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 18, 2019

Word count: 4689

Abstract

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

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

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

Revision History

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

Document revisions.

Declaration of Academic Integrity

Name: Ben David Lancaster
Date: August 18, 2019

Table of Contents

Chapter 1

Introduction

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

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

1.1 Why Multi-core?

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

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

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

1.2 Why RISC?

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

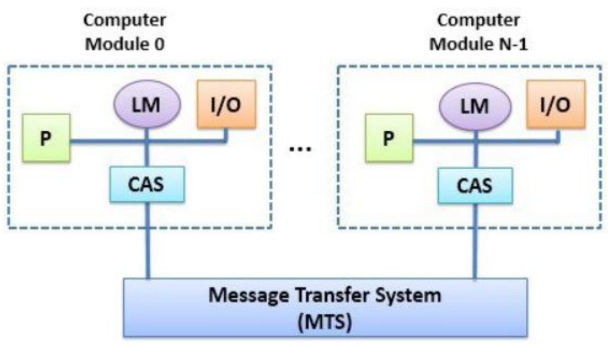


Figure 2.1: A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [3].

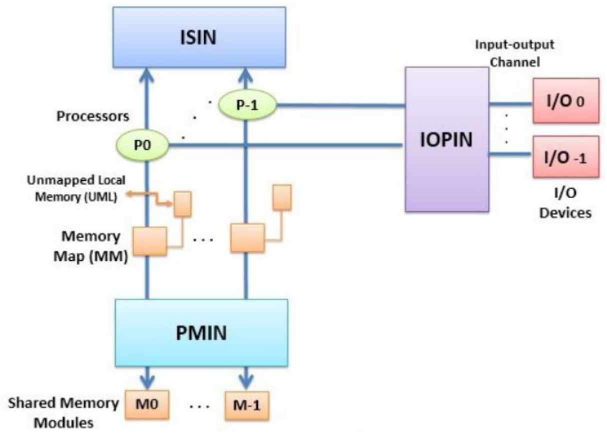


Figure 2.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [3].

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].

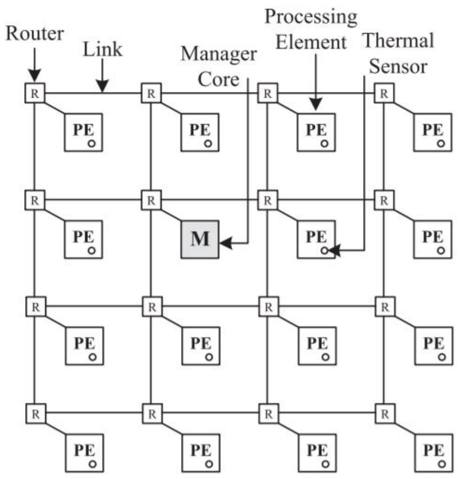


Figure 2.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [6].

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.

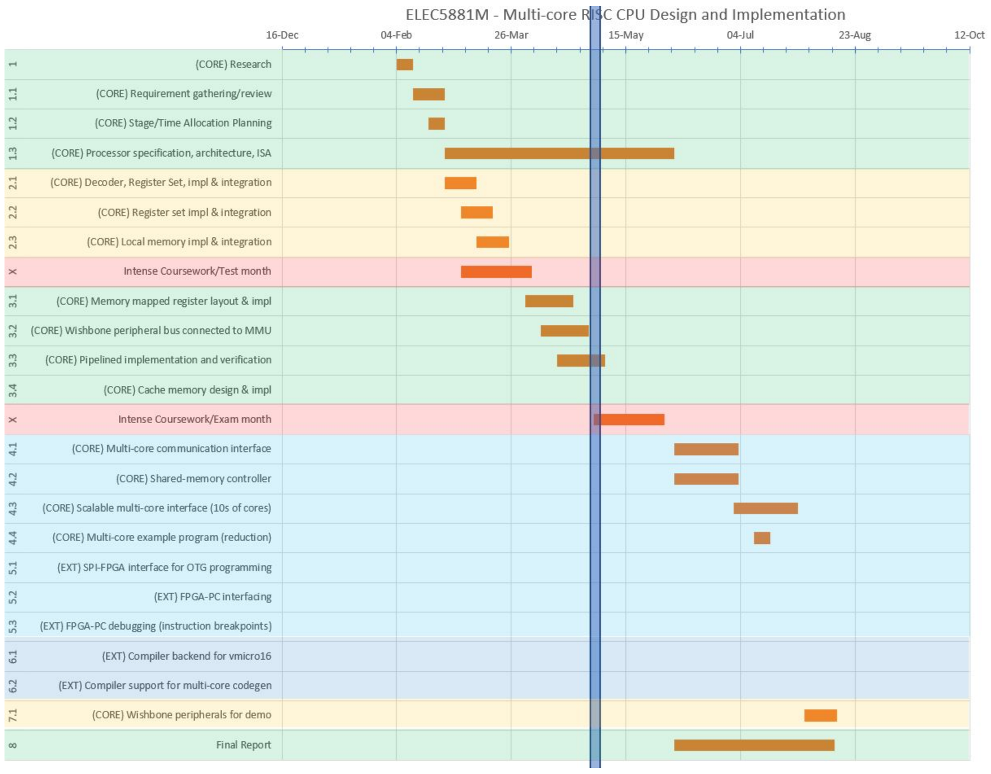


Figure 3.1: Project stages in a Gantt chart.

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.

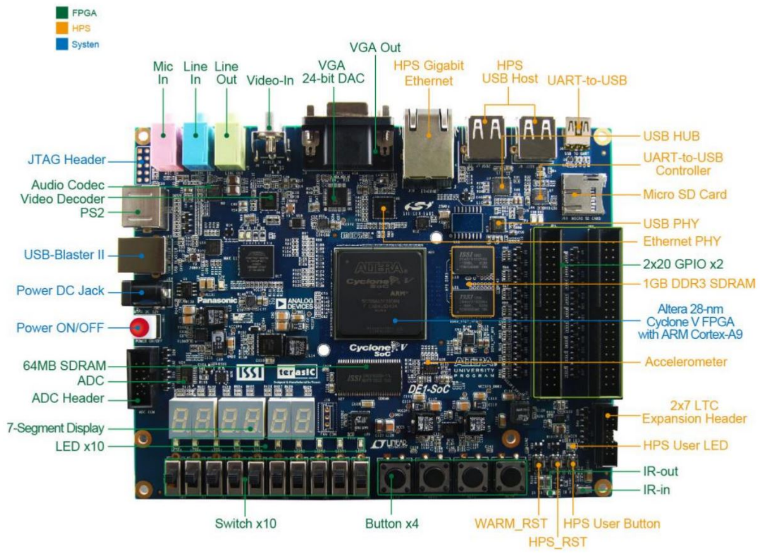


Figure 3.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [10].

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.

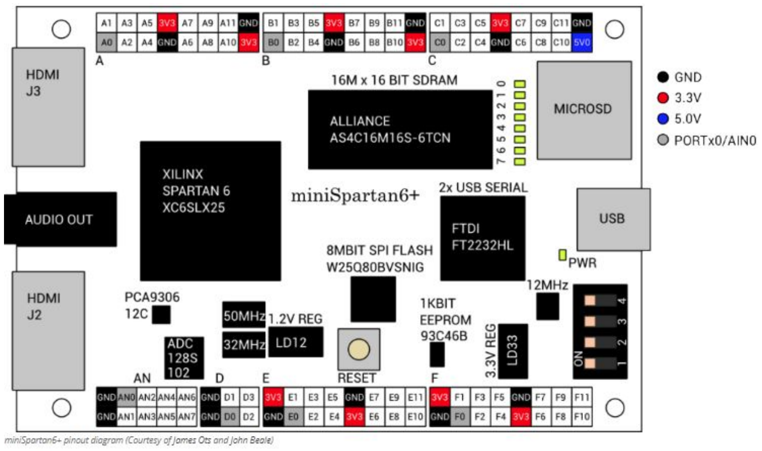


Figure 3.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [11].

3.3.2 Software Resources

Intel Quartus

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

Xilinx ISE Webpack

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

Verilator

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

3.4 Legal and Ethical Considerations

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

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

The processor does not store/track/predict software instructions. The processor uses pipelining techniques to improve performance which results in future instructions entering the pipeline even if the software's logical sequence does not include these instructions. This could result in security vulnerabilities similar to Intel's Spectre vulnerability [12].

Chapter 4

Single-core Design

While the majority of this report will focus on the multi-processing functionality of this project, it is important understand the design decisions of the single core that will be instantiated many times.

4.1 Introduction

4.1.1 Design and Implementation

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

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

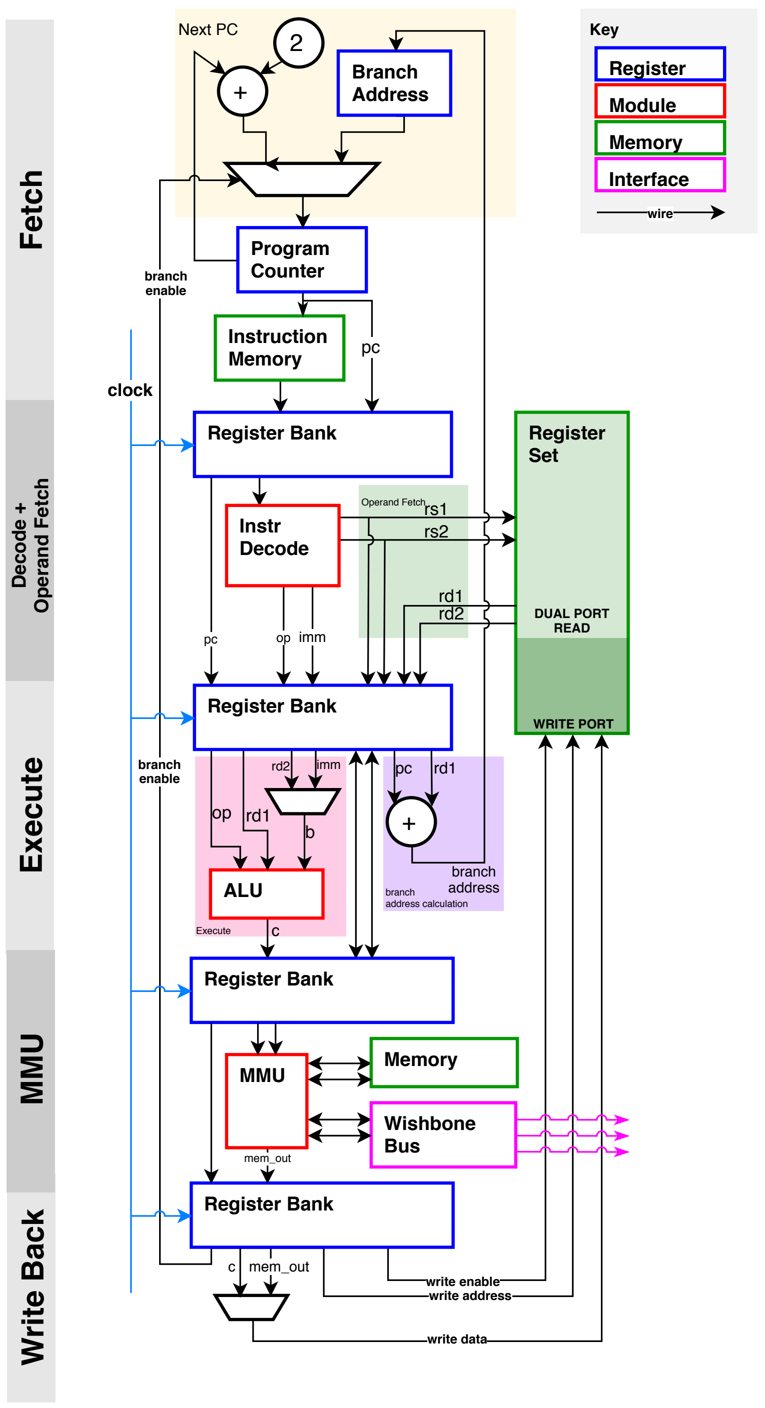


Figure 4.1: Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

Instruction and Data Memory

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

Register File

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

Pipelining

Chapter 5

Interconnect

5.1 Introduction

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

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

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

5.1.1 Comparison of On-chip Buses

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

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

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

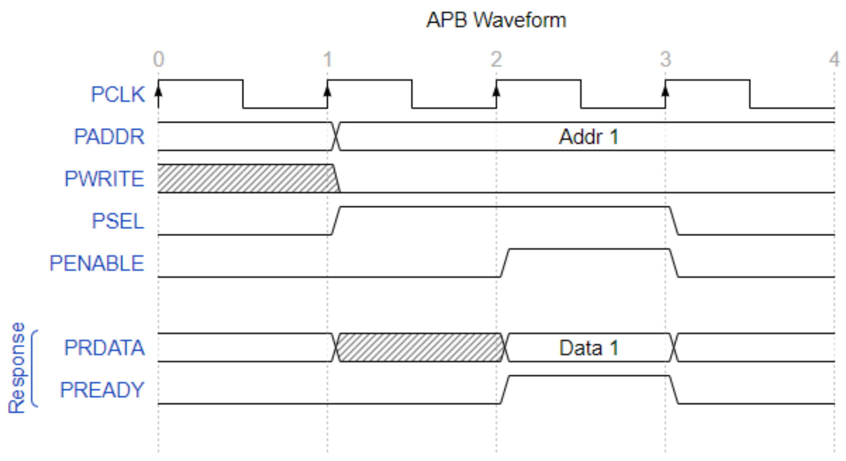


Figure 5.1: Waveform showing an APB read transaction.

5.2 Overview

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

A block diagram of this project is shown in Figure 5.2

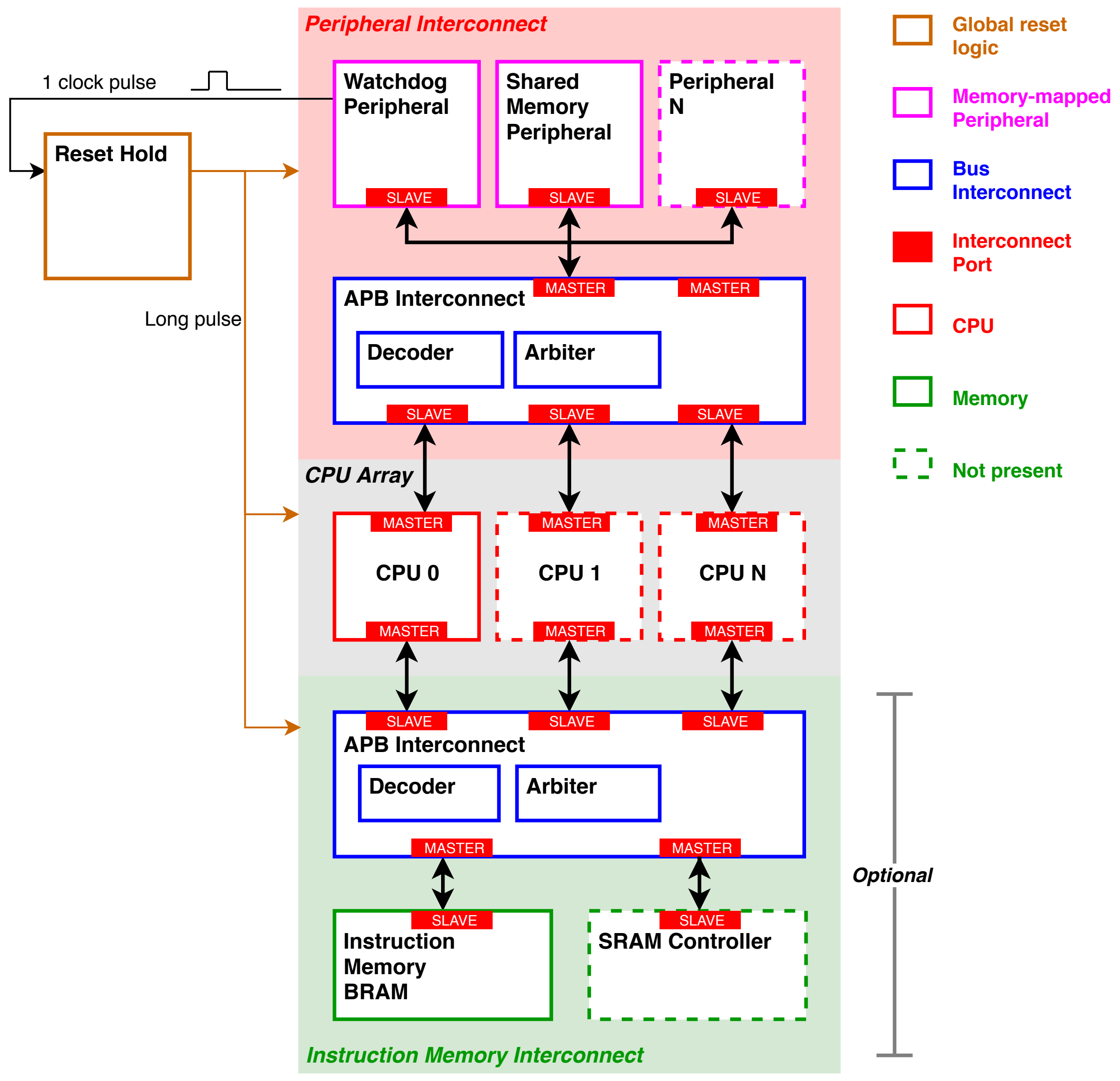


Figure 5.2: Block diagram of the Vmicro16 system-on-chip.

5.2.1 Design Considerations

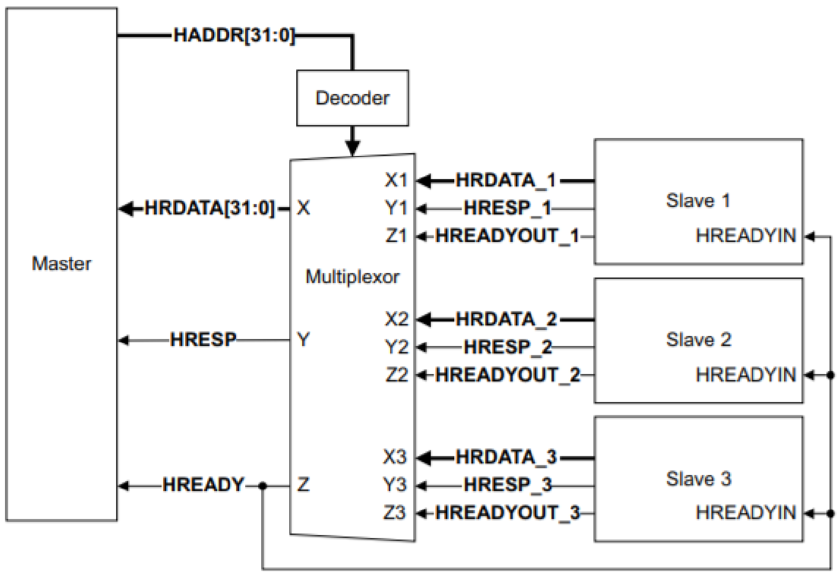
There are several design issues to consider for this project. These are listed below:

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

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

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

5.3 Interfaces



5.3.1 Master to Slave Interface

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

5.3.2 Multi-master Support

Design Goals

DG1. Foo
Bing

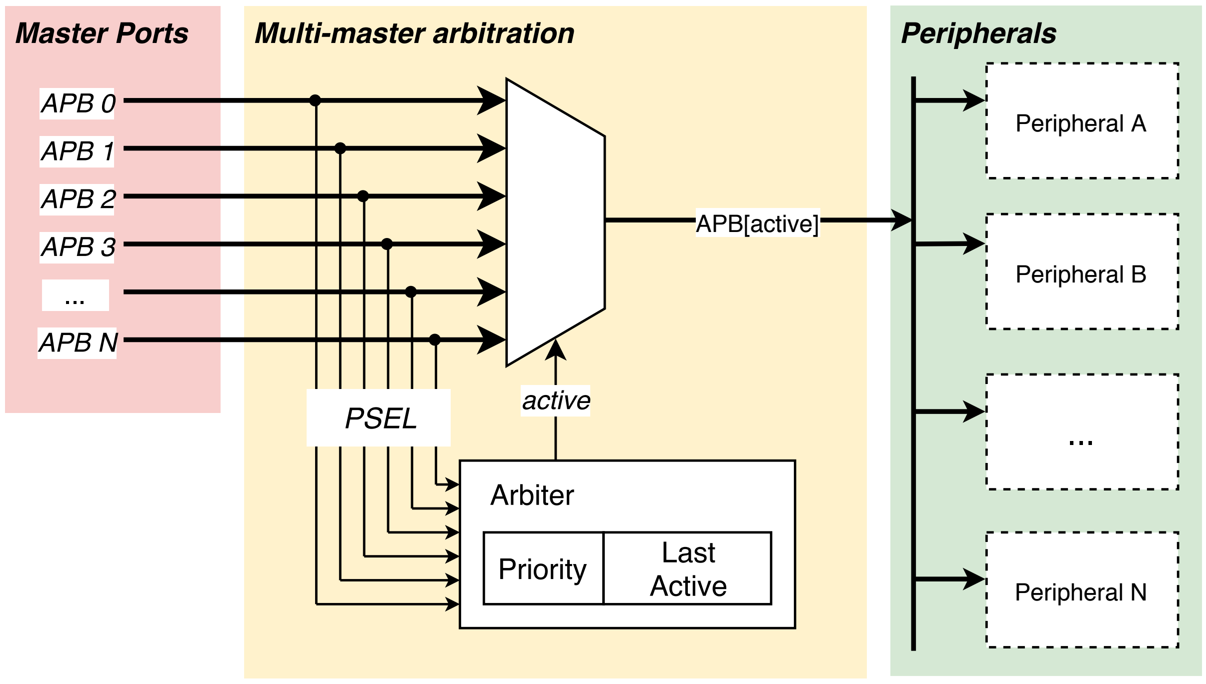


Figure 5.3: Foo

```
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 5.4: Variable size inputs and outputs to the interconnect.

83	62	41	20	0
Core N-1	$\cdots$	Core 1	Core 0	

5.4 Further Work

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

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

- Supports multiple master interfaces.

Chapter 6

Memory Mapping

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

6.1 Introduction

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

6.2 Address Decoding

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

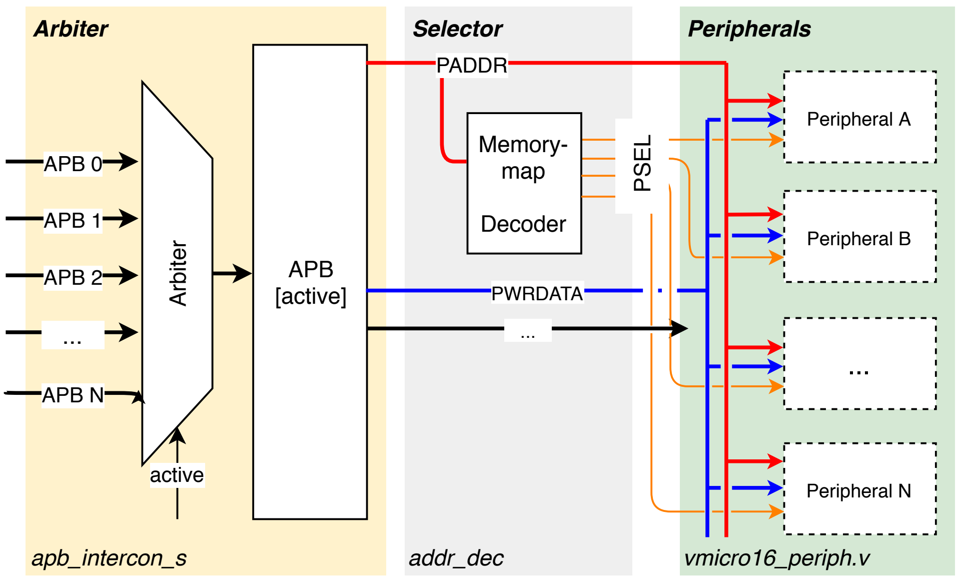


Figure 6.1: Schematic showing the address decoder (addr_dec) accepting the active PADDR signal and outputting PSEL chip enable signals to each peripheral.

6.2.1 Decoder Optimisations

Performing a 16-bit equality comparison of the `PADDR` signal against each peripheral memory address consumes a significant amount of logic. Depending on the synthesis tools and FPGA features, a 16-bit comparator might require a fixed 16-bit value input to compare against (where the 0s are inverted) and a wide-AND to reduce and compare [13, 14]. An example 4-bit comparator is shown below in Figure 6.2.

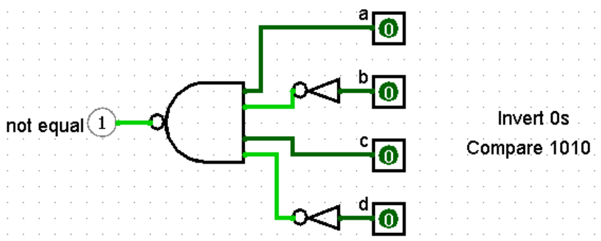


Figure 6.2: Example 4-bit binary comparator which compares the bits (a, b, c, d) to the constant value 1010. The 0s of the constant are inverted and then all are passed to a wide-AND.

As we are targeting FPGAs, which use LUTs to implement combinatorial logic, we can conveniently utilise Verilog's == operator on fairly large operands without worrying about consuming too many resources. The targeted FPGA devices in this project, the Cyclone V and Spartan 6, feature 6-input LUTs which allow 64 different configurations [15, 16]. Knowing this, we can design the address decoder to utilise the FPGA's LUTs more effectively and reduce it's footprint significantly.

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

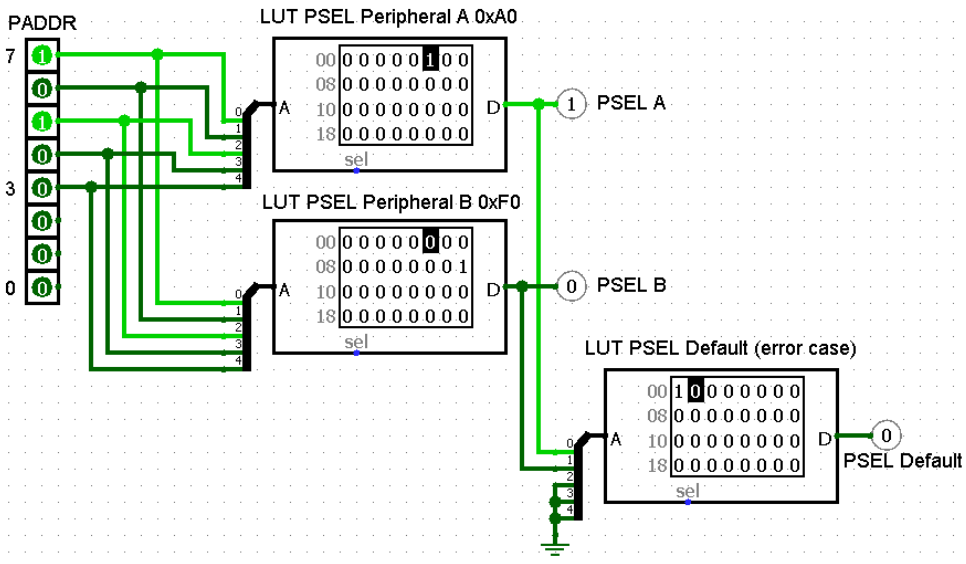

PADDR
LUT PSEL Peripheral A 0xA0
PSEL A
LUT PSEL Peripheral B 0xF0
PSEL B
LUT PSEL Default (error case)
PSEL Default
sel

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

The address decoding methods discussed above are examples of *full-address* decoding, where each bit (whether required or not) is compared. It is possible to further reduce the required logic by utilising *partial-address* decoding [17]. Partial-address decoding can reduce logic requirements by not using all bits. For example, if bits in address `0x0100` do not conflict with bits in other addresses (i.e. bit 8 is high in more than 1 address), then the address decoder needs only concern bit 8, not the other bits. This is visualised in Figure 6.4 below. This method is utilised in the MMU's address decoder (module `vmicro16_mmu` in `vmicro16.v:181`). As this is an optimisation per core, significant resources can be saved when a large number of cores are used.

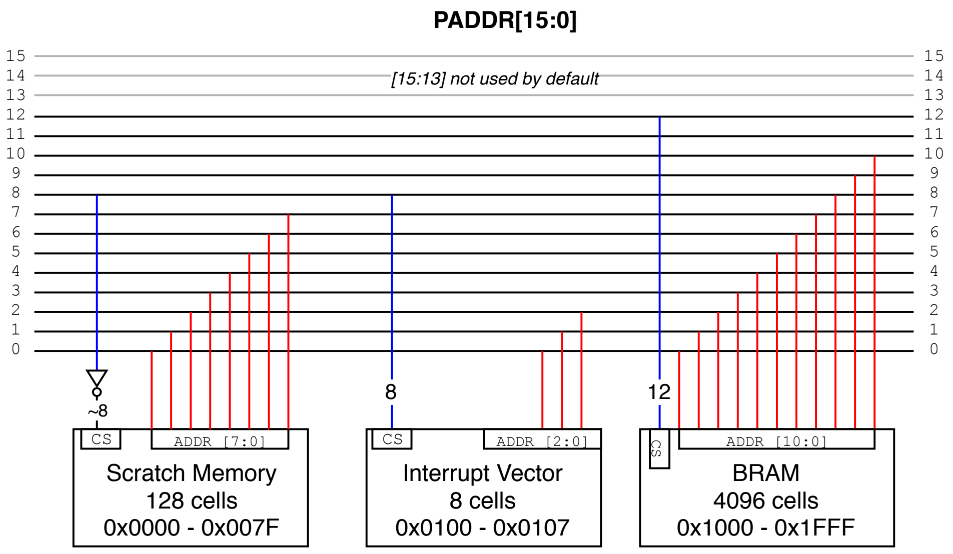


Figure 6.4: Partial address decoding used by the Vmicro16 SoC design. Each peripheral shown only needs to decode a signal bit to determine if it is enabled.

6.3 Memory Map

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

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

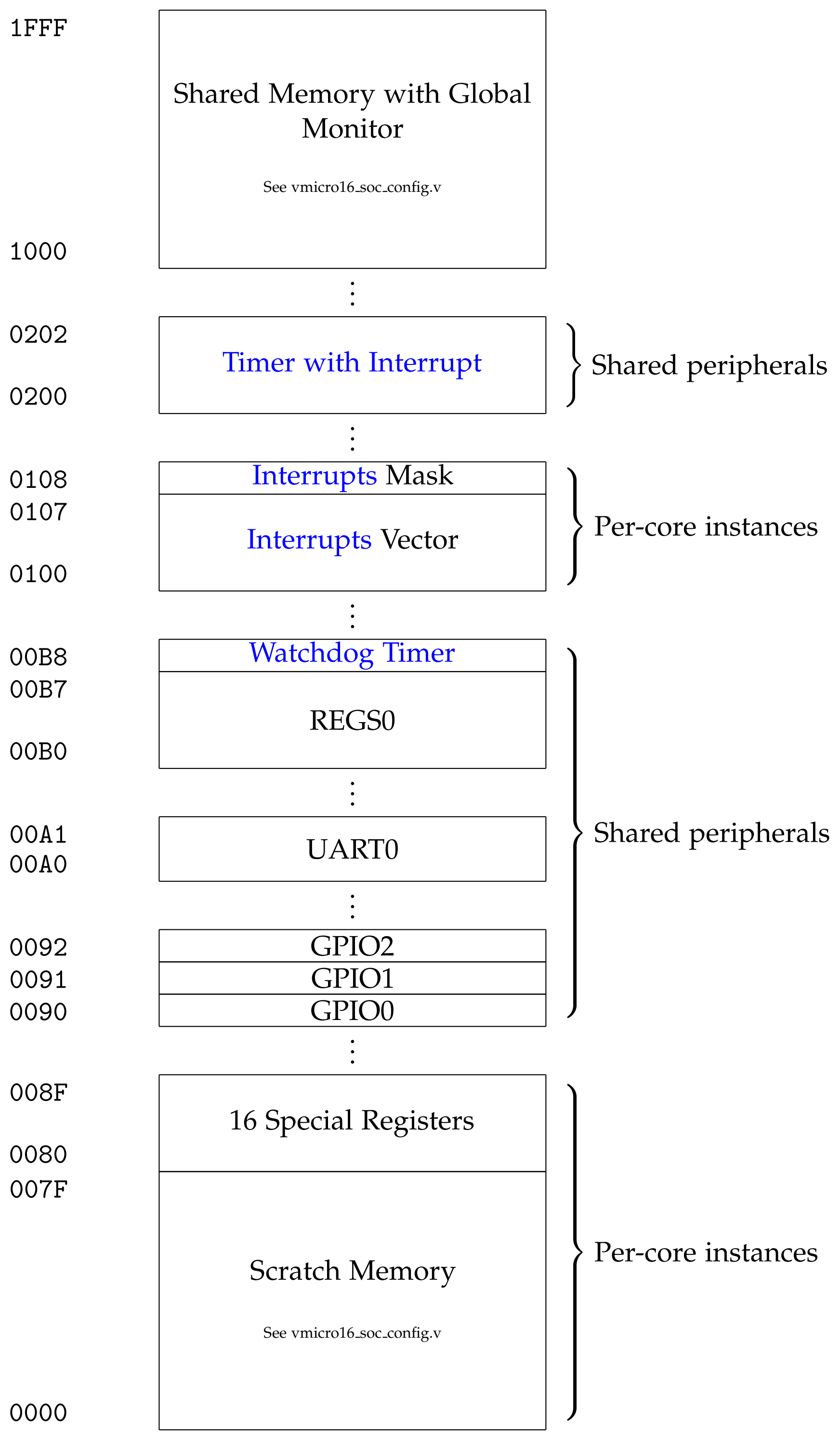


Figure 6.5: Memory map showing addresses of various memory sections.

Chapter 7

Interrupts

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

7.1 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.

7.2 Hardware Implementation

7.2.1 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 7.1 visually describes this process.

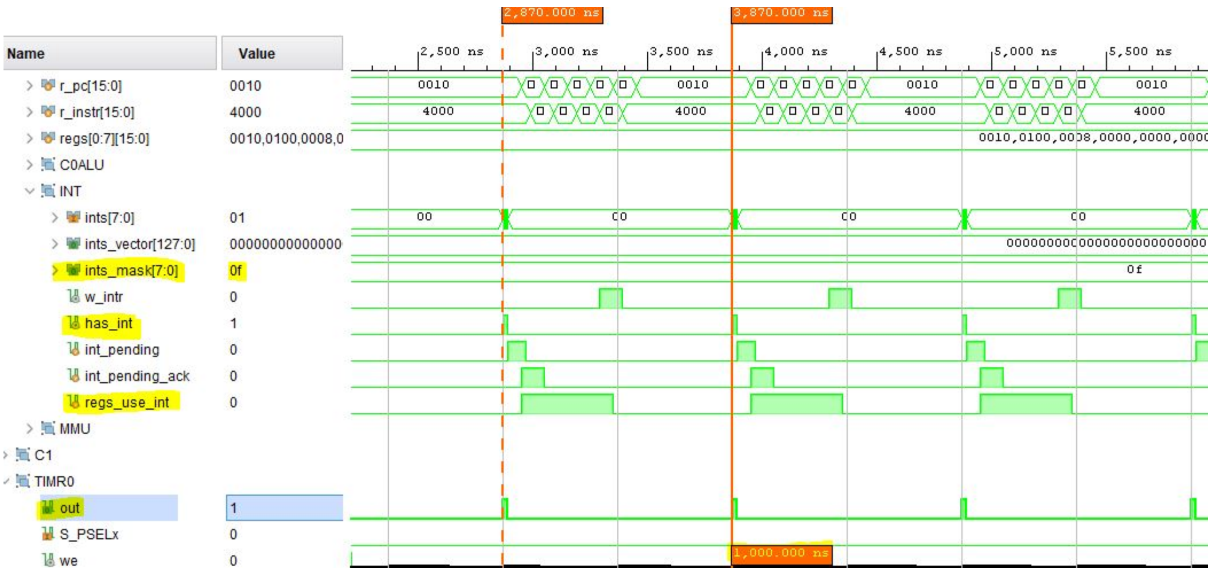


Figure 7.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return instruction (indicated by w_intr) the processor returns to normal mode and restores the normal state.

7.3 Software Interface

To enable software to

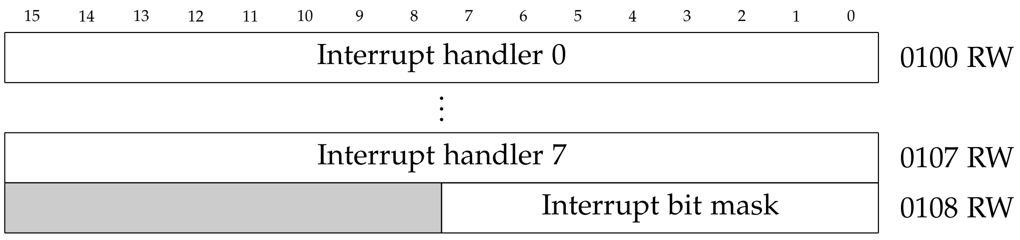


Figure 7.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

7.3.1 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.

7.3.2 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

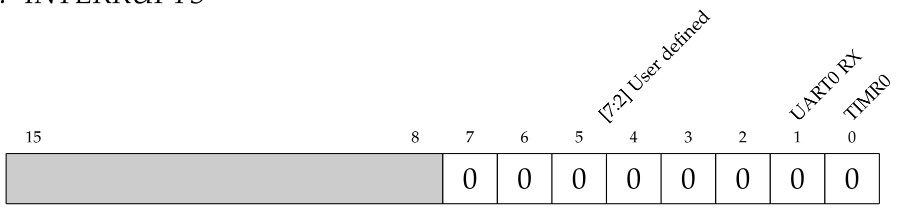


Figure 7.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

allows for multi-processor designs where each core can be used for a particular interrupt source, improving the time response to the interrupt for time critical programs. The Interrupt Mask 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.

7.3.3 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.

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

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

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

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

7.4 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 8

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.

8.1 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.

8.2 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 **??**.

15–0	
Reset Watchdog	00B8 W

8.3 GPIO Interface

15–0	
GPIO0 Output	0090 RW
GPIO1 Output	0091 RW
GPIO1 Input	0092 R

8.4 Timer with Interrupt

15–3	2	1	0	
Load Value				0200 RW
▨	I	R	S	0201 W
Prescaler				0202 W

8.5 UART Interface

15–8	7–2	1	0	
▨	Transmit Data			00A0 W
▨	Receive Data			00A1 R
▨	▨	E	I	00A2 R/W

Chapter 9

Multi-core Communication

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

9.1 Introduction

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

9.1.1 Design Goals

- **Support common synchronisation primitives.**
  Software should be able to implement common synchronisation primitives, such as mutexes, semaphores, and memory barriers, to perform atomic operations and avoid race conditions, which are critical in parallel and concurrent software applications.
- **Context identification.**
  The SoC should expose configuration information such as: the number of processing cores, amount of shared and scratch memory, and the `CORE_ID`, to each thread.

9.1.2 Context Identification

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

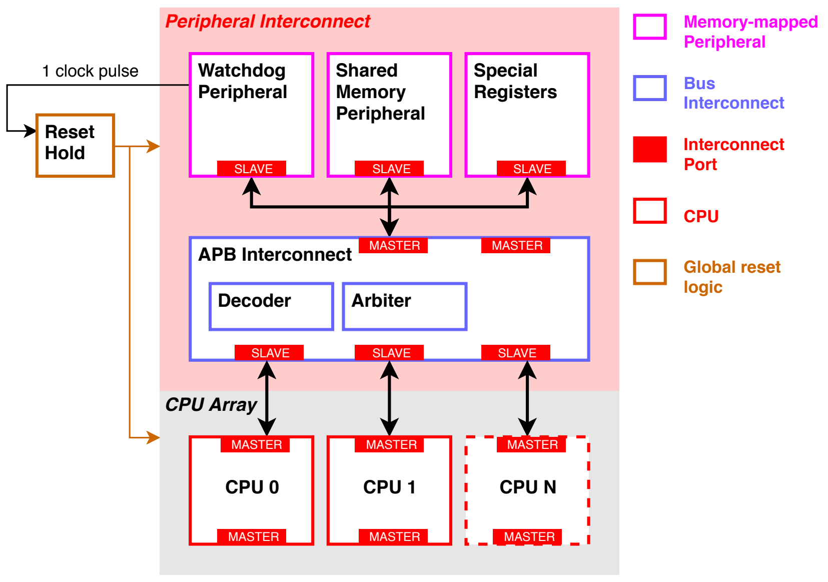


Figure 9.1: Block digram showing the main multi-processing components: the CPU array and a peripheral interconnect used for core synchronisation.

This information is provided through the Special Registers peripheral (0x0080 - 0x008F), shown in Figure 9.1. This register set provides relevant information for writing software that can dynamically scale for various SoC configurations.

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

Figure 9.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

9.1.3 Thread Synchronisation

In multi-threaded software it is important

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

Mutexes

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

In this project, mutexes are provided by the processor through the Shared Memory Peripheral (0x1000 to 0x1FFF) which provides a large RAM-style memory accessible by all cores through the peripheral interconnect bus. This large memory is explicitly defined to use the FPGA's BRAM blocks using Xilinx's Verilog `ram_style="block"` attribute to avoid wasting LUTs when using high core counts. The peripheral allows each memory cell to be *locked*, meaning that only the cell owner can modify it's contents. This is implemented by using another large memory, `locks`, to store the `CORE_ID` + 1 of the owner, as shown in Figure 9.3. In this system, a lock containing the value zero indicates an unlocked cell. As `CORE_ID`s are indexed from zero, one is added to each cell. For example, if core two wants to lock a memory cell, the value three is written to the lock.

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

Figure 9.3: RAM and lock memories instantiated by the shared memory peripheral.

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

Figure 9.4 shows a simple assembly function to lock a memory cell.

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

Figure 9.4: Assembly code for locking a mutex. r1 is the address to lock. r3 is zero. r4 is the branch address.

Barriers

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

The Vmicro16 processor provides barrier synchronisation through the Shared Memory Peripheral. Like the mutex code, the barrier code uses the `LWEX` and `SWEX` instructions to lock a memory cell. Instead of immediately checking the lock as an abstract object, the barrier code treats the cell as a normal memory cell containing a numeric value. Figure 9.5 shows a software example of this. When the `barrier_reached` code is reached, the code will increment the shared memory value by 1, indicating that the number of threads that have reached this point has increased by one (r5). The `barrier_wait` function is then entered which waits until this numeric value (r5) is equal to the number of threads (r7) in the system. If this is true, then all threads have reached the `barrier_wait` function and can continue with normal program execution.

9.2 Design Challenges

9.2.1 Memory Constraints

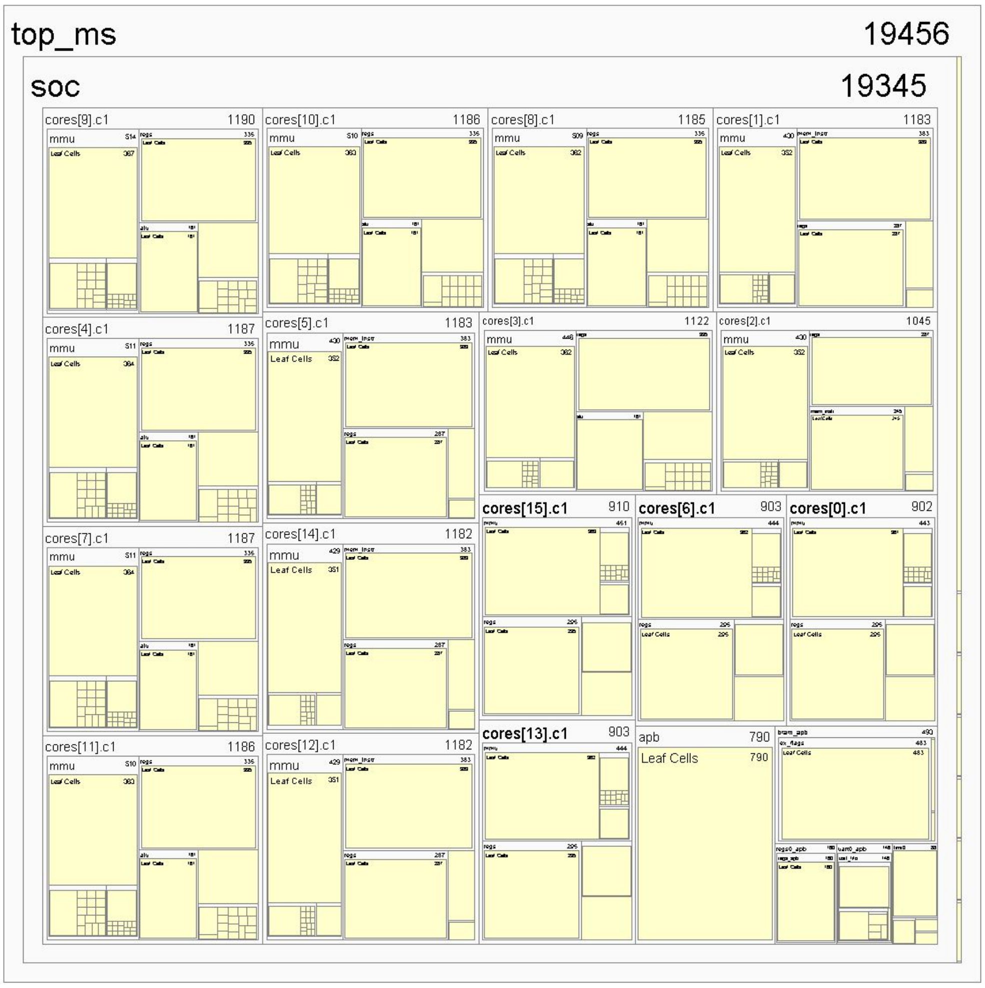


Figure 9.6: ●

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

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

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

Figure 9.5: Assembly code for a memory barrier. Threads will wait in the barrier_wait function until all other threads have reached that code point.

Chapter 10

Analysis & Results

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] 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.

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

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

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