

Multi-core RISC Processor Design and Implementation (Rev. 2.02)

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

May 1, 2019

Word count: 4689

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 multiprocessor principles and functionality to it.

Current multiprocessor 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 software programs.

Future tasks have been planned and will focus on the second stage of the project. Work will start on designing a loosely coupled multiprocessor communication interface and bringing them to the single-core processor.

Revision History

Date	Version	Changes
10/04/2019	2.02	Update future stages.
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.

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: May 1, 2019

Table of Contents

1	Introduction	4
1.1	Why Multi-core?	4
1.2	Why RISC?	5
1.3	Why FPGA?	5
2	Background	6
2.1	Amdahl's Law and Parallelism	6
2.2	Loosely and Tightly Coupled Processors	6
2.3	Network-on-chip Architectures	7
3	Project Overview	9
3.1	Project Deliverables	9
3.1.1	Core Deliverables (CD)	9
3.1.2	Extended Deliverables (ED)	10
3.2	Project Timeline	11
3.2.1	Project Stages	11
3.2.2	Project Stage Detail	12
3.2.3	Timeline	13
3.3	Resources	13
3.3.1	Hardware Resources	13
3.3.2	Software Resources	14
3.4	Legal and Ethical Considerations	15
4	Current Progress	16
4.1	RISC Core	16
4.1.1	Instruction Set Architecture	16
4.1.2	Design and Implementation	19
4.1.3	Verification	23
5	Future Work	25
5.1	Project Status	25
5.1.1	Updated Project Time Line	25
5.1.2	Future Work	25
6	Conclusion	27
	References	28
	Appendix A - Code Listing	29

Chapter 1

Introduction

1.1 Why Multi-core?	4
1.2 Why RISC?	5
1.3 Why FPGA?	5

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.

My previous experience with FPGAs in previous projects will reduce risk and learning times and allow for more time to be spent on adding and extending features (discusses further in section 3.1).

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 resource limitation will not be problem as the project aims to produce a small and minimal design specifically for learning about multi-core architectures.

Chapter 2

Background

2.1 Amdahl's Law and Parallelism	6
2.2 Loosely and Tightly Coupled Processors	6
2.3 Network-on-chip Architectures	7

2.1 Amdahl's Law and Parallelism

In many applications, not restricted to software, there may exist many opportunities for processes or algorithms to be performed in parallel. These algorithms can be split into two parts: a serial part that cannot be parallelised, and a part that can be parallelised. Amdahl's Law defines a formula for calculating the maximum *speedup* of a process with potential parallelism opportunities when ran in parallel with n many processors. Speedup is a term used to describe the potential performance improvements of an algorithm using an enhanced resource (in this case, adding parallel processors) compared to the original algorithm. Amdahl's Law is defined below, where the potential speedup S_p is dependant on the portion of program that can be parallelised p and the number of processing cores n :

$$S_p = \frac{1}{(1 - p) + \frac{p}{n}} \quad (2.1)$$

This formula will be used throughout the project to gauge the performance of the multi-core design running various software algorithms.

2.2 Loosely and Tightly Coupled Processors

Multiprocessor systems can be generalised into two architectures: loosely and tightly coupled, and each architecture has advantages and disadvantages. In loosely coupled systems, each processing node is self-contained – each node has its own dedicated memory and IO modules. Communication between nodes is performed over a *Message Transfer System (MTS)* [3] in a master-slave control architecture.

Scalability in loosely coupled systems is generally easier to implement as each node can simply be appended to the shared MTS interface without large modifications to the rest of the system. Scalability is an important concern in this project as I wish to test the developed solution with a range of processing nodes.

As loosely coupled system's nodes feature their own memory and IO modules, they generally perform better in cases where interaction between nodes is not prominent – each

node can store a separate part of the software program in its memory module allowing simultaneous executing of the program.

In scenarios where inter-node communication is prominent however, access to the MTS interface must be scheduled to avoid access conflicts which introduces delays and idle times in the software programs execution, resulting in lower throughput. Figure 2.1 shows a general layout of a loosely coupled multiprocessor system.

Tightly coupled systems feature processing nodes that do not have their own dedicated memory or IO modules – each node is directly connected to a shared memory module using a dedicated port. In scenarios where inter-node communication is prominent, tightly coupled systems are generally better suited as nodes are directly connected to a shared memory and do not need to wait to use a shared bus.

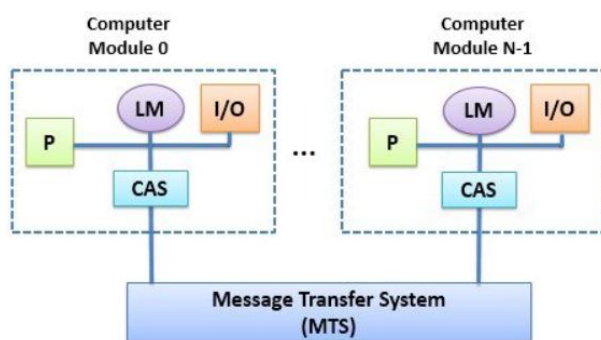


Figure 2.1: A loosely coupled multiprocessor system. Each node features its own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [3].

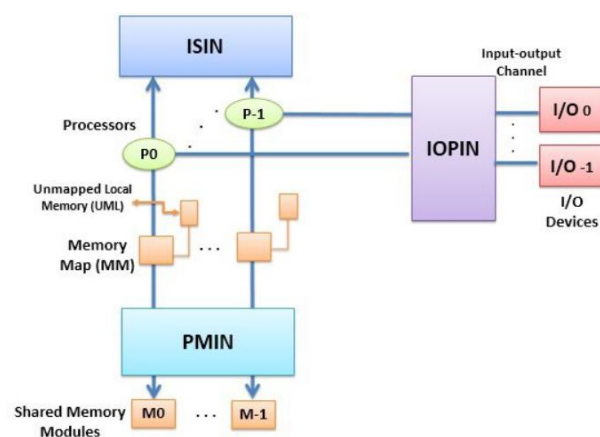


Figure 2.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [3].

This project will utilise a loosely coupled architecture due to its easier scalability implementation and my previous experience with the design of single-core processors. Although it will require a scheduler to access the MTS, the experience and knowledge gained from this task will be greatly beneficial for future projects.

2.3 Network-on-chip Architectures

Network-on-chip (NoC) architectures implement on-chip communication mechanisms that are based on network communication principles, such as routing, switching, and massive scalability [4]. NoC's can generally support hundreds to millions of processing cores. Figure 2.3 shows an example 16-core network-on-chip architecture. NoC's can scale to very large sizes while not sacrificing performance because each processor core is able to drive the network rather than needing to wait for a shared bus to become free before doing so.

The greater the number of cores in a network-on-chip design, the greater quality of service (QoS) problems arise. As such, network-on-chip architectures suffer the same problems as networks, such as fairness and throughput [5].

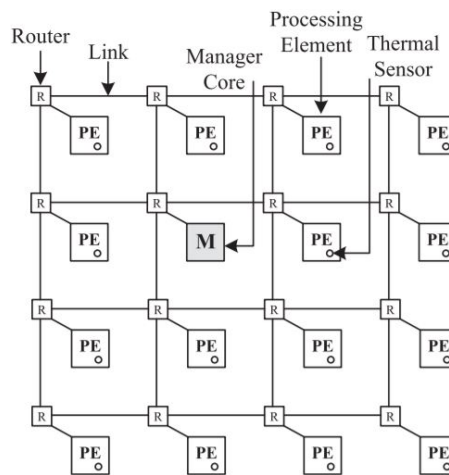


Figure 2.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [6].

Chapter 3

Project Overview

3.1	Project Deliverables	9
3.1.1	Core Deliverables (CD)	9
3.1.2	Extended Deliverables (ED)	10
3.2	Project Timeline	11
3.2.1	Project Stages	11
3.2.2	Project Stage Detail	12
3.2.3	Timeline	13
3.3	Resources	13
3.3.1	Hardware Resources	13
3.3.2	Software Resources	14
3.4	Legal and Ethical Considerations	15

This chapter discusses the the project’s requirements, goals, 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.

The instruction set will be the primary interface to control the processor from software. An instruction set will be required to implement the custom multi-core communication interface.

It was decided to design a new instruction set rather than to extend an existing architecture as this will increase my knowledge of the constraints to consider when designing instruction sets and processors.

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

The Verilog RISC core will be able to run software program written for the instruction set architecture.

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

The interconnect will be a chief requirement to enable multi-core communication. The interconnect should support up to 32 cores, however FPGA implementation constraints may limit this due to limited resources.

The interconnect will control communication between the cores to enable software parallelism.

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

To evaluate the effectiveness of the developed solution, a serial and parallel implementation of a simple computing algorithm (parallel reduction, sorting) will be ran on the processor and it's performance analysed. Effectiveness will be rated on total algorithm run-time and the speed-up gained by adding more cores.

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

The developed solution should be generic and portable to allow it to be used across a wide-range of FPGA vendors and devices.

Verilog is a generic implementation-independent hardware-description language and so designing implementation specific modules is recommended.

A key consideration for this requirement is to consider the varying hard IP provided by the FPGA vendors (such as BRAM, ethernet, and PCIe [7, 8]). To overcome this problem, the developed Verilog code will conditionally compile where vendor specific requirements are present.

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 [9] 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. Initial 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.

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.

3.2.2 Project Stage Detail

Stages 1.0 through 1.2 – Research and Project Conception

These stages cover initial research of existing problems and solutions in the multiprocessor area. The instruction set architecture is also proposed that later stages will implement.

Stages 2.1 through 2.3 – Processor module Design, Implementation, and Integration

These stages cover the design, implementation, and integration of key processor core modules such as the instruction decoder, register sets and local memory. Integration of all the modules is a challenging task because some modules have both asynchronous and synchronous signals that need to be timed correctly in order for other modules to receive valid data. An example of this is the register set which has asynchronous read ports that are later clocked in the instruction decode stage.

Stages 3.1 through 3.4 – Advanced Processor Implementation

These stages add advanced features to the processor to provide a more functional product. Although these stages are classified as extended, their technical requirement to design and implement is not great and so are have time allocations in the project schedule. The extended features that these stages introduce are: pipelined processor stages – to drastically increase processor performance; provide a memory-mapped peripheral interface through the MMU; provide a Wishbone master interface to the MMU – allowing external peripherals such as GPIO and LCD displays to be utilised in a modular fashion; and to implement a cache memory for each processor core.

Stages 4.1 through 4.4 – Multiprocessor Functionality

These stages are dedicated to adding multiprocessor functionality using a loosely coupled architecture to the processor.

Stages 5.1 through 5.3 – Debugging Features

These stages cover debugging features and are classified as extended due to the large development time required to implement them as well as not being related to multiprocessor systems.

Stages 6.1 through 6.2 – Compiler Backends

These stages cover the implementation of a compiler backend to ease software writing and programming of the processor.

Stage 7.1 – Wishbone Peripherals

Additional Wishbone peripherals, such as SPI and timers will be added to produce a more useful multiprocessor system.

Stage 8.1 – Final Report

This stage is dedicated to the final report write-up. It is expected to be an iterative task that is active throughout the lifespan of the project.

3.2.3 Timeline

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

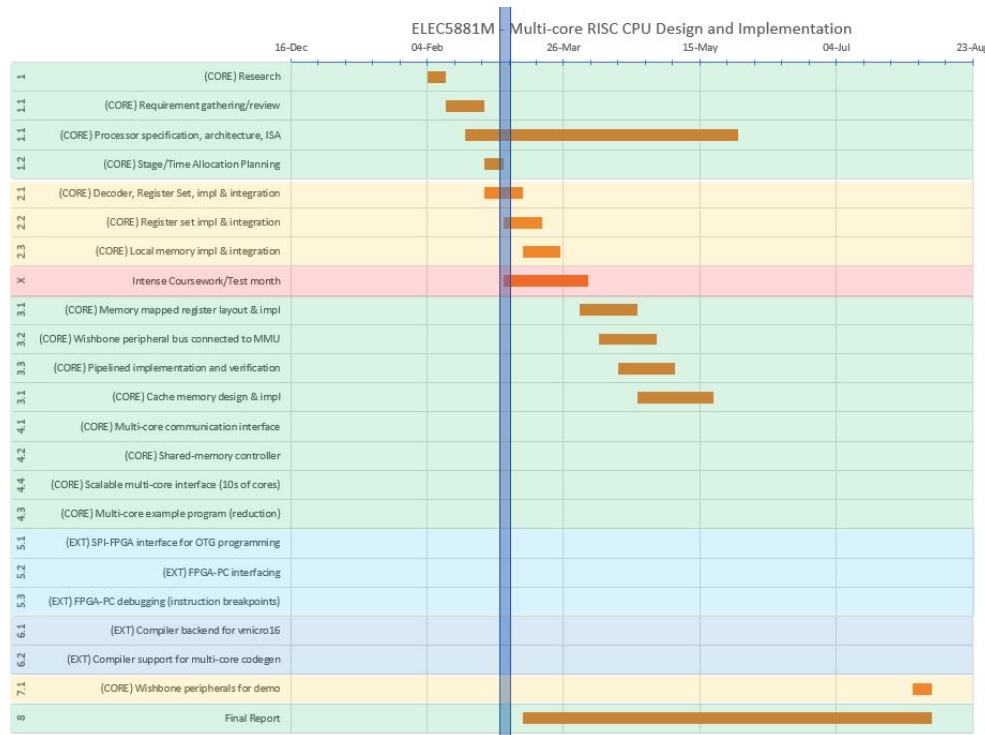


Figure 3.1: Project stages in a Gantt chart.

3.3 Resources

This section describes the hardware and software resources required to fulfil 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 FPGA.

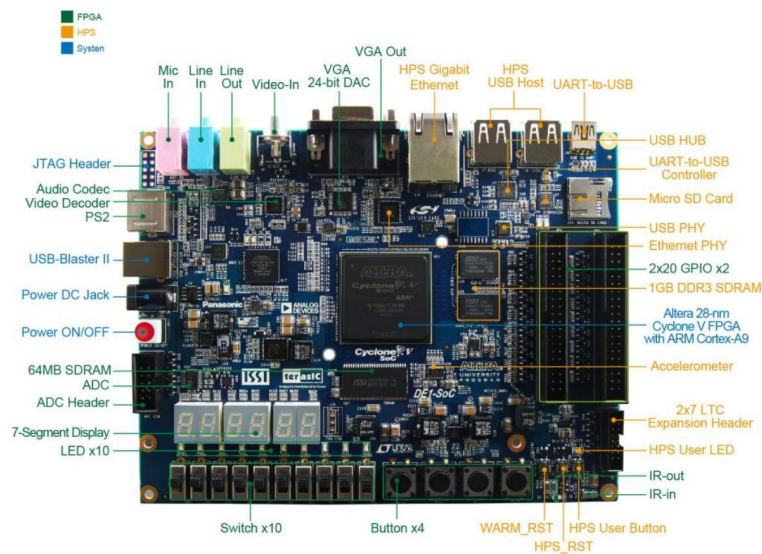


Figure 3.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [10].

Minispartan 6+ FPGA Development Board

The Minispartan 6+ is a hobbyist FPGA development board with fewer peripherals than the DE1-SoC. The board features a Xilinx Spartan 6 XC6LX9 which has far fewer resources than the DE1-SoC's Cyclone V however it's simplicity and my familiarity with Xilinx's software suite will speed up development. The development board is shown in Figure 3.3.

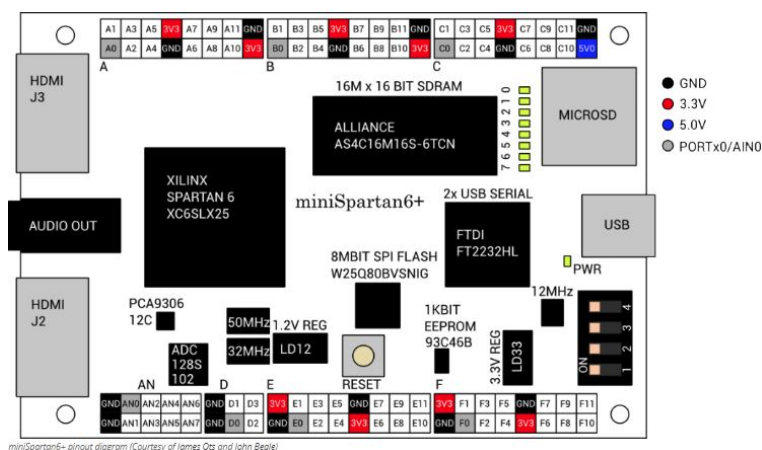


Figure 3.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [11].

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 Webpack 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 used 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 these instructions. This could result in security vulnerabilities similar to Intel's Spectre vulnerability [12].

Chapter 4

Current Progress

4.1	RISC Core	16
4.1.1	Instruction Set Architecture	16
4.1.2	Design and Implementation	19
4.1.3	Verification	23

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. The core has been nicknamed *Vmicro16* – short for Verilog microprocessor 16-bit.

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 at least 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 register selects is that instructions are always destructive – they always *destroy* existing data in the destination register (e.g. $R0 = \text{ADD } R0 \ R1$) unlike constructive instructions that provide a unique register select for the destination (e.g. $R2 = \text{ADD } R0 \ R1$).

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 to be performed by the ALU.

As shown in Figure 4.1, frequently used commands such as incrementing/decrementing and logical comparisons 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 instruction 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 comparison'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=0
BR_GE	01000	Rd		0000 0100	S=0
BR_L	01000	Rd		0000 0101	S != 0
BR_LE	01000	Rd		0000 0110	Z=1 or (S != 0)
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.

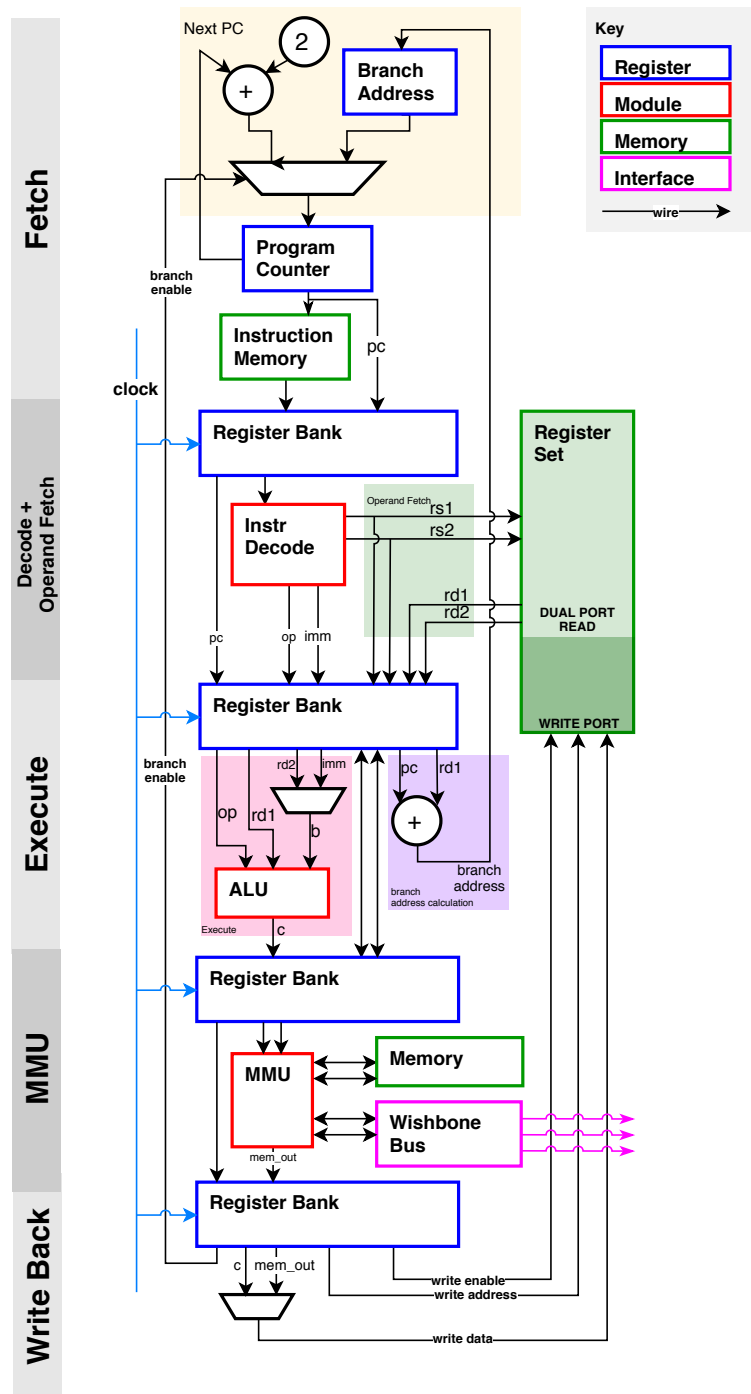


Figure 4.2: Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

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-

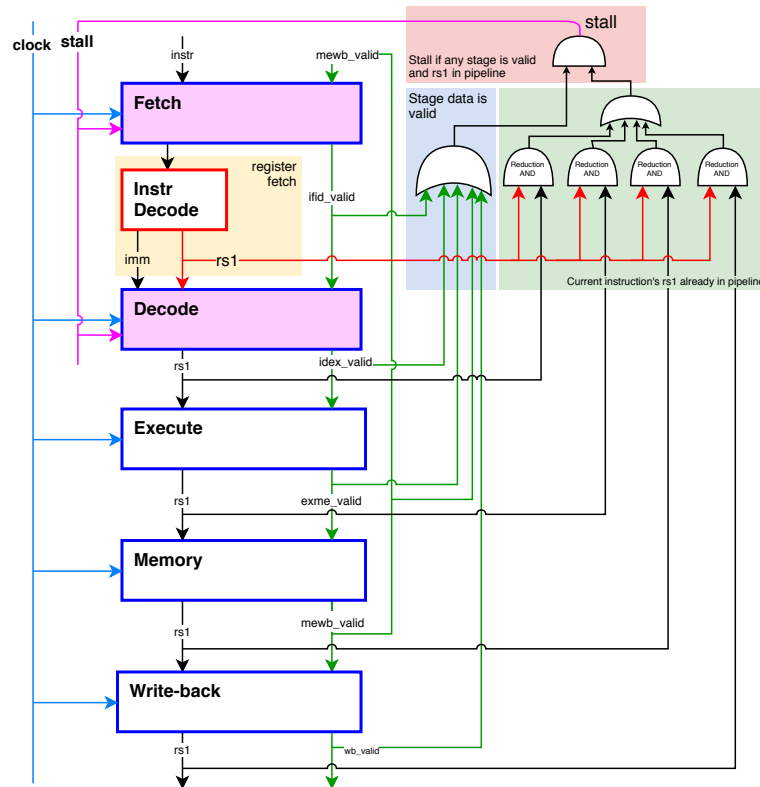


Figure 4.3: Pipeline data hazard detection. The register selects are passed forward through each stage and compared to the IDEX (latest instruction) register selects. If they match, the latest instruction depends on the output of an instruction in the pipeline, the IFID and IDEX stages are stalled to allow the instruction in the pipeline to commit.

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

Proposed Memory Mapped Addresses

The peripheral addresses are currently based on classes. For example, a memory-mapped address may use the upper byte to address a peripheral and the lower byte to address a register/function in that peripheral.

Later in the project, I plan to rewrite the addressing scheme to use a simpler address format which is closer to commonly used peripheral addressing schemes used today. 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: Provisional memory-mapped addresses table.

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 operate 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.

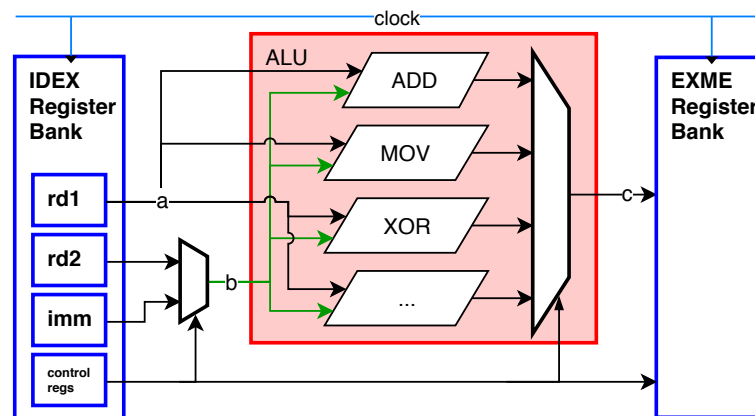


Figure 4.4: Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

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.

```

322     always @(*) case (op)
323         // branch/nop, output nothing
324         `VMICRO16_ALU_BR,
325         `VMICRO16_ALU_NOP:      c = 0;
326         // load/store addresses (use value in rd2)
327         `VMICRO16_ALU_LW,
328         `VMICRO16_ALU_SW:      c = b;
329         // bitwise operations
330         `VMICRO16_ALU_BIT_OR:   c = a | b;
331         `VMICRO16_ALU_BIT_XOR:  c = a ^ b;
332         `VMICRO16_ALU_BIT_AND:  c = a & b;
333         `VMICRO16_ALU_BIT_NOT:  c = ~(b);
334         `VMICRO16_ALU_BIT_LSHFT: c = a << b;
335         `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.

```

224     always @(*) case (opcode)
225         `VMICR016_OP_HALT, // TODO: stop ifid
226         `VMICR016_OP_NOP:      alu_op = `VMICR016_ALU_NOP;
227
228         `VMICR016_OP_LW:      alu_op = `VMICR016_ALU_LW;
229         `VMICR016_OP_SW:      alu_op = `VMICR016_ALU_SW;
230
231         `VMICR016_OP_MOV:      alu_op = `VMICR016_ALU_MOV;
232         `VMICR016_OP_MOVI:     alu_op = `VMICR016_ALU_MOVI;
233         `VMICR016_OP_MOVI_L:   alu_op = `VMICR016_ALU_MOVI_L;
234
235         `VMICR016_OP_BR:      alu_op = `VMICR016_ALU_BR;
236
237         `VMICR016_OP_BIT:      casez (simm5)
238             `VMICR016_OP_BIT_OR:    alu_op = `VMICR016_ALU_BIT_OR;
239             `VMICR016_OP_BIT_XOR:   alu_op = `VMICR016_ALU_BIT_XOR;
240             `VMICR016_OP_BIT_AND:   alu_op = `VMICR016_ALU_BIT_AND;
241             `VMICR016_OP_BIT_NOT:   alu_op = `VMICR016_ALU_BIT_NOT;
242             `VMICR016_OP_BIT_LSHFT: alu_op = `VMICR016_ALU_BIT_LSHFT;
243             `VMICR016_OP_BIT_RSHFT: alu_op = `VMICR016_ALU_BIT_RSHFT;
244             default:                alu_op = `VMICR016_ALU_BAD; endcase
245

```

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 VMICR016_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. For now, it is easier and faster to spot erroneous states by hand due to the large complexity of the pipeline. Later in the project, automatic test benches will be utilised.

Known Bugs

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 instruction 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 Work

5.1	Project Status	25
5.1.1	Updated Project Time Line	25
5.1.2	Future Work	25

5.1 Project Status

Four months have passed since the start of the project and significant progress has been made to the final deliverable.

The current active stage is 3.3 *Pipeline Implementation and Verification* where the processor pipeline is being verified against a range of simple software sequences. It is important that this verification is thorough and the output is bug free as future additions to the processor will utilise this foundation.

5.1.1 Updated Project Time Line

The project table described in section 3.2 did not allocate times for stages 4.1 and later. This was due to expected high demand from other modules and exams in this time period and so it was decided to not allocate times that would later not be followed.

Now that this time period is closer, time allocations have been assigned for stages 4, 7, and 8. The state of stage 5's extended deliverables, to implement debugging interfaces, have changed from *Unknown* to *Cancelled* due to expected high workload from other modules in the next month. The cancellation of these stages will not severely affect the final functionality of the deliverable however it will make debugging the processor slightly more difficult. It was decided to remove these extended features to allow for more time to be spent on core functionality.

The updated project status is shown in Table 5.1 and in Figure 5.1.

5.1.2 Future Work

May and early June are reserved for work on other modules and preparation for exams. From mid-June, work will resume on verifying the end of stage 3 and then work will start on stage 4 (focussed on designing and implementing multiprocessor features). After stage 4, software algorithms will be compiled for the ISA and evaluated against Amdahl's Law.

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 08		On-going
3.3	Pipeline implementation and verification	Apr 15		On-going
3.4	Cache memory design & impl	Apr 22		Cancelled
4.1	Multi-core communication interface	Jun 05	x	Planned
4.2	Shared-memory controller	Jun 05	x	Planned
4.3	Scalable multi-core interface (10s of cores)	Jul 01	x	Planned
4.4	Multi-core example program (reduction)	Jul 10	x	Planned
5.1	SPI-FPGA interface for OTG programming	TBD		Cancelled
5.2	FPGA-PC interfacing	TBD		Cancelled
5.3	FPGA-PC debugging (instruction breakpoints)	TBD		Cancelled
6.1	Compiler backend for vmicro16	TBD		Unknown
6.2	Compiler support for multi-core codegen	TBD		Unknown
7.1	Wishbone peripherals for demo	Aug 01	x	Planned
8.1	Final Report	Jun 05	x	Planned

Table 5.1: Updated project stages.

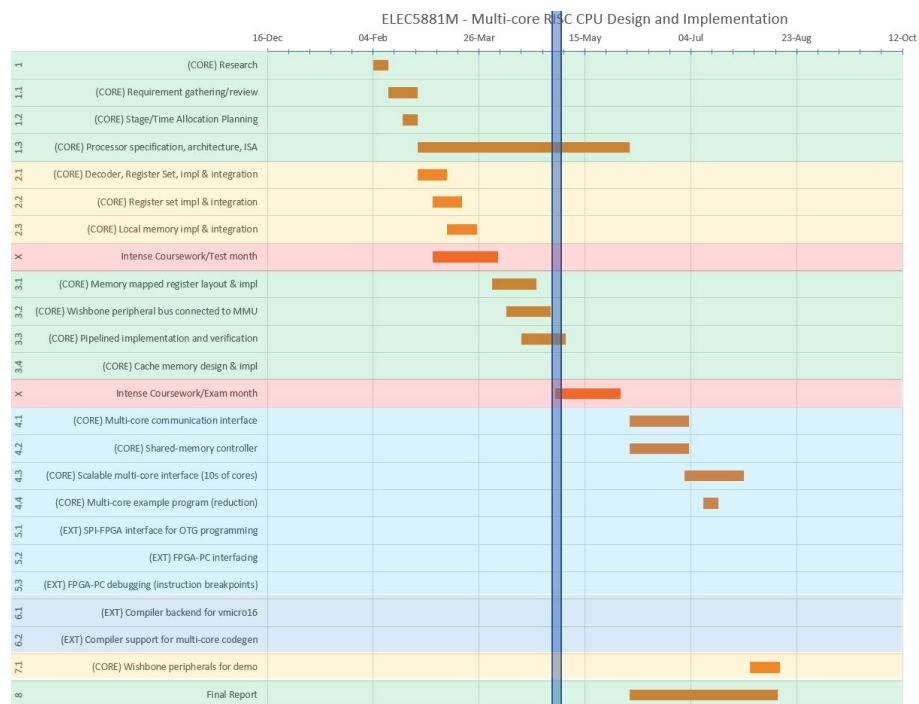


Figure 5.1: Updated project time gantt chart showing time allocations for stage 4.

Chapter 6

Conclusion

With the end of Moore's Law coming, processor designers must use other strategies to continue improving performance of processors – multiprocessor and parallelism being a primary strategy. This projects sets out to improve my knowledge on multiprocessor communication by designing, implementing, and verifying a multiprocessor – and I believe starting from scratch is the best way to accomplish this learning task.

To date, a compact 16-bit RISC instruction set has been designed and implemented in a Verilog single-core processor. Whilst single-core verification is still on-going, good progress has been made and extended deliverables from stage 3, such as instruction pipelining and memory-mapped peripherals via a Wishbone bus, has been implemented successfully.

Stage 5's extended deliverables and the cache memory have been cancelled but they do not effect the core functionality of the processor. The planned project time-line for future stages is realistic and accomplishing the projects goals appears achievable.

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] Tech Differences, "Difference between loosely coupled and tightly coupled multiprocessor system (with comparison chart)," Jul 2017. [Online]. Available: <https://techdifferences.com/difference-between-loosely-coupled-and-tightly-coupled-multiprocessor-system.html> (Accessed 2019-04-20).
- [4] L. Benini and G. De Micheli, "Networks on Chips: A new SoC paradigm," *Computer*, vol. 35, pp. 70–78, 02 2002.
- [5] D. Zhu, L. Chen, S. Yue, T. M. Pinkston, and M. Pedram, "Balancing On-Chip Network Latency in Multi-application Mapping for Chip-Multiprocessors," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 872–881.
- [6] N. Chatterjee, S. Paul, and S. Chattopadhyay, "Fault-tolerant dynamic task mapping and scheduling for network-on-chip-based multicore platform," *ACM Transactions on Embedded Computing Systems*, vol. 16, pp. 1–24, 05 2017.
- [7] Xilinx, *Spartan-6 FPGA Block RAM Resources*, Xilinx.
- [8] Altera, *Recommended HDL Coding Styles - QII51007-9.0.0*, Altera.
- [9] B. Lancaster, "FPGA-based RISC Microprocessor and Compiler," vol. 3.14, pp. 37–50. [Online]. Available: <https://github.com/bendl/prco304> (Accessed March 2018).
- [10] Terasic Technologies, "SoC Platform - Cyclone - DE1-SoC Board." [Online]. Available: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836> (Accessed 2019-04-20).
- [11] *MiniSpartan6+*, Scarab Hardware, 2014. [Online]. Available: <https://www.scarabhardware.com/minispartan6/> (Accessed 2019-04-20).
- [12] 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.

Appendix A - Code Listing

vmicro16.v

The single core RISC processor is defined in this file. It contains many submodules such as the decoder and local memory.

```
1 // This file contains multiple modules.
2 // Verilator likes 1 file for each module
3 /* verilator lint_off DECLFILENAME */
4 /* verilator lint_off UNUSED */
5 /* verilator lint_off BLKSEQ */
6 /* verilator lint_off WIDTH */
7
8 // Include Vmicro16 ISA containing definitions for the bits
9 `include "vmicro16_isa.v"
10
11 // This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
12 // https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
13 // https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
14 // https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf
15 module vmicro16_bram # (
16     parameter MEM_WIDTH    = 16,
17     parameter MEM_DEPTH    = 256
18 ) (
19     input clk, input reset,
20
21     input [MEM_WIDTH-1:0] mem_addr,
22     input [MEM_WIDTH-1:0] mem_in,
23     input mem_we,
24     output reg [MEM_WIDTH-1:0] mem_out
25 );
26 // memory vector
27 reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];
28
29 // not synthesizable
30 integer i;
31 initial for (i = 0; i < MEM_DEPTH; i = i + 1) mem[i] <= 0;
32
33 always @(posedge clk) begin
34     // synchronous WRITE_FIRST (page 13)
35     if (mem_we) begin
36         mem[mem_addr] <= mem_in;
37         $display($time, "\tMEM: W mem[%h] <= %h", mem_addr, mem_in);
38     end else begin
39         mem_out <= mem[mem_addr];
40     end
41 end
42
43 // TODO: Reset impl = every clock while reset is asserted, clear each cell
44 // one at a time, mem[i++] <= 0
45 endmodule
46
47 // Wishbone wrapper around the register file to use as a peripheral
48 module vmicro16_regs_wb # (
49     parameter CELL_WIDTH    = 16,
50     parameter CELL_DEPTH    = 8,
51     parameter CELL_SEL_BITS = 3,
52     parameter CELL_DEFAULTS = 0,
53     parameter DEBUG_NAME    = "",
54     parameter PIPELINE_READ = 0
55 ) (
56     input clk, input reset,
57
58     // wishbone slave interface
59     input wb_stb_i,
60     input wb_cyc_i,
61     input wb_we_i,
62     input [CELL_WIDTH-1:0] wb_addr_i,
63     input [CELL_WIDTH-1:0] wb_data_i,
64     output [CELL_WIDTH-1:0] wb_data_o,
65     output wb_ack_o,
66     output wb_stall_o,
67     output wb_err_o
68 );
69 // embedded register data
70 wire [CELL_SEL_BITS-1:0] reg_rs1;
71 wire [CELL_WIDTH-1:0] reg_rd1;
```

```

72     wire                reg_we;
73     wire [CELL_WIDTH-1:0] reg_wd;
74
75     reg selected;
76     always @(*) begin
77         if (wb_stb_i)
78             selected = 1'b1;
79         else if (selected && wb_cyc_i)
80             selected = 1'b1;
81         else if (selected && !wb_cyc_i)
82             selected = 1'b0;
83         else
84             selected = 1'b0;
85     end
86
87     assign reg_we = wb_we_i && wb_stb_i;
88     assign reg_rs1 = wb_addr_i[CELL_SEL_BITS-1:0];
89     assign reg_wd = wb_data_i;
90
91     // Only stall on write requests
92     //assign wb_stall_o = wb_cyc_i && wb_we_i;
93     // TODO: Stall for 1 clock if pipelined
94     assign wb_stall_o = PIPELINE_READ ? wb_stb_i : 1'b0;
95     // Only use the output bus when we're selected and it's a read req
96     assign wb_data_o = (selected && !wb_we_i) ? reg_rd1 : {(CELL_WIDTH-1){1'bZ}};
97     // TODO: ack on same stb clock allowed?
98     assign wb_ack_o = (wb_cyc_i && !wb_stall_o) ? 1'b1 : 1'bZ;
99
100    vmicro16_regs # (
101        .CELL_WIDTH      (CELL_WIDTH),
102        .CELL_DEPTH      (CELL_DEPTH),
103        .CELL_SEL_BITS   (CELL_SEL_BITS),
104        .CELL_DEFAULTS   (CELL_DEFAULTS),
105        .DEBUG_NAME      (DEBUG_NAME),
106        .PIPELINE_READ   (PIPELINE_READ)
107    ) vmicro16_regs_wb_i (
108        .clk              (clk)
109        ,.reset           (reset)
110        // Read port 1 (INDEX)
111        ,.rs1             (reg_rs1)
112        ,.rd1             (reg_rd1)
113        // Read port 2 (INDEX) unused
114        ,.rs2             (dec_rs2)
115        ,.rd2             (reg_rd2)
116        // Write port (MEWB)
117        ,.we              (reg_we)
118        ,.ws1             (reg_rs1)
119        ,.wd              (reg_wd)
120    );
121
122    module vmicro16_regs # (
123        parameter CELL_WIDTH      = 16,
124        parameter CELL_DEPTH      = 8,
125        parameter CELL_SEL_BITS   = 3,
126        parameter CELL_DEFAULTS   = 0,
127        parameter DEBUG_NAME      = "",
128        parameter PIPELINE_READ   = 0
129    ) (
130        input clk, input reset,
131        // ID/EX stage reg reads
132        // Dual port register reads
133        input [CELL_SEL_BITS-1:0] rs1, // port 1
134        output reg [CELL_WIDTH-1:0] rd1,
135        input [CELL_SEL_BITS-1:0] rs2, // port 2
136        output reg [CELL_WIDTH-1:0] rd2,
137        // EX/WB final stage write back
138        input we,
139        input [CELL_SEL_BITS-1:0] ws1,
140        input [CELL_WIDTH-1:0] wd
141    );
142
143    reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;
144
145    // Initialise registers with default values
146    // Really only used for special registers used by the soc
147    // TODO: How to do this on reset?
148    initial if (CELL_DEFAULTS) $readmemh(CELL_DEFAULTS, regs);
149
150    integer i;
151    always @(posedge clk)
152    begin
153        if (reset)
154            if (CELL_DEFAULTS) $readmemh(CELL_DEFAULTS, regs); // TODO:
155            else for(i = 0; i < CELL_DEPTH; i = i + 1) regs[i] <= {(CELL_WIDTH-1){1'b0}};
156        else if (we) begin
157            $display($time, "\tREGS #s: Writing %h to reg[%d]", DEBUG_NAME, wd, ws1);
158            $display($time, "\t\t\t\t\t %h %h %h %h | %h %h %h %h |",
159                regs[0], regs[1], regs[2], regs[3], regs[4], regs[5], regs[6], regs[7]);
160            // Perform the right
161            regs[ws1] <= wd;
162        end
163    end
164
165    generate if (PIPELINE_READ)
166    always @(posedge clk)
167    begin
168        if (reset) begin
169            rd1 <= {(CELL_WIDTH-1){1'b0}};
170            rd2 <= {(CELL_WIDTH-1){1'b0}};
171        end else begin
172            rd1 <= regs[rs1];
173            rd2 <= regs[rs2];
174        end
175    end
176
177 end

```



```

171     else
172         always @(*) begin
173             rd1 = regs[rs1];
174             rd2 = regs[rs2];
175         end
176     endgenerate
177 endmodule
178
179 // Decoder is hard to parameterise as it's very closely linked to the ISA.
180 module vmicro16_dec # (
181     parameter INSTR_WIDTH    = 16,
182     parameter INSTR_OP_WIDTH = 5,
183     parameter INSTR_RS_WIDTH = 3,
184     parameter ALU_OP_WIDTH   = 5
185 ) (
186     //input clk, // not used yet (all combinational)
187     //input reset, // not used yet (all combinational)
188
189     input  [INSTR_WIDTH-1:0]  instr,
190
191     output [INSTR_OP_WIDTH-1:0] opcode,
192     output [INSTR_RS_WIDTH-1:0] rd,
193     output [INSTR_RS_WIDTH-1:0] ra,
194     output [7:0]               imm8,
195     output [11:0]              imm12,
196     output [4:0]               simm5,
197
198     // This can be freely increased without affecting the isa
199     output reg [ALU_OP_WIDTH-1:0] alu_op,
200
201     output reg has_imm8,
202     output reg has_imm12,
203     output reg has_we,
204     output reg has_br,
205     output reg has_mem,
206     output reg has_mem_we,
207
208     output halt
209
210     // TODO: Use to identify bad instruction and
211     //       raise exceptions
212     //, output is_bad
213 );
214
215 assign opcode = instr[15:11];
216 assign rd     = instr[10:8];
217 assign ra     = instr[7:5];
218 assign imm8   = instr[7:0];
219 assign imm12  = instr[11:0];
220 assign simm5  = instr[4:0];
221 // Special opcodes
222 assign halt   = (opcode == `VMICRO16_OP_HALT);
223
224 // exme_op
225 always @(*) case (opcode)
226     `VMICRO16_OP_HALT, // TODO: stop ifid
227     `VMICRO16_OP_NOP:      alu_op = `VMICRO16_ALU_NOP;
228
229     `VMICRO16_OP_LW:      alu_op = `VMICRO16_ALU_LW;
230     `VMICRO16_OP_SW:      alu_op = `VMICRO16_ALU_SW;
231
232     `VMICRO16_OP_MOV:     alu_op = `VMICRO16_ALU_MOV;
233     `VMICRO16_OP_MOVI:    alu_op = `VMICRO16_ALU_MOVI;
234     `VMICRO16_OP_MOVI_L:  alu_op = `VMICRO16_ALU_MOVI_L;
235
236     `VMICRO16_OP_BR:      alu_op = `VMICRO16_ALU_BR;
237
238     `VMICRO16_OP_BIT:      casez (simm5)
239         `VMICRO16_OP_BIT_OR:    alu_op = `VMICRO16_ALU_BIT_OR;
240         `VMICRO16_OP_BIT_XOR:   alu_op = `VMICRO16_ALU_BIT_XOR;
241         `VMICRO16_OP_BIT_AND:   alu_op = `VMICRO16_ALU_BIT_AND;
242         `VMICRO16_OP_BIT_NOT:   alu_op = `VMICRO16_ALU_BIT_NOT;
243         `VMICRO16_OP_BIT_LSHFT: alu_op = `VMICRO16_ALU_BIT_LSHFT;
244         `VMICRO16_OP_BIT_RSHFT: alu_op = `VMICRO16_ALU_BIT_RSHFT;
245         default:                alu_op = `VMICRO16_ALU_BAD; endcase
246
247     `VMICRO16_OP_ARITH_U:    casez (simm5)
248         `VMICRO16_OP_ARITH_UADD: alu_op = `VMICRO16_ALU_ARITH_UADD;
249         `VMICRO16_OP_ARITH_USUB: alu_op = `VMICRO16_ALU_ARITH_USUB;
250         `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
251         default:                alu_op = `VMICRO16_ALU_BAD; endcase
252
253     `VMICRO16_OP_ARITH_S:    casez (simm5)
254         `VMICRO16_OP_ARITH_SADD: alu_op = `VMICRO16_ALU_ARITH_SADD;
255         `VMICRO16_OP_ARITH_SSUB: alu_op = `VMICRO16_ALU_ARITH_SSUB;
256         `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
257         default:                alu_op = `VMICRO16_ALU_BAD; endcase
258
259     default: begin
260         alu_op = `VMICRO16_ALU_BAD;
261         $display($time, "\tDEC: unknown opcode: %h", opcode);
262     end
263 endcase
264
265 // Register writes
266 always @(*) case (opcode)
267     `VMICRO16_OP_LW,
268     `VMICRO16_OP_MOV,
269     `VMICRO16_OP_MOVI,
270     `VMICRO16_OP_MOVI_L,

```

```

270         `VMICRO16_OP_ARITH_U,
271         `VMICRO16_OP_ARITH_S,
272         `VMICRO16_OP_CMP,
273         `VMICRO16_OP_SETC:      has_we = 1'b1;
274         default:                has_we = 1'b0;
275     endcase
276
277     // Contains 8-bit immediate
278     always @(*) case (opcode)
279         `VMICRO16_OP_MOVI,
280         `VMICRO16_OP_CMP:      has_imm8 = 1'b1;
281         default:                has_imm8 = 1'b0;
282     endcase
283
284     // Contains 12-bit immediate
285     always @(*) case (opcode)
286         `VMICRO16_OP_MOVI_L:   has_imm12 = 1'b1;
287         default:                has_imm12 = 1'b0;
288     endcase
289
290     // Will branch the pc
291     always @(*) case (opcode)
292         `VMICRO16_OP_BR:       has_br = 1'b1;
293         default:                has_br = 1'b0;
294     endcase
295
296     // Requires external memory
297     always @(*) case (opcode)
298         `VMICRO16_OP_LW,
299         `VMICRO16_OP_SW:       has_mem = 1'b1;
300         default:                has_mem = 1'b0;
301     endcase
302
303     // Requires external memory write
304     always @(*) case (opcode)
305         `VMICRO16_OP_SW:       has_mem_we = 1'b1;
306         default:                has_mem_we = 1'b0;
307     endcase
308 endmodule
309
310
311 module vmicro16_alu # (
312     parameter OP_WIDTH = 5,
313     parameter DATA_WIDTH = 16
314 ) (
315     // input clk, // TODO: make clocked
316
317     input [OP_WIDTH-1:0] op,
318     input [DATA_WIDTH-1:0] a, // rs1/dst
319     input [DATA_WIDTH-1:0] b, // rs2
320     output reg [DATA_WIDTH-1:0] c
321 );
322     always @(*) case (op)
323         // branch/nop, output nothing
324         `VMICRO16_ALU_BR,
325         `VMICRO16_ALU_NOP:      c = 0;
326         // load/store addresses (use value in rd2)
327         `VMICRO16_ALU_LW,
328         `VMICRO16_ALU_SW:      c = b;
329         // bitwise operations
330         `VMICRO16_ALU_BIT_OR:   c = a | b;
331         `VMICRO16_ALU_BIT_XOR:  c = a ^ b;
332         `VMICRO16_ALU_BIT_AND:  c = a & b;
333         `VMICRO16_ALU_BIT_NOT:  c = ~(b);
334         `VMICRO16_ALU_BIT_LSHFT: c = a << b;
335         `VMICRO16_ALU_BIT_RSHFT: c = a >> b;
336
337         `VMICRO16_ALU_MOV:      c = b;
338         `VMICRO16_ALU_MOVI:     c = b;
339         `VMICRO16_ALU_MOVI_L:   c = b;
340
341         `VMICRO16_ALU_ARITH_UADD: c = a + b;
342         `VMICRO16_ALU_ARITH_USUB: c = a - b;
343         // TODO: ALU should have simm5 as input
344         `VMICRO16_ALU_ARITH_UADDI: c = a + b;
345
346         `VMICRO16_ALU_ARITH_SADD: c = $signed(a) + $signed(b);
347         `VMICRO16_ALU_ARITH_SSUB: c = $signed(a) - $signed(b);
348         // TODO: ALU should have simm5 as input
349         `VMICRO16_ALU_ARITH_SSUBI: c = $signed(a) + $signed(b);
350
351         // TODO: Parameterise
352         default: begin
353             $display($time, "\tALU: unknown op: %h", op);
354             c = {(DATA_WIDTH-1){1'bZZZZ}};
355         end
356     endcase
357 endmodule
358
359 module vmicro16_ifid (
360     input clk,
361     input reset,
362     input stall,
363
364     input mewb_valid,
365     input jmping,
366
367     input [15:0] wb_jmp_target,

```

```

369
370     output reg      ifid_valid,
371     output reg [15:0] ifid_pc,
372     output reg [15:0] ifid_instr
373 );
374     reg [7:0] mem_cache [0:31];
375     integer i;
376     initial begin
377         $display($time, "\tResetting mem");
378         for(i = 0; i < 32; i = i + 1) mem_cache[i] = 8'h00;
379         mem_cache[32-1] = { 8'hAA };
380
381         /*
382         // Single cycle register writes
383         mem_cache[0] = {'VMICRO16_OP_MOVI, 3'h0}; mem_cache[1] = { 8'h00 };
384         mem_cache[2] = {'VMICRO16_OP_MOVI, 3'h1}; mem_cache[3] = { 8'h01 };
385         mem_cache[4] = {'VMICRO16_OP_MOVI, 3'h2}; mem_cache[5] = { 8'h02 };
386         mem_cache[6] = {'VMICRO16_OP_MOVI, 3'h3}; mem_cache[7] = { 8'h03 };
387         mem_cache[8] = {'VMICRO16_OP_MOVI, 3'h4}; mem_cache[9] = { 8'h04 };
388         mem_cache[10] = {'VMICRO16_OP_MOVI, 3'h5}; mem_cache[11] = { 8'h05 };
389         mem_cache[12] = {'VMICRO16_OP_MOVI, 3'h6}; mem_cache[13] = { 8'h06 };
390         mem_cache[14] = {'VMICRO16_OP_MOVI, 3'h7}; mem_cache[15] = { 8'h07 };
391         mem_cache[16] = {'VMICRO16_OP_HALT, 3'h0}; mem_cache[17] = { 8'h00 };
392         /**/
393
394         /*
395         mem_cache[0] = {'VMICRO16_OP_MOVI, 3'h0}; mem_cache[1] = { 8'h00 };
396         mem_cache[2] = {'VMICRO16_OP_MOVI, 3'h1}; mem_cache[3] = { -8'd02 };
397         mem_cache[4] = {'VMICRO16_OP_ARITH_U, 3'h0}; mem_cache[5] = {3'h7, 1'b0, 4'h1};
398         mem_cache[6] = {'VMICRO16_OP_ARITH_U, 3'h0}; mem_cache[7] = {3'h7, 1'b0, 4'h3};
399         mem_cache[8] = {'VMICRO16_OP_ARITH_U, 3'h0}; mem_cache[9] = {3'h7, 1'b0, 4'h5};
400         mem_cache[10] = {'VMICRO16_OP_BR, 3'h1}; mem_cache[11] = {'VMICRO16_OP_BR_U};
401         mem_cache[12] = {'VMICRO16_OP_NOP, 3'h0}; mem_cache[13] = {8'h0};
402         mem_cache[14] = {'VMICRO16_OP_NOP, 3'h0}; mem_cache[15] = {8'h0};
403         mem_cache[16] = {'VMICRO16_OP_NOP, 3'h0}; mem_cache[17] = {8'h0};
404         mem_cache[18] = {'VMICRO16_OP_NOP, 3'h0}; mem_cache[19] = {8'h0};
405         mem_cache[20] = {'VMICRO16_OP_HALT, 3'h0}; mem_cache[21] = {8'h00};
406         /**/
407
408         mem_cache[0] = {'VMICRO16_OP_MOVI, 3'h0}; mem_cache[1] = { 8'h7F };
409         mem_cache[2] = {'VMICRO16_OP_MOVI, 3'h1}; mem_cache[3] = { 8'h0A };
410         mem_cache[4] = {'VMICRO16_OP_SW, 3'h0}; mem_cache[5] = { 3'h01, 5'h03 };
411         mem_cache[6] = {'VMICRO16_OP_HALT, 3'h0}; mem_cache[7] = {8'h00};
412
413     end
414
415     reg [15:0] pc;
416     initial pc = 0;
417
418     always @(posedge clk) begin
419         if (reset) begin
420             ifid_valid <= 0;
421             ifid_instr <= 0;
422             ifid_pc <= 0;
423             pc <= 0;
424         end else begin
425             ifid_valid <= !jumping;
426             if (mewb_valid && jumping) begin
427                 $display($time, "\tJumping to %h", wb_jmp_target);
428                 pc <= wb_jmp_target;
429             end
430             else if (!stall) begin
431                 // TODO: vmicro16_mmu is single port
432                 // so we require a cache to do this
433                 ifid_instr <= {mem_cache[pc], mem_cache[pc+1]};
434                 ifid_pc <= pc; // Only for simulation
435                 pc <= pc + 16'h2;
436             end
437         end
438     end
439 endmodule
440
441 module vmicro16_idx (
442     input clk,
443     input reset,
444
445     input [15:0] ifid_pc, output reg [15:0] idx_pc,
446     input [15:0] ifid_instr, output reg [15:0] idx_instr,
447
448     output reg [4:0] idx_op,
449
450     // register data pipe
451     output reg [15:0] idx_rd1,
452     output reg [15:0] idx_rd2,
453
454     // not clocked
455     output [4:0] dec_op,
456     output [2:0] reg_rs1, output [2:0] reg_rs2,
457     input [15:0] reg_rd1, input [15:0] reg_rd2,
458     output dec_has_imm8,
459
460     // computed rd3 data
461     output reg [15:0] idx_rd3,
462
463     // register select pipe
464     output reg [2:0] idx_rs1,
465     output reg [2:0] idx_rs2,
466
467

```

```

468     output reg idex_has_br,
469     output reg idex_has_we,
470     output reg idex_has_mem,
471     output reg idex_has_mem_we,
472
473     output dec_halt,
474
475     input stall, input jmping,
476     input ifid_valid, output reg idex_valid
477 );
478     wire dec_has_br;
479     wire dec_has_simm5;
480     wire dec_has_we;
481     wire dec_has_mem;
482     wire dec_has_mem_we;
483     wire [4:0] alu_op;
484     wire [7:0] dec_imm8;
485     wire [4:0] dec_simm5;
486     vmicro16_dec decoder (
487         .instr      (ifid_instr),
488         .opcode      (dec_op),
489         .rd          (reg_rs1),
490         .ra          (reg_rs2),
491         .imm8        (dec_imm8),
492         .has_imm8     (dec_has_imm8 ),
493         .simm5        (dec_simm5 ),
494         .has_br       (dec_has_br  ),
495         .has_we       (dec_has_we  ),
496         //.has_bad     (dec_has_bad  ),
497         .has_mem      (dec_has_mem  ),
498         .has_mem_we   (dec_has_mem_we ),
499         .alu_op       (alu_op),
500         .halt        (dec_halt)
501     );
502
503     // Clock values through the pipeline
504     always @(posedge clk)
505     if (!reset) begin
506         if (!stall) begin
507             // Move previous stage regs into this stage
508             idex_pc <= ifid_pc; // Only for simulation
509             idex_rd1 <= reg_rd1; // clock the decoder outputs into regs
510             idex_rd2 <= reg_rd2; // clock the decoder outputs into regs
511             idex_rs1 <= reg_rs1; // destination register
512             idex_rs2 <= reg_rs2; // operand register
513             // store decoded instr
514             idex_op <= alu_op;
515             idex_has_br <= dec_has_br;
516             idex_has_we <= dec_has_we;
517             idex_has_mem <= dec_has_mem;
518             idex_has_mem_we <= dec_has_mem_we;
519
520             if ((dec_op == `VMICRO16_OP_SW) || (dec_op == `VMICRO16_OP_LW))
521                 idex_rd3 <= reg_rd2 + { {11{dec_imm8[4]}}, dec_simm5 };
522             else if (dec_has_imm8)
523                 idex_rd3 <= { {8{dec_imm8[7]}}, dec_imm8 };
524             else if ((dec_op == `VMICRO16_OP_ARITH_U && ifid_instr[4] == 0))
525                 idex_rd3 <= reg_rd2 + { {12{1'b0}}, ifid_instr[3:0] };
526             else if ((dec_op == `VMICRO16_OP_ARITH_S && ifid_instr[4] == 0))
527                 idex_rd3 <= reg_rd2 + { {12{ifid_instr[3]}}, ifid_instr[3:0] };
528             else
529                 idex_rd3 <= reg_rd2;
530
531             idex_valid <= stall ? 1'b0 : (ifid_valid && !jmping);
532         end else begin
533             idex_valid <= 1'b0;
534             idex_has_we <= 1'b0;
535             idex_has_mem <= 1'b0;
536             idex_has_mem_we <= 1'b0;
537             idex_rd1 <= 1'b0;
538             idex_rd2 <= 1'b0;
539             idex_rd3 <= 1'b0;
540         end
541     end
542 endmodule
543
544 module vmicro16_exme (
545     input clk,
546     input reset,
547
548     input [15:0] idex_pc, output reg [15:0] exme_pc,
549
550     input [4:0] idex_op, output reg [4:0] exme_op,
551     output reg [15:0] exme_d,
552     input [15:0] idex_rd1, output reg [15:0] exme_d2,
553     // input [15:0] idex_rd2,
554     input [15:0] idex_rd3,
555
556     input [2:0] idex_rs1, output reg [2:0] exme_rs1,
557     input [2:0] idex_rs2, output reg [2:0] exme_rs2,
558
559     input idex_has_br, output reg exme_has_br,
560     input idex_has_we, output reg exme_has_we,
561     input idex_has_mem, output reg exme_has_mem,
562     input idex_has_mem_we, output reg exme_has_mem_we,
563     input idex_valid,
564     input jmping, output reg exme_valid,
565
566

```

```

567     output reg [15:0] exme_jump_target
568 );
569 // ALU
570 wire [15:0] alu_c;
571 vmicro16_alu alu (
572     .op(index_op),
573     .d1(index_rd1),
574     .d2(index_rd3),
575     .q(alu_c)
576 );
577
578 always @(posedge clk)
579 if (!reset) begin
580     // Move previous stage regs into this stage
581     exme_pc      <= index_pc; // Only for simulation
582     // exme_d contains the result data value or
583     // address for LW/SW
584     exme_d       <= alu_c;
585     exme_d2      <= index_rd1;
586     // exme_rs contains the destination register for
587     // the data value or memory after it's fetched
588     exme_rs1     <= index_rs1;
589     exme_rs2     <= index_rs2;
590
591     exme_has_br  <= index_has_br;
592     exme_has_we  <= index_has_we;
593     exme_has_mem <= index_has_mem;
594     exme_has_mem_we <= index_has_mem_we;
595
596     exme_valid   <= index_valid && !jumping;
597
598     // Relative PC jmp target, PC = PC + rd1
599     exme_jump_target <= (index_has_br) ?
600         (index_pc + index_rd1) :
601         1'b0;
602 end else begin
603     exme_valid   <= 1'b0;
604     exme_d       <= 16'h0;
605     exme_d2      <= 16'h0;
606     exme_has_mem <= 1'b0;
607     exme_has_mem_we <= 1'b0;
608 end
609 endmodule
610
611 module vmicro16_mewb (
612     input clk,
613     input reset,
614
615     input [15:0] exme_pc, output reg [15:0] mewb_pc,
616
617     input [15:0] exme_d, output reg [15:0] mewb_d,
618     input [15:0] mem_out, output reg [15:0] mewb_d,
619
620     input [15:0] exme_d2, output reg [15:0] mewb_d2,
621     input [2:0] exme_rs1, output reg [2:0] mewb_rs1,
622
623     input [15:0] exme_jump_target,
624     output reg [15:0] mewb_jump_target,
625
626     input exme_has_br, output reg mewb_has_br,
627     input exme_has_we, output reg mewb_has_we,
628     input exme_has_mem, output reg mewb_has_mem,
629     input exme_has_mem_we, output reg mewb_has_mem_we,
630
631     input mem_valid,
632     input exme_valid, output reg mewb_valid,
633
634     input jumping
635 );
636 // MEWB stage
637 always @(posedge clk)
638 if (!reset) begin
639     if (exme_valid) begin
640         // Move previous stage regs into this stage
641         mewb_pc      <= exme_pc; // Only for simulation
642
643         if (exme_has_mem) mewb_d <= mem_out; // from mmu
644         else mewb_d <= exme_d; // from alu
645
646         mewb_d2      <= exme_d2;
647         mewb_rs1     <= exme_rs1;
648         mewb_jump_target <= exme_jump_target;
649
650         mewb_has_br  <= exme_has_br;
651         mewb_has_we  <= exme_has_we;
652         mewb_has_mem <= exme_has_mem;
653         mewb_has_mem_we <= exme_has_mem_we;
654
655         if (exme_has_mem)
656             // LW
657             if (!exme_has_mem_we)
658                 $display($time, "\tMEWB: LW: r[%h] <= mem[%h]",
659                     exme_rs1, exme_d);
660         mewb_valid <= exme_valid && !jumping && mem_valid;
661     end else begin
662         mewb_valid   <= 1'b0;
663         mewb_has_br  <= 1'b0;
664         mewb_has_we  <= 1'b0;
665         mewb_has_mem <= 1'b0;

```

```

666         mewb_has_mem    <= 1'b0;
667         mewb_has_mem_we <= 1'b0;
668     end
669 endmodule
670
671 module vmicro16_wb (
672     input clk,
673     input reset,
674
675     input jmping,
676
677     input [15:0] mewb_d,          output reg [15:0] wb_d,
678     input [2:0]  mewb_rs1,        output reg [2:0]  wb_rs1,
679     input        mewb_has_we,     output reg        wb_we,
680     input        mewb_has_br,     output reg        wb_has_br,
681     input [15:0] mewb_jmp_target, output reg [15:0] wb_jmp_target,
682
683     input mewb_valid,             output reg wb_valid
684 );
685 // WB stage
686 always @(posedge clk)
687 if (!reset) begin
688     if (mewb_valid) begin
689         wb_d      <= mewb_d;
690         wb_we      <= mewb_has_we;
691         wb_rs1     <= mewb_rs1;
692         wb_has_br  <= mewb_has_br;
693         wb_jmp_target <= mewb_jmp_target;
694     end
695     wb_valid <= mewb_valid && !jmping;
696 end else begin
697     wb_valid <= 1'b0;
698     wb_we    <= 1'b0;
699     wb_has_br <= 1'b0;
700     wb_jmp_target <= 1'b0;
701 end
702 endmodule
703
704 module vmicro16_mmu # (
705     parameter MEM_WIDTH    = 16,
706     parameter MEM_DEPTH    = 1024,
707
708     parameter MEM_RVAL     = 16'h00CC
709 ) (
710     input clk,
711     input reset,
712
713     output valid,
714
715     input req,
716
717     input [15:0] mem_addr,
718     input [15:0] mem_in,
719     input        mem_we,
720     input [1:0]  mem_why, // TODO: apply to mem_out
721     output reg [15:0] mem_out,
722
723     // wishbone peripheral master
724     output reg wb_mosi_stb_o_regs,
725     //output wb_stb_o_xx,
726     output reg wb_mosi_cyc_o,
727     output [15:0] wb_mosi_addr_o,
728     output        wb_mosi_we_o,
729     output [15:0] wb_mosi_data_o, // seperate data_o and data_i buses
730     input [15:0] wb_miso_data_i, // seperate data_o and data_i buses
731
732     input        wb_miso_ack_i
733 );
734
735 wire [15:0] bram_out;
736
737 ///////////////////////////////////////////////////
738 //      TODO: CLEANUP
739 ///////////////////////////////////////////////////
740 reg active = 0;
741 always @(*)
742 if (req) begin
743     active = 1'b1;
744 end else if (wb_miso_ack_i) begin
745     active = 1'b0;
746 end else
747     active = active;
748
749 ///////////////////////////////////////////////////
750 //      TODO: CLEANUP
751 ///////////////////////////////////////////////////
752 always @(*)
753 if (reset)
754     mem_out = 16'h0;
755 else if (active && wb_mosi_stb_o_regs) begin
756     wb_mosi_cyc_o = 1'b1;
757     if (wb_miso_ack_i) begin
758         mem_out = wb_miso_data_i;
759     end
760 end else if (active) begin
761     mem_out = bram_out;
762 end else begin
763     wb_mosi_cyc_o = 1'b0;
764     // TODO: mem_out isn't valid in this state, output high z or 0?

```

```

765         mem_out = 16'hZZ;
766     end
767
768     // bram memory is always single clk, wb is unknown
769     assign valid = active ? wb_miso_ack_i : 1'b1;
770
771     ///////////////////////////////////////////////////
772     //      TODO: CLEANUP
773     ///////////////////////////////////////////////////
774     // Virtual memory translator
775     always @(*) begin
776         // zero all peripherals
777         wb_mosi_stb_o_regs = 0;
778
779         if (req) begin
780             // enable peripherals when their address is selected
781             casez(mem_addr)
782                 15'h01??: wb_mosi_stb_o_regs = 1'b1;
783                 default:  wb_mosi_stb_o_regs = 1'b0;
784             endcase
785         end
786     end
787
788     wire wb_active      = wb_mosi_cyc_o; // deprecated
789     wire wb_stb         = wb_mosi_stb_o_regs; // || req;
790     //assign wb_mosi_cyc_o = wb_stb;
791     assign wb_mosi_we_o  = wb_active ? mem_we  : 1'b0;
792     assign wb_mosi_addr_o = wb_active ? mem_addr : 16'h00;
793     assign wb_mosi_data_o = wb_active ? mem_in  : 16'h00;
794
795     //assign mem_out      = wb_active ? wb_miso_data_i : bram_out;
796
797
798     // TODO: Should this be inside the mmu or outside?
799     wire bram_we = mem_we && !wb_active;
800     vmicro16_bram # (
801         .MEM_WIDTH(MEM_WIDTH), // TODO: mem 16b or 8b wide?
802         .MEM_DEPTH(MEM_DEPTH)
803     ) bram (
804         .clk      (clk),
805         .reset    (reset),
806         // port 1
807         .mem_addr (mem_addr),
808         .mem_in   (mem_in),
809         .mem_we    (bram_we),
810         .mem_out   (bram_out)
811     );
812
813     // reset must be held long for atleast MEM_SIZE clocks
814     // to fully erase the bram.
815     // E.g. Xilinx bram 1024 cells = 1024 clocks = ~21us
816     // TODO: implement with a dfa
817 endmodule
818
819 module vmicro16_cpu (
820     input clk,
821     input reset,
822
823     // wishbone peripheral master interface
824     // driven by mmu
825     output wb_mosi_stb_o_regs, // ...
826     output wb_mosi_cyc_o,
827     output wb_mosi_we_o,
828     output [15:0] wb_mosi_addr_o,
829     output [15:0] wb_mosi_data_o, // seperate data_o and data_i buses
830     input  [15:0] wb_miso_data_i, // seperate data_o and data_i buses
831     input  wb_miso_ack_i
832 );
833
834     wire [4:0] dec_op;
835     wire [7:0] dec_imm8;
836     wire      dec_has_imm8;
837     wire [4:0] dec_simm5;
838     wire      dec_has_br;
839     wire      dec_has_we;
840     wire      dec_has_mem;
841     wire      dec_has_mem_we;
842     wire      dec_has_bad;
843
844     wire [15:0] ifid_pc;
845     wire [15:0] ifid_instr;
846
847     wire [15:0] reg_rd1;
848     wire [15:0] reg_rd2;
849     wire [2:0]  reg_rs1;
850     wire [2:0]  reg_rs2;
851
852     wire ifid_valid;
853     wire idex_valid;
854     wire exme_valid;
855     wire mewb_valid;
856     wire wb_valid;
857     wire mem_valid;
858     wire dec_halt;
859     wire wb_has_br;
860
861     wire [2:0] idex_rs1;
862     wire [2:0] exme_rs1;
863     wire [2:0] mewb_rs1;
864     wire [2:0] wb_rs1;

```

```

864
865 // nop = not any bits set in dec_op
866 wire nop = ~(|dec_op);
867 wire stall_ifid = (~nop) && ifid_valid;
868 wire stall_index = (~nop && index_valid) && ((reg_rs1 == index_rs1) || ((~dec_has_imm8) && (reg_rs2 == index_rs1)));
869 wire stall_exme = (~nop && exme_valid) && ((reg_rs1 == exme_rs1) || ((~dec_has_imm8) && (reg_rs2 == exme_rs1)));
870 wire stall_mewb = (~nop && mewb_valid) && ((reg_rs1 == mewb_rs1) || ((~dec_has_imm8) && (reg_rs2 == mewb_rs1)));
871 wire stall_wb = (~nop && wb_valid) && ((reg_rs1 == wb_rs1) || ((~dec_has_imm8) && (reg_rs2 == wb_rs1)));
872 wire stall_mem = 1'b0;
873 wire stall = |{ stall_index,
874                stall_exme,
875                stall_mewb,
876                stall_wb,
877                dec_halt,
878                !mem_valid };
879 wire jumping = (wb_valid && wb_has_br);
880
881
882 wire [15:0] wb_d;
883 wire [15:0] wb_jmp_target;
884 wire wb_we;
885 wire wb_we_w = reset ? 1'b0 : (wb_we && wb_valid);
886 vmicro16_regs # (
887     .CELL_WIDTH(16),
888     .CELL_DEPTH(8)
889 ) regs (
890     .clk (clk),
891     .reset (reset),
892
893     .rs1 (reg_rs1),
894     .rd1 (reg_rd1),
895
896     .rs2 (reg_rs2),
897     .rd2 (reg_rd2),
898
899     .we (wb_we_w),
900     .ws1 (wb_rs1),
901     .wd (wb_d)
902 );
903
904 // stage_ifid
905 vmicro16_ifid stage_ifid (
906     .clk (clk),
907     .reset (reset),
908     .stall (stall),
909     .jumping (jumping),
910     .wb_jmp_target (wb_jmp_target),
911     .mewb_valid (mewb_valid),
912     .ifid_valid (ifid_valid),
913     .ifid_pc (ifid_pc),
914     .ifid_instr (ifid_instr)
915 );
916
917 wire [15:0] index_pc;
918 wire [15:0] index_instr;
919 wire [2:0] index_rs2;
920 wire [15:0] index_rd1;
921 wire [15:0] index_rd2;
922 wire [15:0] index_rd3;
923 wire [4:0] index_op;
924 wire index_has_br;
925 wire index_has_mem;
926 wire index_has_mem_we;
927 wire index_has_we;
928 vmicro16_index stage_index (
929     .clk (clk),
930     .reset (reset),
931
932     .ifid_pc (ifid_pc),
933     .index_pc (index_pc),
934
935     .ifid_instr (ifid_instr),
936     .index_instr (index_instr),
937
938     // not clocked
939     .dec_op (dec_op),
940     .reg_rs1 (reg_rs1),
941     .reg_rs2 (reg_rs2),
942     .reg_rd1 (reg_rd1),
943     .reg_rd2 (reg_rd2),
944     .dec_has_imm8 (dec_has_imm8),
945
946     .index_rd1 (index_rd1),
947     .index_rd2 (index_rd2),
948     .index_rd3 (index_rd3),
949
950     .index_rs1 (index_rs1),
951     .index_rs2 (index_rs2),
952
953     .index_has_br (index_has_br),
954     .index_has_we (index_has_we),
955     .index_has_mem (index_has_mem),
956     .index_has_mem_we (index_has_mem_we),
957
958     .dec_halt (dec_halt),
959
960     .stall (stall),
961     .jumping (jumping),
962

```



```

963         .ifid_valid      (ifid_valid),
964         .idex_valid      (idex_valid),
965
966         .idex_op         (idex_op)
967     );
968
969     // NEW
970     wire [15:0] exme_pc;
971     wire [4:0]  exme_op;
972     wire [15:0] exme_d;
973     wire [15:0] exme_d2;
974     // PASS
975     wire [2:0]  exme_rs2;
976     wire       exme_has_br;
977     wire       exme_has_we;
978     wire       exme_has_mem;
979     wire       exme_has_mem_we;
980     wire [15:0] exme_jmp_target;
981     vmicro16_exme stage_exme (
982         .clk             (clk),
983         .reset           (reset),
984         // Status registers
985         .jumping         (jumping),
986         // Pass through registers
987         .idex_pc         (idex_pc),           .exme_pc         (exme_pc),
988         .idex_rs1        (idex_rs1),         .exme_rs1         (exme_rs1),
989         .idex_rs2        (idex_rs2),         .exme_rs2         (exme_rs2),
990         .idex_has_br     (idex_has_br),       .exme_has_br     (exme_has_br),
991         .idex_has_we     (idex_has_we),       .exme_has_we     (exme_has_we),
992         .idex_has_mem     (idex_has_mem),     .exme_has_mem     (exme_has_mem),
993         .idex_has_mem_we (idex_has_mem_we),   .exme_has_mem_we (exme_has_mem_we),
994         .idex_valid      (idex_valid),        .exme_valid      (exme_valid),
995         // ALU ops
996         .idex_op         (idex_op),           .exme_op         (exme_op), //PASS
997         .exme_d          (exme_d),
998         .idex_rd1        (idex_rd1),         .exme_d2         (exme_d2), //PASS
999         .idex_rd3        (idex_rd3),
1000         .exme_jmp_target (exme_jmp_target)
1001     );
1002
1003
1004
1005     wire [15:0] mem_out;
1006     // If SW, use calculated address
1007     wire [15:0] mem_addr = exme_has_mem ? exme_d : 16'h00;
1008     // If SW, use register value
1009     wire [15:0] mem_in  = exme_has_mem ? exme_d2 : exme_d;
1010     wire        mem_we  = reset ? 1'b0 : (exme_has_mem_we & exme_valid);
1011     wire [1:0]  mem_why = 2'b00; // TODO: implement in ISA
1012     vmicro16_mmu mmu (
1013         .clk             (clk),
1014         .reset           (reset),
1015
1016         .req             (exme_has_mem && exme_valid),
1017         .valid           (mem_valid),
1018
1019         .mem_addr        (mem_addr),
1020         .mem_in          (mem_in),
1021         .mem_we          (mem_we),
1022         .mem_why         (mem_why),
1023         .mem_out         (mem_out),
1024
1025         // wishbone master interface
1026         // TODO: Add to top level cpu
1027         .wb_mosi_stb_o_regs (wb_mosi_stb_o_regs),
1028         .wb_mosi_cyc_o     (wb_mosi_cyc_o),
1029         .wb_mosi_we_o      (wb_mosi_we_o),
1030         .wb_mosi_addr_o    (wb_mosi_addr_o),
1031         .wb_mosi_data_o    (wb_mosi_data_o),
1032         .wb_miso_data_i    (wb_miso_data_i),
1033         .wb_miso_ack_i     (wb_miso_ack_i)
1034     );
1035
1036     wire [15:0] mewb_pc;
1037     wire [15:0] mewb_d;
1038     wire [15:0] mewb_d2;
1039     wire [15:0] mewb_jmp_target;
1040     wire        mewb_has_mem;
1041     wire        mewb_has_mem_we;
1042     wire        mewb_has_br;
1043     wire        mewb_has_we;
1044     vmicro16_mewb stage_mewb (
1045         .clk             (clk),
1046         .reset           (reset),
1047
1048         .jumping         (jumping),
1049
1050         .mem_out         (mem_out),
1051         .mem_valid       (mem_valid),
1052
1053         .exme_pc         (exme_pc),
1054         .mewb_pc         (mewb_pc),
1055
1056         .exme_d          (exme_d),
1057         .mewb_d          (mewb_d),
1058         .exme_d2         (exme_d2),
1059         .mewb_d2         (mewb_d2),
1060         .exme_rs1        (exme_rs1),
1061         .mewb_rs1        (mewb_rs1),

```

```

1062
1063         .exme_jmp_target (exme_jmp_target),
1064         .mewb_jmp_target (mewb_jmp_target),
1065
1066         .exme_has_br      (exme_has_br),
1067         .mewb_has_br      (mewb_has_br),
1068         .exme_has_we      (exme_has_we),
1069         .mewb_has_we      (mewb_has_we),
1070         .exme_has_mem     (exme_has_mem),
1071         .mewb_has_mem     (mewb_has_mem),
1072         .exme_has_mem_we  (exme_has_mem_we),
1073         .mewb_has_mem_we  (mewb_has_mem_we),
1074
1075         .exme_valid       (exme_valid),
1076         .mewb_valid       (mewb_valid)
1077     );
1078
1079
1080     // WB stage
1081     vmicro16_wb stage_wb (
1082         .clk             (clk),
1083         .reset           (reset),
1084
1085         .mewb_d          (mewb_d),
1086         .wb_d            (wb_d),
1087
1088         .mewb_rs1        (mewb_rs1),
1089         .wb_rs1          (wb_rs1),
1090
1091         .mewb_has_we     (mewb_has_we),
1092         .wb_we           (wb_we),
1093
1094         .mewb_has_br     (mewb_has_br),
1095         .wb_has_br       (wb_has_br),
1096
1097         .mewb_jmp_target (mewb_jmp_target),
1098         .wb_jmp_target   (wb_jmp_target),
1099
1100         .jumping         (jumping),
1101
1102         .mewb_valid      (mewb_valid),
1103         .wb_valid        (wb_valid)
1104     );
1105
1106
1107
1108
1109     endmodule

```

vmicro16_soc.v

```

1  module vmicro16_soc (
2      input clk,
3      input reset
4  );
5      // Internal wishbone master interface
6      wire  wb_mosi_stb_o_regs;
7      wire  wb_mosi_cyc_o;
8      wire [15:0] wb_mosi_addr_o;
9      wire  wb_mosi_we_o;
10     wire [15:0] wb_mosi_data_o; // seperate data_o and data_i buses
11     wire [15:0] wb_miso_data_i; // seperate data_o and data_i buses
12     wire  wb_miso_ack_i;
13
14     vmicro16_regs_wb # (
15         .CELL_WIDTH      (16),
16         .CELL_DEPTH      (8),
17         .CELL_SEL_BITS   (3),
18         // Default soc conf registers
19         .CELL_DEFAULTS   ("../../soc_sregs.txt"),
20         .DEBUG_NAME      ("soc_regs")
21     ) soc_regs (
22         .clk             (clk),
23         .reset           (reset),
24
25         .wb_stb_i        (wb_mosi_stb_o_regs),
26         .wb_cyc_i        (wb_mosi_cyc_o),
27         .wb_we_i         (wb_mosi_we_o),
28         .wb_addr_i       (wb_mosi_addr_o),
29         .wb_data_i       (wb_mosi_data_o),
30         .wb_data_o       (wb_miso_data_i),
31         .wb_ack_o        (wb_miso_ack_i),
32         // .wb_stall_o     (wb_stall_o),
33         // .wb_err_o      (wb_err_o),
34     );
35
36     vmicro16_cpu core (
37         .clk             (clk),
38         .reset           (reset),
39
40         // Wishbone master interface
41         .wb_mosi_stb_o_regs (wb_mosi_stb_o_regs),
42         .wb_mosi_cyc_o     (wb_mosi_cyc_o),
43         .wb_mosi_addr_o    (wb_mosi_addr_o),
44         .wb_mosi_we_o     (wb_mosi_we_o),

```

```

45         .wb_mosi_data_o      (wb_mosi_data_o),
46         .wb_miso_data_i      (wb_miso_data_i),
47         .wb_miso_ack_i       (wb_miso_ack_i)
48     );
49
50 endmodule

```

vmicro16_isa.v

```

1  // Vmicro16 multi-core instruction set
2
3  // TODO: Remove NOP by making a register write/read always 0
4  `define VMICRO16_OP_NOP      5'b00000
5  `define VMICRO16_OP_LW      5'b00001
6  `define VMICRO16_OP_SW      5'b00010
7  `define VMICRO16_OP_BIT     5'b00011
8  `define VMICRO16_OP_BIT_OR  5'b00000
9  `define VMICRO16_OP_BIT_XOR 5'b00001
10 `define VMICRO16_OP_BIT_AND 5'b00010
11 `define VMICRO16_OP_BIT_NOT 5'b00011
12 `define VMICRO16_OP_BIT_LSHFT 5'b00100
13 `define VMICRO16_OP_BIT_RSHFT 5'b00101
14 `define VMICRO16_OP_MOV      5'b00100
15 `define VMICRO16_OP_MOVI     5'b00101
16 `define VMICRO16_OP_MOVI_L   5'b10000
17 `define VMICRO16_OP_ARITH_U   5'b00110
18 `define VMICRO16_OP_ARITH_UADD 5'b11111
19 `define VMICRO16_OP_ARITH_USUB 5'b10000
20 `define VMICRO16_OP_ARITH_UADDI 5'b0????
21 `define VMICRO16_OP_ARITH_S   5'b00111
22 `define VMICRO16_OP_ARITH_SADD 5'b11111
23 `define VMICRO16_OP_ARITH_SSUB 5'b10000
24 `define VMICRO16_OP_ARITH_SSUBI 5'b0????
25 `define VMICRO16_OP_BR       5'b01000
26 // TODO: wasted upper nibble bits in BR
27 `define VMICRO16_OP_BR_U      8'h00
28 `define VMICRO16_OP_BR_E      8'h01
29 `define VMICRO16_OP_BR_NE     8'h02
30 `define VMICRO16_OP_BR_G      8'h03
31 `define VMICRO16_OP_BR_GE     8'h04
32 `define VMICRO16_OP_BR_L      8'h05
33 `define VMICRO16_OP_BR_LE     8'h06
34 `define VMICRO16_OP_BR_S      8'h07
35 `define VMICRO16_OP_BR_NS     8'h08
36 `define VMICRO16_OP_CMP       5'b01001
37 `define VMICRO16_OP_SETC      5'b01010
38 `define VMICRO16_OP_HALT      5'b01011
39
40 // microcode operations
41 `define VMICRO16_ALU_BIT_OR    5'h00
42 `define VMICRO16_ALU_BIT_XOR  5'h01
43 `define VMICRO16_ALU_BIT_AND  5'h02
44 `define VMICRO16_ALU_BIT_NOT  5'h03
45 `define VMICRO16_ALU_BIT_LSHFT 5'h04
46 `define VMICRO16_ALU_BIT_RSHFT 5'h05
47 `define VMICRO16_ALU_LW       5'h06
48 `define VMICRO16_ALU_SW       5'h07
49 `define VMICRO16_ALU_NOP      5'h08
50 `define VMICRO16_ALU_MOV      5'h09
51 `define VMICRO16_ALU_MOVI     5'h0a
52 `define VMICRO16_ALU_MOVI_L   5'h0b
53 `define VMICRO16_ALU_ARITH_UADD 5'h0c
54 `define VMICRO16_ALU_ARITH_USUB 5'h0d
55 `define VMICRO16_ALU_ARITH_SADD 5'h0e
56 `define VMICRO16_ALU_ARITH_SSUB 5'h0f
57 `define VMICRO16_ALU_BR_U     5'h10
58 `define VMICRO16_ALU_BR_E     5'h11
59 `define VMICRO16_ALU_BR_NE     5'h12
60 `define VMICRO16_ALU_BR_G     5'h13
61 `define VMICRO16_ALU_BR_GE     5'h14
62 `define VMICRO16_ALU_BR_L     5'h15
63 `define VMICRO16_ALU_BR_LE     5'h16
64 `define VMICRO16_ALU_BR_S     5'h17
65 `define VMICRO16_ALU_BR_NS     5'h18
66 `define VMICRO16_ALU_CMP       5'h19
67 `define VMICRO16_ALU_SETC      5'h1a
68 `define VMICRO16_ALU_ARITH_UADDI 5'h1b
69 `define VMICRO16_ALU_ARITH_SSUBI 5'h1c
70 `define VMICRO16_ALU_BR       5'h1d
71 `define VMICRO16_ALU_SPARE     5'h1e
72 `define VMICRO16_ALU_BAD       5'h1f

```