC interfacing	TBD		Unknown
5.3	FPGA-PC debugging (instruction breakpoints)	TBD		Unknown
6.1	Compiler backend for vmicro16	TBD		Unknown
6.2	Compiler support for multi-core codegen	TBD		Unknown
7.1	Wishbone peripherals for demo	TBD	x	Planned
8.1	Final Report	TBD	x	Planned

Table 5.1: Project stages throughout the life cycle of the project.

References

[1] V. Subramanian, "Multiple gate field-effect transistors for future CMOS technologies," *IETE Technical review*, vol. 27, no. 6, pp. 446–454, 2010.

[2] M. J. Flynn, *Computer architecture: Pipelined and parallel processor design*. Jones & Bartlett Learning, 1995.

[3] T. Technologies, "Soc platform - cyclone - de1-soc board." [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

[4] *MiniSpartan6+*, Scarab Hardware, 2014. [Online]. Available: https://www.scarabhardware.com/minispartan6/

[5] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[6] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "Openpiton: An open source manycore research framework," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2. ACM, 2016, pp. 217–232.

[7] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–10.

[8] S. Binet, P. Calafiura, S. Snyder, W. Wiedenmann, and F. Winklmeier, "Harnessing multicores: Strategies and implementations in atlas," in *Journal of Physics: Conference Series*, vol. 219, no. 4. IOP Publishing, 2010, p. 042002.