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

ELEC5881M - Final Report

Ben David Lancaster
Student ID: 201280376

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

Supervisor: Dr. David Cowell
Assessor: Mr David Moore

University of Leeds
School of Electrical and Electronic Engineering

August 26, 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.

Declaration of Academic Integrity

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

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

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

Name: Ben David Lancaster
Date: August 26, 2019

Acknowledgements

I would like to thank my supervisor, David Cowell, and assessor, David Moore, for giving me the opportunity to explore a project of my choosing.

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.

Table of Contents

List of Figures

List of Tables

List of Listings

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 challenges of software and hardware 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 [5], 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 [6]. 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)* [1] 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.

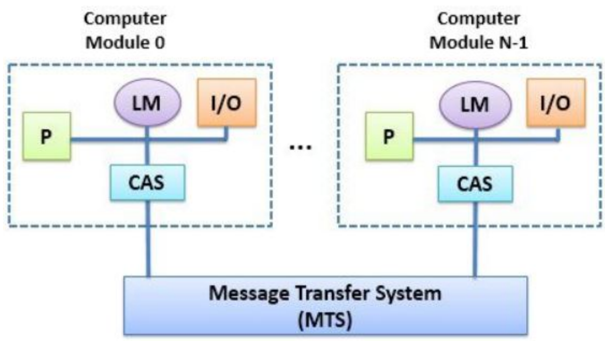


Figure 2.1: A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [1].

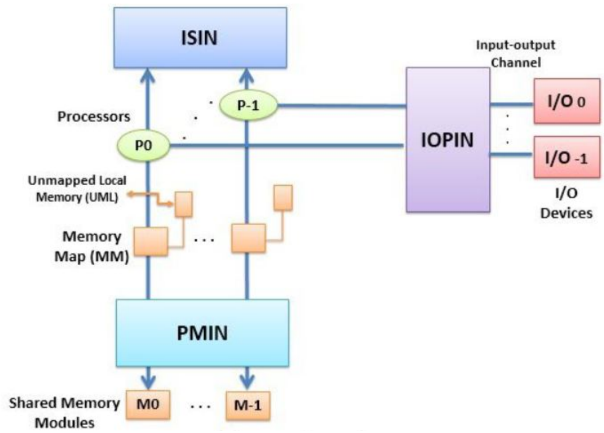


Figure 2.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [1].

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 [7]. 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 [8].

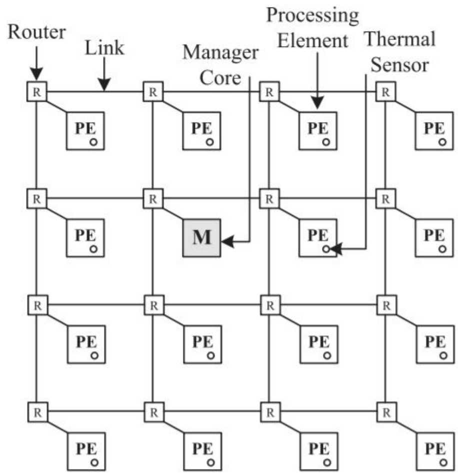


Figure 2.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [2].

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 [9, 10]). 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 [11] compiler to support the ISA from CD1. This will make it easier to build complex multi-core software for the processor.

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

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

ED7 Store instruction memory in SPI flash.

ED8 Reprogram instruction memory at runtime from host computer.

ED9 Processor external debugger using host-processor link.

3.2 Project Timeline

3.2.1 Project Stages

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

The project stages are shown in Table 3.1.

3.2.2 Project Stage Detail

Stages 1.0 through 1.2 – Research and Project Conception

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

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

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

Stages 3.1 through 3.4 – Advanced Processor Implementation

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

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

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

Stages 4.1 through 4.4 – Multiprocessor Functionality

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

Stages 5.1 through 5.3 – Debugging Features

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

Stages 6.1 through 6.2 – Compiler Backends

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

Stage 7.1 – Wishbone Peripherals

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

Stage 8.1 – Final Report

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

3.2.3 Timeline

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

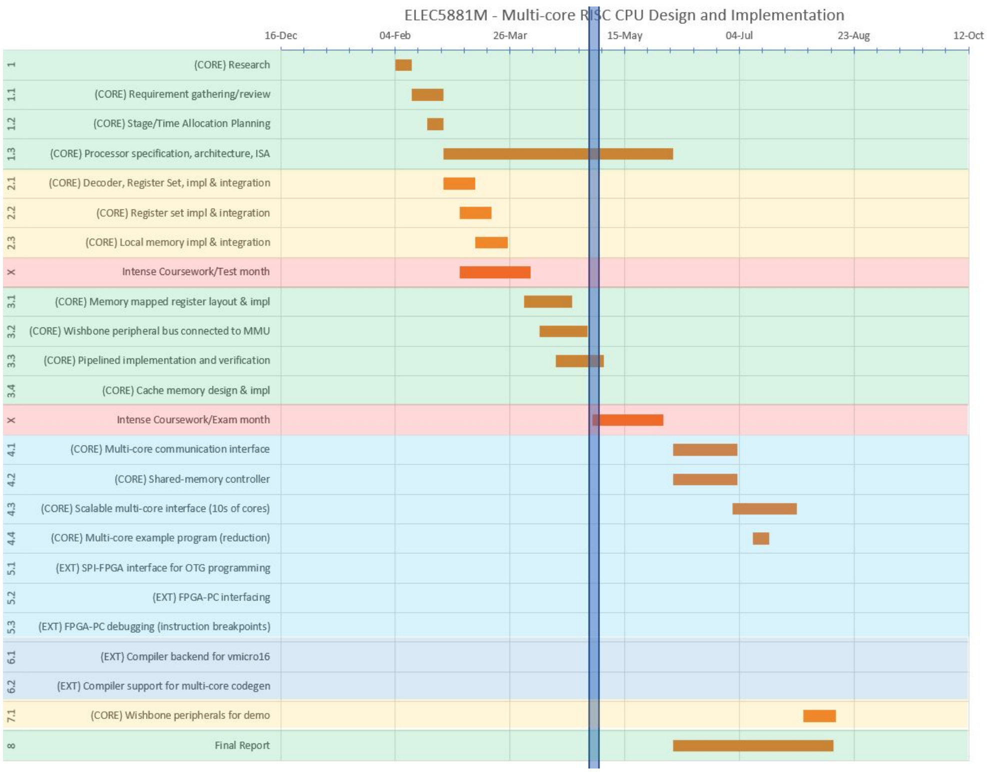


Figure 3.1: Project stages in a Gantt chart.

3.3 Resources

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

3.3.1 Hardware Resources

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

Terasic DE1-SoC Development Board

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

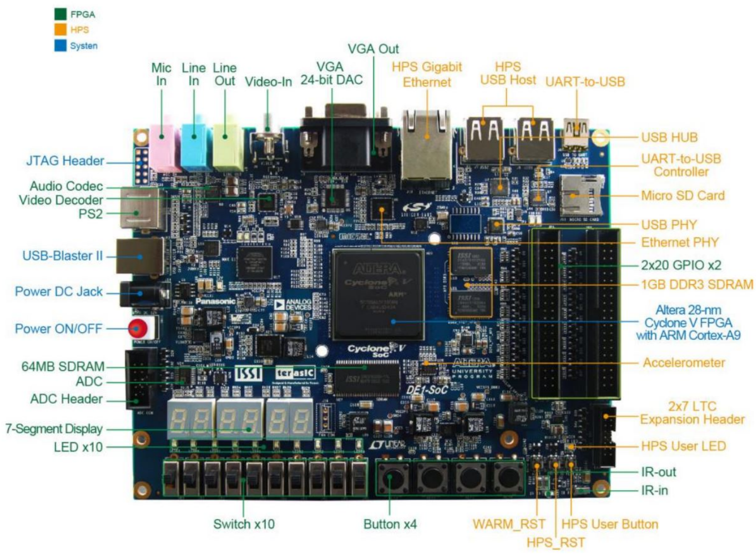


Figure 3.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [3].

Minispartan 6+ FPGA Development Board

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

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

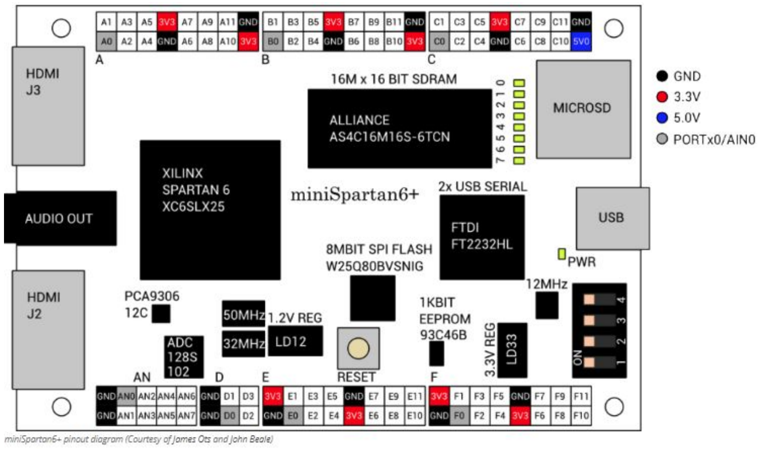


Figure 3.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [4].

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 [12].

Chapter 4

Single-core Design

4.1 Introduction

While the majority of this report will focus on the multi-processing functionality of this project, it is important understand the design decisions of the single core to understand the features and limitations of the multi-core system-on-chip as a whole.

4.2 Design and Implementation

The single-core design is a traditional 5-stage RISC processor (fetch, decode, execute, memory, write-back). The core uses separate instruction and data memories in the style of a Harvard architecture [**?**].

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.

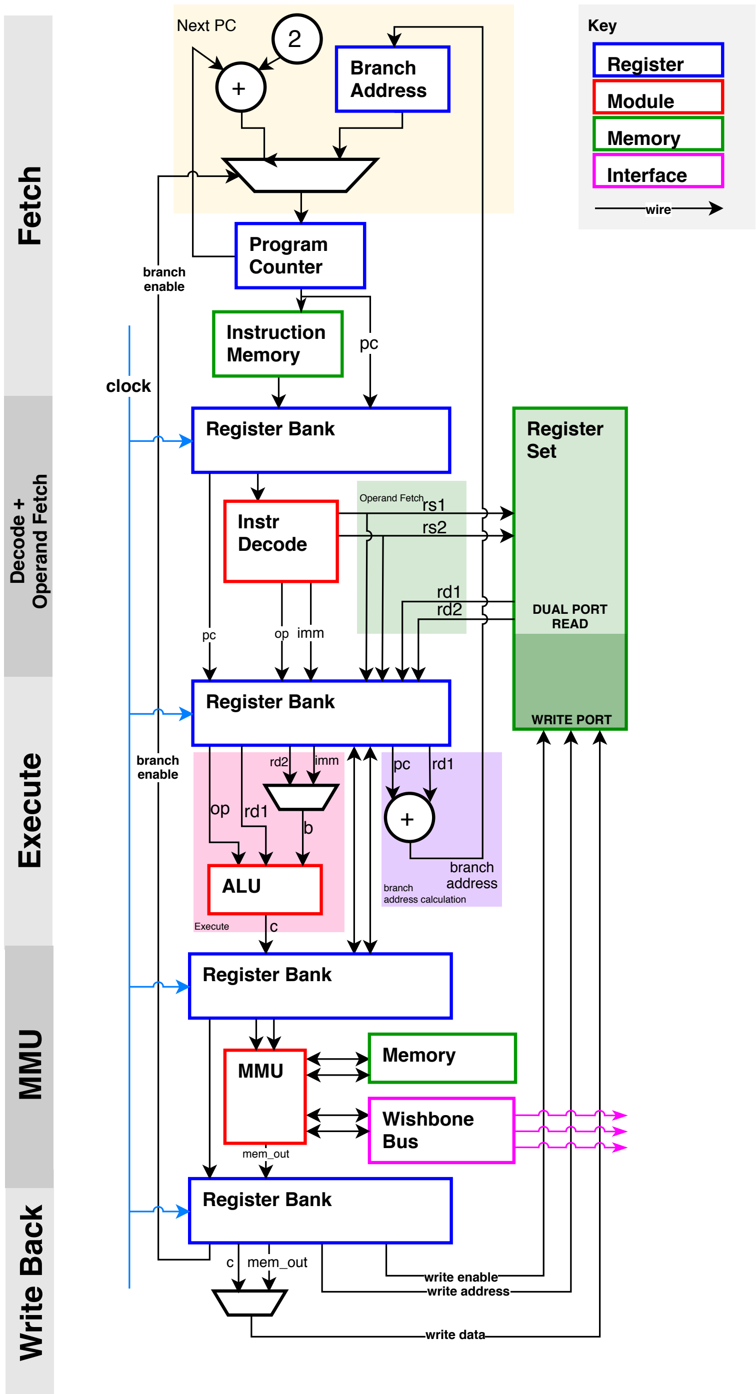


Figure 4.1: Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

A small reduction in size within the single-core will result in substantial size reductions in

4.2.1 Instruction Set Architecture

Core deliverable **CD1** details the background for the requirement of a custom instruction set architecture. The 16-bit instruction set listing is shown in Figure B.2.

In this proposed architecture, most instructions are *destructive*, meaning that source operands also act as the destination, hence effectively *destroying* the original source operand.

This design decision reduces the complexity of the ISA as traditional three operand instructions, for example `add r0, r0, r1`, can be encoded using only two operands `add r0, r1`. However, this does increase the complexity of compilers as they may need to make temporary copies of registers as the instructions will *destroy* the original source data.

The instruction set is split into 7 categories (highlighted by colours in Figure B.2):

- Special instructions, such as halting and interrupt returns;
- Bitwise operations, such as XOR and AND;
- Signed arithmetic;
- Unsigned arithmetic;
- Conditional branches and compare instructions;
- and Load/store instructions, with their atomic equivalents.

4.2.2 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 transparently uses the existing `LW[EX]/SW[EX]` to easily provide an arbitrary number of peripherals/special purpose addresses to the software running on the processor.

4.2.3 Instruction and Data Memory

The design uses separate instruction and data memories similar to a Harvard architecture computer. This architecture was chosen due because it is generally easier to implement, however later resulted in design challenges in large multi-core designs. This is discussed later in the report.

Each single-core has it's own *scratch* memory – a small RAM-like memory which can be used for stack-space and arrays too large to fit into the 8 registers. These memories are provided as is – meaning it's up to the software to implement and provide any stack-frame, function, and calling, functionality. Each core also features it's own read-only instruction memory that is programmed at compile time of the design, or via the `UART0` reciever interface (discussed later). Both of these memories map onto synchronous, read-first, single-port, FPGA block RAMs to minimise LUT requirements.

Users can customise the size of these memories by tweaking the following parameters in the `vmicro16_soc_config.v` file: `DEF_MEM_INSTR_DEPTH` for the instruction memory, and `DEF_MEM_SCRATCH_DEPTH` for the scratch memory.

4.2.4 ALU Design

The Vmicro16's ALU is an asynchronous module that has 3 inputs: data a; data b; and opcode op; and outputs data 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.

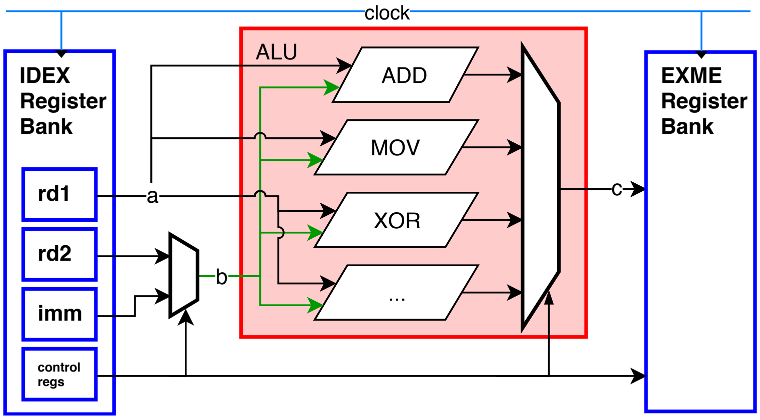


Figure 4.2: Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

The ALU also performs comparison (`CMP`) operations in which it returns flags similar to X86's overflow, signed, and zero, flags. The combination of these flags can be used to easily compute relationships between the two input operands. For example, if the zero flag is not equal to the signed flag, then the relationship between inputs a and b is that $a < b$.

```
module branch (
    input [3:0] flags,
    input [7:0] cond,
    output reg  en
);
    always @(*)
        case (cond)
            `VMICRO16_OP_BR_U:  en = 1;
            `VMICRO16_OP_BR_E:  en = (flags[`VMICRO16_SFLAG_Z] == 1);
            `VMICRO16_OP_BR_NE: en = (flags[`VMICRO16_SFLAG_Z] == 0);
            `VMICRO16_OP_BR_G:  en = (flags[`VMICRO16_SFLAG_Z] == 0) &&
                                     (flags[`VMICRO16_SFLAG_N] == flags[`VMICRO16_SFLAG_V]);
            `VMICRO16_OP_BR_L:  en = (flags[`VMICRO16_SFLAG_Z] != flags[`VMICRO16_SFLAG_N]);
            `VMICRO16_OP_BR_GE: en = (flags[`VMICRO16_SFLAG_Z] == flags[`VMICRO16_SFLAG_N]);
            `VMICRO16_OP_BR_LE: en = (flags[`VMICRO16_SFLAG_Z] == 1) ||
                                     (flags[`VMICRO16_SFLAG_N] != flags[`VMICRO16_SFLAG_V]);
            default:            en = 0;
        endcase
endmodule
```

Listing 1: ALU branch detection using flags: zero (Z), overflow (V), and negative (N).

The Verilog implementation of the ALU is shown in Listing 2. The ALU's asynchronous output is clocked with other registers, such as destination register `rs1` and other control signals, in the `EXME` register bank.

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

Listing 2: Vmicro16's ALU implementation named vmicro16_alu. vmicro16.v

4.2.5 Decoder Design

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

```
always @(*) case (instr[15:11])
    `VMICRO16_OP_BR:               alu_op = `VMICRO16_ALU_BR;
    `VMICRO16_OP_MULT:             alu_op = `VMICRO16_ALU_MULT;

    `VMICRO16_OP_CMP:              alu_op = `VMICRO16_ALU_CMP;
    `VMICRO16_OP_SETC:             alu_op = `VMICRO16_ALU_SETC;

    `VMICRO16_OP_BIT:      casez (instr[4:0])
        `VMICRO16_OP_BIT_OR:       alu_op = `VMICRO16_ALU_BIT_OR;
        `VMICRO16_OP_BIT_XOR:      alu_op = `VMICRO16_ALU_BIT_XOR;
        `VMICRO16_OP_BIT_AND:      alu_op = `VMICRO16_ALU_BIT_AND;
        `VMICRO16_OP_BIT_NOT:      alu_op = `VMICRO16_ALU_BIT_NOT;
        `VMICRO16_OP_BIT_LSHFT:    alu_op = `VMICRO16_ALU_BIT_LSHFT;
        `VMICRO16_OP_BIT_RSHFT:    alu_op = `VMICRO16_ALU_BIT_RSHFT;
        default:                   alu_op = `VMICRO16_ALU_BAD; endcase
```

Listing 4: Vmicro16's decoder module code showing nested bit switches to determine the intended opcode. vmicro16.v

In Listing 4, it can be seen that the first 4 opcode cases (BR, MULT, CMP, SETC) are represented using the same 15-11 (opcode) bits, however the BIT instructions share the same opcode and so require another bit range to be compared to determine the output function.

4.2.6 Pipelining

In the interim progress update, the processor design featured *instruction pipelining* to meet requirement **ED1**. Instruction pipelining allows instructions executions to be overlapped in the pipeline, resulting in higher throughput (up to one instruction per clock) at the expense of 5-6 clocks of latency and *significant* code complexity. As the development of the project shifted from single-core to multi-core, it became obvious that the complexity of the pipelined processor would inhibit the integration of multi-core functionality. It was decided to remove the instruction pipelining functionality and use a simpler state-machine based pipeline that is much simpler to extend and would cause fewer challenges later in the project.

4.2.7 Design Optimisations

In a design that has many instantiations of the same component, a small resource saving improvement within the component can have a significant overall savings improvement if it is instantiated many times. Project requirement **CD5** requires the design to be compiled for a range of FPGA sizes, and so space saving optimisations are considered.

Register Set Size Improvements

A register set in a CPU is a fast, temporary, and small memory that software instructions directly manipulate to perform computation. In the Vmicro16 instruction set, eight registers named r0 to r7 are available to software. The instruction set allows up to two registers to be references in most instructions, for example the instruction `add r0, r1` tells the processor to perform the following actions:

Clock 1. Fetch r0 and r1 from the register set

Clock 2. Add the two values together in the ALU

Clock 3. Store the result back the register set in r0

For Clock 1, it was originally decided to use a dual port register set (meaning that two data reads can be performed in a single clock, in this case r0 and r1), however due to the asynchronous design of the register set (for speed) the RTL produced consumed a significant amount of FPGA resources, approximately 256 flip-flops (16 (data width) * 8 (registers) * 2 (ports)). To reduce this, it was decided to split task 1 into two steps over two clock cycles using a single-port register set. This required the processor pipe-line to use another clock cycle resulting in slightly lower performance, however the size improvements will allow for more cores to be instantiated in the design. This optimisation is also applied to the interrupt register set, resulting in a saving of approximately 256 flip-flops per core (128 in the normal mode register set, and 128 in the interrupt register set). As shown, adding a single clock delay saves a significant amount of LUTs. This saving will be amplified in designs with many cores.

4.3 Interrupts

Interrupts are a technique used by processors to run software functions when an event occurs within the processor, such as exceptions, or signalled from an external source, such as a UART receiver signalling it has received new data. Today, it is common for micro-controllers, soft-processors, and desktop processors, to all feature interrupts. Modern implementations support an *interrupt vector* which is a memory array that contains addresses to different *interrupt handlers* (a software function called when a particular interrupt is received).

Although interrupts are not a requirement for a multi-core system, it was decided to implement this functionality to boost my understanding of such systems. In addition, example demos provided with this project are better visualised with a interrupt functionality.

4.3.1 Overview

The interrupt functionality in this project supports the following:

- Per-core 8 cell interrupt vector accessible to software.
  Software programs running on the Vmicro16 processor can edit the interrupt vector to add their own interrupt handlers at runtime.

- Fast context switching.
  A dedicated interrupt register set is multiplexed with the normal mode register set to provide faster context switching. It should be noted that only the registers are saved during a context switch. The means that the stack is not saved. A schematic of the register multiplex is shown in Figure B.1.

- Parametrised interrupt sources and widths.
  Users can configure the width of the interrupt in signals and the data width per interrupt source via the `vmicro16_soc_config.v`. By default, 8 interrupt sources are available and each can provide 8-bits of data.

4.3.2 Hardware Implementation

Context Switching

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

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

The timing diagram in Figure 4.3 shows the behavioural logic for the TIMR0 interrupt source.

Figure 4.3: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return instruction (indicated by w_intr) the processor returns to normal mode and restores the normal state.

4.3.3 Software Interface

A memory-mapped software interface is provided through the MMU to allow easy software control of the interrupt behaviour. The interface is provided at the address range 0x0100 to 0x0108. This interface is per-core allowing each core to individually control what interrupts it receives and what functions to call upon an interrupt. This enables complex functionality, such as allowing each core to execute different functions upon the same interrupt.

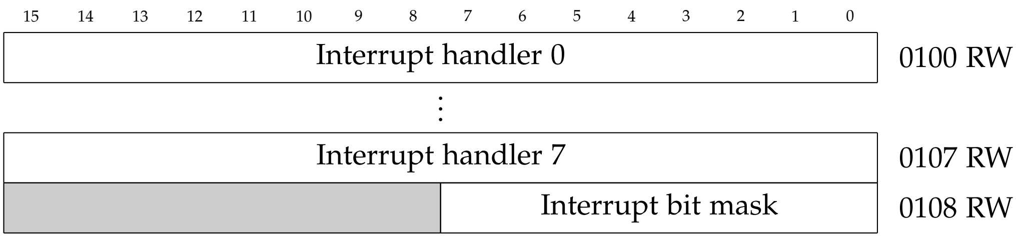


Figure 4.4: The interrupt vector (0x0100 - 0x0107) consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

Interrupt Vector (0x0100-0x0107)

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

Interrupt Mask (0x0108)

The interrupt mask is a per-core register that is used to mask/listen specific interrupt sources. This enables processing cores to individually select which interrupts they respond to. This allows for multi-processor designs where each core can be used for a particular interrupt

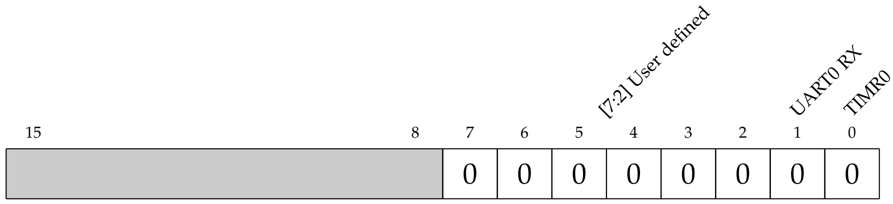


Figure 4.5: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign. Bit 0 is assigned to TIMR0's interval timer. Bit 1 is assigned to the UART0's receiver (unassigned if DEF_USE_REPROG is enabled).

source, improving the time response to the interrupt for time critical programs. The Interrupt Mask register is an 8-bit read/write register where each bit corresponds to a particular interrupt source and each bit corresponds with the interrupt handler in the interrupt vector. The interrupt mask register is shown in Figure 4.5.

Software Example

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

```
setup_interrupts:
    // Set interrupt vector at 0x100
    // Move address of isr0 function to vector[0]
    movi    r0, isr0
    // create 0x100 value by left shifting 1 8 bits
    movi    r1, #0x1
    movi    r2, #0x8
    lshft   r1, r2
    // write isr0 address to vector[0]
    sw      r0, r1

enable_interrupts:
    // enable all interrupts by writing 0x0f to 0x108
    movi    r0, #0x0f
    sw      r0, r1 + #0x8 // (0x100 + 0x8 = 0x108)
    halt                  // enter low power idle state

isr0:                     // arbitrary name
    movi    r0, #0xff     // do something
    intr                  // return from interrupt
```

A more complex example software program utilising interrupts and the TIMR0 interrupt is described in section D.1.

4.3.4 Design Improvements

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

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

4.4 Verification

Various verification techniques are employed to ensure correct operation of the processor.

The first technique involves using static assertions to identify incorrect configuration parameters at compile time, such as having zero instruction memory and scratch memory depth. These assertions use the `static_assert` for top level checks and `static_assert_ng` for checks inside `generate` blocks.

The second verification technique is to use assertions in `always` blocks to identify incorrect behavioural states. This is done using the `rassert` (run-time assert) macro.

The third verification technique is to use automatic verifying test benches. These test benches drive components of the processor, such as the ALU and decoder, and check the output against the correct value. This uses the `rassert` macro.

The final method of verification is to verify the complete design via a behavioural test bench. The design is passed a compiled software program with a known expected output, and is ran until the `r_halt` signal is raised. The test bench then checks the value on the `debug0`, `debug1`, and `debug2` signals against the expected value. If this matches, then it is assumed that sub-components of the design also operate correctly. This technique does not monitor the states of sub-components and statistics (such as time taken to execute an instruction), there leaves the possibility that some components could have entered an illegal state.

Chapter 5

Interconnect

5.1 Introduction

The Vmicro16 processor needs to communicate with multiple peripheral modules (such as UART, timers, GPIO, and more) to provide useful functionality for the end user.

Previous peripheral interface designs of mine have been directly connected to a main driver with unique inputs and outputs that the peripheral required. For example, a timer peripheral would have dedicated wires for it's load and prescaler values, wires for enabling and resetting, and wires for reading. A memory peripheral would have wires for it's address, read and write data, and a write enable signal. This resulted in each peripheral having a unique interface and unique logic for driving the peripheral, which consumed significant amounts of limited FPGA resources.

It can be seen that many of the peripherals need similar inputs and outputs (for example read and write data signals, write enables, and addresses), and because of this, a standard interface can be used to interface with each peripheral. Using a standard interface can reduce logic requirements as each peripheral can be driven by a single driver.

5.1.1 Comparison of On-chip Buses

The choice of on-chip interconnect has changed multiple times over the life-cycle of this project, primary due to ease of implementation and resource requirements.

Originally, it was planned to use the Wishbone bus [?] due to it's popularity within open-source FPGA modules and good quality documentation.

Late in the project, it was decided to use the AMBA APB protocol [?] as it is more commonly used in large commercial designs and understanding how the interface worked would better benefit myself. APB describes an intuitive and easy to implement 2-state interface aimed at communicating with low-throughput devices, such as UARTs, timers, and watchdogs.

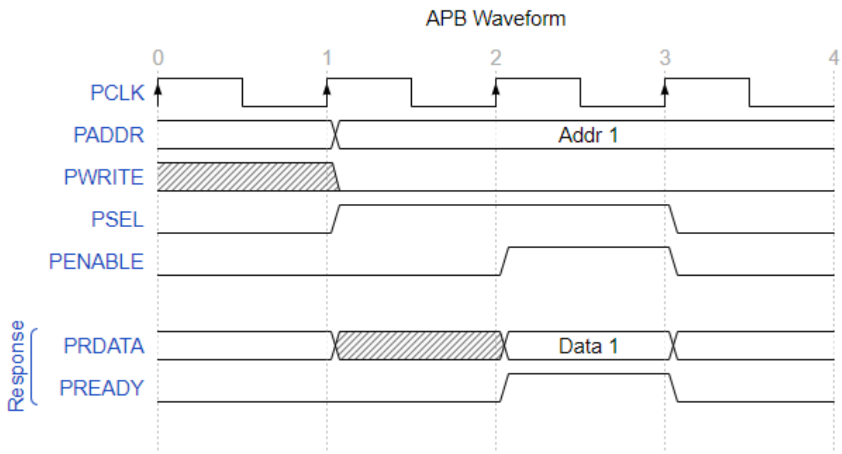


Figure 5.1: Waveform showing an APB read transaction.

5.2 Overview

The system-on-chip design is split into 3 main parts: peripheral interconnect (red), CPU array (gray), and the instruction memory interconnect (green).

A block diagram of this project is shown in Figure 5.2

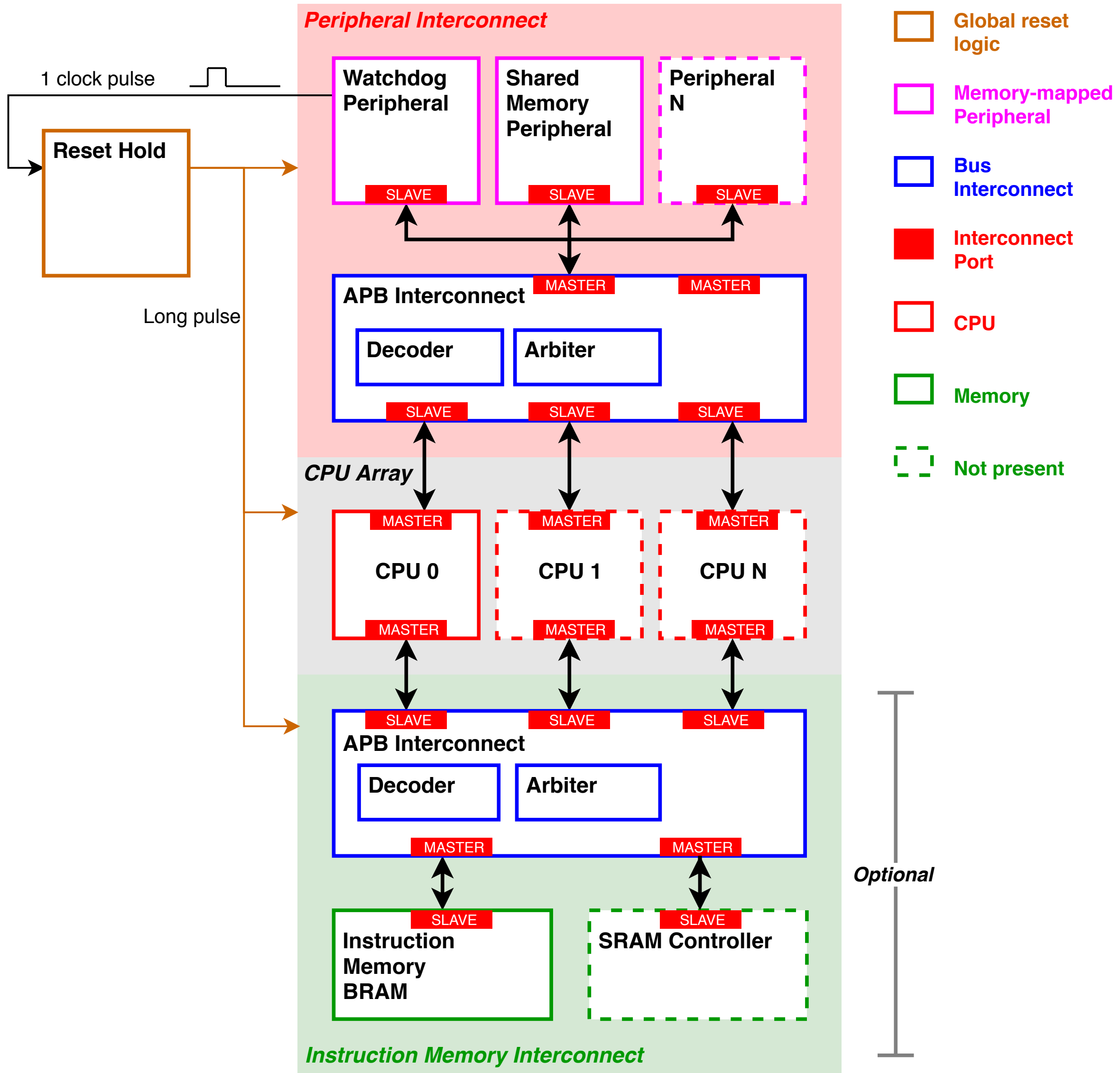


Figure 5.2: Block diagram of the Vmicro16 system-on-chip.

5.2.1 Design Considerations

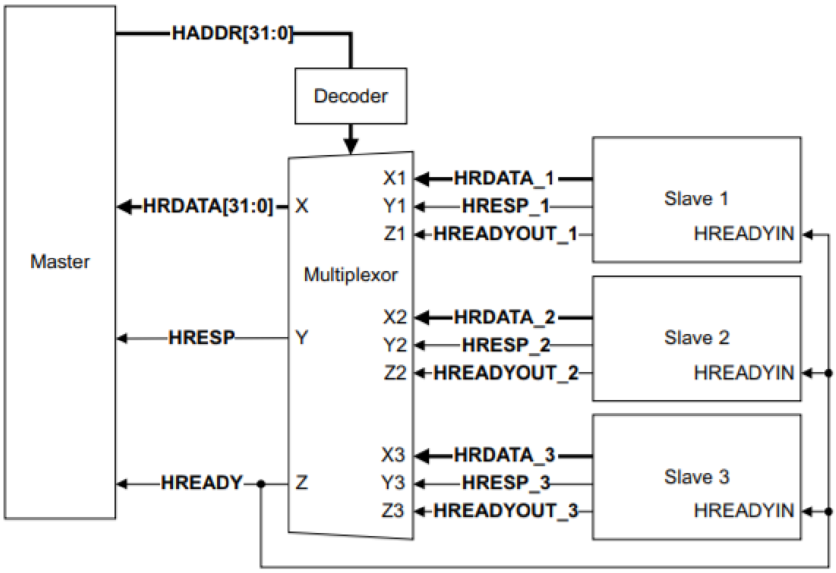
There are several design issues to consider for this project. These are listed below:

- **Design size limitations**
  The target devices for this project are small to medium sized FPGAs (featuring approximately 10,000 to 30,000 logic cells). Because of this, it is important to use a bus interconnect that has a small logic footprint yet is able to scale reasonably well.

- **Ease of implementation**
  The interconnect and any peripherals should be easy to implement within the time allocations specified in Figure 3.1.

- **Scalable**
  The interconnect should allow for easy scalability of master and slave interfaces with minimal code changes.

5.3 Interfaces



5.3.1 Master to Slave Interface

20	19	18	17	16	15	0	
LE	SE	CORE_ID			Address		PADDR[20:0]
					Write data		PWDATA[15:0]
					Read Data		PRDATA[15:0]
						WE	PWRITE[0:0]
						EN	PENABLE[0:0]

5.3.2 Multi-master Support

In this design, each processor can act as an APB master to communicate with peripherals, for example to write a value to UART or to the shared memory peripheral. Because each core runs independently from other cores, it is likely, especially is many-core systems, that two or more processors will want to use the peripheral bus at the same time.

As the peripheral and instruction interconnects use a shared one-to-many (one master to many slaves) bus architecture, only one master can use the bus at any-time. To enable multiple masters to use the bus, a device called an *arbiter* must be used to control which master gets access to drive the shared interconnect.

Arbiters can vary in complexity, mostly relative to throughput requirements.

An ideal arbiter for this interconnect, which ideally features many, possibly tens of, high-throughput masters, would likely feature a priority-based and pipelined arbiter with various devices to improve performance such as cache-coherencies.

Overview

Due to this project's limited time, and my personal knowledge in this area, a simple rotating arbiter is used. This arbitration scheme is likely the simplest that can be thought of. A schematic of arbiter interconnect is shown in Figure 5.3.

In this scheme, access to the bus is given incrementally to each master port, even if the master port has not requested to use the bus. The active master port can use the bus for as long as it requires, and signals it has finished by lowering the PSEL signal. When the PSEL signal is lowered, the arbiter grants access to the next master port. If this next master port has not raised it's PSEL signal (i.e. it has not requested access to the bus) then the arbiter grants access to the next master port, and so on. In Verilog, this is simply an incremental counter which is used to index the master ports array. To support a variable number of master ports, the width of each APB signal is multiplied by the number of cores, as shown in Listing 7.

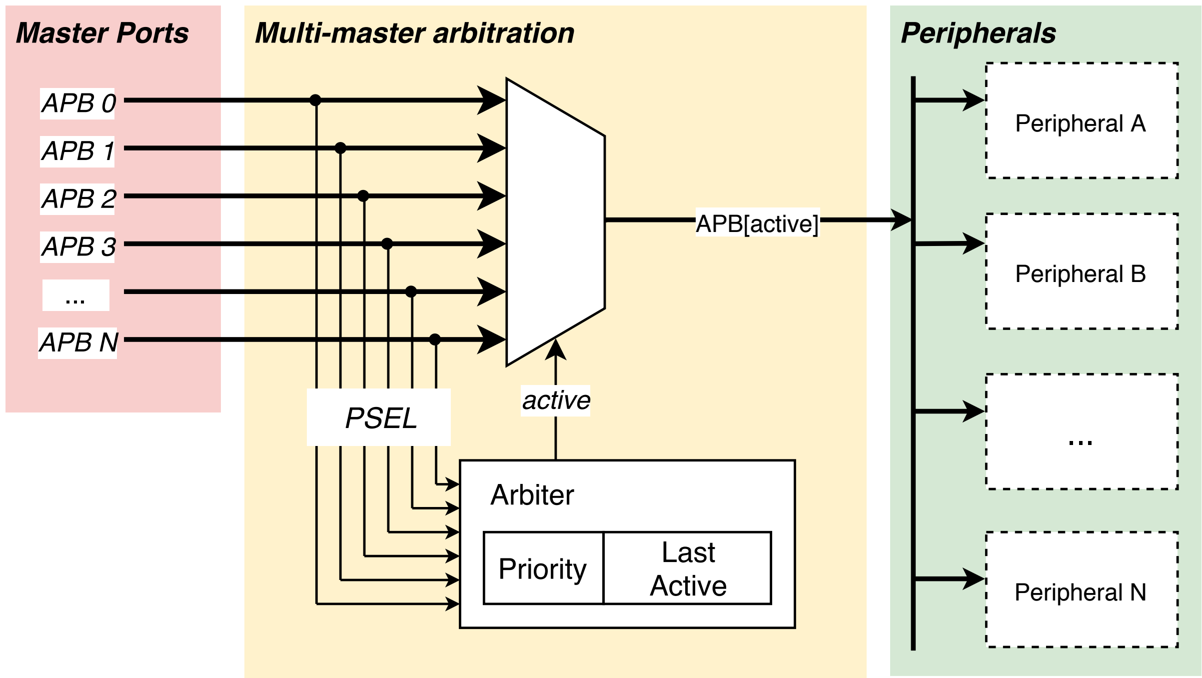


Figure 5.3: Foo

83	62	41	20	0
Core N-1	⋯	Core 1	Core 0	

5.4 Further Work

The submitted design is acceptable for a multi-core system as it fulfils the following requirements:

- Support an arbitrary number of peripherals.
- Supports memory-mapped address decoding.
- Supports multiple master interfaces.

Arbiter Performance Improvements

However, it fails in the performance aspect. A one clock penalty occurs if the next master port has not requested the bus. This may seem a small price to pay for such a simple arbiter design, however it can add up significantly in many-core designs. For example, if core #0 performs some action on the bus, but core #10 is the next master that wants to use the bus, then the arbiter will waste time incremental granting access to cores #1 to #9 which do not need the bus. This is also made worse when one of the cores is blocking access to a peripheral resource, such as through a mutex or semaphore.

To overcome this penalty, a scheme could use an algorithm to find the next master port requesting access, and grant access directly to it when the current master has finished. Another scheme could be to use a priority encoder. Here, a hard-coded lookup table (LUT) could be used, where the inputs are each master port's PSEL signal (acting as a bus request line) and the output being which master to grant access to. As this is targetting FPGA devices, this implemented would require few LUT resources for the arbiter, due to the hard-coded LUT approach. An example of this is given in M. Weber's *Arbiter: Design Ideas and Coding Styles* [13, p. 2].

APB Bus Errors and Recovery

This project's implementation of a multi-master APB interconnect does not provide a method of detecting errors and stalls. This is mainly due to time constraints.

An easy error that could be detected is `PADDR` addresses that do not fall into a memory-mapped address range. This can easily and cheaply be detected in the address decoding module. This will be discussed in detail in the next chapter.

As previously stated, the active bus master can take control of the bus for as long as it wants to. This is useful for high-throughput transactions, such as memory operations to global memory, but detecting a stalled or glitched operation is not immediately identifiable. If an active master stalls or glitches, it may not be able to lower the PSEL line which appears to the arbiter that the transaction is still happening normally. To overcome this, a timer could be used to detect stalled operations and reset the affected peripheral (essential a watchdog but for an interconnect).

Chapter 6

Memory Mapping

The Vmicro16 processor uses a memory-mapping scheme to communicate with peripherals and other cores. This chapter describes the design decisions and implementation of the memory-map used in this project.

6.1 Introduction

Memory mapping is a common technique used by CPUs, micro-controllers, and other system-on-chip devices, that enables peripherals and other devices to be accessed via a memory address on a common bus. In a processor use-case, this allows for the reuse of existing instructions (commonly memory load/store instructions) to communicate with external peripherals with little additional logic.

6.2 Address Decoding

An address decoder is used to determine the peripheral that the address is requesting. The address decoder module, `addr_dec` in `apb_intercon.v`, takes the 16-bit `PADDR` from the active APB interface and checks for set bits to determine which peripheral to select. The decoder outputs a chip enable signal `PSEL` for the selected peripheral. For example, if bit 12 is set in `PADDR` then the shared memory peripheral's `PSEL` is set high and others to low. A schematic for the decoder is shown in Figure 6.1.

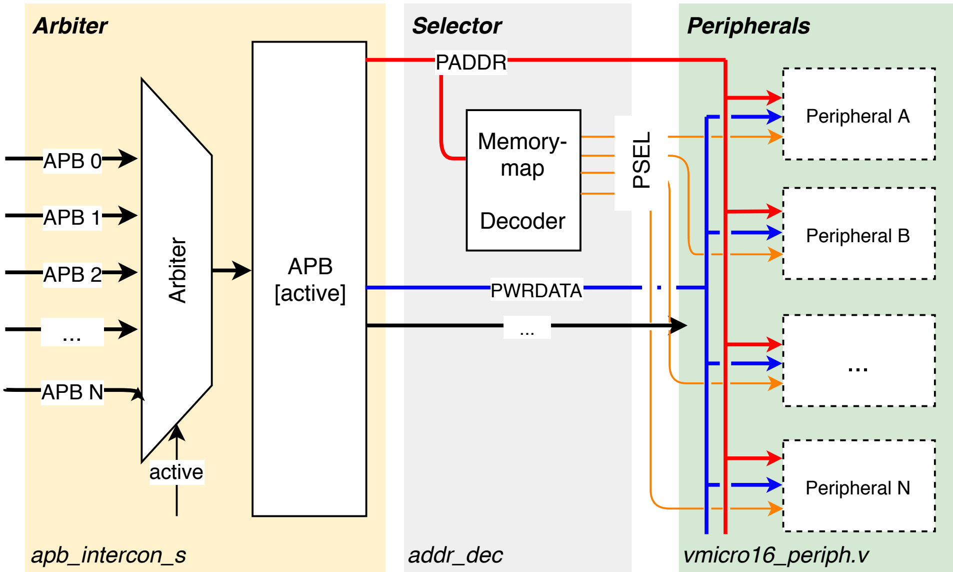


Figure 6.1: Schematic showing the address decoder (addr_dec) accepting the active PADDR signal and outputting PSEL chip enable signals to each peripheral.

6.2.1 Decoder Optimisations

Performing a 16-bit equality comparison of the PADDR signal against each peripheral memory address consumes a significant amount of logic. Depending on the synthesis tools and FPGA features, a 16-bit comparator might require a fixed 16-bit value input to compare against (where the 0s are inverted) and a wide-AND to reduce and compare [14, 15]. An example 4-bit comparator is shown below in Figure 6.2.

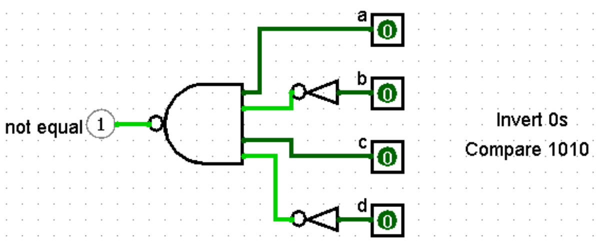


Figure 6.2: Example 4-bit binary comparator which compares the bits (a, b, c, d) to the constant value 1010. The 0s of the constant are inverted and then all are passed to a wide-AND.

As we are targeting FPGAs, which use LUTs to implement combinatorial logic, we can conveniently utilise Verilog's == operator on fairly large operands without worrying about consuming too many resources. The targeted FPGA devices in this project, the Cyclone V and Spartan 6, feature 6-input LUTs which allow 64 different configurations [16, 17]. Knowing this, we can design the address decoder to utilise the FPGA's LUTs more effectively and reduce it's footprint significantly.

We can use part of the PADDR signal as a chip select and the other bits as sub-addresses to interface with the peripheral. The addressing bits are passed into the FPGA's 6-input LUTs which are programmed (via the bitstream) to output 1 or 0 depending on the address. Figure 6.3 below shows a LUT based approach to address decoding which will utilise approximately one ALM/CLB module per peripheral chip select (PSEL) and one for error detection. This method of comparison (LUT based) is utilised in the `addr_dec` module in `apb_intercon.v`.

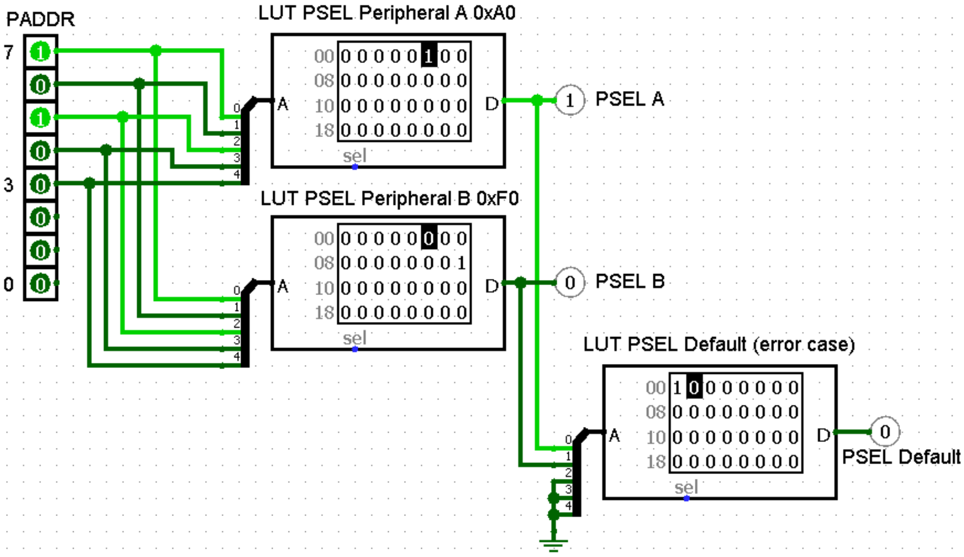


Figure 6.3: Bits [7:3] of an 8-bit PADDR signal are used as inputs to 5-bit LUTs to generate a PSEL signal. In addition, a default error case is shown allowing the address decoder to detect incorrect PADDR values (e.g. if no PSEL signals are generated).

The address decoding methods discussed above are examples of *full-address* decoding, where each bit (whether required or not) is compared. It is possible to further reduce the required logic by utilising *partial-address* decoding [18]. Partial-address decoding can reduce logic requirements by not using all bits. For example, if bits in address `0x0100` do not conflict with bits in other addresses (i.e. bit 8 is high in more than 1 address), then the address decoder needs only concern bit 8, not the other bits. This is visualised in Figure 6.4 below. This method is utilised in the MMU's address decoder (module `vmicro16_mmu` in `vmicro16.v:181`). As this is an optimisation per core, significant resources can be saved when a large number of cores are used.

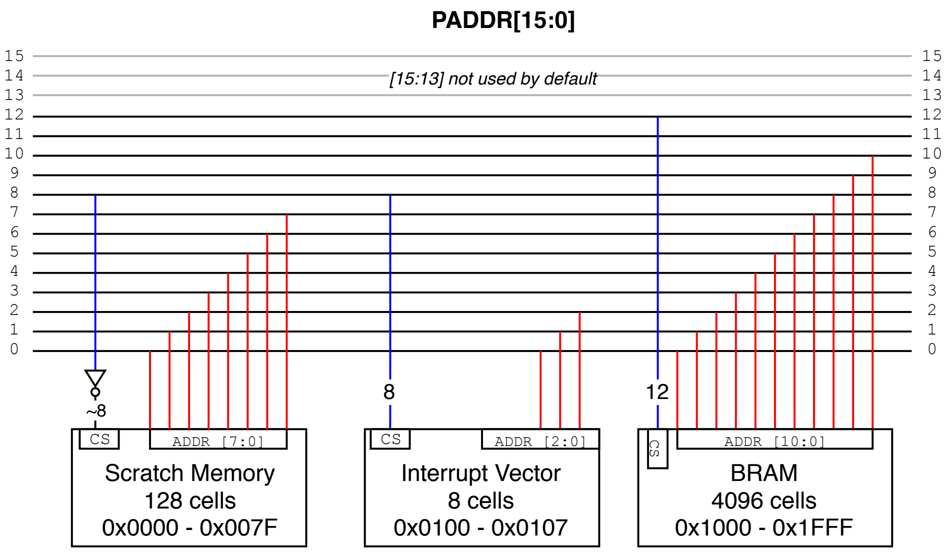


Figure 6.4: Partial address decoding used by the Vmicro16 SoC design. Each peripheral shown only needs to decode a signal bit to determine if it is enabled.

6.3 Memory Map

The system-on-chip's memory map is shown below in Figure 6.5. The addresses for each peripheral have been carefully chosen for both:

- Easy software access – creating addresses via software requires few instructions (normally one to four `MOVI` and `LSHIFT` instructions to address `0x0000` to `0xffff`), which increases software performance.
- and Reducing address decoding logic – most addresses can be decoded using partial decoding techniques.

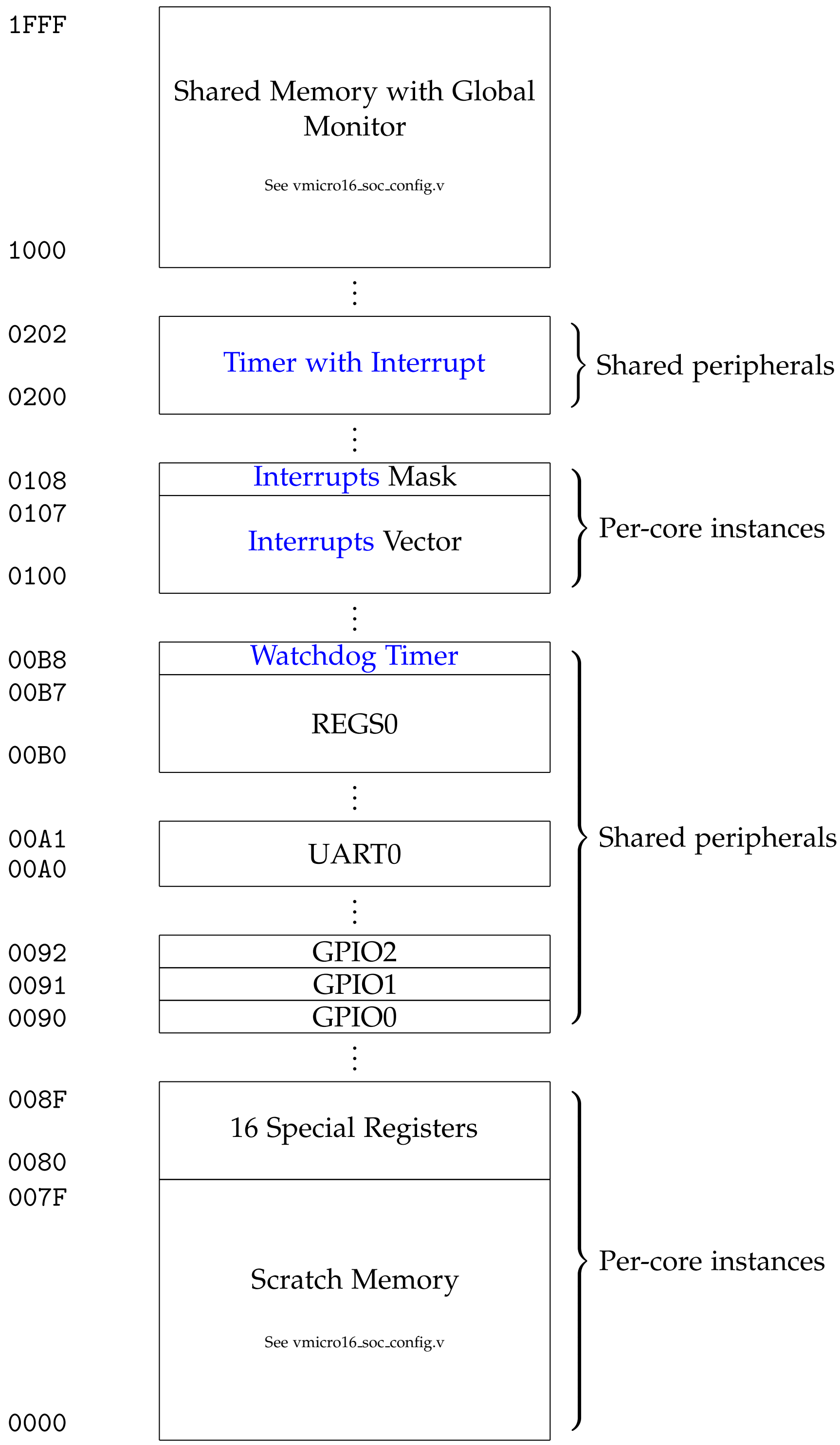


Figure 6.5: Memory map showing addresses of various memory sections.

Chapter 7

Multi-core Communication

So far we have discussed the features and design of the Vmicro16 system-on-chip. This section will discuss the multi-processing functionality and how to use it.

7.1 Introduction

Multi-processing functionality is the primary deliverable of this project.

7.1.1 Design Goals

- **Support common synchronisation primitives.**
  Software should be able to implement common synchronisation primitives, such as mutexes, semaphores, and memory barriers, to perform atomic operations and avoid race conditions, which are critical in parallel and concurrent software applications.

- **Context identification.**
  The SoC should expose configuration information such as: the number of processing cores, amount of shared and scratch memory, and the `CORE_ID`, to each thread.

7.1.2 Context Identification

A goal of the multi-processing functionality of this project is allow software written for it to be run on any number of cores. This means that a software program will scale to use all cores in the SoC without needing to rewrite the software. To enable this functionality, the software must be able to read contextual information about the SoC, such as the number of cores, how much global and scratch memory is available, and what the `CORE_ID` of the current core is.

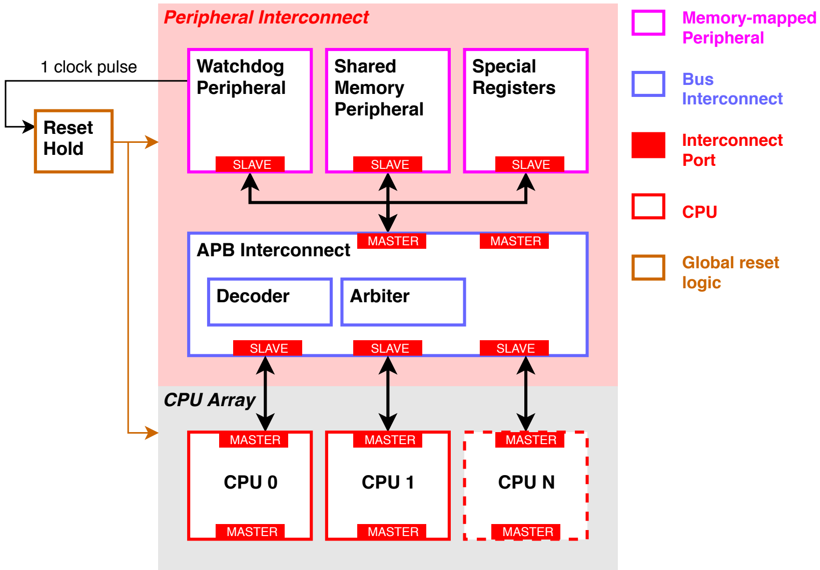


Figure 7.1: Block digram showing the main multi-processing components: the CPU array and a peripheral interconnect used for core synchronisation.

This information is provided through the Special Registers peripheral (0x0080 - 0x008F), shown in Figure 7.1. This register set provides relevant information for writing software that can dynamically scale for various SoC configurations.

15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
	CORE_ID	0080 R
	NUM_CORES	0081 R
SHARED_MEMORY cells (default 4096)		0082 R
	NUM_PERIPHERALS	0083 R
SCRATCH_MEMORY cells (default 64)		0084 RW
User defined		0085 RW
⋮		
User defined		008F RW

Figure 7.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

7.1.3 Thread Synchronisation

In multi-threaded software it is important

The mutex functionality is implemented using a similar scheme to that of ARM's *Global Monitor* [?].

Mutexes

In software, a mutex is an object used to control access to a shared resource. The term *object* is used as it's implementation is normally platform dependant, meaning that the processor may provide a hardware mechanism or is left for the operating system to provide.

In this project, mutexes are provided by the processor through the Shared Memory Peripheral (0x1000 to 0x1FFF) which provides a large RAM-style memory accessible by all cores through the peripheral interconnect bus. This large memory is explicitly defined to use the FPGA's BRAM blocks using Xilinx's Verilog `ram_style="block"` attribute to avoid wasting LUTs when using high core counts. The peripheral allows each memory cell to be *locked*, meaning that only the cell owner can modify it's contents. This is implemented by using another large memory, `locks`, to store the `CORE_ID` + 1 of the owner, as shown in Listing 5. In this system, a lock containing the value 0 indicates an unlocked cell. As `CORE_ID`s are indexed from zero, 1 is arithmetically added to each cell. For example, if core #2 wants to lock a memory cell, the value 3 is written to the lock.

```
reg [15:0]          ram    [0:8191]; // 16KB large RAM memory
reg [clog2(CORES):0] locks  [0:8181]; // memory cell owner
```

Listing 5: RAM and lock memories instantiated by the shared memory peripheral.

To lock and unlock cells, the instructions `LWEX` and `SWEX` instructions are used. These instructions are similar to the `LW`/`SW` instructions but provide locking functionality. The *EX* in the instruction names indicate *exclusive access*. `LWEX` is used to read memory contents (like `LW`) and also lock the cell if not already locked. If a core attempts to lock an already locked cell, the lock does not change. Unlocking is done by the `SWEX` instruction, which conditionally writes to the memory cell if it is locked by the same core. Unlike `SW`, `SWEX` returns a zero for success and one for failure if it is locked by another core.

Figure 7.3 shows a simple assembly function to lock a memory cell.

```
lock_mutex:
        // attempt lock
        lwex r0, r1
        // check success
        swex r0, r1
        cmp  r0, r3
        // if not equal (NE), retry
        movi r4, lock_mutex
        br   r4, BR_NE
critical:
    // core has the mutex
```

Figure 7.3: Assembly code for locking a mutex. r1 is the address to lock. r3 is zero. r4 is the branch address.

Barriers

Barriers are a useful software sequence used to block execution until all other threads (or a subset) have reached the same point. Barriers are often used for broadcast and gather actions (sending values to each core or receiving them). They are also used to synchronise program execution if some threads have more work to do than others.

The Vmicro16 processor provides barrier synchronisation through the Shared Memory Peripheral. Like the mutex code, the barrier code uses the `LWEX` and `SWEX` instructions to lock a memory cell. Instead of immediately checking the lock as an abstract object, the barrier code treats the cell as a normal memory cell containing a numeric value. Listing 6 shows a software example of this. When the `barrier_reached` code is reached, the code will increment the shared memory value by 1, indicating that the number of threads that have reached this

```
barrier_reached:
    // load latest count
    lwex    r0, r5
    // try increment count
    // increment by 1
    addi    r0, r3 + #0x01
    // attempt store
    swex    r0, r5

    // check success (== 0)
    cmp     r0, r3
    // branch if failed
    movi    r4, barrier_reached
    br      r4, BR_NE

barrier_wait:
    // load the count
    lw      r0, r5
    // compare with number of threads
    cmp     r0, r7
    // jump back to barrier if not equal
    movi    r4, barrier_wait
    br      r4, BR_NE
```

Listing 6: Assembly code for a memory barrier. Threads will wait in the barrier_wait function until all other threads have reached that code point.

point has increased by one (r5). The `barrier_wait` function is then entered which waits until this numeric value (r5) is equal to the number of threads (r7) in the system. If this is true, then all threads have reached the `barrier_wait` function and can continue with normal program execution.

Chapter 8

Analysis & Results

So far the system's design, implementation, and example usage, has been presented and discussed.

8.1 Introduction

This chapter presents analytic information

8.2 Implementation Analysis

This section analysis the synthesised and implemented system-on-chip design to see the effect of increasing core counts.

8.2.1 Design Size

Constraints

As discussed in Chapter 4 Single-core Design, each processor core features two memories: instruction and scratch memory, which can both map onto synchronous, single-port, FPGA BRAM blocks. While this will reduce LUT requirements in designs with few cores, it becomes a non-trivial problem as the core counts increase. FPGAs have a fixed number of hard-BRAM blocks available for inference by the HDL compiler, for example the low-end Xilinx Spartan-6 XC6SLX9 FGPA features 32 18 Kb BRAM blocks [19, p. 2], and the Cyclone V 5CSEMA5F31C6N (used in the DE1-SoC) has 397 10 Kb blocks [20, p. 22].

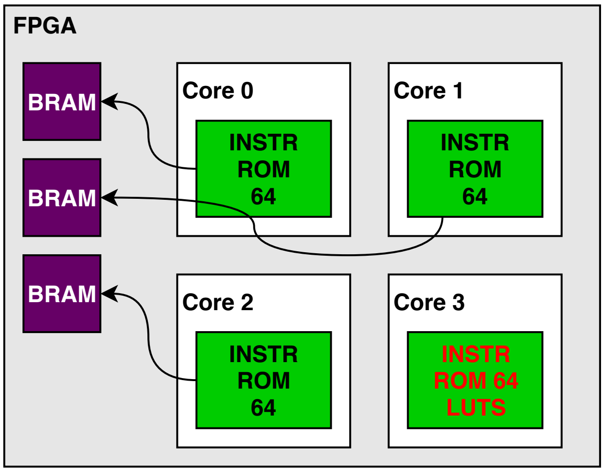


Figure 8.1: A theoretical FPGA device with 3 BRAM blocks running a 4-core design. Each core can map onto a BRAM block, however as there are more cores than BRAM blocks available, some core memories will be implemented as distributed RAM, or in the worse case using ALMs.

As shown in Figure 8.1, as the number of processor cores increasing they eventually outnumber the available BRAM blocks, resulting in their memories being implemented in either distributed RAMs or ALMs, both of which can consume significant logic resources of the FPGA which reduces the maximum possible core count.

8.3 Scenario Performance

To evaluate the performance of the system-on-chip, scenarios encompassing computational problems that are reflective of real-world applications are compiled and ran on the design.

8.3.1 Scenario Overview

The scenario is a software program that runs a parallel implementation of the summation function, i.e. `sum [1..10]` which returns 55. While this may seem too simple at first to measure performance of a multi-core system-on-chip, the function is actually quite appropriate as it encompasses various parallel problems, such as: a fixed time/size serial part; broadcasting of the data set (in this case the range of the summation); thread synchronisation (to know when the data is ready and to schedule gathering of intermediary results); and is highly scalable.

The summation task flow is as follows:

1. Root (core #0) broadcasts the range of the summation (i.e. sum 1 to 10) to all cores via the global shared memory.

2. Non-root cores wait for this broadcast to finish (memory barrier), then calculate their own subset of the range to sum. For example, if Root broadcasts that there are 240 samples and 10 cores in the system, each core calculates the subset size:

$$240/10 = 24 \tag{8.1}$$

calculations starting from:

$$ID_{CORE} * 24 \tag{8.2}$$

For example, Core #5 will start its 24 sample subset summation from

$$5 * 24 = 120 \tag{8.3}$$

effectively performing `sum [120..123]`.

3. All cores perform an intermediary summation over their subset of the range (serial part).
4. All cores attempt to add their intermediary result to a global sum value in global shared memory (mutex).
5. All cores halt, signalling that their work has been committed to the global shared memory and have finished the program.

This program is written in assembly in the file `sw/demos/asm/sum64.s` and can be compiled using the assembly compiler (developed for deliverable **ED4**) using the command below. The assembly compiler outputs the file `asm.s.hex` containing hex instruction words for use in Verilog's `$readmemh` function. This data is used for each core's instruction memory. The assembly program is also shown in Section D.2.

```
python sw/asm.py sw/demos/asm/sum64.s
```

8.3.2 Performance Measurements

Behavioural simulation will be used to measure the following metrics to estimate general performance of the system-on-chip:

- Total program run-time.
  This is the time from when the reset signal is de-asserted to when all cores have halted. Each core has an output `halt` signal which the SoC can use to determine if all cores have halted using `wire all_halted = &core_halts;`.
- Time spent on the serial part.
  The serial part of this scenario consists of the intermediary summation of it's subset range. As each core is performing this task, the average will be used.
- Time spent on communication.
  This includes time spent on thread synchronisation, i.e. waiting for the global memory to become available and waiting on the root to finish broadcast. Again, the average time will be used.
- Time spent fetching instructions.
  Instruction fetches occur during stage `STAGE_IF` of the pipeline. The behavioural test bench will record the number of clock cycles each core spends in this state, then calculate the average time spent fetching instructions.

8.3.3 Performance Results

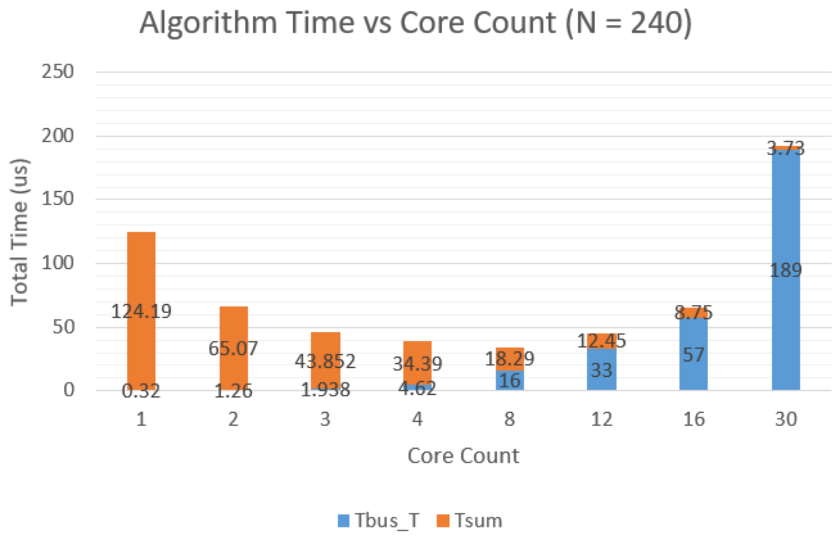


Figure 8.2: Chart showing how the communication times (Tbus) and serial times (Tsum) changes with core count.

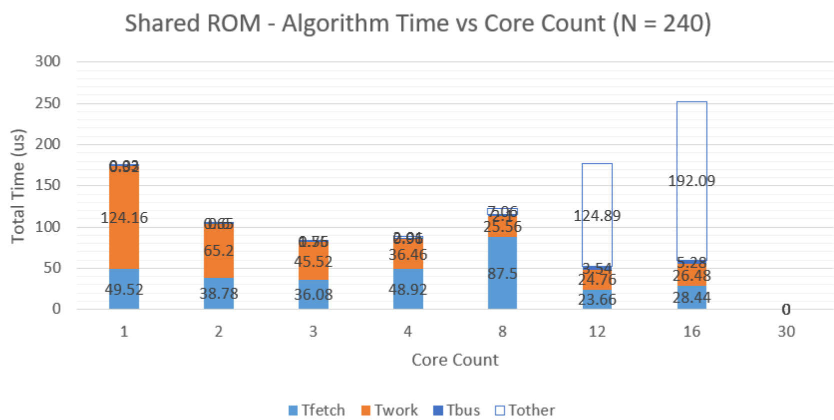


Figure 8.3: Similar to Figure 8.2 but using shared instruction memory to reduce block memory requirements per core.

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] M. Weber, "Arbiters: Design Ideas and Coding Styles," p. 2.

[14] A. Palchaudhuri and R. S. Chakraborty, *High Performance Integer Arithmetic Circuit Design on FPGA: Architecture, Implementation and Design Automation*. Springer, 2015, vol. 51.

[15] V. Salauyou and M. Gruszewski, "Designing of hierarchical structures for binary comparators on fpga/soc," in *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 2015, pp. 386–396.

[16] Xilinx, *Spartan-6 FPGA Configurable Logic Block - User Guide - UG384*, Xilinx.

[17] Altera, *Cyclone V Device Handbook - Device Interfaces and Integration - CV-5V2*, Altera.

[18] A. S. Tanenbaum, *Structured Computer Organization*. Pearson Education India, 2016.

[19] Xilinx, *Spartan-6 Family Overview - DS160*, Xilinx.

[20] Intel, *Cyclone V Device Overview - CV-51001*, Intel.

Appendix A

Peripheral Information

To provide user's with useful functionality, common system-on-chip peripherals were created. This section describes each peripheral and it's design decisions. The full memory-map is shown in Figure 6.5.

A.1 Special Registers

From the software perspective, it is important for both the developer and software algorithms to know the target system's architecture to better utilise the resources available to them. Software written for one architecture with N cores must also run on an architecture with M cores. To enable such portability, the software must query the system for information such as: number of processor cores and the current core identifier. Without this information, the developer would be required to produce software for each individual architecture (e.g. an Intel i5 with 4 cores or an Intel i7 with 8 cores, or an NVIDIA GTX 970 with 1664 CUDA cores.

The special register peripheral is shown below.

15–8	7–0	
	CORE_ID	0080 R
	NUM_CORES	0081 R
SHARED_MEMORY cells (default 4096)		0082 R
	NUM_PERIPHERALS	0083 R
SCRATCH_MEMORY cells (default 64)		0084 RW
User defined		0085 RW
⋮		
User defined		008F RW

Figure A.1: Vmicro16 Special Registers layout (0x0080 - 0x008F).

A.2 Watchdog Timer

In any multi-threaded system there exists the possibility for a deadlock – a state where all threads are in a waiting state – and algorithm execution is forever blocked. This can occur either by poor software programming or incorrect thread arbitration by the processor. A common method of detecting a deadlock is to make each thread signal that it is not blocked by resetting a countdown timer. If the countdown timer is not reset, it will eventually reach zero and it is assumed that all threads are blocked as none have reset the countdown.

In this system-on-chip design, software can reset the watchdog timer by writing any 16-bit value to the address `0x00B8`.

This peripheral is optional and can be enabled using the configuration parameters described in Configuration Options.

15–0	
Reset Watchdog	00B8 W

A.3 GPIO Interface

15–8	7–0	
	GPIO0 Output	0090 RW
GPIO1 Output		0091 RW
	GPIO2 Output	0092 RW
	GPIO3 Input	0093 R

On the DE1-SoC board, GPIO0 is assigned to the LEDs, and GPIO1 and GPIO2 to the 6 seven-segment displays.

A.4 Timer with Interrupt

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td colspan="16">Load Value</td><td>0200 RW</td></tr>
<tr><td colspan="13"></td><td>I</td><td>R</td><td>S</td><td>0201 W</td></tr>
<tr><td colspan="16">Prescaler</td><td>0202 W</td></tr>
</table>

Clock Frequency Uses top level FPGA clock (normally 50 MHz).

Load Value Value to count down from each clock.

I Interrupt enable bit. Default 0.

R Reset Load Value and Prescaler values to their last written value.

S Start the timer countdown. 1 = start. 0 = stop.

Prescaler Number of clocks per FPGA clock to wait between each decrement.

A.5 UART Interface

<table>
<tr><td>15</td><td>8</td><td>7</td><td>1</td><td>0</td><td></td></tr>
<tr><td colspan="2"></td><td colspan="3">Transmit Data</td><td>00A0 W</td></tr>
<tr><td colspan="2"></td><td colspan="3">Receive Data</td><td>00A1 R</td></tr>
<tr><td colspan="4"></td><td>E</td><td>I</td><td>00A2 RW</td></tr>
</table>

E Enable the UART component.

I Enable an interrupt upon receiving new data. Default 1.

> Note: If `DEF_USE_REPROG` is enabled in `vmicro16_soc_config.v` then the receiver port will be reserved for programming the instruction memory, resulting in reads and writes to addresses 0x00A1 and 0x00A2 to return 0.

Appendix B

Additional Figures

```
input       [MASTER_PORTS*BUS_WIDTH-1:0]   S_PADDR,
input       [MASTER_PORTS-1:0]             S_PWRITE,
input       [MASTER_PORTS-1:0]             S_PSELx,
input       [MASTER_PORTS-1:0]             S_PENABLE,
input       [MASTER_PORTS*DATA_WIDTH-1:0]  S_PWDATA,
output reg  [MASTER_PORTS*DATA_WIDTH-1:0]  S_PRDATA,
output reg  [MASTER_PORTS-1:0]             S_PREADY,
```

Listing 7: Variable size inputs and outputs to the interconnect.

B.1 Register Set Multiplex

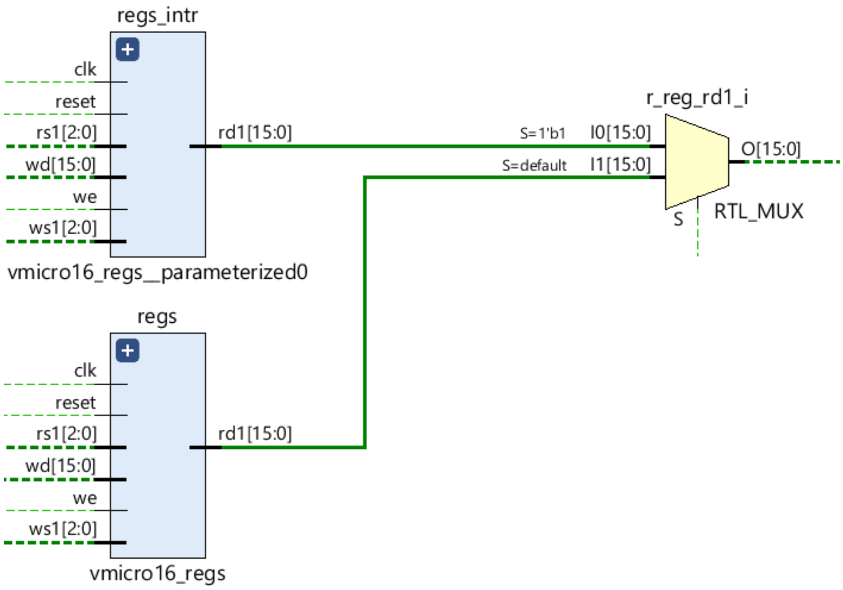


Figure B.1: Normal mode (bottom) and interrupt mode (top) register sets are multiplexed to switch between contexts.

B.2 Instruction Set Architecture

	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
SPCL	**00000**	**11 bits**			**NOP**
SPCL	00000	11h'000			NOP
SPCL	00000	11h'001			HALT
SPCL	00000	11h'002			Return from interrupt
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	Imm8		Rd <= (Imm8 _f_ SZO) ? 1 : 0
MULT	01011	Rd	Ra	X	Rd <= uRd * uRa
HALT	01100	X			
LWEX	01101	Rd	Ra	s5	Rd <= RAM[Ra+s5]
SWEX	01110	Rd	Ra	s5	RAM[Ra+s5] <= Rd Rd <= 0|1 if success

Figure B.2: Vmicro16 instruction set architecture.

Appendix C

Configuration Options

The following configuration options are defined in `vmicro16_soc_config.v`.

> Defaults with empty/blank values signifies that the preprocessor define is commented out/not defined/disabled by default/computed by other parameters.

C.1 System-on-chip Configuration Options

Macro	Default	Purpose
CORES	4	Number of CPU cores in the SoC
SLAVES	8	Number of peripherals
DEF_USE_WATCHDOG	//	Enable watchdog module to recover from deadlocks and infinite loops
DEF_GLOBAL_RESET	//	Enable synchronous reset logic. Will consume more LUT resources. Does not reset BRAM blocks.

Table C.1: SoC Configuration Options

C.2 Core Options

Macro	Default	Purpose
DATA_WIDTH	16	Width of CPU registers in bits
DEF_CORE_HAS_INSTR_MEM	//	Enable a per core instruction memory cache
DEF_MEM_INSTR_DEPTH	64	Instruction memory cache per core
DEF_MEM_SCRATCH_DEPTH	64	RW RAM per core
DEF_ALU_HW_MULT	1	Enable/disable HW multiply (1 clock)
FIX_T3	//	Enable a T3 state for the APB transaction
DEF_USE_REPROG	//	Programme instruction memory via UART0. Requires DEF_GLOBAL_RESET. Enabling this will reserve the UART0 RX port for exclusive use for programming the instruction memory. Software reads of UART0 RX will return 0.

Table C.2: Core Options

C.3 Peripheral Options

Macro	Default	Purpose
APB_WIDTH		AMBA APB PADDR signal width
APB_PSELX_GPIO0	0	GPIO0 index
APB_PSELX_UART0	1	UART0 index
APB_PSELX_REGS0	2	REGS0 index
APB_PSELX_BRAM0	3	BRAM0 index
APB_PSELX_GPIO1	4	GPIO1 index
APB_PSELX_GPIO2	5	GPIO2 index
APB_PSELX_TIMR0	6	TIMR0 index
APB_BRAM0_CELLS	4096	Shared memory words
DEF_MMU_TIM0_S	16'h0000	Per core scratch memory start/end address
DEF_MMU_TIM0_E	16'h007F	″
DEF_MMU_SREG_S	16'h0080	Per core special registers start/end address
DEF_MMU_SREG_E	16'h008F	″
DEF_MMU_GPIO0_S	16'h0090	Shared GPIOn start/end address
DEF_MMU_GPIO0_E	16'h0090	″
DEF_MMU_GPIO1_S	16'h0091	″
DEF_MMU_GPIO1_E	16'h0091	″
DEF_MMU_GPIO2_S	16'h0092	″
DEF_MMU_GPIO2_E	16'h0092	″
DEF_MMU_UART0_S	16'h00A0	Shared UART start/end address
DEF_MMU_UART0_E	16'h00A1	″
DEF_MMU_REGS0_S	16'h00B0	Shared registers start/end address
DEF_MMU_REGS0_E	16'h00B7	″
DEF_MMU_BRAM0_S	16'h1000	Shared memory with global monitor start/end address
DEF_MMU_BRAM0_E	16'h1FFF	″
DEF_MMU_TIMR0_S	16'h0200	Shared timer peripheral start/end address
DEF_MMU_TIMR0_E	16'h0202	″

Table C.3: Peripheral Options

Appendix D

Viva Demonstration Examples

D.1 2-core Timer Interrupt and ISR

This example demo, shown during the viva, blinks an LED every 0.5 seconds via a timer interrupt. Core 0 sets up the interrupt vector (by writing the isr0 function address to the interrupt vector) and enables all interrupt sources. Core 1 sets up the timer interval peripheral to produce an interrupt every 0.5 seconds. Core 1 also performs the interrupt handler (isr0): toggle an LED, write the state to UART0, and resets the watchdog.

```
// interrupts.s
//   Toggle LED in ISR

entry:
    // get core idx 0x80 in r7
    movi    r7, #0x80
    lw      r7, r7

    // core1 sets up the timer
    // Core0 enables interrupts and performs the isr
    cmp     r7, r0
    movi    r0, timer
    br      r0, BR_NE

    // Set interrupt vector (0)
    movi    r0, isr0
    movi    r1, #0x1
    movi    r2, #0x08
    lshft   r1, r2
    sw      r0, r1

    // enable all interrupts
    movi    r0, #0x0f
    sw      r0, r1 + #0x8

    // enter idle state
    halt    r0, r0

timer:
    // set timr0 address 0x200 into r0
    // shift left 8 places
    movi    r0, #0x01
    movi    r1, #0x09
    lshft   r0, r1

    // Set load value
    //movi    r1, #0x31
    //sw      r1, r0
    // test we the expected value back
    //lw      r2, r0

    // set load = 0x3000
```

```
    movi    r1, #0x3
    movi    r2, #0x0C
    //movi    r2, #0x04
    lshft   r1, r2
    sw      r1, r0

    // Set prescale value to 0x1000
    // 20ns * load * prescaler = nanosecond delay
    // 20ns * 10000 * 5000     = 1.0s
    // 20.0 * 0x3000 * 0x1000  = ~1.0s
    movi    r1, #0x1
    // 1.0 second
    //movi    r2, #0x0C
    // 0.5 second
    movi    r2, #0x0B
    // 0.25 second
    //movi    r2, #0x0a
    // 0.0625 second
    //movi    r2, #0x04
    lshft   r1, r2
    sw      r1, r0 + #0x02

    // Start the timer (write 0x0001 to 0x0101)
    movi    r1, #0x01
    sw      r1, r0 + #0x01

exit:
    // enter idle state
    halt    r0, r0

isr0:
    movi    r0, #0x90
    lw      r1, r0
    // xor with 1
    movi    r2, #0x1
    xor     r1, r2
    // write back
    sw      r1, r0

    // write ascii value to uart0
    movi    r0, #0xa0
    movi    r2, #0x30
    add     r1, r2
    sw      r1, r0

    // reset watchdog
    movi    r0, #0xb8
    sw      r1, r0

    // return from interrupt
    intr    r0, r0
```

D.2 1-160 Core Parallel Summation

This example demo performs a parallel summation of numbers 1 to 320. The algorithm *assigns* each core a subset of the summation space. It does this using the core's ID and the number of cores in the system. The following formulas determine where the subset begins and ends for each core. Core 0 broadcasts the number to sum to then each core calculates its subset start and end positions. Each core then performs a summation over it's subset then adds the result to a global shared value. After pushes it's results, the global shared value will contain the final summation result.

$$N_{samples} = 320 \tag{D.1}$$

$$N_{threads} = 64 \tag{D.2}$$

$$subset = N_{samples} / N_{threads} \tag{D.3}$$

$$start = ID * subset \tag{D.4}$$

$$end = start + subset \tag{D.5}$$

```
// sum64.s
// Simple 1-160 core summation program

// Set up common values, such as: Core id (r6),
//   number of threads (cores) (r7), shared memory addresses (r5)
entry:
    // Core id in r6
    movi    r0, #0x80
    lw      r0, r0
    // store in r6
    mov     r6, r0

    // get number of threads
    movi    r0, #0x81
    lw      r0, r0
    // store in r7
    mov     r7, r0

    // BRAM0 shared memory 0x1000
    movi    r5, #0x01
    movi    r2, #0x0C
    lshft   r5, r2

jmp_to_barrier:
    // NOT_ROOT
    //   wait at barrier
    cmp     r6, r3
    movi    r4, barrier_arrive
    br      r4, BR_NE

    // ROOT
    //   calculates nsamples_per_thread
    //       ns = 100
    //       nst = ns / (num_threads)
    //       nst = ns >> (num_threads - 1)
    //       r0 = (num_threads -1) WRONG!!!

root_broadcast:
    // The root (core idx 0) broadcasts the number of samples
    // 16 cores
    //movi    r4, #0x14
    // 32 cores
    //movi    r4, #0x0a
    // 64 cores
    movi    r4, #0x05
    // 80 cores
    //movi    r4, #0x04
    // 160 cores
    //movi    r4, #0x02

    // ROOT
    // Do the broadcast
    //   write nsamples_per_thread to shared bram (broadcast)
    //   0x1001
    sw      r4, r5 + #0x01

```

```
// Reach the barrier to tell everone
// that we have arrived
barrier_arrive:
    // load latest count
    lwex    r0, r5
    // try increment count
    // increment by 1
    addi    r0, r3 + #0x01
    // attempt store
    swex    r0, r5
    // check success (== 0)
    cmp     r0, r3
    // branch if failed
    movi    r4, barrier_arrive
    br      r4, BR_NE

// Wait in an infinite loop
// for all cores to 'arrive'
barrier:
    // load the count
    lw      r0, r5
    // compare with number of threads
    cmp     r0, r7
    // jump back to barrier if not equal
    movi    r4, barrier
    br      r4, BR_NE

// EACH CORE
// All cores have arrived and in sync
synced1:
    // Retrieve load the nsamples_per_thread
    lw      r4, r5 + #0x01
    // Calculate nstart = idx * nsamples_per_thread
    //   in r2
    mov     r2, r6
    mult    r2, r4

    // Loop limit in r4
    // samples_per_thread -> samples_per_thread + nstart
    add     r4, r2

// Perform the summation in a tight for loop
//   Sum numbers from nstart to limit
sum_loop:
    // sum += i
    add     r1, r2
    // increment i
    addi    r2, r3 + #0x01
    // check end
    cmp     r2, r4
    movi    r0, sum_loop
    br      r0, BR_NE

// Summation of the subset finished, result is in r1
//   Now use a mutex to add it to the global sum value in shared mem
sum_mutex:
    // load latest count
    lwex    r0, r5 + #0x2
    // try increment count
    // increment by 1
    add     r0, r1
    // make copy as swex has a return value
    mov     r2, r0
    // attempt store
    swex    r0, r5 + #0x02
    // check success (== 0)
    cmp     r0, r3
    // branch if failed
    movi    r4, sum_mutex
    br      r4, BR_NE

// Write the latest global sum value to gpio1
write_gpio:
    movi    r3, #0x91
    sw      r2, r3

// Write the latest global sum value to uart0 tx
write_uart_done:
    movi    r3, #0xa0
    movi    r2, #0x30
    add     r2, r6
    sw      r2, r3

// This core has finished
// Enter a low power state
exit:
    halt    r0, r0
```

Appendix E

Code Listing

E.1 SoC Code Listing

E.1.1 vmicro16_soc_config.v

Configuration file for configuring the vmicro16_soc.v and vmicro16.v features.

```
// Configuration defines for the vmicro16_soc and vmicro16 cpu.

`ifndef VMICRO16_SOC_CONFIG_H
`define VMICRO16_SOC_CONFIG_H

`include "clog2.v"

`define FORMAL

`define CORES           8
`define SLAVES          8

////////////////////////////////////////////////////////////
// Core parameters
////////////////////////////////////////////////////////////
// Per core instruction memory
//  Set this to give each core its own instruction memory cache
`define DEF_CORE_HAS_INSTR_MEM

// Top level data width for registers, memory cells, bus widths
`define DATA_WIDTH      16

// Set this to use a workaround for the MMU's APB T2 clock
//`define FIX_T3

// Instruction memory (read only)
//    Must be large enough to support software program.
`ifdef DEF_CORE_HAS_INSTR_MEM
    // 64 16-bit words per core
    `define DEF_MEM_INSTR_DEPTH 64
`else
    // 4096 16-bit words global
    `define DEF_MEM_INSTR_DEPTH 4096
`endif

// Scratch memory (read/write) on each core.
//    See `DEF_MMU_TIM0_* defines for info.
`define DEF_MEM_SCRATCH_DEPTH 64

```

```
// Enables hardware multiplier and mult rr instruction
`define DEF_ALU_HW_MULT 1

// Enables global reset (requires more luts)
//`define DEF_GLOBAL_RESET

// Enable a watch dog timer to reset the soc if threadlocked
//`define DEF_USE_WATCHDOG

// Enables instruction memory programming via UART0
//`define DEF_USE_REPROG

`ifdef DEF_USE_REPROG
    `ifndef DEF_GLOBAL_RESET
        `error_DEF_USE_REPROG_requires_DEF_GLOBAL_RESET
    `endif
`endif

////////////////////////////////////////////////////////////
// Memory mapping
////////////////////////////////////////////////////////////
`define APB_WIDTH       (2 + `clog2(`CORES) + `DATA_WIDTH)

`define APB_PSELX_GPIO0 0
`define APB_PSELX_UART0 1
`define APB_PSELX_REGS0 2
`define APB_PSELX_BRAM0 3
`define APB_PSELX_GPIO1 4
`define APB_PSELX_GPIO2 5
`define APB_PSELX_TIMR0 6
`define APB_PSELX_WDOG0 7

`define APB_GPIO0_PINS  8
`define APB_GPIO1_PINS  16
`define APB_GPIO2_PINS  8

// Shared memory words
`define APB_BRAM0_CELLS 4096

////////////////////////////////////////////////////////////
// Memory mapping
////////////////////////////////////////////////////////////
// TIM0
// Number of scratch memory cells per core
`define DEF_MMU_TIM0_CELLS  64
`define DEF_MMU_TIM0_S      16'h0000
`define DEF_MMU_TIM0_E      16'h007F
// SREG
`define DEF_MMU_SREG_S      16'h0080
`define DEF_MMU_SREG_E      16'h008F
// GPIO0
`define DEF_MMU_GPIO0_S     16'h0090
`define DEF_MMU_GPIO0_E     16'h0090
// GPIO1
`define DEF_MMU_GPIO1_S     16'h0091
`define DEF_MMU_GPIO1_E     16'h0091
// GPIO2
`define DEF_MMU_GPIO2_S     16'h0092
`define DEF_MMU_GPIO2_E     16'h0092
// UART0
`define DEF_MMU_UART0_S     16'h00A0
`define DEF_MMU_UART0_E     16'h00A1
// REGS0
`define DEF_MMU_REGS0_S     16'h00B0
`define DEF_MMU_REGS0_E     16'h00B7
// WDOG0
`define DEF_MMU_WDOG0_S     16'h00B8
`define DEF_MMU_WDOG0_E     16'h00B8
// BRAM0
`define DEF_MMU_BRAM0_S     16'h1000
`define DEF_MMU_BRAM0_E     16'h1fff
// TIMR0
`define DEF_MMU_TIMR0_S     16'h0200
`define DEF_MMU_TIMR0_E     16'h0202

////////////////////////////////////////////////////////////
// Interrupts
////////////////////////////////////////////////////////////
// Enable/disable interrupts
//   Disabling will free up resources for other features
`define DEF_ENABLE_INT
// Number of interrupt in signals
`define DEF_NUM_INT     8
// Default interrupt bitmask (0 = hidden, 1 = enabled)
`define DEF_INT_MASK    0
// Bit position of the TIMR0 interrupt signal
`define DEF_INT_TIMR0   0
// Interrupt vector memory location
`define DEF_MMU_INTSV_S     16'h0100
`define DEF_MMU_INTSV_E     16'h0107
```

```
// Interrupt vector memory location
`define DEF_MMU_INTSM_S      16'h0108
`define DEF_MMU_INTSM_E      16'h0108


`endif
```

E.1.2 top_ms.v

Top level module that connects the SoC design to hardware pins on the FPGA.

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

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


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

    // 3v3 input from the s6 on the de1soc
    input           S6_3v3,

    // SSDs
    output [6:0] ssd0,
    output [6:0] ssd1,
    output [6:0] ssd2,
    output [6:0] ssd3,
    output [6:0] ssd4,
    output [6:0] ssd5
);
    //wire [15:0]          M_PADDR;
    //wire                 M_PWRITE;
    //wire [5-1:0]         M_PSELx;   // not shared
    //wire                 M_PENABLE;
    //wire [15:0]          M_PWDATA;
    //wire [15:0]          M_PRDATA; // input to intercon
    //wire                 M_PREADY; // input to intercon

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

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

        //.M_PADDR    (M_PADDR),
        //.M_PWRITE   (M_PWRITE),
```

```
        //.M_PSELx    (M_PSELx),
        //.M_PENABLE  (M_PENABLE),
        //.M_PWDATA   (M_PWDATA),
        //.M_PRDATA   (M_PRDATA),
        //.M_PREADY   (M_PREADY),

        // UART
        .uart_tx (TXD),
        .uart_rx (RXD),

        // GPIO
        .gpio0   (LEDS[3:0]),
        .gpio1   (gpio1),
        .gpio2   (gpio2),

        // DBUG
        .dbug0   (LEDS[4])
        //.dbug1    (LEDS[7:4])
    );

    assign LEDS[7:5] = {TXD, RXD, S6_3v3};

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

endmodule
```

E.1.3 vmicro16_soc.v

```
//
//

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

module pow_reset # (
    parameter INIT  = 1,
    parameter N     = 8
) (
    input       clk,
    input       reset,
    output reg  resethold
);
    initial resethold = INIT ? (N-1) : 0;

    always @(*)
        resethold = |hold;

    reg [`clog2(N)-1:0] hold = (N-1);
    always @(posedge clk)
        if (reset)
            hold <= N-1;
        else
            if (hold)
                hold <= hold - 1;
endmodule

// Vmicro16 multi-core SoC with various peripherals
// and interrupts
module vmicro16_soc (
    input clk,
    input reset,

    // UART0
    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                              halt,
```

```
    //
    output      [`CORES-1:0]         dbug0,
    output      [`CORES*8-1:0]       dbug1
);
    wire [`CORES-1:0] w_halt;
    assign halt = &w_halt;

    assign dbug0 = w_halt;

    // Watchdog reset pulse signal.
    //   Passed to pow_reset to generate a longer reset pulse
    wire wdreset;
    wire prog_prog;

    // soft register reset hold for brams and registers
    wire soft_reset;
    `ifdef DEF_GLOBAL_RESET
        pow_reset # (
            .INIT       (1),
            .N          (8)
        ) por_inst (
            .clk        (clk),
            `ifdef DEF_USE_WATCHDOG
            .reset      (reset | wdreset | prog_prog),
            `else
            .reset      (reset),
            `endif
            .resethold  (soft_reset)
        );
    `else
        assign soft_reset = 0;
    `endif

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

    // Master apb interfaces
     wire [`CORES*`APB_WIDTH-1:0]   w_PADDR;
     wire [`CORES-1:0]              w_PWRITE;
     wire [`CORES-1:0]              w_PSELx;
     wire [`CORES-1:0]              w_PENABLE;
     wire [`CORES*`DATA_WIDTH-1:0]  w_PWDATA;
     wire [`CORES*`DATA_WIDTH-1:0]  w_PRDATA;
     wire [`CORES-1:0]              w_PREADY;

    // Interrupts
`ifdef DEF_ENABLE_INT
    wire [`DEF_NUM_INT-1:0]              ints;
    wire [`DEF_NUM_INT*`DATA_WIDTH-1:0]  ints_data;
    assign ints[7:1] = 0;
    assign ints_data[`DEF_NUM_INT*`DATA_WIDTH-1:`DATA_WIDTH] =
                {`DEF_NUM_INT*(`DATA_WIDTH-1){1'b0}};
`endif

    apb_intercon_s # (
        .MASTER_PORTS   (`CORES),
        .SLAVE_PORTS    (`SLAVES),
        .BUS_WIDTH      (`APB_WIDTH),
        .DATA_WIDTH     (`DATA_WIDTH),
        .HAS_PSELX_ADDR (1)
    ) apb (
        .clk        (clk),
        .reset      (soft_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)
    );

`ifdef DEF_USE_WATCHDOG
    vmicro16_watchdog_apb # (
```

```
            .BUS_WIDTH  (`APB_WIDTH),
            .NAME       ("WDOG0")
        ) wdog0_apb (
            .clk        (clk),
            .reset      (),
            // apb slave to master interface
            .S_PADDR    (),
            .S_PWRITE   (M_PWRITE),
            .S_PSELx    (M_PSELx[`APB_PSELX_WDOG0]),
            .S_PENABLE  (M_PENABLE),
            .S_PWDATA   (),
            .S_PRDATA   (),
            .S_PREADY   (M_PREADY[`APB_PSELX_WDOG0]),

            .wdreset    (wdreset)
        );
  `endif

        vmicro16_gpio_apb # (
            .BUS_WIDTH  (`APB_WIDTH),
            .DATA_WIDTH (`DATA_WIDTH),
            .PORTS      (`APB_GPIO0_PINS),
            .NAME       ("GPIO0")
        ) gpio0_apb (
            .clk        (clk),
            .reset      (soft_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*`DATA_WIDTH +: `DATA_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
            .gpio       (gpio0)
        );

        // GPIO1 for Seven segment displays (16 pin)
        vmicro16_gpio_apb # (
            .BUS_WIDTH  (`APB_WIDTH),
            .DATA_WIDTH (`DATA_WIDTH),
            .PORTS      (`APB_GPIO1_PINS),
            .NAME       ("GPIO1")
        ) gpio1_apb (
            .clk        (clk),
            .reset      (soft_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*`DATA_WIDTH +: `DATA_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
            .gpio       (gpio1)
        );

        // GPIO2 for Seven segment displays (8 pin)
        vmicro16_gpio_apb # (
            .BUS_WIDTH  (`APB_WIDTH),
            .DATA_WIDTH (`DATA_WIDTH),
            .PORTS      (`APB_GPIO2_PINS),
            .NAME       ("GPIO2")
        ) gpio2_apb (
            .clk        (clk),
            .reset      (soft_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*`DATA_WIDTH +: `DATA_WIDTH]),
            .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
            .gpio       (gpio2)
        );

        apb_uart_tx # (
            .DATA_WIDTH (8),
            .ADDR_EXP   (4) //2^^4 = 16 FIFO words
        ) uart0_apb (
            .clk        (clk),
            .reset      (soft_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*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
        // uart wires
        .tx_wire    (uart_tx),
        .rx_wire    ()
    );

    timer_apb timr0 (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_TIMR0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_TIMR0])
        //
        `ifdef DEF_ENABLE_INT
        ,.out       (ints     [`DEF_INT_TIMR0]),
         .int_data  (ints_data[`DEF_INT_TIMR0*`DATA_WIDTH +: `DATA_WIDTH])
        `endif
    );

    // Shared register set for system-on-chip info
    // R0 = number of cores
    vmicro16_regs_apb # (
        .BUS_WIDTH          (`APB_WIDTH),
        .DATA_WIDTH         (`DATA_WIDTH),
        .CELL_DEPTH         (8),
        .PARAM_DEFAULTS_R0  (`CORES),
        .PARAM_DEFAULTS_R1  (`SLAVES)
    ) regs0_apb (
        .clk        (clk),
        .reset      (soft_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*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
    );

    vmicro16_bram_ex_apb # (
        .BUS_WIDTH     (`APB_WIDTH),
        .MEM_WIDTH     (`DATA_WIDTH),
        .MEM_DEPTH     (`APB_BRAM0_CELLS),
        .CORE_ID_BITS (`clog2(`CORES))
    ) bram_apb (
        .clk        (clk),
        .reset      (soft_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*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
    );

    // There must be atleast 1 core
    `static_assert(`CORES > 0)
    `static_assert(`DEF_MEM_INSTR_DEPTH > 0)
    `static_assert(`DEF_MMU_TIM0_CELLS > 0)


    // Single instruction memory
`ifndef DEF_CORE_HAS_INSTR_MEM
    // slave input/outputs from interconnect
    wire [`APB_WIDTH-1:0]          instr_M_PADDR;
    wire                           instr_M_PWRITE;
    wire [1-1:0]                   instr_M_PSELx;  // not shared
    wire                           instr_M_PENABLE;
    wire [`DATA_WIDTH-1:0]         instr_M_PWDATA;
    wire [1*`DATA_WIDTH-1:0]       instr_M_PRDATA; // slave response
    wire [1-1:0]                   instr_M_PREADY; // slave response

    // Master apb interfaces
    wire [`CORES*`APB_WIDTH-1:0]   instr_w_PADDR;
    wire [`CORES-1:0]              instr_w_PWRITE;
    wire [`CORES-1:0]              instr_w_PSELx;
    wire [`CORES-1:0]              instr_w_PENABLE;
    wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PWDATA;
    wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PRDATA;
    wire [`CORES-1:0]              instr_w_PREADY;
```

```

    `ifdef DEF_USE_REPROG
        wire [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] prog_addr;
        wire [`DATA_WIDTH-1:0] prog_data;
        wire prog_we;
        uart_prog rom_prog (
            .clk        (clk),
            .reset      (reset | wdreset),
            // input stream
            .uart_rx    (uart_rx),
            // programmer
            .addr       (prog_addr),
            .data       (prog_data),
            .we         (prog_we),
            .prog       (prog_prog)
        );
    `endif

    `ifdef DEF_USE_REPROG
        vmicro16_bram_prog_apb
    `else
        vmicro16_bram_apb
    `endif
    # (
        .BUS_WIDTH      (`APB_WIDTH),
        .MEM_WIDTH      (`DATA_WIDTH),
        .MEM_DEPTH      (`DEF_MEM_INSTR_DEPTH),
        .USE_INITS      (1),
        .NAME           ("INSTR_ROM_G")
    ) instr_rom_apb (
        .clk            (clk),
        .reset          (reset),
        .S_PADDR        (instr_M_PADDR),
        .S_PWRITE       (0),
        .S_PSELx        (instr_M_PSELx),
        .S_PENABLE      (instr_M_PENABLE),
        .S_PWDATA       (0),
        .S_PRDATA       (instr_M_PRDATA),
        .S_PREADY       (instr_M_PREADY)

        `ifdef DEF_USE_REPROG
            ,
            .addr      (prog_addr),
            .data      (prog_data),
            .we        (prog_we),
            .prog      (prog_prog)
        `endif
    );

    apb_intercon_s # (
        .MASTER_PORTS   (`CORES),
        .SLAVE_PORTS    (1),
        .BUS_WIDTH      (`APB_WIDTH),
        .DATA_WIDTH     (`DATA_WIDTH),
        .HAS_PSELX_ADDR (0)
    ) apb_instr_intercon (
        .clk        (clk),
        .reset      (soft_reset),
        // APB master from cores
        // master
        .S_PADDR    (instr_w_PADDR),
        .S_PWRITE   (instr_w_PWRITE),
        .S_PSELx    (instr_w_PSELx),
        .S_PENABLE  (instr_w_PENABLE),
        .S_PWDATA   (instr_w_PWDATA),
        .S_PRDATA   (instr_w_PRDATA),
        .S_PREADY   (instr_w_PREADY),
        // shared bus slaves
        // slave outputs
        .M_PADDR    (instr_M_PADDR),
        .M_PWRITE   (instr_M_PWRITE),
        .M_PSELx    (instr_M_PSELx),
        .M_PENABLE  (instr_M_PENABLE),
        .M_PWDATA   (instr_M_PWDATA),
        .M_PRDATA   (instr_M_PRDATA),
        .M_PREADY   (instr_M_PREADY)
    );
`endif

    genvar i;
    generate for(i = 0; i < `CORES; i = i + 1) begin : cores

        vmicro16_core # (
            .CORE_ID            (i),
            .DATA_WIDTH         (`DATA_WIDTH),

            .MEM_INSTR_DEPTH    (`DEF_MEM_INSTR_DEPTH),
            .MEM_SCRATCH_DEPTH  (`DEF_MMU_TIM0_CELLS)
        ) c1 (
```

```
            .clk        (clk),
            .reset      (soft_reset),

            // debug
            .halt       (w_halt[i]),

            // interrupts
            .ints       (ints),
            .ints_data  (ints_data),

            // Output master port 1
            .w_PADDR    (w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
            .w_PWRITE   (w_PWRITE  [i]                           ),
            .w_PSELx    (w_PSELx   [i]                           ),
            .w_PENABLE  (w_PENABLE [i]                           ),
            .w_PWDATA   (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w_PRDATA   (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w_PREADY   (w_PREADY  [i]                           )

`ifndef DEF_CORE_HAS_INSTR_MEM
            // APB instruction rom
            , // Output master port 2
            .w2_PADDR   (instr_w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
            //.w2_PWRITE  (instr_w_PWRITE  [i]                           ),
            .w2_PSELx   (instr_w_PSELx   [i]                           ),
            .w2_PENABLE (instr_w_PENABLE [i]                           ),
            //.w2_PWDATA  (instr_w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w2_PRDATA  (instr_w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w2_PREADY  (instr_w_PREADY  [i]                           )
`endif
        );
    end
    endgenerate


    ////////////////////////////////////////////////////
    // Formal Verification
    ////////////////////////////////////////////////////
    `ifdef FORMAL
    wire all_halted = &w_halt;
    ////////////////////////////////////////////////////
    // Count number of clocks each core is spending on
    //    bus transactions
    ////////////////////////////////////////////////////
    reg [15:0] bus_core_times       [0:`CORES-1]; // bus work
    reg [15:0] core_work_times      [0:`CORES-1]; // serial work
    reg [15:0] instr_fetch_times    [0:`CORES-1]; // instruction fetches
    integer i2;
    initial
        for(i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
            bus_core_times[i2] = 0;
            core_work_times[i2] = 0;
        end

    // total bus time
    generate
        genvar g2;
        for (g2 = 0; g2 < `CORES; g2 = g2 + 1) begin : formal_for_times
              always @(posedge clk) begin
                    if (w_PSELx[g2])
                         bus_core_times[g2] <= bus_core_times[g2] + 1;

                    // Core working time
                    `ifndef DEF_CORE_HAS_INSTR_MEM
                         if (!w_PSELx[g2] && !instr_w_PSELx[g2])
                    `else
                         if (!w_PSELx[g2])
                    `endif
                              if (!w_halt[g2])
                                    core_work_times[g2] <= core_work_times[g2] + 1;

              end
          end
    endgenerate

    reg [15:0] bus_time_average = 0;
    reg [15:0] bus_reqs_average = 0;
    reg [15:0] fetch_time_average = 0;
    reg [15:0] work_time_average = 0;
    //
    always @(all_halted) begin
        for (i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
            bus_time_average   = bus_time_average   + bus_core_times[i2];
            bus_reqs_average   = bus_reqs_average   + bus_core_reqs_count[i2];
            work_time_average  = work_time_average  + core_work_times[i2];
            fetch_time_average = fetch_time_average + instr_fetch_times[i2];
        end

        bus_time_average   = bus_time_average   / `CORES;
```

```
        bus_reqs_average   = bus_reqs_average   / `CORES;
        work_time_average  = work_time_average  / `CORES;
        fetch_time_average = fetch_time_average / `CORES;
    end

    ////////////////////////////////////////////////////
    // Count number of bus requests per core
    ////////////////////////////////////////////////////
    // 1 clock delay of w_PSELx
    reg [`CORES-1:0] bus_core_reqs_last;
    // rising edges of each
    wire [`CORES-1:0] bus_core_reqs_real;
    // storage for counters for each core
    reg [15:0] bus_core_reqs_count [0:`CORES-1];
    initial
        for(i2 = 0; i2 < `CORES; i2 = i2 + 1)
            bus_core_reqs_count[i2] = 0;

    // 1 clk delay to detect rising edge
    always @(posedge clk)
        bus_core_reqs_last <= w_PSELx;

    generate
        genvar g3;
              for (g3 = 0; g3 < `CORES; g3 = g3 + 1) begin : formal_for_reqs
              // Detect new reqs for each core
              assign bus_core_reqs_real[g3] = w_PSELx[g3] >
                                                          bus_core_reqs_last[g3];

              always @(posedge clk)
                    if (bus_core_reqs_real[g3])
                         bus_core_reqs_count[g3] <= bus_core_reqs_count[g3] + 1;

         end
    endgenerate


    `ifndef DEF_CORE_HAS_INSTR_MEM
        ////////////////////////////////////////////////////
        // Time waiting for instruction fetches
        //   from global  memory
        ////////////////////////////////////////////////////
        integer i3;
        initial
            for(i3 = 0; i3 < `CORES; i3 = i3 + 1)
                instr_fetch_times[i3] = 0;

        // total bus time
        // Instruction fetches occur on the w2 master port
        generate
            genvar g4;
            for (g4 = 0; g4 < `CORES; g4 = g4 + 1) begin : formal_for_fetch_times
                always @(posedge clk)
                    if (instr_w_PSELx[g4])
                        instr_fetch_times[g4] <= instr_fetch_times[g4] + 1;
            end
        endgenerate
    `endif


    `endif // end FORMAL

endmodule
```

E.1.4 vmicro16.v

Vmicro16 CPU core module.

```
// 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"



```

```
// 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
module vmicro16_bram # (
    parameter MEM_WIDTH     = 16,
    parameter MEM_DEPTH     = 64,
    parameter CORE_ID       = 0,
    parameter USE_INITS     = 0,
    parameter PARAM_DEFAULTS_R0 = 0,
    parameter PARAM_DEFAULTS_R1 = 0,
    parameter PARAM_DEFAULTS_R2 = 0,
    parameter PARAM_DEFAULTS_R3 = 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
    (* ram_style = "block" *)
    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;
        mem[0] = PARAM_DEFAULTS_R0;
        mem[1] = PARAM_DEFAULTS_R1;
        mem[2] = PARAM_DEFAULTS_R2;
        mem[3] = PARAM_DEFAULTS_R3;

        if (USE_INITS) begin
            //`define TEST_SW
            `ifdef TEST_SW
            $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\verilog_memh.txt", mem);
            `endif

            `define TEST_ASM
            `ifdef TEST_ASM
            $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\asm.s.hex", mem);
            `endif

            //`define TEST_COND
            `ifdef TEST_COND
            mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
            mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
            `endif

            //`define TEST_CMP
            `ifdef TEST_CMP
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0A};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h0B};
            mem[2] = {`VMICRO16_OP_CMP,     3'h1, 3'h0, 5'h1};
            `endif

            //`define TEST_LWEX
            `ifdef TEST_LWEX
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC5};
            mem[1] = {`VMICRO16_OP_SW,      3'h0, 3'h0, 5'h1};
            mem[2] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h1};
            mem[3] = {`VMICRO16_OP_LWEX,    3'h2, 3'h0, 5'h1};
            mem[4] = {`VMICRO16_OP_SWEX,    3'h3, 3'h0, 5'h1};
            `endif

            //`define TEST_MULTICORE
            `ifdef TEST_MULTICORE
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h80};
            mem[4] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
            mem[5] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[6] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[7] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[8] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
            mem[9] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h0};
            `endif

            //`define TEST_BR
            `ifdef TEST_BR
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h3, 8'h3};
            mem[2] = {`VMICRO16_OP_MOVI,    3'h1, 8'h2};
            mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
            mem[4] = {`VMICRO16_OP_BR,      3'h3, `VMICRO16_OP_BR_U};
```

```
            mem[5] = {`VMICRO16_OP_MOVI,    3'h0, 8'hFF};
            `endif

            //`define ALL_TEST
            `ifdef ALL_TEST
            // Standard all test
            // REGS0
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h81};
            mem[1] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
            mem[2] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
            // GPIO0
            mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
            mem[4] = {`VMICRO16_OP_MOVI,    3'h1, 8'hD};
            mem[5] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[6] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
            // TIM0
            mem[7] = {`VMICRO16_OP_MOVI,    3'h0, 8'h07};
            mem[8] = {`VMICRO16_OP_LW,      3'h3, 3'h0, 5'h03};
            // UART0
            mem[9]  = {`VMICRO16_OP_MOVI,    3'h0, 8'hA0};      // UART0
            mem[10] = {`VMICRO16_OP_MOVI,    3'h1, 8'h41};      // ascii A
            mem[11] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[12] = {`VMICRO16_OP_MOVI,    3'h1, 8'h42}; // ascii B
            mem[13] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[14] = {`VMICRO16_OP_MOVI,    3'h1, 8'h43}; // ascii C
            mem[15] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[16] = {`VMICRO16_OP_MOVI,    3'h1, 8'h44}; // ascii D
            mem[17] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[18] = {`VMICRO16_OP_MOVI,    3'h1, 8'h45}; // ascii D
            mem[19] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[20] = {`VMICRO16_OP_MOVI,    3'h1, 8'h46}; // ascii E
            mem[21] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            // BRAM0
            mem[22] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
            mem[23] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
            mem[24] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
            mem[25] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
            // GPIO1 (SSD 24-bit port)
            mem[26] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
            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
    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


module vmicro16_core_mmu # (
    parameter MEM_WIDTH     = 16,
    parameter MEM_DEPTH     = 64,

    parameter CORE_ID       = 3'h0,
    parameter CORE_ID_BITS  = `clog2(`CORES)
) (
    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,
    input                       mmu_lwex,
    input                       mmu_swex,
    output reg [MEM_WIDTH-1:0]  mmu_out,

    // interrupts
    output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
    output reg [`DEF_NUM_INT-1:0]             ints_mask,

    // TO APB interconnect
    output reg [`APB_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 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);

    // more luts than below but easier
    //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 intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
    //              && (mmu_addr <= `DEF_MMU_INTSV_E);
    //wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
    //              && (mmu_addr <= `DEF_MMU_INTSM_E);

    wire tim0_en = ~mmu_addr[12] && ~mmu_addr[9] && ~mmu_addr[7];
    wire sreg_en = mmu_addr[7] && ~mmu_addr[4] && ~mmu_addr[5];
    wire intv_en = mmu_addr[8] && ~mmu_addr[3];
    wire intm_en = mmu_addr[8] && mmu_addr[3];

    wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
    wire tim0_we   = (tim0_en && mmu_we);
    wire intv_we   = (intv_en && mmu_we);
    wire intm_we   = (intm_en && mmu_we);

    // Special register selects
    localparam SPECIAL_REGS = 8;
    wire [MEM_WIDTH-1:0] sr_val;

    // Interrupt vector and mask
    initial ints_vector = 0;
    initial ints_mask   = 0;
    wire [2:0] intv_addr = mmu_addr[`clog2(`DEF_NUM_INT)-1:0];
    always @(posedge clk)
        if (intv_we)
            ints_vector[intv_addr*`DATA_WIDTH +: `DATA_WIDTH] <= mmu_in;

    always @(posedge clk)
        if (intm_we)
            ints_mask <= mmu_in;


    always @(ints_vector)
        $display($time,
                "\tC%d\t\tints_vector W: | %h %h %h %h | %h %h %h %h |",
                 CORE_ID,
            ints_vector[0*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[1*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[2*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[3*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[4*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[5*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[6*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[7*`DATA_WIDTH +: `DATA_WIDTH]
            );

    always @(intm_we)
        $display($time, "\tC%d\t\tintm_we W: %b", CORE_ID, ints_mask);

    // Output port
    always @(*)
        if       (tim0_en) mmu_out = tim0_out;
        else if (sreg_en) mmu_out = sr_val;
        else if (intv_en) mmu_out = ints_vector[mmu_addr[2:0]*`DATA_WIDTH
```

```
                                                  +: `DATA_WIDTH];
        else if (intm_en) mmu_out = ints_mask;
        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_lwex,
                                      mmu_swex,
                                      CORE_ID[CORE_ID_BITS-1:0],
                                      mmu_addr[MEM_WIDTH-1:0]};

                        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

    (* ram_style = "block" *)
    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (SPECIAL_REGS),
        .USE_INITS  (0),
        .PARAM_DEFAULTS_R0  (CORE_ID),
        .PARAM_DEFAULTS_R1  (`CORES),
        .PARAM_DEFAULTS_R2  (`APB_BRAMO_CELLS),
        .PARAM_DEFAULTS_R3  (`SLAVES),
        .NAME       ("ram_sr")
    ) ram_sr (
        .clk        (clk),
        .reset      (reset),
        .mem_addr   (mmu_addr[`clog2(SPECIAL_REGS)-1:0]),
        .mem_in     (),
        .mem_we     (),
```

```
        .mem_out    (sr_val)
    );

    // Each M core has a TIM0 scratch memory
    (* ram_style = "block" *)
    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .USE_INITS  (0),
        .NAME       ("TIM0")
    ) TIM0 (
        .clk        (clk),
        .reset      (reset),
        .mem_addr   (mmu_addr[7:0]),
        .mem_in     (mmu_in),
        .mem_we     (tim0_we),
        .mem_out    (tim0_out)
    );
endmodule



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
);
    (* ram_style = "distributed" *)
    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

    `ifdef ICARUS
        always @(regs)
            $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
                CORE_ID,
                regs[0], regs[1], regs[2], regs[3],
                regs[4], regs[5], regs[6], regs[7]);
    `endif

    always @(posedge clk)
        if (reset) begin
            for(i = 0; i < CELL_DEPTH; i = i + 1)
                regs[i] <= 0;
            regs[0] <= PARAM_DEFAULTS_R0;
            regs[1] <= PARAM_DEFAULTS_R1;
        end
        else if (we) begin
            $display($time, "\tC%02h: REGS #%s: Writing %h to reg[%d]",
                CORE_ID, DEBUG_NAME, wd, ws1);

            // Perform the write
            regs[ws1] <= wd;
        end

    // sync writes, async reads
    assign rd1 = regs[rs1];
    //assign rd2 = regs[rs2];
endmodule

```

```
module vmicro16_dec # (
    parameter INSTR_WIDTH    = 16,
    parameter INSTR_OP_WIDTH = 5,
    parameter INSTR_RS_WIDTH = 3,
    parameter ALU_OP_WIDTH   = 5
) (
    //input clk,   // not used yet (all combinational)
    //input reset, // not used yet (all combinational)

    input  [INSTR_WIDTH-1:0]    instr,

    output [INSTR_OP_WIDTH-1:0] opcode,
    output [INSTR_RS_WIDTH-1:0] rd,
    output [INSTR_RS_WIDTH-1:0] ra,
    output [3:0]                imm4,
    output [7:0]                imm8,
    output [11:0]               imm12,
    output [4:0]                simm5,

    // This can be freely increased without affecting the isa
    output reg [ALU_OP_WIDTH-1:0] alu_op,

    output reg has_imm4,
    output reg has_imm8,
    output reg has_imm12,
    output reg has_we,
    output reg has_br,
    output reg has_mem,
    output reg has_mem_we,
    output reg has_cmp,

    output halt,
    output intr,

    output reg has_lwex,
    output reg has_swex

    // TODO: Use to identify bad instruction and
    //       raise exceptions
    //,output     is_bad
);
    assign opcode = instr[15:11];
    assign rd     = instr[10:8];
    assign ra     = instr[7:5];
    assign imm4   = instr[3:0];
    assign imm8   = instr[7:0];
    assign imm12  = instr[11:0];
    assign simm5  = instr[4:0];

    // exme_op
    always @(*) case (opcode)
        `VMICRO16_OP_SPCL: casez(instr[11:0])
            `VMICRO16_OP_SPCL_NOP,
            `VMICRO16_OP_SPCL_HALT,
            `VMICRO16_OP_SPCL_INTR:   alu_op = `VMICRO16_ALU_NOP;
            default:                  alu_op = `VMICRO16_ALU_NOP; endcase

        `VMICRO16_OP_LW:              alu_op = `VMICRO16_ALU_LW;
        `VMICRO16_OP_SW:              alu_op = `VMICRO16_ALU_SW;
        `VMICRO16_OP_LWEX:            alu_op = `VMICRO16_ALU_LW;
        `VMICRO16_OP_SWEX:            alu_op = `VMICRO16_ALU_SW;

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

        `VMICRO16_OP_BR:              alu_op = `VMICRO16_ALU_BR;
        `VMICRO16_OP_MULT:            alu_op = `VMICRO16_ALU_MULT;

        `VMICRO16_OP_CMP:             alu_op = `VMICRO16_ALU_CMP;
        `VMICRO16_OP_SETC:            alu_op = `VMICRO16_ALU_SETC;

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

        `VMICRO16_OP_ARITH_U:     casez (simm5)
            `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
            `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
            `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
            default:                  alu_op = `VMICRO16_ALU_BAD; endcase

        `VMICRO16_OP_ARITH_S:     casez (simm5)
            `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
            `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
            `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
```

```
            default:                  alu_op = `VMICRO16_ALU_BAD; endcase

        default: begin
                                      alu_op = `VMICRO16_ALU_NOP;
            $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
        end
    endcase

    // Special opcodes
    //assign nop  == ((opcode == `VMICRO16_OP_SPCL) & (~instr[0]));
    assign halt = ((opcode == `VMICRO16_OP_SPCL) &   instr[0]);
    assign intr = ((opcode == `VMICRO16_OP_SPCL) &   instr[1]);

    // Register writes
    always @(*) case (opcode)
        `VMICRO16_OP_LWEX,
        `VMICRO16_OP_SWEX,
        `VMICRO16_OP_LW,
        `VMICRO16_OP_MOV,
        `VMICRO16_OP_MOVI,
        //`VMICRO16_OP_MOVI_L,
        `VMICRO16_OP_ARITH_U,
        `VMICRO16_OP_ARITH_S,
        `VMICRO16_OP_SETC,
        `VMICRO16_OP_BIT,
        `VMICRO16_OP_MULT:      has_we = 1'b1;
        default:                has_we = 1'b0;
    endcase

    // Contains 4-bit immediate
    always @(*)
        if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
            ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
            has_imm4 = 1'b1;
        else
            has_imm4 = 1'b0;

    // Contains 8-bit immediate
    always @(*) case (opcode)
        `VMICRO16_OP_MOVI,
        `VMICRO16_OP_BR:        has_imm8 = 1'b1;
        default:                has_imm8 = 1'b0;
    endcase

    //// Contains 12-bit immediate
    //always @(*) case (opcode)
    //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
    //    default:                has_imm12 = 1'b0;
    //endcase

    // Will branch the pc
    always @(*) case (opcode)
        `VMICRO16_OP_BR:    has_br = 1'b1;
        default:            has_br = 1'b0;
    endcase

    // Requires external memory
    always @(*) case (opcode)
        `VMICRO16_OP_LW,
        `VMICRO16_OP_SW,
        `VMICRO16_OP_LWEX,
        `VMICRO16_OP_SWEX:  has_mem = 1'b1;
        default:            has_mem = 1'b0;
    endcase

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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



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

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

    output [7:0] dbug,

    output       halt,

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

    // APB master to slave interface (apb_intercon)
    output   [`APB_WIDTH-1:0]    w_PADDR,
    output                       w_PWRITE,
    output                       w_PSELx,
    output                       w_PENABLE,
    output   [DATA_WIDTH-1:0]    w_PWDATA,
    input    [DATA_WIDTH-1:0]    w_PRDATA,
    input                        w_PREADY

`ifndef DEF_CORE_HAS_INSTR_MEM
    , // APB master interface to slave instruction memory
    output reg [`APB_WIDTH-1:0]    w2_PADDR,
    output reg                     w2_PWRITE,
    output reg                     w2_PSELx,
    output reg                     w2_PENABLE,
    output reg [DATA_WIDTH-1:0]    w2_PWDATA,
    input      [DATA_WIDTH-1:0]    w2_PRDATA,
    input                          w2_PREADY
`endif
);
    localparam STATE_IF = 0;
    localparam STATE_R1 = 1;
    localparam STATE_R2 = 2;
    localparam STATE_ME = 3;
    localparam STATE_WB = 4;
    localparam STATE_FE = 5;
    localparam STATE_IDLE = 6;
    localparam STATE_HALT = 7;
    reg  [2:0] r_state = STATE_IF;

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

    assign dbug = {7'h00, w_halt};
    assign halt = w_halt;
```

```
    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  [DATA_WIDTH-1:0] r_instr_rdd = 0;
    reg  [DATA_WIDTH-1:0] 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                  r_instr_has_lwex;
    wire                  r_instr_has_swex;

    wire [DATA_WIDTH-1:0] r_alu_out;

    wire [DATA_WIDTH-1:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
    wire [DATA_WIDTH-1:0] r_mem_scratch_in   = r_instr_rdd;
    wire [DATA_WIDTH-1:0] 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 [DATA_WIDTH-1:0] r_reg_rd1_s;
    wire [DATA_WIDTH-1:0] r_reg_rd1_i;
    wire [DATA_WIDTH-1:0] r_reg_rd1 = regs_use_int ? r_reg_rd1_i : r_reg_rd1_s;
    //wire [15:0] r_reg_rd2;
    wire [DATA_WIDTH-1:0] 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
    wire        w_intr;
    wire        w_branch_en;
    wire        w_branching   = r_instr_has_br && w_branch_en;
    reg  [3:0]  r_cmp_flags   = 4'h00; // N, Z, C, V

    always @(r_cmp_flags)
        $display($time, "\tC%02h:\tALU CMP: %b", CORE_ID, r_cmp_flags);

    // 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

    reg regs_use_int = 0;
    `ifdef DEF_ENABLE_INT
    wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_vector;
    wire [`DEF_NUM_INT-1:0]             ints_mask;
    wire                                has_int = ints & ints_mask;
    reg int_pending = 0;
    reg int_pending_ack = 0;
    always @(posedge clk)
        if (int_pending_ack)
            // We've now branched to the isr
            int_pending <= 0;
        else if (has_int)
            // Notify fsm to switch to the ints_vector at the last stage
            int_pending <= 1;
        else if (w_intr)
            // Return to Interrupt instruction called,
            //   so we've finished with the interrupt
            int_pending <= 0;
    `endif

    // Next program counter logic
    reg [`DATA_WIDTH-1:0] next_pc = 0;
    always @(posedge clk)
        if (reset)
            r_pc <= 0;
        else if (r_state == STATE_WB) begin
            `ifdef DEF_ENABLE_INT
            if (int_pending) begin
                $display($time, "\tC%02h: Jumping to ISR: %h",
                    CORE_ID,
                    ints_vector[0 +: `DATA_WIDTH]);
                // TODO: check bounds
```

```
                    // Save state
                    r_pc_saved      <= r_pc + 1;
                    regs_use_int    <= 1;
                    int_pending_ack <= 1;
                    // Jump to ISR
                    r_pc            <= ints_vector[0 +: `DATA_WIDTH];
                end else if (w_intr) begin
                    $display($time, "\tC%02h: Returning from ISR: %h",
                        CORE_ID, r_pc_saved);

                    // Restore state
                    r_pc            <= r_pc_saved;
                    regs_use_int    <= 0;
                    int_pending_ack <= 0;
                end else
                `endif
                if (w_branching) begin
                    $display($time, "\tC%02h: branching to %h", CORE_ID, r_instr_rdd);
                    r_pc            <= r_instr_rdd;

                    `ifdef DEF_ENABLE_INT
                        int_pending_ack <= 0;
                    `endif
                end else if (r_pc < (MEM_INSTR_DEPTH-1)) begin
                    // normal increment
                    // pc <= pc + 1
                    r_pc            <= r_pc + 1;

                    `ifdef DEF_ENABLE_INT
                        int_pending_ack <= 0;
                    `endif
                end
            end // end r_state == STATE_WB
            else if (r_state == STATE_HALT) begin
                `ifdef DEF_ENABLE_INT
                // Only an interrupt can return from halt
                // duplicate code form STATE_ME!
                if (int_pending) begin
                    $display($time, "\tC%02h: Jumping to ISR: %h", CORE_ID, ints_vector[0 +: `DATA_WIDTH]);
                    // TODO: check bounds
                    // Save state
                    r_pc_saved      <= r_pc;// + 1; HALT = stay with same PC
                    regs_use_int    <= 1;
                    int_pending_ack <= 1;
                    // Jump to ISR
                    r_pc            <= ints_vector[0 +: `DATA_WIDTH];
                end else if (w_intr) begin
                    $display($time, "\tC%02h: Returning from ISR: %h", CORE_ID, r_pc_saved);
                    r_pc            <= r_pc_saved;
                    regs_use_int    <= 0;
                    int_pending_ack <= 0;
                end
                `endif
            end

`ifndef DEF_CORE_HAS_INSTR_MEM
    initial w2_PSELx   = 0;
    initial w2_PENABLE = 0;
    initial w2_PADDR   = 0;
`endif

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

`ifdef DEF_CORE_HAS_INSTR_MEM
            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
            // wait for global instruction rom to give us our instruction
            if (r_state == STATE_IF) begin
                // wait for ready signal
                if (!w2_PREADY) begin
                    w2_PSELx   <= 1;
                    w2_PWRITE  <= 0;
                    w2_PENABLE <= 1;
```

```
                        w2_PWDATA  <= 0;
                        w2_PADDR   <= r_pc;
                    end else begin
                        w2_PSELx   <= 0;
                        w2_PWRITE  <= 0;
                        w2_PENABLE <= 0;
                        w2_PWDATA  <= 0;

                        r_instr <= w2_PRDATA;

                        $display("");
                        $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
                        $display($time, "\tC%02h: INSTR: %h", CORE_ID, w2_PRDATA);

                        r_state <= STATE_R1;
                    end
                end
`endif

                else if (r_state == STATE_R1) begin
                    if (w_halt) begin
                        $display("");
                        $display("");
                        $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
                        r_state <= STATE_HALT;
                    end else begin
                        // primary operand
                        r_instr_rdd <= r_reg_rd1;
                        r_state     <= STATE_R2;
                    end
                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;
                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
                    if (r_instr_has_cmp) begin
                        $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
                        r_cmp_flags <= r_alu_out[3:0];
                    end

                    r_state <= STATE_FE;
                end
                else if (r_state == STATE_FE)
                    r_state <= STATE_IF;
                else if (r_state == STATE_HALT) begin
                    `ifdef DEF_ENABLE_INT
                        if (int_pending) begin
                            r_state <= STATE_FE;
                        end
                    `endif
                end
            end

`ifdef DEF_CORE_HAS_INSTR_MEM
    // Instruction ROM
    (* rom_style = "distributed" *)
    vmicro16_bram # (
        .MEM_WIDTH      (DATA_WIDTH),
        .MEM_DEPTH      (MEM_INSTR_DEPTH),
        .CORE_ID        (CORE_ID),
        .USE_INITS      (1),
        .NAME           ("INSTR_MEM")
    ) mem_instr (
        .clk            (clk),
        .reset          (reset),
        // port 1
        .mem_addr       (r_pc),
        .mem_in         (0),
        .mem_we         (1'b0),  // ROM
        .mem_out        (w_mem_instr_out)
    );
`endif

```

```
    // MMU
    vmicro16_core_mmu # (
        .MEM_WIDTH      (DATA_WIDTH),
        .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
        .CORE_ID        (CORE_ID)
    ) mmu (
        .clk            (clk),
        .reset          (reset),
        .req            (r_mem_scratch_req),
        .busy           (r_mem_scratch_busy),
        // interrupts
        .ints_vector    (ints_vector),
        .ints_mask      (ints_mask),
        // port 1
        .mmu_addr       (r_mem_scratch_addr),
        .mmu_in         (r_mem_scratch_in),
        .mmu_we         (r_mem_scratch_we),
        .mmu_lwex       (r_instr_has_lwex),
        .mmu_swex       (r_instr_has_swex),
        .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
    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           (w_halt),
        .intr           (w_intr),
        .has_lwex       (r_instr_has_lwex),
        .has_swex       (r_instr_has_swex)
    );

    // Software registers
    vmicro16_regs # (
        .CORE_ID    (CORE_ID),
        .CELL_WIDTH (`DATA_WIDTH)
    ) regs (
        .clk        (clk),
        .reset      (reset),
        // async port 0
        .rs1        (r_reg_rs1),
        .rd1        (r_reg_rd1_s),
        // async port 1
        //.rs2        (),
        //.rd2        (),
        // write port
        .we         (r_reg_we && ~regs_use_int),
        .ws1        (r_instr_rsd),
        .wd         (r_reg_wd)
    );

    // Interrupt replacement registers
    `ifdef DEF_ENABLE_INT
    vmicro16_regs # (
        .CORE_ID    (CORE_ID),
        .CELL_WIDTH (`DATA_WIDTH),
        .DEBUG_NAME ("REGSINT")
    ) regs_intr (
        .clk        (clk),
        .reset      (reset),
        // async port 0
        .rs1        (r_reg_rs1),
        .rd1        (r_reg_rd1_i),
        // async port 1
        //.rs2        (),
```

```
        //.rd2        (),
        // write port
        .we         (r_reg_we && regs_use_int),
        .ws1        (r_instr_rsd),
        .wd         (r_reg_wd)
    );
    `endif

    // ALU
    vmicro16_alu # (
        .CORE_ID(CORE_ID)
    ) alu (
        .op         (r_instr_alu_op),
        .a          (r_instr_rdd),
        .b          (r_instr_rda),
        .flags      (r_cmp_flags),
        // async output
        .c          (r_alu_out)
    );

    branch branch_check (
        .flags      (r_cmp_flags),
        .cond       (r_instr_imm8),
        .en         (w_branch_en)
    );

endmodule
```

E.2 Peripheral Code Listing

Various memory-mapped APB peripherals, such as GPIO, UART, timers, and memory.

```
// Vmicro16 peripheral modules

`include "vmicro16_soc_config.v"
`include "formal.v"

// Simple watchdog peripheral
module vmicro16_watchdog_apb # (
    parameter BUS_WIDTH   = 16,
    parameter NAME        = "WD",
    parameter CLK_HZ      = 50_000_000
) (
    input clk,
    input reset,

    // APB Slave to master interface
    input  [0:0]                S_PADDR, // not used (optimised out)
    input                       S_PWRITE,
    input                       S_PSELx,
    input                       S_PENABLE,
    input  [0:0]                S_PWDATA,

    // prdata not used
    output [0:0]                S_PRDATA,
    output                      S_PREADY,

    // watchdog reset, active high
    output reg                  wdreset
);
    //assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    wire   we       = (S_PSELx & S_PENABLE & S_PWRITE);

    // countdown timer
    reg [`clog2(CLK_HZ)-1:0] timer = CLK_HZ;

    wire w_wdreset = (timer == 0);

    // infer a register to aid timing
    initial wdreset = 0;
    always @(posedge clk)
        wdreset <= w_wdreset;

    always @(posedge clk)
        if (we) begin
            $display($time, "\t\%s <= RESET", NAME);
            timer <= CLK_HZ;
        end else begin
            timer <= timer - 1;
        end
endmodule
```

```

module timer_apb # (
    parameter CLK_HZ = 50_000_000
) (
    input clk,
    input reset,

    input clk_en,

    // 0 16-bit value   R/W
    // 1 16-bit control R    b0 = start, b1 = reset
    // 2 16-bit prescaler
    input      [1:0]                S_PADDR,

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

    output reg [`DATA_WIDTH-1:0]    S_PRDATA,
    output                          S_PREADY,

    output out,
    output [`DATA_WIDTH-1:0] int_data
);
    //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    wire   en       = (S_PSELx & S_PENABLE);
    wire   we       = (en & S_PWRITE);

    reg [`DATA_WIDTH-1:0] r_counter = 0;
    reg [`DATA_WIDTH-1:0] r_load = 0;
    reg [`DATA_WIDTH-1:0] r_pres = 0;
    reg [`DATA_WIDTH-1:0] r_ctrl = 0;

    localparam CTRL_START = 0;
    localparam CTRL_RESET = 1;
    localparam CTRL_INT   = 2;

    localparam ADDR_LOAD = 2'b00;
    localparam ADDR_CTRL = 2'b01;
    localparam ADDR_PRES = 2'b10;

    always @(*) begin
        S_PRDATA = 0;
        if (en)
            case(S_PADDR)
                ADDR_LOAD: S_PRDATA = r_counter;
                ADDR_CTRL: S_PRDATA = r_ctrl;
                //ADDR_CTRL: S_PRDATA = r_pres;
                default:   S_PRDATA = 0;
            endcase
    end

    // prescaler counts from r_pres to 0, emitting a stb signal
    //   to enable the r_counter step
    reg [`DATA_WIDTH-1:0] r_pres_counter = 0;
    wire counter_en = (r_pres_counter == 0);
    always @(posedge clk)
        if (r_pres_counter == 0)
            r_pres_counter <= r_pres;
        else
            r_pres_counter <= r_pres_counter - 1;

    always @(posedge clk)
        if (we)
            case(S_PADDR)
                // Write to the load register:
                //   Set load register
                //   Set counter register
                ADDR_LOAD: begin
                    r_load          <= S_PWDATA;
                    r_counter       <= S_PWDATA;
                    $display($time, "\t\ttimr0: WRITE LOAD: %h", S_PWDATA);
                end
                ADDR_CTRL: begin
                    r_ctrl   <= S_PWDATA;
                    $display($time, "\t\ttimr0: WRITE CTRL: %h", S_PWDATA);
                end
                ADDR_PRES: begin
                    r_pres   <= S_PWDATA;
                    $display($time, "\t\ttimr0: WRITE PRES: %h", S_PWDATA);
                end
            endcase
        else
            if (r_ctrl[CTRL_START]) begin
                if (r_counter == 0)
                    r_counter <= r_load;
                else if(counter_en)
                    r_counter <= r_counter -1;
```

```
            end else if (r_ctrl[CTRL_RESET])
                r_counter <= r_load;

    // generate the output pulse when r_counter == 0
    //   out = (counter reached zero && counter started)
    assign out      = (r_counter == 0) && r_ctrl[CTRL_START]; // && r_ctrl[CTRL_INT];
    assign int_data = {`DATA_WIDTH{1'b1}};
endmodule


// APB wrapped programmable vmicro16_bram
module vmicro16_bram_prog_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter APB_PADDR    = 0,
    parameter USE_INITS    = 0,
    parameter NAME         = "BRAMPROG",
    parameter CORE_ID      = 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,

    // interface to program the instruction memory
    input     [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] addr,
    input     [`DATA_WIDTH-1:0]                  data,
    input                                        we,
    input                                        prog
);
    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;
    wire s_we = (S_PSELx & S_PENABLE & S_PWRITE);

    wire [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] mem_addr = we ? addr : S_PADDR;
    wire [`DATA_WIDTH-1:0]                  mem_data = we ? data : S_PWDATA;
    wire                                    mem_we   = we | s_we;

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .NAME       ("BRAMPROG"),
        .USE_INITS  (0),
        .CORE_ID    (-1)
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

        .mem_addr   (mem_addr),
        .mem_in     (mem_data),
        .mem_we     (mem_we),
        .mem_out    (mem_out)
    );
endmodule

// APB wrapped vmicro16_bram
module vmicro16_bram_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter APB_PADDR    = 0,
    parameter USE_INITS    = 0,
    parameter NAME         = "BRAM",
    parameter CORE_ID      = 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\t%s => %h", NAME, mem_out);

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

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .NAME       (NAME),
        .USE_INITS  (1),
        .CORE_ID    (-1)
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

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

// Shared memory with hardware monitor (LWEX/SWEX)
module vmicro16_bram_ex_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter CORE_ID_BITS = 3,
    parameter SWEX_SUCCESS = 16'h0000,
    parameter SWEX_FAIL    = 16'h0001
) (
    input clk,
    input reset,

    // |19    |18    |16             |15          0|
    // | LWEX | SWEX | 3 bit CORE_ID |     S_PADDR |
    input   [`APB_WIDTH-1:0]         S_PADDR,

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

    output reg [MEM_WIDTH-1:0]       S_PRDATA,
    output                           S_PREADY
);
    // exclusive flag checks
    wire [MEM_WIDTH-1:0] mem_out;
    reg                  swex_success = 0;

    localparam ADDR_BITS = `clog2(MEM_DEPTH);

    // hack to create a 1 clock delay to S_PREADY
    // for bram to be ready
    reg cdelay = 1;
    always @(posedge clk)
        if (S_PSELx)
            cdelay <= 0;
        else
            cdelay <= 1;

    //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE & (!cdelay)) ? 1'b1       : 1'b0;
    assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
    wire   en       = (S_PSELx & S_PENABLE);

    // 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         = `APB_WIDTH -1;
    localparam PADDR_CORE_ID_MSB     = TOP_BIT_INDEX - 2;
    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                      swex        = S_PADDR[TOP_BIT_INDEX-1];
    wire [CORE_ID_BITS-1:0]   core_id     = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
    // CORE_ID to write to ex_flags register
    wire [ADDR_BITS-1:0]      mem_addr    = S_PADDR[ADDR_BITS-1:0];

    wire [CORE_ID_BITS:0]     ex_flags_read;
    wire                      is_locked      = |ex_flags_read;
```

```
    wire                    is_locked_self = is_locked && (core_id == (ex_flags_read-1));

    // Check exclusive access flags
    always @(*) begin
        swex_success = 0;
        if (en)
            // bug!
            if (!swex && !lwex)
                swex_success = 1;
            else if (swex)
                if (is_locked && !is_locked_self)
                    // someone else has locked it
                    swex_success = 0;
                else if (is_locked && is_locked_self)
                    swex_success = 1;
    end

    always @(*)
        if (swex)
            if (swex_success)
                S_PRDATA = SWEX_SUCCESS;
            else
                S_PRDATA = SWEX_FAIL;
        else
            S_PRDATA = mem_out;

    wire reg_we = en && ((lwex && !is_locked)
                    || (swex && swex_success));

    reg  [CORE_ID_BITS:0] reg_wd;
    always @(*) begin
        reg_wd = {{CORE_ID_BITS}{1'b0}};

        if (en)
            // if wanting to lock the addr
            if (lwex)
                // and not already locked
                if (!is_locked) begin
                    reg_wd = (core_id + 1);
                end
            else if (swex)
                if (is_locked && is_locked_self)
                    reg_wd = {{CORE_ID_BITS}{1'b0}};
    end

    // Exclusive flag for each memory cell
    vmicro16_bram # (
        .MEM_WIDTH  (CORE_ID_BITS + 1),
        .MEM_DEPTH  (MEM_DEPTH),
        .USE_INITS  (0),
        .NAME       ("rexram")
    ) ram_exflags (
        .clk        (clk),
        .reset      (reset),

        .mem_addr   (mem_addr),
        .mem_in     (reg_wd),
        .mem_we     (reg_we),
        .mem_out    (ex_flags_read)
    );

    always @(*)
        if (S_PSELx && S_PENABLE)
            $display($time, "\t\tBRAMex[%h] READ %h\tCORE: %h",
                mem_addr, mem_out, S_PADDR[16 +: CORE_ID_BITS]);

    always @(posedge clk)
        if (we)
            $display($time, "\t\tBRAMex[%h] WRITE %h\tCORE: %h",
                mem_addr, S_PWDATA, S_PADDR[16 +: CORE_ID_BITS]);

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .USE_INITS  (0),
        .NAME       ("BRAMexinst")
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

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

// Simple APB memory-mapped register set
module vmicro16_regs_apb # (
```

```
    parameter BUS_WIDTH         = 16,
    parameter DATA_WIDTH        = 16,
    parameter CELL_DEPTH        = 8,
    parameter PARAM_DEFAULTS_R0 = 0,
    parameter PARAM_DEFAULTS_R1 = 0
) (
    input clk,
    input reset,
    // APB Slave to master interface
    input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [DATA_WIDTH-1:0]         S_PWDATA,

    output [DATA_WIDTH-1:0]         S_PRDATA,
    output                          S_PREADY
);
    wire [DATA_WIDTH-1:0] rd1;

    assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1  : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    assign reg_we   = (S_PSELx & S_PENABLE & S_PWRITE);

    always @(*)
        if (reg_we)
            $display($time, "\t\tREGS_APB[%h] <= %h",
                S_PADDR, S_PWDATA);

    always @(*)
        `rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))

    vmicro16_regs # (
        .CELL_DEPTH         (CELL_DEPTH),
        .CELL_WIDTH         (DATA_WIDTH),
        .PARAM_DEFAULTS_R0  (PARAM_DEFAULTS_R0),
        .PARAM_DEFAULTS_R1  (PARAM_DEFAULTS_R1)
    ) regs_apb (
        .clk    (clk),
        .reset  (reset),
        // port 1
        .rs1    (S_PADDR),
        .rd1    (rd1),
        .we     (reg_we),
        .ws1    (S_PADDR),
        .wd     (S_PWDATA)
        // port 2 unconnected
        //.rs2    (),
        //.rd2    ()
    );
endmodule

// Simple GPIO write only peripheral
module vmicro16_gpio_apb # (
    parameter BUS_WIDTH  = 16,
    parameter DATA_WIDTH = 16,
    parameter PORTS      = 8,
    parameter NAME       = "GPIO"
) (
    input clk,
    input reset,
    // APB Slave to master interface
    input  [0:0]                    S_PADDR, // not used (optimised out)
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [DATA_WIDTH-1:0]         S_PWDATA,

    output [DATA_WIDTH-1:0]         S_PRDATA,
    output                          S_PREADY,
    output reg [PORTS-1:0]          gpio
);
    assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);

    always @(posedge clk)
        if (reset)
            gpio <= 0;
        else if (ports_we) begin
            $display($time, "\t\%s <= %h", NAME, S_PWDATA[PORTS-1:0]);
            gpio <= S_PWDATA[PORTS-1:0];
        end
endmodule
```