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

August 18, 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: August 18, 2019

Table of Contents

Chapter 1

Introduction

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

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

1.1 Why Multi-core?

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

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

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

1.2   Why RISC?

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

In many applications, not restricted to software, there may exists 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 parallised, 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. Amdalh'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:

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

This formula will be used throughout the project to gauge the 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 it's own dedicated memory and IO modules. Communication between nodes is performed over a *Message Transfer System (MTS)* [**?**] 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 there 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 it's 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.
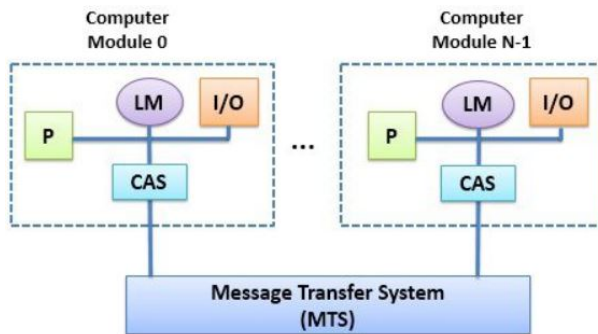
Figure 2.1: A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [?].
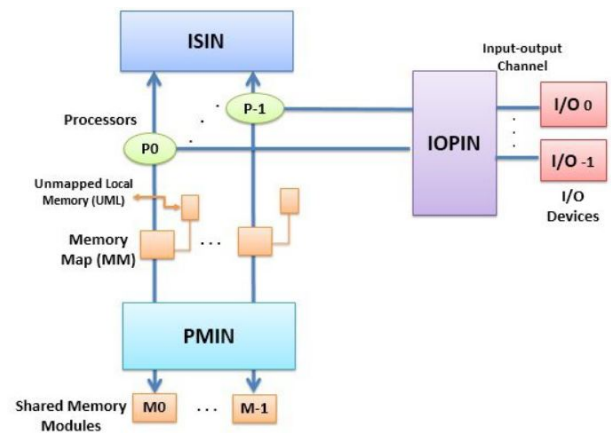
Figure 2.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [?].

This project will utilise a loosely coupled architecture due to it's 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 [?]. 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 [?].
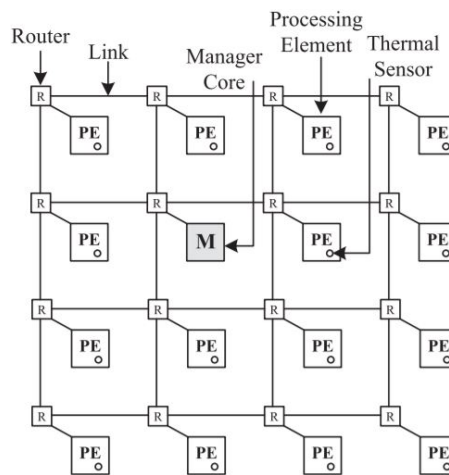
Figure 2.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [**?**].

Chapter 3

Project Overview

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 [? ?]). 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 [?] compiler to support the ISA from1 CD1. This will make it easier to build complex multi-core software for the processor.

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

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

ED7 Store instruction memory in SPI flash.

ED8 Reprogram instruction memory at runtime from host computer.

ED9 Processor external debugger using host-processor link.

3.2 Project Timeline

3.2.1 Project Stages

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

The project stages are shown in Table 3.1.

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.
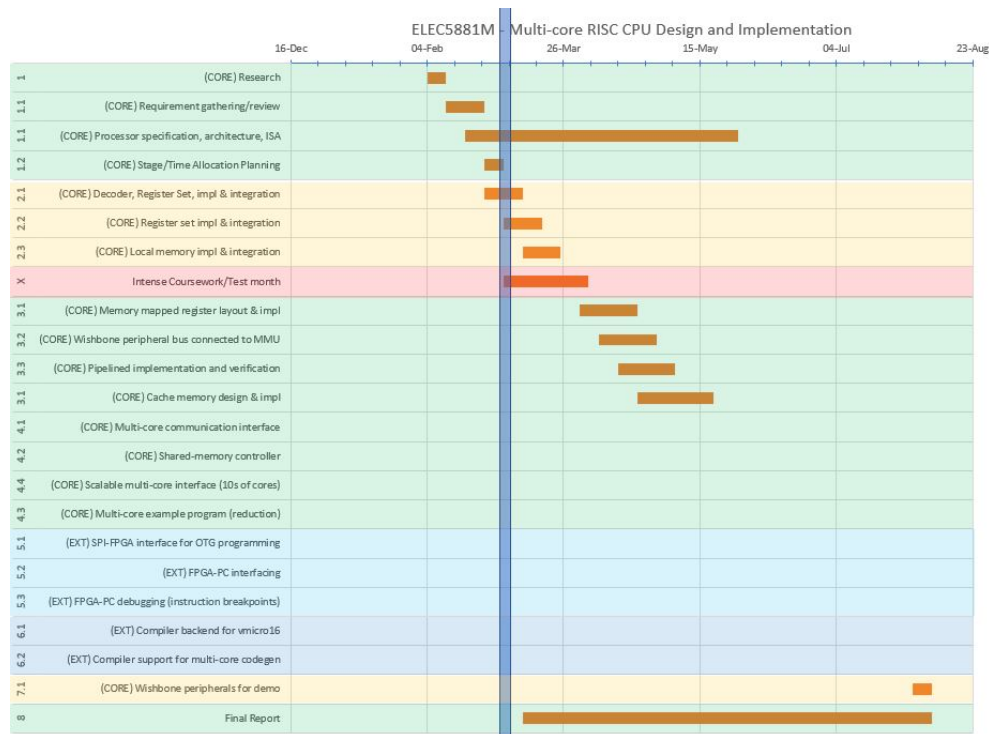
Figure 3.1: Project stages in a Gantt chart.

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.

Minispartan 6+ FPGA Development Board

The Minispartan 6+ is a hobbyist FGPA 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-
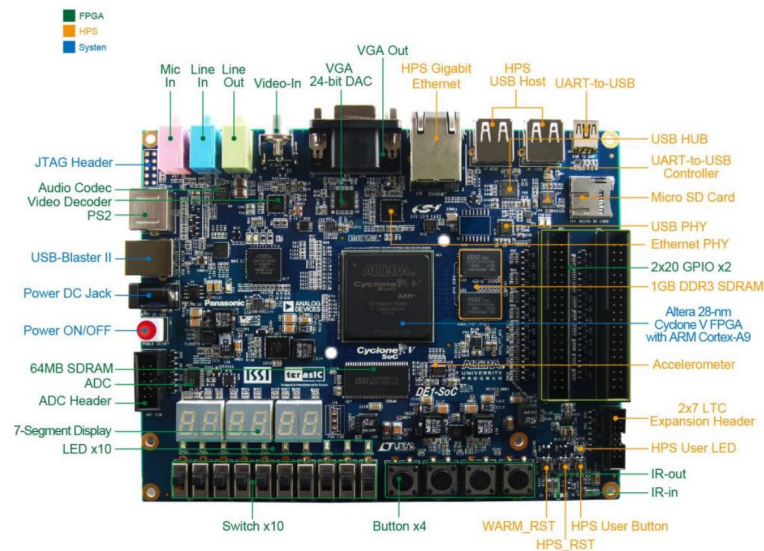
Figure 3.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [2].

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.
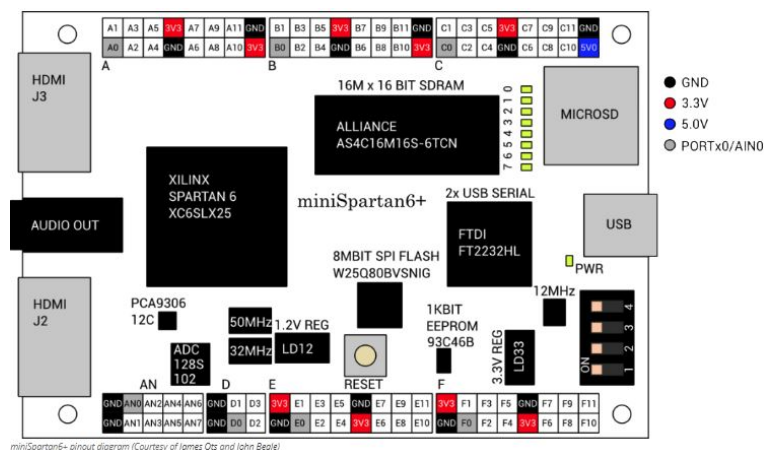

Figure 3.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [3].

3.3.2 Software Resources

Intel Quartus

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

Xilinx ISE Webpack

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

Verilator

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

3.4 Legal and Ethical Considerations

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

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

The processor does not store/track/predict software instructions. The processor uses pipelining techniques to improve performance which results in future instructions entering the pipeline even if the software's logical sequence does not include these instructions. This could result in security vulnerabilities similar to Intel's Spectre vulnerability [4].

Chapter 4

Current Progress

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

4.1   RISC Core

Following the project time line described in section 3.2, the first couple months have been dedicated to the design and implementation of the instruction set architecture and RISC core with stages 1-3. Good progress has been made in both deliverables, the ISA and the RISC core, and the progress is on-time with the initial project time line. 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 atleast 16 opcodes available to be provide multi-core communication and functionality. This number should be adequate to support these features when the work begins on the multi-core project stages (stages 4-7).

Design Goals

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

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

ISA2 **Be able to select at least two registers for common instructions.**

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

ISA3 **Reduce bit-space for frequently used instructions (MOV, MOVI, ADD).**

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

ISA4 **Provide frequently used actions as options for existing instructions.**

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

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

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

ISA5 **Provide instructions for performing bitwise manipulations.**

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

ISA6 **Provide instructions for explicitly performing signed and unsigned arithmetic.**

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

Without explicit unsigned and signed instructions, extra instructions would be required to perform addition and subtraction. In addition, due to two's complement representation of

signed numbers, the highest immediate operand value would be halved, resulting in more instructions to reach the desired value.

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

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

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

4.1.2   Design and Implementation

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

To satisfy **CD5**, the Verilog code will be self-contained in a single file. This reduces the hierarchical complexity and eases cross-vendor project set-up as only a single file is required to be included. A disadvantage with this single file approach is that some external Verilog verification tools that I plan to use, such as Verilator, do not currently support multiple Verilog modules (due to an unfixed bug) within a single file.
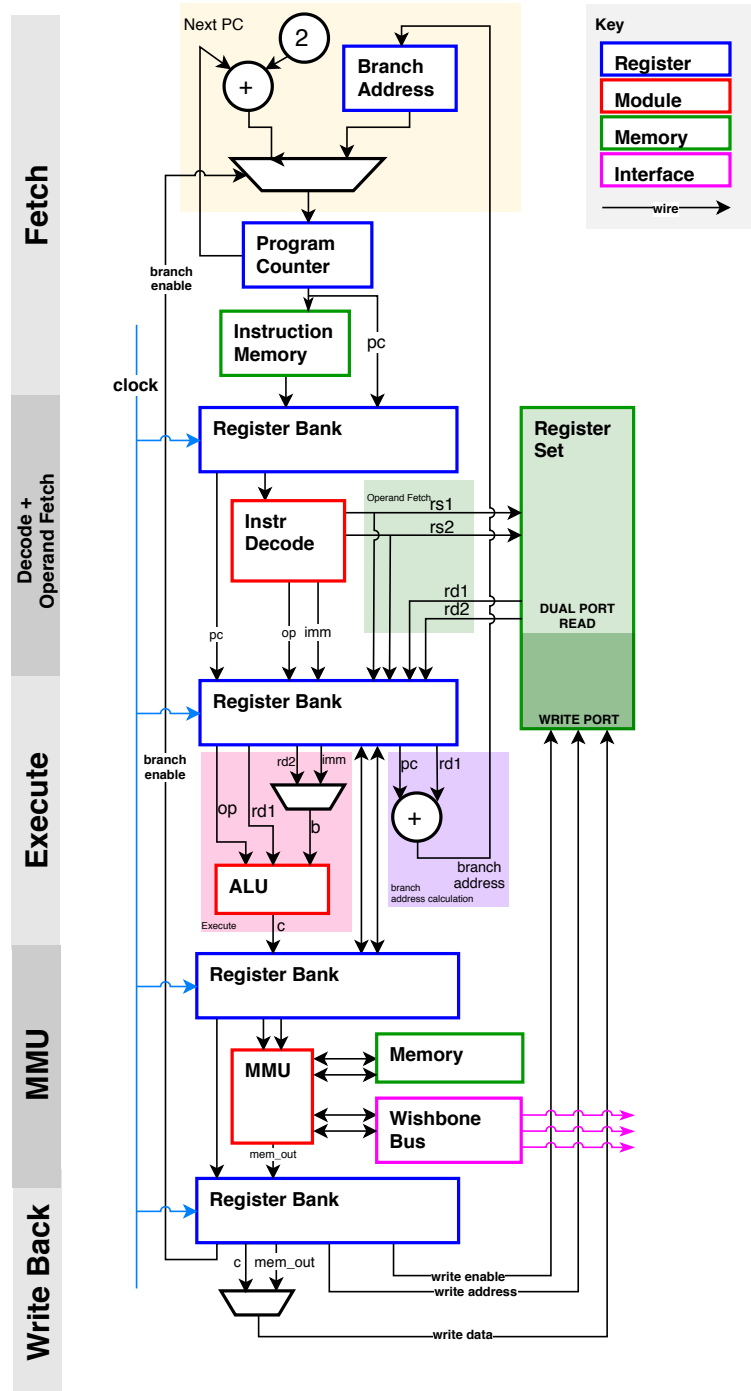
Figure 4.2: Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

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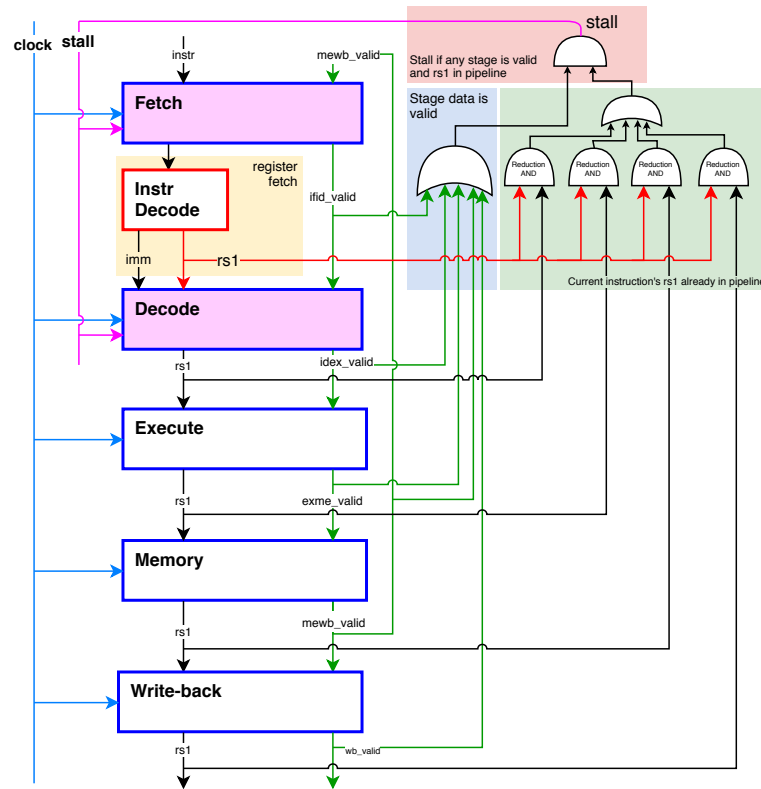
Figure 4.3: Pipeline data hazard detection. The register selects are passed forward through each stage and compared to the IDEX (latest instruction) register selects. If they match, the latest instruction depends on the output of an instruction in the pipeline, the IFID and IDEX stages are stalled to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

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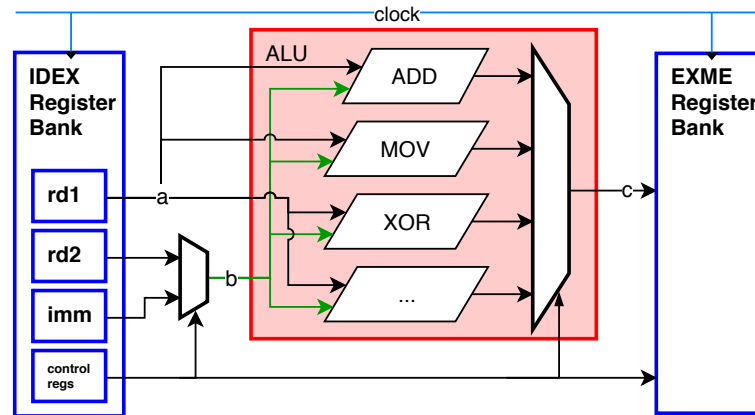
Figure 4.4: Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

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                    end
323
324                MMU_STATE_T3: begin
325                    // Slave has output a ready signal (finished)
326                    M_PENABLE <= 0;
327                    M_PADDR   <= 0;
328                    M_PWDATA  <= 0;
329                    M_PSELx   <= 0;
330                    M_PWRITE  <= 0;
331                    // Clock the peripheral output into a reg,
332                    //   to output on the next clock cycle
333                    per_out   <= M_PRDATA;
334
335                    mmu_state <= MMU_STATE_T1;
```

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
225     // more luts than below but easier
226     //wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
227     //             && (mmu_addr <= `DEF_MMU_TIM0_E);
228     //wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
229     //             && (mmu_addr <= `DEF_MMU_SREG_E);
230     //wire intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
231     //             && (mmu_addr <= `DEF_MMU_INTSV_E);
232     //wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
233     //             && (mmu_addr <= `DEF_MMU_INTSM_E);
234
235     wire tim0_en = ~mmu_addr[12] && ~mmu_addr[9] && ~mmu_addr[7];
236     wire sreg_en = mmu_addr[7] && ~mmu_addr[4] && ~mmu_addr[5];
237     wire intv_en = mmu_addr[8] && ~mmu_addr[3];
238     wire intm_en = mmu_addr[8] && mmu_addr[3];
239
240     wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
241     wire tim0_we   = (tim0_en && mmu_we);
242     wire intv_we   = (intv_en && mmu_we);
243     wire intm_we   = (intm_en && mmu_we);
244
245     // Special register selects
```

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

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

4.1.3 Verification

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

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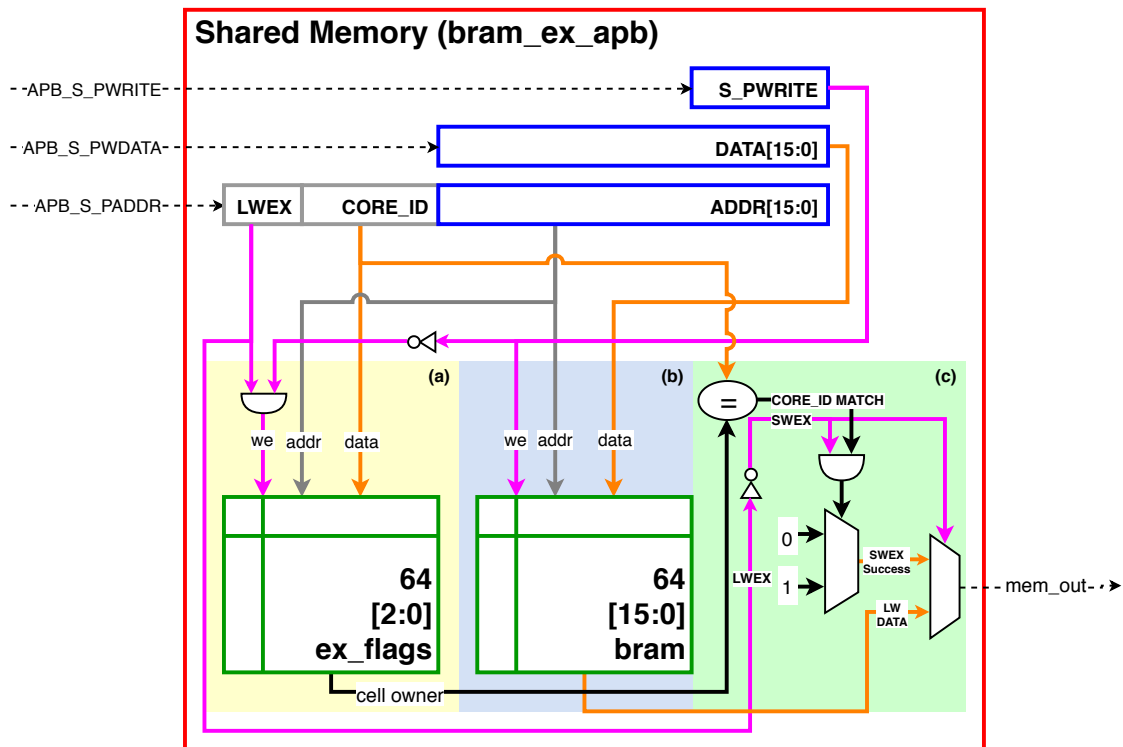

Figure 5.1: Caption for BRAMex

The current active stage is *3.3 Pipeline Implementation and Verification* where the processor pipeline is being verified against of 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 5.2.

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.
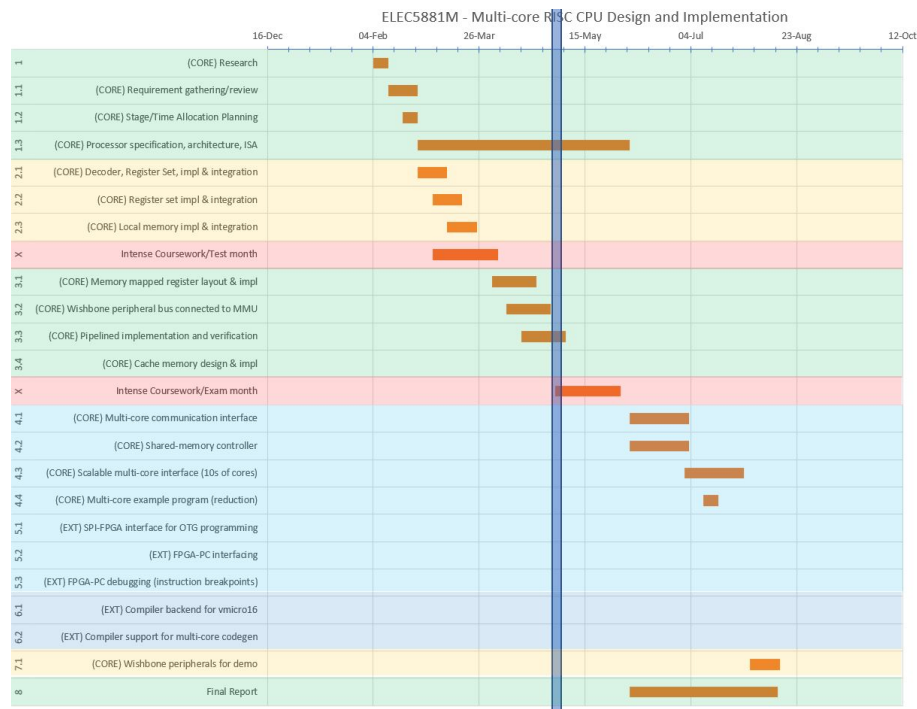
Figure 5.2: Updated project time gantt chart showing time allocations for stage 4.

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.

Chapter 6

Conclusion

With the end of Moore's Law looming, 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 project's 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] T. Technologies, "Soc platform - cyclone - de1-soc board." [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

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

[4] 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.

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

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

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

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.

```verilog
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  `include "clog2.v"
12  `include "formal.v"
13
14
15
16  // This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
17  //    https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
18  //    https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
19  //    https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf
20  module vmicro16_bram # (
21      parameter MEM_WIDTH     = 16,
22      parameter MEM_DEPTH     = 64,
23      parameter CORE_ID       = 0,
24      parameter USE_INITS     = 0,
25      parameter PARAM_DEFAULTS_R0 = 0,
26      parameter PARAM_DEFAULTS_R1 = 0,
27      parameter PARAM_DEFAULTS_R2 = 0,
28      parameter PARAM_DEFAULTS_R3 = 0,
29      parameter NAME          = "BRAM"
30  ) (
31      input clk,
32      input reset,
33
34      input       [`clog2(MEM_DEPTH)-1:0] mem_addr,
35      input       [MEM_WIDTH-1:0]         mem_in,
36      input                               mem_we,
37      output reg [MEM_WIDTH-1:0]          mem_out
38  );
39      // memory vector
40      (* ram_style = "block" *)
41      reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];
42
43      // not synthesizable
44      integer i;
45      initial begin
46          for (i = 0; i < MEM_DEPTH; i = i + 1) mem[i] = 0;
47          mem[0] = PARAM_DEFAULTS_R0;
48          mem[1] = PARAM_DEFAULTS_R1;
49          mem[2] = PARAM_DEFAULTS_R2;
50          mem[3] = PARAM_DEFAULTS_R3;
51
52          if (USE_INITS) begin
53              //`define TEST_SW
54              `ifdef TEST_SW
55              $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\verilog_memh.txt", mem);
56              `endif
57
58              `define TEST_ASM
59              `ifdef TEST_ASM
60              $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\asm.s.hex", mem);
61              `endif
62
63              //`define TEST_COND
64              `ifdef TEST_COND
65              mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
66              mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
67              `endif
68
69              //`define TEST_CMP
70              `ifdef TEST_CMP
71              mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0A};
72              mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h0B};
73              mem[2] = {`VMICRO16_OP_CMP,     3'h1, 3'h0, 5'h1};
74              `endif
75
76              //`define TEST_LWEX
77              `ifdef TEST_LWEX
78              mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC5};
79              mem[1] = {`VMICRO16_OP_SW,      3'h0, 3'h0, 5'h1};
80              mem[2] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h1};
81              mem[3] = {`VMICRO16_OP_LWEX,    3'h2, 3'h0, 5'h1};
```

```verilog
82              mem[4] = {`VMICRO16_OP_SWEX,      3'h3, 3'h0, 5'h1};
83          `endif
84
85          //`define TEST_MULTICORE
86          `ifdef TEST_MULTICORE
87              mem[0] = {`VMICRO16_OP_MOVI,      3'h0, 8'h90};
88              mem[1] = {`VMICRO16_OP_MOVI,      3'h1, 8'h33};
89              mem[2] = {`VMICRO16_OP_SW,        3'h1, 3'h0, 5'h0};
90              mem[3] = {`VMICRO16_OP_MOVI,      3'h0, 8'h80};
91              mem[4] = {`VMICRO16_OP_LW,        3'h2, 3'h0, 5'h0};
92              mem[5] = {`VMICRO16_OP_MOVI,      3'h1, 8'h33};
93              mem[6] = {`VMICRO16_OP_MOVI,      3'h1, 8'h33};
94              mem[7] = {`VMICRO16_OP_MOVI,      3'h1, 8'h33};
95              mem[8] = {`VMICRO16_OP_MOVI,      3'h0, 8'h91};
96              mem[9] = {`VMICRO16_OP_SW,        3'h2, 3'h0, 5'h0};
97          `endif
98
99          //`define TEST_BR
100         `ifdef TEST_BR
101             mem[0] = {`VMICRO16_OP_MOVI,      3'h0, 8'h0};
102             mem[1] = {`VMICRO16_OP_MOVI,      3'h3, 8'h3};
103             mem[2] = {`VMICRO16_OP_MOVI,      3'h1, 8'h2};
104             mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
105             mem[4] = {`VMICRO16_OP_BR,        3'h3, `VMICRO16_OP_BR_U};
106             mem[5] = {`VMICRO16_OP_MOVI,      3'h0, 8'hFF};
107         `endif
108
109         //`define ALL_TEST
110         `ifdef ALL_TEST
111         // Standard all test
112         // REGS0
113             mem[0] = {`VMICRO16_OP_MOVI,      3'h0, 8'h81};
114             mem[1] = {`VMICRO16_OP_SW,        3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
115             mem[2] = {`VMICRO16_OP_SW,        3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
116         // GPIO0
117             mem[3] = {`VMICRO16_OP_MOVI,      3'h0, 8'h90};
118             mem[4] = {`VMICRO16_OP_MOVI,      3'h1, 8'hD};
119             mem[5] = {`VMICRO16_OP_SW,        3'h1, 3'h0, 5'h0};
120             mem[6] = {`VMICRO16_OP_LW,        3'h2, 3'h0, 5'h0};
121         // TIM0
122             mem[7] = {`VMICRO16_OP_MOVI,      3'h0, 8'h07};
123             mem[8] = {`VMICRO16_OP_LW,        3'h3, 3'h0, 5'h03};
124         // UART0
125             mem[9]  = {`VMICRO16_OP_MOVI,     3'h0, 8'hA0};       // UART0
126             mem[10] = {`VMICRO16_OP_MOVI,     3'h1, 8'h41};       // ascii A
127             mem[11] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
128             mem[12] = {`VMICRO16_OP_MOVI,     3'h1, 8'h42}; // ascii B
129             mem[13] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
130             mem[14] = {`VMICRO16_OP_MOVI,     3'h1, 8'h43}; // ascii C
131             mem[15] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
132             mem[16] = {`VMICRO16_OP_MOVI,     3'h1, 8'h44}; // ascii D
133             mem[17] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
134             mem[18] = {`VMICRO16_OP_MOVI,     3'h1, 8'h45}; // ascii D
135             mem[19] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
136             mem[20] = {`VMICRO16_OP_MOVI,     3'h1, 8'h46}; // ascii E
137             mem[21] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
138         // BRAM0
139             mem[22] = {`VMICRO16_OP_MOVI,     3'h0, 8'hC0};
140             mem[23] = {`VMICRO16_OP_MOVI,     3'h1, 8'hA};
141             mem[24] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h5};
142             mem[25] = {`VMICRO16_OP_LW,       3'h2, 3'h0, 5'h5};
143         // GPIO1 (SSD 24-bit port)
144             mem[26] = {`VMICRO16_OP_MOVI,     3'h0, 8'h91};
145             mem[27] = {`VMICRO16_OP_MOVI,     3'h1, 8'h12};
146             mem[28] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
147             mem[29] = {`VMICRO16_OP_LW,       3'h2, 3'h0, 5'h0};
148         // GPIO2
149             mem[30] = {`VMICRO16_OP_MOVI,     3'h0, 8'h92};
150             mem[31] = {`VMICRO16_OP_MOVI,     3'h1, 8'h56};
151             mem[32] = {`VMICRO16_OP_SW,       3'h1, 3'h0, 5'h0};
152         `endif
153
154         //`define TEST_BRAM
155         `ifdef TEST_BRAM
156         // 2 core BRAM0 test
157             mem[0] = {`VMICRO16_OP_MOVI,      3'h0, 8'hC0};
158             mem[1] = {`VMICRO16_OP_MOVI,      3'h1, 8'hA};
159             mem[2] = {`VMICRO16_OP_SW,        3'h1, 3'h0, 5'h5};
160             mem[3] = {`VMICRO16_OP_LW,        3'h2, 3'h0, 5'h5};
161         `endif
162         end
163     end
164
165     always @(posedge clk) begin
166         // synchronous WRITE_FIRST (page 13)
167         if (mem_we) begin
168             mem[mem_addr] <= mem_in;
169             $display($time, "\t\t%s[%h] <= %h",
170                 NAME, mem_addr, mem_in);
171         end else
172             mem_out <= mem[mem_addr];
173     end
174
175     // TODO: Reset impl = every clock while reset is asserted, clear each cell
176     //       one at a time, mem[i++] <= 0
177 endmodule
178
179
180 module vmicro16_core_mmu # (
181     parameter MEM_WIDTH     = 16,
182     parameter MEM_DEPTH     = 64,
183
184     parameter CORE_ID       = 3'h0,
185     parameter CORE_ID_BITS  = `clog2(`CORES)
186 ) (
187     input clk,
188     input reset,
189
190     input  req,
191     output busy,
192
193     // From core
194     input      [MEM_WIDTH-1:0]  mmu_addr,
195     input      [MEM_WIDTH-1:0]  mmu_in,
196     input                       mmu_we,
197     input                       mmu_lwex,
```

```verilog
198        input                           mmu_swex,
199        output reg [MEM_WIDTH-1:0]  mmu_out,
200
201        // interrupts
202        output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
203        output reg [`DEF_NUM_INT-1:0]           ints_mask,
204
205        // TO APB interconnect
206        output reg [`APB_WIDTH-1:0]  M_PADDR,
207        output reg                   M_PWRITE,
208        output reg                   M_PSELx,
209        output reg                   M_PENABLE,
210        output reg [MEM_WIDTH-1:0]   M_PWDATA,
211        // from interconnect
212        input      [MEM_WIDTH-1:0]   M_PRDATA,
213        input                        M_PREADY
214    );
215        localparam MMU_STATE_T1  = 0;
216        localparam MMU_STATE_T2  = 1;
217        localparam MMU_STATE_T3  = 2;
218        reg [1:0]  mmu_state     = MMU_STATE_T1;
219
220        reg  [MEM_WIDTH-1:0] per_out = 0;
221        wire [MEM_WIDTH-1:0] tim0_out;
222
223        assign busy = req || (mmu_state == MMU_STATE_T2);
224
225        // more luts than below but easier
226        //wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
227        //             && (mmu_addr <= `DEF_MMU_TIM0_E);
228        //wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
229        //             && (mmu_addr <= `DEF_MMU_SREG_E);
230        //wire intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
231        //             && (mmu_addr <= `DEF_MMU_INTSV_E);
232        //wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
233        //             && (mmu_addr <= `DEF_MMU_INTSM_E);
234
235        wire tim0_en = ~mmu_addr[12] && ~mmu_addr[9] && ~mmu_addr[7];
236        wire sreg_en = mmu_addr[7] && ~mmu_addr[4] && ~mmu_addr[5];
237        wire intv_en = mmu_addr[8] && ~mmu_addr[3];
238        wire intm_en = mmu_addr[8] && mmu_addr[3];
239
240        wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
241        wire tim0_we   = (tim0_en && mmu_we);
242        wire intv_we   = (intv_en && mmu_we);
243        wire intm_we   = (intm_en && mmu_we);
244
245        // Special register selects
246        localparam SPECIAL_REGS = 8;
247        wire [MEM_WIDTH-1:0] sr_val;
248
249        // Interrupt vector and mask
250        initial ints_vector = 0;
251        initial ints_mask   = 0;
252        wire [2:0] intv_addr = mmu_addr[`clog2(`DEF_NUM_INT)-1:0];
253        always @(posedge clk)
254            if (intv_we)
255                ints_vector[intv_addr*`DATA_WIDTH +: `DATA_WIDTH] <= mmu_in;
256
257        always @(posedge clk)
258            if (intm_we)
259                ints_mask <= mmu_in;
260
261
262        always @(ints_vector)
263            $display($time,
264                    "\tC%d\t\tints_vector W: | %h %h %h %h | %h %h %h %h |",
265                     CORE_ID,
266            ints_vector[0*`DATA_WIDTH +: `DATA_WIDTH],
267            ints_vector[1*`DATA_WIDTH +: `DATA_WIDTH],
268            ints_vector[2*`DATA_WIDTH +: `DATA_WIDTH],
269            ints_vector[3*`DATA_WIDTH +: `DATA_WIDTH],
270            ints_vector[4*`DATA_WIDTH +: `DATA_WIDTH],
271            ints_vector[5*`DATA_WIDTH +: `DATA_WIDTH],
272            ints_vector[6*`DATA_WIDTH +: `DATA_WIDTH],
273            ints_vector[7*`DATA_WIDTH +: `DATA_WIDTH]
274            );
275
276        always @(intm_we)
277            $display($time, "\tC%d\t\tintm_we W: %b", CORE_ID, ints_mask);
278
279        // Output port
280        always @(*)
281            if      (tim0_en) mmu_out = tim0_out;
282            else if (sreg_en) mmu_out = sr_val;
283            else if (intv_en) mmu_out = ints_vector[mmu_addr[2:0]*`DATA_WIDTH
284                                                +: `DATA_WIDTH];
285            else if (intm_en) mmu_out = ints_mask;
286            else              mmu_out = per_out;
287
288        // APB master to slave interface
289        always @(posedge clk)
290            if (reset) begin
291                mmu_state <= MMU_STATE_T1;
292                M_PENABLE <= 0;
293                M_PADDR   <= 0;
294                M_PWDATA  <= 0;
295                M_PSELx   <= 0;
296                M_PWRITE  <= 0;
297            end
298            else
299                casex (mmu_state)
300                    MMU_STATE_T1: begin
301                        if (req && apb_en) begin
302                            M_PADDR   <= {mmu_lwex,
303                                          mmu_swex,
304                                          CORE_ID[CORE_ID_BITS-1:0],
305                                          mmu_addr[MEM_WIDTH-1:0]};
306
307                            M_PWDATA  <= mmu_in;
308                            M_PSELx   <= 1;
309                            M_PWRITE  <= mmu_we;
310
311                            mmu_state <= MMU_STATE_T2;
312                        end
313                    end
```

```
314
315                    `ifdef FIX_T3
316                        MMU_STATE_T2: begin
317                            M_PENABLE <= 1;
318
319                            if (M_PREADY == 1'b1) begin
320                                mmu_state <= MMU_STATE_T3;
321                            end
322                        end
323
324                        MMU_STATE_T3: begin
325                            // Slave has output a ready signal (finished)
326                            M_PENABLE <= 0;
327                            M_PADDR   <= 0;
328                            M_PWDATA  <= 0;
329                            M_PSELx   <= 0;
330                            M_PWRITE  <= 0;
331                            // Clock the peripheral output into a reg,
332                            //   to output on the next clock cycle
333                            per_out   <= M_PRDATA;
334
335                            mmu_state <= MMU_STATE_T1;
336                        end
337                    `else
338                        // No FIX_T3
339                        MMU_STATE_T2: begin
340                            if (M_PREADY == 1'b1) begin
341                                M_PENABLE <= 0;
342                                M_PADDR   <= 0;
343                                M_PWDATA  <= 0;
344                                M_PSELx   <= 0;
345                                M_PWRITE  <= 0;
346                                // Clock the peripheral output into a reg,
347                                //   to output on the next clock cycle
348                                per_out   <= M_PRDATA;
349
350                                mmu_state <= MMU_STATE_T1;
351                            end else begin
352                                M_PENABLE <= 1;
353                            end
354                        end
355                    `endif
356                endcase
357
358        (* ram_style = "block" *)
359        vmicro16_bram # (
360            .MEM_WIDTH  (MEM_WIDTH),
361            .MEM_DEPTH  (SPECIAL_REGS),
362            .USE_INITS  (0),
363            .PARAM_DEFAULTS_R0  (CORE_ID),
364            .PARAM_DEFAULTS_R1  (`CORES),
365            .PARAM_DEFAULTS_R2  (`APB_BRAM0_CELLS),
366            .PARAM_DEFAULTS_R3  (`SLAVES),
367            .NAME       ("ram_sr")
368        ) ram_sr (
369            .clk        (clk),
370            .reset      (reset),
371            .mem_addr   (mmu_addr[`clog2(SPECIAL_REGS)-1:0]),
372            .mem_in     (),
373            .mem_we     (),
374            .mem_out    (sr_val)
375        );
376
377        // Each M core has a TIM0 scratch memory
378        (* ram_style = "block" *)
379        vmicro16_bram # (
380            .MEM_WIDTH  (MEM_WIDTH),
381            .MEM_DEPTH  (MEM_DEPTH),
382            .USE_INITS  (0),
383            .NAME       ("TIM0")
384        ) TIM0 (
385            .clk        (clk),
386            .reset      (reset),
387            .mem_addr   (mmu_addr[7:0]),
388            .mem_in     (mmu_in),
389            .mem_we     (tim0_we),
390            .mem_out    (tim0_out)
391        );
392    endmodule
393
394
395
396    module vmicro16_regs # (
397        parameter CELL_WIDTH        = 16,
398        parameter CELL_DEPTH        = 8,
399        parameter CELL_SEL_BITS     = `clog2(CELL_DEPTH),
400        parameter CELL_DEFAULTS     = 0,
401        parameter DEBUG_NAME        = "",
402        parameter CORE_ID           = 0,
403        parameter PARAM_DEFAULTS_R0 = 16'h0000,
404        parameter PARAM_DEFAULTS_R1 = 16'h0000
405    ) (
406        input clk,
407        input reset,
408        // Dual port register reads
409        input       [CELL_SEL_BITS-1:0]  rs1, // port 1
410        output      [CELL_WIDTH-1   :0]  rd1,
411        //input       [CELL_SEL_BITS-1:0]  rs2, // port 2
412        //output      [CELL_WIDTH-1   :0]  rd2,
413        // EX/WB final stage write back
414        input                            we,
415        input [CELL_SEL_BITS-1:0]        ws1,
416        input [CELL_WIDTH-1:0]           wd
417    );
418        (* ram_style = "distributed" *)
419        reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;
420
421        // Initialise registers with default values
422        //   Really only used for special registers used by the soc
423        // TODO: How to do this on reset?
424        integer i;
425        initial
426            if (CELL_DEFAULTS)
427                $readmemh(CELL_DEFAULTS, regs);
428            else begin
429                for(i = 0; i < CELL_DEPTH; i = i + 1)
```

```
430                    regs[i] = 0;
431                regs[0] = PARAM_DEFAULTS_R0;
432                regs[1] = PARAM_DEFAULTS_R1;
433                end
434
435        `ifdef ICARUS
436            always @(regs)
437                $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
438                    CORE_ID,
439                    regs[0], regs[1], regs[2], regs[3],
440                    regs[4], regs[5], regs[6], regs[7]);
441        `endif
442
443        always @(posedge clk)
444            if (reset) begin
445                for(i = 0; i < CELL_DEPTH; i = i + 1)
446                    regs[i] <= 0;
447                regs[0] <= PARAM_DEFAULTS_R0;
448                regs[1] <= PARAM_DEFAULTS_R1;
449            end
450            else if (we) begin
451                $display($time, "\tC%02h: REGS #%s: Writing %h to reg[%d]",
452                    CORE_ID, DEBUG_NAME, wd, ws1);
453
454                // Perform the write
455                regs[ws1] <= wd;
456            end
457
458        // sync writes, async reads
459        assign rd1 = regs[rs1];
460        //assign rd2 = regs[rs2];
461    endmodule
462
463    module vmicro16_dec # (
464        parameter INSTR_WIDTH    = 16,
465        parameter INSTR_OP_WIDTH = 5,
466        parameter INSTR_RS_WIDTH = 3,
467        parameter ALU_OP_WIDTH   = 5
468    ) (
469        //input clk,   // not used yet (all combinational)
470        //input reset, // not used yet (all combinational)
471
472        input  [INSTR_WIDTH-1:0]   instr,
473
474        output [INSTR_OP_WIDTH-1:0] opcode,
475        output [INSTR_RS_WIDTH-1:0] rd,
476        output [INSTR_RS_WIDTH-1:0] ra,
477        output [3:0]                imm4,
478        output [7:0]                imm8,
479        output [11:0]               imm12,
480        output [4:0]                simm5,
481
482        // This can be freely increased without affecting the isa
483        output reg [ALU_OP_WIDTH-1:0] alu_op,
484
485        output reg has_imm4,
486        output reg has_imm8,
487        output reg has_imm12,
488        output reg has_we,
489        output reg has_br,
490        output reg has_mem,
491        output reg has_mem_we,
492        output reg has_cmp,
493
494        output halt,
495        output intr,
496
497        output reg has_lwex,
498        output reg has_swex
499
500        // TODO: Use to identify bad instruction and
501        //       raise exceptions
502        //,output     is_bad
503    );
504        assign opcode = instr[15:11];
505        assign rd     = instr[10:8];
506        assign ra     = instr[7:5];
507        assign imm4   = instr[3:0];
508        assign imm8   = instr[7:0];
509        assign imm12  = instr[11:0];
510        assign simm5  = instr[4:0];
511
512        // exme_op
513        always @(*) case (opcode)
514            `VMICRO16_OP_SPCL: casez(instr[11:0])
515                `VMICRO16_OP_SPCL_NOP,
516                `VMICRO16_OP_SPCL_HALT,
517                `VMICRO16_OP_SPCL_INTR:   alu_op = `VMICRO16_ALU_NOP;
518                default:                  alu_op = `VMICRO16_ALU_NOP; endcase
519
520            `VMICRO16_OP_LW:              alu_op = `VMICRO16_ALU_LW;
521            `VMICRO16_OP_SW:              alu_op = `VMICRO16_ALU_SW;
522            `VMICRO16_OP_LWEX:            alu_op = `VMICRO16_ALU_LW;
523            `VMICRO16_OP_SWEX:            alu_op = `VMICRO16_ALU_SW;
524
525            `VMICRO16_OP_MOV:             alu_op = `VMICRO16_ALU_MOV;
526            `VMICRO16_OP_MOVI:            alu_op = `VMICRO16_ALU_MOVI;
527
528            `VMICRO16_OP_BR:              alu_op = `VMICRO16_ALU_BR;
529            `VMICRO16_OP_MULT:            alu_op = `VMICRO16_ALU_MULT;
530
531            `VMICRO16_OP_CMP:             alu_op = `VMICRO16_ALU_CMP;
532            `VMICRO16_OP_SETC:            alu_op = `VMICRO16_ALU_SETC;
533
534            `VMICRO16_OP_BIT:     casez (simm5)
535                `VMICRO16_OP_BIT_OR:      alu_op = `VMICRO16_ALU_BIT_OR;
536                `VMICRO16_OP_BIT_XOR:     alu_op = `VMICRO16_ALU_BIT_XOR;
537                `VMICRO16_OP_BIT_AND:     alu_op = `VMICRO16_ALU_BIT_AND;
538                `VMICRO16_OP_BIT_NOT:     alu_op = `VMICRO16_ALU_BIT_NOT;
539                `VMICRO16_OP_BIT_LSHFT:   alu_op = `VMICRO16_ALU_BIT_LSHFT;
540                `VMICRO16_OP_BIT_RSHFT:   alu_op = `VMICRO16_ALU_BIT_RSHFT;
541                default:                  alu_op = `VMICRO16_ALU_BAD; endcase
542
543            `VMICRO16_OP_ARITH_U:     casez (simm5)
544                `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
545                `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
```

```
546                   `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
547                   default:                  alu_op = `VMICRO16_ALU_BAD; endcase
548
549           `VMICRO16_OP_ARITH_S:     casez (simm5)
550                   `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
551                   `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
552                   `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
553                   default:                  alu_op = `VMICRO16_ALU_BAD; endcase
554
555           default: begin
556                                             alu_op = `VMICRO16_ALU_NOP;
557                   $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
558               end
559       endcase
560
561       // Special opcodes
562       //assign nop  == ((opcode == `VMICRO16_OP_SPCL) & (~instr[0]));
563       assign halt = ((opcode == `VMICRO16_OP_SPCL) &   instr[0]);
564       assign intr = ((opcode == `VMICRO16_OP_SPCL) &   instr[1]);
565
566       // Register writes
567       always @(*) case (opcode)
568           `VMICRO16_OP_LWEX,
569           `VMICRO16_OP_SWEX,
570           `VMICRO16_OP_LW,
571           `VMICRO16_OP_MOV,
572           `VMICRO16_OP_MOVI,
573           //`VMICRO16_OP_MOVI_L,
574           `VMICRO16_OP_ARITH_U,
575           `VMICRO16_OP_ARITH_S,
576           `VMICRO16_OP_SETC,
577           `VMICRO16_OP_BIT,
578           `VMICRO16_OP_MULT:      has_we = 1'b1;
579           default:                has_we = 1'b0;
580       endcase
581
582       // Contains 4-bit immediate
583       always @(*)
584           if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
585               ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
586               has_imm4 = 1'b1;
587           else
588               has_imm4 = 1'b0;
589
590       // Contains 8-bit immediate
591       always @(*) case (opcode)
592           `VMICRO16_OP_MOVI,
593           `VMICRO16_OP_BR:        has_imm8 = 1'b1;
594           default:                has_imm8 = 1'b0;
595       endcase
596
597       //// Contains 12-bit immediate
598       //always @(*) case (opcode)
599       //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
600       //    default:                has_imm12 = 1'b0;
601       //endcase
602
603       // Will branch the pc
604       always @(*) case (opcode)
605           `VMICRO16_OP_BR:    has_br = 1'b1;
606           default:            has_br = 1'b0;
607       endcase
608
609       // Requires external memory
610       always @(*) case (opcode)
611           `VMICRO16_OP_LW,
612           `VMICRO16_OP_SW,
613           `VMICRO16_OP_LWEX,
614           `VMICRO16_OP_SWEX:  has_mem = 1'b1;
615           default:            has_mem = 1'b0;
616       endcase
617
618       // Requires external memory write
619       always @(*) case (opcode)
620           `VMICRO16_OP_SW,
621           `VMICRO16_OP_SWEX:  has_mem_we = 1'b1;
622           default:            has_mem_we = 1'b0;
623       endcase
624
625       // Affects status registers (cmp instructions)
626       always @(*) case (opcode)
627           `VMICRO16_OP_CMP:   has_cmp = 1'b1;
628           default:            has_cmp = 1'b0;
629       endcase
630
631       // Performs exclusive checks
632       always @(*) case (opcode)
633           `VMICRO16_OP_LWEX:   has_lwex = 1'b1;
634           default:             has_lwex = 1'b0;
635       endcase
636
637       always @(*) case (opcode)
638           `VMICRO16_OP_SWEX:   has_swex = 1'b1;
639           default:             has_swex = 1'b0;
640       endcase
641   endmodule
642
643
644   module vmicro16_alu # (
645       parameter OP_WIDTH   = 5,
646       parameter DATA_WIDTH = 16,
647       parameter CORE_ID    = 0
648   ) (
649       // input clk, // TODO: make clocked
650
651       input      [OP_WIDTH-1:0]   op,
652       input      [DATA_WIDTH-1:0] a, // rs1/dst
653       input      [DATA_WIDTH-1:0] b, // rs2
654       input      [3:0]            flags,
655       output reg [DATA_WIDTH-1:0] c
656   );
657       localparam TOP_BIT = (DATA_WIDTH-1);
658       // 17-bit register
659       reg [DATA_WIDTH:0] cmp_tmp = 0; // = {carry, [15:0]}
660       wire r_setc;
661
```

```verilog
662        always @(*) begin
663                        cmp_tmp = 0;
664                        case (op)
665            // branch/nop, output nothing
666            `VMICRO16_ALU_BR,
667            `VMICRO16_ALU_NOP:          c = {DATA_WIDTH{1'b0}};
668            // load/store addresses (use value in rd2)
669            `VMICRO16_ALU_LW,
670            `VMICRO16_ALU_SW:           c = b;
671            // bitwise operations
672            `VMICRO16_ALU_BIT_OR:       c = a | b;
673            `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
674            `VMICRO16_ALU_BIT_AND:      c = a & b;
675            `VMICRO16_ALU_BIT_NOT:      c = ~(b);
676            `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
677            `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;
678
679            `VMICRO16_ALU_MOV:          c = b;
680            `VMICRO16_ALU_MOVI:         c = b;
681            `VMICRO16_ALU_MOVI_L:       c = b;
682
683            `VMICRO16_ALU_ARITH_UADD:   c = a + b;
684            `VMICRO16_ALU_ARITH_USUB:   c = a - b;
685            // TODO: ALU should have simm5 as input
686            `VMICRO16_ALU_ARITH_UADDI:  c = a + b;
687
688            `ifdef DEF_ALU_HW_MULT
689                `VMICRO16_ALU_MULT:  c = a * b;
690            `endif
691
692            `VMICRO16_ALU_ARITH_SADD:   c = $signed(a) + $signed(b);
693            `VMICRO16_ALU_ARITH_SSUB:   c = $signed(a) - $signed(b);
694            // TODO: ALU should have simm5 as input
695            `VMICRO16_ALU_ARITH_SSUBI:  c = $signed(a) - $signed(b);
696
697            `VMICRO16_ALU_CMP: begin
698                // TODO: Do a-b in 17-bit register
699                //       Set zero, overflow, carry, signed bits in result
700                cmp_tmp = a - b;
701                c = 0;
702
703                // N   Negative condition code flag
704                // Z   Zero condition code flag
705                // C   Carry condition code flag
706                // V   Overflow condition code flag
707                c[`VMICRO16_SFLAG_N] = cmp_tmp[TOP_BIT];
708                c[`VMICRO16_SFLAG_Z] = (cmp_tmp == 0);
709                c[`VMICRO16_SFLAG_C] = 0; //cmp_tmp[TOP_BIT+1]; // not used
710
711                // Overflow flag
712                // https://stackoverflow.com/questions/30957188/
713                // https://github.com/bend1/prco304/blob/master/prco_core/rtl/prco_alu.v#L50
714                case(cmp_tmp[TOP_BIT+1:TOP_BIT])
715                    2'b01:   c[`VMICRO16_SFLAG_V] = 1;
716                    2'b10:   c[`VMICRO16_SFLAG_V] = 1;
717                    default: c[`VMICRO16_SFLAG_V] = 0;
718                endcase
719
720                $display($time, "\tC%02h: ALU CMP: %h %h = %h = %b", CORE_ID, a, b, cmp_tmp, c[3:0]);
721            end
722
723            `VMICRO16_ALU_SETC: c = { {15{1'b0}}, r_setc };
724
725            // TODO: Parameterise
726            default: begin
727                $display($time, "\tALU: unknown op: %h", op);
728                c       = 0;
729                cmp_tmp = 0;
730            end
731                    endcase
732                    end
733
734        branch setc_check (
735            .flags     (flags),
736            .cond      (b[7:0]),
737            .en        (r_setc)
738        );
739    endmodule
740
741    // flags = 4 bit r_cmp_flags register
742    // cond  = 8 bit VMICRO16_OP_BR_? value. See vmicro16_isa.v
743    module branch (
744        input [3:0] flags,
745        input [7:0] cond,
746        output reg  en
747    );
748        always @(*)
749            case (cond)
750                `VMICRO16_OP_BR_U:  en = 1; `VMICRO16_OP_BR_U:  en = 1;
751                `VMICRO16_OP_BR_E:  en = (flags[`VMICRO16_SFLAG_Z] == 1);
752                `VMICRO16_OP_BR_NE: en = (flags[`VMICRO16_SFLAG_Z] == 0);
753                `VMICRO16_OP_BR_G:  en = (flags[`VMICRO16_SFLAG_Z] == 0) &&
754                                        (flags[`VMICRO16_SFLAG_N] == flags[`VMICRO16_SFLAG_V]);
755                `VMICRO16_OP_BR_L:  en = (flags[`VMICRO16_SFLAG_Z] != flags[`VMICRO16_SFLAG_N]);
756                `VMICRO16_OP_BR_GE: en = (flags[`VMICRO16_SFLAG_Z] == flags[`VMICRO16_SFLAG_N]);
757                `VMICRO16_OP_BR_LE: en = (flags[`VMICRO16_SFLAG_Z] == 1) ||
758                                        (flags[`VMICRO16_SFLAG_N] != flags[`VMICRO16_SFLAG_V]);
759                default:            en = 0;
760            endcase
761    endmodule
762
763
764
765    module vmicro16_core # (
766        parameter DATA_WIDTH        = 16,
767        parameter MEM_INSTR_DEPTH   = 64,
768        parameter MEM_SCRATCH_DEPTH = 64,
769        parameter MEM_WIDTH         = 16,
770
771        parameter CORE_ID           = 3'h0
772    ) (
773        input       clk,
774        input       reset,
775
776        output [7:0] dbug,
777
```

```verilog
778        output         halt,
779
780        // interrupt sources
781        input  [`DEF_NUM_INT-1:0]              ints,
782        input  [`DEF_NUM_INT*`DATA_WIDTH-1:0]  ints_data,
783        output [`DEF_NUM_INT-1:0]              ints_ack,
784
785        // APB master to slave interface (apb_intercon)
786        output [`APB_WIDTH-1:0]    w_PADDR,
787        output                     w_PWRITE,
788        output                     w_PSELx,
789        output                     w_PENABLE,
790        output [DATA_WIDTH-1:0]    w_PWDATA,
791        input  [DATA_WIDTH-1:0]    w_PRDATA,
792        input                      w_PREADY
793
794  `ifndef DEF_CORE_HAS_INSTR_MEM
795        , // APB master interface to slave instruction memory
796        output reg [`APB_WIDTH-1:0]    w2_PADDR,
797        output reg                     w2_PWRITE,
798        output reg                     w2_PSELx,
799        output reg                     w2_PENABLE,
800        output reg [DATA_WIDTH-1:0]    w2_PWDATA,
801        input      [DATA_WIDTH-1:0]    w2_PRDATA,
802        input                          w2_PREADY
803  `endif
804  );
805        localparam STATE_IF = 0;
806        localparam STATE_R1 = 1;
807        localparam STATE_R2 = 2;
808        localparam STATE_ME = 3;
809        localparam STATE_WB = 4;
810        localparam STATE_FE = 5;
811        localparam STATE_IDLE = 6;
812        localparam STATE_HALT = 7;
813        reg  [2:0] r_state = STATE_IF;
814
815        reg  [DATA_WIDTH-1:0] r_pc          = 16'h0000;
816        reg  [DATA_WIDTH-1:0] r_pc_saved    = 16'h0000;
817        reg  [DATA_WIDTH-1:0] r_instr       = 16'h0000;
818        wire [DATA_WIDTH-1:0] w_mem_instr_out;
819        wire                 w_halt;
820
821        assign dbug = {7'h00, w_halt};
822        assign halt = w_halt;
823
824        wire [4:0]           r_instr_opcode;
825        wire [4:0]           r_instr_alu_op;
826        wire [2:0]           r_instr_rsd;
827        wire [2:0]           r_instr_rsa;
828        reg  [DATA_WIDTH-1:0] r_instr_rdd = 0;
829        reg  [DATA_WIDTH-1:0] r_instr_rda = 0;
830        wire [3:0]           r_instr_imm4;
831        wire [7:0]           r_instr_imm8;
832        wire [4:0]           r_instr_simm5;
833        wire                 r_instr_has_imm4;
834        wire                 r_instr_has_imm8;
835        wire                 r_instr_has_we;
836        wire                 r_instr_has_br;
837        wire                 r_instr_has_cmp;
838        wire                 r_instr_has_mem;
839        wire                 r_instr_has_mem_we;
840        wire                 r_instr_halt;
841        wire                 r_instr_has_lwex;
842        wire                 r_instr_has_swex;
843
844        wire [DATA_WIDTH-1:0] r_alu_out;
845
846        wire [DATA_WIDTH-1:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
847        wire [DATA_WIDTH-1:0] r_mem_scratch_in   = r_instr_rdd;
848        wire [DATA_WIDTH-1:0] r_mem_scratch_out;
849        wire                 r_mem_scratch_we    = r_instr_has_mem_we && (r_state == STATE_ME);
850        reg                  r_mem_scratch_req  = 0;
851        wire                 r_mem_scratch_busy;
852
853        reg  [2:0]           r_reg_rs1 = 0;
854        wire [DATA_WIDTH-1:0] r_reg_rd1_s;
855        wire [DATA_WIDTH-1:0] r_reg_rd1_i;
856        wire [DATA_WIDTH-1:0] r_reg_rd1 = regs_use_int ? r_reg_rd1_i : r_reg_rd1_s;
857        //wire [15:0] r_reg_rd2;
858        wire [DATA_WIDTH-1:0] r_reg_wd = (r_instr_has_mem) ? r_mem_scratch_out : r_alu_out;
859        wire                 r_reg_we = r_instr_has_we && (r_state == STATE_WB);
860
861        // branching
862        wire         w_intr;
863        wire         w_branch_en;
864        wire         w_branching   = r_instr_has_br && w_branch_en;
865        reg  [3:0]   r_cmp_flags   = 4'h00; // N, Z, C, V
866
867        always @(r_cmp_flags)
868            $display($time, "\tC%02h:\tALU CMP: %b", CORE_ID, r_cmp_flags);
869
870        // 2 cycle register fetch
871        always @(*) begin
872            r_reg_rs1 = 0;
873            if (r_state == STATE_R1)
874                r_reg_rs1 = r_instr_rsd;
875            else if (r_state == STATE_R2)
876                r_reg_rs1 = r_instr_rsa;
877            else
878                r_reg_rs1 = 3'h0;
879        end
880
881        reg regs_use_int = 0;
882  `ifdef DEF_ENABLE_INT
883        wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_vector;
884        wire [`DEF_NUM_INT-1:0]             ints_mask;
885        wire                                has_int = ints & ints_mask;
886        reg int_pending = 0;
887        reg int_pending_ack = 0;
888        always @(posedge clk)
889            if (int_pending_ack)
890                // We've now branched to the isr
891                int_pending <= 0;
892            else if (has_int)
893                // Notify fsm to switch to the ints_vector at the last stage
```

```verilog
894                     int_pending <= 1;
895                 else if (w_intr)
896                     // Return to Interrupt instruction called,
897                     //   so we've finished with the interrupt
898                     int_pending <= 0;
899             `endif
900
901         // Next program counter logic
902         reg [`DATA_WIDTH-1:0] next_pc = 0;
903         always @(posedge clk)
904             if (reset)
905                 r_pc <= 0;
906             else if (r_state == STATE_WB) begin
907                 `ifdef DEF_ENABLE_INT
908                 if (int_pending) begin
909                     $display($time, "\tC%02h: Jumping to ISR: %h",
910                         CORE_ID,
911                         ints_vector[0 +: `DATA_WIDTH]);
912                     // TODO: check bounds
913                     // Save state
914                     r_pc_saved      <= r_pc + 1;
915                     regs_use_int    <= 1;
916                     int_pending_ack <= 1;
917                     // Jump to ISR
918                     r_pc            <= ints_vector[0 +: `DATA_WIDTH];
919                 end else if (w_intr) begin
920                     $display($time, "\tC%02h: Returning from ISR: %h",
921                         CORE_ID, r_pc_saved);
922
923                     // Restore state
924                     r_pc            <= r_pc_saved;
925                     regs_use_int    <= 0;
926                     int_pending_ack <= 0;
927                 end else
928                 `endif
929                 if (w_branching) begin
930                     $display($time, "\tC%02h: branching to %h", CORE_ID, r_instr_rdd);
931                     r_pc            <= r_instr_rdd;
932
933                     `ifdef DEF_ENABLE_INT
934                         int_pending_ack <= 0;
935                     `endif
936                 end else if (r_pc < (MEM_INSTR_DEPTH-1)) begin
937                     // normal increment
938                     // pc <= pc + 1
939                     r_pc            <= r_pc + 1;
940
941                     `ifdef DEF_ENABLE_INT
942                         int_pending_ack <= 0;
943                     `endif
944                 end
945             end // end r_state == STATE_WB
946             else if (r_state == STATE_HALT) begin
947                 `ifdef DEF_ENABLE_INT
948                 // Only an interrupt can return from halt
949                 // duplicate code form STATE_ME!
950                 if (int_pending) begin
951                     $display($time, "\tC%02h: Jumping to ISR: %h", CORE_ID, ints_vector[0 +: `DATA_WIDTH]);
952                     // TODO: check bounds
953                     // Save state
954                     r_pc_saved      <= r_pc;// + 1; HALT = stay with same PC
955                     regs_use_int    <= 1;
956                     int_pending_ack <= 1;
957                     // Jump to ISR
958                     r_pc            <= ints_vector[0 +: `DATA_WIDTH];
959                 end else if (w_intr) begin
960                     $display($time, "\tC%02h: Returning from ISR: %h", CORE_ID, r_pc_saved);
961                     r_pc            <= r_pc_saved;
962                     regs_use_int    <= 0;
963                     int_pending_ack <= 0;
964                 end
965                 `endif
966             end
967
968     `ifndef DEF_CORE_HAS_INSTR_MEM
969         initial w2_PSELx   = 0;
970         initial w2_PENABLE = 0;
971         initial w2_PADDR   = 0;
972     `endif
973
974         // cpu state machine
975         always @(posedge clk)
976             if (reset) begin
977                 r_state           <= STATE_IF;
978                 r_instr           <= 0;
979                 r_mem_scratch_req <= 0;
980                 r_instr_rdd       <= 0;
981                 r_instr_rda       <= 0;
982             end
983             else begin
984
985     `ifdef DEF_CORE_HAS_INSTR_MEM
986             if (r_state == STATE_IF) begin
987                     r_instr <= w_mem_instr_out;
988
989                     $display("");
990                     $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
991                     $display($time, "\tC%02h: INSTR: %h", CORE_ID, w_mem_instr_out);
992
993                     r_state <= STATE_R1;
994             end
995     `else
996             // wait for global instruction rom to give us our instruction
997             if (r_state == STATE_IF) begin
998                 // wait for ready signal
999                 if (!w2_PREADY) begin
1000                    w2_PSELx   <= 1;
1001                    w2_PWRITE  <= 0;
1002                    w2_PENABLE <= 1;
1003                    w2_PWDATA  <= 0;
1004                    w2_PADDR   <= r_pc;
1005                end else begin
1006                    w2_PSELx   <= 0;
1007                    w2_PWRITE  <= 0;
1008                    w2_PENABLE <= 0;
1009                    w2_PWDATA  <= 0;
```

```verilog
1010
1011                        r_instr <= w2_PRDATA;
1012
1013                        $display("");
1014                        $display($time, "\tC%02h: PC: %h",     CORE_ID, r_pc);
1015                        $display($time, "\tC%02h: INSTR: %h", CORE_ID, w2_PRDATA);
1016
1017                        r_state <= STATE_R1;
1018                    end
1019                end
1020    `endif
1021
1022            else if (r_state == STATE_R1) begin
1023                if (w_halt) begin
1024                    $display("");
1025                    $display("");
1026                    $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
1027                    r_state <= STATE_HALT;
1028                end else begin
1029                    // primary operand
1030                    r_instr_rdd <= r_reg_rd1;
1031                    r_state     <= STATE_R2;
1032                end
1033            end
1034            else if (r_state == STATE_R2) begin
1035                // Choose secondary operand (register or immediate)
1036                if      (r_instr_has_imm8)  r_instr_rda <= r_instr_imm8;
1037                else if (r_instr_has_imm4)  r_instr_rda <= r_reg_rd1 + r_instr_imm4;
1038                else                        r_instr_rda <= r_reg_rd1;
1039
1040                if (r_instr_has_mem) begin
1041                    r_state           <= STATE_ME;
1042                    // Pulse req
1043                    r_mem_scratch_req <= 1;
1044                end else
1045                    r_state <= STATE_WB;
1046            end
1047            else if (r_state == STATE_ME) begin
1048                // Pulse req
1049                r_mem_scratch_req <= 0;
1050                // Wait for MMU to finish
1051                if (!r_mem_scratch_busy)
1052                    r_state <= STATE_WB;
1053            end
1054            else if (r_state == STATE_WB) begin
1055                if (r_instr_has_cmp) begin
1056                    $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
1057                    r_cmp_flags <= r_alu_out[3:0];
1058                end
1059
1060                r_state <= STATE_FE;
1061            end
1062            else if (r_state == STATE_FE)
1063                r_state <= STATE_IF;
1064            else if (r_state == STATE_HALT) begin
1065                `ifdef DEF_ENABLE_INT
1066                    if (int_pending) begin
1067                        r_state <= STATE_FE;
1068                    end
1069                `endif
1070            end
1071        end
1072
1073    `ifdef DEF_CORE_HAS_INSTR_MEM
1074        // Instruction ROM
1075        (* rom_style = "distributed" *)
1076        vmicro16_bram # (
1077            .MEM_WIDTH      (DATA_WIDTH),
1078            .MEM_DEPTH      (MEM_INSTR_DEPTH),
1079            .CORE_ID        (CORE_ID),
1080            .USE_INITS      (1),
1081            .NAME           ("INSTR_MEM")
1082        ) mem_instr (
1083            .clk            (clk),
1084            .reset          (reset),
1085            // port 1
1086            .mem_addr       (r_pc),
1087            .mem_in         (0),
1088            .mem_we         (1'b0),  // ROM
1089            .mem_out        (w_mem_instr_out)
1090        );
1091    `endif
1092
1093        // MMU
1094        vmicro16_core_mmu # (
1095            .MEM_WIDTH      (DATA_WIDTH),
1096            .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
1097            .CORE_ID        (CORE_ID)
1098        ) mmu (
1099            .clk            (clk),
1100            .reset          (reset),
1101            .req            (r_mem_scratch_req),
1102            .busy           (r_mem_scratch_busy),
1103            // interrupts
1104            .ints_vector    (ints_vector),
1105            .ints_mask      (ints_mask),
1106            // port 1
1107            .mmu_addr       (r_mem_scratch_addr),
1108            .mmu_in         (r_mem_scratch_in),
1109            .mmu_we         (r_mem_scratch_we),
1110            .mmu_lwex       (r_instr_has_lwex),
1111            .mmu_swex       (r_instr_has_swex),
1112            .mmu_out        (r_mem_scratch_out),
1113            // APB maste   r to slave
1114            .M_PADDR        (w_PADDR),
1115            .M_PWRITE       (w_PWRITE),
1116            .M_PSELx        (w_PSELx),
1117            .M_PENABLE      (w_PENABLE),
1118            .M_PWDATA       (w_PWDATA),
1119            .M_PRDATA       (w_PRDATA),
1120            .M_PREADY       (w_PREADY)
1121        );
1122
1123        // Instruction decoder
1124        vmicro16_dec dec (
1125            // input
```

```
1126              .instr          (r_instr),
1127              // output async
1128              .opcode         (),
1129              .rd             (r_instr_rsd),
1130              .ra             (r_instr_rsa),
1131              .imm4           (r_instr_imm4),
1132              .imm8           (r_instr_imm8),
1133              .imm12          (),
1134              .simm5          (r_instr_simm5),
1135              .alu_op         (r_instr_alu_op),
1136              .has_imm4       (r_instr_has_imm4),
1137              .has_imm8       (r_instr_has_imm8),
1138              .has_we         (r_instr_has_we),
1139              .has_br         (r_instr_has_br),
1140              .has_cmp        (r_instr_has_cmp),
1141              .has_mem        (r_instr_has_mem),
1142              .has_mem_we     (r_instr_has_mem_we),
1143              .halt           (w_halt),
1144              .intr           (w_intr),
1145              .has_lwex       (r_instr_has_lwex),
1146              .has_swex       (r_instr_has_swex)
1147          );
1148
1149          // Software registers
1150          vmicro16_regs # (
1151              .CORE_ID    (CORE_ID),
1152              .CELL_WIDTH (`DATA_WIDTH)
1153          ) regs (
1154              .clk        (clk),
1155              .reset      (reset),
1156              // async port 0
1157              .rs1        (r_reg_rs1),
1158              .rd1        (r_reg_rd1_s),
1159              // async port 1
1160              //.rs2        (),
1161              //.rd2        (),
1162              // write port
1163              .we         (r_reg_we && ~regs_use_int),
1164              .ws1        (r_instr_rsd),
1165              .wd         (r_reg_wd)
1166          );
1167
1168          // Interrupt replacement registers
1169          `ifdef DEF_ENABLE_INT
1170          vmicro16_regs # (
1171              .CORE_ID    (CORE_ID),
1172              .CELL_WIDTH (`DATA_WIDTH),
1173              .DEBUG_NAME ("REGSINT")
1174          ) regs_intr (
1175              .clk        (clk),
1176              .reset      (reset),
1177              // async port 0
1178              .rs1        (r_reg_rs1),
1179              .rd1        (r_reg_rd1_i),
1180              // async port 1
1181              //.rs2        (),
1182              //.rd2        (),
1183              // write port
1184              .we         (r_reg_we && regs_use_int),
1185              .ws1        (r_instr_rsd),
1186              .wd         (r_reg_wd)
1187          );
1188          `endif
1189
1190          // ALU
1191          vmicro16_alu # (
1192              .CORE_ID(CORE_ID)
1193          ) alu (
1194              .op         (r_instr_alu_op),
1195              .a          (r_instr_rdd),
1196              .b          (r_instr_rda),
1197              .flags      (r_cmp_flags),
1198              // async output
1199              .c          (r_alu_out)
1200          );
1201
1202          branch branch_check (
1203              .flags      (r_cmp_flags),
1204              .cond       (r_instr_imm8),
1205              .en         (w_branch_en)
1206          );
1207
1208      endmodule
```

vmicro16_soc.v

```
1   //
2   //
3
4   `include "vmicro16_soc_config.v"
5   `include "clog2.v"
6   `include "formal.v"
7
8   module pow_reset # (
9       parameter INIT  = 1,
10      parameter N     = 8
11  ) (
12      input       clk,
13      input       reset,
14      output reg  resethold
15  );
16      initial resethold = INIT ? (N-1) : 0;
17
18      always @(*)
19          resethold = |hold;
20
21      reg [`clog2(N)-1:0] hold = (N-1);
22      always @(posedge clk)
23          if (reset)
24              hold <= N-1;
```

```verilog
25              else
26                  if (hold)
27                      hold <= hold - 1;
28      endmodule
29
30      // Vmicro16 multi-core SoC with various peripherals
31      // and interrupts
32      module vmicro16_soc (
33          input clk,
34          input reset,
35
36          // UART0
37          input                       uart_rx,
38          output                      uart_tx,
39          //
40          output [`APB_GPIO0_PINS-1:0]    gpio0,
41          output [`APB_GPIO1_PINS-1:0]    gpio1,
42          output [`APB_GPIO2_PINS-1:0]    gpio2,
43          //
44          output                      halt,
45          //
46          output      [`CORES-1:0]        dbug0,
47          output      [`CORES*8-1:0]      dbug1
48      );
49          wire [`CORES-1:0] w_halt;
50          assign halt = &w_halt;
51
52          assign dbug0 = w_halt;
53
54          // Watchdog reset pulse signal.
55          //   Passed to pow_reset to generate a longer reset pulse
56          wire wdreset;
57          wire prog_prog;
58
59          // soft register reset hold for brams and registers
60          wire soft_reset;
61          `ifdef DEF_GLOBAL_RESET
62              pow_reset # (
63                  .INIT       (1),
64                  .N          (8)
65              ) por_inst (
66                  .clk        (clk),
67                  `ifdef DEF_USE_WATCHDOG
68                  .reset      (reset | wdreset | prog_prog),
69                  `else
70                  .reset      (reset),
71                  `endif
72                  .resethold  (soft_reset)
73              );
74          `else
75              assign soft_reset = 0;
76          `endif
77
78          // Peripherals (master to slave)
79          wire [`APB_WIDTH-1:0]       M_PADDR;
80          wire                       M_PWRITE;
81          wire [`SLAVES-1:0]         M_PSELx;  // not shared
82          wire                       M_PENABLE;
83          wire [`DATA_WIDTH-1:0]     M_PWDATA;
84          wire [`SLAVES*`DATA_WIDTH-1:0] M_PRDATA; // input to intercon
85          wire [`SLAVES-1:0]         M_PREADY; // input
86
87          // Master apb interfaces
88          wire [`CORES*`APB_WIDTH-1:0]   w_PADDR;
89          wire [`CORES-1:0]             w_PWRITE;
90          wire [`CORES-1:0]             w_PSELx;
91          wire [`CORES-1:0]             w_PENABLE;
92          wire [`CORES*`DATA_WIDTH-1:0]  w_PWDATA;
93          wire [`CORES*`DATA_WIDTH-1:0]  w_PRDATA;
94          wire [`CORES-1:0]             w_PREADY;
95
96          // Interrupts
97      `ifdef DEF_ENABLE_INT
98          wire [`DEF_NUM_INT-1:0]               ints;
99          wire [`DEF_NUM_INT*`DATA_WIDTH-1:0]  ints_data;
100         assign ints[7:1] = 0;
101         assign ints_data[`DEF_NUM_INT*`DATA_WIDTH-1:`DATA_WIDTH] =
102                     {`DEF_NUM_INT*(`DATA_WIDTH-1){1'b0}};
103     `endif
104
105         apb_intercon_s # (
106             .MASTER_PORTS   (`CORES),
107             .SLAVE_PORTS    (`SLAVES),
108             .BUS_WIDTH      (`APB_WIDTH),
109             .DATA_WIDTH     (`DATA_WIDTH),
110             .HAS_PSELX_ADDR (1)
111         ) apb (
112             .clk        (clk),
113             .reset      (soft_reset),
114             // APB master to slave
115             .S_PADDR    (w_PADDR),
116             .S_PWRITE   (w_PWRITE),
117             .S_PSELx    (w_PSELx),
118             .S_PENABLE  (w_PENABLE),
119             .S_PWDATA   (w_PWDATA),
120             .S_PRDATA   (w_PRDATA),
121             .S_PREADY   (w_PREADY),
122             // shared bus
123             .M_PADDR    (M_PADDR),
124             .M_PWRITE   (M_PWRITE),
125             .M_PSELx    (M_PSELx),
126             .M_PENABLE  (M_PENABLE),
127             .M_PWDATA   (M_PWDATA),
128             .M_PRDATA   (M_PRDATA),
129             .M_PREADY   (M_PREADY)
130         );
131
132     `ifdef DEF_USE_WATCHDOG
133         vmicro16_watchdog_apb # (
134             .BUS_WIDTH  (`APB_WIDTH),
135             .NAME       ("WDOG0")
136         ) wdog0_apb (
137             .clk        (clk),
138             .reset      (),
139             // apb slave to master interface
140             .S_PADDR    (),
```

```
141             .S_PWRITE   (M_PWRITE),
142             .S_PSELx    (M_PSELx[`APB_PSELX_WDOG0]),
143             .S_PENABLE  (M_PENABLE),
144             .S_PWDATA   (),
145             .S_PRDATA   (),
146             .S_PREADY   (M_PREADY[`APB_PSELX_WDOG0]),
147
148             .wdreset    (wdreset)
149         );
150     `endif
151
152         vmicro16_gpio_apb # (
153             .BUS_WIDTH  (`APB_WIDTH),
154             .DATA_WIDTH (`DATA_WIDTH),
155             .PORTS      (`APB_GPIO0_PINS),
156             .NAME       ("GPIO0")
157         ) gpio0_apb (
158             .clk        (clk),
159             .reset      (soft_reset),
160             // apb slave to master interface
161             .S_PADDR    (M_PADDR),
162             .S_PWRITE   (M_PWRITE),
163             .S_PSELx    (M_PSELx[`APB_PSELX_GPIO0]),
164             .S_PENABLE  (M_PENABLE),
165             .S_PWDATA   (M_PWDATA),
166             .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO0*`DATA_WIDTH +: `DATA_WIDTH]),
167             .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
168             .gpio       (gpio0)
169         );
170
171         // GPIO1 for Seven segment displays (16 pin)
172         vmicro16_gpio_apb # (
173             .BUS_WIDTH  (`APB_WIDTH),
174             .DATA_WIDTH (`DATA_WIDTH),
175             .PORTS      (`APB_GPIO1_PINS),
176             .NAME       ("GPIO1")
177         ) gpio1_apb (
178             .clk        (clk),
179             .reset      (soft_reset),
180             // apb slave to master interface
181             .S_PADDR    (M_PADDR),
182             .S_PWRITE   (M_PWRITE),
183             .S_PSELx    (M_PSELx[`APB_PSELX_GPIO1]),
184             .S_PENABLE  (M_PENABLE),
185             .S_PWDATA   (M_PWDATA),
186             .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO1*`DATA_WIDTH +: `DATA_WIDTH]),
187             .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
188             .gpio       (gpio1)
189         );
190
191         // GPIO2 for Seven segment displays (8 pin)
192         vmicro16_gpio_apb # (
193             .BUS_WIDTH  (`APB_WIDTH),
194             .DATA_WIDTH (`DATA_WIDTH),
195             .PORTS      (`APB_GPIO2_PINS),
196             .NAME       ("GPIO2")
197         ) gpio2_apb (
198             .clk        (clk),
199             .reset      (soft_reset),
200             // apb slave to master interface
201             .S_PADDR    (M_PADDR),
202             .S_PWRITE   (M_PWRITE),
203             .S_PSELx    (M_PSELx[`APB_PSELX_GPIO2]),
204             .S_PENABLE  (M_PENABLE),
205             .S_PWDATA   (M_PWDATA),
206             .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO2*`DATA_WIDTH +: `DATA_WIDTH]),
207             .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
208             .gpio       (gpio2)
209         );
210
211         apb_uart_tx # (
212             .DATA_WIDTH (8),
213             .ADDR_EXP   (4) //2^^4 = 16 FIFO words
214         ) uart0_apb (
215             .clk        (clk),
216             .reset      (soft_reset),
217             // apb slave to master interface
218             .S_PADDR    (M_PADDR),
219             .S_PWRITE   (M_PWRITE),
220             .S_PSELx    (M_PSELx[`APB_PSELX_UART0]),
221             .S_PENABLE  (M_PENABLE),
222             .S_PWDATA   (M_PWDATA),
223             .S_PRDATA   (M_PRDATA[`APB_PSELX_UART0*`DATA_WIDTH +: `DATA_WIDTH]),
224             .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
225             // uart wires
226             .tx_wire    (uart_tx),
227             .rx_wire    ()
228         );
229
230         timer_apb timr0 (
231             .clk        (clk),
232             .reset      (soft_reset),
233             // apb slave to master interface
234             .S_PADDR    (M_PADDR),
235             .S_PWRITE   (M_PWRITE),
236             .S_PSELx    (M_PSELx[`APB_PSELX_TIMR0]),
237             .S_PENABLE  (M_PENABLE),
238             .S_PWDATA   (M_PWDATA),
239             .S_PRDATA   (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
240             .S_PREADY   (M_PREADY[`APB_PSELX_TIMR0])
241             //
242     `ifdef DEF_ENABLE_INT
243             ,.out       (ints    [`DEF_INT_TIMR0]),
244             .int_data   (ints_data[`DEF_INT_TIMR0*`DATA_WIDTH +: `DATA_WIDTH])
245     `endif
246         );
247
248         // Shared register set for system-on-chip info
249         // R0 = number of cores
250         vmicro16_regs_apb # (
251             .BUS_WIDTH          (`APB_WIDTH),
252             .DATA_WIDTH         (`DATA_WIDTH),
253             .CELL_DEPTH         (8),
254             .PARAM_DEFAULTS_R0  (`CORES),
255             .PARAM_DEFAULTS_R1  (`SLAVES)
256         ) regs0_apb (
```

```verilog
257            .clk        (clk),
258            .reset      (soft_reset),
259            // apb slave to master interface
260            .S_PADDR    (M_PADDR),
261            .S_PWRITE   (M_PWRITE),
262            .S_PSELx    (M_PSELx[`APB_PSELX_REGS0]),
263            .S_PENABLE  (M_PENABLE),
264            .S_PWDATA   (M_PWDATA),
265            .S_PRDATA   (M_PRDATA[`APB_PSELX_REGS0*`DATA_WIDTH +: `DATA_WIDTH]),
266            .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
267        );
268
269        vmicro16_bram_ex_apb # (
270            .BUS_WIDTH   (`APB_WIDTH),
271            .MEM_WIDTH   (`DATA_WIDTH),
272            .MEM_DEPTH   (`APB_BRAM0_CELLS),
273            .CORE_ID_BITS (`clog2(`CORES))
274        ) bram_apb (
275            .clk        (clk),
276            .reset      (soft_reset),
277            // apb slave to master interface
278            .S_PADDR    (M_PADDR),
279            .S_PWRITE   (M_PWRITE),
280            .S_PSELx    (M_PSELx[`APB_PSELX_BRAM0]),
281            .S_PENABLE  (M_PENABLE),
282            .S_PWDATA   (M_PWDATA),
283            .S_PRDATA   (M_PRDATA[`APB_PSELX_BRAM0*`DATA_WIDTH +: `DATA_WIDTH]),
284            .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
285        );
286
287        // There must be atleast 1 core
288        `static_assert(`CORES > 0)
289        `static_assert(`DEF_MEM_INSTR_DEPTH > 0)
290        `static_assert(`DEF_MMU_TIMO_CELLS > 0)
291
292
293        // Single instruction memory
294    `ifndef DEF_CORE_HAS_INSTR_MEM
295        // slave input/outputs from interconnect
296        wire [`APB_WIDTH-1:0]        instr_M_PADDR;
297        wire                         instr_M_PWRITE;
298        wire [1-1:0]                 instr_M_PSELx;   // not shared
299        wire                         instr_M_PENABLE;
300        wire [`DATA_WIDTH-1:0]       instr_M_PWDATA;
301        wire [1*`DATA_WIDTH-1:0]     instr_M_PRDATA; // slave response
302        wire [1-1:0]                 instr_M_PREADY; // slave response
303
304        // Master apb interfaces
305        wire [`CORES*`APB_WIDTH-1:0]  instr_w_PADDR;
306        wire [`CORES-1:0]             instr_w_PWRITE;
307        wire [`CORES-1:0]             instr_w_PSELx;
308        wire [`CORES-1:0]             instr_w_PENABLE;
309        wire [`CORES*`DATA_WIDTH-1:0] instr_w_PWDATA;
310        wire [`CORES*`DATA_WIDTH-1:0] instr_w_PRDATA;
311        wire [`CORES-1:0]             instr_w_PREADY;
312
313        `ifdef DEF_USE_REPROG
314            wire [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] prog_addr;
315            wire [`DATA_WIDTH-1:0] prog_data;
316            wire prog_we;
317            uart_prog rom_prog (
318                .clk        (clk),
319                .reset      (reset | wdreset),
320                // input stream
321                .uart_rx    (uart_rx),
322                // programmer
323                .addr       (prog_addr),
324                .data       (prog_data),
325                .we         (prog_we),
326                .prog       (prog_prog)
327            );
328        `endif
329
330        `ifdef DEF_USE_REPROG
331            vmicro16_bram_prog_apb
332        `else
333            vmicro16_bram_apb
334        `endif
335        # (
336            .BUS_WIDTH     (`APB_WIDTH),
337            .MEM_WIDTH     (`DATA_WIDTH),
338            .MEM_DEPTH     (`DEF_MEM_INSTR_DEPTH),
339            .USE_INITS     (1),
340            .NAME          ("INSTR_ROM_G")
341        ) instr_rom_apb (
342            .clk        (clk),
343            .reset      (reset),
344            .S_PADDR    (instr_M_PADDR),
345            .S_PWRITE   (0),
346            .S_PSELx    (instr_M_PSELx),
347            .S_PENABLE  (instr_M_PENABLE),
348            .S_PWDATA   (0),
349            .S_PRDATA   (instr_M_PRDATA),
350            .S_PREADY   (instr_M_PREADY)
351
352            `ifdef DEF_USE_REPROG
353                ,
354                .addr       (prog_addr),
355                .data       (prog_data),
356                .we         (prog_we),
357                .prog       (prog_prog)
358            `endif
359        );
360
361        apb_intercon_s # (
362            .MASTER_PORTS  (`CORES),
363            .SLAVE_PORTS   (1),
364            .BUS_WIDTH     (`APB_WIDTH),
365            .DATA_WIDTH    (`DATA_WIDTH),
366            .HAS_PSELX_ADDR (0)
367        ) apb_instr_intercon (
368            .clk        (clk),
369            .reset      (soft_reset),
370            // APB master from cores
371            // master
372            .S_PADDR    (instr_w_PADDR),
```

```verilog
373            .S_PWRITE    (instr_w_PWRITE),
374            .S_PSELx     (instr_w_PSELx),
375            .S_PENABLE  (instr_w_PENABLE),
376            .S_PWDATA   (instr_w_PWDATA),
377            .S_PRDATA   (instr_w_PRDATA),
378            .S_PREADY   (instr_w_PREADY),
379            // shared bus slaves
380            // slave outputs
381            .M_PADDR    (instr_M_PADDR),
382            .M_PWRITE   (instr_M_PWRITE),
383            .M_PSELx    (instr_M_PSELx),
384            .M_PENABLE  (instr_M_PENABLE),
385            .M_PWDATA   (instr_M_PWDATA),
386            .M_PRDATA   (instr_M_PRDATA),
387            .M_PREADY   (instr_M_PREADY)
388        );
389  `endif
390
391        genvar i;
392        generate for(i = 0; i < `CORES; i = i + 1) begin : cores
393
394            vmicro16_core # (
395                .CORE_ID            (i),
396                .DATA_WIDTH         (`DATA_WIDTH),
397
398                .MEM_INSTR_DEPTH    (`DEF_MEM_INSTR_DEPTH),
399                .MEM_SCRATCH_DEPTH  (`DEF_MMU_TIM0_CELLS)
400            ) c1 (
401                .clk        (clk),
402                .reset      (soft_reset),
403
404                // debug
405                .halt       (w_halt[i]),
406
407                // interrupts
408                .ints       (ints),
409                .ints_data  (ints_data),
410
411                // Output master port 1
412                .w_PADDR    (w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
413                .w_PWRITE   (w_PWRITE  [i]                           ),
414                .w_PSELx    (w_PSELx   [i]                           ),
415                .w_PENABLE  (w_PENABLE [i]                           ),
416                .w_PWDATA   (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
417                .w_PRDATA   (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
418                .w_PREADY   (w_PREADY  [i]                           )
419
420  `ifndef DEF_CORE_HAS_INSTR_MEM
421                // APB instruction rom
422                , // Output master port 2
423                .w2_PADDR   (instr_w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
424                //.w2_PWRITE  (instr_w_PWRITE  [i]                           ),
425                .w2_PSELx    (instr_w_PSELx   [i]                           ),
426                .w2_PENABLE (instr_w_PENABLE [i]                           ),
427                //.w2_PWDATA  (instr_w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
428                .w2_PRDATA  (instr_w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
429                .w2_PREADY  (instr_w_PREADY  [i]                           )
430  `endif
431            );
432        end
433        endgenerate
434
435
436        ////////////////////////////////////////////////////
437        // Formal Verification
438        ////////////////////////////////////////////////////
439        `ifdef FORMAL
440        wire all_halted = &w_halt;
441        ////////////////////////////////////////////////////
442        // Count number of clocks each core is spending on
443        //   bus transactions
444        ////////////////////////////////////////////////////
445        reg [15:0] bus_core_times       [0:`CORES-1];
446        reg [15:0] core_work_times      [0:`CORES-1];
447        reg [15:0] instr_fetch_times    [0:`CORES-1];
448        integer i2;
449        initial
450            for(i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
451                bus_core_times[i2] = 0;
452                core_work_times[i2] = 0;
453            end
454
455        // total bus time
456        generate
457            genvar g2;
458            for (g2 = 0; g2 < `CORES; g2 = g2 + 1) begin : formal_for_times
459                always @(posedge clk) begin
460                    if (w_PSELx[g2])
461                        bus_core_times[g2] <= bus_core_times[g2] + 1;
462
463                    // Core working time
464                    `ifndef DEF_CORE_HAS_INSTR_MEM
465                    if (!w_PSELx[g2] && !instr_w_PSELx[g2])
466                    `else
467                    if (!w_PSELx[g2])
468                    `endif
469                        if (!w_halt[g2])
470                            core_work_times[g2] <= core_work_times[g2] + 1;
471
472                end
473            end
474        endgenerate
475
476        reg [15:0] bus_time_average = 0;
477        reg [15:0] bus_reqs_average = 0;
478        reg [15:0] fetch_time_average = 0;
479        reg [15:0] work_time_average = 0;
480        //
481        always @(all_halted) begin
482            for (i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
483                bus_time_average   = bus_time_average   + bus_core_times[i2];
484                bus_reqs_average   = bus_reqs_average   + bus_core_reqs_count[i2];
485                work_time_average  = work_time_average  + core_work_times[i2];
486                fetch_time_average = fetch_time_average + instr_fetch_times[i2];
487            end
488
```

```verilog
489            bus_time_average   = bus_time_average   / `CORES;
490            bus_reqs_average   = bus_reqs_average   / `CORES;
491            work_time_average  = work_time_average  / `CORES;
492            fetch_time_average = fetch_time_average / `CORES;
493        end
494
495        //////////////////////////////////////////////////
496        // Count number of bus requests per core
497        //////////////////////////////////////////////////
498        // 1 clock delay of w_PSELx
499        reg [`CORES-1:0] bus_core_reqs_last;
500        // rising edges of each
501        wire [`CORES-1:0] bus_core_reqs_real;
502        // storage for counters for each core
503        reg [15:0] bus_core_reqs_count [0:`CORES-1];
504        initial
505            for(i2 = 0; i2 < `CORES; i2 = i2 + 1)
506                bus_core_reqs_count[i2] = 0;
507
508        // 1 clk delay to detect rising edge
509        always @(posedge clk)
510            bus_core_reqs_last <= w_PSELx;
511
512        generate
513            genvar g3;
514                for (g3 = 0; g3 < `CORES; g3 = g3 + 1) begin : formal_for_reqs
515                    // Detect new reqs for each core
516                    assign bus_core_reqs_real[g3] = w_PSELx[g3] >
517                                                       bus_core_reqs_last[g3];
518
519                    always @(posedge clk)
520                        if (bus_core_reqs_real[g3])
521                            bus_core_reqs_count[g3] <= bus_core_reqs_count[g3] + 1;
522
523                 end
524        endgenerate
525
526
527        `ifndef DEF_CORE_HAS_INSTR_MEM
528            //////////////////////////////////////////////////
529            // Time waiting for instruction fetches
530            //   from global  memory
531            //////////////////////////////////////////////////
532            integer i3;
533            initial
534                for(i3 = 0; i3 < `CORES; i3 = i3 + 1)
535                    instr_fetch_times[i3] = 0;
536
537            // total bus time
538            // Instruction fetches occur on the w2 master port
539            generate
540                genvar g4;
541                for (g4 = 0; g4 < `CORES; g4 = g4 + 1) begin : formal_for_fetch_times
542                    always @(posedge clk)
543                        if (instr_w_PSELx[g4])
544                            instr_fetch_times[g4] <= instr_fetch_times[g4] + 1;
545                end
546            endgenerate
547        `endif
548
549
550        `endif // end FORMAL
551
552    endmodule
```

vmicro16_isa.v

```verilog
1   // Vmicro16 multi-core instruction set
2   `include "vmicro16_soc_config.v"
3
4   // TODO: Remove NOP by making a register write/read always 0
5   `define VMICRO16_OP_SPCL          5'b00000
6   `define VMICRO16_OP_LW            5'b00001
7   `define VMICRO16_OP_SW            5'b00010
8   `define VMICRO16_OP_BIT          5'b00011
9   `define VMICRO16_OP_BIT_OR       5'b00000
10  `define VMICRO16_OP_BIT_XOR      5'b00001
11  `define VMICRO16_OP_BIT_AND      5'b00010
12  `define VMICRO16_OP_BIT_NOT      5'b00011
13  `define VMICRO16_OP_BIT_LSHFT    5'b00100
14  `define VMICRO16_OP_BIT_RSHFT    5'b00101
15  `define VMICRO16_OP_MOV          5'b00100
16  `define VMICRO16_OP_MOVI         5'b00101
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  `define VMICRO16_OP_CMP          5'b01001
27  `define VMICRO16_OP_SETC         5'b01010
28  `define VMICRO16_OP_MULT         5'b01011
29  `define VMICRO16_OP_LWEX         5'b01101
30  `define VMICRO16_OP_SWEX         5'b01110
31
32  // Special opcodes
33  `define VMICRO16_OP_SPCL_NOP      11'h000
34  `define VMICRO16_OP_SPCL_HALT     11'h001
35  `define VMICRO16_OP_SPCL_INTR     11'h002
36
37  // TODO: wasted upper nibble bits in BR
38  `define VMICRO16_OP_BR_U         8'h00
39  `define VMICRO16_OP_BR_E         8'h01
40  `define VMICRO16_OP_BR_NE        8'h02
41  `define VMICRO16_OP_BR_G         8'h03
42  `define VMICRO16_OP_BR_GE        8'h04
43  `define VMICRO16_OP_BR_L         8'h05
```

```verilog
44    `define VMICRO16_OP_BR_LE          8'h06
45    `define VMICRO16_OP_BR_S           8'h07
46    `define VMICRO16_OP_BR_NS          8'h08
47
48    // flag bit positions
49    `define VMICRO16_SFLAG_N           4'h03
50    `define VMICRO16_SFLAG_Z           4'h02
51    `define VMICRO16_SFLAG_C           4'h01
52    `define VMICRO16_SFLAG_V           4'h00
53
54    // microcode operations
55    `define VMICRO16_ALU_BIT_OR        5'h00
56    `define VMICRO16_ALU_BIT_XOR       5'h01
57    `define VMICRO16_ALU_BIT_AND       5'h02
58    `define VMICRO16_ALU_BIT_NOT       5'h03
59    `define VMICRO16_ALU_BIT_LSHFT     5'h04
60    `define VMICRO16_ALU_BIT_RSHFT     5'h05
61    `define VMICRO16_ALU_LW            5'h06
62    `define VMICRO16_ALU_SW            5'h07
63    `define VMICRO16_ALU_NOP           5'h08
64    `define VMICRO16_ALU_MOV           5'h09
65    `define VMICRO16_ALU_MOVI          5'h0a
66    `define VMICRO16_ALU_MOVI_L        5'h0b
67    `define VMICRO16_ALU_ARITH_UADD    5'h0c
68    `define VMICRO16_ALU_ARITH_USUB    5'h0d
69    `define VMICRO16_ALU_ARITH_SADD    5'h0e
70    `define VMICRO16_ALU_ARITH_SSUB    5'h0f
71    `define VMICRO16_ALU_BR_U          5'h10
72    `define VMICRO16_ALU_BR_E          5'h11
73    `define VMICRO16_ALU_BR_NE         5'h12
74    `define VMICRO16_ALU_BR_G          5'h13
75    `define VMICRO16_ALU_BR_GE         5'h14
76    `define VMICRO16_ALU_BR_L          5'h15
77    `define VMICRO16_ALU_BR_LE         5'h16
78    `define VMICRO16_ALU_BR_S          5'h17
79    `define VMICRO16_ALU_BR_NS         5'h18
80    `define VMICRO16_ALU_CMP           5'h19
81    `define VMICRO16_ALU_SETC          5'h1a
82    `define VMICRO16_ALU_ARITH_UADDI   5'h1b
83    `define VMICRO16_ALU_ARITH_SSUBI   5'h1c
84    `define VMICRO16_ALU_BR            5'h1d
85    `ifdef DEF_ALU_HW_MULT
86    `define VMICRO16_ALU_MULT          5'h1e
87    `endif
88    `define VMICRO16_ALU_BAD           5'h1f
```