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

ELEC5881M - Interim Report

**Ben David Lancaster**
Student ID: 201280376

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

Supervisor: Dr. David Cowell
Assessor: Mr David Moore

**University of Leeds**
School of Electrical and Electronic Engineering

June 27, 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: June 27, 2019

# Table of Contents

# Chapter 1

# Introduction



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

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

## 1.1 Why Multi-core?

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

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

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

## 1.2 Why RISC?

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

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

## 1.3 Why FPGA?

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

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

FPGAs, however, may not be suitable for prototyping all register-transistor logic (RTL) projects. Larger RTL projects, such as large commercial processors, may greatly exceed the logic cell resources available in today's high-end FPGA devices and may only be prototyped through silicon fabrication, which can be expensive. This resource limitation will not be problem as the project aims to produce a small and minimal design specifically for learning about multi-core architectures.

# Chapter 2

# Background



## 2.1 Amdahl's Law and Parallelism

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

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

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

## 2.2 Loosely and Tightly Coupled Processors

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

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

As loosely coupled system's nodes feature there own memory and IO modules, they generally perform better in cases where interaction between nodes is not prominent – each node can store a separate part of the software program in it's memory module allowing simultaneous executing of the program.

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

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

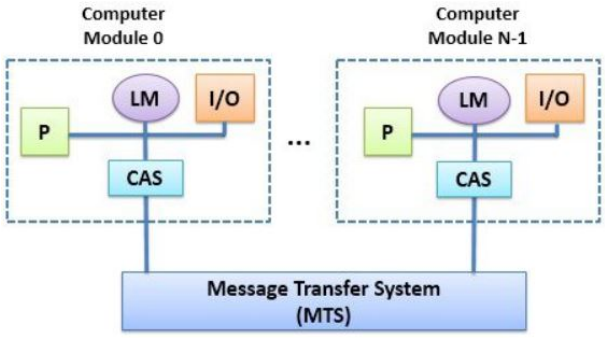


**Figure 2.1:** A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [**?** ].

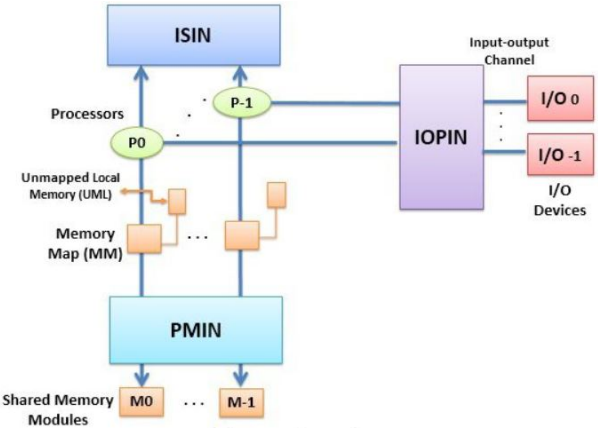


**Figure 2.2:** A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [**?** ].

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

## 2.3 Network-on-chip Architectures

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

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

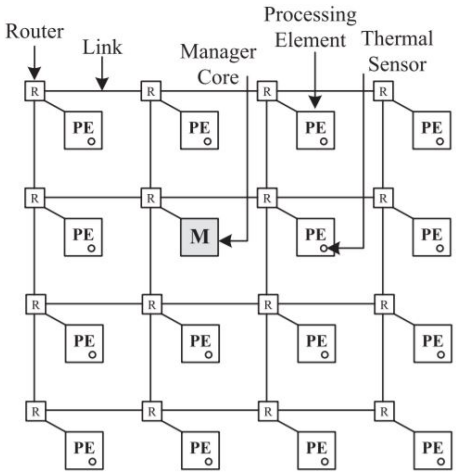


**Figure 2.3:** A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [**?** ].

# Chapter 3

# Project Overview



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

## 3.1 Project Deliverables

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

### 3.1.1 Core Deliverables (CD)

The project's core deliverables are described below.

**CD1 Design a compact 16-bit RISC instruction set architecture.**
The instruction set will be the primary interface to control the processor from software. An instruction set will be required to implement the custom multi-core communication interface.

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

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

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

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

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

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

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

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

### 3.1.2 Extended Deliverables (ED)

The project's extended deliverables are described below.

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

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

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

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

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

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

**ED7** Store instruction memory in SPI flash.

**ED8** Reprogram instruction memory at runtime from host computer.

**ED9** Processor external debugger using host-processor link.

## 3.2 Project Timeline

### 3.2.1 Project Stages

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

The project stages are shown in Table 3.1.

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

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

### 3.2.2 Project Stage Detail

**Stages 1.0 through 1.2 – Research and Project Conception**

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

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

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

**Stages 3.1 through 3.4 – Advanced Processor Implementation**

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

**Stages 4.1 through 4.4 – Multiprocessor Functionality**

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

**Stages 5.1 through 5.3 – Debugging Features**

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

**Stages 6.1 through 6.2 – Compiler Backends**

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

**Stage 7.1 – Wishbone Peripherals**

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

**Stage 8.1 – Final Report**

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

### 3.2.3 Timeline

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

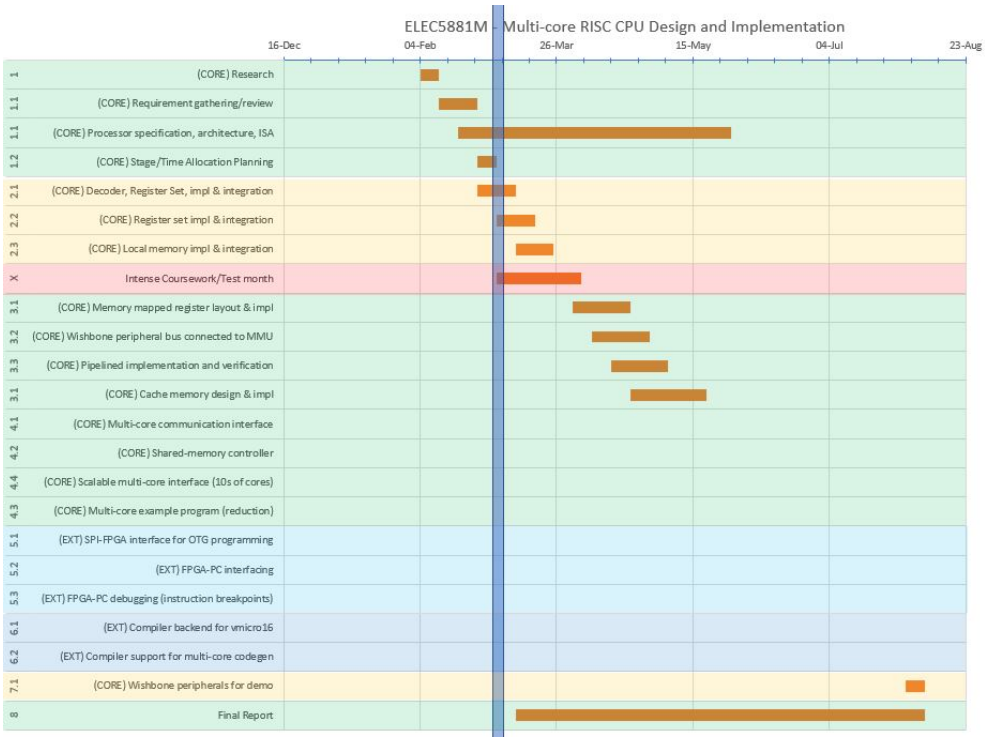


**Figure 3.1:** Project stages in a Gantt chart.

## 3.3 Resources

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

### 3.3.1 Hardware Resources

Core deliverable **CD5** requires the designed RISC core to be implemented and demonstrated on multiple FPGA devices. Although my design should synthesise for physical IC implementation, due to high costs and lengthy production times, it is not a primary development target. Due to having past experience with Xilinx FPGAs from my placement work and experience with Altera from university modules it was decided to target the Xilinx Spartan 6 XC6SLX9 and the Altera Cyclone V.

#### Terasic DE1-SoC Development Board

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

#### Minispartan 6+ FPGA Development Board

The Minispartan 6+ is a hobbyist FGPA development board with fewer peripherals than the DE1-SoC. The board features a Xilinx Spartan 6 XC6LX9 which has far fewer resources than the DE1-

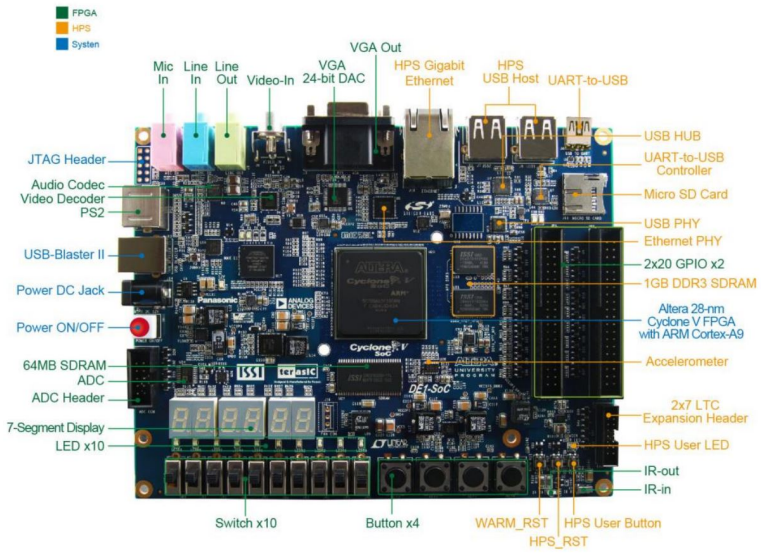


**Figure 3.2:** Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [2].

SoC's Cyclone V however it's simplicity and my familiarity with Xilinx's software suite will speed up development. The development board is shown in Figure 3.3.

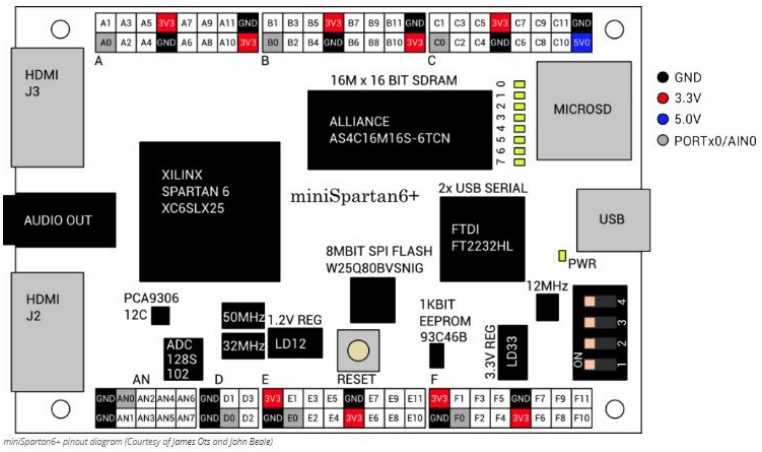


**Figure 3.3:** Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [3].

### 3.3.2 Software Resources

#### Intel Quartus

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

#### Xilinx ISE Webpack

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

**Verilator**

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

## 3.4 Legal and Ethical Considerations

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

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

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

# Chapter 4

# Current Progress



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

## 4.1 RISC Core

Following the project time line described in section 3.2, the first couple months have been dedicated to the design and implementation of the instruction set architecture and RISC core with stages 1-3. Good progress has been made in both deliverables, the ISA and the RISC core, and the progress is on-time with the initial project time line. The core has been nicknamed *Vmicro16* – short for Verilog microprocessor 16-bit.

### 4.1.1 Instruction Set Architecture

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

#### Design Goals

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### 4.1.2 Design and Implementation

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

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

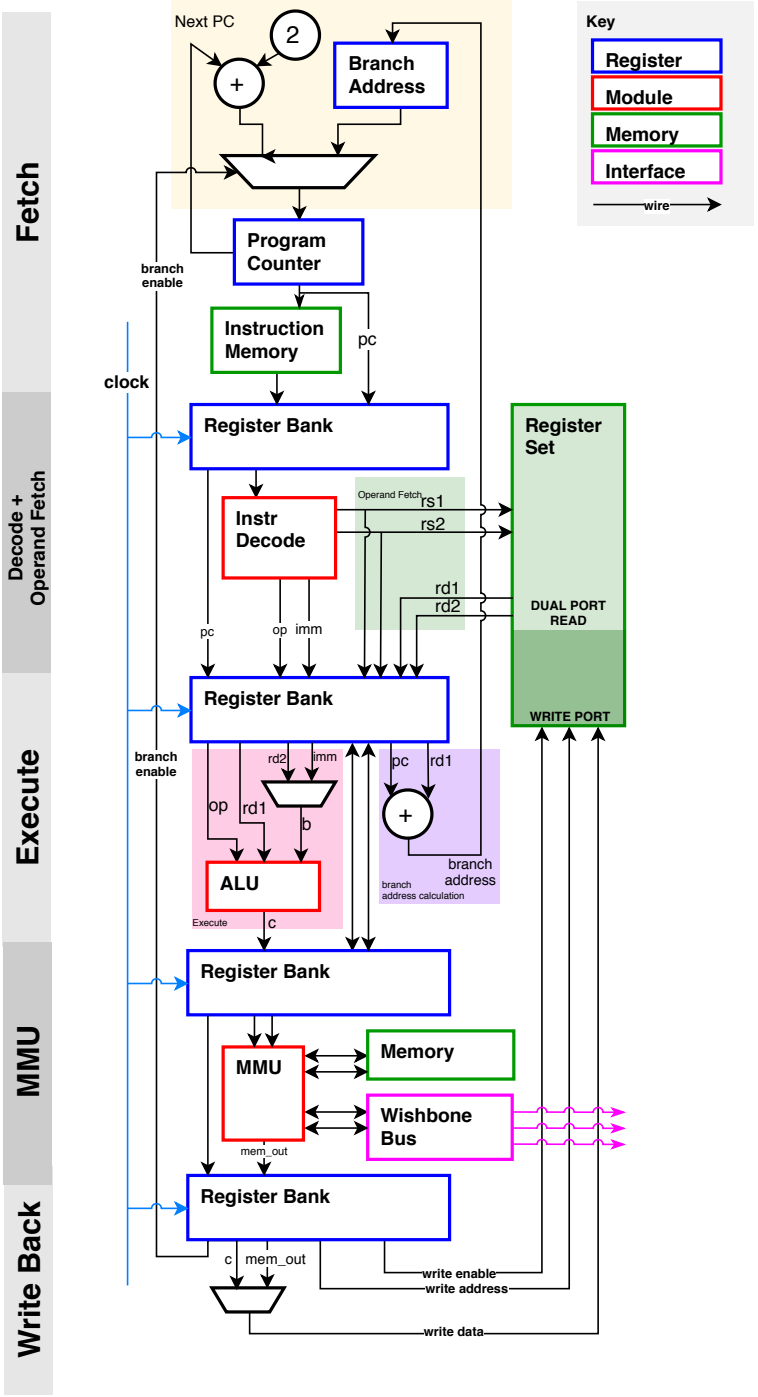


**Figure 4.2:** Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

#### Instruction and Data Memory

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

#### Register File

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

#### Pipelining

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

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

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

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

#### Memory Management Unit

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

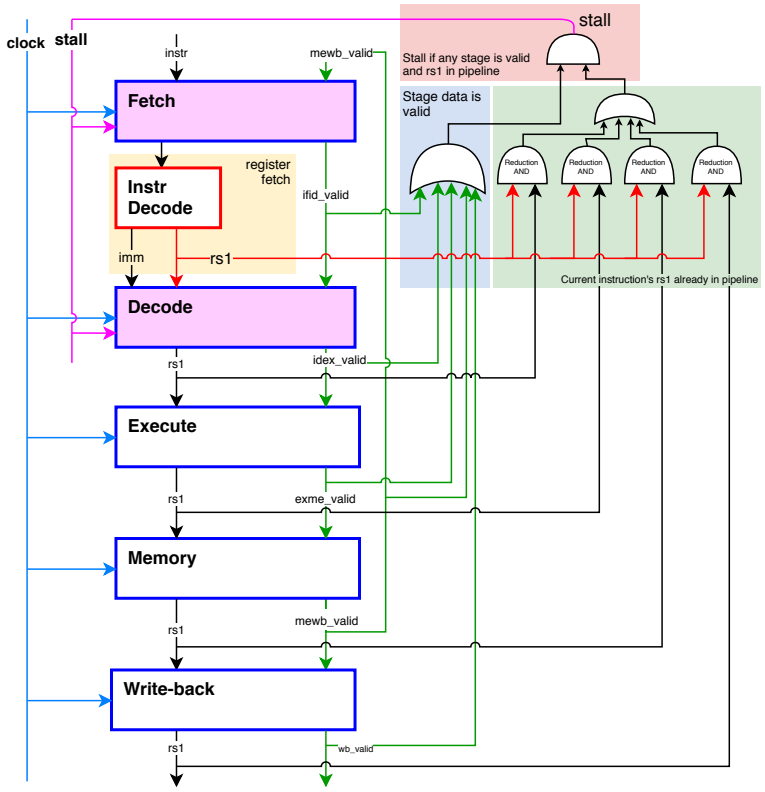


**Figure 4.3:** Pipeline data hazard detection. The register selects are passed forward through each stage and compared to the IDEX (latest instruction) register selects. If they match, the latest instruction depends on the output of an instruction in the pipeline, the IFID and IDEX stages are stalled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

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

### Proposed Memory Mapped Addresses

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

Later in the project, I plan to rewrite the addressing scheme to use a simpler address format which is closer to commonly used peripheral addressing schemes used today. The proposed memory mapped addresses for each system and peripheral are listed below.

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

**Table 4.1:** Provisional memory-mapped addresses table.

### ALU Design

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

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

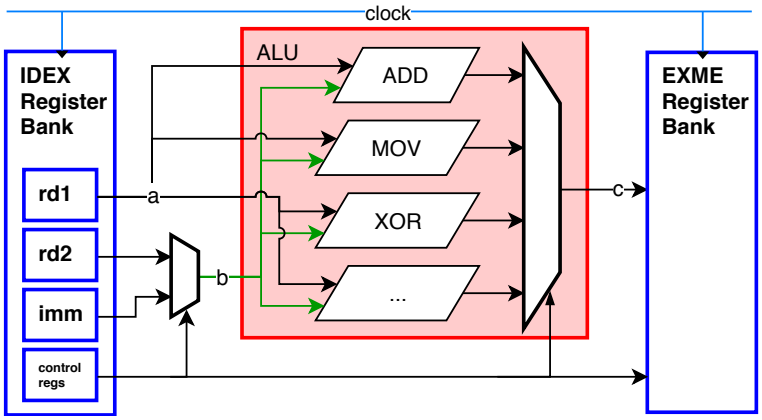


**Figure 4.4:** Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

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

```
322         //`define TEST_BRAM
323         `ifdef TEST_BRAM
324         // 2 core BRAM0 test
325         mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
326         mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
327         mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
328         mem[3] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
329         `endif
330     end
331
332     always @(posedge clk) begin
333         // synchronous WRITE_FIRST (page 13)
334         if (mem_we) begin
335             mem[mem_addr] <= mem_in;
```

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

### Decoder Design

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

```
224 mem[24] = 16'h3fa1;
225 mem[25] = 16'h28a0;
226 mem[26] = 16'h10df;
227 mem[27] = 16'h08dc;
228 mem[28] = 16'h0800;
229 mem[29] = 16'h08dd;
230 mem[30] = 16'h0800;
231 mem[31] = 16'h08de;
232 mem[32] = 16'h0800;
233 mem[33] = 16'h08de;
234 mem[34] = 16'h0800;
235 mem[35] = 16'h3fa1;
236 mem[36] = 16'h10e0;
237 mem[37] = 16'h2830;
238 mem[38] = 16'h0be0;
239 mem[39] = 16'h37a1;
240 mem[40] = 16'h307f;
241 mem[41] = 16'h3fa1;
242 mem[42] = 16'h10e0;
243 mem[43] = 16'h08df;
244 mem[44] = 16'h0be0;
245 mem[45] = 16'h37a1;
```

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

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

### 4.1.3 Verification

Currently, the only verification method used is manual inspection of the output waveforms of a test bench. For now, it is easier and faster to spot erroneous states by hand due to the large complexity of the pipeline. Later in the project, automatic test benches will be utilised.

#### Known Bugs

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

**BUG1 Stall detection does not consider load/store instructions.**

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

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

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

# Chapter 5

# Future Work



## 5.1 Project Status

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

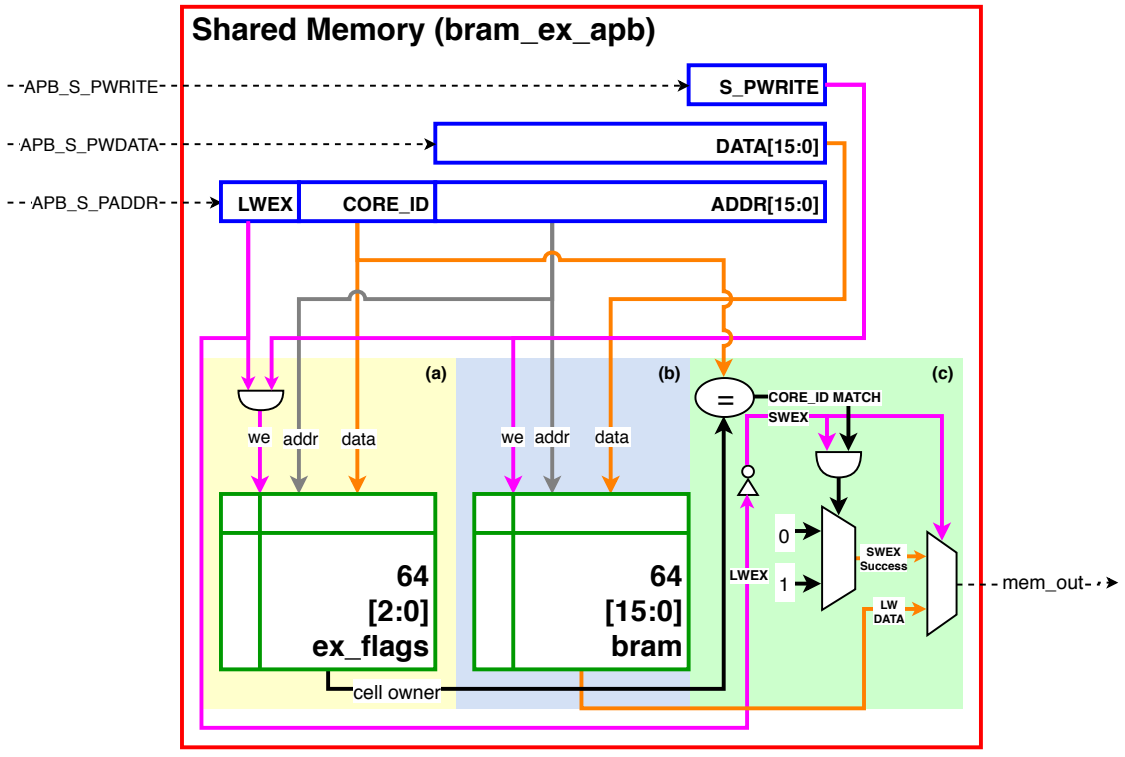


**Figure 5.1:** Caption for BRAMex

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

### 5.1.1 Updated Project Time Line

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

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

The updated project status is shown in Table 5.1 and in Figure 5.2.

### 5.1.2 Future Work

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

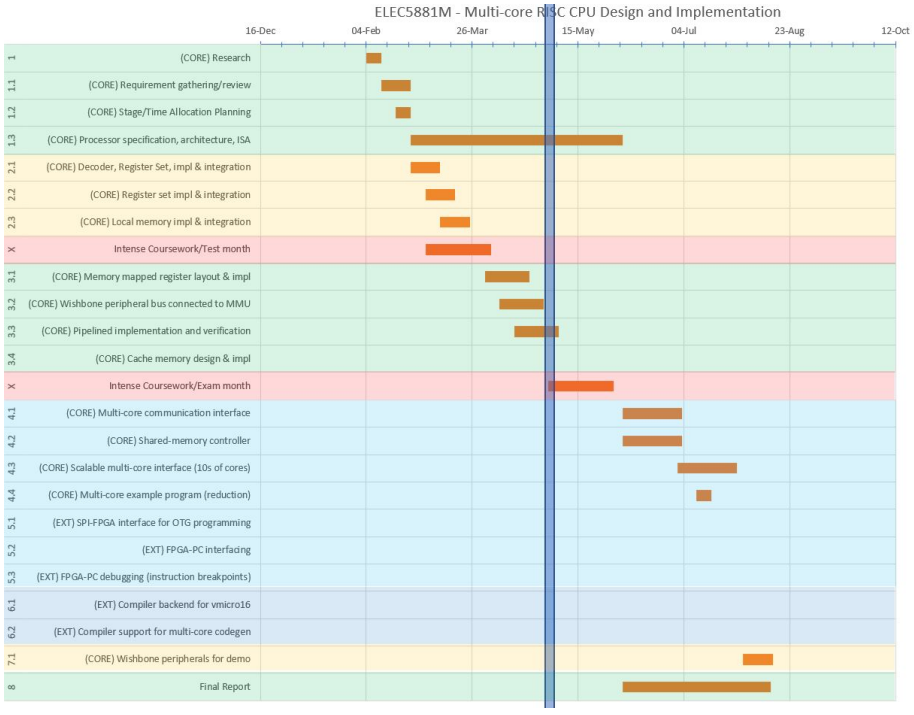


**Figure 5.2:** Updated project time gantt chart showing time allocations for stage 4.

| Stage | Title | Start Date | Core | Status |
|---|---|---|---|---|
| 1.0 | Research | Feb 04 | x | Completed |
| 1.1 | Requirement gathering/review | Feb 11 | x | Completed |
| 1.1 | Processor specification, architecture, ISA | Feb 18 | x | Completed |
| 1.2 | Stage/Time Allocation Planning | Feb 25 | x | Completed |
| 2.1 | Decoder, Register Set, impl & integration | Feb 25 | x | Completed |
| 2.2 | Register set impl & integration | Mar 04 | x | Completed |
| 2.3 | Local memory impl & integration | Mar 11 | x | Completed |
| 3.1 | Memory mapped register layout & impl | Apr 01 | | On-going |
| 3.2 | Wishbone peripheral bus connected to MMU | Apr 08 | | On-going |
| 3.3 | Pipeline implementation and verification | Apr 15 | | On-going |
| 3.4 | Cache memory design & impl | ~~Apr 22~~ | | Cancelled |
| 4.1 | Multi-core communication interface | Jun 05 | x | Planned |
| 4.2 | Shared-memory controller | Jun 05 | x | Planned |
| 4.3 | Scalable multi-core interface (10s of cores) | Jul 01 | x | Planned |
| 4.4 | Multi-core example program (reduction) | Jul 10 | x | Planned |
| 5.1 | SPI-FPGA interface for OTG programming | ~~TBD~~ | | Cancelled |
| 5.2 | FPGA-PC interfacing | ~~TBD~~ | | Cancelled |
| 5.3 | FPGA-PC debugging (instruction breakpoints) | ~~TBD~~ | | Cancelled |
| 6.1 | Compiler backend for vmicro16 | TBD | | Unknown |
| 6.2 | Compiler support for multi-core codegen | TBD | | Unknown |
| 7.1 | Wishbone peripherals for demo | Aug 01 | x | Planned |
| 8.1 | Final Report | Jun 05 | x | Planned |

**Table 5.1:** Updated project stages.

# Chapter 6

# Conclusion

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

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

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

# References

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

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

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

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

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

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

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

# Appendix A - Code Listing

## vmicro16.v

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

```
// 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"


(* keep_hierarchy = "yes" *)
(* dont_touch = "yes" *)
module vmicro16_bram_ex_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter CORE_ID_BITS = 3
) (
    input clk,
    input reset,

    //  19     18           16 15           0
    // | LWEX | 3 bit CORE_ID |     S_PADDR |
    input  [(1 + CORE_ID_BITS + (BUS_WIDTH-1):0] S_PADDR,

    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [BUS_WIDTH-1:0]          S_PWDATA,

    output [BUS_WIDTH-1:0]          S_PRDATA,
    output                          S_PREADY
);
    // exclusive flag checks
    wire [MEM_WIDTH-1:0] mem_out;
    wire [MEM_WIDTH-1:0] mem_out_ex;

    always (*)
        if (we && lwex)
            // SWEX
            // return 0 or 1

    assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out_ex : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1       : 1'b0;
    assign we       = (S_PSELx & S_PENABLE & S_PWRITE);

    // Similar to:
    //   http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204f/Cihbghef.html

    // mem_wd is the CORE_ID sent in bits [18:16]
    localparam TOP_BIT_INDEX         = (1 + CORE_ID_BITS + (BUS_WIDTH-1);
    localparam PADDR_CORE_ID_MSB     = TOP_BIT_INDEX - 1;
    localparam PADDR_CORE_ID_LSB     = PADDR_CORE_ID_MSB - (CORE_ID_BITS-1);

    // [LWEX, CORE_ID, mem_addr] from S_PADDR
    wire                    lwex       = S_PADDR[TOP_BIT_INDEX];
    wire [CORE_ID_BITS-1:0] addr_is_ex;
    wire [CORE_ID_BITS-1:0] mem_wd     = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
    wire [BUS_WIDTH-1:0]    mem_addr   = S_PADDR[BUS_WIDTH-1:0];

    // Exclusive flag for each memory cell
    (* keep_hierarchy = "yes" *)
    vmicro16_regs # (
        // Each cell is for storing the CORE_ID of the core
        // that has exclusive access
        .CELL_WIDTH (CORE_ID_BITS),
        // Same number of cells as the memory
        .CELL_DEPTH (MEM_DEPTH),
        // register exclusive
        .DEBUG_NAME ("REX")
    ) ex_flags (
        .clk        (clk),
        .reset      (reset),
        // async port 0
        .rs1        (mem_addr),
        .rd1        (addr_is_ex),
        // async port 1
```

```
        //.rs2        (),
        //.rd2        (),
        // write port
        .we         (lwex && we),
        .ws1        (mem_addr),
        .wd         (r_reg_wd)
    );

    // Check exclusive access flags
    always @(posedge clk)
        if (we && ex_en)
            // SWEX
            //

    always @(*)
        if (S_PSELx && S_PENABLE)
            $display($time, "\t\tMEM => %h", mem_out);

    always @(posedge clk)
        if (we)
            $display($time, "\t\tBRAM[%h] <= %h", S_PADDR, S_PWDATA);

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .NAME       ("BRAM")
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

        .mem_addr   (S_PADDR),
        .mem_in     (S_PWDATA),
        .mem_we     (we),
        .mem_out    (mem_out)
    );
endmodule


(* keep_hierarchy = "yes" *)
(* dont_touch = "yes" *)
module vmicro16_bram_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter APB_PADDR    = 0
) (
    input clk,
    input reset,
    // APB Slave to master interface
    input  [`clog2(MEM_DEPTH)-1:0]  S_PADDR,
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [BUS_WIDTH-1:0]          S_PWDATA,

    output [BUS_WIDTH-1:0]          S_PRDATA,
    output                          S_PREADY
);
    wire [MEM_WIDTH-1:0] mem_out;

    assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1    : 1'b0;
    assign we       = (S_PSELx & S_PENABLE & S_PWRITE);

    always @(*)
        if (S_PSELx && S_PENABLE)
            $display($time, "\t\tMEM => %h", mem_out);

    always @(posedge clk)
        if (we)
            $display($time, "\t\tBRAM[%h] <= %h", S_PADDR, S_PWDATA);

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .NAME       ("BRAM")
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

        .mem_addr   (S_PADDR),
        .mem_in     (S_PWDATA),
        .mem_we     (we),
        .mem_out    (mem_out)
    );
endmodule


// This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
//   https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
//   https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
//   https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf
(* keep_hierarchy = "yes" *)
module vmicro16_bram # (
    parameter MEM_WIDTH     = 16,
    parameter MEM_DEPTH     = 64,
    parameter CORE_ID       = 0,
    parameter NAME          = "BRAM"
) (
    input clk,
    input reset,

    input      [`clog2(MEM_DEPTH)-1:0] mem_addr,
    input      [MEM_WIDTH-1:0]         mem_in,
    input                              mem_we,
    output reg [MEM_WIDTH-1:0]         mem_out
);
    // memory vector
    reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];

    // not synthesizable
    integer i;
    initial begin
        for (i = 0; i < MEM_DEPTH; i = i + 1) mem[i] = 0;
        //$readmemh("../../test.hex", mem);

```

```
198             `define TEST_COMPILER
199             `ifdef TEST_COMPILER
200     mem[0] = 16'h2f3f;
201     mem[1] = 16'h2903;
202     mem[2] = 16'h4100;
203     mem[3] = 16'h3fa1;
204     mem[4] = 16'h16e0;
205     mem[5] = 16'h26e0;
206     mem[6] = 16'h3fa1;
207     mem[7] = 16'h2890;
208     mem[8] = 16'h10d9;
209     mem[9] = 16'h3fa1;
210     mem[10] = 16'h2891;
211     mem[11] = 16'h10da;
212     mem[12] = 16'h3fa1;
213     mem[13] = 16'h2892;
214     mem[14] = 16'h10db;
215     mem[15] = 16'h3fa1;
216     mem[16] = 16'h2880;
217     mem[17] = 16'h10dc;
218     mem[18] = 16'h3fa1;
219     mem[19] = 16'h28b0;
220     mem[20] = 16'h10dd;
221     mem[21] = 16'h3fa1;
222     mem[22] = 16'h28b1;
223     mem[23] = 16'h10de;
224     mem[24] = 16'h3fa1;
225     mem[25] = 16'h28a0;
226     mem[26] = 16'h10df;
227     mem[27] = 16'h08dc;
228     mem[28] = 16'h0800;
229     mem[29] = 16'h08dd;
230     mem[30] = 16'h0800;
231     mem[31] = 16'h08de;
232     mem[32] = 16'h0800;
233     mem[33] = 16'h08de;
234     mem[34] = 16'h0800;
235     mem[35] = 16'h3fa1;
236     mem[36] = 16'h10e0;
237     mem[37] = 16'h2830;
238     mem[38] = 16'h0be0;
239     mem[39] = 16'h37a1;
240     mem[40] = 16'h307f;
241     mem[41] = 16'h3fa1;
242     mem[42] = 16'h10e0;
243     mem[43] = 16'h08df;
244     mem[44] = 16'h0be0;
245     mem[45] = 16'h37a1;
246     mem[46] = 16'h1300;
247     mem[47] = 16'h27c0;
248     mem[48] = 16'h0ee0;
249     mem[49] = 16'h37a1;
250     mem[50] = 16'h6000;
251             `endif
252
253             //`define TEST_MULTICORE
254             `ifdef TEST_MULTICORE
255             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
256             mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
257             mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
258             mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h80};
259             mem[4] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
260             mem[5] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
261             mem[6] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
262             mem[7] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
263             mem[8] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
264             mem[9] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h0};
265             `endif
266
267             //`define TEST_BR
268             `ifdef TEST_BR
269             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0};
270             mem[1] = {`VMICRO16_OP_MOVI,    3'h3, 8'h3};
271             mem[2] = {`VMICRO16_OP_MOVI,    3'h1, 8'h2};
272             mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
273             mem[4] = {`VMICRO16_OP_BR,      3'h3, `VMICRO16_OP_BR_U};
274             mem[5] = {`VMICRO16_OP_MOVI,    3'h0, 8'hFF};
275             `endif
276
277             //`define ALL_TEST
278             `ifdef ALL_TEST
279             // Standard all test
280             // REGS0
281             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h81};
282             mem[1] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
283             mem[2] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
284             // GPIO0
285             mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
286             mem[4] = {`VMICRO16_OP_MOVI,    3'h1, 8'hD};
287             mem[5] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
288             mem[6] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
289             // TIM0
290             mem[7] = {`VMICRO16_OP_MOVI,    3'h0, 8'h07};
291             mem[8] = {`VMICRO16_OP_LW,      3'h3, 3'h0, 5'h03};
292             // UART0
293             mem[9]  = {`VMICRO16_OP_MOVI,    3'h0, 8'hA0};      // UART0
294             mem[10] = {`VMICRO16_OP_MOVI,    3'h1, 8'h41};      // ascii A
295             mem[11] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
296             mem[12] = {`VMICRO16_OP_MOVI,    3'h1, 8'h42}; // ascii B
297             mem[13] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
298             mem[14] = {`VMICRO16_OP_MOVI,    3'h1, 8'h43}; // ascii C
299             mem[15] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
300             mem[16] = {`VMICRO16_OP_MOVI,    3'h1, 8'h44}; // ascii D
301             mem[17] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
302             mem[18] = {`VMICRO16_OP_MOVI,    3'h1, 8'h45}; // ascii D
303             mem[19] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
304             mem[20] = {`VMICRO16_OP_MOVI,    3'h1, 8'h46}; // ascii E
305             mem[21] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
306             // BRAM0
307             mem[22] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
308             mem[23] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
309             mem[24] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
310             mem[25] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
311             // GPIO1 (SSD 24-bit port)
312             mem[26] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
313             mem[27] = {`VMICRO16_OP_MOVI,    3'h1, 8'h12};
```

```
            mem[28] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[29] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
            // GPIO2
            mem[30] = {`VMICRO16_OP_MOVI,    3'h0, 8'h92};
            mem[31] = {`VMICRO16_OP_MOVI,    3'h1, 8'h56};
            mem[32] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            `endif

            //`define TEST_BRAM
            `ifdef TEST_BRAM
            // 2 core BRAM0 test
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
            mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
            mem[3] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
            `endif
        end

        always @(posedge clk) begin
            // synchronous WRITE_FIRST (page 13)
            if (mem_we) begin
                mem[mem_addr] <= mem_in;
                $display($time, "\t\t%s[%h] <= %h",
                        NAME, mem_addr, mem_in);
            end else
                mem_out <= mem[mem_addr];
        end

        // TODO: Reset impl = every clock while reset is asserted, clear each cell
        //       one at a time, mem[i++] <= 0
    endmodule

    (* keep_hierarchy = "yes" *)
    module vmicro16_core_mmu # (
        parameter MEM_WIDTH     = 16,
        parameter MEM_DEPTH     = 64,

        parameter CORE_ID       = 0
    ) (
        input clk,
        input reset,

        input  req,
        output busy,

        // From core
        input      [MEM_WIDTH-1:0]  mmu_addr,
        input      [MEM_WIDTH-1:0]  mmu_in,
        input                       mmu_we,
        output reg [MEM_WIDTH-1:0]  mmu_out,

        // TO APB interconnect
        output reg [MEM_WIDTH-1:0]   M_PADDR,
        output reg                   M_PWRITE,
        output reg                   M_PSELx,
        output reg                   M_PENABLE,
        output reg [MEM_WIDTH-1:0]   M_PWDATA,
        // from interconnect
        input      [MEM_WIDTH-1:0]   M_PRDATA,
        input                        M_PREADY
    );
        localparam TIM_BITS_ADDR = `clog2(MEM_DEPTH);
        localparam MMU_STATE_T1  = 0;
        localparam MMU_STATE_T2  = 1;
        localparam MMU_STATE_T3  = 2;
        reg [1:0] mmu_state      = MMU_STATE_T1;

        reg  [MEM_WIDTH-1:0] per_out = 0;
        wire [MEM_WIDTH-1:0] tim0_out;

        assign busy = req || (mmu_state == MMU_STATE_T2);

        // tightly integrated memory usage
        wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
                    && (mmu_addr <= `DEF_MMU_TIM0_E);
        wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
                    && (mmu_addr <= `DEF_MMU_SREG_E);


        wire [TIM_BITS_ADDR-1:0] tim0_addr = (mmu_addr - `DEF_MMU_TIM0_S);
        wire                     tim0_we   = (tim0_en && mmu_we);
        wire                     apb_en    = (!tim0_en) && (!sreg_en);

        localparam SPECIAL_REGS = 8;
        wire [`clog2(SPECIAL_REGS)-1:0] sr_sel = (mmu_addr - `DEF_MMU_SREG_S);
        wire [MEM_WIDTH-1:0]            sr_val;

        // Output port
        always @(*)
            if      (tim0_en) mmu_out = tim0_out;
            else if (sreg_en) mmu_out = sr_val;
            else              mmu_out = per_out;

        // APB master to slave interface
        always @(posedge clk)
            if (reset) begin
                mmu_state <= MMU_STATE_T1;
                M_PENABLE <= 0;
                M_PADDR   <= 0;
                M_PWDATA  <= 0;
                M_PSELx   <= 0;
                M_PWRITE  <= 0;
            end
            else
                casex (mmu_state)
                    MMU_STATE_T1: begin
                        if (req && apb_en) begin
                            M_PADDR   <= mmu_addr;
                            M_PWDATA  <= mmu_in;
                            M_PSELx   <= 1;
                            M_PWRITE  <= mmu_we;

                            mmu_state <= MMU_STATE_T2;
                        end
                    end
```

```
                `ifdef FIX_T3
                    MMU_STATE_T2: begin
                        M_PENABLE <= 1;

                        if (M_PREADY == 1'b1) begin
                            mmu_state <= MMU_STATE_T3;
                        end
                    end

                    MMU_STATE_T3: begin
                        // Slave has output a ready signal (finished)
                        M_PENABLE <= 0;
                        M_PADDR   <= 0;
                        M_PWDATA  <= 0;
                        M_PSELx   <= 0;
                        M_PWRITE  <= 0;
                        // Clock the peripheral output into a reg,
                        //   to output on the next clock cycle
                        per_out   <= M_PRDATA;

                        mmu_state <= MMU_STATE_T1;
                    end
                `else
                    // No FIX_T3
                    MMU_STATE_T2: begin
                        if (M_PREADY == 1'b1) begin
                            M_PENABLE <= 0;
                            M_PADDR   <= 0;
                            M_PWDATA  <= 0;
                            M_PSELx   <= 0;
                            M_PWRITE  <= 0;
                            // Clock the peripheral output into a reg,
                            //   to output on the next clock cycle
                            per_out   <= M_PRDATA;

                            mmu_state <= MMU_STATE_T1;
                        end else begin
                            M_PENABLE <= 1;
                        end
                    end
                `endif
            endcase

    vmicro16_regs # (
        .CELL_DEPTH         (SPECIAL_REGS),
        .CELL_WIDTH         (MEM_WIDTH),
        // per core special values
        .PARAM_DEFAULTS_R0  (CORE_ID),
        .PARAM_DEFAULTS_R1  ({16{1'b0}})
    ) regs_apb (
        .clk    (clk),
        .reset  (reset),
        .rs1    (sr_sel),
        .rd1    (sr_val),
        //.rs2    (),
        //.rd2    (),
        .we     (),
        .ws1    (),
        .wd     ()
    );

    // Each M core has a TIM0 scratch memory
    (* keep_hierarchy = "yes" *)
    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .NAME       ("TIM0")
    ) TIM0 (
        .clk        (clk),
        .reset      (reset),
        .mem_addr   (tim0_addr),
        .mem_in     (mmu_in),
        .mem_we     (tim0_we),
        .mem_out    (tim0_out)
    );
endmodule


(* keep_hierarchy = "yes" *)
module vmicro16_regs # (
    parameter CELL_WIDTH        = 16,
    parameter CELL_DEPTH        = 8,
    parameter CELL_SEL_BITS     = `clog2(CELL_DEPTH),
    parameter CELL_DEFAULTS     = 0,
    parameter DEBUG_NAME        = "",
    parameter CORE_ID           = 0,
    parameter PARAM_DEFAULTS_R0 = 16'h0000,
    parameter PARAM_DEFAULTS_R1 = 16'h0000
) (
    input clk,
    input reset,
    // Dual port register reads
    input      [CELL_SEL_BITS-1:0]  rs1, // port 1
    output     [CELL_WIDTH-1   :0]  rd1,
    //input      [CELL_SEL_BITS-1:0]  rs2, // port 2
    //output     [CELL_WIDTH-1   :0]  rd2,
    // EX/WB final stage write back
    input                           we,
    input [CELL_SEL_BITS-1:0]       ws1,
    input [CELL_WIDTH-1:0]          wd
);
    reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;

    // Initialise registers with default values
    //   Really only used for special registers used by the soc
    // TODO: How to do this on reset?
    integer i;
    initial
        if (CELL_DEFAULTS)
            $readmemh(CELL_DEFAULTS, regs);
        else begin
            for(i = 0; i < CELL_DEPTH; i = i + 1)
                regs[i] = 0;
            regs[0] = PARAM_DEFAULTS_R0;
            regs[1] = PARAM_DEFAULTS_R1;
            end
```

```
546
547         always @(regs)
548             $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
549                 CORE_ID,
550                 regs[0], regs[1], regs[2], regs[3],
551                 regs[4], regs[5], regs[6], regs[7]);
552
553         always @(posedge clk)
554             if (reset) begin
555                 for(i = 0; i < CELL_DEPTH; i = i + 1)
556                     regs[i] <= 0;
557                 regs[0] <= PARAM_DEFAULTS_R0;
558                 regs[1] <= PARAM_DEFAULTS_R1;
559             end
560             else if (we) begin
561                 $display($time, "\tC%02h: REGS #%s: Writing %h to reg[%d]",
562                     CORE_ID, DEBUG_NAME, wd, ws1);
563
564                 // Perform the write
565                 regs[ws1] <= wd;
566             end
567
568         assign rd1 = regs[rs1];
569         //assign rd2 = regs[rs2];
570     endmodule
571
572     (* keep_hierarchy = "yes" *)
573     (* dont_touch = "yes" *)
574     module vmicro16_regs_apb # (
575         parameter BUS_WIDTH         = 16,
576         parameter CELL_DEPTH        = 8,
577         parameter PARAM_DEFAULTS_R0 = 0,
578         parameter PARAM_DEFAULTS_R1 = 0
579     ) (
580         input clk,
581         input reset,
582         // APB Slave to master interface
583         input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
584         input                           S_PWRITE,
585         input                           S_PSELx,
586         input                           S_PENABLE,
587         input  [BUS_WIDTH-1:0]          S_PWDATA,
588
589         output [BUS_WIDTH-1:0]          S_PRDATA,
590         output                          S_PREADY
591     );
592         wire [BUS_WIDTH-1:0] rd1;
593
594         assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1  : 16'h0000;
595         assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
596         assign reg_we   = (S_PSELx & S_PENABLE & S_PWRITE);
597
598         always @(*)
599             if (reg_we)
600                 $display($time, "\t\tREGS_APB[%h] <= %h", S_PADDR, S_PWDATA);
601
602         always @(*)
603             `rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))
604
605         vmicro16_regs # (
606             .CELL_DEPTH         (CELL_DEPTH),
607             .PARAM_DEFAULTS_R0  (PARAM_DEFAULTS_R0),
608             .PARAM_DEFAULTS_R1  (PARAM_DEFAULTS_R1)
609         ) regs_apb (
610             .clk    (clk),
611             .reset  (reset),
612
613             .rs1    (S_PADDR),
614             .rd1    (rd1),
615
616             //.rs2    (),
617             //.rd2    (),
618
619             .we     (reg_we),
620             .ws1    (S_PADDR),
621             .wd     (S_PWDATA) // either alu_c or mem_out
622         );
623     endmodule
624
625
626     (*dont_touch="true"*)
627     (* keep_hierarchy = "yes" *)
628     module vmicro16_gpio_apb # (
629         parameter BUS_WIDTH  = 16,
630         parameter PORTS      = 8
631     ) (
632         input clk,
633         input reset,
634         // APB Slave to master interface
635         input  [0:0]                    S_PADDR, // not used (optimised out)
636         input                           S_PWRITE,
637         input                           S_PSELx,
638         input                           S_PENABLE,
639         input  [BUS_WIDTH-1:0]          S_PWDATA,
640
641         output [BUS_WIDTH-1:0]          S_PRDATA,
642         output                          S_PREADY,
643         output reg [PORTS-1:0]          gpio
644     );
645         assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
646         assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
647         assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);
648
649         always @(posedge clk)
650             if (reset)
651                 gpio <= 0;
652             else if (ports_we) begin
653                 $display($time, "\t\tGPIO <= %h", S_PWDATA[PORTS-1:0]);
654                 gpio <= S_PWDATA[PORTS-1:0];
655             end
656     endmodule
657
658     // Decoder is hard to parameterise as it's very closely linked to the ISA.
659     (* keep_hierarchy = "yes" *)
660     module vmicro16_dec # (
661         parameter INSTR_WIDTH    = 16,
```

```
662         parameter INSTR_OP_WIDTH = 5,
663         parameter INSTR_RS_WIDTH = 3,
664         parameter ALU_OP_WIDTH   = 5
665     ) (
666         //input clk,   // not used yet (all combinational)
667         //input reset, // not used yet (all combinational)
668
669         input  [INSTR_WIDTH-1:0]    instr,
670
671         output [INSTR_OP_WIDTH-1:0] opcode,
672         output [INSTR_RS_WIDTH-1:0] rd,
673         output [INSTR_RS_WIDTH-1:0] ra,
674         output [3:0]                imm4,
675         output [7:0]                imm8,
676         output [11:0]               imm12,
677         output [4:0]                simm5,
678
679         // This can be freely increased without affecting the isa
680         output reg [ALU_OP_WIDTH-1:0] alu_op,
681
682         output reg has_imm4,
683         output reg has_imm8,
684         output reg has_imm12,
685         output reg has_we,
686         output reg has_br,
687         output reg has_mem,
688         output reg has_mem_we,
689         output reg has_cmp,
690
691         output halt,
692
693         output reg has_ex
694
695         // TODO: Use to identify bad instruction and
696         //       raise exceptions
697         //,output     is_bad
698     );
699         assign opcode = instr[15:11];
700         assign rd     = instr[10:8];
701         assign ra     = instr[7:5];
702         assign imm4   = instr[3:0];
703         assign imm8   = instr[7:0];
704         assign imm12  = instr[11:0];
705         assign simm5  = instr[4:0];
706         // Special opcodes
707         assign halt   = (opcode == `VMICRO16_OP_HALT);
708
709         // exme_op
710         always @(*) case (opcode)
711             `VMICRO16_OP_HALT,    // TODO: stop ifid
712             `VMICRO16_OP_NOP:             alu_op = `VMICRO16_ALU_NOP;
713
714             `VMICRO16_OP_LW:              alu_op = `VMICRO16_ALU_LW;
715             `VMICRO16_OP_SW:              alu_op = `VMICRO16_ALU_SW;
716             `VMICRO16_OP_LWEX:            alu_op = `VMICRO16_ALU_LW;
717             `VMICRO16_OP_SWEX:            alu_op = `VMICRO16_ALU_SW;
718
719             `VMICRO16_OP_MOV:             alu_op = `VMICRO16_ALU_MOV;
720             `VMICRO16_OP_MOVI:            alu_op = `VMICRO16_ALU_MOVI;
721
722             `VMICRO16_OP_BR:              alu_op = `VMICRO16_ALU_BR;
723             `VMICRO16_OP_MULT:            alu_op = `VMICRO16_ALU_MULT;
724
725             `VMICRO16_OP_BIT:     casez (simm5)
726                 `VMICRO16_OP_BIT_OR:      alu_op = `VMICRO16_ALU_BIT_OR;
727                 `VMICRO16_OP_BIT_XOR:     alu_op = `VMICRO16_ALU_BIT_XOR;
728                 `VMICRO16_OP_BIT_AND:     alu_op = `VMICRO16_ALU_BIT_AND;
729                 `VMICRO16_OP_BIT_NOT:     alu_op = `VMICRO16_ALU_BIT_NOT;
730                 `VMICRO16_OP_BIT_LSHFT:   alu_op = `VMICRO16_ALU_BIT_LSHFT;
731                 `VMICRO16_OP_BIT_RSHFT:   alu_op = `VMICRO16_ALU_BIT_RSHFT;
732                 default:                  alu_op = `VMICRO16_ALU_BAD; endcase
733
734             `VMICRO16_OP_ARITH_U:     casez (simm5)
735                 `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
736                 `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
737                 `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
738                 default:                  alu_op = `VMICRO16_ALU_BAD; endcase
739
740             `VMICRO16_OP_ARITH_S:     casez (simm5)
741                 `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
742                 `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
743                 `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
744                 default:                  alu_op = `VMICRO16_ALU_BAD; endcase
745
746             default: begin
747                                       alu_op = `VMICRO16_ALU_NOP;
748                 $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
749             end
750         endcase
751
752         // Register writes
753         always @(*) case (opcode)
754             `VMICRO16_OP_LW,
755             `VMICRO16_OP_MOV,
756             `VMICRO16_OP_MOVI,
757             //`VMICRO16_OP_MOVI_L,
758             `VMICRO16_OP_ARITH_U,
759             `VMICRO16_OP_ARITH_S,
760             `VMICRO16_OP_SETC,
761             `VMICRO16_OP_MULT:      has_we = 1'b1;
762             default:                has_we = 1'b0;
763         endcase
764
765         // Contains 4-bit immediate
766         always @(*)
767             if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
768                 ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
769                 has_imm4 = 1'b1;
770             else
771                 has_imm4 = 1'b0;
772
773         // Contains 8-bit immediate
774         always @(*) case (opcode)
775             `VMICRO16_OP_MOVI,
776             `VMICRO16_OP_BR:        has_imm8 = 1'b1;
777             default:                has_imm8 = 1'b0;
```

```
778         endcase
779
780         //// Contains 12-bit immediate
781         //always @(*) case (opcode)
782         //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
783         //    default:                has_imm12 = 1'b0;
784         //endcase
785
786         // Will branch the pc
787         always @(*) case (opcode)
788             `VMICRO16_OP_BR:    has_br = 1'b1;
789             default:            has_br = 1'b0;
790         endcase
791
792         // Requires external memory
793         always @(*) case (opcode)
794             `VMICRO16_OP_LW,
795             `VMICRO16_OP_SW,
796             `VMICRO16_OP_LWEX,
797             `VMICRO16_OP_SWEX:  has_mem = 1'b1;
798             default:            has_mem = 1'b0;
799         endcase
800
801         // Requires external memory write
802         always @(*) case (opcode)
803             `VMICRO16_OP_SW,
804             `VMICRO16_OP_SWEX:  has_mem_we = 1'b1;
805             default:            has_mem_we = 1'b0;
806         endcase
807
808         // Affects status registers (cmp instructions)
809         always @(*) case (opcode)
810             `VMICRO16_OP_CMP:   has_cmp = 1'b1;
811             default:            has_cmp = 1'b0;
812         endcase
813
814         // Performs exclusive checks
815         always @(*) case (opcode)
816             `VMICRO16_OP_LWEX,
817             `VMICRO16_OP_SWEX:   has_ex = 1'b1;
818             default:             has_ex = 1'b0;
819         endcase
820     endmodule
821
822     (* keep_hierarchy = "yes" *)
823     module vmicro16_alu # (
824         parameter OP_WIDTH   = 5,
825         parameter DATA_WIDTH = 16
826     ) (
827         // input clk, // TODO: make clocked
828
829         input      [OP_WIDTH-1:0]   op,
830         input      [DATA_WIDTH-1:0] a, // rs1/dst
831         input      [DATA_WIDTH-1:0] b, // rs2
832         output reg [DATA_WIDTH-1:0] c
833     );
834         reg [4:0] cmp_tmp = 0;
835
836         always @(*) case (op)
837             // branch/nop, output nothing
838             `VMICRO16_ALU_BR,
839             `VMICRO16_ALU_NOP:          c = 0;
840             // load/store addresses (use value in rd2)
841             `VMICRO16_ALU_LW,
842             `VMICRO16_ALU_SW:           c = b;
843             // bitwise operations
844             `VMICRO16_ALU_BIT_OR:       c = a | b;
845             `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
846             `VMICRO16_ALU_BIT_AND:      c = a & b;
847             `VMICRO16_ALU_BIT_NOT:      c = ~(b);
848             `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
849             `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;
850
851             `VMICRO16_ALU_MOV:          c = b;
852             `VMICRO16_ALU_MOVI:         c = b;
853             `VMICRO16_ALU_MOVI_L:       c = b;
854
855             `VMICRO16_ALU_ARITH_UADD:   c = a + b;
856             `VMICRO16_ALU_ARITH_USUB:   c = a - b;
857             // TODO: ALU should have simm5 as input
858             `VMICRO16_ALU_ARITH_UADDI:  c = a + b;
859
860             `ifdef DEF_ALU_HW_MULT
861                 `VMICRO16_ALU_MULT:  c = a * b;
862             `endif
863
864             `VMICRO16_ALU_ARITH_SADD:   c = $signed(a) + $signed(b);
865             `VMICRO16_ALU_ARITH_SSUB:   c = $signed(a) - $signed(b);
866             // TODO: ALU should have simm5 as input
867             `VMICRO16_ALU_ARITH_SSUBI:  c = $signed(a) - $signed(b);
868
869             `VMICRO16_ALU_CMP: begin
870                 // TODO: Do a-b in 17-bit register
871                 //       Set zero, overflow, carry, signed bits in result
872                 cmp_tmp = a - b;
873                 c = 0;
874                 c[`VMICRO16_SFLAG_U] = 1;
875                 c[`VMICRO16_SFLAG_Z] = (cmp_tmp == 0);
876                 c[`VMICRO16_SFLAG_L] = (a < b);
877             end
878
879             // TODO: Parameterise
880             default: begin
881                 $display($time, "\tALU: unknown op: %h", op);
882                 c = 16'h0000;
883             end
884         endcase
885     endmodule
886
887     (*dont_touch="true"*)
888     (* keep_hierarchy = "yes" *)
889     module vmicro16_core # (
890         parameter MEM_INSTR_DEPTH   = 64,
891         parameter MEM_SCRATCH_DEPTH = 64,
892         parameter MEM_WIDTH         = 16,
893
```

```
    parameter CORE_ID           = 0
) (
    input        clk,
    input        reset,

    output [7:0] dbug_pc,

    // APB master to slave interface (apb_intercon)
    output  [MEM_WIDTH-1:0]     w_PADDR,
    output                      w_PWRITE,
    output                      w_PSELx,
    output                      w_PENABLE,
    output  [MEM_WIDTH-1:0]     w_PWDATA,
    input   [MEM_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;
    reg  [2:0] r_state = STATE_IF;

    reg  [15:0] r_pc          = 16'h0000;
    reg  [15:0] r_instr       = 16'h0000;
    wire [15:0] w_mem_instr_out;

    assign dbug_pc = r_pc[7:0];

    wire [4:0]  r_instr_opcode;
    wire [4:0]  r_instr_alu_op;
    wire [2:0]  r_instr_rsd;
    wire [2:0]  r_instr_rsa;
    reg  [15:0] r_instr_rdd = 0;
    reg  [15:0] r_instr_rda = 0;
    wire [3:0]  r_instr_imm4;
    wire [7:0]  r_instr_imm8;
    wire [4:0]  r_instr_simm5;
    wire        r_instr_has_imm4;
    wire        r_instr_has_imm8;
    wire        r_instr_has_we;
    wire        r_instr_has_br;
    wire        r_instr_has_cmp;
    wire        r_instr_has_mem;
    wire        r_instr_has_mem_we;
    wire        r_instr_halt;

    wire [15:0] r_alu_out;

    wire [15:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
    wire [15:0] r_mem_scratch_in   = r_instr_rdd;
    wire [15:0] r_mem_scratch_out;
    wire        r_mem_scratch_we   = r_instr_has_mem_we && (r_state == STATE_ME);
    reg         r_mem_scratch_req  = 0;
    wire        r_mem_scratch_busy;

    reg  [2:0]  r_reg_rs1 = 0;
    wire [15:0] r_reg_rd1;
    //wire [15:0] r_reg_rd2;
    wire [15:0] r_reg_wd = (r_instr_has_mem) ? r_mem_scratch_out : r_alu_out;
    wire        r_reg_we = r_instr_has_we && (r_state == STATE_WB);

    // branching
    reg         r_branch_en   = 0;
    wire        w_branching   = r_instr_has_br && r_branch_en;
    reg  [4:0]  r_cmp_flags   = 5'h00; // Z, O, S, L, etc.
    reg  [15:0] r_cmp_result  = 5'h00; // a - b

    always @(posedge clk)
        if (r_instr_has_cmp)
            r_cmp_flags <= r_alu_out;

    always @(posedge clk)
        if (r_instr_has_br)
            case (r_instr_imm8)
                `VMICRO16_OP_BR_U:  r_branch_en <= 1;
                `VMICRO16_OP_BR_E:  r_branch_en <= r_cmp_flags[`VMICRO16_SFLAG_Z];
                `VMICRO16_OP_BR_L:  r_branch_en <= r_cmp_flags[`VMICRO16_SFLAG_L];
                default:            r_branch_en <= 0;
            endcase

    // 2 cycle register fetch
    always @(*) begin
        r_reg_rs1 = 0;
        if (r_state == STATE_R1)
            r_reg_rs1 = r_instr_rsd;
        else if (r_state == STATE_R2)
            r_reg_rs1 = r_instr_rsa;
        else
            r_reg_rs1 = 3'h0;
    end

    // cpu state machine
    always @(posedge clk)
        if (reset) begin
            r_pc              <= 0;
            r_state           <= STATE_IF;
            r_instr           <= 0;
            r_mem_scratch_req <= 0;
            r_instr_rdd       <= 0;
            r_instr_rda       <= 0;
        end
        else begin
            if (r_state == STATE_IF) begin
                r_instr <= w_mem_instr_out;

                $display("");
                $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
                $display($time, "\tC%02h: INSTR: %h", CORE_ID, w_mem_instr_out);

                r_state <= STATE_R1;
            end
            else if (r_state == STATE_R1) begin
                // primary operand
                r_instr_rdd <= r_reg_rd1;
                r_state     <= STATE_R2;
```

```
            end
            else if (r_state == STATE_R2) begin
                // Choose secondary operand (register or immediate)
                if      (r_instr_has_imm8)  r_instr_rda <= r_instr_imm8;
                else if (r_instr_has_imm4)  r_instr_rda <= r_reg_rd1 + r_instr_imm4;
                else                        r_instr_rda <= r_reg_rd1;

                if (r_instr_has_mem) begin
                    r_state           <= STATE_ME;
                    // Pulse req
                    r_mem_scratch_req <= 1;
                end else
                    r_state <= STATE_WB;


                if (w_branching) begin
                    $display($time, "\tbranching to %h", r_instr_rdd);
                    r_pc <= r_instr_rdd;
                end
                else if (r_pc < (MEM_INSTR_DEPTH-1))
                    r_pc <= r_pc + 1;

            end
            else if (r_state == STATE_ME) begin
                // Pulse req
                r_mem_scratch_req <= 0;
                // Wait for MMU to finish
                if (!r_mem_scratch_busy) r_state <= STATE_WB;
            end
            else if (r_state == STATE_WB) begin
                r_state <= STATE_IF;
            end
        end

    // Instruction ROM
    (* keep_hierarchy = "yes" *)
    vmicro16_bram # (
        .MEM_WIDTH      (16),
        .MEM_DEPTH      (MEM_INSTR_DEPTH),
        .CORE_ID        (CORE_ID),
        .NAME           ("INSTR_MEM")
    ) mem_instr (
        .clk            (clk),
        .reset          (reset),
        // port 1
        .mem_addr       (r_pc),
        .mem_in         (16'h0000),
        .mem_we         (1'b0),  // ROM
        .mem_out        (w_mem_instr_out)
    );

    // MMU
    (* keep_hierarchy = "yes" *)
    vmicro16_core_mmu # (
        .MEM_WIDTH      (`DATA_WIDTH),
        .MEM_DEPTH      (`DEF_MMU_TIM0_CELLS),
        .CORE_ID        (CORE_ID)
    ) mmu (
        .clk            (clk),
        .reset          (reset),
        .req            (r_mem_scratch_req),
        .busy           (r_mem_scratch_busy),
        // port 1
        .mmu_addr       (r_mem_scratch_addr),
        .mmu_in         (r_mem_scratch_in),
        .mmu_we         (r_mem_scratch_we),
        .mmu_out        (r_mem_scratch_out),
        // APB maste    r to slave
        .M_PADDR        (w_PADDR),
        .M_PWRITE       (w_PWRITE),
        .M_PSELx        (w_PSELx),
        .M_PENABLE      (w_PENABLE),
        .M_PWDATA       (w_PWDATA),
        .M_PRDATA       (w_PRDATA),
        .M_PREADY       (w_PREADY)
    );

    // Instruction decoder
    (* keep_hierarchy = "yes" *)
    vmicro16_dec dec (
        // input
        .instr          (r_instr),
        // output async
        .opcode         (),
        .rd             (r_instr_rsd),
        .ra             (r_instr_rsa),
        .imm4           (r_instr_imm4),
        .imm8           (r_instr_imm8),
        .imm12          (),
        .simm5          (r_instr_simm5),
        .alu_op         (r_instr_alu_op),
        .has_imm4       (r_instr_has_imm4),
        .has_imm8       (r_instr_has_imm8),
        .has_we         (r_instr_has_we),
        .has_br         (r_instr_has_br),
        .has_cmp        (r_instr_has_cmp),
        .has_mem        (r_instr_has_mem),
        .has_mem_we     (r_instr_has_mem_we),
        .halt           ()
    );

    // Software registers
    (* keep_hierarchy = "yes" *)
    vmicro16_regs # (
        .CORE_ID (CORE_ID)
    ) regs (
        .clk        (clk),
        .reset      (reset),
        // async port 0
        .rs1        (r_reg_rs1),
        .rd1        (r_reg_rd1),
        // async port 1
        //.rs2        (),
        //.rd2        (),
        // write port
        .we         (r_reg_we),
```

```
        .ws1        (r_instr_rsd),
        .wd         (r_reg_wd)
    );

    // ALU
    (* keep_hierarchy = "yes" *)
    vmicro16_alu alu (
        .op         (r_instr_alu_op),
        .a          (r_instr_rdd),
        .b          (r_instr_rda),
        // async output
        .c          (r_alu_out)
    );

endmodule
```

# vmicro16_soc.v

```
//
//

`include "vmicro16_soc_config.v"
`include "clog2.v"

(*dont_touch="true"*)
(* keep_hierarchy = "yes" *)
module vmicro16_soc (
    input clk,
    input reset,

    //input  uart_rx,
    output                          uart_tx,
    output [`APB_GPIO0_PINS-1:0]    gpio0,
    output [`APB_GPIO1_PINS-1:0]    gpio1,
    output [`APB_GPIO2_PINS-1:0]    gpio2,

    output reg [7:0]                dbug0,
    output     [`CORES*8:0]         dbug1
);
    initial dbug0 = 0;
    always @(posedge clk)
        dbug0 <= dbug0 + 1;

    // Peripherals (master to slave)
    (*dont_touch="true"*) wire [`APB_WIDTH-1:0]         M_PADDR;
    (*dont_touch="true"*) wire                          M_PWRITE;
    (*dont_touch="true"*) wire [`SLAVES-1:0]            M_PSELx;  // not shared
    (*dont_touch="true"*) wire                          M_PENABLE;
    (*dont_touch="true"*) wire [`APB_WIDTH-1:0]         M_PWDATA;
    (*dont_touch="true"*) wire [`SLAVES*`APB_WIDTH-1:0] M_PRDATA; // input to intercon
    (*dont_touch="true"*) wire [`SLAVES-1:0]            M_PREADY; // input

    // Master apb interfaces
    (*dont_touch="true"*) wire [`CORES*`APB_WIDTH-1:0] w_PADDR;
    (*dont_touch="true"*) wire [`CORES-1:0]            w_PWRITE;
    (*dont_touch="true"*) wire [`CORES-1:0]            w_PSELx;
    (*dont_touch="true"*) wire [`CORES-1:0]            w_PENABLE;
    (*dont_touch="true"*) wire [`CORES*`APB_WIDTH-1:0] w_PWDATA;
    (*dont_touch="true"*) wire [`CORES*`APB_WIDTH-1:0] w_PRDATA;
    (*dont_touch="true"*) wire [`CORES-1:0]            w_PREADY;

    (*dont_touch="true"*)
    (* keep_hierarchy = "yes" *)
    apb_intercon_s # (
        .MASTER_PORTS(`CORES),
        .SLAVE_PORTS (`SLAVES)
    ) apb (
        .clk        (clk),
        .reset      (reset),
        // APB master to slave
        .S_PADDR    (w_PADDR),
        .S_PWRITE   (w_PWRITE),
        .S_PSELx    (w_PSELx),
        .S_PENABLE  (w_PENABLE),
        .S_PWDATA   (w_PWDATA),
        .S_PRDATA   (w_PRDATA),
        .S_PREADY   (w_PREADY),
        // shared bus
        .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)
    );

    (*dont_touch="true"*)
    (* keep_hierarchy = "yes" *)
    vmicro16_gpio_apb # (
        .BUS_WIDTH  (`APB_WIDTH),
        .PORTS      (`APB_GPIO0_PINS)
    ) gpio0_apb (
        .clk        (clk),
        .reset      (reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_GPIO0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO0*`APB_WIDTH +: `APB_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
        .gpio       (gpio0)
    );

    // GPIO1 for Seven segment displays (16 pin)
    (*dont_touch="true"*)
    (* keep_hierarchy = "yes" *)
    vmicro16_gpio_apb # (
```

```
            .BUS_WIDTH  (`APB_WIDTH),
            .PORTS      (`APB_GPIO1_PINS)
        ) gpio1_apb (
            .clk        (clk),
            .reset      (reset),
            // apb slave to master interface
            .S_PADDR    (M_PADDR),
            .S_PWRITE   (M_PWRITE),
            .S_PSELx    (M_PSELx[`APB_PSELX_GPIO1]),
            .S_PENABLE  (M_PENABLE),
            .S_PWDATA   (M_PWDATA),
            .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO1*`APB_WIDTH +: `APB_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
            .gpio       (gpio1)
        );

        // GPIO2 for Seven segment displays (8 pin)
        (*dont_touch="true"*)
        (* keep_hierarchy = "yes" *)
        vmicro16_gpio_apb # (
            .BUS_WIDTH  (`APB_WIDTH),
            .PORTS      (`APB_GPIO2_PINS)
        ) gpio2_apb (
            .clk        (clk),
            .reset      (reset),
            // apb slave to master interface
            .S_PADDR    (M_PADDR),
            .S_PWRITE   (M_PWRITE),
            .S_PSELx    (M_PSELx[`APB_PSELX_GPIO2]),
            .S_PENABLE  (M_PENABLE),
            .S_PWDATA   (M_PWDATA),
            .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO2*`APB_WIDTH +: `APB_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
            .gpio       (gpio2)
        );

        (*dont_touch="true"*)
        (* keep_hierarchy = "yes" *)
        apb_uart_tx uart0_apb (
            .clk        (clk),
            .reset      (reset),
            // apb slave to master interface
            .S_PADDR    (M_PADDR),
            .S_PWRITE   (M_PWRITE),
            .S_PSELx    (M_PSELx[`APB_PSELX_UART0]),
            .S_PENABLE  (M_PENABLE),
            .S_PWDATA   (M_PWDATA),
            .S_PRDATA   (M_PRDATA[`APB_PSELX_UART0*`APB_WIDTH +: `APB_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
            // uart wires
            .tx_wire    (uart_tx),
            .rx_wire    (uart_rx)
        );

        // Shared register set for system-on-chip info
        // R0 = number of cores
        (*dont_touch="true"*)
        (* keep_hierarchy = "yes" *)
        vmicro16_regs_apb # (
            .BUS_WIDTH          (`APB_WIDTH),
            .CELL_DEPTH         (8),
            .PARAM_DEFAULTS_R0  (`CORES),
            .PARAM_DEFAULTS_R1  (`SLAVES)
        ) regs0_apb (
            .clk        (clk),
            .reset      (reset),
            // apb slave to master interface
            .S_PADDR    (M_PADDR),
            .S_PWRITE   (M_PWRITE),
            .S_PSELx    (M_PSELx[`APB_PSELX_REGS0]),
            .S_PENABLE  (M_PENABLE),
            .S_PWDATA   (M_PWDATA),
            .S_PRDATA   (M_PRDATA[`APB_PSELX_REGS0*`APB_WIDTH +: `APB_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
        );

        (*dont_touch="true"*)
        (* keep_hierarchy = "yes" *)
        vmicro16_bram_ex_apb # (
            .MEM_WIDTH    (`APB_WIDTH),
            .MEM_DEPTH    (`APB_BRAM0_CELLS),
            .CORE_ID_BITS (`clog2(`CORES))
        ) bram_apb (
            .clk        (clk),
            .reset      (reset),
            // apb slave to master interface
            .S_PADDR    (M_PADDR),
            .S_PWRITE   (M_PWRITE),
            .S_PSELx    (M_PSELx[`APB_PSELX_BRAM0]),
            .S_PENABLE  (M_PENABLE),
            .S_PWDATA   (M_PWDATA),
            .S_PRDATA   (M_PRDATA[`APB_PSELX_BRAM0*`APB_WIDTH +: `APB_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
        );

        genvar i;
        generate for(i = 0; i < `CORES; i = i + 1) begin : cores
            (* keep_hierarchy = "yes" *)
            vmicro16_core # (
                .CORE_ID    (i)
            ) c1 (
                .clk        (clk),
                .reset      (reset),
                .dbug_pc    (dbug1[i*8 +: 8]),

                .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  [`APB_WIDTH*i +: `APB_WIDTH] ),
                .w_PRDATA   (w_PRDATA  [`APB_WIDTH*i +: `APB_WIDTH] ),
                .w_PREADY   (w_PREADY  [i]                          )
            );
        end
        endgenerate

```

```
209
210 endmodule
```

# vmicro16_isa.v

```
1   // Vmicro16 multi-core instruction set
2   `include "vmicro16_soc_config.v"
3
4   // TODO: Remove NOP by making a register write/read always 0
5   `define VMICRO16_OP_NOP          5'b00000
6   `define VMICRO16_OP_LW           5'b00001
7   `define VMICRO16_OP_SW           5'b00010
8   `define VMICRO16_OP_BIT          5'b00011
9   `define VMICRO16_OP_BIT_OR       5'b00000
10  `define VMICRO16_OP_BIT_XOR      5'b00001
11  `define VMICRO16_OP_BIT_AND      5'b00010
12  `define VMICRO16_OP_BIT_NOT      5'b00011
13  `define VMICRO16_OP_BIT_LSHFT    5'b00100
14  `define VMICRO16_OP_BIT_RSHFT    5'b00101
15  `define VMICRO16_OP_MOV          5'b00100
16  `define VMICRO16_OP_MOVI         5'b00101
17  `define VMICRO16_OP_ARITH_U      5'b00110
18  `define VMICRO16_OP_ARITH_UADD   5'b11111
19  `define VMICRO16_OP_ARITH_USUB   5'b10000
20  `define VMICRO16_OP_ARITH_UADDI  5'b0????
21  `define VMICRO16_OP_ARITH_S      5'b00111
22  `define VMICRO16_OP_ARITH_SADD   5'b11111
23  `define VMICRO16_OP_ARITH_SSUB   5'b10000
24  `define VMICRO16_OP_ARITH_SSUBI  5'b0????
25  `define VMICRO16_OP_BR           5'b01000
26  `define VMICRO16_OP_CMP          5'b01001
27  `define VMICRO16_OP_SETC         5'b01010
28  `define VMICRO16_OP_MULT         5'b01011
29  `define VMICRO16_OP_HALT         5'b01100
30  `define VMICRO16_OP_LWEX         5'b01101
31  `define VMICRO16_OP_SWEX         5'b01110
32
33  // TODO: wasted upper nibble bits in BR
34  `define VMICRO16_OP_BR_U         8'h00
35  `define VMICRO16_OP_BR_E         8'h01
36  `define VMICRO16_OP_BR_NE        8'h02
37  `define VMICRO16_OP_BR_G         8'h03
38  `define VMICRO16_OP_BR_GE        8'h04
39  `define VMICRO16_OP_BR_L         8'h05
40  `define VMICRO16_OP_BR_LE        8'h06
41  `define VMICRO16_OP_BR_S         8'h07
42  `define VMICRO16_OP_BR_NS        8'h08
43
44  // microcode operations
45  `define VMICRO16_ALU_BIT_OR      5'h00
46  `define VMICRO16_ALU_BIT_XOR     5'h01
47  `define VMICRO16_ALU_BIT_AND     5'h02
48  `define VMICRO16_ALU_BIT_NOT     5'h03
49  `define VMICRO16_ALU_BIT_LSHFT   5'h04
50  `define VMICRO16_ALU_BIT_RSHFT   5'h05
51  `define VMICRO16_ALU_LW          5'h06
52  `define VMICRO16_ALU_SW          5'h07
53  `define VMICRO16_ALU_NOP         5'h08
54  `define VMICRO16_ALU_MOV         5'h09
55  `define VMICRO16_ALU_MOVI        5'h0a
56  `define VMICRO16_ALU_MOVI_L      5'h0b
57  `define VMICRO16_ALU_ARITH_UADD  5'h0c
58  `define VMICRO16_ALU_ARITH_USUB  5'h0d
59  `define VMICRO16_ALU_ARITH_SADD  5'h0e
60  `define VMICRO16_ALU_ARITH_SSUB  5'h0f
61  `define VMICRO16_ALU_BR_U        5'h10
62  `define VMICRO16_ALU_BR_E        5'h11
63  `define VMICRO16_ALU_BR_NE       5'h12
64  `define VMICRO16_ALU_BR_G        5'h13
65  `define VMICRO16_ALU_BR_GE       5'h14
66  `define VMICRO16_ALU_BR_L        5'h15
67  `define VMICRO16_ALU_BR_LE       5'h16
68  `define VMICRO16_ALU_BR_S        5'h17
69  `define VMICRO16_ALU_BR_NS       5'h18
70  `define VMICRO16_ALU_CMP         5'h19
71  `define VMICRO16_ALU_SETC        5'h1a
72  `define VMICRO16_ALU_ARITH_UADDI 5'h1b
73  `define VMICRO16_ALU_ARITH_SSUBI 5'h1c
74  `define VMICRO16_ALU_BR          5'h1d
75  `ifdef DEF_ALU_HW_MULT
76  `define VMICRO16_ALU_MULT        5'h1e
77  `endif
78  `define VMICRO16_ALU_BAD         5'h1f
79
80  `define VMICRO16_SFLAG_U         5'h00
81  `define VMICRO16_SFLAG_Z         5'h01
82  `define VMICRO16_SFLAG_L         5'h02
```