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

ELEC5881M - Final 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

July 5, 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: July 5, 2019

Table of Contents

Chapter 1

Memory Mapping

The Vmicro16 processor uses a memory-mapping scheme to communicate with peripherals and other cores.

1.1 Memory Map

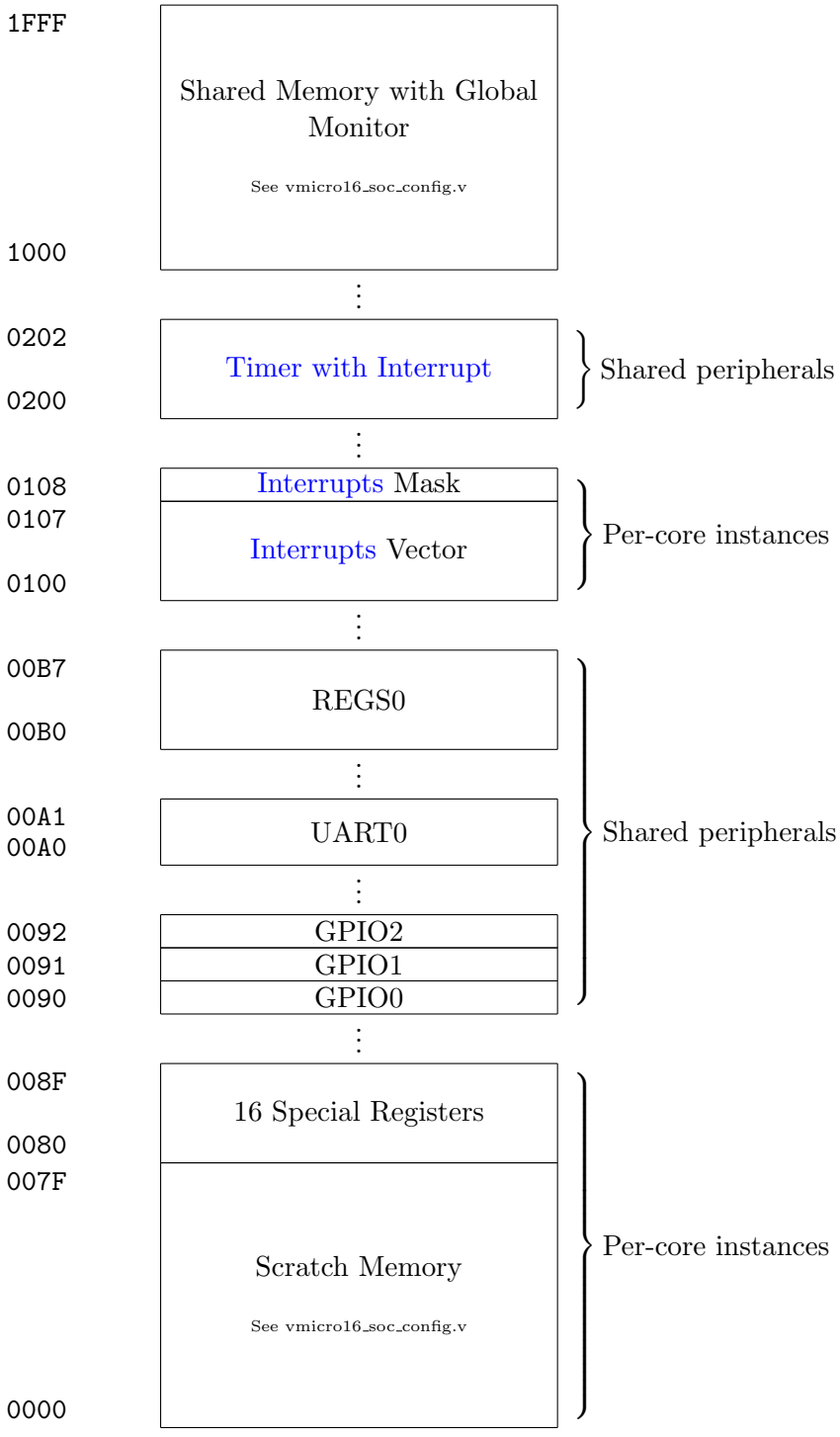


Figure 1.1: Memory map showing addresses of various memory sections.

1.2 Special Registers

From the software perspective, it is important for both the developer and software algorithms to know the target system's architecture to better utilise the resources available to them. Software written for one architecture with N cores must also run on an architecture with M cores. To enable such portability, the software must query the system for information such as: number of processor cores and the current core identifier. Without this information, the developer would be required to produce software for each individual architecture (e.g. an Intel i5 with 4 cores or an Intel i7 with 8 cores, or an NVIDIA GTX 970 with.

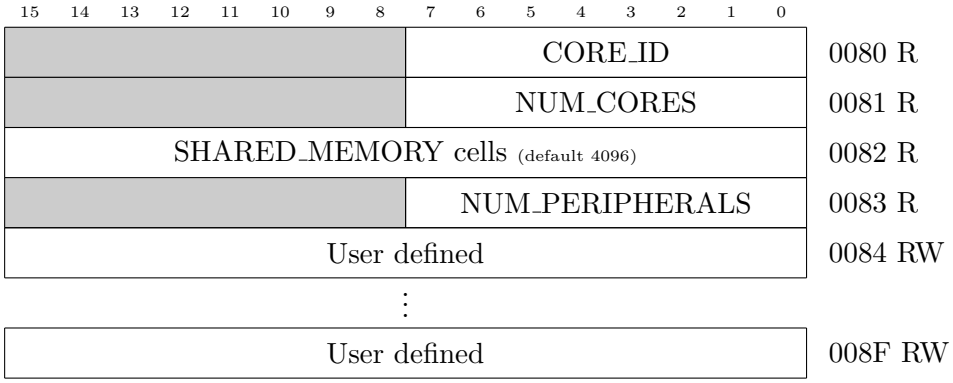


Figure 1.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

Chapter 2

Interrupts

This section describes the design, considerations, and implementation, of interrupt functionality within the Vmicro16 processor.

2.1 Why Interrupts?

Interrupts are used to enable asynchronous behaviour within a processor.

Interrupts are commonly used to signal actions from asynchronous sources, for example an input button or from a UART receiver signalling that data has been received.

2.2 Hardware Implementation

2.2.1 Context Switching

When acting upon an incoming interrupt the current state the processor must be saved so that changes from the interrupt handler, such as register writes and branches, do not affect the current state. After the interrupt handler function signals it has finished (by using the *Interrupt Return* `intr` instruction) the saved state is restored. In the case of the Vmicro16 processor, the program counter `r_pc[15:0]` and register set `regs` instance are the only states that are saved. Going forth, the terms *normal mode* and *interrupt mode* are used to describe what registers the processor should use when executing instructions.

When saving the state, to avoid clocking 128 bits (8 registers of 16 bits) into another register, which would increase timing delays and logic elements, a dedicated register set for the interrupt mode (`regs_isr`) is multiplexed with the normal mode register set (`regs`). Then depending on the mode (identified by the register `regs_use_int`) the processor can easily switch between the two large states without significantly affecting timing.

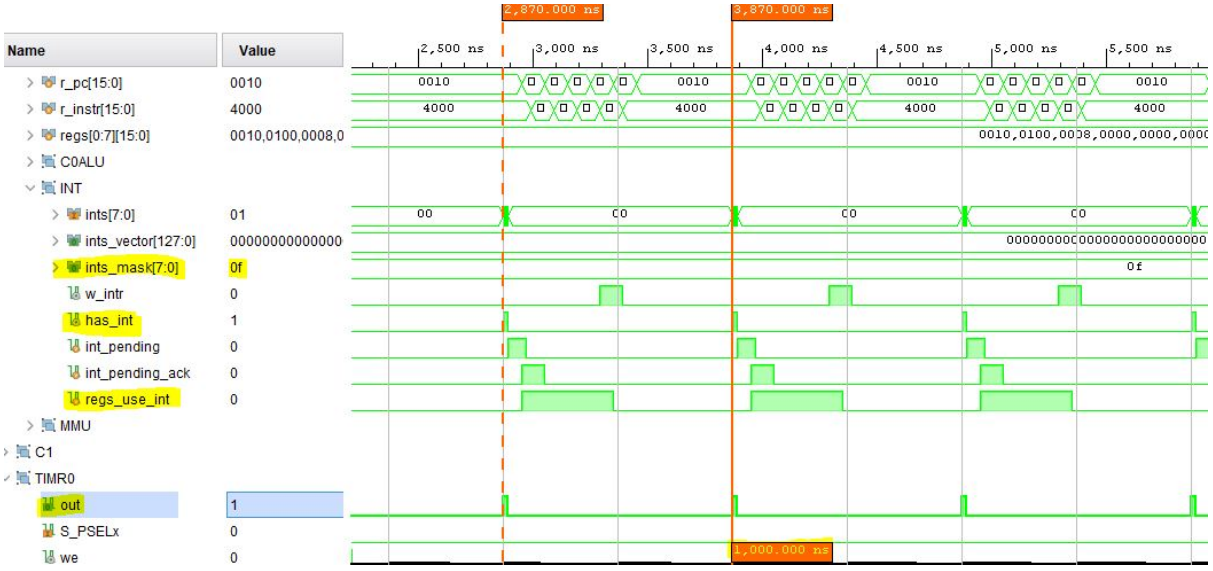


Figure 2.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return instruction (indicated by w_intr) the processor returns to normal mode and restores the normal state.

The timing diagram in Figure 2.1 visually describes this process.

2.3 Software Interface

To enable software to

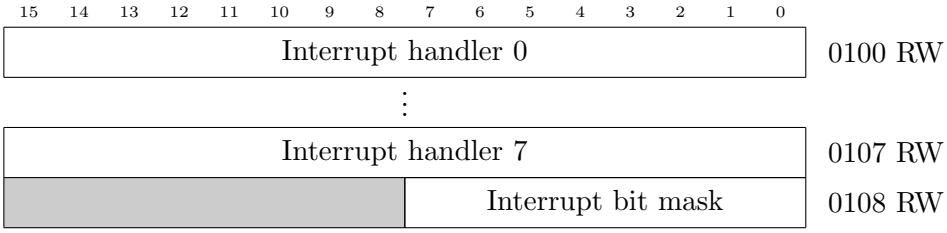


Figure 2.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

Interrupt Vector (0x0100-0x0107)

The interrupt vector is a per-core register that is used to store the addresses of interrupt handlers. An interrupt handler is simply a software function residing in instruction memory that is branched to when a particular interrupt is received.

Interrupt Mask (0x0108)

The interrupt mask is a per-core register that is used to mask/listen specific interrupt sources. This enables processing cores to individually select which interrupts they respond to. This allows for multi-processor designs where each core can be used for a particular interrupt source, improving the time response to the interrupt for time critical programs. The Interrupt Mask

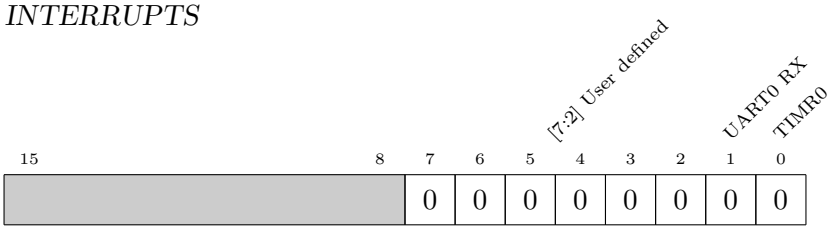


Figure 2.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

register is an 8-bit read/write register where each bit corresponds to a particular interrupt source and each bit corresponds with the interrupt handler in the interrupt vector.

2.3.1 Software Example

To better understand the usage of the described interrupt registers, a simple software program is described below. The following software program produces a simple and power efficient routine to initialise the interrupt vector and interrupt mask.

```
1  entry:
2      // Set interrupt vector at 0x100
3      // Move address of isr0 function to vector[0]
4      movi    r0, isr0
5      // create 0x100 value by left shifting 1 8 bits
6      movi    r1, #0x1
7      movi    r2, #0x8
8      lshft   r1, r2
9      // write isr0 address to vector[0]
10     sw      r0, r1
11
12     // enable all interrupts by writing 0x0f to 0x108
13     movi    r0, #0x0f
14     sw      r0, r1 + #0x8
15     halt                    // enter low power idle state
16
17 isr0:                       // arbitrary name
18     movi    r0, #0xff       // do something
19     intr                    // return from interrupt
```

A more complex example software program utilising interrupts and the TIMR0 interrupt is described in section **??**.

2.4 Design Improvements

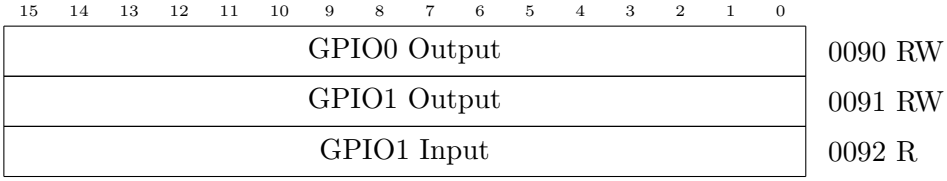
The hardware and software interrupt design have changed throughout the projects cycle. In initial versions of the interrupt implementation, the software program, while waiting for an interrupt, would be in a tight infinite loop (branching to the same instruction). This resulted in the processor using all pipeline stages during this time. The pipeline stages produce many logic transitions and memory fetches which raise power consumption and temperatures. This is quite noticeable especially when running on the Spartan-6 LX9 FPGA.

To improve this, it was decided to implement a new state within the processor's state machine that, when entered, did not produce high frequency logic transitions or memory fetches. The `HALT` instruction was modified to enter this state and the only way to leave is from an interrupt or top-level reset. This removes the need for a software infinite loop that produces high frequency logic transitions (decoding, ALU, register reads, etc.) and memory fetches.

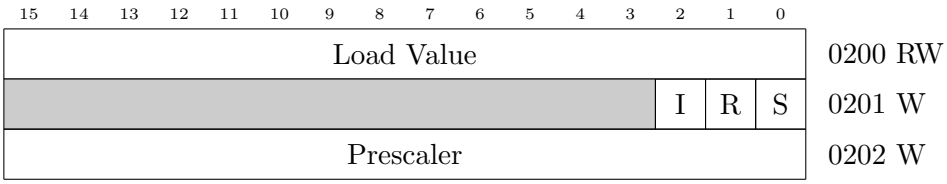
Chapter 3

Peripherals

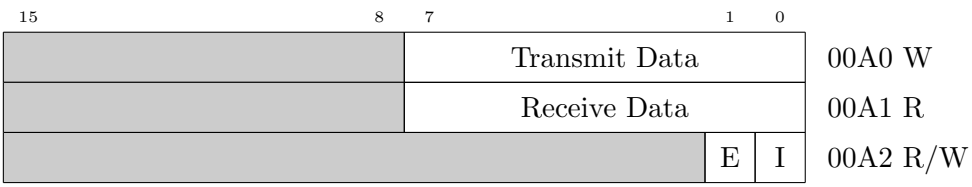
3.1 GPIO Interface



3.2 Timer with Interrupt



3.3 UART Interface



Chapter 4

System-on-Chip Layout

The Vmicro16 processor uses

Figure 4.1: •

Chapter 5

Interconnect

The Vmicro16 processor uses

Chapter 6

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.

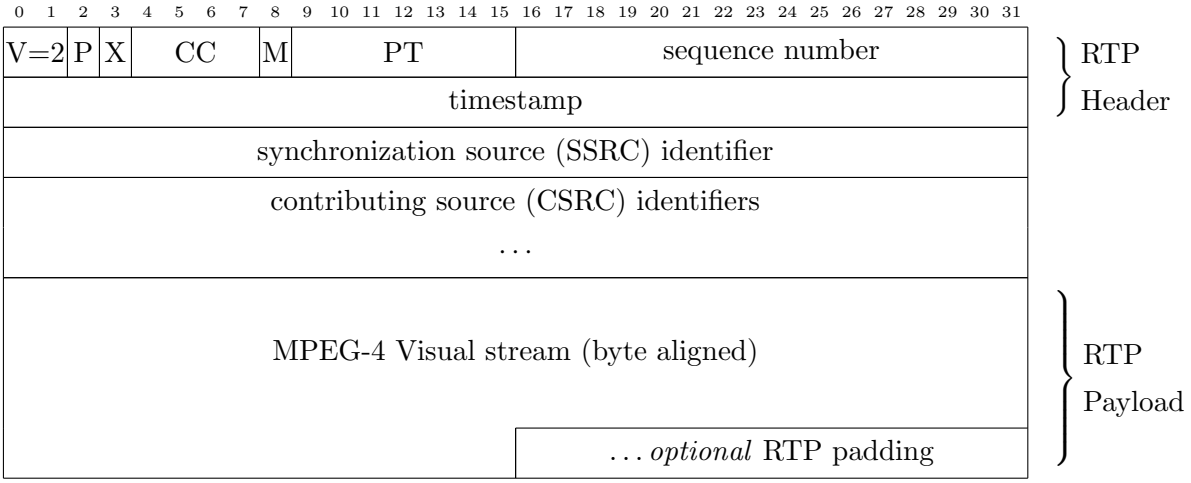


Figure 6.1: Foo

6.1 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.

6.2 Why RISC?

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

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

6.3 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 8.1).

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 7

Background

7.1 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:

$$S_p = \frac{1}{(1-p) + \frac{p}{n}} \tag{7.1}$$

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

7.2 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)* [3] in a master-slave control architecture.

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

As loosely coupled system's nodes feature 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 7.1 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.

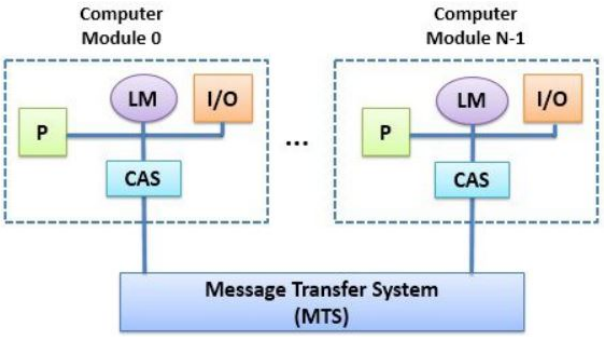


Figure 7.1: A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [3].

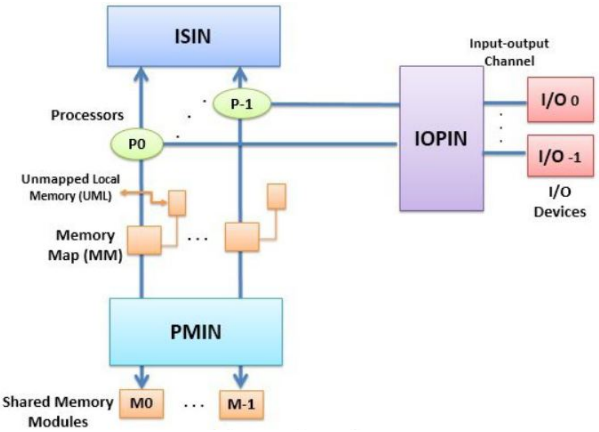


Figure 7.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [3].

This project will utilise a loosely coupled architecture due to 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.

7.3 Network-on-chip Architectures

Network-on-chip (NoC) architectures implement on-chip communication mechanisms that are based on network communication principles, such as routing, switching, and massive scalability [4]. NoC's can generally support hundreds to millions of processing cores. Figure 7.3 shows an example 16-core network-on-chip architecture. NoC's can scale to very large sizes while not sacrificing performance because each processor core is able to drive the network rather than needing to wait for a shared bus to become free before doing so.

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

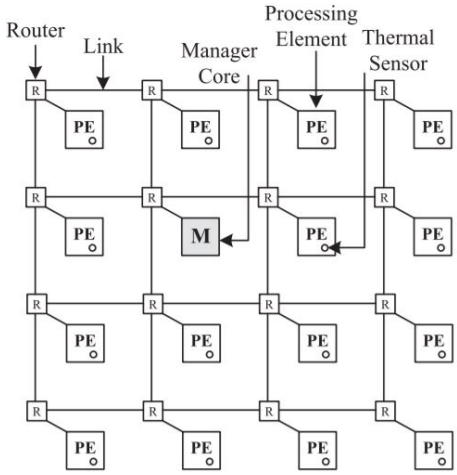


Figure 7.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [6].

Chapter 8

Project Overview

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

8.1 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.

8.1.1 Core Deliverables (CD)

The project's core deliverables are described below.

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

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

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

CD2 Design and implement a Verilog RISC core that implements the ISA in CD1.
The Verilog RISC core will be able to run software program written for the instruction set architecture.

CD3 Design and implement an on-chip interconnect for multi-core processing (2 to 32 cores) using the RISC core from CD2.
The interconnect will be a chief requirement to enable multi-core communication. The interconnect should support up to 32 cores, however FPGA implementation constraints may limit this due to limited resources.

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

CD4 Analyse performance of serial and parallel software algorithms, such as parallel DFT, on the processor.
To evaluate the effectiveness of the developed solution, a serial and parallel implementation of a simple computing algorithm (parallel reduction, sorting) will be ran on the processor and it's performance analysed. Effectiveness will be rated on total algorithm run-time and the speed-up gained by adding more cores.

CD5 Allow the RISC core to be easily compiled to multiple FPGA vendors (Xilinx, Altera).
The developed solution should be generic and portable to allow it to be used across a wide-range of FPGA vendors and devices.

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

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

8.1.2 Extended Deliverables (ED)

The project's extended deliverables are described below.

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

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

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

ED4 Design a compiler-backend for the PRCO304 [9] compiler to support the ISA from1 CD1. This will make it easier to build complex multi-core software for the processor.

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

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

ED7 Store instruction memory in SPI flash.

ED8 Reprogram instruction memory at runtime from host computer.

ED9 Processor external debugger using host-processor link.

8.2 Project Timeline

8.2.1 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 8.1.

8.2.2 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.

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 8.1: Project stages throughout the life cycle of the project.

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.

8.2.3 Timeline

The project stages from Table 8.1 are displayed below in a Gantt chart.

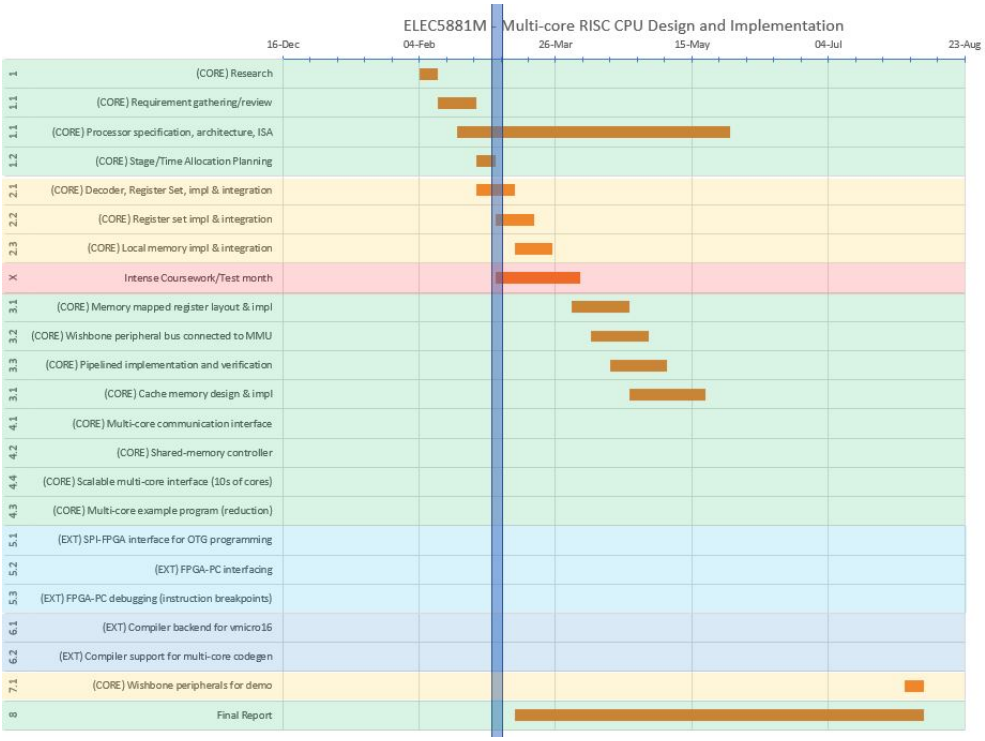


Figure 8.1: Project stages in a Gantt chart.

8.3 Resources

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

8.3.1 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 8.2 shows the peripherals (green) available to the FPGA.

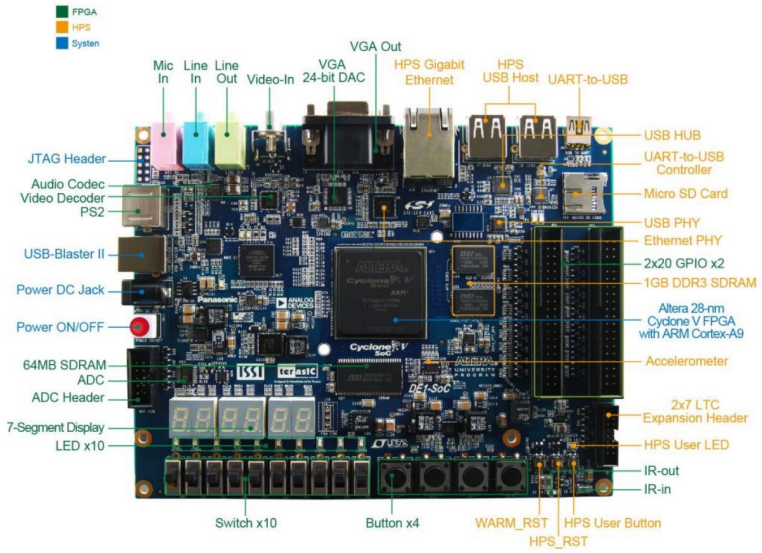


Figure 8.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [10].

Minispartan 6+ FPGA Development Board

The Minispartan 6+ is a hobbyist 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-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 8.3.

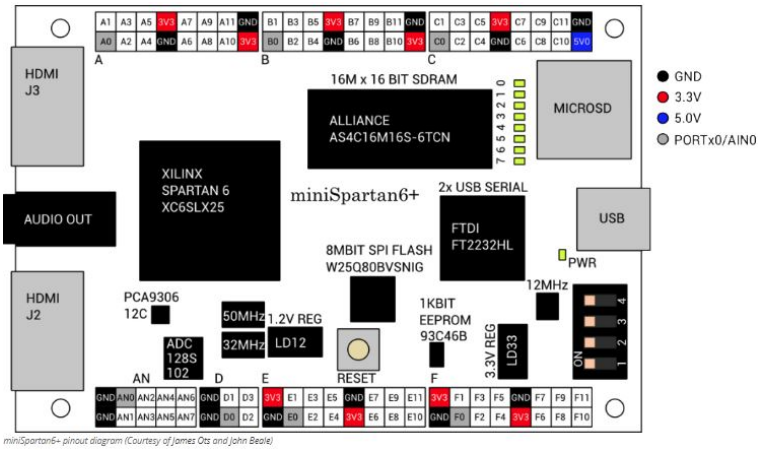


Figure 8.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [11].

8.3.2 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.

8.4 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 [12].

Chapter 9

Current Progress

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

9.1 RISC Core

Following the project time line described in section 8.2, 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.

9.1.1 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 9.1, 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 9.1: Initial Vmicro16 16-bit instruction set architecture. Coloured regions represent instruction families (bitwise, branching, arithmetic, etc.).

The ISA table is shown in Figure 9.1. 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.

9.1.2 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.

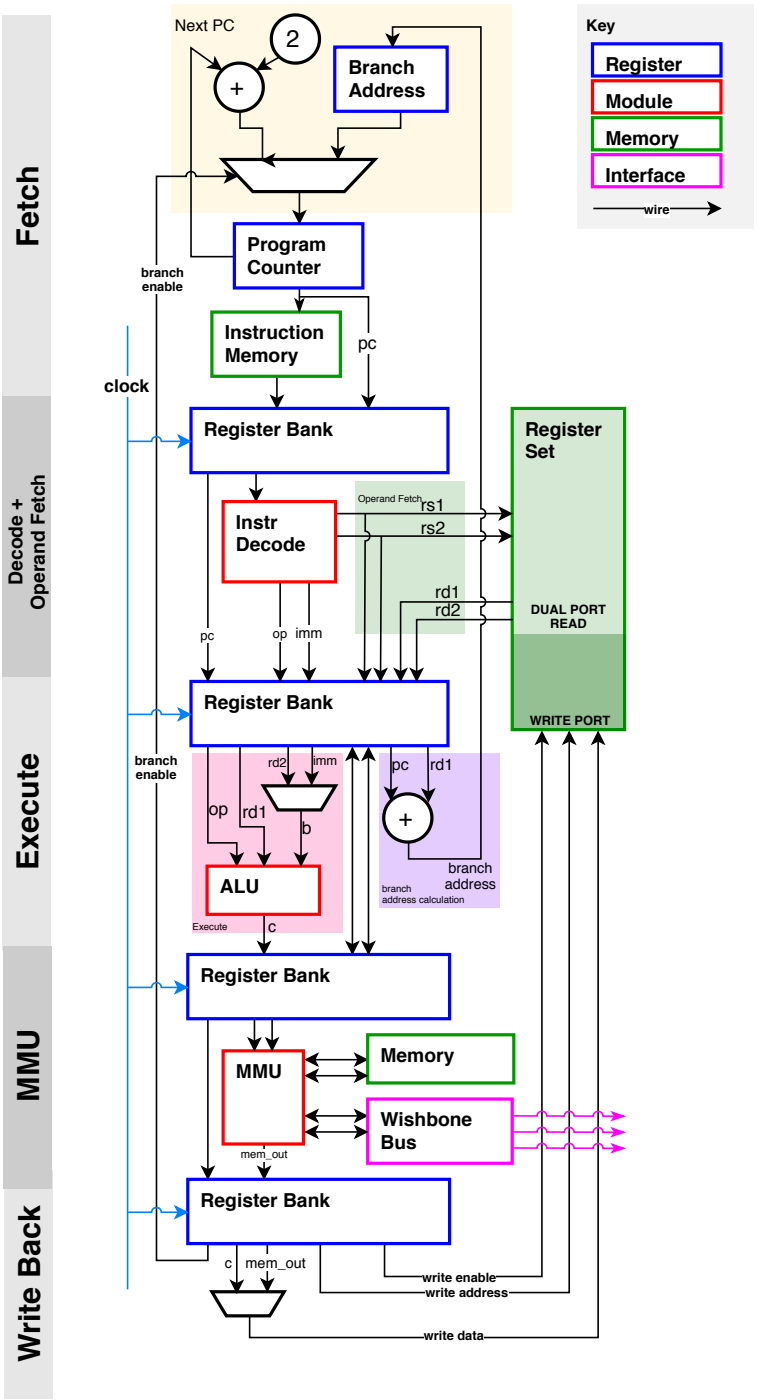


Figure 9.2: Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

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 9.3.

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

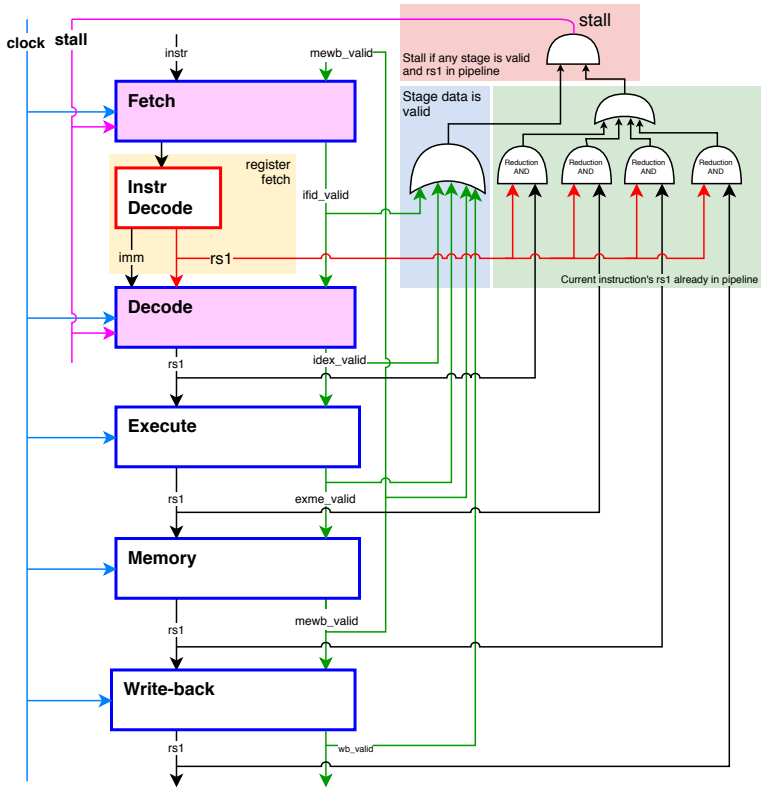


Figure 9.3: Pipeline data hazard detection. The register selects are passed forward through each stage and compared to the IDEX (latest instruction) register selects. If they match, the latest instruction depends on the output of an instruction in the pipeline, the IFID and IDEX stages are stalled to allow the instruction in the pipeline to commit.

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 9.1: 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.

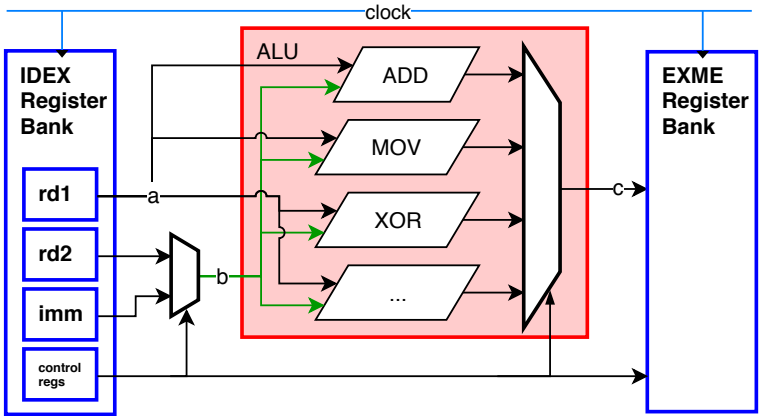


Figure 9.4: Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

The Verilog implementation of the ALU is shown in Figure 9.5. 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     input                       mmu_lwex,
323     input                       mmu_swex,
324     output reg [MEM_WIDTH-1:0]  mmu_out,
325
326     // interrupts
327     output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
328     output reg [`DEF_NUM_INT-1:0]             ints_mask,
329
330     // TO APB interconnect
331     output reg [`APB_WIDTH-1:0]  M_PADDR,
332     output reg                   M_PWRITE,
333     output reg                   M_PSELx,
334     output reg                   M_PENABLE,
335     output reg [MEM_WIDTH-1:0]   M_PWDATA,
```

Figure 9.5: 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             //`define TEST_BR
225             `ifdef TEST_BR
226             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0};
227             mem[1] = {`VMICRO16_OP_MOVI,    3'h3, 8'h3};
228             mem[2] = {`VMICRO16_OP_MOVI,    3'h1, 8'h2};
229             mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
230             mem[4] = {`VMICRO16_OP_BR,      3'h3, `VMICRO16_OP_BR_U};
231             mem[5] = {`VMICRO16_OP_MOVI,    3'h0, 8'hFF};
232             `endif
233
234             //`define ALL_TEST
235             `ifdef ALL_TEST
236             // Standard all test
237             // REGS0
238             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h81};
239             mem[1] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
240             mem[2] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
241             // GPIO0
242             mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
243             mem[4] = {`VMICRO16_OP_MOVI,    3'h1, 8'hD};
244             mem[5] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
245             mem[6] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
```

Figure 9.6: Vmicro16's decoder module code showing nested bit switches to determine the intended opcode. vmicro16.v

In Figure 9.6, 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.

9.1.3 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 10

Future Work

10.1 Project Status

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

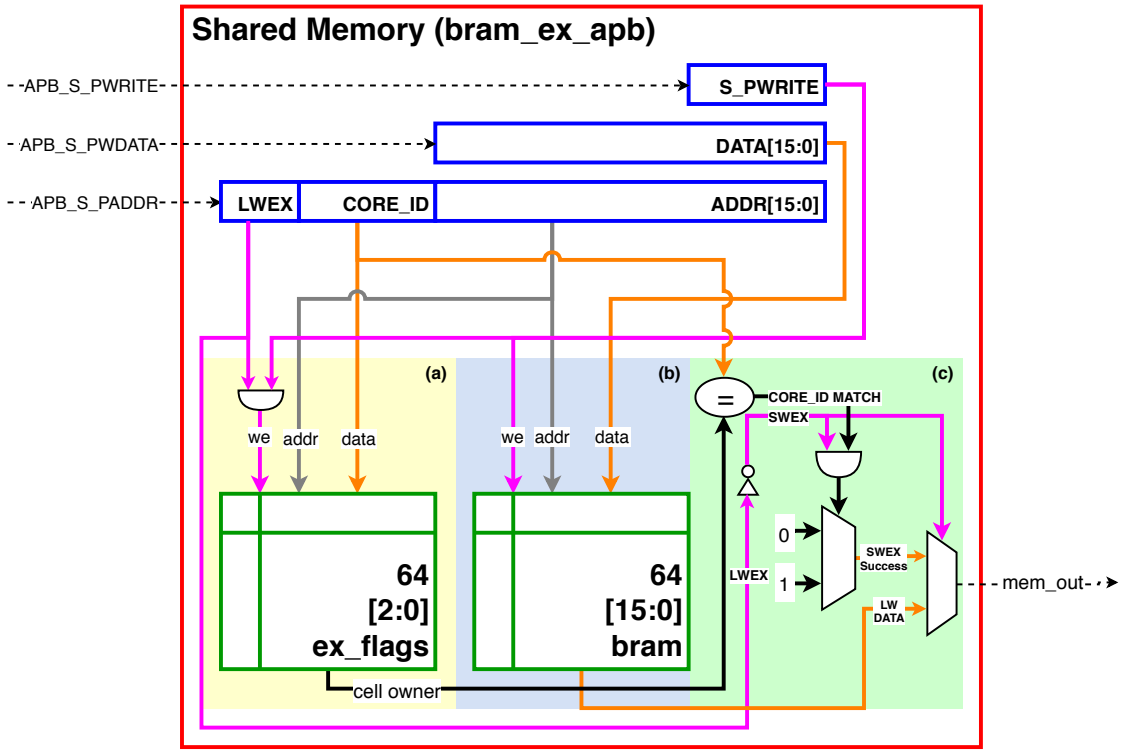


Figure 10.1: Caption for BRAMex

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.

10.1.1 Updated Project Time Line

The project table described in section 8.2 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 10.1 and in Figure 10.2.

10.1.2 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.

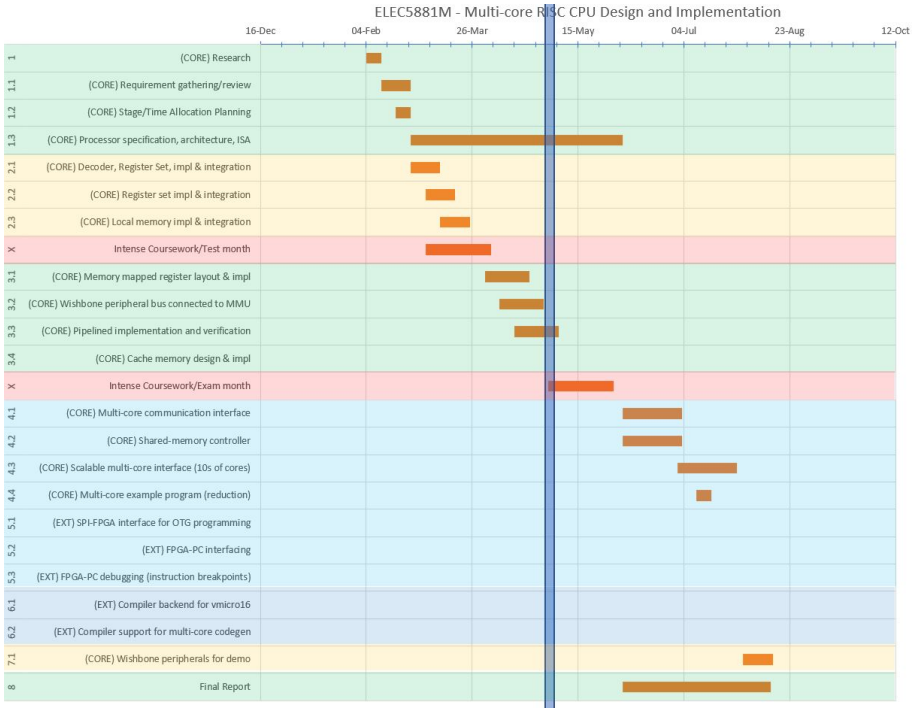


Figure 10.2: Updated project time gantt chart showing time allocations for stage 4.

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 10.1: Updated project stages.

Chapter 11

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

[3] Tech Differences, "Difference between loosely coupled and tightly coupled multiprocessor system (with comaprison chart)," Jul 2017. [Online]. Available: https://techdifferences.com/difference-between-loosely-coupled-and-tightly-coupled-multiprocessor-system.html (Accessed 2019-04-20).

[4] L. Benini and G. De Micheli, "Networks on Chips: A new SoC paradigm," *Computer*, vol. 35, pp. 70–78, 02 2002.

[5] D. Zhu, L. Chen, S. Yue, T. M. Pinkston, and M. Pedram, "Balancing On-Chip Network Latency in Multi-application Mapping for Chip-Multiprocessors," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 872–881.

[6] N. Chatterjee, S. Paul, and S. Chattopadhyay, "Fault-tolerant dynamic task mapping and scheduling for network-on-chip-based multicore platform," *ACM Transactions on Embedded Computing Systems*, vol. 16, pp. 1–24, 05 2017.

[7] Xilinx, *Spartan-6 FPGA Block RAM Resources*, Xilinx.

[8] Altera, *Recommended HDL Coding Styles - QII51007-9.0.0*, Altera.

[9] B. Lancaster, "FPGA-based RISC Microprocessor and Compiler," vol. 3.14, pp. 37–50. [Online]. Available: https://github.com/bendl/prco304 (Accessed March 2018).

[10] Terasic Technologies, "SoC Platform - Cyclone - DE1-SoC Board." [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836 (Accessed 2019-04-20).

[11] *MiniSpartan6+*, Scarab Hardware, 2014. [Online]. Available: https://www.scarabhardware.com/minispartan6/ (Accessed 2019-04-20).

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

Appendix A - Code Listing

top_ms.v

The top level implementation file is described here.

```
1   module seven_display # (
2       parameter INVERT = 1
3   ) (
4       input  [3:0] n,
5       output [6:0] segments
6   );
7       reg [6:0] bits;
8       assign segments = (INVERT ? ~bits : bits);
9
10      always @(n)
11      case (n)
12          4'h0: bits = 7'b0111111; // 0
13          4'h1: bits = 7'b0000110; // 1
14          4'h2: bits = 7'b1011011; // 2
15          4'h3: bits = 7'b1001111; // 3
16          4'h4: bits = 7'b1100110; // 4
17          4'h5: bits = 7'b1101101; // 5
18          4'h6: bits = 7'b1111101; // 6
19          4'h7: bits = 7'b0000111; // 7
20          4'h8: bits = 7'b1111111; // 8
21          4'h9: bits = 7'b1100111; // 9
22          4'hA: bits = 7'b1110111; // A
23          4'hB: bits = 7'b1111100; // B
24          4'hC: bits = 7'b0111001; // C
25          4'hD: bits = 7'b1011110; // D
26          4'hE: bits = 7'b1111001; // E
27          4'hF: bits = 7'b1110001; // F
28      endcase
29  endmodule
30
31  // minispartan6+ XC6SLX9
32  module top_ms # (
33      parameter GPIO_PINS = 8
34  ) (
35      input           CLK50,
36      input  [3:0]    SW,
37      // UART
38      //input           RXD,
39      output          TXD,
40      // Peripherals
41      output [7:0]    LEDS,
42
43      // SSDs
44      output [6:0] ssd0,
45      output [6:0] ssd1,
46      output [6:0] ssd2,
47      output [6:0] ssd3,
48      output [6:0] ssd4,
49      output [6:0] ssd5
50  );
51      localparam POR_CLKS  = 8;
52      reg [3:0]  por_timer = 0;
53      reg        por_done  = 0;
54      reg        por_reset = 1;
55      always @(posedge CLK50)
56          if (!por_done) begin
57              por_reset <= 1;
58              if (por_timer < POR_CLKS)
59                  por_timer <= por_timer + 1;
60              else
61                  por_done <= 1;
62          end
63          else
64              por_reset <= 0;
65
66      //wire [15:0]         M_PADDR;
67      //wire                M_PWRITE;
68      //wire [5-1:0]        M_PSELx;  // not shared
69      //wire                M_PENABLE;
70      //wire [15:0]         M_PWDATA;
71      //wire [15:0]         M_PRDATA; // input to intercon
72      //wire                M_PREADY; // input to intercon
```

```

    wire [7:0]  gpio0;
    wire [15:0] gpio1;
    wire [7:0]  gpio2;


    vmicro16_soc soc (
        .clk     (CLK50),
        .reset   (por_reset | (~SW[0])),

        //.M_PADDR    (M_PADDR),
        //.M_PWRITE   (M_PWRITE),
        //.M_PSELx    (M_PSELx),
        //.M_PENABLE  (M_PENABLE),
        //.M_PWDATA   (M_PWDATA),
        //.M_PRDATA   (M_PRDATA),
        //.M_PREADY   (M_PREADY),

        .uart_tx (TXD),
        .gpio0   (LEDS[3:0]),
        .gpio1   (gpio1),
        .gpio2   (gpio2),

        //.dbug0   (LEDS[3:0]),
        .dbug1   (LEDS[7:4])
    );

    // SSD displays (split across 2 gpio ports 1 and 2)
    wire [3:0] ssd_chars [0:5];
    assign ssd_chars[0] = gpio1[3:0];
    assign ssd_chars[1] = gpio1[7:4];
    assign ssd_chars[2] = gpio1[11:8];
    assign ssd_chars[3] = gpio1[15:12];
    assign ssd_chars[4] = gpio2[3:0];
    assign ssd_chars[5] = gpio2[7:4];
    seven_display ssd_0 (.n(ssd_chars[0]), .segments (ssd0));
    seven_display ssd_1 (.n(ssd_chars[1]), .segments (ssd1));
    seven_display ssd_2 (.n(ssd_chars[2]), .segments (ssd2));
    seven_display ssd_3 (.n(ssd_chars[3]), .segments (ssd3));
    seven_display ssd_4 (.n(ssd_chars[4]), .segments (ssd4));
    seven_display ssd_5 (.n(ssd_chars[5]), .segments (ssd5));

endmodule
```

apb_intercon.v

The

```
module seven_display # (
    parameter INVERT = 1
) (
    input  [3:0] n,
    output [6:0] segments
);
    reg [6:0] bits;
    assign segments = (INVERT ? ~bits : bits);

    always @(n)
    case (n)
        4'h0: bits = 7'b0111111; // 0
        4'h1: bits = 7'b0000110; // 1
        4'h2: bits = 7'b1011011; // 2
        4'h3: bits = 7'b1001111; // 3
        4'h4: bits = 7'b1100110; // 4
        4'h5: bits = 7'b1101101; // 5
        4'h6: bits = 7'b1111101; // 6
        4'h7: bits = 7'b0000111; // 7
        4'h8: bits = 7'b1111111; // 8
        4'h9: bits = 7'b1100111; // 9
        4'hA: bits = 7'b1110111; // A
        4'hB: bits = 7'b1111100; // B
        4'hC: bits = 7'b0111001; // C
        4'hD: bits = 7'b1011110; // D
        4'hE: bits = 7'b1111001; // E
        4'hF: bits = 7'b1110001; // F
    endcase
endmodule

// minispartan6+ XC6SLX9
module top_ms # (
    parameter GPIO_PINS = 8
) (
    input           CLK50,
    input  [3:0]    SW,
    // UART
    //input           RXD,
    output          TXD,
    // Peripherals
    output [7:0]    LEDS,

    // SSDs
    output [6:0] ssd0,
    output [6:0] ssd1,
    output [6:0] ssd2,
```

```
    output [6:0] ssd3,
    output [6:0] ssd4,
    output [6:0] ssd5
);
    localparam POR_CLKS  = 8;
    reg [3:0]  por_timer = 0;
    reg        por_done  = 0;
    reg        por_reset = 1;
    always @(posedge CLK50)
        if (!por_done) begin
            por_reset <= 1;
            if (por_timer < POR_CLKS)
                por_timer <= por_timer + 1;
            else
                por_done <= 1;
        end
        else
            por_reset <= 0;

    //wire [15:0]         M_PADDR;
    //wire                M_PWRITE;
    //wire [5-1:0]        M_PSELx;  // not shared
    //wire                M_PENABLE;
    //wire [15:0]         M_PWDATA;
    //wire [15:0]         M_PRDATA; // input to intercon
    //wire                M_PREADY; // input to intercon

    wire [7:0]  gpio0;
    wire [15:0] gpio1;
    wire [7:0]  gpio2;


    vmicro16_soc soc (
        .clk     (CLK50),
        .reset   (por_reset | (~SW[0])),

        //.M_PADDR    (M_PADDR),
        //.M_PWRITE   (M_PWRITE),
        //.M_PSELx    (M_PSELx),
        //.M_PENABLE  (M_PENABLE),
        //.M_PWDATA   (M_PWDATA),
        //.M_PRDATA   (M_PRDATA),
        //.M_PREADY   (M_PREADY),

        .uart_tx (TXD),
        .gpio0   (LEDS[3:0]),
        .gpio1   (gpio1),
        .gpio2   (gpio2),

        //.dbug0   (LEDS[3:0]),
        .dbug1   (LEDS[7:4])
    );

    // SSD displays (split across 2 gpio ports 1 and 2)
    wire [3:0] ssd_chars [0:5];
    assign ssd_chars[0] = gpio1[3:0];
    assign ssd_chars[1] = gpio1[7:4];
    assign ssd_chars[2] = gpio1[11:8];
    assign ssd_chars[3] = gpio1[15:12];
    assign ssd_chars[4] = gpio2[3:0];
    assign ssd_chars[5] = gpio2[7:4];
    seven_display ssd_0 (.n(ssd_chars[0]), .segments (ssd0));
    seven_display ssd_1 (.n(ssd_chars[1]), .segments (ssd1));
    seven_display ssd_2 (.n(ssd_chars[2]), .segments (ssd2));
    seven_display ssd_3 (.n(ssd_chars[3]), .segments (ssd3));
    seven_display ssd_4 (.n(ssd_chars[4]), .segments (ssd4));
    seven_display ssd_5 (.n(ssd_chars[5]), .segments (ssd5));

endmodule
```

vmicro16.v

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

```
// This file contains multiple modules.
//   Verilator likes 1 file for each module
/* verilator lint_off DECLFILENAME */
/* verilator lint_off UNUSED */
/* verilator lint_off BLKSEQ */
/* verilator lint_off WIDTH */

// Include Vmicro16 ISA containing definitions for the bits
`include "vmicro16_isa.v"

`include "clog2.v"
`include "formal.v"
```

```
18
19  module vmicro16_bram_apb # (
20      parameter BUS_WIDTH    = 16,
21      parameter MEM_WIDTH    = 16,
22      parameter MEM_DEPTH    = 64,
23      parameter APB_PADDR    = 0
24  ) (
25      input clk,
26      input reset,
27      // APB Slave to master interface
28      input  [`clog2(MEM_DEPTH)-1:0]  S_PADDR,
29      input                           S_PWRITE,
30      input                           S_PSELx,
31      input                           S_PENABLE,
32      input  [BUS_WIDTH-1:0]          S_PWDATA,
33
34      output [BUS_WIDTH-1:0]          S_PRDATA,
35      output                          S_PREADY
36  );
37      wire [MEM_WIDTH-1:0] mem_out;
38
39      assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out : 16'h0000;
40      assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1    : 1'b0;
41      assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
42
43      always @(*)
44          if (S_PSELx && S_PENABLE)
45              $display($time, "\t\tMEM => %h", mem_out);
46
47      always @(posedge clk)
48          if (we)
49              $display($time, "\t\tBRAM[%h] <= %h", S_PADDR, S_PWDATA);
50
51      vmicro16_bram # (
52          .MEM_WIDTH  (MEM_WIDTH),
53          .MEM_DEPTH  (MEM_DEPTH),
54          .NAME       ("BRAM")
55      ) bram_apb (
56          .clk        (clk),
57          .reset      (reset),
58
59          .mem_addr   (S_PADDR),
60          .mem_in     (S_PWDATA),
61          .mem_we     (we),
62          .mem_out    (mem_out)
63      );
64  endmodule
65
66
67  // This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
68  //   https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
69  //   https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
70  //   https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf
71
72  module vmicro16_bram # (
73      parameter MEM_WIDTH     = 16,
74      parameter MEM_DEPTH     = 64,
75      parameter CORE_ID       = 0,
76      parameter USE_INITS     = 0,
77      parameter PARAM_DEFAULTS_R0 = 0,
78      parameter PARAM_DEFAULTS_R1 = 0,
79      parameter NAME          = "BRAM"
80  ) (
81      input clk,
82      input reset,
83
84      input      [`clog2(MEM_DEPTH)-1:0] mem_addr,
85      input      [MEM_WIDTH-1:0]         mem_in,
86      input                              mem_we,
87      output reg [MEM_WIDTH-1:0]         mem_out
88  );
89      // memory vector
90      reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];
91
92      // not synthesizable
93      integer i;
94      initial begin
95          for (i = 0; i < MEM_DEPTH; i = i + 1) mem[i] = 0;
96          mem[0] = PARAM_DEFAULTS_R0;
97          mem[1] = PARAM_DEFAULTS_R1;
98
99          if (USE_INITS) begin
100             //`define TEST_SW
101             `ifdef TEST_SW
102             $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\verilog_memh.txt", mem);
103             `endif
104
105             `define TEST_ASM
106             `ifdef TEST_ASM
107             $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\asm.s.hex", mem);
108             `endif
109
110             //`define TEST_COMPILER
111             `ifdef TEST_COMPILER
112     mem[0] = 16'h2f3f;
113     mem[1] = 16'h2903;
114     mem[2] = 16'h4100;
115     mem[3] = 16'h3fa1;
116     mem[4] = 16'h16e0;
```

```
117         mem[5] = 16'h26e0;
118         mem[6] = 16'h3fa1;
119         mem[7] = 16'h2890;
120         mem[8] = 16'h10d8;
121         mem[9] = 16'h3fa1;
122         mem[10] = 16'h2891;
123         mem[11] = 16'h10d9;
124         mem[12] = 16'h3fa1;
125         mem[13] = 16'h2892;
126         mem[14] = 16'h10da;
127         mem[15] = 16'h3fa1;
128         mem[16] = 16'h28a0;
129         mem[17] = 16'h10db;
130         mem[18] = 16'h3fa1;
131         mem[19] = 16'h2880;
132         mem[20] = 16'h10dc;
133         mem[21] = 16'h3fa1;
134         mem[22] = 16'h28b0;
135         mem[23] = 16'h10dd;
136         mem[24] = 16'h3fa1;
137         mem[25] = 16'h28b1;
138         mem[26] = 16'h10de;
139         mem[27] = 16'h3fa1;
140         mem[28] = 16'h08dc;
141         mem[29] = 16'h0800;
142         mem[30] = 16'h3fa1;
143         mem[31] = 16'h10e0;
144         mem[32] = 16'h2801;
145         mem[33] = 16'h0be0;
146         mem[34] = 16'h37a1;
147         mem[35] = 16'h4b00;
148         mem[36] = 16'h5001;
149         mem[37] = 16'h2b00;
150         mem[38] = 16'h4860;
151         mem[39] = 16'h292c;
152         mem[40] = 16'h4101;
153         mem[41] = 16'h2864;
154         mem[42] = 16'h292e;
155         mem[43] = 16'h4100;
156         mem[44] = 16'h0000;
157         mem[45] = 16'h28c8;
158         mem[46] = 16'h0000;
159         mem[47] = 16'h08dc;
160         mem[48] = 16'h0800;
161         mem[49] = 16'h3fa1;
162         mem[50] = 16'h10e0;
163         mem[51] = 16'h2805;
164         mem[52] = 16'h0be0;
165         mem[53] = 16'h37a1;
166         mem[54] = 16'h5860;
167         mem[55] = 16'h10df;
168         mem[56] = 16'h08df;
169         mem[57] = 16'h3fa1;
170         mem[58] = 16'h10e0;
171         mem[59] = 16'h2830;
172         mem[60] = 16'h0be0;
173         mem[61] = 16'h37a1;
174         mem[62] = 16'h307f;
175         mem[63] = 16'h3fa1;
176         mem[64] = 16'h10e0;
177         mem[65] = 16'h08db;
178         mem[66] = 16'h0be0;
179         mem[67] = 16'h37a1;
180         mem[68] = 16'h1300;
181         mem[69] = 16'h2832;
182         mem[70] = 16'h27c0;
183         mem[71] = 16'h0ee0;
184         mem[72] = 16'h37a1;
185         mem[73] = 16'h6000;
186 `endif
187
188 //`define TEST_COND
189 `ifdef TEST_COND
190 mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
191 mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
192 `endif
193
194 //`define TEST_CMP
195 `ifdef TEST_CMP
196 mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0A};
197 mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h0B};
198 mem[2] = {`VMICRO16_OP_CMP,     3'h1, 3'h0, 5'h1};
199 `endif
200
201 //`define TEST_LWEX
202 `ifdef TEST_LWEX
203 mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC5};
204 mem[1] = {`VMICRO16_OP_SW,      3'h0, 3'h0, 5'h1};
205 mem[2] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h1};
206 mem[3] = {`VMICRO16_OP_LWEX,    3'h2, 3'h0, 5'h1};
207 mem[4] = {`VMICRO16_OP_SWEX,    3'h3, 3'h0, 5'h1};
208 `endif
209
210 //`define TEST_MULTICORE
211 `ifdef TEST_MULTICORE
212 mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
213 mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
214 mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
215 mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h80};
```

```
216             mem[4] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
217             mem[5] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
218             mem[6] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
219             mem[7] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
220             mem[8] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
221             mem[9] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h0};
222             `endif
223
224             //`define TEST_BR
225             `ifdef TEST_BR
226             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0};
227             mem[1] = {`VMICRO16_OP_MOVI,    3'h3, 8'h3};
228             mem[2] = {`VMICRO16_OP_MOVI,    3'h1, 8'h2};
229             mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
230             mem[4] = {`VMICRO16_OP_BR,      3'h3, `VMICRO16_OP_BR_U};
231             mem[5] = {`VMICRO16_OP_MOVI,    3'h0, 8'hFF};
232             `endif
233
234             //`define ALL_TEST
235             `ifdef ALL_TEST
236             // Standard all test
237             // REGS0
238             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h81};
239             mem[1] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
240             mem[2] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
241             // GPIO0
242             mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
243             mem[4] = {`VMICRO16_OP_MOVI,    3'h1, 8'hD};
244             mem[5] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
245             mem[6] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
246             // TIM0
247             mem[7] = {`VMICRO16_OP_MOVI,    3'h0, 8'h07};
248             mem[8] = {`VMICRO16_OP_LW,      3'h3, 3'h0, 5'h03};
249             // UART0
250             mem[9]  = {`VMICRO16_OP_MOVI,    3'h0, 8'hA0};      // UART0
251             mem[10] = {`VMICRO16_OP_MOVI,    3'h1, 8'h41};      // ascii A
252             mem[11] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
253             mem[12] = {`VMICRO16_OP_MOVI,    3'h1, 8'h42}; // ascii B
254             mem[13] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
255             mem[14] = {`VMICRO16_OP_MOVI,    3'h1, 8'h43}; // ascii C
256             mem[15] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
257             mem[16] = {`VMICRO16_OP_MOVI,    3'h1, 8'h44}; // ascii D
258             mem[17] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
259             mem[18] = {`VMICRO16_OP_MOVI,    3'h1, 8'h45}; // ascii D
260             mem[19] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
261             mem[20] = {`VMICRO16_OP_MOVI,    3'h1, 8'h46}; // ascii E
262             mem[21] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
263             // BRAM0
264             mem[22] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
265             mem[23] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
266             mem[24] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
267             mem[25] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
268             // GPIO1 (SSD 24-bit port)
269             mem[26] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
270             mem[27] = {`VMICRO16_OP_MOVI,    3'h1, 8'h12};
271             mem[28] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
272             mem[29] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
273             // GPIO2
274             mem[30] = {`VMICRO16_OP_MOVI,    3'h0, 8'h92};
275             mem[31] = {`VMICRO16_OP_MOVI,    3'h1, 8'h56};
276             mem[32] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
277             `endif
278
279             //`define TEST_BRAM
280             `ifdef TEST_BRAM
281             // 2 core BRAM0 test
282             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
283             mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
284             mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
285             mem[3] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
286             `endif
287         end
288     end
289
290     always @(posedge clk) begin
291         // synchronous WRITE_FIRST (page 13)
292         if (mem_we) begin
293             mem[mem_addr] <= mem_in;
294             $display($time, "\t\t%s[%h] <= %h",
295                     NAME, mem_addr, mem_in);
296         end else
297             mem_out <= mem[mem_addr];
298     end
299
300     // TODO: Reset impl = every clock while reset is asserted, clear each cell
301     //       one at a time, mem[i++] <= 0
302 endmodule
303
304
305 module vmicro16_core_mmu # (
306     parameter MEM_WIDTH     = 16,
307     parameter MEM_DEPTH     = 64,
308
309     parameter CORE_ID       = 3'h0,
310     parameter CORE_ID_BITS  = `clog2(`CORES)
311 ) (
312     input clk,
313     input reset,
314
```

```
315       input  req,
316       output busy,
317
318       // From core
319       input      [MEM_WIDTH-1:0]  mmu_addr,
320       input      [MEM_WIDTH-1:0]  mmu_in,
321       input                       mmu_we,
322       input                       mmu_lwex,
323       input                       mmu_swex,
324       output reg [MEM_WIDTH-1:0]  mmu_out,
325
326       // interrupts
327       output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
328       output reg [`DEF_NUM_INT-1:0]             ints_mask,
329
330       // TO APB interconnect
331       output reg [`APB_WIDTH-1:0]  M_PADDR,
332       output reg                   M_PWRITE,
333       output reg                   M_PSELx,
334       output reg                   M_PENABLE,
335       output reg [MEM_WIDTH-1:0]   M_PWDATA,
336       // from interconnect
337       input      [MEM_WIDTH-1:0]   M_PRDATA,
338       input                        M_PREADY
339   );
340       localparam MMU_STATE_T1  = 0;
341       localparam MMU_STATE_T2  = 1;
342       localparam MMU_STATE_T3  = 2;
343       reg [1:0]  mmu_state      = MMU_STATE_T1;
344
345       reg  [MEM_WIDTH-1:0] per_out = 0;
346       wire [MEM_WIDTH-1:0] tim0_out;
347
348       assign busy = req || (mmu_state == MMU_STATE_T2);
349
350       // tightly integrated memory usage
351       wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
352                   && (mmu_addr <= `DEF_MMU_TIM0_E);
353       wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
354                   && (mmu_addr <= `DEF_MMU_SREG_E);
355       wire intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
356                   && (mmu_addr <= `DEF_MMU_INTSV_E);
357       wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
358                   && (mmu_addr <= `DEF_MMU_INTSM_E);
359
360       wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
361       wire tim0_we   = (tim0_en && mmu_we);
362       wire intv_we   = (intv_en && mmu_we);
363       wire intm_we   = (intm_en && mmu_we);
364
365       // Special register selects
366       localparam SPECIAL_REGS = 8;
367       wire [MEM_WIDTH-1:0] sr_val;
368
369       // Interrupt vector and mask
370       initial ints_vector = 0;
371       initial ints_mask   = 0;
372       wire [2:0] intv_addr = mmu_addr[`clog2(`DEF_NUM_INT)-1:0];
373       always @(posedge clk)
374           if (intv_we)
375               ints_vector[intv_addr*`DATA_WIDTH +: `DATA_WIDTH] <= mmu_in;
376
377       always @(posedge clk)
378           if (intm_we)
379               ints_mask <= mmu_in;
380
381
382       always @(ints_vector)
383           $display($time, "\tC%d\t\tints_vector W: | %h %h %h %h | %h %h %h %h |", CORE_ID,
384               ints_vector[0*`DATA_WIDTH +: `DATA_WIDTH],
385               ints_vector[1*`DATA_WIDTH +: `DATA_WIDTH],
386               ints_vector[2*`DATA_WIDTH +: `DATA_WIDTH],
387               ints_vector[3*`DATA_WIDTH +: `DATA_WIDTH],
388               ints_vector[4*`DATA_WIDTH +: `DATA_WIDTH],
389               ints_vector[5*`DATA_WIDTH +: `DATA_WIDTH],
390               ints_vector[6*`DATA_WIDTH +: `DATA_WIDTH],
391               ints_vector[7*`DATA_WIDTH +: `DATA_WIDTH]
392               );
393
394       always @(intm_we)
395           $display($time, "\tC%d\t\tintm_we W: %b", CORE_ID, ints_mask);
396
397       // Output port
398       always @(*)
399           if      (tim0_en) mmu_out = tim0_out;
400           else if (sreg_en) mmu_out = sr_val;
401           else if (intv_en) mmu_out = ints_vector[mmu_addr[2:0]*`DATA_WIDTH +: `DATA_WIDTH];
402           else if (intm_en) mmu_out = ints_mask;
403           else              mmu_out = per_out;
404
405       // APB master to slave interface
406       always @(posedge clk)
407           if (reset) begin
408               mmu_state <= MMU_STATE_T1;
409               M_PENABLE <= 0;
410               M_PADDR   <= 0;
411               M_PWDATA  <= 0;
412               M_PSELx   <= 0;
413               M_PWRITE  <= 0;
```

```
414         end
415         else
416             casex (mmu_state)
417                 MMU_STATE_T1: begin
418                     if (req && apb_en) begin
419                         M_PADDR   <= {mmu_lwex, mmu_swex, CORE_ID[CORE_ID_BITS-1:0], mmu_addr[MEM_WIDTH-1:0]};
420                         M_PWDATA  <= mmu_in;
421                         M_PSELx   <= 1;
422                         M_PWRITE  <= mmu_we;
423 
424                         mmu_state <= MMU_STATE_T2;
425                     end
426                 end
427 
428                 `ifdef FIX_T3
429                     MMU_STATE_T2: begin
430                         M_PENABLE <= 1;
431 
432                         if (M_PREADY == 1'b1) begin
433                             mmu_state <= MMU_STATE_T3;
434                         end
435                     end
436 
437                     MMU_STATE_T3: begin
438                         // Slave has output a ready signal (finished)
439                         M_PENABLE <= 0;
440                         M_PADDR   <= 0;
441                         M_PWDATA  <= 0;
442                         M_PSELx   <= 0;
443                         M_PWRITE  <= 0;
444                         // Clock the peripheral output into a reg,
445                         //   to output on the next clock cycle
446                         per_out   <= M_PRDATA;
447 
448                         mmu_state <= MMU_STATE_T1;
449                     end
450                 `else
451                     // No FIX_T3
452                     MMU_STATE_T2: begin
453                         if (M_PREADY == 1'b1) begin
454                             M_PENABLE <= 0;
455                             M_PADDR   <= 0;
456                             M_PWDATA  <= 0;
457                             M_PSELx   <= 0;
458                             M_PWRITE  <= 0;
459                             // Clock the peripheral output into a reg,
460                             //   to output on the next clock cycle
461                             per_out   <= M_PRDATA;
462 
463                             mmu_state <= MMU_STATE_T1;
464                         end else begin
465                             M_PENABLE <= 1;
466                         end
467                     end
468                 `endif
469             endcase
470 
471     vmicro16_bram # (
472         .MEM_WIDTH  (MEM_WIDTH),
473         .MEM_DEPTH  (SPECIAL_REGS),
474         .USE_INITS  (0),
475         .PARAM_DEFAULTS_R0  (CORE_ID),
476         .PARAM_DEFAULTS_R1  (`DEF_INT_MASK),
477         .NAME       ("ram_sr")
478     ) ram_sr (
479         .clk        (clk),
480         .reset      (reset),
481         .mem_addr   (mmu_addr[`clog2(SPECIAL_REGS)-1:0]),
482         .mem_in     (),
483         .mem_we     (),
484         .mem_out    (sr_val)
485     );
486 
487     // Each M core has a TIM0 scratch memory
488     vmicro16_bram # (
489         .MEM_WIDTH  (MEM_WIDTH),
490         .MEM_DEPTH  (MEM_DEPTH),
491         .USE_INITS  (0),
492         .NAME       ("TIM0")
493     ) TIM0 (
494         .clk        (clk),
495         .reset      (reset),
496         .mem_addr   (mmu_addr[7:0]),
497         .mem_in     (mmu_in),
498         .mem_we     (tim0_we),
499         .mem_out    (tim0_out)
500     );
501 endmodule
502 
503 
504 
505 module vmicro16_regs # (
506     parameter CELL_WIDTH        = 16,
507     parameter CELL_DEPTH        = 8,
508     parameter CELL_SEL_BITS     = `clog2(CELL_DEPTH),
509     parameter CELL_DEFAULTS     = 0,
510     parameter DEBUG_NAME        = "",
511     parameter CORE_ID           = 0,
512     parameter PARAM_DEFAULTS_R0 = 16'h0000,
```

```
513         parameter PARAM_DEFAULTS_R1 = 16'h0000
514     ) (
515         input clk,
516         input reset,
517         // Dual port register reads
518         input      [CELL_SEL_BITS-1:0]  rs1, // port 1
519         output     [CELL_WIDTH-1   :0]  rd1,
520         //input      [CELL_SEL_BITS-1:0]  rs2, // port 2
521         //output     [CELL_WIDTH-1   :0]  rd2,
522         // EX/WB final stage write back
523         input                           we,
524         input [CELL_SEL_BITS-1:0]       ws1,
525         input [CELL_WIDTH-1:0]          wd
526     );
527         reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;
528
529         // Initialise registers with default values
530         //   Really only used for special registers used by the soc
531         // TODO: How to do this on reset?
532         integer i;
533         initial
534             if (CELL_DEFAULTS)
535                 $readmemh(CELL_DEFAULTS, regs);
536             else begin
537                 for(i = 0; i < CELL_DEPTH; i = i + 1)
538                     regs[i] = 0;
539                 regs[0] = PARAM_DEFAULTS_R0;
540                 regs[1] = PARAM_DEFAULTS_R1;
541                 end
542
543         always @(regs)
544             $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
545                 CORE_ID,
546                 regs[0], regs[1], regs[2], regs[3],
547                 regs[4], regs[5], regs[6], regs[7]);
548
549         always @(posedge clk)
550             if (reset) begin
551                 for(i = 0; i < CELL_DEPTH; i = i + 1)
552                     regs[i] <= 0;
553                 regs[0] <= PARAM_DEFAULTS_R0;
554                 regs[1] <= PARAM_DEFAULTS_R1;
555             end
556             else if (we) begin
557                 $display($time, "\tC%02h: REGS #%s: Writing %h to reg[%d]",
558                     CORE_ID, DEBUG_NAME, wd, ws1);
559
560                 // Perform the write
561                 regs[ws1] <= wd;
562             end
563
564         // sync writes, async reads
565         assign rd1 = regs[rs1];
566         //assign rd2 = regs[rs2];
567     endmodule
568
569
570
571     module vmicro16_regs_apb # (
572         parameter BUS_WIDTH         = 16,
573         parameter DATA_WIDTH        = 16,
574         parameter CELL_DEPTH        = 8,
575         parameter PARAM_DEFAULTS_R0 = 0,
576         parameter PARAM_DEFAULTS_R1 = 0
577     ) (
578         input clk,
579         input reset,
580         // APB Slave to master interface
581         input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
582         input                           S_PWRITE,
583         input                           S_PSELx,
584         input                           S_PENABLE,
585         input  [DATA_WIDTH-1:0]          S_PWDATA,
586
587         output [DATA_WIDTH-1:0]          S_PRDATA,
588         output                          S_PREADY
589     );
590         wire [DATA_WIDTH-1:0] rd1;
591
592         assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1  : 16'h0000;
593         assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
594         assign reg_we   = (S_PSELx & S_PENABLE & S_PWRITE);
595
596         always @(*)
597             if (reg_we)
598                 $display($time, "\t\tREGS_APB[%h] <= %h", S_PADDR, S_PWDATA);
599
600         always @(*)
601             `rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))
602
603         vmicro16_regs # (
604             .CELL_DEPTH         (CELL_DEPTH),
605             .CELL_WIDTH         (DATA_WIDTH),
606             .PARAM_DEFAULTS_R0  (PARAM_DEFAULTS_R0),
607             .PARAM_DEFAULTS_R1  (PARAM_DEFAULTS_R1)
608         ) regs_apb (
609             .clk    (clk),
610             .reset  (reset),
611
```

```
612          .rs1    (S_PADDR),
613          .rd1    (rd1),
614
615          //.rs2    (),
616          //.rd2    (),
617
618          .we     (reg_we),
619          .ws1    (S_PADDR),
620          .wd     (S_PWDATA) // either alu_c or mem_out
621      );
622  endmodule
623
624
625
626
627  module vmicro16_gpio_apb # (
628      parameter BUS_WIDTH  = 16,
629      parameter DATA_WIDTH = 16,
630      parameter PORTS      = 8,
631      parameter NAME       = "GPIO"
632  ) (
633      input clk,
634      input reset,
635      // APB Slave to master interface
636      input  [0:0]                    S_PADDR, // not used (optimised out)
637      input                           S_PWRITE,
638      input                           S_PSELx,
639      input                           S_PENABLE,
640      input  [DATA_WIDTH-1:0]         S_PWDATA,
641
642      output [DATA_WIDTH-1:0]          S_PRDATA,
643      output                          S_PREADY,
644      output reg [PORTS-1:0]          gpio
645  );
646      assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
647      assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
648      assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);
649
650      always @(posedge clk)
651          if (reset)
652              gpio <= 0;
653          else if (ports_we) begin
654              $display($time, "\t\%s <= %h", NAME, S_PWDATA[PORTS-1:0]);
655              gpio <= S_PWDATA[PORTS-1:0];
656          end
657  endmodule
658
659  // Decoder is hard to parameterise as it's very closely linked to the ISA.
660
661  module vmicro16_dec # (
662      parameter INSTR_WIDTH    = 16,
663      parameter INSTR_OP_WIDTH = 5,
664      parameter INSTR_RS_WIDTH = 3,
665      parameter ALU_OP_WIDTH   = 5
666  ) (
667      //input clk,   // not used yet (all combinational)
668      //input reset, // not used yet (all combinational)
669
670      input  [INSTR_WIDTH-1:0]    instr,
671
672      output [INSTR_OP_WIDTH-1:0] opcode,
673      output [INSTR_RS_WIDTH-1:0] rd,
674      output [INSTR_RS_WIDTH-1:0] ra,
675      output [3:0]                imm4,
676      output [7:0]                imm8,
677      output [11:0]               imm12,
678      output [4:0]                simm5,
679
680      // This can be freely increased without affecting the isa
681      output reg [ALU_OP_WIDTH-1:0] alu_op,
682
683      output reg has_imm4,
684      output reg has_imm8,
685      output reg has_imm12,
686      output reg has_we,
687      output reg has_br,
688      output reg has_mem,
689      output reg has_mem_we,
690      output reg has_cmp,
691
692      output halt,
693      output intr,
694
695      output reg has_lwex,
696      output reg has_swex
697
698      // TODO: Use to identify bad instruction and
699      //       raise exceptions
700      //,output     is_bad
701  );
702      assign opcode = instr[15:11];
703      assign rd     = instr[10:8];
704      assign ra     = instr[7:5];
705      assign imm4   = instr[3:0];
706      assign imm8   = instr[7:0];
707      assign imm12  = instr[11:0];
708      assign simm5  = instr[4:0];
709
710      // exme_op
```

```
711         always @(*) case (opcode)
712             `VMICRO16_OP_SPCL: casez(instr[11:0])
713                 `VMICRO16_OP_SPCL_NOP,
714                 `VMICRO16_OP_SPCL_HALT,
715                 `VMICRO16_OP_SPCL_INTR:   alu_op = `VMICRO16_ALU_NOP;
716                 default:                  alu_op = `VMICRO16_ALU_NOP; endcase
717 
718             `VMICRO16_OP_LW:              alu_op = `VMICRO16_ALU_LW;
719             `VMICRO16_OP_SW:              alu_op = `VMICRO16_ALU_SW;
720             `VMICRO16_OP_LWEX:            alu_op = `VMICRO16_ALU_LW;
721             `VMICRO16_OP_SWEX:            alu_op = `VMICRO16_ALU_SW;
722 
723             `VMICRO16_OP_MOV:             alu_op = `VMICRO16_ALU_MOV;
724             `VMICRO16_OP_MOVI:            alu_op = `VMICRO16_ALU_MOVI;
725 
726             `VMICRO16_OP_BR:              alu_op = `VMICRO16_ALU_BR;
727             `VMICRO16_OP_MULT:            alu_op = `VMICRO16_ALU_MULT;
728 
729             `VMICRO16_OP_CMP:             alu_op = `VMICRO16_ALU_CMP;
730             `VMICRO16_OP_SETC:            alu_op = `VMICRO16_ALU_SETC;
731 
732             `VMICRO16_OP_BIT:     casez (simm5)
733                 `VMICRO16_OP_BIT_OR:      alu_op = `VMICRO16_ALU_BIT_OR;
734                 `VMICRO16_OP_BIT_XOR:     alu_op = `VMICRO16_ALU_BIT_XOR;
735                 `VMICRO16_OP_BIT_AND:     alu_op = `VMICRO16_ALU_BIT_AND;
736                 `VMICRO16_OP_BIT_NOT:     alu_op = `VMICRO16_ALU_BIT_NOT;
737                 `VMICRO16_OP_BIT_LSHFT:   alu_op = `VMICRO16_ALU_BIT_LSHFT;
738                 `VMICRO16_OP_BIT_RSHFT:   alu_op = `VMICRO16_ALU_BIT_RSHFT;
739                 default:                  alu_op = `VMICRO16_ALU_BAD; endcase
740 
741             `VMICRO16_OP_ARITH_U:     casez (simm5)
742                 `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
743                 `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
744                 `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
745                 default:                  alu_op = `VMICRO16_ALU_BAD; endcase
746 
747             `VMICRO16_OP_ARITH_S:     casez (simm5)
748                 `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
749                 `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
750                 `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
751                 default:                  alu_op = `VMICRO16_ALU_BAD; endcase
752 
753             default: begin
754                                       alu_op = `VMICRO16_ALU_NOP;
755                 $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
756             end
757         endcase
758 
759         // Special opcodes
760         //assign nop  == ((opcode == `VMICRO16_OP_SPCL) & (~instr[0]));
761         assign halt = ((opcode == `VMICRO16_OP_SPCL) &   instr[0]);
762         assign intr = ((opcode == `VMICRO16_OP_SPCL) &   instr[1]);
763 
764         // Register writes
765         always @(*) case (opcode)
766             `VMICRO16_OP_LWEX,
767             `VMICRO16_OP_SWEX,
768             `VMICRO16_OP_LW,
769             `VMICRO16_OP_MOV,
770             `VMICRO16_OP_MOVI,
771             //`VMICRO16_OP_MOVI_L,
772             `VMICRO16_OP_ARITH_U,
773             `VMICRO16_OP_ARITH_S,
774             `VMICRO16_OP_SETC,
775             `VMICRO16_OP_BIT,
776             `VMICRO16_OP_MULT:      has_we = 1'b1;
777             default:                has_we = 1'b0;
778         endcase
779 
780         // Contains 4-bit immediate
781         always @(*)
782             if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
783                 ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
784                 has_imm4 = 1'b1;
785             else
786                 has_imm4 = 1'b0;
787 
788         // Contains 8-bit immediate
789         always @(*) case (opcode)
790             `VMICRO16_OP_MOVI,
791             `VMICRO16_OP_BR:        has_imm8 = 1'b1;
792             default:                has_imm8 = 1'b0;
793         endcase
794 
795         //// Contains 12-bit immediate
796         //always @(*) case (opcode)
797         //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
798         //    default:                has_imm12 = 1'b0;
799         //endcase
800 
801         // Will branch the pc
802         always @(*) case (opcode)
803             `VMICRO16_OP_BR:    has_br = 1'b1;
804             default:            has_br = 1'b0;
805         endcase
806 
807         // Requires external memory
808         always @(*) case (opcode)
809             `VMICRO16_OP_LW,
```

```
        `VMICRO16_OP_SW,
        `VMICRO16_OP_LWEX,
        `VMICRO16_OP_SWEX:  has_mem = 1'b1;
        default:            has_mem = 1'b0;
    endcase

    // Requires external memory write
    always @(*) case (opcode)
        `VMICRO16_OP_SW,
        `VMICRO16_OP_SWEX:  has_mem_we = 1'b1;
        default:            has_mem_we = 1'b0;
    endcase

    // Affects status registers (cmp instructions)
    always @(*) case (opcode)
        `VMICRO16_OP_CMP:   has_cmp = 1'b1;
        default:            has_cmp = 1'b0;
    endcase

    // Performs exclusive checks
    always @(*) case (opcode)
        `VMICRO16_OP_LWEX:   has_lwex = 1'b1;
        default:             has_lwex = 1'b0;
    endcase

    always @(*) case (opcode)
        `VMICRO16_OP_SWEX:   has_swex = 1'b1;
        default:             has_swex = 1'b0;
    endcase
endmodule


module vmicro16_alu # (
    parameter OP_WIDTH   = 5,
    parameter DATA_WIDTH = 16,
    parameter CORE_ID    = 0
) (
    // input clk, // TODO: make clocked

    input      [OP_WIDTH-1:0]   op,
    input      [DATA_WIDTH-1:0] a, // rs1/dst
    input      [DATA_WIDTH-1:0] b, // rs2
    input      [3:0]            flags,
    output reg [DATA_WIDTH-1:0] c
);
    localparam TOP_BIT = (DATA_WIDTH-1);
    // 17-bit register
    reg [DATA_WIDTH:0] cmp_tmp = 0; // = {carry, [15:0]}
    wire r_setc;

    always @(*) begin
                        cmp_tmp = 0;
                        case (op)
        // branch/nop, output nothing
        `VMICRO16_ALU_BR,
        `VMICRO16_ALU_NOP:          c = {DATA_WIDTH{1'b0}};
        // load/store addresses (use value in rd2)
        `VMICRO16_ALU_LW,
        `VMICRO16_ALU_SW:           c = b;
        // bitwise operations
        `VMICRO16_ALU_BIT_OR:       c = a | b;
        `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
        `VMICRO16_ALU_BIT_AND:      c = a & b;
        `VMICRO16_ALU_BIT_NOT:      c = ~(b);
        `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
        `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;

        `VMICRO16_ALU_MOV:          c = b;
        `VMICRO16_ALU_MOVI:         c = b;
        `VMICRO16_ALU_MOVI_L:       c = b;

        `VMICRO16_ALU_ARITH_UADD:   c = a + b;
        `VMICRO16_ALU_ARITH_USUB:   c = a - b;
        // TODO: ALU should have simm5 as input
        `VMICRO16_ALU_ARITH_UADDI:  c = a + b;

        `ifdef DEF_ALU_HW_MULT
            `VMICRO16_ALU_MULT:  c = a * b;
        `endif

        `VMICRO16_ALU_ARITH_SADD:   c = $signed(a) + $signed(b);
        `VMICRO16_ALU_ARITH_SSUB:   c = $signed(a) - $signed(b);
        // TODO: ALU should have simm5 as input
        `VMICRO16_ALU_ARITH_SSUBI:  c = $signed(a) - $signed(b);

        `VMICRO16_ALU_CMP: begin
            // TODO: Do a-b in 17-bit register
            //       Set zero, overflow, carry, signed bits in result
            cmp_tmp = a - b;
            c = 0;

            // N   Negative condition code flag
            // Z   Zero condition code flag
            // C   Carry condition code flag
            // V   Overflow condition code flag
            c[`VMICRO16_SFLAG_N] = cmp_tmp[TOP_BIT];
            c[`VMICRO16_SFLAG_Z] = (cmp_tmp == 0);
            c[`VMICRO16_SFLAG_C] = 0; //cmp_tmp[TOP_BIT+1]; // not used
```

```
            // Overflow flag
            // https://stackoverflow.com/questions/30957188/
            // https://github.com/bendl/prco304/blob/master/prco_core/rtl/prco_alu.v#L50
            case(cmp_tmp[TOP_BIT+1:TOP_BIT])
                2'b01:   c[`VMICRO16_SFLAG_V] = 1;
                2'b10:   c[`VMICRO16_SFLAG_V] = 1;
                default: c[`VMICRO16_SFLAG_V] = 0;
            endcase

            $display($time, "\tC%02h: ALU CMP: %h %h = %h = %b", CORE_ID, a, b, cmp_tmp, c[3:0]);
        end

        `VMICRO16_ALU_SETC: c = { {15{1'b0}}, r_setc };

        // TODO: Parameterise
        default: begin
            $display($time, "\tALU: unknown op: %h", op);
            c       = 0;
            cmp_tmp = 0;
        end
                endcase
                end

    branch setc_check (
        .flags      (flags),
        .cond       (b[7:0]),
        .en         (r_setc)
    );
endmodule

// flags = 4 bit r_cmp_flags register
// cond  = 8 bit VMICRO16_OP_BR_? value. See vmicro16_isa.v
module branch (
    input [3:0] flags,
    input [7:0] cond,
    output reg  en
);
    always @(*)
        case (cond)
            `VMICRO16_OP_BR_U:  en = 1; `VMICRO16_OP_BR_U:  en = 1;
            `VMICRO16_OP_BR_E:  en = (flags[`VMICRO16_SFLAG_Z] == 1);
            `VMICRO16_OP_BR_NE: en = (flags[`VMICRO16_SFLAG_Z] == 0);
            `VMICRO16_OP_BR_G:  en = (flags[`VMICRO16_SFLAG_Z] == 0) &&
                                     (flags[`VMICRO16_SFLAG_N] == flags[`VMICRO16_SFLAG_V]);
            `VMICRO16_OP_BR_L:  en = (flags[`VMICRO16_SFLAG_Z] != flags[`VMICRO16_SFLAG_N]);
            `VMICRO16_OP_BR_GE: en = (flags[`VMICRO16_SFLAG_Z] == flags[`VMICRO16_SFLAG_N]);
            `VMICRO16_OP_BR_LE: en = (flags[`VMICRO16_SFLAG_Z] == 1) ||
                                     (flags[`VMICRO16_SFLAG_N] != flags[`VMICRO16_SFLAG_V]);
            default:            en = 0;
        endcase
endmodule



module vmicro16_core # (
    parameter DATA_WIDTH        = 16,
    parameter MEM_INSTR_DEPTH   = 64,
    parameter MEM_SCRATCH_DEPTH = 64,
    parameter MEM_WIDTH         = 16,

    parameter CORE_ID           = 3'h0
) (
    input        clk,
    input        reset,

    output [7:0] dbug,

    // interrupt sources
    input  [`DEF_NUM_INT-1:0]             ints,
    input  [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_data,
    output [`DEF_NUM_INT-1:0]             ints_ack,

    // APB master to slave interface (apb_intercon)
    output  [`APB_WIDTH-1:0]    w_PADDR,
    output                      w_PWRITE,
    output                      w_PSELx,
    output                      w_PENABLE,
    output  [DATA_WIDTH-1:0]    w_PWDATA,
    input   [DATA_WIDTH-1:0]    w_PRDATA,
    input                       w_PREADY
);
    localparam STATE_IF = 0;
    localparam STATE_R1 = 1;
    localparam STATE_R2 = 2;
    localparam STATE_ME = 3;
    localparam STATE_WB = 4;
    localparam STATE_FE = 5;
    localparam STATE_IDLE = 6;
    localparam STATE_HALT = 7;
    reg  [2:0] r_state = STATE_IF;

    reg  [DATA_WIDTH-1:0] r_pc          = 16'h0000;
    reg  [DATA_WIDTH-1:0] r_pc_saved    = 16'h0000;
    reg  [DATA_WIDTH-1:0] r_instr       = 16'h0000;
    wire [DATA_WIDTH-1:0] w_mem_instr_out;
    wire                  w_halt;

    assign dbug = {7'h00, w_halt};

```

```
1008        wire [4:0]            r_instr_opcode;
1009        wire [4:0]            r_instr_alu_op;
1010        wire [2:0]            r_instr_rsd;
1011        wire [2:0]            r_instr_rsa;
1012        reg  [DATA_WIDTH-1:0] r_instr_rdd = 0;
1013        reg  [DATA_WIDTH-1:0] r_instr_rda = 0;
1014        wire [3:0]            r_instr_imm4;
1015        wire [7:0]            r_instr_imm8;
1016        wire [4:0]            r_instr_simm5;
1017        wire                  r_instr_has_imm4;
1018        wire                  r_instr_has_imm8;
1019        wire                  r_instr_has_we;
1020        wire                  r_instr_has_br;
1021        wire                  r_instr_has_cmp;
1022        wire                  r_instr_has_mem;
1023        wire                  r_instr_has_mem_we;
1024        wire                  r_instr_halt;
1025        wire                  r_instr_has_lwex;
1026        wire                  r_instr_has_swex;
1027
1028        wire [DATA_WIDTH-1:0] r_alu_out;
1029
1030        wire [DATA_WIDTH-1:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
1031        wire [DATA_WIDTH-1:0] r_mem_scratch_in   = r_instr_rdd;
1032        wire [DATA_WIDTH-1:0] r_mem_scratch_out;
1033        wire                  r_mem_scratch_we   = r_instr_has_mem_we && (r_state == STATE_ME);
1034        reg                   r_mem_scratch_req  = 0;
1035        wire                  r_mem_scratch_busy;
1036
1037        reg  [2:0]            r_reg_rs1 = 0;
1038        wire [DATA_WIDTH-1:0] r_reg_rd1_s;
1039        wire [DATA_WIDTH-1:0] r_reg_rd1_i;
1040        wire [DATA_WIDTH-1:0] r_reg_rd1 = regs_use_int ? r_reg_rd1_i : r_reg_rd1_s;
1041        //wire [15:0] r_reg_rd2;
1042        wire [DATA_WIDTH-1:0] r_reg_wd = (r_instr_has_mem) ? r_mem_scratch_out : r_alu_out;
1043        wire                  r_reg_we = r_instr_has_we && (r_state == STATE_WB);
1044
1045        // branching
1046        wire        w_intr;
1047        wire        w_branch_en;
1048        wire        w_branching   = r_instr_has_br && w_branch_en;
1049        reg  [3:0]  r_cmp_flags   = 4'h00; // N, Z, C, V
1050
1051        always @(r_cmp_flags)
1052            $display($time, "\tC%02h:\tALU CMP: %b", CORE_ID, r_cmp_flags);
1053
1054        // 2 cycle register fetch
1055        always @(*) begin
1056            r_reg_rs1 = 0;
1057            if (r_state == STATE_R1)
1058                r_reg_rs1 = r_instr_rsd;
1059            else if (r_state == STATE_R2)
1060                r_reg_rs1 = r_instr_rsa;
1061            else
1062                r_reg_rs1 = 3'h0;
1063        end
1064
1065        wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_vector;
1066        wire [`DEF_NUM_INT-1:0]             ints_mask;
1067        wire                                has_int = ints & ints_mask;
1068        reg int_pending = 0;
1069        reg int_pending_ack = 0;
1070        reg regs_use_int = 0;
1071        always @(posedge clk)
1072            if (int_pending_ack)
1073                // We've now branched to the isr
1074                int_pending <= 0;
1075            else if (has_int)
1076                // Notify fsm to switch to the ints_vector at the last stage
1077                int_pending <= 1;
1078            else if (w_intr)
1079                // Return to Interrupt instruction called,
1080                //   so we've finished with the interrupt
1081                int_pending <= 0;
1082
1083
1084        // cpu state machine
1085        always @(posedge clk)
1086            if (reset) begin
1087                r_pc              <= 0;
1088                r_state           <= STATE_IF;
1089                r_instr           <= 0;
1090                r_mem_scratch_req <= 0;
1091                r_instr_rdd       <= 0;
1092                r_instr_rda       <= 0;
1093            end
1094            else begin
1095
1096                if (r_state == STATE_IF) begin
1097                    if (w_halt) begin
1098                        $display("");
1099                        $display("");
1100                        $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
1101                        r_state <= STATE_HALT;
1102                    end else begin
1103                        r_instr <= w_mem_instr_out;
1104
1105                        $display("");
1106                        $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
```

```
1107                    $display($time, "\tC%02h: INSTR: %h", CORE_ID, w_mem_instr_out);
1108
1109
1110                    r_state <= STATE_R1;
1111                end
1112            end
1113            else if (r_state == STATE_R1) begin
1114                // primary operand
1115                r_instr_rdd <= r_reg_rd1;
1116                r_state     <= STATE_R2;
1117            end
1118            else if (r_state == STATE_R2) begin
1119                // Choose secondary operand (register or immediate)
1120                if      (r_instr_has_imm8)  r_instr_rda <= r_instr_imm8;
1121                else if (r_instr_has_imm4)  r_instr_rda <= r_reg_rd1 + r_instr_imm4;
1122                else                        r_instr_rda <= r_reg_rd1;
1123
1124                if (r_instr_has_mem) begin
1125                    r_state           <= STATE_ME;
1126                    // Pulse req
1127                    r_mem_scratch_req <= 1;
1128                end else
1129                    r_state <= STATE_WB;
1130            end
1131            else if (r_state == STATE_ME) begin
1132                // Pulse req
1133                r_mem_scratch_req <= 0;
1134                // Wait for MMU to finish
1135                if (!r_mem_scratch_busy)
1136                    r_state <= STATE_WB;
1137            end
1138            else if (r_state == STATE_WB) begin
1139                if (r_instr_has_cmp) begin
1140                    $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
1141                    r_cmp_flags <= r_alu_out[3:0];
1142                end
1143
1144                if (int_pending) begin
1145                    $display($time, "\tC%02h: Jumping to ISR: %h", CORE_ID, ints_vector[0 +: `DATA_WIDTH]);
1146                    // TODO: check bounds
1147                    // Save state
1148                    r_pc_saved      <= r_pc + 1;
1149                    regs_use_int    <= 1;
1150                    int_pending_ack <= 1;
1151                    // Jump to ISR
1152                    r_pc            <= ints_vector[0 +: `DATA_WIDTH];
1153                end else if (w_intr) begin
1154                    $display($time, "\tC%02h: Returning from ISR: %h", CORE_ID, r_pc_saved);
1155                    r_pc            <= r_pc_saved;
1156                    regs_use_int    <= 0;
1157                    int_pending_ack <= 0;
1158                end else if (w_branching) begin
1159                    $display($time, "\tC%02h: branching to %h", CORE_ID, r_instr_rdd);
1160                    r_pc            <= r_instr_rdd;
1161                    int_pending_ack <= 0;
1162                end
1163                else if (r_pc < (MEM_INSTR_DEPTH-1)) begin
1164                    r_pc            <= r_pc + 1;
1165                    int_pending_ack <= 0;
1166                end
1167
1168                r_state <= STATE_FE;
1169            end
1170            else if (r_state == STATE_FE) begin
1171                r_state <= STATE_IF;
1172            end
1173            else if (r_state == STATE_HALT) begin
1174            end
1175        end
1176
1177    // Instruction ROM
1178    vmicro16_bram # (
1179        .MEM_WIDTH      (DATA_WIDTH),
1180        .MEM_DEPTH      (MEM_INSTR_DEPTH),
1181        .CORE_ID        (CORE_ID),
1182        .USE_INITS      (1),
1183        .NAME           ("INSTR_MEM")
1184    ) mem_instr (
1185        .clk            (clk),
1186        .reset          (reset),
1187        // port 1
1188        .mem_addr       (r_pc),
1189        .mem_in         (0),
1190        .mem_we         (1'b0),  // ROM
1191        .mem_out        (w_mem_instr_out)
1192    );
1193
1194    // MMU
1195    vmicro16_core_mmu # (
1196        .MEM_WIDTH      (DATA_WIDTH),
1197        .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
1198        .CORE_ID        (CORE_ID)
1199    ) mmu (
1200        .clk            (clk),
1201        .reset          (reset),
1202        .req            (r_mem_scratch_req),
1203        .busy           (r_mem_scratch_busy),
1204        // interrupts
1205        .ints_vector    (ints_vector),
```

```
1206            .ints_mask      (ints_mask),
1207            // port 1
1208            .mmu_addr       (r_mem_scratch_addr),
1209            .mmu_in         (r_mem_scratch_in),
1210            .mmu_we         (r_mem_scratch_we),
1211            .mmu_lwex       (r_instr_has_lwex),
1212            .mmu_swex       (r_instr_has_swex),
1213            .mmu_out        (r_mem_scratch_out),
1214            // APB maste    r to slave
1215            .M_PADDR        (w_PADDR),
1216            .M_PWRITE       (w_PWRITE),
1217            .M_PSELx        (w_PSELx),
1218            .M_PENABLE      (w_PENABLE),
1219            .M_PWDATA       (w_PWDATA),
1220            .M_PRDATA       (w_PRDATA),
1221            .M_PREADY       (w_PREADY)
1222        );
1223
1224        // Instruction decoder
1225        vmicro16_dec dec (
1226            // input
1227            .instr          (r_instr),
1228            // output async
1229            .opcode         (),
1230            .rd             (r_instr_rsd),
1231            .ra             (r_instr_rsa),
1232            .imm4           (r_instr_imm4),
1233            .imm8           (r_instr_imm8),
1234            .imm12          (),
1235            .simm5          (r_instr_simm5),
1236            .alu_op         (r_instr_alu_op),
1237            .has_imm4       (r_instr_has_imm4),
1238            .has_imm8       (r_instr_has_imm8),
1239            .has_we         (r_instr_has_we),
1240            .has_br         (r_instr_has_br),
1241            .has_cmp        (r_instr_has_cmp),
1242            .has_mem        (r_instr_has_mem),
1243            .has_mem_we     (r_instr_has_mem_we),
1244            .halt           (w_halt),
1245            .intr           (w_intr),
1246            .has_lwex       (r_instr_has_lwex),
1247            .has_swex       (r_instr_has_swex)
1248        );
1249
1250        // Software registers
1251        vmicro16_regs # (
1252            .CORE_ID    (CORE_ID),
1253            .CELL_WIDTH (`DATA_WIDTH)
1254        ) regs (
1255            .clk        (clk),
1256            .reset      (reset),
1257            // async port 0
1258            .rs1        (r_reg_rs1),
1259            .rd1        (r_reg_rd1_s),
1260            // async port 1
1261            //.rs2        (),
1262            //.rd2        (),
1263            // write port
1264            .we         (r_reg_we && ~regs_use_int),
1265            .ws1        (r_instr_rsd),
1266            .wd         (r_reg_wd)
1267        );
1268
1269        // Interrupt replacement registers
1270        vmicro16_regs # (
1271            .CORE_ID    (CORE_ID),
1272            .CELL_WIDTH (`DATA_WIDTH),
1273            .DEBUG_NAME ("REGSINT")
1274        ) regs_intr (
1275            .clk        (clk),
1276            .reset      (reset),
1277            // async port 0
1278            .rs1        (r_reg_rs1),
1279            .rd1        (r_reg_rd1_i),
1280            // async port 1
1281            //.rs2        (),
1282            //.rd2        (),
1283            // write port
1284            .we         (r_reg_we && regs_use_int),
1285            .ws1        (r_instr_rsd),
1286            .wd         (r_reg_wd)
1287        );
1288
1289        // ALU
1290        vmicro16_alu # (
1291            .CORE_ID(CORE_ID)
1292        ) alu (
1293            .op         (r_instr_alu_op),
1294            .a          (r_instr_rdd),
1295            .b          (r_instr_rda),
1296            .flags      (r_cmp_flags),
1297            // async output
1298            .c          (r_alu_out)
1299        );
1300
1301        branch branch_check (
1302            .flags      (r_cmp_flags),
1303            .cond       (r_instr_imm8),
1304            .en         (w_branch_en)
```

```
1305        );
1306
1307    endmodule
```

vmicro16_soc.v

```
   1    //
   2    //
   3
   4    `include "vmicro16_soc_config.v"
   5    `include "clog2.v"
   6
   7    module timer_apb # (
   8        parameter CLK_HZ = 50_000_000
   9    ) (
  10        input clk,
  11        input reset,
  12
  13        input clk_en,
  14
  15        // 0 16-bit value   R/W
  16        // 1 16-bit control R    b0 = start, b1 = reset
  17        // 2 16-bit prescaler
  18        input      [1:0]                S_PADDR,
  19
  20        input                           S_PWRITE,
  21        input                           S_PSELx,
  22        input                           S_PENABLE,
  23        input      [`DATA_WIDTH-1:0]    S_PWDATA,
  24
  25        output reg [`DATA_WIDTH-1:0]    S_PRDATA,
  26        output                          S_PREADY,
  27
  28        output out,
  29        output [`DATA_WIDTH-1:0] int_data
  30    );
  31        //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
  32        assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
  33        wire   en       = (S_PSELx & S_PENABLE);
  34        wire   we       = (en & S_PWRITE);
  35
  36        reg [`DATA_WIDTH-1:0] r_counter = 0;
  37        reg [`DATA_WIDTH-1:0] r_load = 0;
  38        reg [`DATA_WIDTH-1:0] r_pres = 0;
  39        reg [`DATA_WIDTH-1:0] r_ctrl = 0;
  40
  41        localparam CTRL_START = 0;
  42        localparam CTRL_RESET = 1;
  43
  44        localparam ADDR_LOAD = 2'b00;
  45        localparam ADDR_CTRL = 2'b01;
  46        localparam ADDR_PRES = 2'b10;
  47
  48        always @(*) begin
  49            S_PRDATA = 0;
  50            if (en)
  51                case(S_PADDR)
  52                    ADDR_LOAD: S_PRDATA = r_counter;
  53                    ADDR_CTRL: S_PRDATA = r_ctrl;
  54                    //ADDR_CTRL: S_PRDATA = r_pres;
  55                    default:   S_PRDATA = 0;
  56                endcase
  57        end
  58
  59        // prescaler counts from r_pres to 0, emitting a stb signal
  60        //   to enable the r_counter step
  61        reg [`DATA_WIDTH-1:0] r_pres_counter = 0;
  62        wire counter_en = (r_pres_counter == 0);
  63        always @(posedge clk)
  64            if (r_pres_counter == 0)
  65                r_pres_counter <= r_pres;
  66            else
  67                r_pres_counter <= r_pres_counter - 1;
  68
  69        always @(posedge clk)
  70            if (we)
  71                case(S_PADDR)
  72                    // Write to the load register:
  73                    //   Set load register
  74                    //   Set counter register
  75                    ADDR_LOAD: begin
  76                        r_load          <= S_PWDATA;
  77                        r_counter       <= S_PWDATA;
  78                        $display($time, "\t\ttimr0: WRITE LOAD: %h", S_PWDATA);
  79                    end
  80                    ADDR_CTRL: begin
  81                        r_ctrl   <= S_PWDATA;
  82                        $display($time, "\t\ttimr0: WRITE CTRL: %h", S_PWDATA);
  83                    end
  84                    ADDR_PRES: begin
  85                        r_pres   <= S_PWDATA;
  86                        $display($time, "\t\ttimr0: WRITE PRES: %h", S_PWDATA);
  87                    end
  88                endcase
  89            else
```

```
90              if (r_ctrl[CTRL_START]) begin
91                  if (r_counter == 0)
92                      r_counter <= r_load;
93                  else if(counter_en)
94                      r_counter <= r_counter -1;
95              end else if (r_ctrl[CTRL_RESET])
96                  r_counter <= r_load;
97
98      // generate the output pulse when r_counter == 0
99      //   out = (counter reached zero && counter started)
100     assign out      = (r_counter == 0) && r_ctrl[CTRL_START];
101     assign int_data = {`DATA_WIDTH{1'b1}};
102 endmodule
103
104 // Shared memory with hardware monitor (LWEX/SWEX)
105
106
107 module vmicro16_bram_ex_apb # (
108     parameter BUS_WIDTH    = 16,
109     parameter MEM_WIDTH    = 16,
110     parameter MEM_DEPTH    = 64,
111     parameter CORE_ID_BITS = 3,
112     parameter SWEX_SUCCESS = 16'h0000,
113     parameter SWEX_FAIL    = 16'h0001
114 ) (
115     input clk,
116     input reset,
117
118     // |19    |18    |16             |15          0|
119     // | LWEX | SWEX | 3 bit CORE_ID |     S_PADDR |
120     input  [`APB_WIDTH-1:0]         S_PADDR,
121
122     input                           S_PWRITE,
123     input                           S_PSELx,
124     input                           S_PENABLE,
125     input  [MEM_WIDTH-1:0]          S_PWDATA,
126
127     output reg [MEM_WIDTH-1:0]      S_PRDATA,
128     output                          S_PREADY
129 );
130     // exclusive flag checks
131     wire [MEM_WIDTH-1:0] mem_out;
132     wire [MEM_WIDTH-1:0] mem_out_ex;
133     reg                  swex_success = 0;
134
135     localparam ADDR_BITS = `clog2(MEM_DEPTH);
136
137     // hack to create a 1 clock delay to S_PREADY
138     // for bram to be ready
139     reg cdelay = 1;
140     always @(posedge clk)
141         if (S_PSELx)
142             cdelay <= 0;
143         else
144             cdelay <= 1;
145
146     //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
147     assign S_PREADY = (S_PSELx & S_PENABLE & (!cdelay)) ? 1'b1       : 1'b0;
148     assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
149     wire   en       = (S_PSELx & S_PENABLE);
150
151     // Similar to:
152     //   http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204f/Cihbghef.html
153
154     // mem_wd is the CORE_ID sent in bits [18:16]
155     localparam TOP_BIT_INDEX         = `APB_WIDTH -1;
156     localparam PADDR_CORE_ID_MSB     = TOP_BIT_INDEX - 2;
157     localparam PADDR_CORE_ID_LSB     = PADDR_CORE_ID_MSB - (CORE_ID_BITS-1);
158
159     // [LWEX, CORE_ID, mem_addr] from S_PADDR
160     wire                    lwex        = S_PADDR[TOP_BIT_INDEX];
161     wire                    swex        = S_PADDR[TOP_BIT_INDEX-1];
162     wire [CORE_ID_BITS-1:0] core_id     = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
163     // CORE_ID to write to ex_flags register
164     wire [ADDR_BITS-1:0]    mem_addr    = S_PADDR[ADDR_BITS-1:0];
165
166     wire [CORE_ID_BITS:0]   ex_flags_read;
167     wire                    is_locked      = |ex_flags_read;
168     wire                    is_locked_self = is_locked && (core_id == (ex_flags_read-1));
169
170     // Check exclusive access flags
171     always @(*) begin
172         swex_success = 0;
173         if (en)
174             if (swex)
175                 if (is_locked && !is_locked_self)
176                     // someone else has locked it
177                     swex_success = 0;
178                 else if (is_locked && is_locked_self)
179                     swex_success = 1;
180     end
181
182     always @(*)
183         if (swex)
184             if (swex_success)
185                 S_PRDATA = SWEX_SUCCESS;
186             else
187                 S_PRDATA = SWEX_FAIL;
188         else
```

```
189             S_PRDATA = mem_out;
190
191     wire reg_we = en && ((lwex && !is_locked)
192                     || (swex && swex_success));
193
194     reg  [CORE_ID_BITS:0] reg_wd;
195     always @(*) begin
196         reg_wd = {{CORE_ID_BITS}{1'b0}};
197
198         if (en)
199             // if wanting to lock the addr
200             if (lwex)
201                 // and not already locked
202                 if (!is_locked) begin
203                     reg_wd = (core_id + 1);
204                 end
205             else if (swex)
206                 if (is_locked && is_locked_self)
207                     reg_wd = {{CORE_ID_BITS}{1'b0}};
208     end
209
210     // Exclusive flag for each memory cell
211     vmicro16_bram # (
212         .MEM_WIDTH  (CORE_ID_BITS + 1),
213         .MEM_DEPTH  (MEM_DEPTH),
214         .USE_INITS  (0),
215         .NAME       ("rexram")
216     ) ram_exflags (
217         .clk        (clk),
218         .reset      (reset),
219
220         .mem_addr   (mem_addr),
221         .mem_in     (reg_wd),
222         .mem_we     (reg_we),
223         .mem_out    (ex_flags_read)
224     );
225
226     always @(*)
227         if (S_PSELx && S_PENABLE)
228             $display($time, "\t\tBRAMex[%h] READ %h\tCORE: %h", mem_addr, mem_out, S_PADDR[16 +: CORE_ID_BITS]);
229
230     always @(posedge clk)
231         if (we)
232             $display($time, "\t\tBRAMex[%h] WRITE %h\tCORE: %h", mem_addr, S_PWDATA, S_PADDR[16 +: CORE_ID_BITS]);
233
234     vmicro16_bram # (
235         .MEM_WIDTH  (MEM_WIDTH),
236         .MEM_DEPTH  (MEM_DEPTH),
237         .USE_INITS  (0),
238         .NAME       ("BRAMexinst")
239     ) bram_apb (
240         .clk        (clk),
241         .reset      (reset),
242
243         .mem_addr   (mem_addr),
244         .mem_in     (S_PWDATA),
245         .mem_we     (we && swex_success),
246         .mem_out    (mem_out)
247     );
248 endmodule
249
250
251
252
253 module vmicro16_soc (
254     input clk,
255     input reset,
256
257     //input  uart_rx,
258     output                          uart_tx,
259     output [`APB_GPIO0_PINS-1:0]    gpio0,
260     output [`APB_GPIO1_PINS-1:0]    gpio1,
261     output [`APB_GPIO2_PINS-1:0]    gpio2,
262
263     output     [7:0]                dbug0,
264     output     [`CORES*8:0]         dbug1
265 );
266     genvar di;
267     generate for(di = 0; di < `CORES; di = di + 1) begin : gen_dbug0
268         assign dbug0[di] = dbug1[di*8];
269     end
270     endgenerate
271
272     // Peripherals (master to slave)
273      wire [`APB_WIDTH-1:0]          M_PADDR;
274      wire                           M_PWRITE;
275      wire [`SLAVES-1:0]             M_PSELx;  // not shared
276      wire                           M_PENABLE;
277      wire [`DATA_WIDTH-1:0]         M_PWDATA;
278      wire [`SLAVES*`DATA_WIDTH-1:0] M_PRDATA; // input to intercon
279      wire [`SLAVES-1:0]             M_PREADY; // input
280
281     // Master apb interfaces
282      wire [`CORES*`APB_WIDTH-1:0]   w_PADDR;
283      wire [`CORES-1:0]              w_PWRITE;
284      wire [`CORES-1:0]              w_PSELx;
285      wire [`CORES-1:0]              w_PENABLE;
286      wire [`CORES*`DATA_WIDTH-1:0]  w_PWDATA;
287      wire [`CORES*`DATA_WIDTH-1:0]  w_PRDATA;
```

```
288         wire [`CORES-1:0]              w_PREADY;
289
290     // Interrupts
291     wire [`DEF_NUM_INT-1:0]              ints;
292     wire [`DEF_NUM_INT*`DATA_WIDTH-1:0]  ints_data;
293     assign ints[7:1] = 0;
294     assign ints_data[`DEF_NUM_INT*`DATA_WIDTH-1:`DATA_WIDTH] = {`DEF_NUM_INT*(`DATA_WIDTH-1){1'b0}};
295
296
297
298     apb_intercon_s # (
299         .MASTER_PORTS(`CORES),
300         .SLAVE_PORTS (`SLAVES),
301         .BUS_WIDTH   (`APB_WIDTH),
302         .DATA_WIDTH  (`DATA_WIDTH)
303     ) apb (
304         .clk        (clk),
305         .reset      (reset),
306         // APB master to slave
307         .S_PADDR    (w_PADDR),
308         .S_PWRITE   (w_PWRITE),
309         .S_PSELx    (w_PSELx),
310         .S_PENABLE  (w_PENABLE),
311         .S_PWDATA   (w_PWDATA),
312         .S_PRDATA   (w_PRDATA),
313         .S_PREADY   (w_PREADY),
314         // shared bus
315         .M_PADDR    (M_PADDR),
316         .M_PWRITE   (M_PWRITE),
317         .M_PSELx    (M_PSELx),
318         .M_PENABLE  (M_PENABLE),
319         .M_PWDATA   (M_PWDATA),
320         .M_PRDATA   (M_PRDATA),
321         .M_PREADY   (M_PREADY)
322     );
323
324
325
326     vmicro16_gpio_apb # (
327         .BUS_WIDTH  (`APB_WIDTH),
328         .DATA_WIDTH (`DATA_WIDTH),
329         .PORTS      (`APB_GPIO0_PINS),
330         .NAME       ("GPIO0")
331     ) gpio0_apb (
332         .clk        (clk),
333         .reset      (reset),
334         // apb slave to master interface
335         .S_PADDR    (M_PADDR),
336         .S_PWRITE   (M_PWRITE),
337         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO0]),
338         .S_PENABLE  (M_PENABLE),
339         .S_PWDATA   (M_PWDATA),
340         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO0*`DATA_WIDTH +: `DATA_WIDTH]),
341         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
342         .gpio       (gpio0)
343     );
344
345     // GPIO1 for Seven segment displays (16 pin)
346
347
348     vmicro16_gpio_apb # (
349         .BUS_WIDTH  (`APB_WIDTH),
350         .DATA_WIDTH (`DATA_WIDTH),
351         .PORTS      (`APB_GPIO1_PINS),
352         .NAME       ("GPIO1")
353     ) gpio1_apb (
354         .clk        (clk),
355         .reset      (reset),
356         // apb slave to master interface
357         .S_PADDR    (M_PADDR),
358         .S_PWRITE   (M_PWRITE),
359         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO1]),
360         .S_PENABLE  (M_PENABLE),
361         .S_PWDATA   (M_PWDATA),
362         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO1*`DATA_WIDTH +: `DATA_WIDTH]),
363         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
364         .gpio       (gpio1)
365     );
366
367     // GPIO2 for Seven segment displays (8 pin)
368
369
370     vmicro16_gpio_apb # (
371         .BUS_WIDTH  (`APB_WIDTH),
372         .DATA_WIDTH (`DATA_WIDTH),
373         .PORTS      (`APB_GPIO2_PINS),
374         .NAME       ("GPIO2")
375     ) gpio2_apb (
376         .clk        (clk),
377         .reset      (reset),
378         // apb slave to master interface
379         .S_PADDR    (M_PADDR),
380         .S_PWRITE   (M_PWRITE),
381         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO2]),
382         .S_PENABLE  (M_PENABLE),
383         .S_PWDATA   (M_PWDATA),
384         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO2*`DATA_WIDTH +: `DATA_WIDTH]),
385         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
386         .gpio       (gpio2)
```

```
387     );
388
389
390
391     apb_uart_tx uart0_apb (
392         .clk        (clk),
393         .reset      (reset),
394         // apb slave to master interface
395         .S_PADDR    (M_PADDR),
396         .S_PWRITE   (M_PWRITE),
397         .S_PSELx    (M_PSELx[`APB_PSELX_UART0]),
398         .S_PENABLE  (M_PENABLE),
399         .S_PWDATA   (M_PWDATA),
400         .S_PRDATA   (M_PRDATA[`APB_PSELX_UART0*`DATA_WIDTH +: `DATA_WIDTH]),
401         .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
402         // uart wires
403         .tx_wire    (uart_tx),
404         .rx_wire    (uart_rx)
405     );
406
407
408
409     timer_apb timr0 (
410         .clk        (clk),
411         .reset      (reset),
412         // apb slave to master interface
413         .S_PADDR    (M_PADDR),
414         .S_PWRITE   (M_PWRITE),
415         .S_PSELx    (M_PSELx[`APB_PSELX_TIMR0]),
416         .S_PENABLE  (M_PENABLE),
417         .S_PWDATA   (M_PWDATA),
418         .S_PRDATA   (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
419         .S_PREADY   (M_PREADY[`APB_PSELX_TIMR0]),
420         //
421         .out        (ints     [`DEF_INT_TIMR0]),
422         .int_data   (ints_data[`DEF_INT_TIMR0*`DATA_WIDTH +: `DATA_WIDTH])
423     );
424
425     // Shared register set for system-on-chip info
426     // R0 = number of cores
427
428
429     vmicro16_regs_apb # (
430         .BUS_WIDTH          (`APB_WIDTH),
431         .DATA_WIDTH         (`DATA_WIDTH),
432         .CELL_DEPTH         (8),
433         .PARAM_DEFAULTS_R0  (`CORES),
434         .PARAM_DEFAULTS_R1  (`SLAVES)
435     ) regs0_apb (
436         .clk        (clk),
437         .reset      (reset),
438         // apb slave to master interface
439         .S_PADDR    (M_PADDR),
440         .S_PWRITE   (M_PWRITE),
441         .S_PSELx    (M_PSELx[`APB_PSELX_REGS0]),
442         .S_PENABLE  (M_PENABLE),
443         .S_PWDATA   (M_PWDATA),
444         .S_PRDATA   (M_PRDATA[`APB_PSELX_REGS0*`DATA_WIDTH +: `DATA_WIDTH]),
445         .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
446     );
447
448
449
450     vmicro16_bram_ex_apb # (
451         .BUS_WIDTH    (`APB_WIDTH),
452         .MEM_WIDTH    (`DATA_WIDTH),
453         .MEM_DEPTH    (`APB_BRAM0_CELLS),
454         .CORE_ID_BITS (`clog2(`CORES))
455     ) bram_apb (
456         .clk        (clk),
457         .reset      (reset),
458         // apb slave to master interface
459         .S_PADDR    (M_PADDR),
460         .S_PWRITE   (M_PWRITE),
461         .S_PSELx    (M_PSELx[`APB_PSELX_BRAM0]),
462         .S_PENABLE  (M_PENABLE),
463         .S_PWDATA   (M_PWDATA),
464         .S_PRDATA   (M_PRDATA[`APB_PSELX_BRAM0*`DATA_WIDTH +: `DATA_WIDTH]),
465         .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
466     );
467
468     genvar i;
469     generate for(i = 0; i < `CORES; i = i + 1) begin : cores
470
471         vmicro16_core # (
472             .CORE_ID            (i),
473             .DATA_WIDTH         (`DATA_WIDTH),
474
475             .MEM_INSTR_DEPTH    (`DEF_MEM_INSTR_DEPTH),
476             .MEM_SCRATCH_DEPTH  (`DEF_MMU_TIM0_CELLS)
477         ) c1 (
478             .clk        (clk),
479             .reset      (reset),
480             .dbug       (dbug1[i*8 +: 8]),
481
482             .ints       (ints),
483             .ints_data  (ints_data),
484
485             .w_PADDR    (w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH] ),
```

```
            .w_PWRITE   (w_PWRITE  [i]                        ),
            .w_PSELx    (w_PSELx   [i]                        ),
            .w_PENABLE  (w_PENABLE [i]                        ),
            .w_PWDATA   (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH] ),
            .w_PRDATA   (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH] ),
            .w_PREADY   (w_PREADY  [i]                        )
        );
    end
    endgenerate


endmodule
```

vmicro16_isa.v

```
// Vmicro16 multi-core instruction set
`include "vmicro16_soc_config.v"

// TODO: Remove NOP by making a register write/read always 0
`define VMICRO16_OP_SPCL         5'b00000
`define VMICRO16_OP_LW           5'b00001
`define VMICRO16_OP_SW           5'b00010
`define VMICRO16_OP_BIT          5'b00011
`define VMICRO16_OP_BIT_OR       5'b00000
`define VMICRO16_OP_BIT_XOR      5'b00001
`define VMICRO16_OP_BIT_AND      5'b00010
`define VMICRO16_OP_BIT_NOT      5'b00011
`define VMICRO16_OP_BIT_LSHFT    5'b00100
`define VMICRO16_OP_BIT_RSHFT    5'b00101
`define VMICRO16_OP_MOV          5'b00100
`define VMICRO16_OP_MOVI         5'b00101
`define VMICRO16_OP_ARITH_U      5'b00110
`define VMICRO16_OP_ARITH_UADD   5'b11111
`define VMICRO16_OP_ARITH_USUB   5'b10000
`define VMICRO16_OP_ARITH_UADDI  5'b0????
`define VMICRO16_OP_ARITH_S      5'b00111
`define VMICRO16_OP_ARITH_SADD   5'b11111
`define VMICRO16_OP_ARITH_SSUB   5'b10000
`define VMICRO16_OP_ARITH_SSUBI  5'b0????
`define VMICRO16_OP_BR           5'b01000
`define VMICRO16_OP_CMP          5'b01001
`define VMICRO16_OP_SETC         5'b01010
`define VMICRO16_OP_MULT         5'b01011
`define VMICRO16_OP_LWEX         5'b01101
`define VMICRO16_OP_SWEX         5'b01110

// Special opcodes
`define VMICRO16_OP_SPCL_NOP     11'h000
`define VMICRO16_OP_SPCL_HALT    11'h001
`define VMICRO16_OP_SPCL_INTR    11'h002

// TODO: wasted upper nibble bits in BR
`define VMICRO16_OP_BR_U         8'h00
`define VMICRO16_OP_BR_E         8'h01
`define VMICRO16_OP_BR_NE        8'h02
`define VMICRO16_OP_BR_G         8'h03
`define VMICRO16_OP_BR_GE        8'h04
`define VMICRO16_OP_BR_L         8'h05
`define VMICRO16_OP_BR_LE        8'h06
`define VMICRO16_OP_BR_S         8'h07
`define VMICRO16_OP_BR_NS        8'h08

// flag bit positions
`define VMICRO16_SFLAG_N         4'h03
`define VMICRO16_SFLAG_Z         4'h02
`define VMICRO16_SFLAG_C         4'h01
`define VMICRO16_SFLAG_V         4'h00

// microcode operations
`define VMICRO16_ALU_BIT_OR      5'h00
`define VMICRO16_ALU_BIT_XOR     5'h01
`define VMICRO16_ALU_BIT_AND     5'h02
`define VMICRO16_ALU_BIT_NOT     5'h03
`define VMICRO16_ALU_BIT_LSHFT   5'h04
`define VMICRO16_ALU_BIT_RSHFT   5'h05
`define VMICRO16_ALU_LW          5'h06
`define VMICRO16_ALU_SW          5'h07
`define VMICRO16_ALU_NOP         5'h08
`define VMICRO16_ALU_MOV         5'h09
`define VMICRO16_ALU_MOVI        5'h0a
`define VMICRO16_ALU_MOVI_L      5'h0b
`define VMICRO16_ALU_ARITH_UADD  5'h0c
`define VMICRO16_ALU_ARITH_USUB  5'h0d
`define VMICRO16_ALU_ARITH_SADD  5'h0e
`define VMICRO16_ALU_ARITH_SSUB  5'h0f
`define VMICRO16_ALU_BR_U        5'h10
`define VMICRO16_ALU_BR_E        5'h11
`define VMICRO16_ALU_BR_NE       5'h12
`define VMICRO16_ALU_BR_G        5'h13
`define VMICRO16_ALU_BR_GE       5'h14
`define VMICRO16_ALU_BR_L        5'h15
`define VMICRO16_ALU_BR_LE       5'h16
`define VMICRO16_ALU_BR_S        5'h17
`define VMICRO16_ALU_BR_NS       5'h18
`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
```