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 16, 2019

Word count: 4689

Abstract

This interim report details the 4-month progress on a project to design, implement, and verify, a multi-core FPGA RISC processor. The project has been split into two stages: firstly to build a functional single-core RISC processor, and then secondly to add multiprocessor principles and functionality to it.

Current multiprocessor and network-on-chip communication methods have been discussed and how they could be included in this multi-core RISC design. To-date, a 16-bit instruction set architecture has been designed featuring common load/store instructions, comparison, and bitwise operations. A single-core processor has been implemented in Verilog and verified using simulations/test benches running various simple software programs.

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

Revision History

Date	Version	Changes
10/04/2019	2.02	Update future stages.
05/04/2019	2.01	Fix processor RTL diagram.
04/04/2019	2.00	Initial processor RTL diagram.
01/04/2019	1.00	Initial section outline.

Document revisions.

Declaration of Academic Integrity

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

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

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

Name: Ben David Lancaster
Date: August 16, 2019

Table of Contents

Chapter 1

Introduction

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

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

1.1 Why Multi-core?

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

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

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

1.2 Why RISC?

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

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

1.3 Why FPGA?

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

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

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

Chapter 2

Background

2.1 Amdahl's Law and Parallelism

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

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

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

In scenarios where inter-node communication is prominent however, access to the MTS interface must be scheduled to avoid access conflicts which introduces delays and idle times in the software programs execution, resulting in lower throughput. Figure 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.

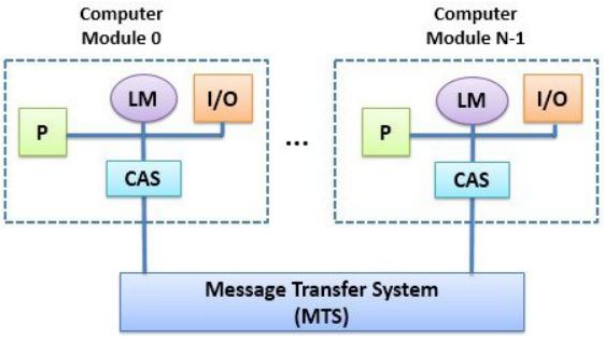


Figure 2.1: A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [3].

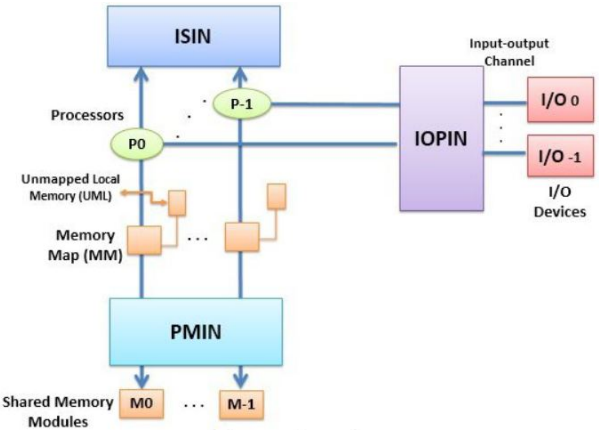


Figure 2.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [3].

This project will utilise a loosely coupled architecture due to 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 [4]. NoC's can generally support hundreds to millions of processing cores. Figure 2.3 shows an example 16-core network-on-chip architecture. NoC's can scale to very large sizes while not sacrificing performance because each processor core is able to drive the network rather than needing to wait for a shared bus to become free before doing so.

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

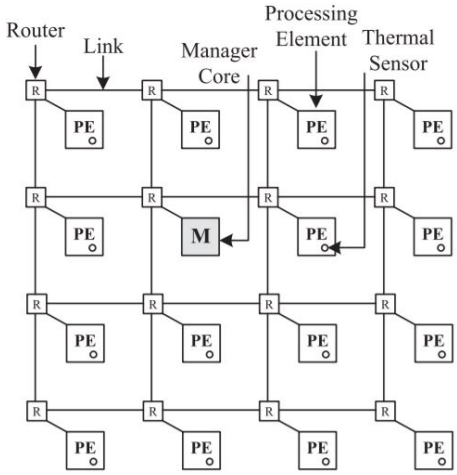


Figure 2.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [6].

Chapter 3

Project Overview

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

3.1 Project Deliverables

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

3.1.1 Core Deliverables (CD)

The project's core deliverables are described below.

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

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

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

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

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

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

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

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

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

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

3.1.2 Extended Deliverables (ED)

The project's extended deliverables are described below.

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

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

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

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

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

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

ED7 Store instruction memory in SPI flash.

ED8 Reprogram instruction memory at runtime from host computer.

ED9 Processor external debugger using host-processor link.

3.2 Project Timeline

3.2.1 Project Stages

The project is split up into many stages to aid planning and management of the project. There are 8 unique stage areas: 1. 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.

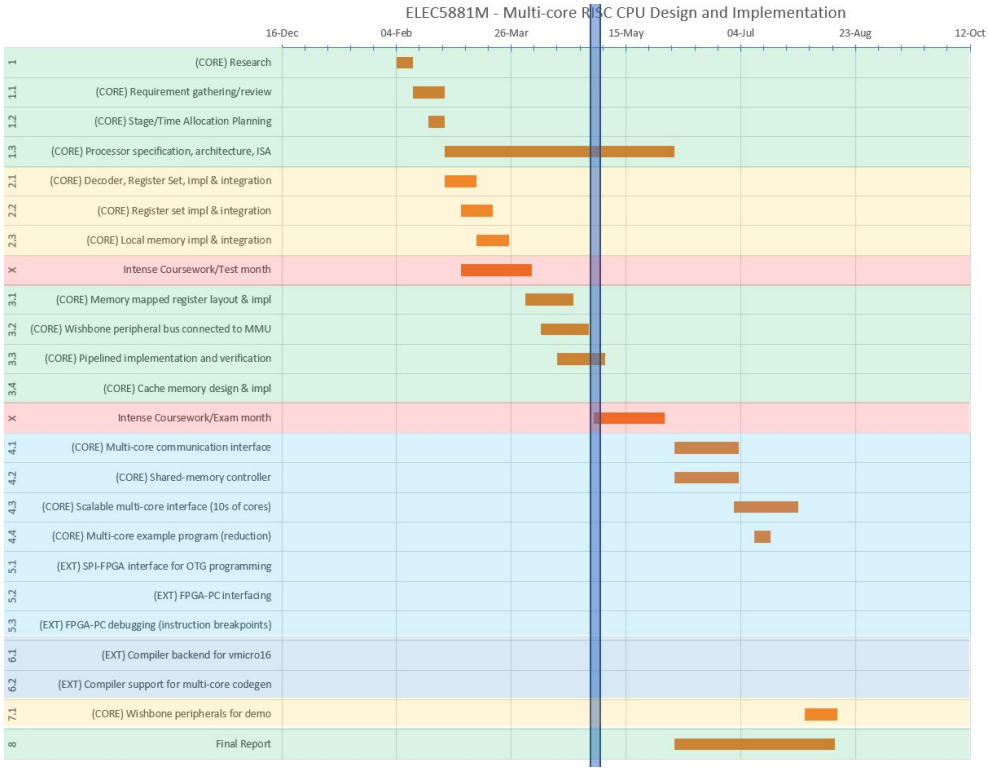


Figure 3.1: Project stages in a Gantt chart.

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.

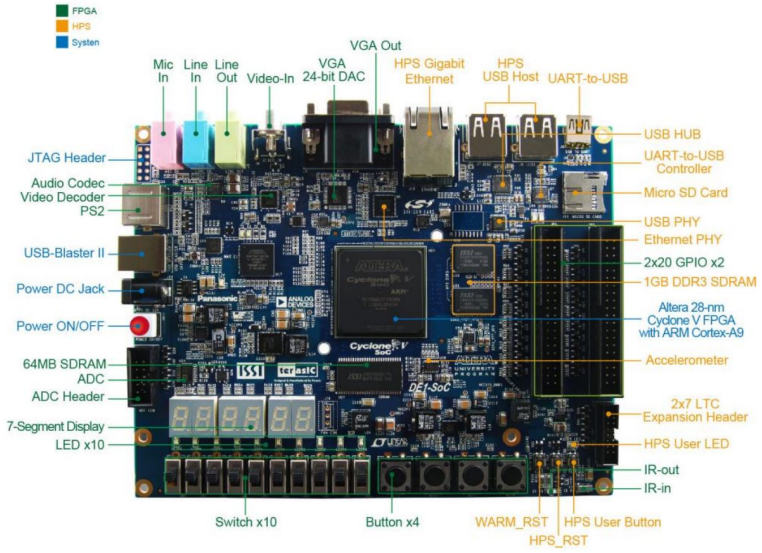


Figure 3.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [10].

Minispartan 6+ FPGA Development Board

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

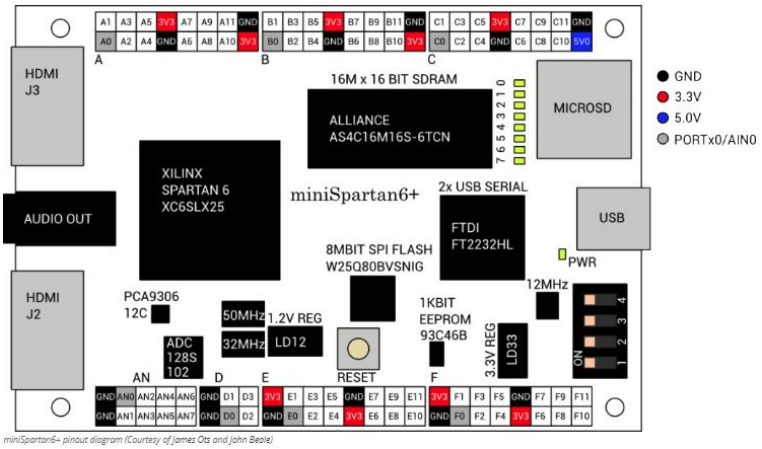


Figure 3.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [11].

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

Interconnect

4.1 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.

4.1.1 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.

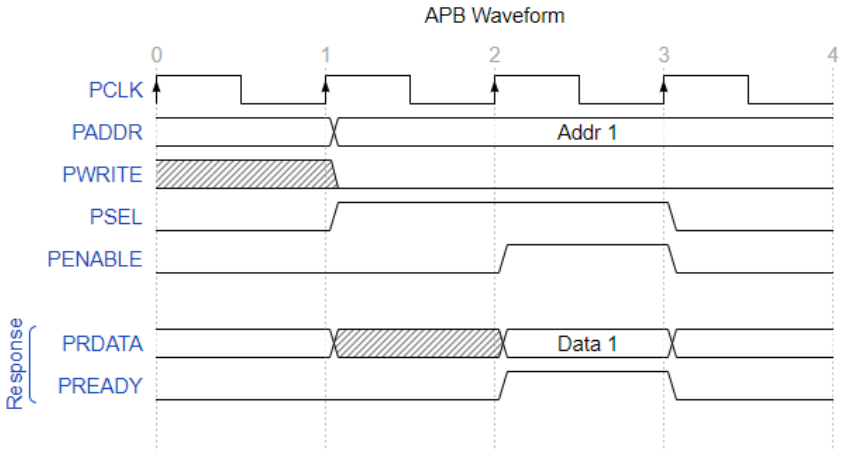


Figure 4.1: Waveform showing an APB read transaction.

4.2 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 4.2

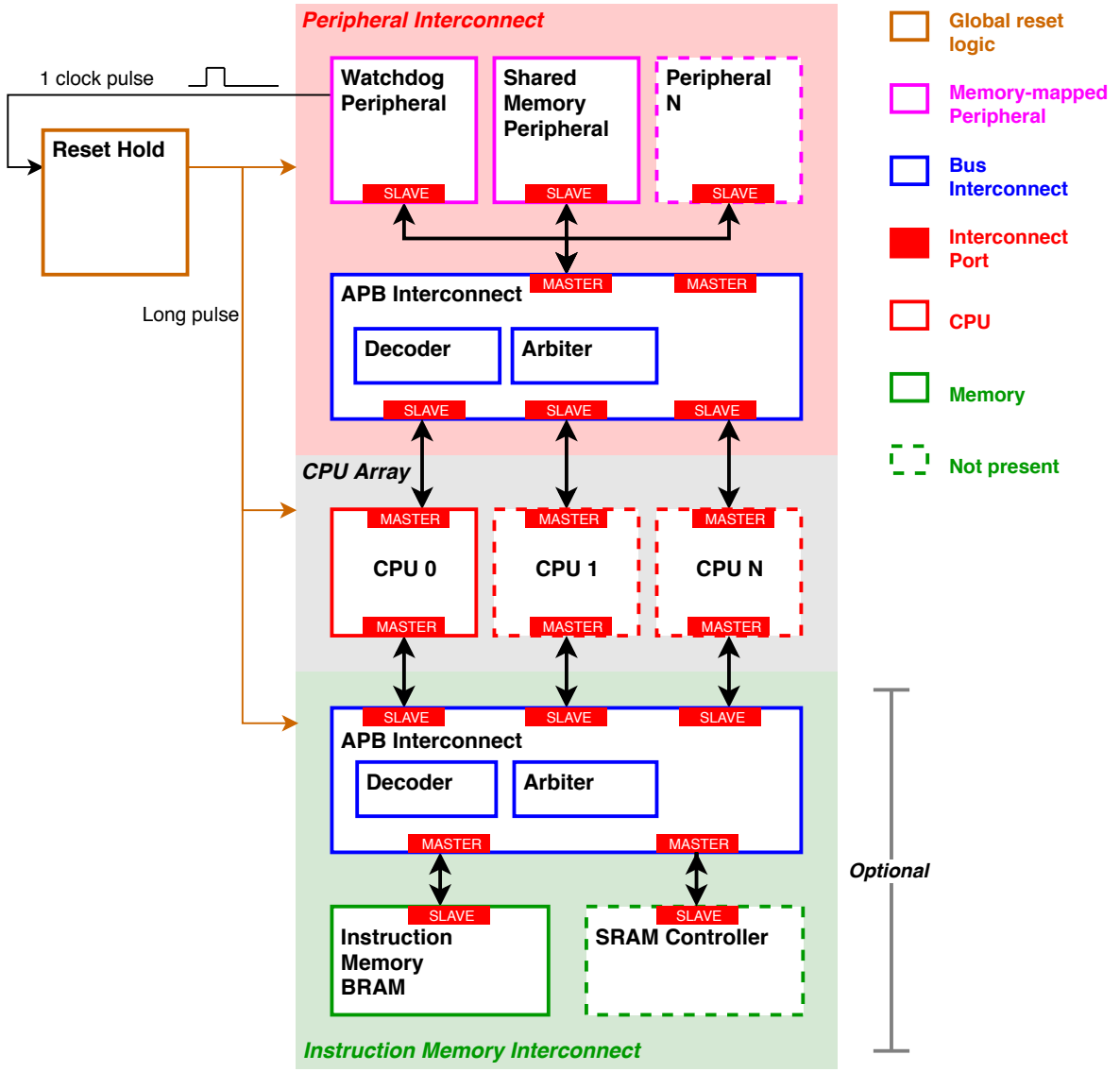


Figure 4.2: Block diagram of the Vmicro16 system-on-chip.

4.2.1 Design Considerations

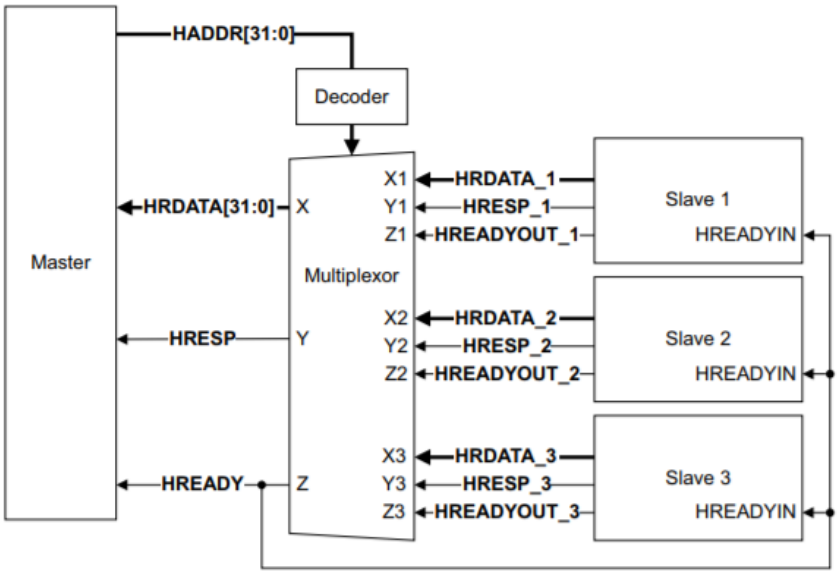
There are several design issues to consider for this project. These are listed below:

- **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.

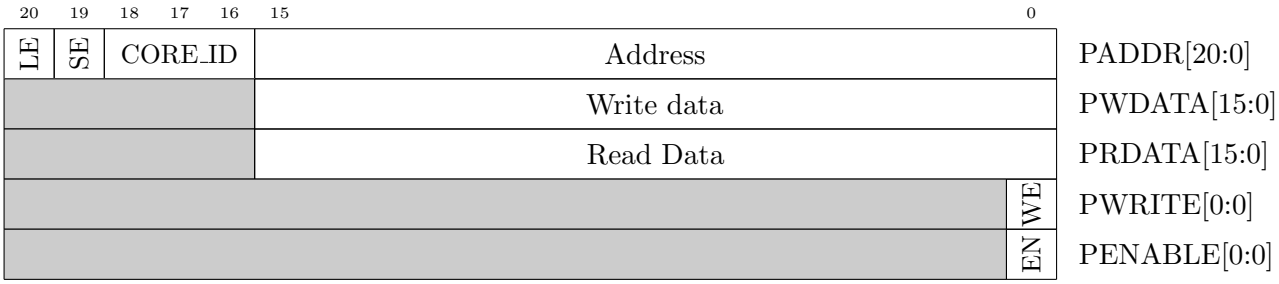
- **Ease of implementation**
  The interconnect and any peripherals should be easy to implement within a reasonable time.

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

4.3 Interfaces



4.3.1 Master to Slave Interface



4.3.2 Multi-master Support

Design Goals

DG1. Foo

Bing

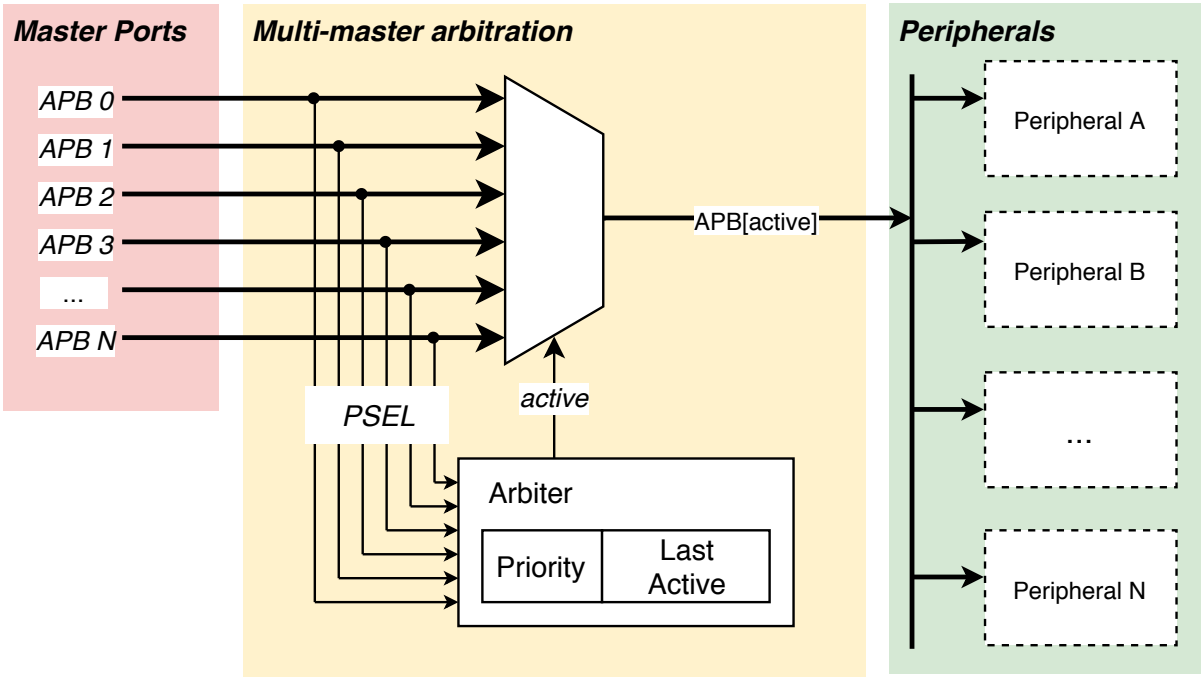
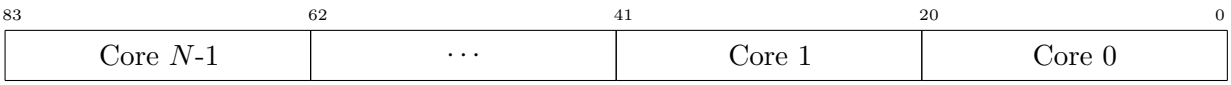


Figure 4.3: Foo

```
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,
```

Figure 4.4: Variable size inputs and outputs to the interconnect.



4.4 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.

Chapter 5

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.

5.1 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.

5.2 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 5.1.

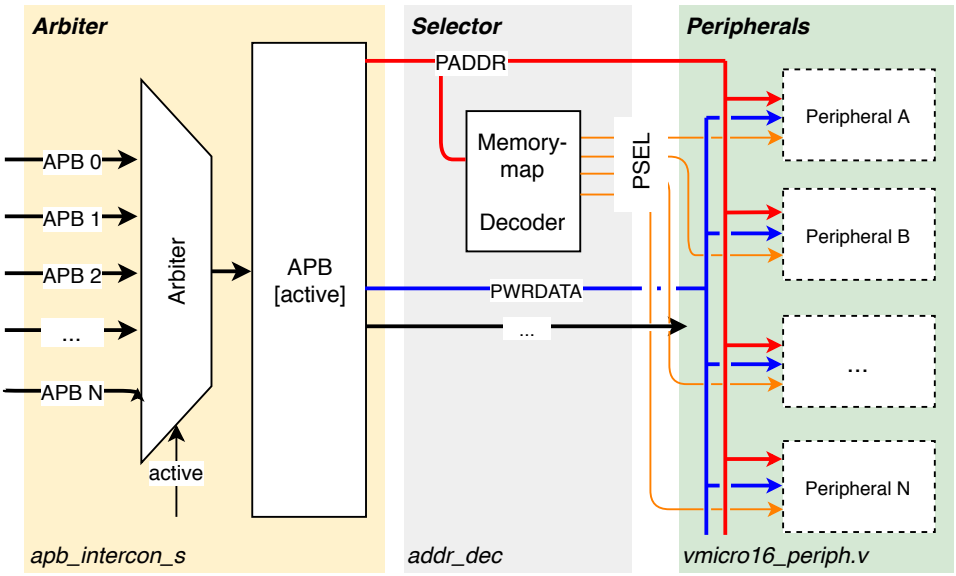


Figure 5.1: Schematic showing the address decoder (addr_dec) accepting the active PADDR signal and outputting PSEL chip enable signals to each peripheral.

5.2.1 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 [13, 15]. An example 4-bit comparator is shown below in Figure 5.2.

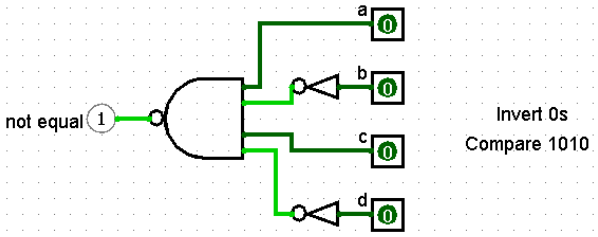


Figure 5.2: Example 4-bit binary comparator which compares the bits (a, b, c, d) to the constant value 1010. The 0s of the constant are inverted and then all are passed to a wide-AND.

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. 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 5.3 below shows a LUT based approach to address decoding which will utilise approximately 3 ALM modules (including error detection). This method of comparison (LUT based) is utilised

in the `addr_dec` module in `apb_intercon.v`.

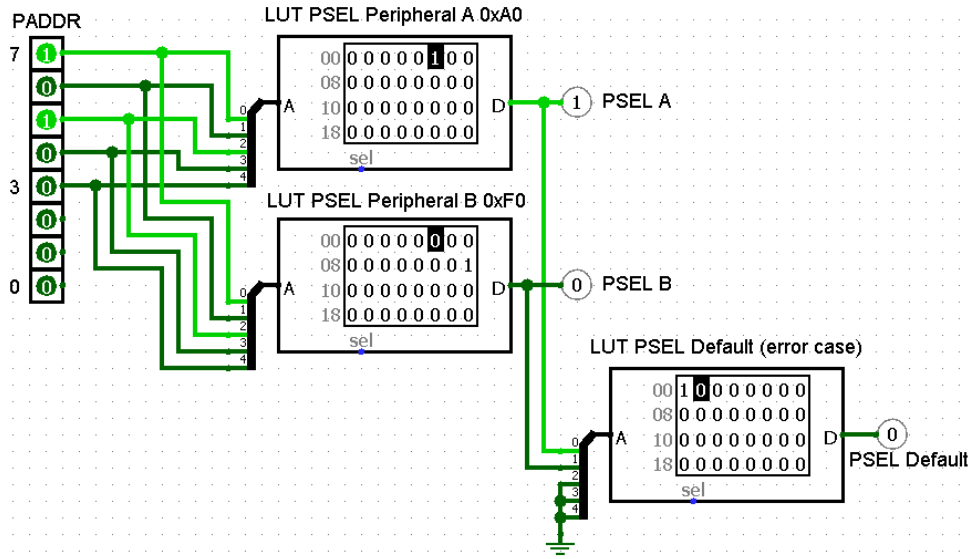


Figure 5.3: Bits [7:3] of an 8-bit PADDR signal are used as inputs to 5-bit LUTs to generate a PSEL signal. In addition, a default error case is shown allowing the address decoder to detect incorrect PADDR values (e.g. if no PSEL signals are generated).

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 [14]. 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 5.4 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.

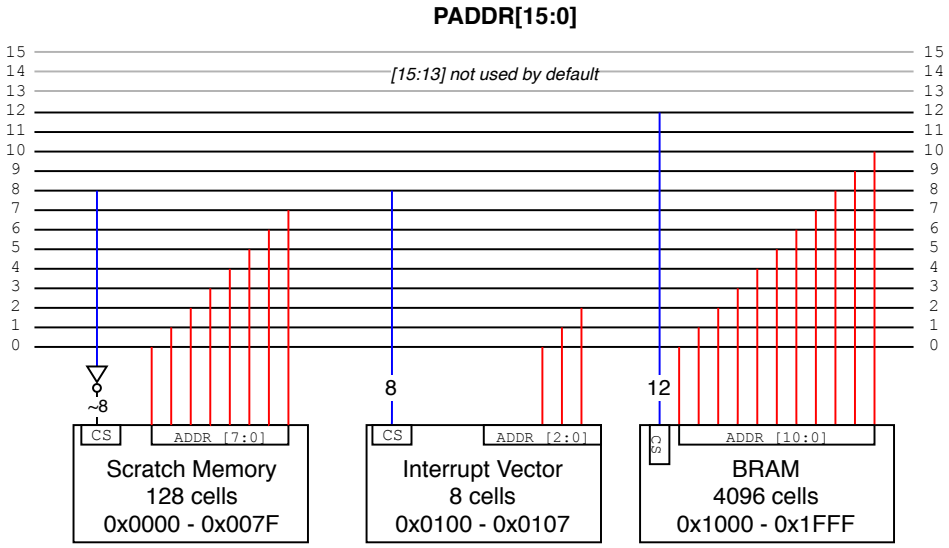


Figure 5.4: Partial address decoding used by the Vmicro16 SoC design. Each peripheral shown only needs to decode a signal bit to determine if it is enabled.

5.3 Memory Map

The system-on-chip's memory map is shown below in Figure 5.5. 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.

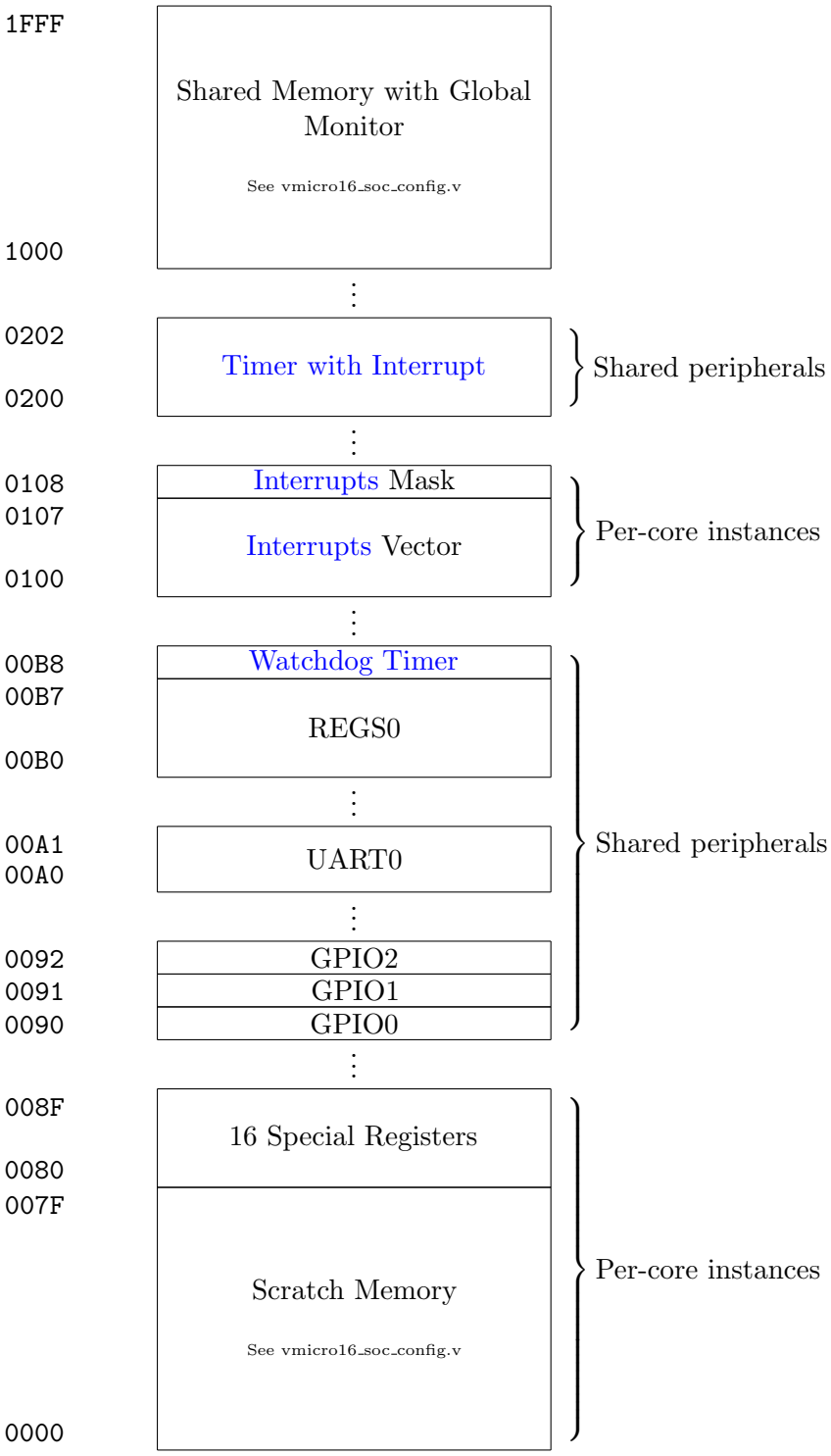


Figure 5.5: Memory map showing addresses of various memory sections.

Chapter 6

Interrupts

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

6.1 Why Interrupts?

Interrupts are used to enable asynchronous behaviour within a processor.

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

6.2 Hardware Implementation

6.2.1 Context Switching

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

When saving the state, to avoid clocking 128 bits (8 registers of 16 bits) into another register (which would increase timing delays and logic elements), a dedicated register set for the interrupt mode (`regs_isr`) is multiplexed with the normal mode register set (`regs`). Then depending on

the mode (identified by the register `regs_use_int`) the processor can easily switch between the two large states without significantly affecting timing.

The timing diagram in Figure 6.1 visually describes this process.

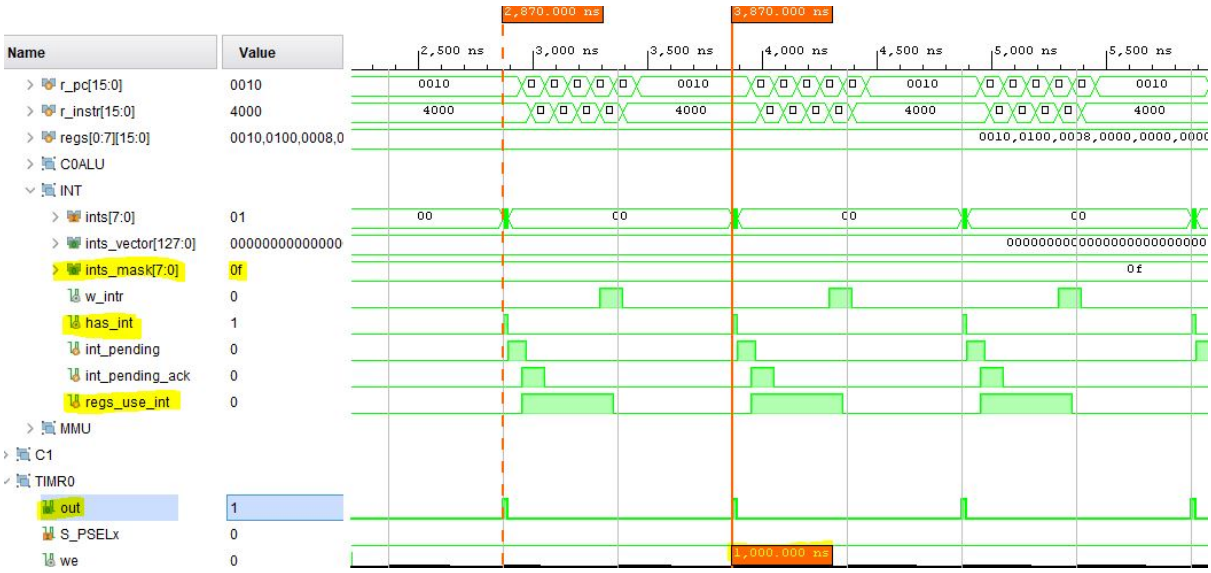


Figure 6.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return instruction (indicated by w_intr) the processor returns to normal mode and restores the normal state.

6.3 Software Interface

To enable software to

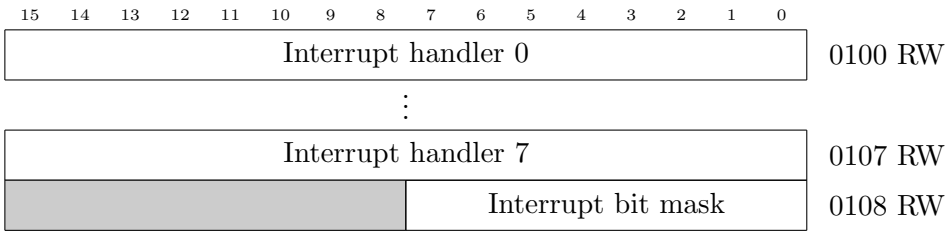


Figure 6.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

6.3.1 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.

6.3.2 Interrupt Mask (0x0108)

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

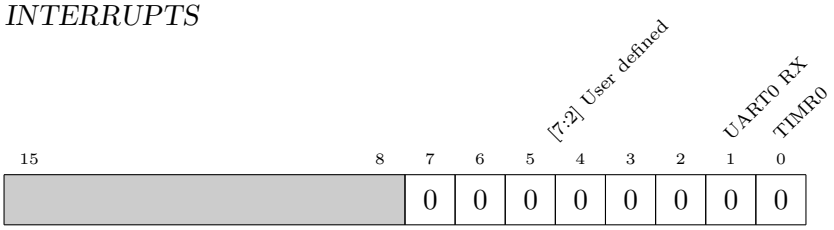


Figure 6.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

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.

6.3.3 Software Example

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

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

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

6.4 Design Improvements

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

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

Chapter 7

Peripherals

To provide user's with useful functionality, common system-on-chip peripherals were created. This section describes each peripheral and it's design decisions.

7.1 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.

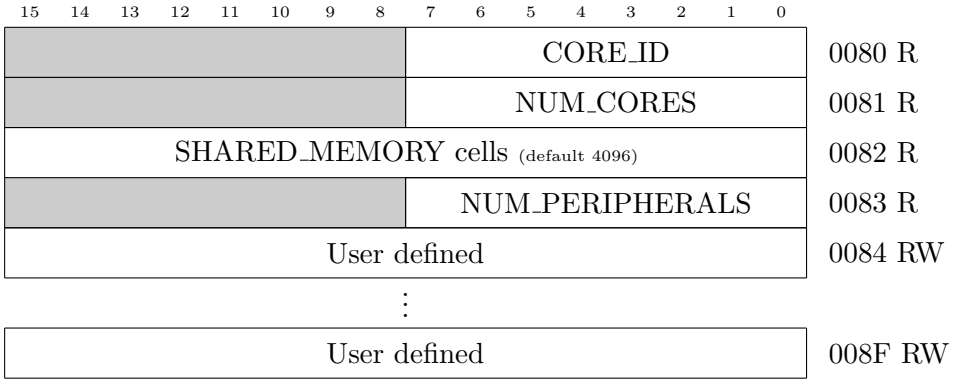


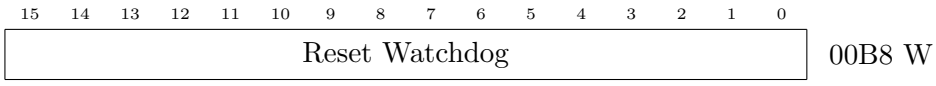
Figure 7.1: Vmicro16 Special Registers layout (0x0080 - 0x008F).

7.2 Watchdog Timer

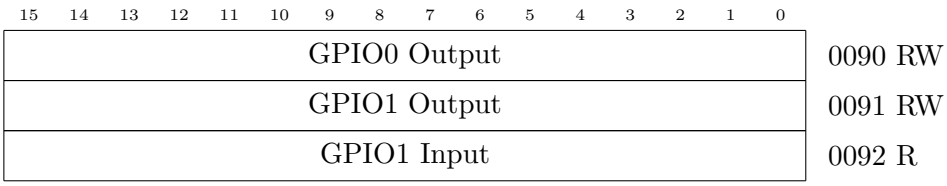
In any multi-threaded system there exists the possibility for a deadlock – a state where all threads are in a waiting state – and algorithm execution is forever blocked. This can occur either by poor software programming or incorrect thread arbitration by the processor. A common method of detecting a 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`.

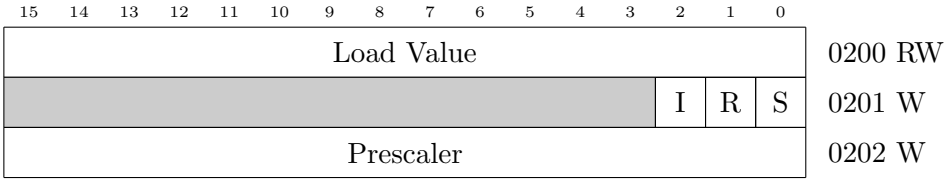
This peripheral is optional and can be enabled using the configuration parameters described in Configuration Options.



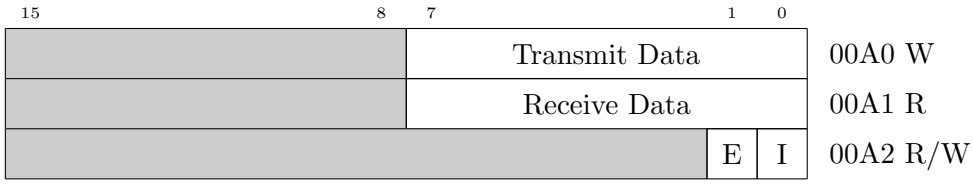
7.3 GPIO Interface



7.4 Timer with Interrupt



7.5 UART Interface



Chapter 8

System-on-Chip Layout

The Vmicro16 processor uses

Figure 8.1: •

Chapter 9

Analysis & Results

Appendix A

Configuration Options

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

A.1 SoC Options

Macro	Default	Purpose
CORES	4	Number of CPU cores in the SoC
SLAVES	7	Number of peripherals
DEF_USE_WATCHDOG		Enable watchdog module to detect deadlocks and infinite loops

Table A.1: SoC Configuration Options

A.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_GLOBAL_RESET	//	Enable synchronous reset logic
DEF_USE_REPROG	//	Programme instruction memory via UART0. Requires DEF_GLOBAL_RESET

Table A.2: Core Options

A.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 A.3: Peripheral Options

Appendix B

Code Listing

B.1 vmicro16_soc_config.v

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

```
1  // Configuration defines for the vmicro16_soc and vmicro16 cpu.
2
3  `ifndef VMICRO16_SOC_CONFIG_H
4  `define VMICRO16_SOC_CONFIG_H
5
6  `include "clog2.v"
7
8  `define FORMAL
9
10 `define CORES           8
11 `define SLAVES          8
12
13 ////////////////////////////////////////////////////////
14 // Core parameters
15 ////////////////////////////////////////////////////////
16 // Per core instruction memory
17 //  Set this to give each core its own instruction memory cache
18 `define DEF_CORE_HAS_INSTR_MEM
19
20 // Top level data width for registers, memory cells, bus widths
21 `define DATA_WIDTH      16
22
23 // Set this to use a workaround for the MMU's APB T2 clock
24 //`define FIX_T3
25
26 // Instruction memory (read only)
27 //   Must be large enough to support software program.
28 `ifdef DEF_CORE_HAS_INSTR_MEM
29     // 64 16-bit words per core
30     `define DEF_MEM_INSTR_DEPTH 64
31 `else
32     // 4096 16-bit words global
33     `define DEF_MEM_INSTR_DEPTH 4096
34 `endif
35
36 // Scratch memory (read/write) on each core.
37 //   See `DEF_MMU_TIMO_* defines for info.
38 `define DEF_MEM_SCRATCH_DEPTH 64
39
40 // Enables hardware multiplier and mult rr instruction
41 `define DEF_ALU_HW_MULT 1
42
43 // Enables global reset (requires more luts)
44 //`define DEF_GLOBAL_RESET
45
46 // Enable a watch dog timer to reset the soc if threadlocked
```

```
47  //`define DEF_USE_WATCHDOG
48
49  // Enables instruction memory programming via UART0
50  //`define DEF_USE_REPROG
51
52  `ifdef DEF_USE_REPROG
53      `ifndef DEF_GLOBAL_RESET
54          `error_DEF_USE_REPROG_requires_DEF_GLOBAL_RESET
55      `endif
56  `endif
57
58  ////////////////////////////////////////////////////////
59  // Memory mapping
60  ////////////////////////////////////////////////////////
61  `define APB_WIDTH       (2 + `clog2(`CORES) + `DATA_WIDTH)
62
63  `define APB_PSELX_GPIO0 0
64  `define APB_PSELX_UART0 1
65  `define APB_PSELX_REGS0 2
66  `define APB_PSELX_BRAM0 3
67  `define APB_PSELX_GPIO1 4
68  `define APB_PSELX_GPIO2 5
69  `define APB_PSELX_TIMR0 6
70  `define APB_PSELX_WDOG0 7
71
72  `define APB_GPIO0_PINS  8
73  `define APB_GPIO1_PINS  16
74  `define APB_GPIO2_PINS  8
75
76  // Shared memory words
77  `define APB_BRAM0_CELLS 4096
78
79  ////////////////////////////////////////////////////////
80  // Memory mapping
81  ////////////////////////////////////////////////////////
82  // TIM0
83  // Number of scratch memory cells per core
84  `define DEF_MMU_TIM0_CELLS  64
85  `define DEF_MMU_TIM0_S      16'h0000
86  `define DEF_MMU_TIM0_E      16'h007F
87  // SREG
88  `define DEF_MMU_SREG_S      16'h0080
89  `define DEF_MMU_SREG_E      16'h008F
90  // GPIO0
91  `define DEF_MMU_GPIO0_S     16'h0090
92  `define DEF_MMU_GPIO0_E     16'h0090
93  // GPIO1
94  `define DEF_MMU_GPIO1_S     16'h0091
95  `define DEF_MMU_GPIO1_E     16'h0091
96  // GPIO2
97  `define DEF_MMU_GPIO2_S     16'h0092
98  `define DEF_MMU_GPIO2_E     16'h0092
99  // UART0
100 `define DEF_MMU_UART0_S     16'h00A0
101 `define DEF_MMU_UART0_E     16'h00A1
102 // REGS0
103 `define DEF_MMU_REGS0_S     16'h00B0
104 `define DEF_MMU_REGS0_E     16'h00B7
105 // WDOG0
106 `define DEF_MMU_WDOG0_S     16'h00B8
107 `define DEF_MMU_WDOG0_E     16'h00B8
108 // BRAM0
109 `define DEF_MMU_BRAM0_S     16'h1000
110 `define DEF_MMU_BRAM0_E     16'h1fff
111 // TIMR0
112 `define DEF_MMU_TIMR0_S     16'h0200
113 `define DEF_MMU_TIMR0_E     16'h0202
114
115 ////////////////////////////////////////////////////////
116 // Interrupts
117 ////////////////////////////////////////////////////////
118 // Enable/disable interrupts
119 //   Disabling will free up resources for other features
120 `define DEF_ENABLE_INT
121 // Number of interrupt in signals
122 `define DEF_NUM_INT     8
123 // Default interrupt bitmask (0 = hidden, 1 = enabled)
124 `define DEF_INT_MASK    0
125 // Bit position of the TIMR0 interrupt signal
126 `define DEF_INT_TIMR0   0
127 // Interrupt vector memory location
128 `define DEF_MMU_INTSV_S     16'h0100
129 `define DEF_MMU_INTSV_E     16'h0107
130 // Interrupt vector memory location
131 `define DEF_MMU_INTSM_S     16'h0108
132 `define DEF_MMU_INTSM_E     16'h0108
133
134
135 `endif
```

B.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),
```

```
87
88          // DBUG
89          .dbug0   (LEDS[4])
90          //.dbug1   (LEDS[7:4])
91      );
92
93      assign LEDS[7:5] = {TXD, RXD, S6_3v3};
94
95      // SSD displays (split across 2 gpio ports 1 and 2)
96      wire [3:0] ssd_chars [0:5];
97      assign ssd_chars[0] = gpio1[3:0];
98      assign ssd_chars[1] = gpio1[7:4];
99      assign ssd_chars[2] = gpio1[11:8];
100     assign ssd_chars[3] = gpio1[15:12];
101     assign ssd_chars[4] = gpio2[3:0];
102     assign ssd_chars[5] = gpio2[7:4];
103     seven_display ssd_0 (.n(ssd_chars[0]), .segments (ssd0));
104     seven_display ssd_1 (.n(ssd_chars[1]), .segments (ssd1));
105     seven_display ssd_2 (.n(ssd_chars[2]), .segments (ssd2));
106     seven_display ssd_3 (.n(ssd_chars[3]), .segments (ssd3));
107     seven_display ssd_4 (.n(ssd_chars[4]), .segments (ssd4));
108     seven_display ssd_5 (.n(ssd_chars[5]), .segments (ssd5));
109
110 endmodule
```

B.3 vmicro16_soc.v

```
1   //
2   //
3
4   `include "vmicro16_soc_config.v"
5   `include "clog2.v"
6   `include "formal.v"
7
8   module pow_reset # (
9       parameter INIT  = 1,
10      parameter N     = 8
11  ) (
12      input       clk,
13      input       reset,
14      output reg  resethold
15  );
16      initial resethold = INIT ? (N-1) : 0;
17
18      always @(*)
19          resethold = |hold;
20
21      reg [`clog2(N)-1:0] hold = (N-1);
22      always @(posedge clk)
23          if (reset)
24              hold <= N-1;
25          else
26              if (hold)
27                  hold <= hold - 1;
28  endmodule
29
30  // Vmicro16 multi-core SoC with various peripherals
31  // and interrupts
32  module vmicro16_soc (
33      input clk,
34      input reset,
35
36      // UART0
37      input                           uart_rx,
38      output                          uart_tx,
39      //
40      output [`APB_GPIO0_PINS-1:0]    gpio0,
41      output [`APB_GPIO1_PINS-1:0]    gpio1,
42      output [`APB_GPIO2_PINS-1:0]    gpio2,
43      //
44      output                          halt,
45      //
46      output      [`CORES-1:0]        dbug0,
47      output      [`CORES*8-1:0]      dbug1
48  );
49      wire [`CORES-1:0] w_halt;
50      assign halt = &w_halt;
51
52      assign dbug0 = w_halt;
53
54      // Watchdog reset pulse signal.
55      //   Passed to pow_reset to generate a longer reset pulse
56      wire wdreset;
57      wire prog_prog;
58
59      // soft register reset hold for brams and registers
60      wire soft_reset;
61      `ifdef DEF_GLOBAL_RESET
```

```
62          pow_reset # (
63              .INIT       (1),
64              .N          (8)
65          ) por_inst (
66              .clk        (clk),
67              `ifdef DEF_USE_WATCHDOG
68              .reset      (reset | wdreset | prog_prog),
69              `else
70              .reset      (reset),
71              `endif
72              .resethold  (soft_reset)
73          );
74      `else
75          assign soft_reset = 0;
76      `endif
77
78      // Peripherals (master to slave)
79       wire [`APB_WIDTH-1:0]          M_PADDR;
80       wire                           M_PWRITE;
81       wire [`SLAVES-1:0]             M_PSELx;  // not shared
82       wire                           M_PENABLE;
83       wire [`DATA_WIDTH-1:0]         M_PWDATA;
84       wire [`SLAVES*`DATA_WIDTH-1:0] M_PRDATA; // input to intercon
85       wire [`SLAVES-1:0]             M_PREADY; // input
86
87      // Master apb interfaces
88       wire [`CORES*`APB_WIDTH-1:0]   w_PADDR;
89       wire [`CORES-1:0]              w_PWRITE;
90       wire [`CORES-1:0]              w_PSELx;
91       wire [`CORES-1:0]              w_PENABLE;
92       wire [`CORES*`DATA_WIDTH-1:0]  w_PWDATA;
93       wire [`CORES*`DATA_WIDTH-1:0]  w_PRDATA;
94       wire [`CORES-1:0]              w_PREADY;
95
96      // Interrupts
97  `ifdef DEF_ENABLE_INT
98      wire [`DEF_NUM_INT-1:0]             ints;
99      wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_data;
100     assign ints[7:1] = 0;
101     assign ints_data[`DEF_NUM_INT*`DATA_WIDTH-1:`DATA_WIDTH] =
102                 {`DEF_NUM_INT*(`DATA_WIDTH-1){1'b0}};
103 `endif
104
105     apb_intercon_s # (
106         .MASTER_PORTS   (`CORES),
107         .SLAVE_PORTS    (`SLAVES),
108         .BUS_WIDTH      (`APB_WIDTH),
109         .DATA_WIDTH     (`DATA_WIDTH),
110         .HAS_PSELX_ADDR (1)
111     ) apb (
112         .clk        (clk),
113         .reset      (soft_reset),
114         // APB master to slave
115         .S_PADDR    (w_PADDR),
116         .S_PWRITE   (w_PWRITE),
117         .S_PSELx    (w_PSELx),
118         .S_PENABLE  (w_PENABLE),
119         .S_PWDATA   (w_PWDATA),
120         .S_PRDATA   (w_PRDATA),
121         .S_PREADY   (w_PREADY),
122         // shared bus
123         .M_PADDR    (M_PADDR),
124         .M_PWRITE   (M_PWRITE),
125         .M_PSELx    (M_PSELx),
126         .M_PENABLE  (M_PENABLE),
127         .M_PWDATA   (M_PWDATA),
128         .M_PRDATA   (M_PRDATA),
129         .M_PREADY   (M_PREADY)
130     );
131
132 `ifdef DEF_USE_WATCHDOG
133     vmicro16_watchdog_apb # (
134         .BUS_WIDTH  (`APB_WIDTH),
135         .NAME       ("WDOG0")
136     ) wdog0_apb (
137         .clk        (clk),
138         .reset      (),
139         // apb slave to master interface
140         .S_PADDR    (),
141         .S_PWRITE   (M_PWRITE),
142         .S_PSELx    (M_PSELx[`APB_PSELX_WDOG0]),
143         .S_PENABLE  (M_PENABLE),
144         .S_PWDATA   (),
145         .S_PRDATA   (),
146         .S_PREADY   (M_PREADY[`APB_PSELX_WDOG0]),
147
148         .wdreset    (wdreset)
149     );
150 `endif
151
152     vmicro16_gpio_apb # (
```

```
153         .BUS_WIDTH  (`APB_WIDTH),
154         .DATA_WIDTH (`DATA_WIDTH),
155         .PORTS      (`APB_GPIO0_PINS),
156         .NAME       ("GPIO0")
157     ) gpio0_apb (
158         .clk        (clk),
159         .reset      (soft_reset),
160         // apb slave to master interface
161         .S_PADDR    (M_PADDR),
162         .S_PWRITE   (M_PWRITE),
163         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO0]),
164         .S_PENABLE  (M_PENABLE),
165         .S_PWDATA   (M_PWDATA),
166         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO0*`DATA_WIDTH +: `DATA_WIDTH]),
167         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
168         .gpio       (gpio0)
169     );
170
171     // GPIO1 for Seven segment displays (16 pin)
172     vmicro16_gpio_apb # (
173         .BUS_WIDTH  (`APB_WIDTH),
174         .DATA_WIDTH (`DATA_WIDTH),
175         .PORTS      (`APB_GPIO1_PINS),
176         .NAME       ("GPIO1")
177     ) gpio1_apb (
178         .clk        (clk),
179         .reset      (soft_reset),
180         // apb slave to master interface
181         .S_PADDR    (M_PADDR),
182         .S_PWRITE   (M_PWRITE),
183         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO1]),
184         .S_PENABLE  (M_PENABLE),
185         .S_PWDATA   (M_PWDATA),
186         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO1*`DATA_WIDTH +: `DATA_WIDTH]),
187         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
188         .gpio       (gpio1)
189     );
190
191     // GPIO2 for Seven segment displays (8 pin)
192     vmicro16_gpio_apb # (
193         .BUS_WIDTH  (`APB_WIDTH),
194         .DATA_WIDTH (`DATA_WIDTH),
195         .PORTS      (`APB_GPIO2_PINS),
196         .NAME       ("GPIO2")
197     ) gpio2_apb (
198         .clk        (clk),
199         .reset      (soft_reset),
200         // apb slave to master interface
201         .S_PADDR    (M_PADDR),
202         .S_PWRITE   (M_PWRITE),
203         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO2]),
204         .S_PENABLE  (M_PENABLE),
205         .S_PWDATA   (M_PWDATA),
206         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO2*`DATA_WIDTH +: `DATA_WIDTH]),
207         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
208         .gpio       (gpio2)
209     );
210
211     apb_uart_tx # (
212         .DATA_WIDTH (8),
213         .ADDR_EXP   (4) //2^^4 = 16 FIFO words
214     ) uart0_apb (
215         .clk        (clk),
216         .reset      (soft_reset),
217         // apb slave to master interface
218         .S_PADDR    (M_PADDR),
219         .S_PWRITE   (M_PWRITE),
220         .S_PSELx    (M_PSELx[`APB_PSELX_UART0]),
221         .S_PENABLE  (M_PENABLE),
222         .S_PWDATA   (M_PWDATA),
223         .S_PRDATA   (M_PRDATA[`APB_PSELX_UART0*`DATA_WIDTH +: `DATA_WIDTH]),
224         .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
225         // uart wires
226         .tx_wire    (uart_tx),
227         .rx_wire    ()
228     );
229
230     timer_apb timr0 (
231         .clk        (clk),
232         .reset      (soft_reset),
233         // apb slave to master interface
234         .S_PADDR    (M_PADDR),
235         .S_PWRITE   (M_PWRITE),
236         .S_PSELx    (M_PSELx[`APB_PSELX_TIMR0]),
237         .S_PENABLE  (M_PENABLE),
238         .S_PWDATA   (M_PWDATA),
239         .S_PRDATA   (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
240         .S_PREADY   (M_PREADY[`APB_PSELX_TIMR0])
241         //
242         `ifdef DEF_ENABLE_INT
243         ,.out       (ints     [`DEF_INT_TIMR0]),
```

```
244             .int_data  (ints_data[`DEF_INT_TIMRO*`DATA_WIDTH +: `DATA_WIDTH])
245         `endif
246     );
247
248     // Shared register set for system-on-chip info
249     // RO = number of cores
250     vmicro16_regs_apb # (
251         .BUS_WIDTH          (`APB_WIDTH),
252         .DATA_WIDTH         (`DATA_WIDTH),
253         .CELL_DEPTH         (8),
254         .PARAM_DEFAULTS_RO  (`CORES),
255         .PARAM_DEFAULTS_R1  (`SLAVES)
256     ) regs0_apb (
257         .clk        (clk),
258         .reset      (soft_reset),
259         // apb slave to master interface
260         .S_PADDR    (M_PADDR),
261         .S_PWRITE   (M_PWRITE),
262         .S_PSELx    (M_PSELx[`APB_PSELX_REGS0]),
263         .S_PENABLE  (M_PENABLE),
264         .S_PWDATA   (M_PWDATA),
265         .S_PRDATA   (M_PRDATA[`APB_PSELX_REGS0*`DATA_WIDTH +: `DATA_WIDTH]),
266         .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
267     );
268
269     vmicro16_bram_ex_apb # (
270         .BUS_WIDTH    (`APB_WIDTH),
271         .MEM_WIDTH    (`DATA_WIDTH),
272         .MEM_DEPTH    (`APB_BRAM0_CELLS),
273         .CORE_ID_BITS (`clog2(`CORES))
274     ) bram_apb (
275         .clk        (clk),
276         .reset      (soft_reset),
277         // apb slave to master interface
278         .S_PADDR    (M_PADDR),
279         .S_PWRITE   (M_PWRITE),
280         .S_PSELx    (M_PSELx[`APB_PSELX_BRAM0]),
281         .S_PENABLE  (M_PENABLE),
282         .S_PWDATA   (M_PWDATA),
283         .S_PRDATA   (M_PRDATA[`APB_PSELX_BRAM0*`DATA_WIDTH +: `DATA_WIDTH]),
284         .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
285     );
286
287     // There must be atleast 1 core
288     `static_assert(`CORES > 0)
289     `static_assert(`DEF_MEM_INSTR_DEPTH > 0)
290     `static_assert(`DEF_MMU_TIM0_CELLS > 0)
291
292
293     // Single instruction memory
294 `ifndef DEF_CORE_HAS_INSTR_MEM
295     // slave input/outputs from interconnect
296     wire [`APB_WIDTH-1:0]          instr_M_PADDR;
297     wire                           instr_M_PWRITE;
298     wire [1-1:0]                   instr_M_PSELx;  // not shared
299     wire                           instr_M_PENABLE;
300     wire [`DATA_WIDTH-1:0]         instr_M_PWDATA;
301     wire [1*`DATA_WIDTH-1:0]       instr_M_PRDATA; // slave response
302     wire [1-1:0]                   instr_M_PREADY; // slave response
303
304     // Master apb interfaces
305     wire [`CORES*`APB_WIDTH-1:0]   instr_w_PADDR;
306     wire [`CORES-1:0]              instr_w_PWRITE;
307     wire [`CORES-1:0]              instr_w_PSELx;
308     wire [`CORES-1:0]              instr_w_PENABLE;
309     wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PWDATA;
310     wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PRDATA;
311     wire [`CORES-1:0]              instr_w_PREADY;
312
313     `ifdef DEF_USE_REPROG
314         wire [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] prog_addr;
315         wire [`DATA_WIDTH-1:0] prog_data;
316         wire prog_we;
317         uart_prog rom_prog (
318             .clk        (clk),
319             .reset      (reset | wdreset),
320             // input stream
321             .uart_rx    (uart_rx),
322             // programmer
323             .addr       (prog_addr),
324             .data       (prog_data),
325             .we         (prog_we),
326             .prog       (prog_prog)
327         );
328     `endif
329
330     `ifdef DEF_USE_REPROG
331         vmicro16_bram_prog_apb
332     `else
333         vmicro16_bram_apb
334     `endif
```

```
335     # (
336         .BUS_WIDTH      (`APB_WIDTH),
337         .MEM_WIDTH      (`DATA_WIDTH),
338         .MEM_DEPTH      (`DEF_MEM_INSTR_DEPTH),
339         .USE_INITS      (1),
340         .NAME           ("INSTR_ROM_G")
341     ) instr_rom_apb (
342         .clk            (clk),
343         .reset          (reset),
344         .S_PADDR        (instr_M_PADDR),
345         .S_PWRITE       (0),
346         .S_PSELx        (instr_M_PSELx),
347         .S_PENABLE      (instr_M_PENABLE),
348         .S_PWDATA       (0),
349         .S_PRDATA       (instr_M_PRDATA),
350         .S_PREADY       (instr_M_PREADY)
351 
352         `ifdef DEF_USE_REPROG
353             ,
354             .addr       (prog_addr),
355             .data       (prog_data),
356             .we         (prog_we),
357             .prog       (prog_prog)
358         `endif
359     );
360 
361     apb_intercon_s # (
362         .MASTER_PORTS   (`CORES),
363         .SLAVE_PORTS    (1),
364         .BUS_WIDTH      (`APB_WIDTH),
365         .DATA_WIDTH     (`DATA_WIDTH),
366         .HAS_PSELX_ADDR (0)
367     ) apb_instr_intercon (
368         .clk        (clk),
369         .reset      (soft_reset),
370         // APB master from cores
371         // master
372         .S_PADDR    (instr_w_PADDR),
373         .S_PWRITE   (instr_w_PWRITE),
374         .S_PSELx    (instr_w_PSELx),
375         .S_PENABLE  (instr_w_PENABLE),
376         .S_PWDATA   (instr_w_PWDATA),
377         .S_PRDATA   (instr_w_PRDATA),
378         .S_PREADY   (instr_w_PREADY),
379         // shared bus slaves
380         // slave outputs
381         .M_PADDR    (instr_M_PADDR),
382         .M_PWRITE   (instr_M_PWRITE),
383         .M_PSELx    (instr_M_PSELx),
384         .M_PENABLE  (instr_M_PENABLE),
385         .M_PWDATA   (instr_M_PWDATA),
386         .M_PRDATA   (instr_M_PRDATA),
387         .M_PREADY   (instr_M_PREADY)
388     );
389 `endif
390 
391     genvar i;
392     generate for(i = 0; i < `CORES; i = i + 1) begin : cores
393 
394         vmicro16_core # (
395             .CORE_ID            (i),
396             .DATA_WIDTH         (`DATA_WIDTH),
397 
398             .MEM_INSTR_DEPTH    (`DEF_MEM_INSTR_DEPTH),
399             .MEM_SCRATCH_DEPTH  (`DEF_MMU_TIMO_CELLS)
400         ) c1 (
401             .clk        (clk),
402             .reset      (soft_reset),
403 
404             // debug
405             .halt       (w_halt[i]),
406 
407             // interrupts
408             .ints       (ints),
409             .ints_data  (ints_data),
410 
411             // Output master port 1
412             .w_PADDR    (w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
413             .w_PWRITE   (w_PWRITE  [i]                           ),
414             .w_PSELx    (w_PSELx   [i]                           ),
415             .w_PENABLE  (w_PENABLE [i]                           ),
416             .w_PWDATA   (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
417             .w_PRDATA   (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
418             .w_PREADY   (w_PREADY  [i]                           )
419 
420 `ifndef DEF_CORE_HAS_INSTR_MEM
421             // APB instruction rom
422             , // Output master port 2
423             .w2_PADDR   (instr_w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
424             //.w2_PWRITE  (instr_w_PWRITE  [i]                           ),
425             .w2_PSELx   (instr_w_PSELx   [i]                           ),
```

```
426             .w2_PENABLE (instr_w_PENABLE [i]                          ),
427             //.w2_PWDATA  (instr_w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
428             .w2_PRDATA  (instr_w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
429             .w2_PREADY  (instr_w_PREADY  [i]                          )
430 `endif
431         );
432     end
433     endgenerate
434 
435 
436     ////////////////////////////////////////////////////
437     // Formal Verification
438     ////////////////////////////////////////////////////
439     `ifdef FORMAL
440     wire all_halted = &w_halt;
441     ////////////////////////////////////////////////////
442     // Count number of clocks each core is spending on
443     //   bus transactions
444     ////////////////////////////////////////////////////
445     reg [15:0] bus_core_times       [0:`CORES-1];
446     reg [15:0] core_work_times      [0:`CORES-1];
447     reg [15:0] instr_fetch_times    [0:`CORES-1];
448     integer i2;
449     initial
450         for(i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
451             bus_core_times[i2] = 0;
452             core_work_times[i2] = 0;
453         end
454 
455     // total bus time
456     generate
457         genvar g2;
458         for (g2 = 0; g2 < `CORES; g2 = g2 + 1) begin : formal_for_times
459               always @(posedge clk) begin
460                     if (w_PSELx[g2])
461                         bus_core_times[g2] <= bus_core_times[g2] + 1;
462 
463                     // Core working time
464                     `ifndef DEF_CORE_HAS_INSTR_MEM
465                         if (!w_PSELx[g2] && !instr_w_PSELx[g2])
466                     `else
467                         if (!w_PSELx[g2])
468                     `endif
469                             if (!w_halt[g2])
470                                 core_work_times[g2] <= core_work_times[g2] + 1;
471 
472               end
473           end
474     endgenerate
475 
476     reg [15:0] bus_time_average = 0;
477     reg [15:0] bus_reqs_average = 0;
478     reg [15:0] fetch_time_average = 0;
479     reg [15:0] work_time_average = 0;
480     //
481     always @(all_halted) begin
482         for (i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
483             bus_time_average   = bus_time_average   + bus_core_times[i2];
484             bus_reqs_average   = bus_reqs_average   + bus_core_reqs_count[i2];
485             work_time_average  = work_time_average  + core_work_times[i2];
486             fetch_time_average = fetch_time_average + instr_fetch_times[i2];
487         end
488 
489         bus_time_average   = bus_time_average   / `CORES;
490         bus_reqs_average   = bus_reqs_average   / `CORES;
491         work_time_average  = work_time_average  / `CORES;
492         fetch_time_average = fetch_time_average / `CORES;
493     end
494 
495     ////////////////////////////////////////////////////
496     // Count number of bus requests per core
497     ////////////////////////////////////////////////////
498     // 1 clock delay of w_PSELx
499     reg [`CORES-1:0] bus_core_reqs_last;
500     // rising edges of each
501     wire [`CORES-1:0] bus_core_reqs_real;
502     // storage for counters for each core
503     reg [15:0] bus_core_reqs_count [0:`CORES-1];
504     initial
505         for(i2 = 0; i2 < `CORES; i2 = i2 + 1)
506             bus_core_reqs_count[i2] = 0;
507 
508     // 1 clk delay to detect rising edge
509     always @(posedge clk)
510         bus_core_reqs_last <= w_PSELx;
511 
512     generate
513         genvar g3;
514               for (g3 = 0; g3 < `CORES; g3 = g3 + 1) begin : formal_for_reqs
515               // Detect new reqs for each core
516               assign bus_core_reqs_real[g3] = w_PSELx[g3] >
```

```
                                                        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
```

B.4 vmicro16_periph.v

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);
```

```
46              timer <= CLK_HZ;
47          end else begin
48              timer <= timer - 1;
49          end
50  endmodule
51
52  module timer_apb # (
53      parameter CLK_HZ = 50_000_000
54  ) (
55      input clk,
56      input reset,
57
58      input clk_en,
59
60      // 0 16-bit value   R/W
61      // 1 16-bit control R    b0 = start, b1 = reset
62      // 2 16-bit prescaler
63      input       [1:0]                S_PADDR,
64
65      input                            S_PWRITE,
66      input                            S_PSELx,
67      input                            S_PENABLE,
68      input       [`DATA_WIDTH-1:0]    S_PWDATA,
69
70      output reg  [`DATA_WIDTH-1:0]    S_PRDATA,
71      output                           S_PREADY,
72
73      output out,
74      output [`DATA_WIDTH-1:0] int_data
75  );
76      //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
77      assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
78      wire   en       = (S_PSELx & S_PENABLE);
79      wire   we       = (en & S_PWRITE);
80
81      reg [`DATA_WIDTH-1:0] r_counter = 0;
82      reg [`DATA_WIDTH-1:0] r_load = 0;
83      reg [`DATA_WIDTH-1:0] r_pres = 0;
84      reg [`DATA_WIDTH-1:0] r_ctrl = 0;
85
86      localparam CTRL_START = 0;
87      localparam CTRL_RESET = 1;
88      localparam CTRL_INT   = 2;
89
90      localparam ADDR_LOAD = 2'b00;
91      localparam ADDR_CTRL = 2'b01;
92      localparam ADDR_PRES = 2'b10;
93
94      always @(*) begin
95          S_PRDATA = 0;
96          if (en)
97              case(S_PADDR)
98                  ADDR_LOAD: S_PRDATA = r_counter;
99                  ADDR_CTRL: S_PRDATA = r_ctrl;
100                 //ADDR_CTRL: S_PRDATA = r_pres;
101                 default:   S_PRDATA = 0;
102             endcase
103     end
104
105     // prescaler counts from r_pres to 0, emitting a stb signal
106     //   to enable the r_counter step
107     reg [`DATA_WIDTH-1:0] r_pres_counter = 0;
108     wire counter_en = (r_pres_counter == 0);
109     always @(posedge clk)
110         if (r_pres_counter == 0)
111             r_pres_counter <= r_pres;
112         else
113             r_pres_counter <= r_pres_counter - 1;
114
115     always @(posedge clk)
116         if (we)
117             case(S_PADDR)
118                 // Write to the load register:
119                 //   Set load register
120                 //   Set counter register
121                 ADDR_LOAD: begin
122                     r_load          <= S_PWDATA;
123                     r_counter       <= S_PWDATA;
124                     $display($time, "\t\ttimr0: WRITE LOAD: %h", S_PWDATA);
125                 end
126                 ADDR_CTRL: begin
127                     r_ctrl   <= S_PWDATA;
128                     $display($time, "\t\ttimr0: WRITE CTRL: %h", S_PWDATA);
129                 end
130                 ADDR_PRES: begin
131                     r_pres   <= S_PWDATA;
132                     $display($time, "\t\ttimr0: WRITE PRES: %h", S_PWDATA);
133                 end
134             endcase
135         else
136             if (r_ctrl[CTRL_START]) begin
```

```
137                 if (r_counter == 0)
138                     r_counter <= r_load;
139                 else if(counter_en)
140                     r_counter <= r_counter -1;
141             end else if (r_ctrl[CTRL_RESET])
142                 r_counter <= r_load;
143
144     // generate the output pulse when r_counter == 0
145     //   out = (counter reached zero && counter started)
146     assign out      = (r_counter == 0) && r_ctrl[CTRL_START]; // && r_ctrl[CTRL_INT];
147     assign int_data = {`DATA_WIDTH{1'b1}};
148 endmodule
149
150
151 // APB wrapped programmable vmicro16_bram
152 module vmicro16_bram_prog_apb # (
153     parameter BUS_WIDTH    = 16,
154     parameter MEM_WIDTH    = 16,
155     parameter MEM_DEPTH    = 64,
156     parameter APB_PADDR    = 0,
157     parameter USE_INITS    = 0,
158     parameter NAME         = "BRAMPROG",
159     parameter CORE_ID      = 0
160 ) (
161     input clk,
162     input reset,
163     // APB Slave to master interface
164     input  [`clog2(MEM_DEPTH)-1:0]  S_PADDR,
165     input                           S_PWRITE,
166     input                           S_PSELx,
167     input                           S_PENABLE,
168     input  [BUS_WIDTH-1:0]          S_PWDATA,
169
170     output [BUS_WIDTH-1:0]          S_PRDATA,
171     output                          S_PREADY,
172
173     // interface to program the instruction memory
174     input     [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] addr,
175     input     [`DATA_WIDTH-1:0]                  data,
176     input                                        we,
177     input                                        prog
178 );
179     wire [MEM_WIDTH-1:0] mem_out;
180
181     assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out : 16'h0000;
182     assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1    : 1'b0;
183     wire s_we = (S_PSELx & S_PENABLE & S_PWRITE);
184
185     wire [`clog2(`DEF_MEM_INSTR_DEPTH)-1:0] mem_addr = we ? addr : S_PADDR;
186     wire [`DATA_WIDTH-1:0]                  mem_data = we ? data : S_PWDATA;
187     wire                                    mem_we   = we | s_we;
188
189     vmicro16_bram # (
190         .MEM_WIDTH  (MEM_WIDTH),
191         .MEM_DEPTH  (MEM_DEPTH),
192         .NAME       ("BRAMPROG"),
193         .USE_INITS  (0),
194         .CORE_ID    (-1)
195     ) bram_apb (
196         .clk        (clk),
197         .reset      (reset),
198
199         .mem_addr   (mem_addr),
200         .mem_in     (mem_data),
201         .mem_we     (mem_we),
202         .mem_out    (mem_out)
203     );
204 endmodule
205
206 // APB wrapped vmicro16_bram
207 module vmicro16_bram_apb # (
208     parameter BUS_WIDTH    = 16,
209     parameter MEM_WIDTH    = 16,
210     parameter MEM_DEPTH    = 64,
211     parameter APB_PADDR    = 0,
212     parameter USE_INITS    = 0,
213     parameter NAME         = "BRAM",
214     parameter CORE_ID      = 0
215 ) (
216     input clk,
217     input reset,
218     // APB Slave to master interface
219     input  [`clog2(MEM_DEPTH)-1:0]  S_PADDR,
220     input                           S_PWRITE,
221     input                           S_PSELx,
222     input                           S_PENABLE,
223     input  [BUS_WIDTH-1:0]          S_PWDATA,
224
225     output [BUS_WIDTH-1:0]          S_PRDATA,
226     output                          S_PREADY
227 );
```

```
228     wire [MEM_WIDTH-1:0] mem_out;
229
230     assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out : 16'h0000;
231     assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1    : 1'b0;
232     assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
233
234     always @(*)
235         if (S_PSELx && S_PENABLE)
236             $display($time, "\t\t%s => %h", NAME, mem_out);
237
238     always @(posedge clk)
239         if (we)
240             $display($time, "\t\t%s[%h] <= %h", NAME,
241                 S_PADDR, S_PWDATA);
242
243     vmicro16_bram # (
244         .MEM_WIDTH  (MEM_WIDTH),
245         .MEM_DEPTH  (MEM_DEPTH),
246         .NAME       (NAME),
247         .USE_INITS  (1),
248         .CORE_ID    (-1)
249     ) bram_apb (
250         .clk        (clk),
251         .reset      (reset),
252
253         .mem_addr   (S_PADDR),
254         .mem_in     (S_PWDATA),
255         .mem_we     (we),
256         .mem_out    (mem_out)
257     );
258 endmodule
259
260 // Shared memory with hardware monitor (LWEX/SWEX)
261 module vmicro16_bram_ex_apb # (
262     parameter BUS_WIDTH    = 16,
263     parameter MEM_WIDTH    = 16,
264     parameter MEM_DEPTH    = 64,
265     parameter CORE_ID_BITS = 3,
266     parameter SWEX_SUCCESS = 16'h0000,
267     parameter SWEX_FAIL    = 16'h0001
268 ) (
269     input clk,
270     input reset,
271
272     // |19    |18    |16             |15          0|
273     // | LWEX | SWEX | 3 bit CORE_ID |     S_PADDR |
274     input  [`APB_WIDTH-1:0]         S_PADDR,
275
276     input                           S_PWRITE,
277     input                           S_PSELx,
278     input                           S_PENABLE,
279     input  [MEM_WIDTH-1:0]          S_PWDATA,
280
281     output reg [MEM_WIDTH-1:0]      S_PRDATA,
282     output                          S_PREADY
283 );
284     // exclusive flag checks
285     wire [MEM_WIDTH-1:0] mem_out;
286     reg                  swex_success = 0;
287
288     localparam ADDR_BITS = `clog2(MEM_DEPTH);
289
290     // hack to create a 1 clock delay to S_PREADY
291     // for bram to be ready
292     reg cdelay = 1;
293     always @(posedge clk)
294         if (S_PSELx)
295             cdelay <= 0;
296         else
297             cdelay <= 1;
298
299     //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
300     assign S_PREADY = (S_PSELx & S_PENABLE & (!cdelay)) ? 1'b1       : 1'b0;
301     assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
302     wire   en       = (S_PSELx & S_PENABLE);
303
304     // Similar to:
305     //   http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204f/Cihbghef.html
306
307     // mem_wd is the CORE_ID sent in bits [18:16]
308     localparam TOP_BIT_INDEX         = `APB_WIDTH -1;
309     localparam PADDR_CORE_ID_MSB     = TOP_BIT_INDEX - 2;
310     localparam PADDR_CORE_ID_LSB     = PADDR_CORE_ID_MSB - (CORE_ID_BITS-1);
311
312     // [LWEX, CORE_ID, mem_addr] from S_PADDR
313     wire                    lwex        = S_PADDR[TOP_BIT_INDEX];
314     wire                    swex        = S_PADDR[TOP_BIT_INDEX-1];
315     wire [CORE_ID_BITS-1:0] core_id     = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
316     // CORE_ID to write to ex_flags register
317     wire [ADDR_BITS-1:0]    mem_addr    = S_PADDR[ADDR_BITS-1:0];
318
```

```
319     wire [CORE_ID_BITS:0]   ex_flags_read;
320     wire                    is_locked      = |ex_flags_read;
321     wire                    is_locked_self = is_locked && (core_id == (ex_flags_read-1));
322
323     // Check exclusive access flags
324     always @(*) begin
325         swex_success = 0;
326         if (en)
327             // bug!
328             if (!swex && !lwex)
329                 swex_success = 1;
330             else if (swex)
331                 if (is_locked && !is_locked_self)
332                     // someone else has locked it
333                     swex_success = 0;
334                 else if (is_locked && is_locked_self)
335                     swex_success = 1;
336     end
337
338     always @(*)
339         if (swex)
340             if (swex_success)
341                 S_PRDATA = SWEX_SUCCESS;
342             else
343                 S_PRDATA = SWEX_FAIL;
344         else
345             S_PRDATA = mem_out;
346
347     wire reg_we = en && ((lwex && !is_locked)
348                       || (swex && swex_success));
349
350     reg  [CORE_ID_BITS:0] reg_wd;
351     always @(*) begin
352         reg_wd = {{CORE_ID_BITS}{1'b0}};
353
354         if (en)
355             // if wanting to lock the addr
356             if (lwex)
357                 // and not already locked
358                 if (!is_locked) begin
359                     reg_wd = (core_id + 1);
360                 end
361             else if (swex)
362                 if (is_locked && is_locked_self)
363                     reg_wd = {{CORE_ID_BITS}{1'b0}};
364     end
365
366     // Exclusive flag for each memory cell
367     vmicro16_bram # (
368         .MEM_WIDTH  (CORE_ID_BITS + 1),
369         .MEM_DEPTH  (MEM_DEPTH),
370         .USE_INITS  (0),
371         .NAME       ("rexram")
372     ) ram_exflags (
373         .clk        (clk),
374         .reset      (reset),
375
376         .mem_addr   (mem_addr),
377         .mem_in     (reg_wd),
378         .mem_we     (reg_we),
379         .mem_out    (ex_flags_read)
380     );
381
382     always @(*)
383         if (S_PSELx && S_PENABLE)
384             $display($time, "\t\tBRAMex[%h] READ %h\tCORE: %h",
385                 mem_addr, mem_out, S_PADDR[16 +: CORE_ID_BITS]);
386
387     always @(posedge clk)
388         if (we)
389             $display($time, "\t\tBRAMex[%h] WRITE %h\tCORE: %h",
390                 mem_addr, S_PWDATA, S_PADDR[16 +: CORE_ID_BITS]);
391
392     vmicro16_bram # (
393         .MEM_WIDTH  (MEM_WIDTH),
394         .MEM_DEPTH  (MEM_DEPTH),
395         .USE_INITS  (0),
396         .NAME       ("BRAMexinst")
397     ) bram_apb (
398         .clk        (clk),
399         .reset      (reset),
400
401         .mem_addr   (mem_addr),
402         .mem_in     (S_PWDATA),
403         .mem_we     (we && swex_success),
404         .mem_out    (mem_out)
405     );
406 endmodule
407
408 // Simple APB memory-mapped register set
409 module vmicro16_regs_apb # (
```

```
410     parameter BUS_WIDTH         = 16,
411     parameter DATA_WIDTH        = 16,
412     parameter CELL_DEPTH        = 8,
413     parameter PARAM_DEFAULTS_R0 = 0,
414     parameter PARAM_DEFAULTS_R1 = 0
415 ) (
416     input clk,
417     input reset,
418     // APB Slave to master interface
419     input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
420     input                           S_PWRITE,
421     input                           S_PSELx,
422     input                           S_PENABLE,
423     input  [DATA_WIDTH-1:0]         S_PWDATA,
424
425     output [DATA_WIDTH-1:0]         S_PRDATA,
426     output                          S_PREADY
427 );
428     wire [DATA_WIDTH-1:0] rd1;
429
430     assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1  : 16'h0000;
431     assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
432     assign reg_we   = (S_PSELx & S_PENABLE & S_PWRITE);
433
434     always @(*)
435         if (reg_we)
436             $display($time, "\t\tREGS_APB[%h] <= %h",
437                 S_PADDR, S_PWDATA);
438
439     always @(*)
440         `rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))
441
442     vmicro16_regs # (
443         .CELL_DEPTH         (CELL_DEPTH),
444         .CELL_WIDTH         (DATA_WIDTH),
445         .PARAM_DEFAULTS_R0  (PARAM_DEFAULTS_R0),
446         .PARAM_DEFAULTS_R1  (PARAM_DEFAULTS_R1)
447     ) regs_apb (
448         .clk    (clk),
449         .reset  (reset),
450         // port 1
451         .rs1    (S_PADDR),
452         .rd1    (rd1),
453         .we     (reg_we),
454         .ws1    (S_PADDR),
455         .wd     (S_PWDATA)
456         // port 2 unconnected
457         //.rs2    (),
458         //.rd2    ()
459     );
460 endmodule
461
462 // Simple GPIO write only peripheral
463 module vmicro16_gpio_apb # (
464     parameter BUS_WIDTH  = 16,
465     parameter DATA_WIDTH = 16,
466     parameter PORTS      = 8,
467     parameter NAME       = "GPIO"
468 ) (
469     input clk,
470     input reset,
471     // APB Slave to master interface
472     input  [0:0]                    S_PADDR, // not used (optimised out)
473     input                           S_PWRITE,
474     input                           S_PSELx,
475     input                           S_PENABLE,
476     input  [DATA_WIDTH-1:0]         S_PWDATA,
477
478     output [DATA_WIDTH-1:0]         S_PRDATA,
479     output                          S_PREADY,
480     output reg [PORTS-1:0]          gpio
481 );
482     assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
483     assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
484     assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);
485
486     always @(posedge clk)
487         if (reset)
488             gpio <= 0;
489         else if (ports_we) begin
490             $display($time, "\t\%s <= %h", NAME, S_PWDATA[PORTS-1:0]);
491             gpio <= S_PWDATA[PORTS-1:0];
492         end
493 endmodule
```

B.5 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
```

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

```
178
179
180 module vmicro16_core_mmu # (
181     parameter MEM_WIDTH     = 16,
182     parameter MEM_DEPTH     = 64,
183
184     parameter CORE_ID       = 3'h0,
185     parameter CORE_ID_BITS  = `clog2(`CORES)
186 ) (
187     input clk,
188     input reset,
189
190     input  req,
191     output busy,
192
193     // From core
194     input      [MEM_WIDTH-1:0]  mmu_addr,
195     input      [MEM_WIDTH-1:0]  mmu_in,
196     input                       mmu_we,
197     input                       mmu_lwex,
198     input                       mmu_swex,
199     output reg [MEM_WIDTH-1:0]  mmu_out,
200
201     // interrupts
202     output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
203     output reg [`DEF_NUM_INT-1:0]             ints_mask,
204
205     // TO APB interconnect
206     output reg [`APB_WIDTH-1:0]  M_PADDR,
207     output reg                   M_PWRITE,
208     output reg                   M_PSELx,
209     output reg                   M_PENABLE,
210     output reg [MEM_WIDTH-1:0]   M_PWDATA,
211     // from interconnect
212     input      [MEM_WIDTH-1:0]   M_PRDATA,
213     input                        M_PREADY
214 );
215     localparam MMU_STATE_T1  = 0;
216     localparam MMU_STATE_T2  = 1;
217     localparam MMU_STATE_T3  = 2;
218     reg [1:0]  mmu_state      = MMU_STATE_T1;
219
220     reg  [MEM_WIDTH-1:0] per_out = 0;
221     wire [MEM_WIDTH-1:0] tim0_out;
222
223     assign busy = req || (mmu_state == MMU_STATE_T2);
224
225     // more luts than below but easier
226     //wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
227     //            && (mmu_addr <= `DEF_MMU_TIM0_E);
228     //wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
229     //            && (mmu_addr <= `DEF_MMU_SREG_E);
230     //wire intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
231     //            && (mmu_addr <= `DEF_MMU_INTSV_E);
232     //wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
233     //            && (mmu_addr <= `DEF_MMU_INTSM_E);
234
235     wire tim0_en = ~mmu_addr[12] && ~mmu_addr[9] && ~mmu_addr[7];
236     wire sreg_en = mmu_addr[7] && ~mmu_addr[4] && ~mmu_addr[5];
237     wire intv_en = mmu_addr[8] && ~mmu_addr[3];
238     wire intm_en = mmu_addr[8] && mmu_addr[3];
239
240     wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
241     wire tim0_we   = (tim0_en && mmu_we);
242     wire intv_we   = (intv_en && mmu_we);
243     wire intm_we   = (intm_en && mmu_we);
244
245     // Special register selects
246     localparam SPECIAL_REGS = 8;
247     wire [MEM_WIDTH-1:0] sr_val;
248
249     // Interrupt vector and mask
250     initial ints_vector = 0;
251     initial ints_mask   = 0;
252     wire [2:0] intv_addr = mmu_addr[`clog2(`DEF_NUM_INT)-1:0];
253     always @(posedge clk)
254         if (intv_we)
255             ints_vector[intv_addr*`DATA_WIDTH +: `DATA_WIDTH] <= mmu_in;
256
257     always @(posedge clk)
258         if (intm_we)
259             ints_mask <= mmu_in;
260
261
262     always @(ints_vector)
263         $display($time,
264                  "\tC%d\t\tints_vector W: | %h %h %h %h | %h %h %h %h |",
265                   CORE_ID,
266                 ints_vector[0*`DATA_WIDTH +: `DATA_WIDTH],
267                 ints_vector[1*`DATA_WIDTH +: `DATA_WIDTH],
268                 ints_vector[2*`DATA_WIDTH +: `DATA_WIDTH],
```

```
269             ints_vector[3*`DATA_WIDTH +: `DATA_WIDTH],
270             ints_vector[4*`DATA_WIDTH +: `DATA_WIDTH],
271             ints_vector[5*`DATA_WIDTH +: `DATA_WIDTH],
272             ints_vector[6*`DATA_WIDTH +: `DATA_WIDTH],
273             ints_vector[7*`DATA_WIDTH +: `DATA_WIDTH]
274             );
275
276     always @(intm_we)
277         $display($time, "\tC%d\t\tintm_we W: %b", CORE_ID, ints_mask);
278
279     // Output port
280     always @(*)
281         if      (tim0_en) mmu_out = tim0_out;
282         else if (sreg_en) mmu_out = sr_val;
283         else if (intv_en) mmu_out = ints_vector[mmu_addr[2:0]*`DATA_WIDTH
284                                                 +: `DATA_WIDTH];
285         else if (intm_en) mmu_out = ints_mask;
286         else              mmu_out = per_out;
287
288     // APB master to slave interface
289     always @(posedge clk)
290         if (reset) begin
291             mmu_state <= MMU_STATE_T1;
292             M_PENABLE <= 0;
293             M_PADDR   <= 0;
294             M_PWDATA  <= 0;
295             M_PSELx   <= 0;
296             M_PWRITE  <= 0;
297         end
298         else
299             casex (mmu_state)
300                 MMU_STATE_T1: begin
301                     if (req && apb_en) begin
302                         M_PADDR   <= {mmu_lwex,
303                                       mmu_swex,
304                                       CORE_ID[CORE_ID_BITS-1:0],
305                                       mmu_addr[MEM_WIDTH-1:0]};
306
307                         M_PWDATA  <= mmu_in;
308                         M_PSELx   <= 1;
309                         M_PWRITE  <= mmu_we;
310
311                         mmu_state <= MMU_STATE_T2;
312                     end
313                 end
314
315                 `ifdef FIX_T3
316                     MMU_STATE_T2: begin
317                         M_PENABLE <= 1;
318
319                         if (M_PREADY == 1'b1) begin
320                             mmu_state <= MMU_STATE_T3;
321                         end
322                     end
323
324                     MMU_STATE_T3: begin
325                         // Slave has output a ready signal (finished)
326                         M_PENABLE <= 0;
327                         M_PADDR   <= 0;
328                         M_PWDATA  <= 0;
329                         M_PSELx   <= 0;
330                         M_PWRITE  <= 0;
331                         // Clock the peripheral output into a reg,
332                         //   to output on the next clock cycle
333                         per_out   <= M_PRDATA;
334
335                         mmu_state <= MMU_STATE_T1;
336                     end
337                 `else
338                     // No FIX_T3
339                     MMU_STATE_T2: begin
340                         if (M_PREADY == 1'b1) begin
341                             M_PENABLE <= 0;
342                             M_PADDR   <= 0;
343                             M_PWDATA  <= 0;
344                             M_PSELx   <= 0;
345                             M_PWRITE  <= 0;
346                             // Clock the peripheral output into a reg,
347                             //   to output on the next clock cycle
348                             per_out   <= M_PRDATA;
349
350                             mmu_state <= MMU_STATE_T1;
351                         end else begin
352                             M_PENABLE <= 1;
353                         end
354                     end
355                 `endif
356             endcase
357
358     (* ram_style = "block" *)
359     vmicro16_bram # (
```

```
360         .MEM_WIDTH  (MEM_WIDTH),
361         .MEM_DEPTH  (SPECIAL_REGS),
362         .USE_INITS  (0),
363         .PARAM_DEFAULTS_R0  (CORE_ID),
364         .PARAM_DEFAULTS_R1  (`CORES),
365         .PARAM_DEFAULTS_R2  (`APB_BRAM0_CELLS),
366         .PARAM_DEFAULTS_R3  (`SLAVES),
367         .NAME       ("ram_sr")
368     ) ram_sr (
369         .clk        (clk),
370         .reset      (reset),
371         .mem_addr   (mmu_addr[`clog2(SPECIAL_REGS)-1:0]),
372         .mem_in     (),
373         .mem_we     (),
374         .mem_out    (sr_val)
375     );
376
377     // Each M core has a TIM0 scratch memory
378     (* ram_style = "block" *)
379     vmicro16_bram # (
380         .MEM_WIDTH  (MEM_WIDTH),
381         .MEM_DEPTH  (MEM_DEPTH),
382         .USE_INITS  (0),
383         .NAME       ("TIM0")
384     ) TIM0 (
385         .clk        (clk),
386         .reset      (reset),
387         .mem_addr   (mmu_addr[7:0]),
388         .mem_in     (mmu_in),
389         .mem_we     (tim0_we),
390         .mem_out    (tim0_out)
391     );
392 endmodule
393
394
395
396 module vmicro16_regs # (
397     parameter CELL_WIDTH        = 16,
398     parameter CELL_DEPTH        = 8,
399     parameter CELL_SEL_BITS     = `clog2(CELL_DEPTH),
400     parameter CELL_DEFAULTS     = 0,
401     parameter DEBUG_NAME        = "",
402     parameter CORE_ID           = 0,
403     parameter PARAM_DEFAULTS_R0 = 16'h0000,
404     parameter PARAM_DEFAULTS_R1 = 16'h0000
405 ) (
406     input clk,
407     input reset,
408     // Dual port register reads
409     input       [CELL_SEL_BITS-1:0]  rs1, // port 1
410     output      [CELL_WIDTH-1   :0]  rd1,
411     //input       [CELL_SEL_BITS-1:0]  rs2, // port 2
412     //output      [CELL_WIDTH-1   :0]  rd2,
413     // EX/WB final stage write back
414     input                            we,
415     input [CELL_SEL_BITS-1:0]        ws1,
416     input [CELL_WIDTH-1:0]           wd
417 );
418     (* ram_style = "distributed" *)
419     reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;
420
421     // Initialise registers with default values
422     //   Really only used for special registers used by the soc
423     // TODO: How to do this on reset?
424     integer i;
425     initial
426         if (CELL_DEFAULTS)
427             $readmemh(CELL_DEFAULTS, regs);
428         else begin
429             for(i = 0; i < CELL_DEPTH; i = i + 1)
430                 regs[i] = 0;
431             regs[0] = PARAM_DEFAULTS_R0;
432             regs[1] = PARAM_DEFAULTS_R1;
433             end
434
435     `ifdef ICARUS
436         always @(regs)
437             $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
438                 CORE_ID,
439                 regs[0], regs[1], regs[2], regs[3],
440                 regs[4], regs[5], regs[6], regs[7]);
441     `endif
442
443     always @(posedge clk)
444         if (reset) begin
445             for(i = 0; i < CELL_DEPTH; i = i + 1)
446                 regs[i] <= 0;
447             regs[0] <= PARAM_DEFAULTS_R0;
448             regs[1] <= PARAM_DEFAULTS_R1;
449         end
450         else if (we) begin
```

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

```
542
543         `VMICRO16_OP_ARITH_U:     casez (simm5)
544             `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
545             `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
546             `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
547             default:                  alu_op = `VMICRO16_ALU_BAD; endcase
548
549         `VMICRO16_OP_ARITH_S:     casez (simm5)
550             `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
551             `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
552             `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
553             default:                  alu_op = `VMICRO16_ALU_BAD; endcase
554
555         default: begin
556                                       alu_op = `VMICRO16_ALU_NOP;
557             $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
558         end
559     endcase
560
561     // Special opcodes
562     //assign nop  == ((opcode == `VMICRO16_OP_SPCL) & (~instr[0]));
563     assign halt = ((opcode == `VMICRO16_OP_SPCL) &   instr[0]);
564     assign intr = ((opcode == `VMICRO16_OP_SPCL) &   instr[1]);
565
566     // Register writes
567     always @(*) case (opcode)
568         `VMICRO16_OP_LWEX,
569         `VMICRO16_OP_SWEX,
570         `VMICRO16_OP_LW,
571         `VMICRO16_OP_MOV,
572         `VMICRO16_OP_MOVI,
573         //`VMICRO16_OP_MOVI_L,
574         `VMICRO16_OP_ARITH_U,
575         `VMICRO16_OP_ARITH_S,
576         `VMICRO16_OP_SETC,
577         `VMICRO16_OP_BIT,
578         `VMICRO16_OP_MULT:      has_we = 1'b1;
579         default:                has_we = 1'b0;
580     endcase
581
582     // Contains 4-bit immediate
583     always @(*)
584         if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
585             ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
586             has_imm4 = 1'b1;
587         else
588             has_imm4 = 1'b0;
589
590     // Contains 8-bit immediate
591     always @(*) case (opcode)
592         `VMICRO16_OP_MOVI,
593         `VMICRO16_OP_BR:        has_imm8 = 1'b1;
594         default:                has_imm8 = 1'b0;
595     endcase
596
597     //// Contains 12-bit immediate
598     //always @(*) case (opcode)
599     //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
600     //    default:                has_imm12 = 1'b0;
601     //endcase
602
603     // Will branch the pc
604     always @(*) case (opcode)
605         `VMICRO16_OP_BR:    has_br = 1'b1;
606         default:            has_br = 1'b0;
607     endcase
608
609     // Requires external memory
610     always @(*) case (opcode)
611         `VMICRO16_OP_LW,
612         `VMICRO16_OP_SW,
613         `VMICRO16_OP_LWEX,
614         `VMICRO16_OP_SWEX:  has_mem = 1'b1;
615         default:            has_mem = 1'b0;
616     endcase
617
618     // Requires external memory write
619     always @(*) case (opcode)
620         `VMICRO16_OP_SW,
621         `VMICRO16_OP_SWEX:  has_mem_we = 1'b1;
622         default:            has_mem_we = 1'b0;
623     endcase
624
625     // Affects status registers (cmp instructions)
626     always @(*) case (opcode)
627         `VMICRO16_OP_CMP:   has_cmp = 1'b1;
628         default:            has_cmp = 1'b0;
629     endcase
630
631     // Performs exclusive checks
632     always @(*) case (opcode)
```

```
633         `VMICRO16_OP_LWEX:   has_lwex = 1'b1;
634         default:             has_lwex = 1'b0;
635     endcase
636
637     always @(*) case (opcode)
638         `VMICRO16_OP_SWEX:   has_swex = 1'b1;
639         default:             has_swex = 1'b0;
640     endcase
641 endmodule
642
643
644 module vmicro16_alu # (
645     parameter OP_WIDTH   = 5,
646     parameter DATA_WIDTH = 16,
647     parameter CORE_ID    = 0
648 ) (
649     // input clk, // TODO: make clocked
650
651     input      [OP_WIDTH-1:0]   op,
652     input      [DATA_WIDTH-1:0] a, // rs1/dst
653     input      [DATA_WIDTH-1:0] b, // rs2
654     input      [3:0]            flags,
655     output reg [DATA_WIDTH-1:0] c
656 );
657     localparam TOP_BIT = (DATA_WIDTH-1);
658     // 17-bit register
659     reg [DATA_WIDTH:0] cmp_tmp = 0; // = {carry, [15:0]}
660     wire r_setc;
661
662     always @(*) begin
663                         cmp_tmp = 0;
664                         case (op)
665         // branch/nop, output nothing
666         `VMICRO16_ALU_BR,
667         `VMICRO16_ALU_NOP:          c = {DATA_WIDTH{1'b0}};
668         // load/store addresses (use value in rd2)
669         `VMICRO16_ALU_LW,
670         `VMICRO16_ALU_SW:           c = b;
671         // bitwise operations
672         `VMICRO16_ALU_BIT_OR:       c = a | b;
673         `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
674         `VMICRO16_ALU_BIT_AND:      c = a & b;
675         `VMICRO16_ALU_BIT_NOT:      c = ~(b);
676         `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
677         `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;
678
679         `VMICRO16_ALU_MOV:          c = b;
680         `VMICRO16_ALU_MOVI:         c = b;
681         `VMICRO16_ALU_MOVI_L:       c = b;
682
683         `VMICRO16_ALU_ARITH_UADD:   c = a + b;
684         `VMICRO16_ALU_ARITH_USUB:   c = a - b;
685         // TODO: ALU should have simm5 as input
686         `VMICRO16_ALU_ARITH_UADDI:  c = a + b;
687
688         `ifdef DEF_ALU_HW_MULT
689             `VMICRO16_ALU_MULT:  c = a * b;
690         `endif
691
692         `VMICRO16_ALU_ARITH_SADD:   c = $signed(a) + $signed(b);
693         `VMICRO16_ALU_ARITH_SSUB:   c = $signed(a) - $signed(b);
694         // TODO: ALU should have simm5 as input
695         `VMICRO16_ALU_ARITH_SSUBI:  c = $signed(a) - $signed(b);
696
697         `VMICRO16_ALU_CMP: begin
698             // TODO: Do a-b in 17-bit register
699             //       Set zero, overflow, carry, signed bits in result
700             cmp_tmp = a - b;
701             c = 0;
702
703             // N   Negative condition code flag
704             // Z   Zero condition code flag
705             // C   Carry condition code flag
706             // V   Overflow condition code flag
707             c[`VMICRO16_SFLAG_N] = cmp_tmp[TOP_BIT];
708             c[`VMICRO16_SFLAG_Z] = (cmp_tmp == 0);
709             c[`VMICRO16_SFLAG_C] = 0; //cmp_tmp[TOP_BIT+1]; // not used
710
711             // Overflow flag
712             // https://stackoverflow.com/questions/30957188/
713             // https://github.com/bendl/prco304/blob/master/prco_core/rtl/prco_alu.v#L50
714             case(cmp_tmp[TOP_BIT+1:TOP_BIT])
715                 2'b01:   c[`VMICRO16_SFLAG_V] = 1;
716                 2'b10:   c[`VMICRO16_SFLAG_V] = 1;
717                 default: c[`VMICRO16_SFLAG_V] = 0;
718             endcase
719
720             $display($time, "\tC%02h: ALU CMP: %h %h = %h = %b", CORE_ID, a, b, cmp_tmp, c[3:0]);
721         end
722
723         `VMICRO16_ALU_SETC: c = { {15{1'b0}}, r_setc };
```

```
724
725         // TODO: Parameterise
726         default: begin
727             $display($time, "\tALU: unknown op: %h", op);
728             c       = 0;
729             cmp_tmp = 0;
730         end
731                 endcase
732                 end
733
734     branch setc_check (
735         .flags      (flags),
736         .cond       (b[7:0]),
737         .en         (r_setc)
738     );
739 endmodule
740
741 // flags = 4 bit r_cmp_flags register
742 // cond  = 8 bit VMICRO16_OP_BR_? value. See vmicro16_isa.v
743 module branch (
744     input [3:0] flags,
745     input [7:0] cond,
746     output reg  en
747 );
748     always @(*)
749         case (cond)
750             `VMICRO16_OP_BR_U:  en = 1; `VMICRO16_OP_BR_U:  en = 1;
751             `VMICRO16_OP_BR_E:  en = (flags[`VMICRO16_SFLAG_Z] == 1);
752             `VMICRO16_OP_BR_NE: en = (flags[`VMICRO16_SFLAG_Z] == 0);
753             `VMICRO16_OP_BR_G:  en = (flags[`VMICRO16_SFLAG_Z] == 0) &&
754                                      (flags[`VMICRO16_SFLAG_N] == flags[`VMICRO16_SFLAG_V]);
755             `VMICRO16_OP_BR_L:  en = (flags[`VMICRO16_SFLAG_Z] != flags[`VMICRO16_SFLAG_N]);
756             `VMICRO16_OP_BR_GE: en = (flags[`VMICRO16_SFLAG_Z] == flags[`VMICRO16_SFLAG_N]);
757             `VMICRO16_OP_BR_LE: en = (flags[`VMICRO16_SFLAG_Z] == 1) ||
758                                      (flags[`VMICRO16_SFLAG_N] != flags[`VMICRO16_SFLAG_V]);
759             default:            en = 0;
760         endcase
761 endmodule
762
763
764
765 module vmicro16_core # (
766     parameter DATA_WIDTH        = 16,
767     parameter MEM_INSTR_DEPTH   = 64,
768     parameter MEM_SCRATCH_DEPTH = 64,
769     parameter MEM_WIDTH         = 16,
770
771     parameter CORE_ID           = 3'h0
772 ) (
773     input        clk,
774     input        reset,
775
776     output [7:0] dbug,
777
778     output       halt,
779
780     // interrupt sources
781     input  [`DEF_NUM_INT-1:0]             ints,
782     input  [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_data,
783     output [`DEF_NUM_INT-1:0]             ints_ack,
784
785     // APB master to slave interface (apb_intercon)
786     output  [`APB_WIDTH-1:0]    w_PADDR,
787     output                      w_PWRITE,
788     output                      w_PSELx,
789     output                      w_PENABLE,
790     output  [DATA_WIDTH-1:0]    w_PWDATA,
791     input   [DATA_WIDTH-1:0]    w_PRDATA,
792     input                       w_PREADY
793
794 `ifndef DEF_CORE_HAS_INSTR_MEM
795     , // APB master interface to slave instruction memory
796     output reg [`APB_WIDTH-1:0]    w2_PADDR,
797     output reg                     w2_PWRITE,
798     output reg                     w2_PSELx,
799     output reg                     w2_PENABLE,
800     output reg [DATA_WIDTH-1:0]    w2_PWDATA,
801     input      [DATA_WIDTH-1:0]    w2_PRDATA,
802     input                          w2_PREADY
803 `endif
804 );
805     localparam STATE_IF = 0;
806     localparam STATE_R1 = 1;
807     localparam STATE_R2 = 2;
808     localparam STATE_ME = 3;
809     localparam STATE_WB = 4;
810     localparam STATE_FE = 5;
811     localparam STATE_IDLE = 6;
812     localparam STATE_HALT = 7;
813     reg  [2:0] r_state = STATE_IF;
814
```

```
815     reg  [DATA_WIDTH-1:0] r_pc          = 16'h0000;
816     reg  [DATA_WIDTH-1:0] r_pc_saved    = 16'h0000;
817     reg  [DATA_WIDTH-1:0] r_instr       = 16'h0000;
818     wire [DATA_WIDTH-1:0] w_mem_instr_out;
819     wire                  w_halt;
820
821     assign dbug = {7'h00, w_halt};
822     assign halt = w_halt;
823
824     wire [4:0]            r_instr_opcode;
825     wire [4:0]            r_instr_alu_op;
826     wire [2:0]            r_instr_rsd;
827     wire [2:0]            r_instr_rsa;
828     reg  [DATA_WIDTH-1:0] r_instr_rdd = 0;
829     reg  [DATA_WIDTH-1:0] r_instr_rda = 0;
830     wire [3:0]            r_instr_imm4;
831     wire [7:0]            r_instr_imm8;
832     wire [4:0]            r_instr_simm5;
833     wire                  r_instr_has_imm4;
834     wire                  r_instr_has_imm8;
835     wire                  r_instr_has_we;
836     wire                  r_instr_has_br;
837     wire                  r_instr_has_cmp;
838     wire                  r_instr_has_mem;
839     wire                  r_instr_has_mem_we;
840     wire                  r_instr_halt;
841     wire                  r_instr_has_lwex;
842     wire                  r_instr_has_swex;
843
844     wire [DATA_WIDTH-1:0] r_alu_out;
845
846     wire [DATA_WIDTH-1:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
847     wire [DATA_WIDTH-1:0] r_mem_scratch_in   = r_instr_rdd;
848     wire [DATA_WIDTH-1:0] r_mem_scratch_out;
849     wire                  r_mem_scratch_we   = r_instr_has_mem_we && (r_state == STATE_ME);
850     reg                   r_mem_scratch_req  = 0;
851     wire                  r_mem_scratch_busy;
852
853     reg  [2:0]            r_reg_rs1 = 0;
854     wire [DATA_WIDTH-1:0] r_reg_rd1_s;
855     wire [DATA_WIDTH-1:0] r_reg_rd1_i;
856     wire [DATA_WIDTH-1:0] r_reg_rd1 = regs_use_int ? r_reg_rd1_i : r_reg_rd1_s;
857     //wire [15:0] r_reg_rd2;
858     wire [DATA_WIDTH-1:0] r_reg_wd = (r_instr_has_mem) ? r_mem_scratch_out : r_alu_out;
859     wire                  r_reg_we = r_instr_has_we && (r_state == STATE_WB);
860
861     // branching
862     wire        w_intr;
863     wire        w_branch_en;
864     wire        w_branching   = r_instr_has_br && w_branch_en;
865     reg  [3:0]  r_cmp_flags   = 4'h00; // N, Z, C, V
866
867     always @(r_cmp_flags)
868         $display($time, "\tC%02h:\tALU CMP: %b", CORE_ID, r_cmp_flags);
869
870     // 2 cycle register fetch
871     always @(*) begin
872         r_reg_rs1 = 0;
873         if (r_state == STATE_R1)
874             r_reg_rs1 = r_instr_rsd;
875         else if (r_state == STATE_R2)
876             r_reg_rs1 = r_instr_rsa;
877         else
878             r_reg_rs1 = 3'h0;
879     end
880
881     reg regs_use_int = 0;
882     `ifdef DEF_ENABLE_INT
883     wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_vector;
884     wire [`DEF_NUM_INT-1:0]             ints_mask;
885     wire                                has_int = ints & ints_mask;
886     reg int_pending = 0;
887     reg int_pending_ack = 0;
888     always @(posedge clk)
889         if (int_pending_ack)
890             // We've now branched to the isr
891             int_pending <= 0;
892         else if (has_int)
893             // Notify fsm to switch to the ints_vector at the last stage
894             int_pending <= 1;
895         else if (w_intr)
896             // Return to Interrupt instruction called,
897             //   so we've finished with the interrupt
898             int_pending <= 0;
899     `endif
900
901     // Next program counter logic
902     reg [`DATA_WIDTH-1:0] next_pc = 0;
903     always @(posedge clk)
904         if (reset)
905             r_pc <= 0;
```

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

```
997             if (r_state == STATE_IF) begin
998                 // wait for ready signal
999                 if (!w2_PREADY) begin
1000                    w2_PSELx   <= 1;
1001                    w2_PWRITE  <= 0;
1002                    w2_PENABLE <= 1;
1003                    w2_PWDATA  <= 0;
1004                    w2_PADDR   <= r_pc;
1005                end else begin
1006                    w2_PSELx   <= 0;
1007                    w2_PWRITE  <= 0;
1008                    w2_PENABLE <= 0;
1009                    w2_PWDATA  <= 0;
1010
1011                    r_instr <= w2_PRDATA;
1012
1013                    $display("");
1014                    $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
1015                    $display($time, "\tC%02h: INSTR: %h", CORE_ID, w2_PRDATA);
1016
1017                    r_state <= STATE_R1;
1018                end
1019            end
1020    `endif
1021
1022            else if (r_state == STATE_R1) begin
1023                if (w_halt) begin
1024                    $display("");
1025                    $display("");
1026                    $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
1027                    r_state <= STATE_HALT;
1028                end else begin
1029                    // primary operand
1030                    r_instr_rdd <= r_reg_rd1;
1031                    r_state     <= STATE_R2;
1032                end
1033            end
1034            else if (r_state == STATE_R2) begin
1035                // Choose secondary operand (register or immediate)
1036                if      (r_instr_has_imm8)  r_instr_rda <= r_instr_imm8;
1037                else if (r_instr_has_imm4)  r_instr_rda <= r_reg_rd1 + r_instr_imm4;
1038                else                        r_instr_rda <= r_reg_rd1;
1039
1040                if (r_instr_has_mem) begin
1041                    r_state           <= STATE_ME;
1042                    // Pulse req
1043                    r_mem_scratch_req <= 1;
1044                end else
1045                    r_state <= STATE_WB;
1046            end
1047            else if (r_state == STATE_ME) begin
1048                // Pulse req
1049                r_mem_scratch_req <= 0;
1050                // Wait for MMU to finish
1051                if (!r_mem_scratch_busy)
1052                    r_state <= STATE_WB;
1053            end
1054            else if (r_state == STATE_WB) begin
1055                if (r_instr_has_cmp) begin
1056                    $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
1057                    r_cmp_flags <= r_alu_out[3:0];
1058                end
1059
1060                r_state <= STATE_FE;
1061            end
1062            else if (r_state == STATE_FE)
1063                r_state <= STATE_IF;
1064            else if (r_state == STATE_HALT) begin
1065                `ifdef DEF_ENABLE_INT
1066                    if (int_pending) begin
1067                        r_state <= STATE_FE;
1068                    end
1069                `endif
1070            end
1071        end
1072
1073    `ifdef DEF_CORE_HAS_INSTR_MEM
1074        // Instruction ROM
1075        (* rom_style = "distributed" *)
1076        vmicro16_bram # (
1077            .MEM_WIDTH      (DATA_WIDTH),
1078            .MEM_DEPTH      (MEM_INSTR_DEPTH),
1079            .CORE_ID        (CORE_ID),
1080            .USE_INITS      (1),
1081            .NAME           ("INSTR_MEM")
1082        ) mem_instr (
1083            .clk            (clk),
1084            .reset          (reset),
1085            // port 1
1086            .mem_addr       (r_pc),
1087            .mem_in         (0),
```

```
1088            .mem_we         (1'b0),  // ROM
1089            .mem_out        (w_mem_instr_out)
1090        );
1091    `endif
1092
1093        // MMU
1094        vmicro16_core_mmu # (
1095            .MEM_WIDTH      (DATA_WIDTH),
1096            .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
1097            .CORE_ID        (CORE_ID)
1098        ) mmu (
1099            .clk            (clk),
1100            .reset          (reset),
1101            .req            (r_mem_scratch_req),
1102            .busy           (r_mem_scratch_busy),
1103            // interrupts
1104            .ints_vector    (ints_vector),
1105            .ints_mask      (ints_mask),
1106            // port 1
1107            .mmu_addr       (r_mem_scratch_addr),
1108            .mmu_in         (r_mem_scratch_in),
1109            .mmu_we         (r_mem_scratch_we),
1110            .mmu_lwex       (r_instr_has_lwex),
1111            .mmu_swex       (r_instr_has_swex),
1112            .mmu_out        (r_mem_scratch_out),
1113            // APB maste    r to slave
1114            .M_PADDR        (w_PADDR),
1115            .M_PWRITE       (w_PWRITE),
1116            .M_PSELx        (w_PSELx),
1117            .M_PENABLE      (w_PENABLE),
1118            .M_PWDATA       (w_PWDATA),
1119            .M_PRDATA       (w_PRDATA),
1120            .M_PREADY       (w_PREADY)
1121        );
1122
1123        // Instruction decoder
1124        vmicro16_dec dec (
1125            // input
1126            .instr          (r_instr),
1127            // output async
1128            .opcode         (),
1129            .rd             (r_instr_rsd),
1130            .ra             (r_instr_rsa),
1131            .imm4           (r_instr_imm4),
1132            .imm8           (r_instr_imm8),
1133            .imm12          (),
1134            .simm5          (r_instr_simm5),
1135            .alu_op         (r_instr_alu_op),
1136            .has_imm4       (r_instr_has_imm4),
1137            .has_imm8       (r_instr_has_imm8),
1138            .has_we         (r_instr_has_we),
1139            .has_br         (r_instr_has_br),
1140            .has_cmp        (r_instr_has_cmp),
1141            .has_mem        (r_instr_has_mem),
1142            .has_mem_we     (r_instr_has_mem_we),
1143            .halt           (w_halt),
1144            .intr           (w_intr),
1145            .has_lwex       (r_instr_has_lwex),
1146            .has_swex       (r_instr_has_swex)
1147        );
1148
1149        // Software registers
1150        vmicro16_regs # (
1151            .CORE_ID    (CORE_ID),
1152            .CELL_WIDTH (`DATA_WIDTH)
1153        ) regs (
1154            .clk        (clk),
1155            .reset      (reset),
1156            // async port 0
1157            .rs1        (r_reg_rs1),
1158            .rd1        (r_reg_rd1_s),
1159            // async port 1
1160            //.rs2        (),
1161            //.rd2        (),
1162            // write port
1163            .we         (r_reg_we && ~regs_use_int),
1164            .ws1        (r_instr_rsd),
1165            .wd         (r_reg_wd)
1166        );
1167
1168        // Interrupt replacement registers
1169        `ifdef DEF_ENABLE_INT
1170        vmicro16_regs # (
1171            .CORE_ID    (CORE_ID),
1172            .CELL_WIDTH (`DATA_WIDTH),
1173            .DEBUG_NAME ("REGSINT")
1174        ) regs_intr (
1175            .clk        (clk),
1176            .reset      (reset),
1177            // async port 0
1178            .rs1        (r_reg_rs1),
```

```
1179            .rd1        (r_reg_rd1_i),
1180            // async port 1
1181            //.rs2        (),
1182            //.rd2        (),
1183            // write port
1184            .we         (r_reg_we && regs_use_int),
1185            .ws1        (r_instr_rsd),
1186            .wd         (r_reg_wd)
1187        );
1188        `endif
1189
1190        // ALU
1191        vmicro16_alu # (
1192            .CORE_ID(CORE_ID)
1193        ) alu (
1194            .op         (r_instr_alu_op),
1195            .a          (r_instr_rdd),
1196            .b          (r_instr_rda),
1197            .flags      (r_cmp_flags),
1198            // async output
1199            .c          (r_alu_out)
1200        );
1201
1202        branch branch_check (
1203            .flags      (r_cmp_flags),
1204            .cond       (r_instr_imm8),
1205            .en         (w_branch_en)
1206        );
1207
1208    endmodule
```

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.