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

ELEC5881M - Final Report

Ben David Lancaster
Student ID: 201280376

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

Supervisor: Dr. David Cowell
Assessor: Mr David Moore

University of Leeds
School of Electrical and Electronic Engineering

July 22, 2019

Word count: 4689

Abstract

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

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

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

Revision History

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

Document revisions.

Declaration of Academic Integrity

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

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

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

Name: Ben David Lancaster
Date: July 22, 2019

Table of Contents

Chapter 1

Memory Mapping

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

1.1 Memory Map

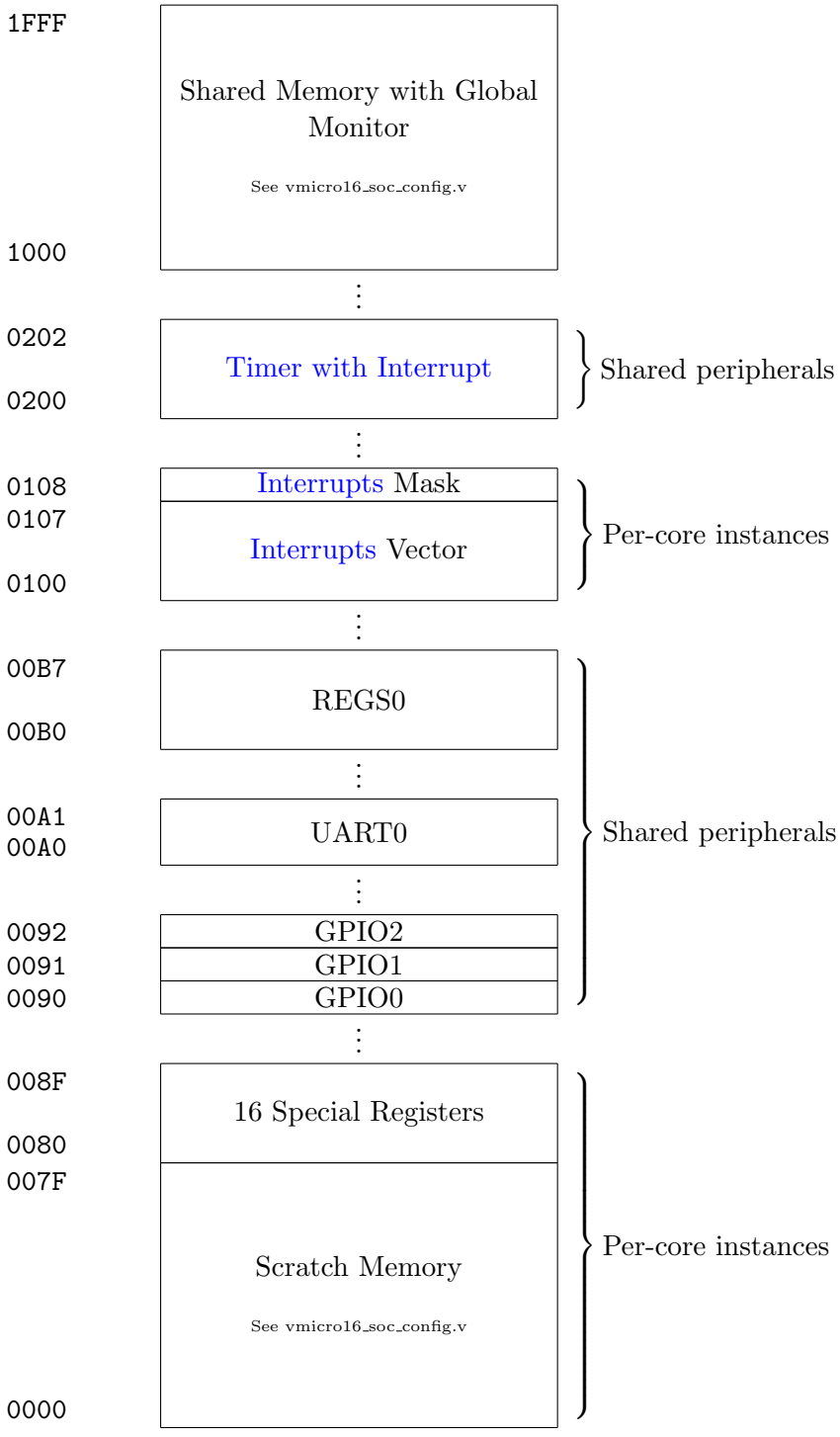


Figure 1.1: Memory map showing addresses of various memory sections.

1.2 Special Registers

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

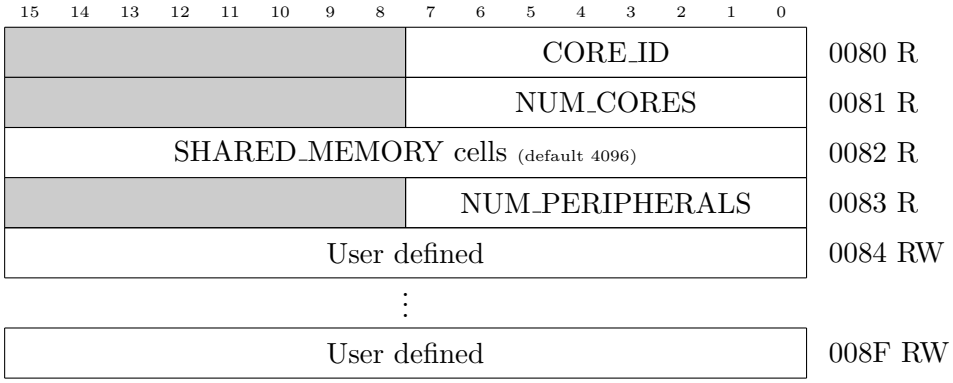


Figure 1.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

Chapter 2

Interrupts

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

2.1 Why Interrupts?

Interrupts are used to enable asynchronous behaviour within a processor.

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

2.2 Hardware Implementation

2.2.1 Context Switching

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

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

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

The timing diagram in Figure 2.1 visually describes this process.

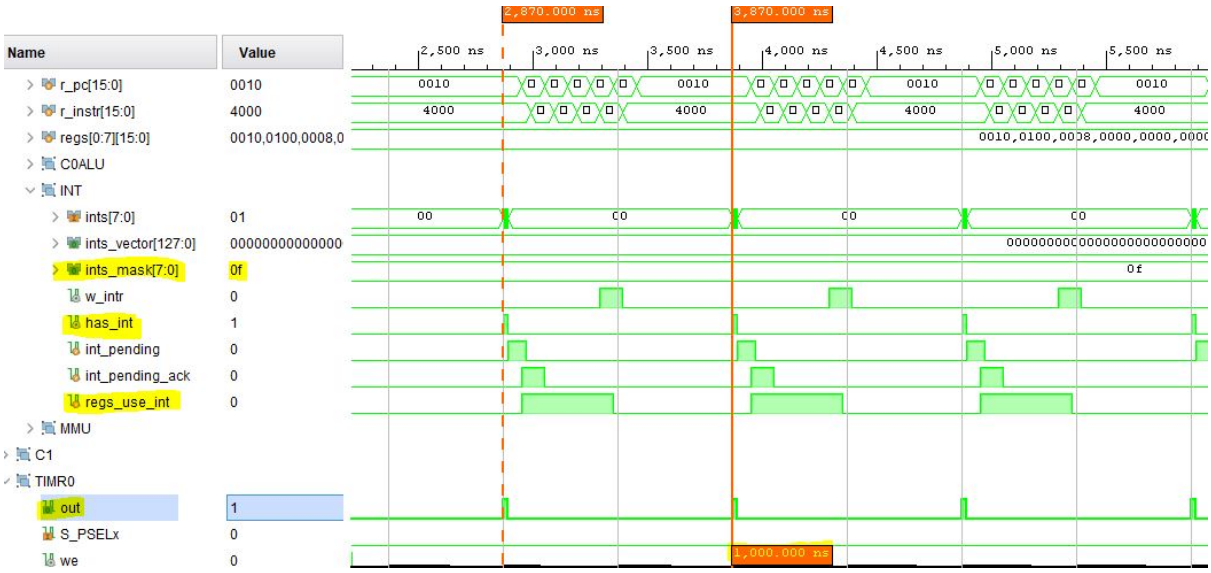


Figure 2.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return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

2.3 Software Interface

To enable software to

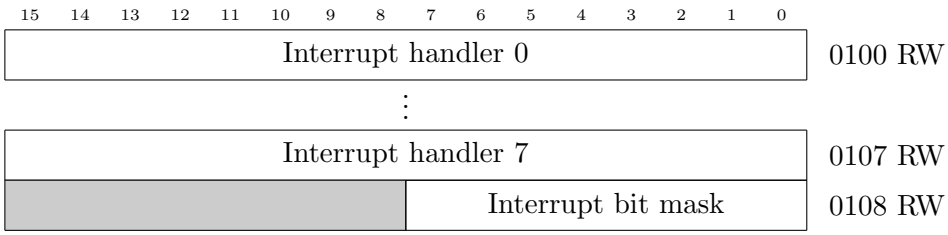


Figure 2.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

2.3.1 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.

2.3.2 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,

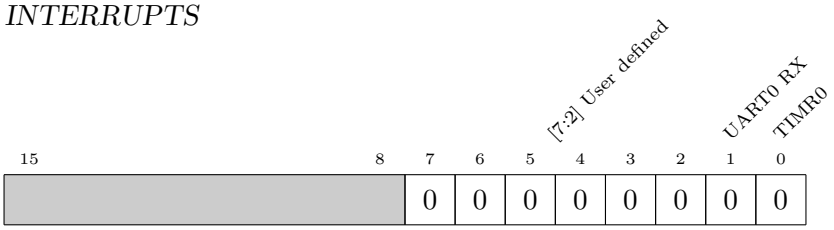


Figure 2.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

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.

2.3.3 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 **??**.

2.4 Design Improvements

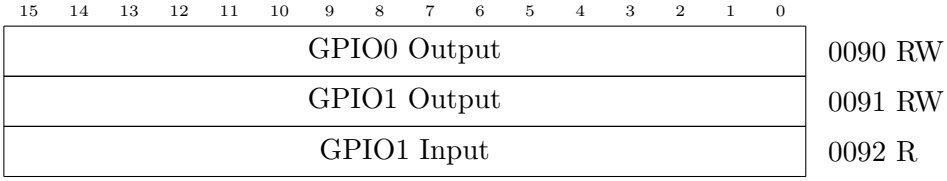
The hardware and software interrupt design have changed throughout the projects cycle. In initial versions of the interrupt implementation, the software program, while waiting for an interrupt, would be in a tight infinite loop (branching to the same instruction). This resulted in the processor using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

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

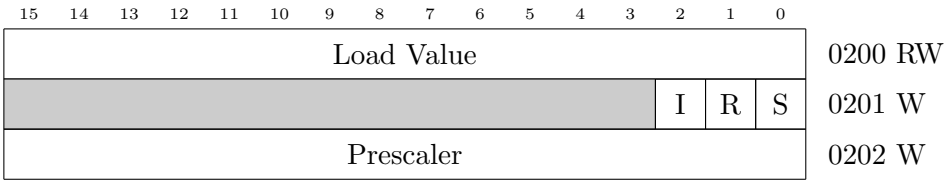
Chapter 3

Peripherals

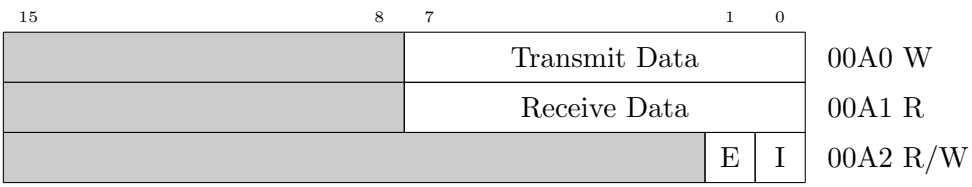
3.1 GPIO Interface



3.2 Timer with Interrupt



3.3 UART Interface



Chapter 4

System-on-Chip Layout

The Vmicro16 processor uses

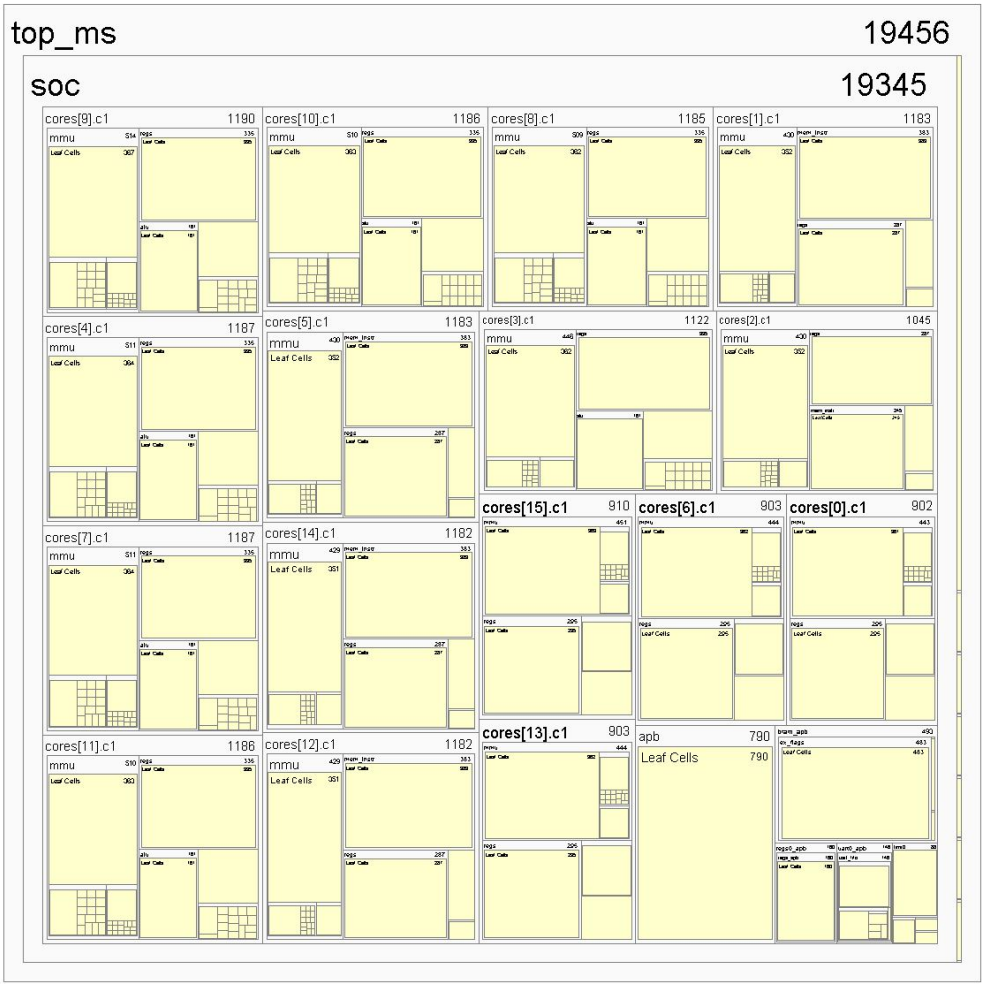


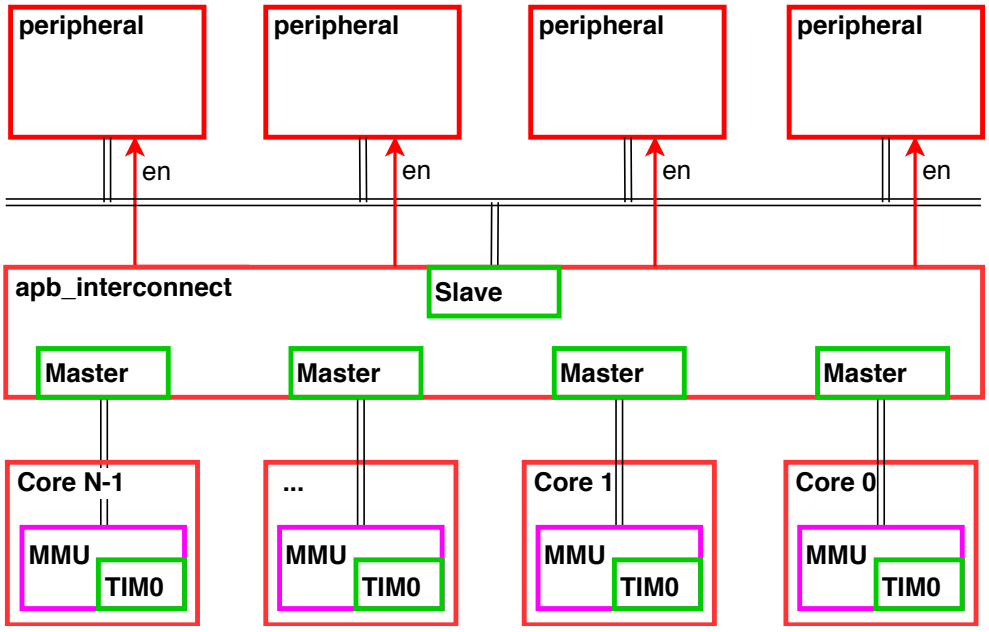
Figure 4.1:

Chapter 5

Interconnect

5.1 Introduction

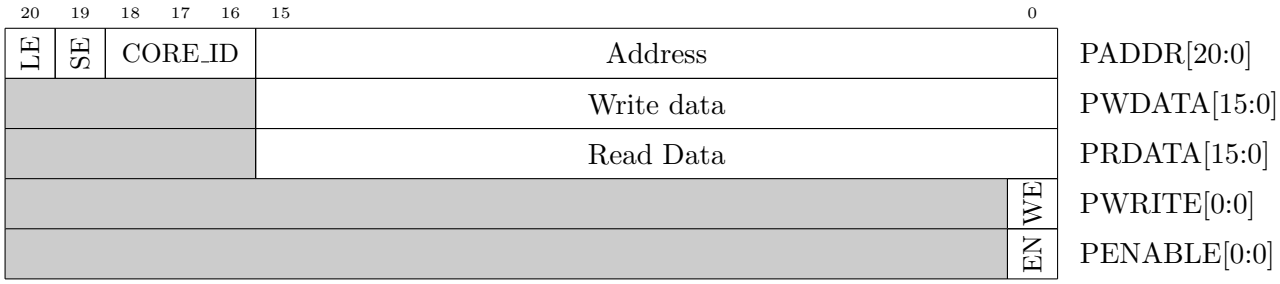
5.2 Overview



5.2.1 Design Considerations

5.3 Interconnect Interface

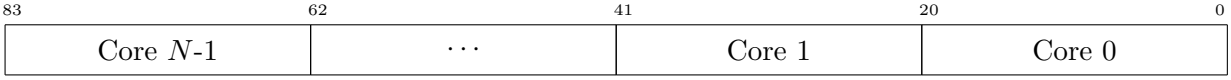
5.3.1 Master to Slave Interface



5.3.2 Variable Core Support

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



5.4 Shared Bus Arbitration

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

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

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	”

Appendix B

Code Listing

B.1 My inspiration

Appendix C

Drawings

C.1 My inspiration BEN

Chapter 6

Introduction interim

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

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

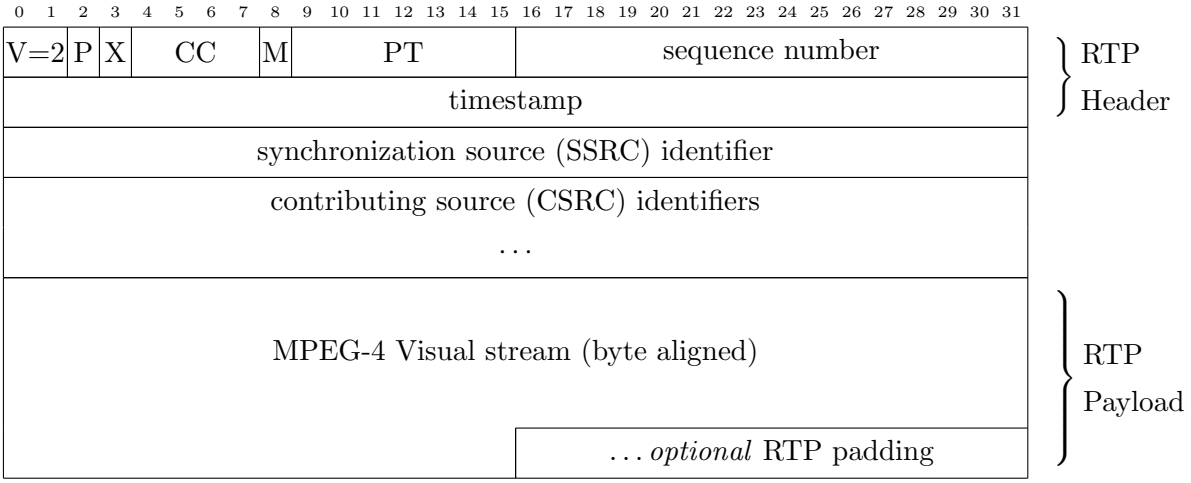


Figure 6.1: Foo

6.1 Why Multi-core?

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

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

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

6.2 Why RISC?

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

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

6.3 Why FPGA?

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

My previous experience with FPGAs in previous projects will reduce risk and learning times and allow for more time to be spent on adding and extending features (discusses further in section 8.1).

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

Chapter 7

Background

7.1 Amdahl's Law and Parallelism

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

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

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

7.2 Loosely and Tightly Coupled Processors

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

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

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

In scenarios where inter-node communication is prominent however, access to the MTS interface must be scheduled to avoid access conflicts which introduces delays and idle times in the software programs execution, resulting in lower throughput. Figure 7.1 shows a general layout of a loosely coupled multiprocessor system.

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

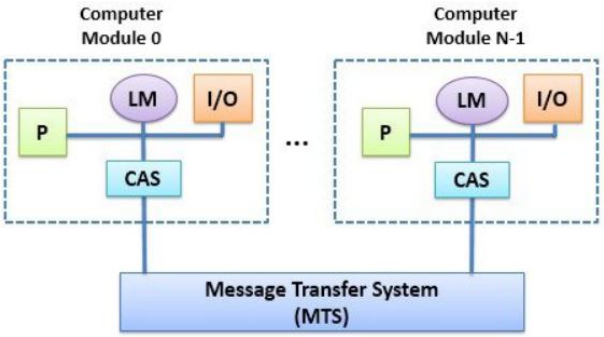


Figure 7.1: A loosely coupled multiprocessor system. Each node features it's own memory and IO modules and uses a Message Transfer System to perform inter-node communication. Image source: [**?**].

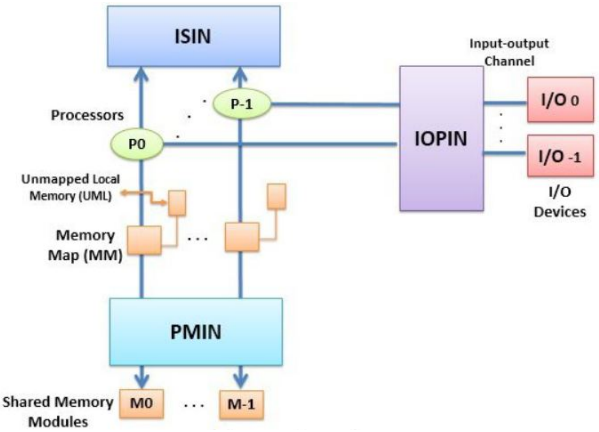


Figure 7.2: A tightly coupled multiprocessor system. Nodes are directly connected to memory and IO modules. Image source: [**?**].

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

7.3 Network-on-chip Architectures

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

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

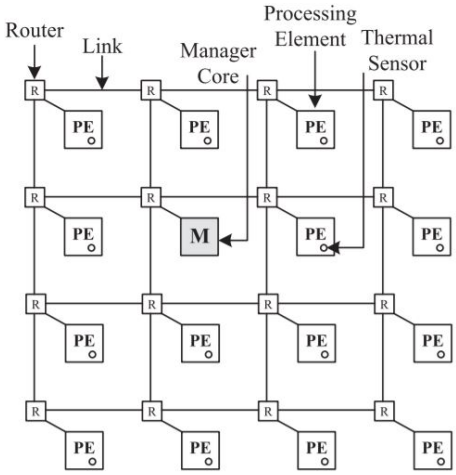


Figure 7.3: A multiprocessor network-on-chip architecture with 16 processing nodes. Nodes are connected in a grid formation with routers and links. Image source: [**?**].

Chapter 8

Project Overview

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

8.1 Project Deliverables

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

8.1.1 Core Deliverables (CD)

The project's core deliverables are described below.

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

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

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

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

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

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

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

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

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

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

8.1.2 Extended Deliverables (ED)

The project's extended deliverables are described below.

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

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

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

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

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

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

ED7 Store instruction memory in SPI flash.

ED8 Reprogram instruction memory at runtime from host computer.

ED9 Processor external debugger using host-processor link.

8.2 Project Timeline

8.2.1 Project Stages

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

The project stages are shown in Table 8.1.

8.2.2 Project Stage Detail

Stages 1.0 through 1.2 – Research and Project Conception

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

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

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

Stages 3.1 through 3.4 – Advanced Processor Implementation

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

Stages 4.1 through 4.4 – Multiprocessor Functionality

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

Stages 5.1 through 5.3 – Debugging Features

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

Stages 6.1 through 6.2 – Compiler Backends

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

Stage 7.1 – Wishbone Peripherals

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

Stage 8.1 – Final Report

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

8.2.3 Timeline

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

8.3 Resources

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

8.3.1 Hardware Resources

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

Terasic DE1-SoC Development Board

The Terasic DE1-SoC development board features a large Cyclone V FPGA and many peripherals, such as seven-segment displays, 64 MB SDRAM, ADCs, and buttons and switches, which will aid demonstration of the project. The development board is available through the

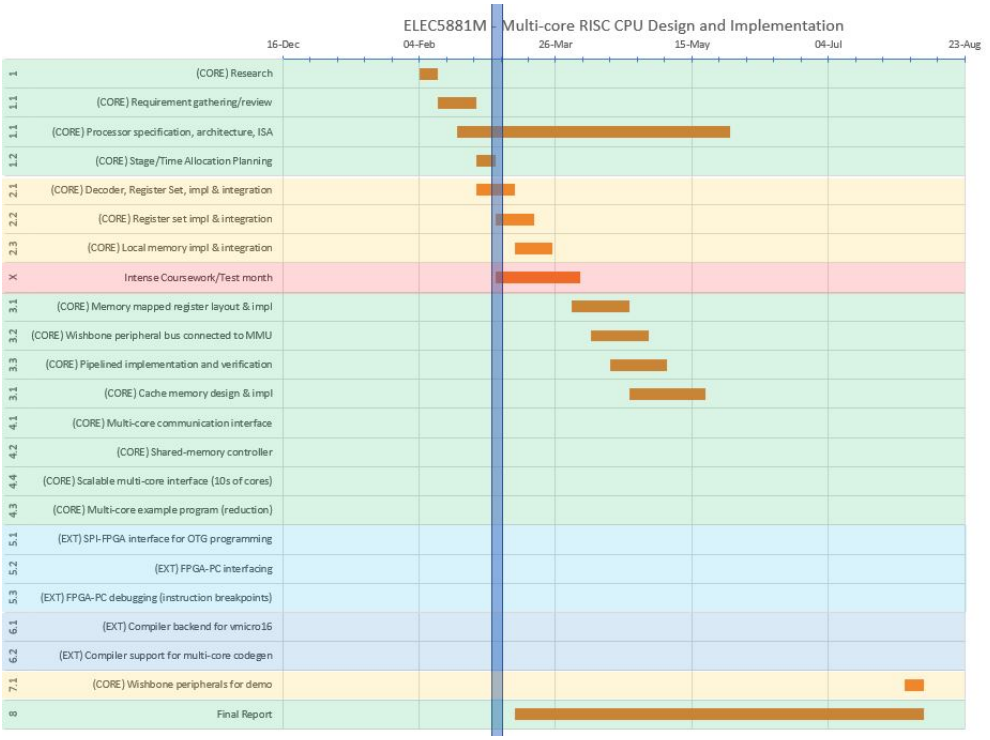


Figure 8.1: Project stages in a Gantt chart.

university so the cost is negligible. Figure 8.2 shows the peripherals (green) available to the FPGA.

Minispartan 6+ FPGA Development Board

The Minispartan 6+ is a hobbyist FGPA development board with fewer peripherals than the DE1-SoC. The board features a Xilinx Spartan 6 XC6LX9 which has far fewer resources than the DE1-SoC's Cyclone V however it's simplicity and my familiarity with Xilinx's software suite will speed up development. The development board is shown in Figure 8.3.

8.3.2 Software Resources

Intel Quartus

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

Xilinx ISE Webpack

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

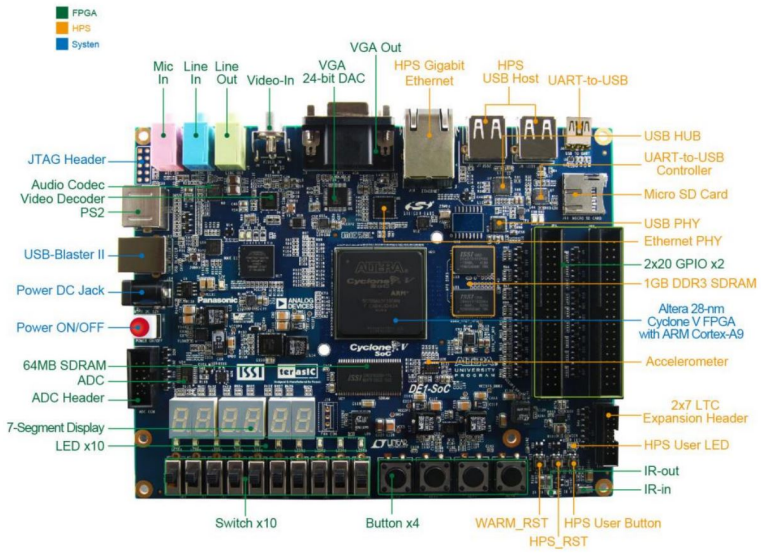


Figure 8.2: Terasic DE1-SoC development board featuring the Altera Cyclone V FPGA and many peripherals. Image source: [**?**].

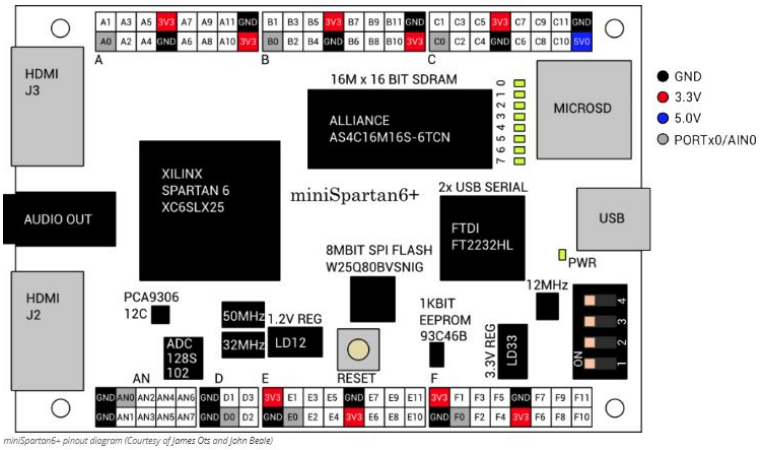


Figure 8.3: Minispartan-6+ development board featuring the Xilinx Spartan 6 XC6SLX9. Note that the XC6SLX9 and XC6SLX25 FPGAs share the same board. Image source: [**?**].

Verilator

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

8.4 Legal and Ethical Considerations

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

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

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

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

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

Chapter 9

Current Progress

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

9.1 RISC Core

Following the project time line described in section 8.2, the first couple months have been dedicated to the design and implementation of the instruction set architecture and RISC core with stages 1-3. Good progress has been made in both deliverables, the ISA and the RISC core, and the progress is on-time with the initial project time line. The core has been nicknamed *Vmicro16* – short for Verilog microprocessor 16-bit.

9.1.1 Instruction Set Architecture

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

Design Goals

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

ISA1 Use a fixed width of 16-bits for all instructions.
This will significantly reduce RTL resources and encourage efficiency by not wasting spare bits. In addition, many SPI flash and RAMs support 16-bit wide data reads

which will allow each instruction fetch to only require one clock cycle, thus increasing processor performance.

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

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

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

As shown in Figure 9.1, frequently used commands such as incrementing/decrementing and logical comparions are provided by setting spare bits to special values. For example, the instructions `ARITH_UADDI` and `ARITH_SSUBI` extend the `ARITH_U` and `ARITH_S` opcodes by filling the spare bit, 4. If this bit is not set (0), the instruction allows for a 4-bit immediate value to be added in addition to the two register selects. The 4-bit immediate allows adding a small number to the ALU which is useful in the case of software for loops where an increment/decrement of more than 1 is required.

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

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

ISA6 Provide instructions for explicitly performing signed and unsigned arithmetic.

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

Without explicit unsigned and signed instructions, extra instructions would be required to perform addition and subtraction. In addition, due to two's complement representation of signed numbers, the highest immediate operand value would be halved, resulting in more instructions to reach the desired value.

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

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

The ISA table is shown in Figure 9.1. The top 5 bits (15-11) are dedicated to the opcode resulting in 32 unique values. Currently only the bits 14-11 are used (`NOP` to `SETC`) leaving the top bit spare. Initially, this bit was reserved to indicate an extended immediate instruction,

`MOVI12`, supporting a large 12-bit immediate value, however later in the design it was decided that the top bit would indicate special instructions dedicated for multi-core operation. This leaves 16 spare unique opcodes for this purpose.

9.1.2 Design and Implementation

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

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

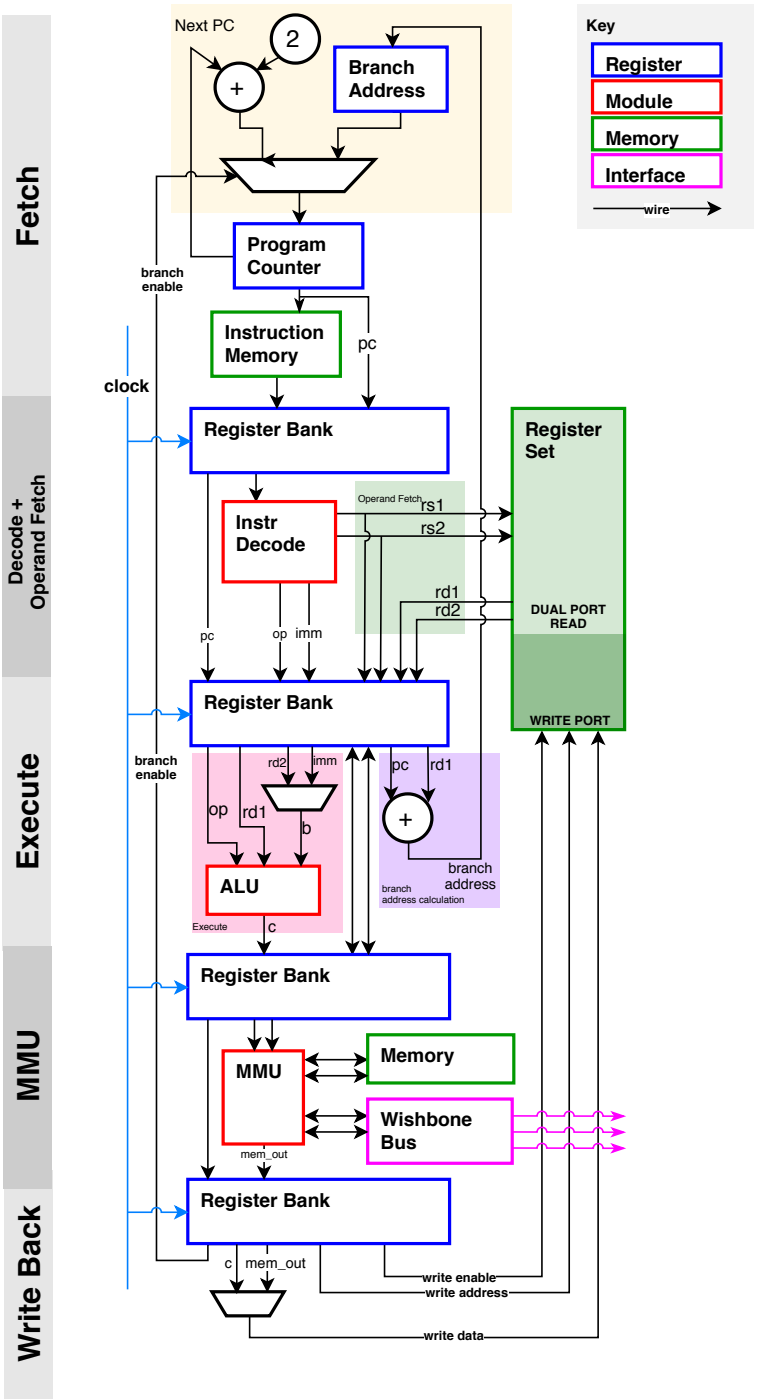


Figure 9.2: Vmicro16 RISC 5-stage RTL diagram showing: instruction pipelining (data passed forward through clocked register banks at each stage); branch address calculation; ALU operand calculation (rd2 or imm); and program counter incrementing.

Instruction and Data Memory

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

Register File

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

Pipelining

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

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

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

This project's RISC processor implements these two hazard detectors and solutions to resolve them. The data hazard resolver implements a `valid` signal that is passed forward from stage to stage. This signal is low when a hazard has occured and indicates that receiving stage should not operate on the previous stage's data. Each stage's `valid` signal is dependant on the previous stages `valid` signal. This allows future stages to stall when a hazard is detected in previous stages. A diagram of the implementation of these hazards in the processor is shown in Figure 9.3.

Memory Management Unit

It was decided to use a memory management unit (MMU) to make it easier and extensible to communicate with external peripherals or additional registers. This method would transparently use the existing `LW/SW` instructions which removes the requirement for a unique instruction for each peripheral.

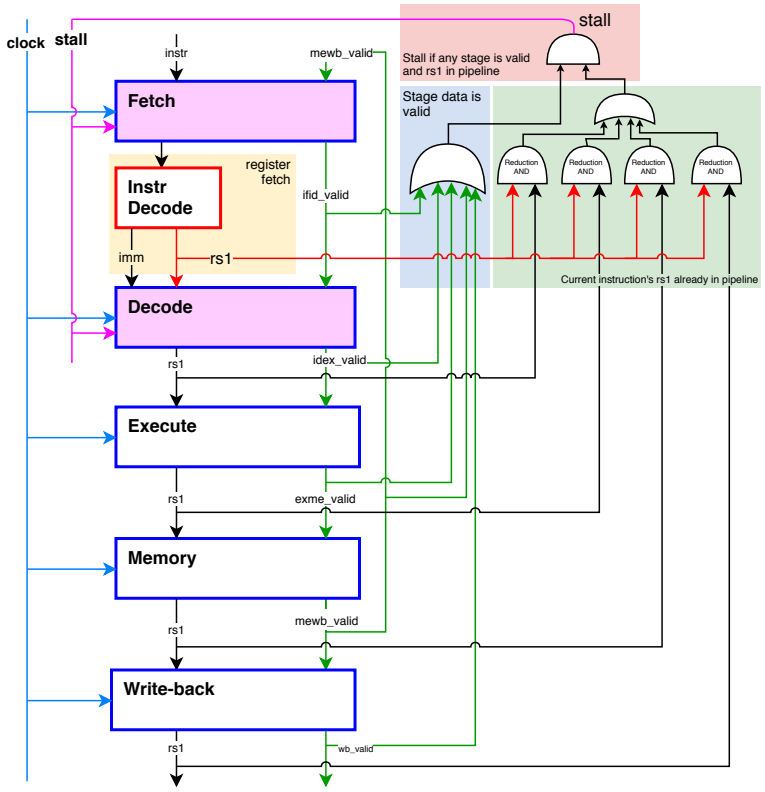


Figure 9.3: Pipeline data hazard detection. The register selects are passed forward through each stage and compared to the IDEX (latest instruction) register selects. If they match, the latest instruction depends on the output of an instruction in the pipeline, the IFID and IDEX stages are stalled to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

Proposed Memory Mapped Addresses

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

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

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

Table 9.1: Provisional memory-mapped addresses table.

ALU Design

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

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

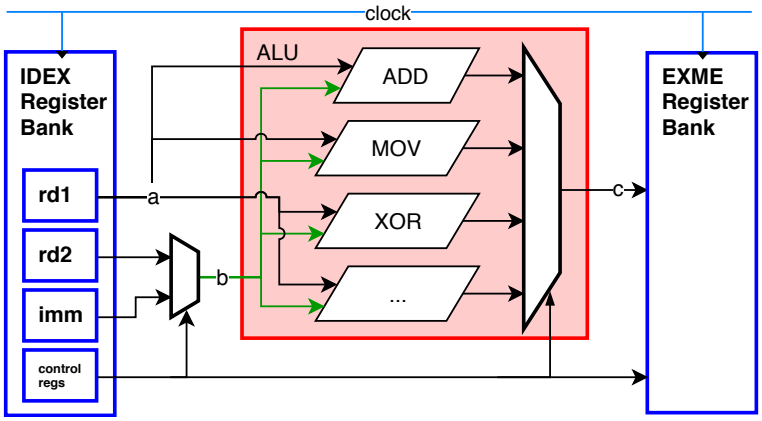


Figure 9.4: Vmicro16 ALU diagram showing clocked inputs from the previous IDEX stage being

The Verilog implementation of the ALU is shown in Figure 9.5. The ALU's asynchronous output is clocked with other registers, such as destination register `rs1` and other control signals, in the `EXME` register bank.

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

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

// APB master to slave interface
always @(posedge clk)
    if (reset) begin
        mmu_state <= MMU_STATE_T1;
```

Figure 9.5: Vmicro16's ALU implementation named vmicro16_alu. vmicro16.v

Decoder Design

Instruction decoding occurs in the between the IFID and IDEX stages. The decoder extracts register selects and operands from the input instruction. The decoder outputs are asynchronous

which allows the register selects to be passed to the register set and register data to be read asynchronously. The register selects and register read data is then clocked into the IDEX register bank.

```
endmodule


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

    parameter CORE_ID       = 3'h0,
    parameter CORE_ID_BITS  = `clog2(`CORES)
) (
    input clk,
    input reset,

    input  req,
    output busy,

    // From core
    input      [MEM_WIDTH-1:0]  mmu_addr,
    input      [MEM_WIDTH-1:0]  mmu_in,
    input                       mmu_we,
    input                       mmu_lwex,
    input                       mmu_swex,
```

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

In Figure 9.6, it can be seen that the first 8 opcode cases are represented using the same 15-11 bits, however the `VMICRO16_OP_BIT` instructions require another bit range to be compared to determine the output opcode.

9.1.3 Verification

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

Known Bugs

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

BUG1 Stall detection does not consider load/store instructions.
Due to instruction pipelining techniques used by the processor and lack of address checking in the `EXME` and `MEWB` stages, `LW` instructions immediately after `SW` instructions:

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

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

Chapter 10

Future Work

10.1 Project Status

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

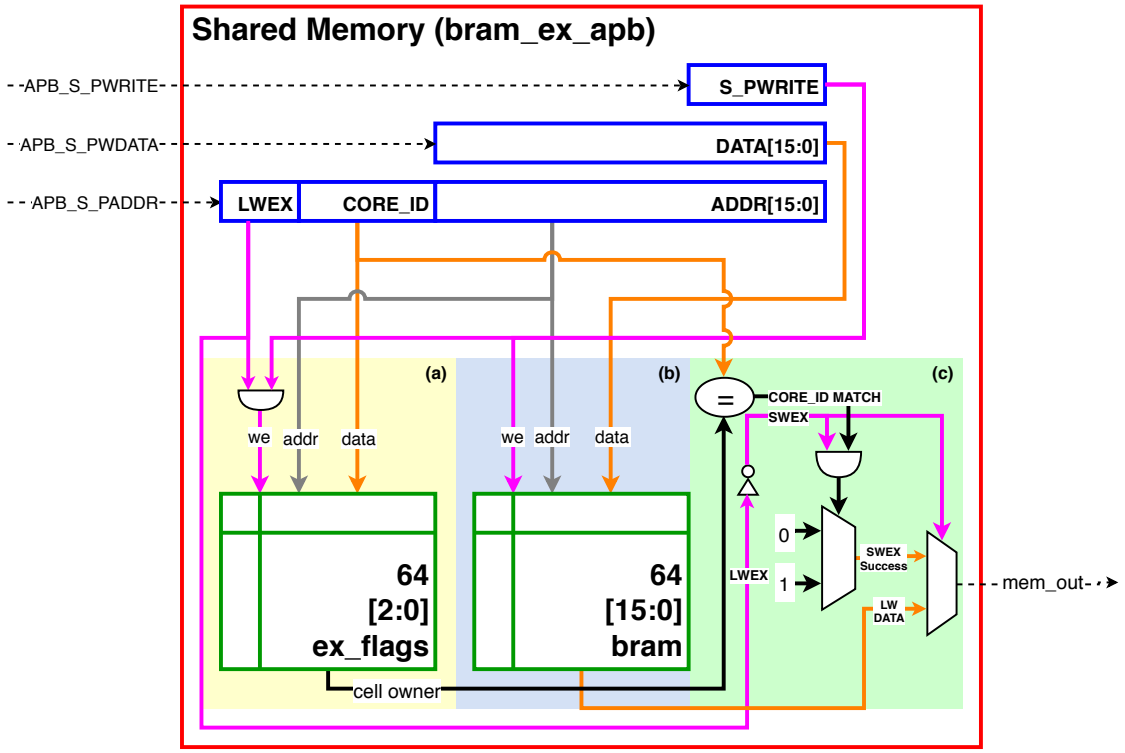


Figure 10.1: Caption for BRAMex

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

utilise this foundation.

10.1.1 Updated Project Time Line

The project table described in section 8.2 did not allocate times for stages 4.1 and later. This was due to expected high demand from other modules and exams in this time period and so it was decided to not allocate times that would later not be followed.

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

The updated project status is shown in Table 10.1 and in Figure 10.2.

10.1.2 Future Work

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

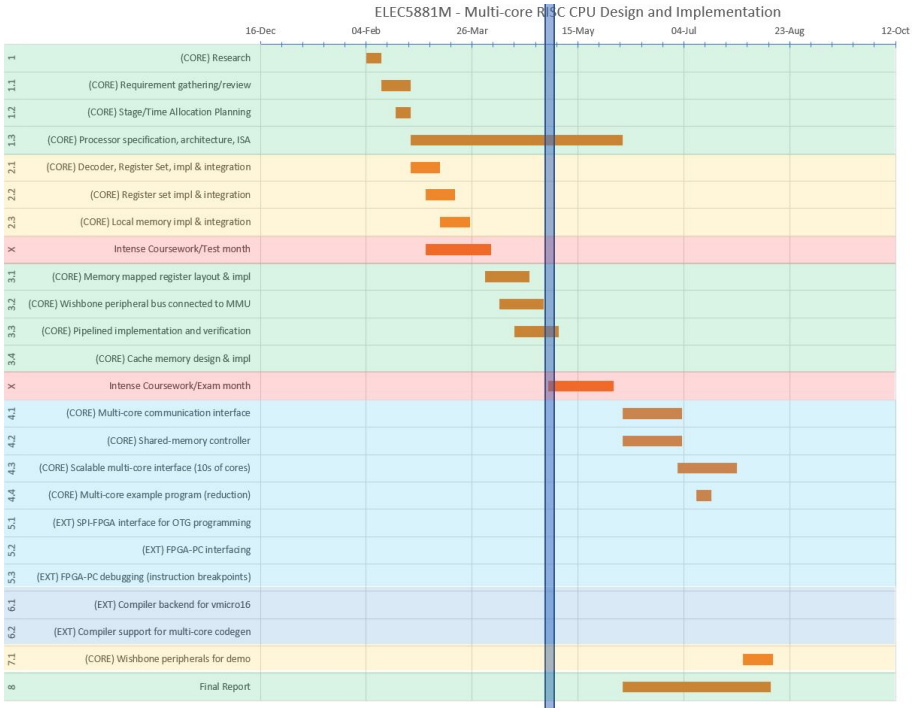


Figure 10.2: Updated project time gantt chart showing time allocations for stage 4.

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

Table 10.1: Updated project stages.

Chapter 11

Conclusion

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

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

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

Appendix A - Code Listing

top_ms.v

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

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

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

    // SSDs
    output [6:0] ssd0,
    output [6:0] ssd1,
    output [6:0] ssd2,
    output [6:0] ssd3,
    output [6:0] ssd4,
    output [6:0] ssd5
);
    //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),

        `ifdef DEF_GLOBAL_RESET
        .reset   ((~SW[0])),
        `else
        .reset   (0),
        `endif

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

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

        //.dbug0   (LEDS[3:0]),
```

apb_intercon.v

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

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

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

    // SSDs
    output [6:0] ssd0,
    output [6:0] ssd1,
    output [6:0] ssd2,
    output [6:0] ssd3,
    output [6:0] ssd4,
    output [6:0] ssd5
);
    //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),

        `ifdef DEF_GLOBAL_RESET
        .reset   ((~SW[0])),
        `else
        .reset   (0),
        `endif

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

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

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

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

The `endmodule`

vmicro16.v

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

```
// 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"

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

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

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

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

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

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

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


// This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
//   https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
//   https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
//   https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf

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
```

vmicro16_soc.v

```
//
//

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

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

    input clk_en,

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

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

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

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

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

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

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

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

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

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

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

// Shared memory with hardware monitor (LWEX/SWEX)
```

vmicro16_isa.v

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

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

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

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

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

// microcode operations
`define VMICRO16_ALU_BIT_OR      5'h00
`define VMICRO16_ALU_BIT_XOR     5'h01
`define VMICRO16_ALU_BIT_AND     5'h02
`define VMICRO16_ALU_BIT_NOT     5'h03
`define VMICRO16_ALU_BIT_LSHFT   5'h04
`define VMICRO16_ALU_BIT_RSHFT   5'h05
`define VMICRO16_ALU_LW          5'h06
`define VMICRO16_ALU_SW          5'h07
`define VMICRO16_ALU_NOP         5'h08
`define VMICRO16_ALU_MOV         5'h09
`define VMICRO16_ALU_MOVI        5'h0a
`define VMICRO16_ALU_MOVI_L      5'h0b
`define VMICRO16_ALU_ARITH_UADD  5'h0c
`define VMICRO16_ALU_ARITH_USUB  5'h0d
`define VMICRO16_ALU_ARITH_SADD  5'h0e
`define VMICRO16_ALU_ARITH_SSUB  5'h0f
`define VMICRO16_ALU_BR_U        5'h10
`define VMICRO16_ALU_BR_E        5'h11
`define VMICRO16_ALU_BR_NE       5'h12
`define VMICRO16_ALU_BR_G        5'h13
`define VMICRO16_ALU_BR_GE       5'h14
`define VMICRO16_ALU_BR_L        5'h15
`define VMICRO16_ALU_BR_LE       5'h16
`define VMICRO16_ALU_BR_S        5'h17
`define VMICRO16_ALU_BR_NS       5'h18
`define VMICRO16_ALU_CMP         5'h19
`define VMICRO16_ALU_SETC        5'h1a
`define VMICRO16_ALU_ARITH_UADDI 5'h1b
`define VMICRO16_ALU_ARITH_SSUBI 5'h1c
`define VMICRO16_ALU_BR          5'h1d
`ifdef DEF_ALU_HW_MULT
`define VMICRO16_ALU_MULT        5'h1e
`endif
`define VMICRO16_ALU_BAD         5'h1f
```