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 26, 2019

Word count: 4689

Abstract

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

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

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

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 26, 2019

Table of Contents

1 Memory Mapping **5**
- 1.1 Memory Map . . . 6
- 1.2 Special Registers . . . 7

2 Interrupts **8**
- 2.1 Why Interrupts? . . . 8
- 2.2 Hardware Implementation . . . 8
  - 2.2.1 Context Switching . . . 8
- 2.3 Software Interface . . . 9
  - 2.3.1 Interrupt Vector (0x0100-0x0107) . . . 9
  - 2.3.2 Interrupt Mask (0x0108) . . . 9
  - 2.3.3 Software Example . . . 10
- 2.4 Design Improvements . . . 10

3 Peripherals **11**
- 3.1 GPIO Interface . . . 11
- 3.2 Timer with Interrupt . . . 11
- 3.3 UART Interface . . . 11

4 System-on-Chip Layout **12**

5 Interconnect **13**
- 5.1 Introduction . . . 13
- 5.2 Overview . . . 13
  - 5.2.1 Design Considerations . . . 14
- 5.3 Peripheral Interconnect Interface . . . 14
  - 5.3.1 Master to Slave Interface . . . 14
  - 5.3.2 Variable Core Support . . . 14
- 5.4 Shared Bus Arbitration . . . 14

6 Analysis & Results **16**

7 Improvements **17**
- 7.1 Foo . . . 17

8 Conclusion **18**
8.1 Foo 18

Appendices **19**

A Configuration Options **19**
A.1 SoC Options 19
A.2 Core Options 19
A.3 Peripheral Options 20

B Code Listing **21**
B.1 vmicro16_soc_config.v 21
B.2 top_ms.v 22
B.3 vmicro16_soc.v 23
B.4 vmicro16_periph.v 28
B.5 vmicro16.v 33

Chapter 1

Memory Mapping

1.1 Memory Map 6
1.2 Special Registers 7

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

1.1 Memory Map

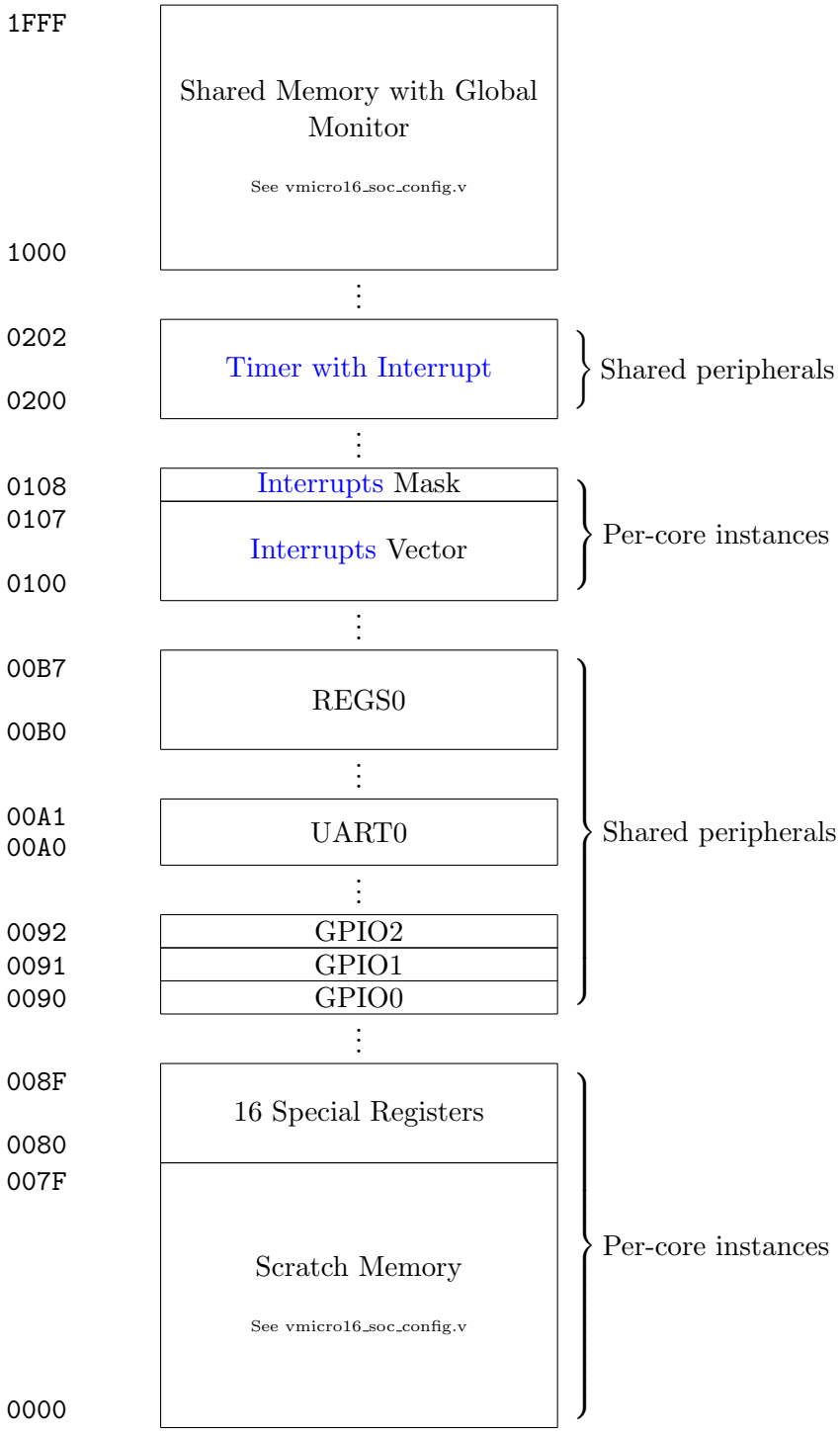


Figure 1.1: Memory map showing addresses of various memory sections.

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.

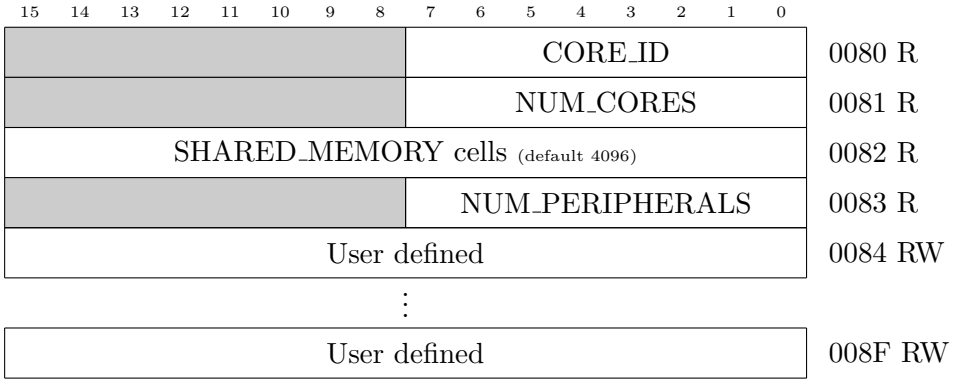


Figure 1.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

Chapter 2

Interrupts

2.1 Why Interrupts? 8
2.2 Hardware Implementation 8
2.2.1 Context Switching 8
2.3 Software Interface 9
2.3.1 Interrupt Vector (0x0100-0x0107) 9
2.3.2 Interrupt Mask (0x0108) 9
2.3.3 Software Example 10
2.4 Design Improvements 10

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.

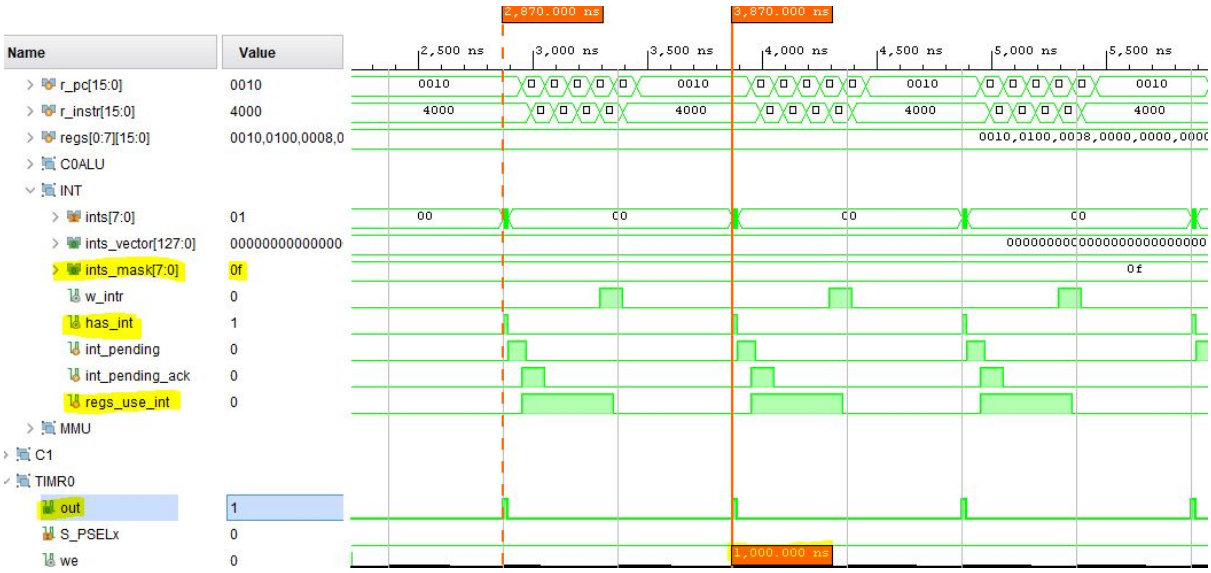


Figure 2.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return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

2.3 Software Interface

To enable software to

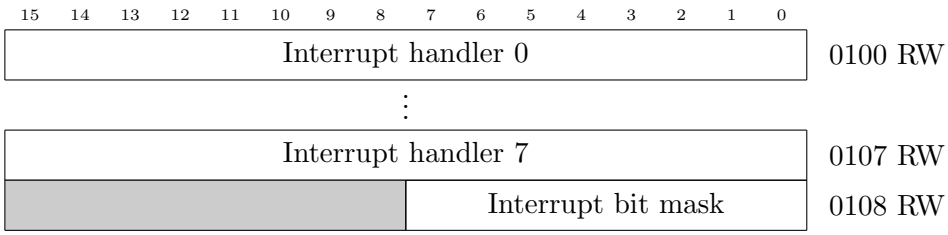


Figure 2.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

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,

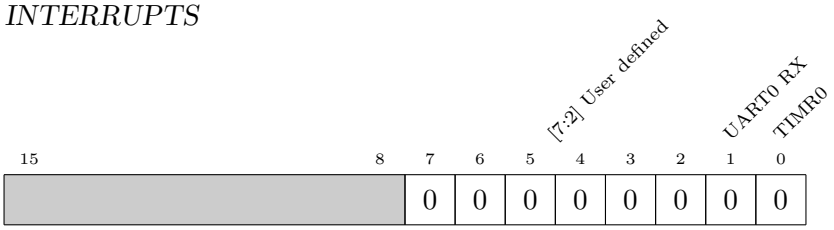


Figure 2.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

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.

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

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

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.

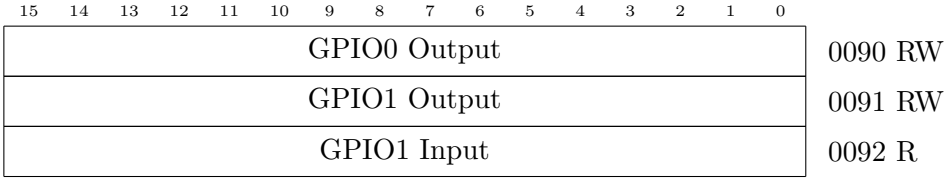
To improve this, it was decided to implement a new state within the processor's state machine that, when entered, did not produce high frequency logic transitions or memory fetches. The `HALT` instruction was modified to enter this state and the only way to leave is from an interrupt or top-level reset. 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

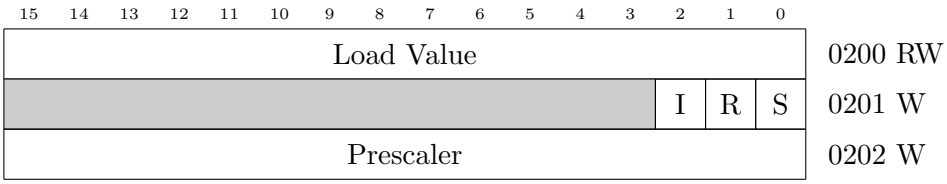
Peripherals

3.1 GPIO Interface 11
3.2 Timer with Interrupt 11
3.3 UART Interface 11

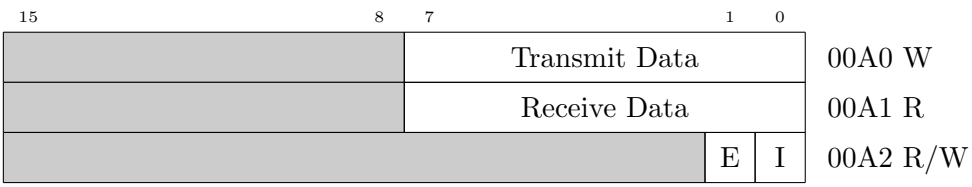
3.1 GPIO Interface



3.2 Timer with Interrupt



3.3 UART Interface



Chapter 4

System-on-Chip Layout

The Vmicro16 processor uses

Figure 4.1:

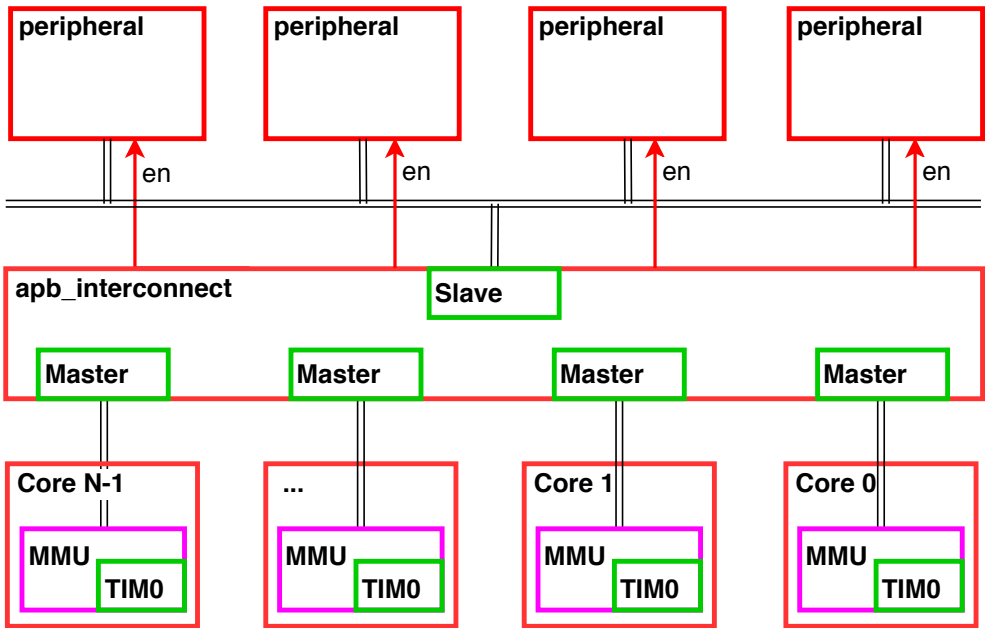
Chapter 5

Interconnect

5.1 Introduction 13
5.2 Overview 13
5.2.1 Design Considerations 14
5.3 Peripheral Interconnect Interface 14
5.3.1 Master to Slave Interface 14
5.3.2 Variable Core Support 14
5.4 Shared Bus Arbitration 14

5.1 Introduction

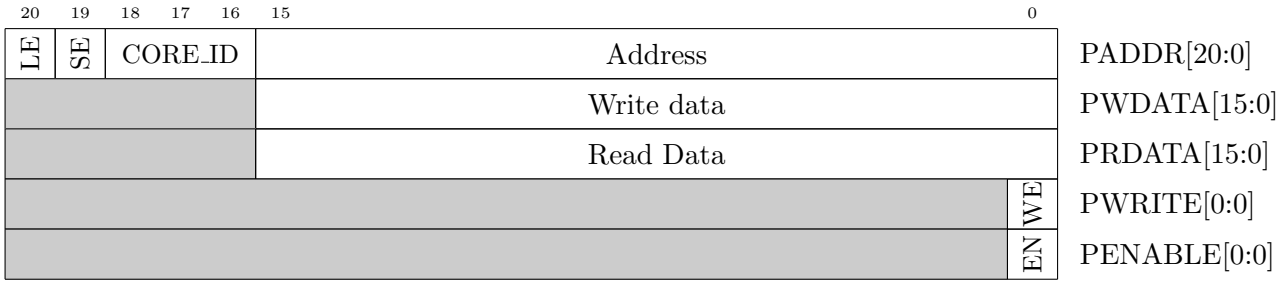
5.2 Overview



5.2.1 Design Considerations

5.3 Peripheral Interconnect Interface

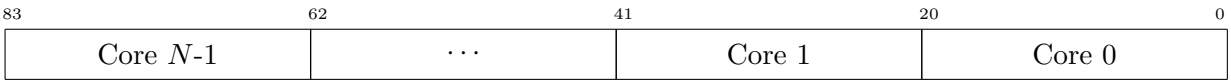
5.3.1 Master to Slave Interface



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

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis

natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Chapter 6

Analysis & Results

Chapter 7

Improvements

7.1 Foo 17

7.1 Foo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Chapter 8

Conclusion

8.1 Foo 18

8.1 Foo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Appendix A

Configuration Options

A.1 SoC Options 19
A.2 Core Options 19
A.3 Peripheral Options 20

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

Table A.1: SoC Configuration Options

A.2 Core Options

Macro	Default	Purpose
DATA_WIDTH	16	Width of CPU registers in bits
DEF_CORE_HAS_INSTR_MEM	//	Enable a per core instruction memory cache
DEF_MEM_INSTR_DEPTH	64	Instruction memory cache per core
DEF_MEM_SCRATCH_DEPTH	64	RW RAM per core
DEF_ALU_HW_MULT	1	Enable/disable HW multiply (1 clock)
FIX_T3	//	Enable a T3 state for the APB transaction

Table A.2: Core Options

A.3 Peripheral Options

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

Table A.3: Peripheral Options

Appendix B

Code Listing

B.1 vmicro16_soc_config.v 21
B.2 top_ms.v 22
B.3 vmicro16_soc.v 23
B.4 vmicro16_periph.v 28
B.5 vmicro16.v 33

B.1 vmicro16_soc_config.v

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

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

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

B.2 top_ms.v

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

```
1  module seven_display # (
2      parameter INVERT = 1
3  ) (
4      input  [3:0] n,
5      output [6:0] segments
6  );
7      reg [6:0] bits;
8      assign segments = (INVERT ? ~bits : bits);
9
10     always @(n)
11     case (n)
```

```
12          4'h0: bits = 7'b0111111; // 0
13          4'h1: bits = 7'b0000110; // 1
14          4'h2: bits = 7'b1011011; // 2
15          4'h3: bits = 7'b1001111; // 3
16          4'h4: bits = 7'b1100110; // 4
17          4'h5: bits = 7'b1101101; // 5
18          4'h6: bits = 7'b1111101; // 6
19          4'h7: bits = 7'b0000111; // 7
20          4'h8: bits = 7'b1111111; // 8
21          4'h9: bits = 7'b1100111; // 9
22          4'hA: bits = 7'b1110111; // A
23          4'hB: bits = 7'b1111100; // B
24          4'hC: bits = 7'b0111001; // C
25          4'hD: bits = 7'b1011110; // D
26          4'hE: bits = 7'b1111001; // E
27          4'hF: bits = 7'b1110001; // F
28      endcase
29  endmodule
30
31  // minispartan6+ XC6SLX9
32  module top_ms # (
33      parameter GPIO_PINS = 8
34  ) (
35      input           CLK50,
36      input  [3:0]    SW,
37      // UART
38      //input           RXD,
39      output          TXD,
40      // Peripherals
41      output [7:0]    LEDS,
42
43      // SSDs
44      output [6:0] ssd0,
45      output [6:0] ssd1,
46      output [6:0] ssd2,
47      output [6:0] ssd3,
48      output [6:0] ssd4,
49      output [6:0] ssd5
50  );
51      //wire [15:0]         M_PADDR;
52      //wire                M_PWRITE;
53      //wire [5-1:0]        M_PSELx;  // not shared
54      //wire                M_PENABLE;
55      //wire [15:0]         M_PWDATA;
56      //wire [15:0]         M_PRDATA; // input to intercon
57      //wire                M_PREADY; // input to intercon
58
59      wire [7:0]  gpio0;
60      wire [15:0] gpio1;
61      wire [7:0]  gpio2;
62
63
64      vmicro16_soc soc (
65          .clk     (CLK50),
66
67          `ifdef DEF_GLOBAL_RESET
68          .reset   ((~SW[0])),
69          `else
70          .reset   (0),
71          `endif
72
73          //.M_PADDR    (M_PADDR),
74          //.M_PWRITE   (M_PWRITE),
75          //.M_PSELx    (M_PSELx),
76          //.M_PENABLE  (M_PENABLE),
77          //.M_PWDATA   (M_PWDATA),
78          //.M_PRDATA   (M_PRDATA),
79          //.M_PREADY   (M_PREADY),
80
81          .uart_tx (TXD),
82          .gpio0   (LEDS[3:0]),
83          .gpio1   (gpio1),
84          .gpio2   (gpio2),
85
86          //.dbug0   (LEDS[3:0]),
87          .dbug1   (LEDS[7:4])
88      );
89
90      // SSD displays (split across 2 gpio ports 1 and 2)
91      wire [3:0] ssd_chars [0:5];
92      assign ssd_chars[0] = gpio1[3:0];
93      assign ssd_chars[1] = gpio1[7:4];
94      assign ssd_chars[2] = gpio1[11:8];
95      assign ssd_chars[3] = gpio1[15:12];
96      assign ssd_chars[4] = gpio2[3:0];
97      assign ssd_chars[5] = gpio2[7:4];
98      seven_display ssd_0 (.n(ssd_chars[0]), .segments (ssd0));
99      seven_display ssd_1 (.n(ssd_chars[1]), .segments (ssd1));
100     seven_display ssd_2 (.n(ssd_chars[2]), .segments (ssd2));
101     seven_display ssd_3 (.n(ssd_chars[3]), .segments (ssd3));
102     seven_display ssd_4 (.n(ssd_chars[4]), .segments (ssd4));
```

```
103     seven_display ssd_5 (.n(ssd_chars[5]), .segments (ssd5));
104
105 endmodule
```

B.3 vmicro16_soc.v

```
1   //
2   //
3
4   `include "vmicro16_soc_config.v"
5   `include "clog2.v"
6   `include "formal.v"
7
8   // Vmicro16 multi-core SoC with various peripherals
9   // and interrupts
10  module vmicro16_soc (
11      input clk,
12      input reset,
13
14      //input  uart_rx,
15      output                          uart_tx,
16      output [`APB_GPIO0_PINS-1:0]    gpio0,
17      output [`APB_GPIO1_PINS-1:0]    gpio1,
18      output [`APB_GPIO2_PINS-1:0]    gpio2,
19
20      output                          halt,
21
22      output      [`CORES-1:0]        dbug0,
23      output      [`CORES*8-1:0]      dbug1
24  );
25      genvar di;
26      generate for(di = 0; di < `CORES; di = di + 1) begin : gen_dbug0
27          assign dbug0[di] = dbug1[di*8];
28      end
29      endgenerate
30
31      wire [`CORES-1:0] w_halt;
32      assign halt = &w_halt;
33
34      // Peripherals (master to slave)
35       wire [`APB_WIDTH-1:0]          M_PADDR;
36       wire                           M_PWRITE;
37       wire [`SLAVES-1:0]             M_PSELx;  // not shared
38       wire                           M_PENABLE;
39       wire [`DATA_WIDTH-1:0]         M_PWDATA;
40       wire [`SLAVES*`DATA_WIDTH-1:0] M_PRDATA; // input to intercon
41       wire [`SLAVES-1:0]             M_PREADY; // input
42
43      // Master apb interfaces
44       wire [`CORES*`APB_WIDTH-1:0]   w_PADDR;
45       wire [`CORES-1:0]              w_PWRITE;
46       wire [`CORES-1:0]              w_PSELx;
47       wire [`CORES-1:0]              w_PENABLE;
48       wire [`CORES*`DATA_WIDTH-1:0]  w_PWDATA;
49       wire [`CORES*`DATA_WIDTH-1:0]  w_PRDATA;
50       wire [`CORES-1:0]              w_PREADY;
51
52      // Interrupts
53  `ifdef DEF_ENABLE_INT
54      wire [`DEF_NUM_INT-1:0]             ints;
55      wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_data;
56      assign ints[7:1] = 0;
57      assign ints_data[`DEF_NUM_INT*`DATA_WIDTH-1:`DATA_WIDTH] =
58                  {`DEF_NUM_INT*(`DATA_WIDTH-1){1'b0}};
59  `endif
60
61      apb_intercon_s # (
62          .MASTER_PORTS   (`CORES),
63          .SLAVE_PORTS    (`SLAVES),
64          .BUS_WIDTH      (`APB_WIDTH),
65          .DATA_WIDTH     (`DATA_WIDTH),
66          .HAS_PSELX_ADDR (1)
67      ) apb (
68          .clk        (clk),
69          .reset      (reset),
70          // APB master to slave
71          .S_PADDR    (w_PADDR),
72          .S_PWRITE   (w_PWRITE),
73          .S_PSELx    (w_PSELx),
74          .S_PENABLE  (w_PENABLE),
75          .S_PWDATA   (w_PWDATA),
76          .S_PRDATA   (w_PRDATA),
77          .S_PREADY   (w_PREADY),
78          // shared bus
79          .M_PADDR    (M_PADDR),
80          .M_PWRITE   (M_PWRITE),
81          .M_PSELx    (M_PSELx),
82          .M_PENABLE  (M_PENABLE),
```

```
83          .M_PWDATA   (M_PWDATA),
84          .M_PRDATA   (M_PRDATA),
85          .M_PREADY   (M_PREADY)
86      );
87
88      vmicro16_gpio_apb # (
89          .BUS_WIDTH  (`APB_WIDTH),
90          .DATA_WIDTH (`DATA_WIDTH),
91          .PORTS      (`APB_GPIO0_PINS),
92          .NAME       ("GPIO0")
93      ) gpio0_apb (
94          .clk        (clk),
95          .reset      (reset),
96          // apb slave to master interface
97          .S_PADDR    (M_PADDR),
98          .S_PWRITE   (M_PWRITE),
99          .S_PSELx    (M_PSELx[`APB_PSELX_GPIO0]),
100         .S_PENABLE  (M_PENABLE),
101         .S_PWDATA   (M_PWDATA),
102         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO0*`DATA_WIDTH +: `DATA_WIDTH]),
103         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
104         .gpio       (gpio0)
105     );
106
107     // GPIO1 for Seven segment displays (16 pin)
108     vmicro16_gpio_apb # (
109         .BUS_WIDTH  (`APB_WIDTH),
110         .DATA_WIDTH (`DATA_WIDTH),
111         .PORTS      (`APB_GPIO1_PINS),
112         .NAME       ("GPIO1")
113     ) gpio1_apb (
114         .clk        (clk),
115         .reset      (reset),
116         // apb slave to master interface
117         .S_PADDR    (M_PADDR),
118         .S_PWRITE   (M_PWRITE),
119         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO1]),
120         .S_PENABLE  (M_PENABLE),
121         .S_PWDATA   (M_PWDATA),
122         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO1*`DATA_WIDTH +: `DATA_WIDTH]),
123         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
124         .gpio       (gpio1)
125     );
126
127     // GPIO2 for Seven segment displays (8 pin)
128     vmicro16_gpio_apb # (
129         .BUS_WIDTH  (`APB_WIDTH),
130         .DATA_WIDTH (`DATA_WIDTH),
131         .PORTS      (`APB_GPIO2_PINS),
132         .NAME       ("GPIO2")
133     ) gpio2_apb (
134         .clk        (clk),
135         .reset      (reset),
136         // apb slave to master interface
137         .S_PADDR    (M_PADDR),
138         .S_PWRITE   (M_PWRITE),
139         .S_PSELx    (M_PSELx[`APB_PSELX_GPIO2]),
140         .S_PENABLE  (M_PENABLE),
141         .S_PWDATA   (M_PWDATA),
142         .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO2*`DATA_WIDTH +: `DATA_WIDTH]),
143         .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
144         .gpio       (gpio2)
145     );
146
147     apb_uart_tx # (
148         .DATA_WIDTH (8),
149         .ADDR_EXP   (4) //2^^4 = 16 FIFO words
150     ) uart0_apb (
151         .clk        (clk),
152         .reset      (reset),
153         // apb slave to master interface
154         .S_PADDR    (M_PADDR),
155         .S_PWRITE   (M_PWRITE),
156         .S_PSELx    (M_PSELx[`APB_PSELX_UART0]),
157         .S_PENABLE  (M_PENABLE),
158         .S_PWDATA   (M_PWDATA),
159         .S_PRDATA   (M_PRDATA[`APB_PSELX_UART0*`DATA_WIDTH +: `DATA_WIDTH]),
160         .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
161         // uart wires
162         .tx_wire    (uart_tx),
163         .rx_wire    (uart_rx)
164     );
165
166     timer_apb timr0 (
167         .clk        (clk),
168         .reset      (reset),
169         // apb slave to master interface
170         .S_PADDR    (M_PADDR),
171         .S_PWRITE   (M_PWRITE),
172         .S_PSELx    (M_PSELx[`APB_PSELX_TIMR0]),
173         .S_PENABLE  (M_PENABLE),
```

```
174          .S_PWDATA   (M_PWDATA),
175          .S_PRDATA   (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
176          .S_PREADY   (M_PREADY[`APB_PSELX_TIMR0])
177          //
178          `ifdef DEF_ENABLE_INT
179          ,.out       (ints     [`DEF_INT_TIMR0]),
180           .int_data  (ints_data[`DEF_INT_TIMR0*`DATA_WIDTH +: `DATA_WIDTH])
181          `endif
182      );
183
184      // Shared register set for system-on-chip info
185      // R0 = number of cores
186      vmicro16_regs_apb # (
187          .BUS_WIDTH          (`APB_WIDTH),
188          .DATA_WIDTH         (`DATA_WIDTH),
189          .CELL_DEPTH         (8),
190          .PARAM_DEFAULTS_R0  (`CORES),
191          .PARAM_DEFAULTS_R1  (`SLAVES)
192      ) regs0_apb (
193          .clk        (clk),
194          .reset      (reset),
195          // apb slave to master interface
196          .S_PADDR    (M_PADDR),
197          .S_PWRITE   (M_PWRITE),
198          .S_PSELx    (M_PSELx[`APB_PSELX_REGS0]),
199          .S_PENABLE  (M_PENABLE),
200          .S_PWDATA   (M_PWDATA),
201          .S_PRDATA   (M_PRDATA[`APB_PSELX_REGS0*`DATA_WIDTH +: `DATA_WIDTH]),
202          .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
203      );
204
205      vmicro16_bram_ex_apb # (
206          .BUS_WIDTH    (`APB_WIDTH),
207          .MEM_WIDTH    (`DATA_WIDTH),
208          .MEM_DEPTH    (`APB_BRAM0_CELLS),
209          .CORE_ID_BITS (`clog2(`CORES))
210      ) bram_apb (
211          .clk        (clk),
212          .reset      (reset),
213          // apb slave to master interface
214          .S_PADDR    (M_PADDR),
215          .S_PWRITE   (M_PWRITE),
216          .S_PSELx    (M_PSELx[`APB_PSELX_BRAM0]),
217          .S_PENABLE  (M_PENABLE),
218          .S_PWDATA   (M_PWDATA),
219          .S_PRDATA   (M_PRDATA[`APB_PSELX_BRAM0*`DATA_WIDTH +: `DATA_WIDTH]),
220          .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
221      );
222
223      // There must be atleast 1 core
224      `static_assert(`CORES > 0)
225      `static_assert(`DEF_MEM_INSTR_DEPTH > 0)
226      `static_assert(`DEF_MMU_TIM0_CELLS > 0)
227
228
229      // Single instruction memory
230  `ifndef DEF_CORE_HAS_INSTR_MEM
231      // slave input/outputs from interconnect
232      wire [`APB_WIDTH-1:0]          instr_M_PADDR;
233      wire                           instr_M_PWRITE;
234      wire [1-1:0]                   instr_M_PSELx;  // not shared
235      wire                           instr_M_PENABLE;
236      wire [`DATA_WIDTH-1:0]         instr_M_PWDATA;
237      wire [1*`DATA_WIDTH-1:0]       instr_M_PRDATA; // slave response
238      wire [1-1:0]                   instr_M_PREADY; // slave response
239
240      // Master apb interfaces
241      wire [`CORES*`APB_WIDTH-1:0]   instr_w_PADDR;
242      wire [`CORES-1:0]              instr_w_PWRITE;
243      wire [`CORES-1:0]              instr_w_PSELx;
244      wire [`CORES-1:0]              instr_w_PENABLE;
245      wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PWDATA;
246      wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PRDATA;
247      wire [`CORES-1:0]              instr_w_PREADY;
248
249      vmicro16_bram_apb # (
250          .BUS_WIDTH      (`APB_WIDTH),
251          .MEM_WIDTH      (`DATA_WIDTH),
252          .MEM_DEPTH      (`DEF_MEM_INSTR_DEPTH),
253          .USE_INITS      (1),
254          .NAME           ("INSTR_ROM_G")
255      ) instr_rom_apb (
256          .clk            (clk),
257          .reset          (reset),
258          .S_PADDR        (instr_M_PADDR),
259          .S_PWRITE       (),
260          .S_PSELx        (instr_M_PSELx),
261          .S_PENABLE      (instr_M_PENABLE),
262          .S_PWDATA       (),
263          .S_PRDATA       (instr_M_PRDATA),
264          .S_PREADY       (instr_M_PREADY)
```

```
265     );
266
267     apb_intercon_s # (
268         .MASTER_PORTS   (`CORES),
269         .SLAVE_PORTS    (1),
270         .BUS_WIDTH      (`APB_WIDTH),
271         .DATA_WIDTH     (`DATA_WIDTH),
272         .HAS_PSELX_ADDR (0)
273     ) apb_instr_intercon (
274         .clk        (clk),
275         .reset      (reset),
276         // APB master from cores
277         // master
278         .S_PADDR    (instr_w_PADDR),
279         .S_PWRITE   (instr_w_PWRITE),
280         .S_PSELx    (instr_w_PSELx),
281         .S_PENABLE  (instr_w_PENABLE),
282         .S_PWDATA   (instr_w_PWDATA),
283         .S_PRDATA   (instr_w_PRDATA),
284         .S_PREADY   (instr_w_PREADY),
285         // shared bus slaves
286         // slave outputs
287         .M_PADDR    (instr_M_PADDR),
288         .M_PWRITE   (instr_M_PWRITE),
289         .M_PSELx    (instr_M_PSELx),
290         .M_PENABLE  (instr_M_PENABLE),
291         .M_PWDATA   (instr_M_PWDATA),
292         .M_PRDATA   (instr_M_PRDATA),
293         .M_PREADY   (instr_M_PREADY)
294     );
295 `endif
296
297     genvar i;
298     generate for(i = 0; i < `CORES; i = i + 1) begin : cores
299
300         vmicro16_core # (
301             .CORE_ID            (i),
302             .DATA_WIDTH         (`DATA_WIDTH),
303
304             .MEM_INSTR_DEPTH    (`DEF_MEM_INSTR_DEPTH),
305             .MEM_SCRATCH_DEPTH  (`DEF_MMU_TIMO_CELLS)
306         ) c1 (
307             .clk        (clk),
308             .reset      (reset),
309
310             // debug
311             .halt       (w_halt[i]),
312
313             // interrupts
314             .ints       (ints),
315             .ints_data  (ints_data),
316
317             // Output master port 1
318             .w_PADDR    (w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
319             .w_PWRITE   (w_PWRITE  [i]                           ),
320             .w_PSELx    (w_PSELx   [i]                           ),
321             .w_PENABLE  (w_PENABLE [i]                           ),
322             .w_PWDATA   (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
323             .w_PRDATA   (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
324             .w_PREADY   (w_PREADY  [i]                           )
325
326 `ifndef DEF_CORE_HAS_INSTR_MEM
327             // APB instruction rom
328             , // Output master port 2
329             .w2_PADDR   (instr_w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
330             //.w2_PWRITE  (instr_w_PWRITE  [i]                           ),
331             .w2_PSELx   (instr_w_PSELx   [i]                           ),
332             .w2_PENABLE (instr_w_PENABLE [i]                           ),
333             //.w2_PWDATA  (instr_w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
334             .w2_PRDATA  (instr_w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
335             .w2_PREADY  (instr_w_PREADY  [i]                           )
336 `endif
337         );
338     end
339     endgenerate
340
341
342     ////////////////////////////////////////////////////
343     // Formal Verification
344     ////////////////////////////////////////////////////
345     `ifdef FORMAL
346     wire all_halted = &w_halt;
347     ////////////////////////////////////////////////////
348     // Count number of clocks each core is spending on
349     //   bus transactions
350     ////////////////////////////////////////////////////
351     reg [15:0] bus_core_times [0:`CORES-1];
352     reg [15:0] core_work_times [0:`CORES-1];
353     integer i2;
354     initial
355         for(i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
```

```
356             bus_core_times[i2] = 0;
357             core_work_times[i2] = 0;
358         end
359
360     // total bus time
361     generate
362         genvar g2;
363         for (g2 = 0; g2 < `CORES; g2 = g2 + 1)
364         always @(posedge clk) begin
365             if (w_PSELx[g2])
366                 bus_core_times[g2] <= bus_core_times[g2] + 1;
367
368             // Core working time
369             if (!w_PSELx[g2] && !instr_w_PSELx[g2])
370                 if (!w_halt[g2])
371                     core_work_times[g2] <= core_work_times[g2] + 1;
372         end
373     endgenerate
374
375     reg [15:0] bus_time_average = 0;
376     reg [15:0] bus_reqs_average = 0;
377     reg [15:0] fetch_time_average = 0;
378     reg [15:0] work_time_average = 0;
379     //
380     always @(all_halted) begin
381         for (i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
382             bus_time_average   = bus_time_average   + bus_core_times[i2];
383             bus_reqs_average   = bus_reqs_average   + bus_core_reqs_count[i2];
384             fetch_time_average = fetch_time_average + instr_fetch_times[i2];
385             work_time_average  = work_time_average  + core_work_times[i2];
386         end
387
388         bus_time_average   = bus_time_average   / `CORES;
389         bus_reqs_average   = bus_reqs_average   / `CORES;
390         fetch_time_average = fetch_time_average / `CORES;
391         work_time_average  = work_time_average  / `CORES;
392     end
393
394     ////////////////////////////////////////////////////
395     // Count number of bus requests per core
396     ////////////////////////////////////////////////////
397     // 1 clock delay of w_PSELx
398     reg [`CORES-1:0] bus_core_reqs_last;
399     // rising edges of each
400     wire [`CORES-1:0] bus_core_reqs_real;
401     // storage for counters for each core
402     reg [15:0] bus_core_reqs_count [0:`CORES-1];
403     initial
404         for(i2 = 0; i2 < `CORES; i2 = i2 + 1)
405             bus_core_reqs_count[i2] = 0;
406
407     // 1 clk delay to detect rising edge
408     always @(posedge clk)
409         bus_core_reqs_last <= w_PSELx;
410
411     generate
412         genvar g3;
413         for (g3 = 0; g3 < `CORES; g3 = g3 + 1) begin
414         // Detect new reqs for each core
415         assign bus_core_reqs_real[g3] = w_PSELx[g3] >
416                                         bus_core_reqs_last[g3];
417
418         always @(posedge clk)
419             if (bus_core_reqs_real[g3])
420                 bus_core_reqs_count[g3] <= bus_core_reqs_count[g3] + 1;
421
422     end
423     endgenerate
424
425
426     ////////////////////////////////////////////////////
427     // Time waiting for instruction fetches
428     //   from global  memory
429     ////////////////////////////////////////////////////
430     reg [15:0] instr_fetch_times [0:`CORES-1];
431     integer i3;
432     initial
433         for(i3 = 0; i3 < `CORES; i3 = i3 + 1)
434             instr_fetch_times[i3] = 0;
435
436     // total bus time
437     // Instruction fetches occur on the w2 master port
438     generate
439         genvar g4;
440         for (g4 = 0; g4 < `CORES; g4 = g4 + 1)
441             always @(posedge clk)
442                 if (instr_w_PSELx[g4])
443                     instr_fetch_times[g4] <= instr_fetch_times[g4] + 1;
444     endgenerate
445
446
```

```
447     `endif // end FORMAL
448
449 endmodule
```

B.4 vmicro16_periph.v

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

```
1   // Vmicro16 peripheral modules
2
3   `include "vmicro16_soc_config.v"
4
5   module timer_apb # (
6       parameter CLK_HZ = 50_000_000
7   ) (
8       input clk,
9       input reset,
10
11      input clk_en,
12
13      // 0 16-bit value   R/W
14      // 1 16-bit control R    b0 = start, b1 = reset
15      // 2 16-bit prescaler
16      input       [1:0]               S_PADDR,
17
18      input                           S_PWRITE,
19      input                           S_PSELx,
20      input                           S_PENABLE,
21      input       [`DATA_WIDTH-1:0]   S_PWDATA,
22
23      output reg  [`DATA_WIDTH-1:0]   S_PRDATA,
24      output                          S_PREADY,
25
26      output out,
27      output [`DATA_WIDTH-1:0] int_data
28  );
29      //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
30      assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
31      wire   en       = (S_PSELx & S_PENABLE);
32      wire   we       = (en & S_PWRITE);
33
34      reg [`DATA_WIDTH-1:0] r_counter = 0;
35      reg [`DATA_WIDTH-1:0] r_load = 0;
36      reg [`DATA_WIDTH-1:0] r_pres = 0;
37      reg [`DATA_WIDTH-1:0] r_ctrl = 0;
38
39      localparam CTRL_START = 0;
40      localparam CTRL_RESET = 1;
41      localparam CTRL_INT   = 2;
42
43      localparam ADDR_LOAD = 2'b00;
44      localparam ADDR_CTRL = 2'b01;
45      localparam ADDR_PRES = 2'b10;
46
47      always @(*) begin
48          S_PRDATA = 0;
49          if (en)
50              case(S_PADDR)
51                  ADDR_LOAD: S_PRDATA = r_counter;
52                  ADDR_CTRL: S_PRDATA = r_ctrl;
53                  //ADDR_CTRL: S_PRDATA = r_pres;
54                  default:   S_PRDATA = 0;
55              endcase
56      end
57
58      // prescaler counts from r_pres to 0, emitting a stb signal
59      //   to enable the r_counter step
60      reg [`DATA_WIDTH-1:0] r_pres_counter = 0;
61      wire counter_en = (r_pres_counter == 0);
62      always @(posedge clk)
63          if (r_pres_counter == 0)
64              r_pres_counter <= r_pres;
65          else
66              r_pres_counter <= r_pres_counter - 1;
67
68      always @(posedge clk)
69          if (we)
70              case(S_PADDR)
71                  // Write to the load register:
72                  //   Set load register
73                  //   Set counter register
74                  ADDR_LOAD: begin
75                      r_load          <= S_PWDATA;
76                      r_counter       <= S_PWDATA;
77                      $display($time, "\t\ttimr0: WRITE LOAD: %h", S_PWDATA);
78                  end
```

```
79                  ADDR_CTRL: begin
80                      r_ctrl   <= S_PWDATA;
81                      $display($time, "\t\ttimr0: WRITE CTRL: %h", S_PWDATA);
82                  end
83                  ADDR_PRES: begin
84                      r_pres   <= S_PWDATA;
85                      $display($time, "\t\ttimr0: WRITE PRES: %h", S_PWDATA);
86                  end
87              endcase
88          else
89              if (r_ctrl[CTRL_START]) begin
90                  if (r_counter == 0)
91                      r_counter <= r_load;
92                  else if(counter_en)
93                      r_counter <= r_counter -1;
94              end else if (r_ctrl[CTRL_RESET])
95                  r_counter <= r_load;
96
97      // generate the output pulse when r_counter == 0
98      //   out = (counter reached zero && counter started)
99      assign out      = (r_counter == 0) && r_ctrl[CTRL_START]; // && r_ctrl[CTRL_INT];
100     assign int_data = {`DATA_WIDTH{1'b1}};
101 endmodule
102
103 // APB wrapped vmicro16_bram
104 module vmicro16_bram_apb # (
105     parameter BUS_WIDTH    = 16,
106     parameter MEM_WIDTH    = 16,
107     parameter MEM_DEPTH    = 64,
108     parameter APB_PADDR    = 0,
109     parameter USE_INITS    = 0,
110     parameter NAME         = "BRAM",
111     parameter CORE_ID      = 0
112 ) (
113     input clk,
114     input reset,
115     // APB Slave to master interface
116     input  [`clog2(MEM_DEPTH)-1:0]  S_PADDR,
117     input                           S_PWRITE,
118     input                           S_PSELx,
119     input                           S_PENABLE,
120     input  [BUS_WIDTH-1:0]          S_PWDATA,
121
122     output [BUS_WIDTH-1:0]          S_PRDATA,
123     output                          S_PREADY
124 );
125     wire [MEM_WIDTH-1:0] mem_out;
126
127     assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out : 16'h0000;
128     assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1    : 1'b0;
129     assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
130
131     always @(*)
132         if (S_PSELx && S_PENABLE)
133             $display($time, "\t\t%s => %h", NAME, mem_out);
134
135     always @(posedge clk)
136         if (we)
137             $display($time, "\t\t%s[%h] <= %h", NAME,
138                 S_PADDR, S_PWDATA);
139
140     vmicro16_bram # (
141         .MEM_WIDTH  (MEM_WIDTH),
142         .MEM_DEPTH  (MEM_DEPTH),
143         .NAME       (NAME),
144         .USE_INITS  (1),
145         .CORE_ID    (-1)
146     ) bram_apb (
147         .clk        (clk),
148         .reset      (reset),
149
150         .mem_addr   (S_PADDR),
151         .mem_in     (S_PWDATA),
152         .mem_we     (we),
153         .mem_out    (mem_out)
154     );
155 endmodule
156
157 // Shared memory with hardware monitor (LWEX/SWEX)
158 module vmicro16_bram_ex_apb # (
159     parameter BUS_WIDTH    = 16,
160     parameter MEM_WIDTH    = 16,
161     parameter MEM_DEPTH    = 64,
162     parameter CORE_ID_BITS = 3,
163     parameter SWEX_SUCCESS = 16'h0000,
164     parameter SWEX_FAIL    = 16'h0001
165 ) (
166     input clk,
167     input reset,
168
169     // |19    |18    |16            |15          0|
```

```
170     // | LWEX | SWEX | 3 bit CORE_ID |     S_PADDR |
171     input  [`APB_WIDTH-1:0]         S_PADDR,
172
173     input                           S_PWRITE,
174     input                           S_PSELx,
175     input                           S_PENABLE,
176     input  [MEM_WIDTH-1:0]          S_PWDATA,
177
178     output reg [MEM_WIDTH-1:0]      S_PRDATA,
179     output                          S_PREADY
180 );
181     // exclusive flag checks
182     wire [MEM_WIDTH-1:0] mem_out;
183     reg                  swex_success = 0;
184
185     localparam ADDR_BITS = `clog2(MEM_DEPTH);
186
187     // hack to create a 1 clock delay to S_PREADY
188     // for bram to be ready
189     reg cdelay = 1;
190     always @(posedge clk)
191         if (S_PSELx)
192             cdelay <= 0;
193         else
194             cdelay <= 1;
195
196     //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
197     assign S_PREADY = (S_PSELx & S_PENABLE & (!cdelay)) ? 1'b1       : 1'b0;
198     assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
199     wire   en       = (S_PSELx & S_PENABLE);
200
201     // Similar to:
202     //   http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204f/Cihbghef.html
203
204     // mem_wd is the CORE_ID sent in bits [18:16]
205     localparam TOP_BIT_INDEX         = `APB_WIDTH -1;
206     localparam PADDR_CORE_ID_MSB     = TOP_BIT_INDEX - 2;
207     localparam PADDR_CORE_ID_LSB     = PADDR_CORE_ID_MSB - (CORE_ID_BITS-1);
208
209     // [LWEX, CORE_ID, mem_addr] from S_PADDR
210     wire                    lwex        = S_PADDR[TOP_BIT_INDEX];
211     wire                    swex        = S_PADDR[TOP_BIT_INDEX-1];
212     wire [CORE_ID_BITS-1:0] core_id     = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
213     // CORE_ID to write to ex_flags register
214     wire [ADDR_BITS-1:0]    mem_addr    = S_PADDR[ADDR_BITS-1:0];
215
216     wire [CORE_ID_BITS:0]   ex_flags_read;
217     wire                    is_locked      = |ex_flags_read;
218     wire                    is_locked_self = is_locked && (core_id == (ex_flags_read-1));
219
220     // Check exclusive access flags
221     always @(*) begin
222         swex_success = 0;
223         if (en)
224             // bug!
225             if (!swex && !lwex)
226                 swex_success = 1;
227             else if (swex)
228                 if (is_locked && !is_locked_self)
229                     // someone else has locked it
230                     swex_success = 0;
231                 else if (is_locked && is_locked_self)
232                     swex_success = 1;
233     end
234
235     always @(*)
236         if (swex)
237             if (swex_success)
238                 S_PRDATA = SWEX_SUCCESS;
239             else
240                 S_PRDATA = SWEX_FAIL;
241         else
242             S_PRDATA = mem_out;
243
244     wire reg_we = en && ((lwex && !is_locked)
245                       || (swex && swex_success));
246
247     reg  [CORE_ID_BITS:0] reg_wd;
248     always @(*) begin
249         reg_wd = {{CORE_ID_BITS}{1'b0}};
250
251         if (en)
252             // if wanting to lock the addr
253             if (lwex)
254                 // and not already locked
255                 if (!is_locked) begin
256                     reg_wd = (core_id + 1);
257                 end
258             else if (swex)
259                 if (is_locked && is_locked_self)
260                     reg_wd = {{CORE_ID_BITS}{1'b0}};
```

```
261       end
262
263       // Exclusive flag for each memory cell
264       vmicro16_bram # (
265           .MEM_WIDTH  (CORE_ID_BITS + 1),
266           .MEM_DEPTH  (MEM_DEPTH),
267           .USE_INITS  (0),
268           .NAME       ("rexram")
269       ) ram_exflags (
270           .clk        (clk),
271           .reset      (reset),
272
273           .mem_addr   (mem_addr),
274           .mem_in     (reg_wd),
275           .mem_we     (reg_we),
276           .mem_out    (ex_flags_read)
277       );
278
279       always @(*)
280           if (S_PSELx && S_PENABLE)
281               $display($time, "\t\tBRAMex[%h] READ %h\tCORE: %h", mem_addr, mem_out, S_PADDR[16 +: CORE_ID_BITS]);
282
283       always @(posedge clk)
284           if (we)
285               $display($time, "\t\tBRAMex[%h] WRITE %h\tCORE: %h", mem_addr, S_PWDATA, S_PADDR[16 +: CORE_ID_BITS]);
286
287       vmicro16_bram # (
288           .MEM_WIDTH  (MEM_WIDTH),
289           .MEM_DEPTH  (MEM_DEPTH),
290           .USE_INITS  (0),
291           .NAME       ("BRAMexinst")
292       ) bram_apb (
293           .clk        (clk),
294           .reset      (reset),
295
296           .mem_addr   (mem_addr),
297           .mem_in     (S_PWDATA),
298           .mem_we     (we && swex_success),
299           .mem_out    (mem_out)
300       );
301   endmodule
302
303   // Simple APB memory-mapped register set
304   module vmicro16_regs_apb # (
305       parameter BUS_WIDTH         = 16,
306       parameter DATA_WIDTH        = 16,
307       parameter CELL_DEPTH        = 8,
308       parameter PARAM_DEFAULTS_R0 = 0,
309       parameter PARAM_DEFAULTS_R1 = 0
310   ) (
311       input clk,
312       input reset,
313       // APB Slave to master interface
314       input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
315       input                           S_PWRITE,
316       input                           S_PSELx,
317       input                           S_PENABLE,
318       input  [DATA_WIDTH-1:0]         S_PWDATA,
319
320       output [DATA_WIDTH-1:0]         S_PRDATA,
321       output                          S_PREADY
322   );
323       wire [DATA_WIDTH-1:0] rd1;
324
325       assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1  : 16'h0000;
326       assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
327       assign reg_we   = (S_PSELx & S_PENABLE & S_PWRITE);
328
329       always @(*)
330           if (reg_we)
331               $display($time, "\t\tREGS_APB[%h] <= %h",
332                   S_PADDR, S_PWDATA);
333
334       always @(*)
335           `rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))
336
337       vmicro16_regs # (
338           .CELL_DEPTH         (CELL_DEPTH),
339           .CELL_WIDTH         (DATA_WIDTH),
340           .PARAM_DEFAULTS_R0  (PARAM_DEFAULTS_R0),
341           .PARAM_DEFAULTS_R1  (PARAM_DEFAULTS_R1)
342       ) regs_apb (
343           .clk    (clk),
344           .reset  (reset),
345           // port 1
346           .rs1    (S_PADDR),
347           .rd1    (rd1),
348           .we     (reg_we),
349           .ws1    (S_PADDR),
350           .wd     (S_PWDATA)
351           // port 2 unconnected
```

```
352         //.rs2    (),
353         //.rd2    ()
354     );
355 endmodule
356 
357 // Simple GPIO write only peripheral
358 module vmicro16_gpio_apb # (
359     parameter BUS_WIDTH  = 16,
360     parameter DATA_WIDTH = 16,
361     parameter PORTS      = 8,
362     parameter NAME       = "GPIO"
363 ) (
364     input clk,
365     input reset,
366     // APB Slave to master interface
367     input  [0:0]                    S_PADDR, // not used (optimised out)
368     input                           S_PWRITE,
369     input                           S_PSELx,
370     input                           S_PENABLE,
371     input  [DATA_WIDTH-1:0]         S_PWDATA,
372 
373     output [DATA_WIDTH-1:0]         S_PRDATA,
374     output                          S_PREADY,
375     output reg [PORTS-1:0]          gpio
376 );
377     assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
378     assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
379     assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);
380 
381     always @(posedge clk)
382         if (reset)
383             gpio <= 0;
384         else if (ports_we) begin
385             $display($time, "\t\%s <= %h", NAME, S_PWDATA[PORTS-1:0]);
386             gpio <= S_PWDATA[PORTS-1:0];
387         end
388 endmodule
```

B.5 vmicro16.v

Vmicro16 CPU core module.

```
1   // This file contains multiple modules.
2   //   Verilator likes 1 file for each module
3   /* verilator lint_off DECLFILENAME */
4   /* verilator lint_off UNUSED */
5   /* verilator lint_off BLKSEQ */
6   /* verilator lint_off WIDTH */
7 
8   // Include Vmicro16 ISA containing definitions for the bits
9   `include "vmicro16_isa.v"
10 
11  `include "clog2.v"
12  `include "formal.v"
13 
14 
15 
16  // This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
17  //   https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
18  //   https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
19  //   https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf
20  module vmicro16_bram # (
21      parameter MEM_WIDTH     = 16,
22      parameter MEM_DEPTH     = 64,
23      parameter CORE_ID       = 0,
24      parameter USE_INITS     = 0,
25      parameter PARAM_DEFAULTS_R0 = 0,
26      parameter PARAM_DEFAULTS_R1 = 0,
27      parameter PARAM_DEFAULTS_R2 = 0,
28      parameter PARAM_DEFAULTS_R3 = 0,
29      parameter NAME          = "BRAM"
30  ) (
31      input clk,
32      input reset,
33 
34      input      [`clog2(MEM_DEPTH)-1:0] mem_addr,
35      input      [MEM_WIDTH-1:0]         mem_in,
36      input                              mem_we,
37      output reg [MEM_WIDTH-1:0]         mem_out
38  );
39      // memory vector
40      (* ram_style = "block" *)
41      reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];
42 
43      // not synthesizable
44      integer i;
```

```
45      initial begin
46          for (i = 0; i < MEM_DEPTH; i = i + 1) mem[i] = 0;
47          mem[0] = PARAM_DEFAULTS_R0;
48          mem[1] = PARAM_DEFAULTS_R1;
49          mem[2] = PARAM_DEFAULTS_R2;
50          mem[3] = PARAM_DEFAULTS_R3;
51
52          if (USE_INITS) begin
53              //`define TEST_SW
54              `ifdef TEST_SW
55              $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\verilog_memh.txt", mem);
56              `endif
57
58              `define TEST_ASM
59              `ifdef TEST_ASM
60              $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\asm.s.hex", mem);
61              `endif
62
63              //`define TEST_COND
64              `ifdef TEST_COND
65              mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
66              mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
67              `endif
68
69              //`define TEST_CMP
70              `ifdef TEST_CMP
71              mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0A};
72              mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h0B};
73              mem[2] = {`VMICRO16_OP_CMP,     3'h1, 3'h0, 5'h1};
74              `endif
75
76              //`define TEST_LWEX
77              `ifdef TEST_LWEX
78              mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC5};
79              mem[1] = {`VMICRO16_OP_SW,      3'h0, 3'h0, 5'h1};
80              mem[2] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h1};
81              mem[3] = {`VMICRO16_OP_LWEX,    3'h2, 3'h0, 5'h1};
82              mem[4] = {`VMICRO16_OP_SWEX,    3'h3, 3'h0, 5'h1};
83              `endif
84
85              //`define TEST_MULTICORE
86              `ifdef TEST_MULTICORE
87              mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
88              mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
89              mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
90              mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h80};
91              mem[4] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
92              mem[5] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
93              mem[6] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
94              mem[7] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
95              mem[8] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
96              mem[9] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h0};
97              `endif
98
99              //`define TEST_BR
100             `ifdef TEST_BR
101             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0};
102             mem[1] = {`VMICRO16_OP_MOVI,    3'h3, 8'h3};
103             mem[2] = {`VMICRO16_OP_MOVI,    3'h1, 8'h2};
104             mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
105             mem[4] = {`VMICRO16_OP_BR,      3'h3, `VMICRO16_OP_BR_U};
106             mem[5] = {`VMICRO16_OP_MOVI,    3'h0, 8'hFF};
107             `endif
108
109             //`define ALL_TEST
110             `ifdef ALL_TEST
111             // Standard all test
112             // REGS0
113             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h81};
114             mem[1] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
115             mem[2] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
116             // GPIO0
117             mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
118             mem[4] = {`VMICRO16_OP_MOVI,    3'h1, 8'hD};
119             mem[5] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
120             mem[6] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
121             // TIM0
122             mem[7] = {`VMICRO16_OP_MOVI,    3'h0, 8'h07};
123             mem[8] = {`VMICRO16_OP_LW,      3'h3, 3'h0, 5'h03};
124             // UART0
125             mem[9]  = {`VMICRO16_OP_MOVI,    3'h0, 8'hA0};      // UART0
126             mem[10] = {`VMICRO16_OP_MOVI,    3'h1, 8'h41};      // ascii A
127             mem[11] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
128             mem[12] = {`VMICRO16_OP_MOVI,    3'h1, 8'h42}; // ascii B
129             mem[13] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
130             mem[14] = {`VMICRO16_OP_MOVI,    3'h1, 8'h43}; // ascii C
131             mem[15] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
132             mem[16] = {`VMICRO16_OP_MOVI,    3'h1, 8'h44}; // ascii D
133             mem[17] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
134             mem[18] = {`VMICRO16_OP_MOVI,    3'h1, 8'h45}; // ascii D
135             mem[19] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
```

```
136             mem[20] = {`VMICRO16_OP_MOVI,    3'h1, 8'h46}; // ascii E
137             mem[21] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
138             // BRAM0
139             mem[22] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
140             mem[23] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
141             mem[24] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
142             mem[25] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
143             // GPIO1 (SSD 24-bit port)
144             mem[26] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
145             mem[27] = {`VMICRO16_OP_MOVI,    3'h1, 8'h12};
146             mem[28] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
147             mem[29] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
148             // GPIO2
149             mem[30] = {`VMICRO16_OP_MOVI,    3'h0, 8'h92};
150             mem[31] = {`VMICRO16_OP_MOVI,    3'h1, 8'h56};
151             mem[32] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
152             `endif
153
154             //`define TEST_BRAM
155             `ifdef TEST_BRAM
156             // 2 core BRAM0 test
157             mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
158             mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
159             mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
160             mem[3] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
161             `endif
162         end
163     end
164
165     always @(posedge clk) begin
166         // synchronous WRITE_FIRST (page 13)
167         if (mem_we) begin
168             mem[mem_addr] <= mem_in;
169             $display($time, "\t\t%s[%h] <= %h",
170                     NAME, mem_addr, mem_in);
171         end else
172             mem_out <= mem[mem_addr];
173     end
174
175     // TODO: Reset impl = every clock while reset is asserted, clear each cell
176     //       one at a time, mem[i++] <= 0
177 endmodule
178
179
180 module vmicro16_core_mmu # (
181     parameter MEM_WIDTH     = 16,
182     parameter MEM_DEPTH     = 64,
183
184     parameter CORE_ID       = 3'h0,
185     parameter CORE_ID_BITS  = `clog2(`CORES)
186 ) (
187     input clk,
188     input reset,
189
190     input  req,
191     output busy,
192
193     // From core
194     input      [MEM_WIDTH-1:0]  mmu_addr,
195     input      [MEM_WIDTH-1:0]  mmu_in,
196     input                       mmu_we,
197     input                       mmu_lwex,
198     input                       mmu_swex,
199     output reg [MEM_WIDTH-1:0]  mmu_out,
200
201     // interrupts
202     output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
203     output reg [`DEF_NUM_INT-1:0]             ints_mask,
204
205     // TO APB interconnect
206     output reg [`APB_WIDTH-1:0]  M_PADDR,
207     output reg                   M_PWRITE,
208     output reg                   M_PSELx,
209     output reg                   M_PENABLE,
210     output reg [MEM_WIDTH-1:0]   M_PWDATA,
211     // from interconnect
212     input      [MEM_WIDTH-1:0]   M_PRDATA,
213     input                        M_PREADY
214 );
215     localparam MMU_STATE_T1  = 0;
216     localparam MMU_STATE_T2  = 1;
217     localparam MMU_STATE_T3  = 2;
218     reg [1:0]  mmu_state      = MMU_STATE_T1;
219
220     reg  [MEM_WIDTH-1:0] per_out = 0;
221     wire [MEM_WIDTH-1:0] tim0_out;
222
223     assign busy = req || (mmu_state == MMU_STATE_T2);
224
225     // tightly integrated memory usage
226     //wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
```

```
227         //            && (mmu_addr <= `DEF_MMU_TIM0_E);
228         //wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
229         //            && (mmu_addr <= `DEF_MMU_SREG_E);
230         //wire intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
231         //            && (mmu_addr <= `DEF_MMU_INTSV_E);
232         //wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
233         //            && (mmu_addr <= `DEF_MMU_INTSM_E);
234
235         wire tim0_en = ~mmu_addr[12] && ~mmu_addr[9] && ~mmu_addr[7];
236         wire sreg_en = mmu_addr[7] && ~mmu_addr[4] && ~mmu_addr[5];
237         wire intv_en = mmu_addr[8] && ~mmu_addr[3];
238         wire intm_en = mmu_addr[8] && mmu_addr[3];
239
240         wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
241         wire tim0_we   = (tim0_en && mmu_we);
242         wire intv_we   = (intv_en && mmu_we);
243         wire intm_we   = (intm_en && mmu_we);
244
245         // Special register selects
246         localparam SPECIAL_REGS = 8;
247         wire [MEM_WIDTH-1:0] sr_val;
248
249         // Interrupt vector and mask
250         initial ints_vector = 0;
251         initial ints_mask   = 0;
252         wire [2:0] intv_addr = mmu_addr[`clog2(`DEF_NUM_INT)-1:0];
253         always @(posedge clk)
254             if (intv_we)
255                 ints_vector[intv_addr*`DATA_WIDTH +: `DATA_WIDTH] <= mmu_in;
256
257         always @(posedge clk)
258             if (intm_we)
259                 ints_mask <= mmu_in;
260
261
262         always @(ints_vector)
263             $display($time,
264                      "\tC%d\t\tints_vector W: | %h %h %h %h | %h %h %h %h |",
265                       CORE_ID,
266                 ints_vector[0*`DATA_WIDTH +: `DATA_WIDTH],
267                 ints_vector[1*`DATA_WIDTH +: `DATA_WIDTH],
268                 ints_vector[2*`DATA_WIDTH +: `DATA_WIDTH],
269                 ints_vector[3*`DATA_WIDTH +: `DATA_WIDTH],
270                 ints_vector[4*`DATA_WIDTH +: `DATA_WIDTH],
271                 ints_vector[5*`DATA_WIDTH +: `DATA_WIDTH],
272                 ints_vector[6*`DATA_WIDTH +: `DATA_WIDTH],
273                 ints_vector[7*`DATA_WIDTH +: `DATA_WIDTH]
274                 );
275
276         always @(intm_we)
277             $display($time, "\tC%d\t\tintm_we W: %b", CORE_ID, ints_mask);
278
279         // Output port
280         always @(*)
281             if      (tim0_en) mmu_out = tim0_out;
282             else if (sreg_en) mmu_out = sr_val;
283             else if (intv_en) mmu_out = ints_vector[mmu_addr[2:0]*`DATA_WIDTH
284                                                    +: `DATA_WIDTH];
285             else if (intm_en) mmu_out = ints_mask;
286             else              mmu_out = per_out;
287
288         // APB master to slave interface
289         always @(posedge clk)
290             if (reset) begin
291                 mmu_state <= MMU_STATE_T1;
292                 M_PENABLE <= 0;
293                 M_PADDR   <= 0;
294                 M_PWDATA  <= 0;
295                 M_PSELx   <= 0;
296                 M_PWRITE  <= 0;
297             end
298             else
299                 casex (mmu_state)
300                     MMU_STATE_T1: begin
301                         if (req && apb_en) begin
302                             M_PADDR   <= {mmu_lwex,
303                                           mmu_swex,
304                                           CORE_ID[CORE_ID_BITS-1:0],
305                                           mmu_addr[MEM_WIDTH-1:0]};
306
307                             M_PWDATA  <= mmu_in;
308                             M_PSELx   <= 1;
309                             M_PWRITE  <= mmu_we;
310
311                             mmu_state <= MMU_STATE_T2;
312                         end
313                     end
314
315                     `ifdef FIX_T3
316                         MMU_STATE_T2: begin
317                             M_PENABLE <= 1;
```

```
318
319                         if (M_PREADY == 1'b1) begin
320                             mmu_state <= MMU_STATE_T3;
321                         end
322                     end
323
324                     MMU_STATE_T3: begin
325                         // Slave has output a ready signal (finished)
326                         M_PENABLE <= 0;
327                         M_PADDR   <= 0;
328                         M_PWDATA  <= 0;
329                         M_PSELx   <= 0;
330                         M_PWRITE  <= 0;
331                         // Clock the peripheral output into a reg,
332                         //   to output on the next clock cycle
333                         per_out   <= M_PRDATA;
334
335                         mmu_state <= MMU_STATE_T1;
336                     end
337                 `else
338                     // No FIX_T3
339                     MMU_STATE_T2: begin
340                         if (M_PREADY == 1'b1) begin
341                             M_PENABLE <= 0;
342                             M_PADDR   <= 0;
343                             M_PWDATA  <= 0;
344                             M_PSELx   <= 0;
345                             M_PWRITE  <= 0;
346                             // Clock the peripheral output into a reg,
347                             //   to output on the next clock cycle
348                             per_out   <= M_PRDATA;
349
350                             mmu_state <= MMU_STATE_T1;
351                         end else begin
352                             M_PENABLE <= 1;
353                         end
354                     end
355                 `endif
356             endcase
357
358     (* ram_style = "block" *)
359     vmicro16_bram # (
360         .MEM_WIDTH  (MEM_WIDTH),
361         .MEM_DEPTH  (SPECIAL_REGS),
362         .USE_INITS  (0),
363         .PARAM_DEFAULTS_R0  (CORE_ID),
364         .PARAM_DEFAULTS_R1  (`CORES),
365         .PARAM_DEFAULTS_R2  (`APB_BRAM0_CELLS),
366         .PARAM_DEFAULTS_R3  (`SLAVES),
367         .NAME       ("ram_sr")
368     ) ram_sr (
369         .clk        (clk),
370         .reset      (reset),
371         .mem_addr   (mmu_addr[`clog2(SPECIAL_REGS)-1:0]),
372         .mem_in     (),
373         .mem_we     (),
374         .mem_out    (sr_val)
375     );
376
377     // Each M core has a TIM0 scratch memory
378     (* ram_style = "block" *)
379     vmicro16_bram # (
380         .MEM_WIDTH  (MEM_WIDTH),
381         .MEM_DEPTH  (MEM_DEPTH),
382         .USE_INITS  (0),
383         .NAME       ("TIM0")
384     ) TIM0 (
385         .clk        (clk),
386         .reset      (reset),
387         .mem_addr   (mmu_addr[7:0]),
388         .mem_in     (mmu_in),
389         .mem_we     (tim0_we),
390         .mem_out    (tim0_out)
391     );
392 endmodule
393
394
395
396 module vmicro16_regs # (
397     parameter CELL_WIDTH        = 16,
398     parameter CELL_DEPTH        = 8,
399     parameter CELL_SEL_BITS     = `clog2(CELL_DEPTH),
400     parameter CELL_DEFAULTS     = 0,
401     parameter DEBUG_NAME        = "",
402     parameter CORE_ID           = 0,
403     parameter PARAM_DEFAULTS_R0 = 16'h0000,
404     parameter PARAM_DEFAULTS_R1 = 16'h0000
405 ) (
406     input clk,
407     input reset,
408     // Dual port register reads
```

```
409      input       [CELL_SEL_BITS-1:0]  rs1, // port 1
410      output      [CELL_WIDTH-1   :0]  rd1,
411      //input       [CELL_SEL_BITS-1:0]  rs2, // port 2
412      //output      [CELL_WIDTH-1   :0]  rd2,
413      // EX/WB final stage write back
414      input                            we,
415      input [CELL_SEL_BITS-1:0]        ws1,
416      input [CELL_WIDTH-1:0]           wd
417  );
418      (* ram_style = "distributed" *)
419      reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;
420
421      // Initialise registers with default values
422      //   Really only used for special registers used by the soc
423      // TODO: How to do this on reset?
424      integer i;
425      initial
426          if (CELL_DEFAULTS)
427              $readmemh(CELL_DEFAULTS, regs);
428          else begin
429              for(i = 0; i < CELL_DEPTH; i = i + 1)
430                  regs[i] = 0;
431              regs[0] = PARAM_DEFAULTS_R0;
432              regs[1] = PARAM_DEFAULTS_R1;
433              end
434
435      `ifdef ICARUS
436          always @(regs)
437              $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
438                  CORE_ID,
439                  regs[0], regs[1], regs[2], regs[3],
440                  regs[4], regs[5], regs[6], regs[7]);
441      `endif
442
443      always @(posedge clk)
444          if (reset) begin
445              for(i = 0; i < CELL_DEPTH; i = i + 1)
446                  regs[i] <= 0;
447              regs[0] <= PARAM_DEFAULTS_R0;
448              regs[1] <= PARAM_DEFAULTS_R1;
449          end
450          else if (we) begin
451              $display($time, "\tC%02h: REGS #%s: Writing %h to reg[%d]",
452                  CORE_ID, DEBUG_NAME, wd, ws1);
453
454              // Perform the write
455              regs[ws1] <= wd;
456          end
457
458      // sync writes, async reads
459      assign rd1 = regs[rs1];
460      //assign rd2 = regs[rs2];
461  endmodule
462
463  module vmicro16_dec # (
464      parameter INSTR_WIDTH    = 16,
465      parameter INSTR_OP_WIDTH = 5,
466      parameter INSTR_RS_WIDTH = 3,
467      parameter ALU_OP_WIDTH   = 5
468  ) (
469      //input clk,   // not used yet (all combinational)
470      //input reset, // not used yet (all combinational)
471
472      input  [INSTR_WIDTH-1:0]    instr,
473
474      output [INSTR_OP_WIDTH-1:0] opcode,
475      output [INSTR_RS_WIDTH-1:0] rd,
476      output [INSTR_RS_WIDTH-1:0] ra,
477      output [3:0]                imm4,
478      output [7:0]                imm8,
479      output [11:0]               imm12,
480      output [4:0]                simm5,
481
482      // This can be freely increased without affecting the isa
483      output reg [ALU_OP_WIDTH-1:0] alu_op,
484
485      output reg has_imm4,
486      output reg has_imm8,
487      output reg has_imm12,
488      output reg has_we,
489      output reg has_br,
490      output reg has_mem,
491      output reg has_mem_we,
492      output reg has_cmp,
493
494      output halt,
495      output intr,
496
497      output reg has_lwex,
498      output reg has_swex
499
```

```
500     // TODO: Use to identify bad instruction and
501     //       raise exceptions
502     //,output     is_bad
503 );
504     assign opcode = instr[15:11];
505     assign rd     = instr[10:8];
506     assign ra     = instr[7:5];
507     assign imm4   = instr[3:0];
508     assign imm8   = instr[7:0];
509     assign imm12  = instr[11:0];
510     assign simm5  = instr[4:0];
511
512     // exme_op
513     always @(*) case (opcode)
514         `VMICRO16_OP_SPCL: casez(instr[11:0])
515             `VMICRO16_OP_SPCL_NOP,
516             `VMICRO16_OP_SPCL_HALT,
517             `VMICRO16_OP_SPCL_INTR:   alu_op = `VMICRO16_ALU_NOP;
518             default:                  alu_op = `VMICRO16_ALU_NOP; endcase
519
520         `VMICRO16_OP_LW:              alu_op = `VMICRO16_ALU_LW;
521         `VMICRO16_OP_SW:              alu_op = `VMICRO16_ALU_SW;
522         `VMICRO16_OP_LWEX:            alu_op = `VMICRO16_ALU_LW;
523         `VMICRO16_OP_SWEX:            alu_op = `VMICRO16_ALU_SW;
524
525         `VMICRO16_OP_MOV:             alu_op = `VMICRO16_ALU_MOV;
526         `VMICRO16_OP_MOVI:            alu_op = `VMICRO16_ALU_MOVI;
527
528         `VMICRO16_OP_BR:              alu_op = `VMICRO16_ALU_BR;
529         `VMICRO16_OP_MULT:            alu_op = `VMICRO16_ALU_MULT;
530
531         `VMICRO16_OP_CMP:             alu_op = `VMICRO16_ALU_CMP;
532         `VMICRO16_OP_SETC:            alu_op = `VMICRO16_ALU_SETC;
533
534         `VMICRO16_OP_BIT:     casez (simm5)
535             `VMICRO16_OP_BIT_OR:      alu_op = `VMICRO16_ALU_BIT_OR;
536             `VMICRO16_OP_BIT_XOR:     alu_op = `VMICRO16_ALU_BIT_XOR;
537             `VMICRO16_OP_BIT_AND:     alu_op = `VMICRO16_ALU_BIT_AND;
538             `VMICRO16_OP_BIT_NOT:     alu_op = `VMICRO16_ALU_BIT_NOT;
539             `VMICRO16_OP_BIT_LSHFT:   alu_op = `VMICRO16_ALU_BIT_LSHFT;
540             `VMICRO16_OP_BIT_RSHFT:   alu_op = `VMICRO16_ALU_BIT_RSHFT;
541             default:                  alu_op = `VMICRO16_ALU_BAD; endcase
542
543         `VMICRO16_OP_ARITH_U:     casez (simm5)
544             `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
545             `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
546             `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
547             default:                  alu_op = `VMICRO16_ALU_BAD; endcase
548
549         `VMICRO16_OP_ARITH_S:     casez (simm5)
550             `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
551             `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
552             `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
553             default:                  alu_op = `VMICRO16_ALU_BAD; endcase
554
555         default: begin
556                                       alu_op = `VMICRO16_ALU_NOP;
557             $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
558         end
559     endcase
560
561     // Special opcodes
562     //assign nop  == ((opcode == `VMICRO16_OP_SPCL) & (~instr[0]));
563     assign halt = ((opcode == `VMICRO16_OP_SPCL) &   instr[0]);
564     assign intr = ((opcode == `VMICRO16_OP_SPCL) &   instr[1]);
565
566     // Register writes
567     always @(*) case (opcode)
568         `VMICRO16_OP_LWEX,
569         `VMICRO16_OP_SWEX,
570         `VMICRO16_OP_LW,
571         `VMICRO16_OP_MOV,
572         `VMICRO16_OP_MOVI,
573         //`VMICRO16_OP_MOVI_L,
574         `VMICRO16_OP_ARITH_U,
575         `VMICRO16_OP_ARITH_S,
576         `VMICRO16_OP_SETC,
577         `VMICRO16_OP_BIT,
578         `VMICRO16_OP_MULT:      has_we = 1'b1;
579         default:                has_we = 1'b0;
580     endcase
581
582     // Contains 4-bit immediate
583     always @(*)
584         if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
585             ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
586             has_imm4 = 1'b1;
587         else
588             has_imm4 = 1'b0;
589
590     // Contains 8-bit immediate
```

```
591     always @(*) case (opcode)
592         `VMICRO16_OP_MOVI,
593         `VMICRO16_OP_BR:        has_imm8 = 1'b1;
594         default:                has_imm8 = 1'b0;
595     endcase
596
597     //// Contains 12-bit immediate
598     //always @(*) case (opcode)
599     //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
600     //    default:                has_imm12 = 1'b0;
601     //endcase
602
603     // Will branch the pc
604     always @(*) case (opcode)
605         `VMICRO16_OP_BR:    has_br = 1'b1;
606         default:            has_br = 1'b0;
607     endcase
608
609     // Requires external memory
610     always @(*) case (opcode)
611         `VMICRO16_OP_LW,
612         `VMICRO16_OP_SW,
613         `VMICRO16_OP_LWEX,
614         `VMICRO16_OP_SWEX:  has_mem = 1'b1;
615         default:            has_mem = 1'b0;
616     endcase
617
618     // Requires external memory write
619     always @(*) case (opcode)
620         `VMICRO16_OP_SW,
621         `VMICRO16_OP_SWEX:  has_mem_we = 1'b1;
622         default:            has_mem_we = 1'b0;
623     endcase
624
625     // Affects status registers (cmp instructions)
626     always @(*) case (opcode)
627         `VMICRO16_OP_CMP:   has_cmp = 1'b1;
628         default:            has_cmp = 1'b0;
629     endcase
630
631     // Performs exclusive checks
632     always @(*) case (opcode)
633         `VMICRO16_OP_LWEX:   has_lwex = 1'b1;
634         default:             has_lwex = 1'b0;
635     endcase
636
637     always @(*) case (opcode)
638         `VMICRO16_OP_SWEX:   has_swex = 1'b1;
639         default:             has_swex = 1'b0;
640     endcase
641 endmodule
642
643
644 module vmicro16_alu # (
645     parameter OP_WIDTH   = 5,
646     parameter DATA_WIDTH = 16,
647     parameter CORE_ID    = 0
648 ) (
649     // input clk, // TODO: make clocked
650
651     input      [OP_WIDTH-1:0]   op,
652     input      [DATA_WIDTH-1:0] a, // rs1/dst
653     input      [DATA_WIDTH-1:0] b, // rs2
654     input      [3:0]            flags,
655     output reg [DATA_WIDTH-1:0] c
656 );
657     localparam TOP_BIT = (DATA_WIDTH-1);
658     // 17-bit register
659     reg [DATA_WIDTH:0] cmp_tmp = 0; // = {carry, [15:0]}
660     wire r_setc;
661
662     always @(*) begin
663                         cmp_tmp = 0;
664                         case (op)
665         // branch/nop, output nothing
666         `VMICRO16_ALU_BR,
667         `VMICRO16_ALU_NOP:          c = {DATA_WIDTH{1'b0}};
668         // load/store addresses (use value in rd2)
669         `VMICRO16_ALU_LW,
670         `VMICRO16_ALU_SW:           c = b;
671         // bitwise operations
672         `VMICRO16_ALU_BIT_OR:       c = a | b;
673         `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
674         `VMICRO16_ALU_BIT_AND:      c = a & b;
675         `VMICRO16_ALU_BIT_NOT:      c = ~(b);
676         `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
677         `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;
678
679         `VMICRO16_ALU_MOV:          c = b;
680         `VMICRO16_ALU_MOVI:         c = b;
681         `VMICRO16_ALU_MOVI_L:       c = b;
```

```
682
683         `VMICRO16_ALU_ARITH_UADD:   c = a + b;
684         `VMICRO16_ALU_ARITH_USUB:   c = a - b;
685         // TODO: ALU should have simm5 as input
686         `VMICRO16_ALU_ARITH_UADDI:  c = a + b;
687
688         `ifdef DEF_ALU_HW_MULT
689             `VMICRO16_ALU_MULT:  c = a * b;
690         `endif
691
692         `VMICRO16_ALU_ARITH_SADD:   c = $signed(a) + $signed(b);
693         `VMICRO16_ALU_ARITH_SSUB:   c = $signed(a) - $signed(b);
694         // TODO: ALU should have simm5 as input
695         `VMICRO16_ALU_ARITH_SSUBI:  c = $signed(a) - $signed(b);
696
697         `VMICRO16_ALU_CMP: begin
698             // TODO: Do a-b in 17-bit register
699             //       Set zero, overflow, carry, signed bits in result
700             cmp_tmp = a - b;
701             c = 0;
702
703             // N   Negative condition code flag
704             // Z   Zero condition code flag
705             // C   Carry condition code flag
706             // V   Overflow condition code flag
707             c[`VMICRO16_SFLAG_N] = cmp_tmp[TOP_BIT];
708             c[`VMICRO16_SFLAG_Z] = (cmp_tmp == 0);
709             c[`VMICRO16_SFLAG_C] = 0; //cmp_tmp[TOP_BIT+1]; // not used
710
711             // Overflow flag
712             // https://stackoverflow.com/questions/30957188/
713             // https://github.com/bendl/prco304/blob/master/prco_core/rtl/prco_alu.v#L50
714             case(cmp_tmp[TOP_BIT+1:TOP_BIT])
715                 2'b01:   c[`VMICRO16_SFLAG_V] = 1;
716                 2'b10:   c[`VMICRO16_SFLAG_V] = 1;
717                 default: c[`VMICRO16_SFLAG_V] = 0;
718             endcase
719
720             $display($time, "\tC%02h: ALU CMP: %h %h = %h = %b", CORE_ID, a, b, cmp_tmp, c[3:0]);
721         end
722
723         `VMICRO16_ALU_SETC: c = { {15{1'b0}}, r_setc };
724
725         // TODO: Parameterise
726         default: begin
727             $display($time, "\tALU: unknown op: %h", op);
728             c       = 0;
729             cmp_tmp = 0;
730         end
731                 endcase
732                 end
733
734     branch setc_check (
735         .flags      (flags),
736         .cond       (b[7:0]),
737         .en         (r_setc)
738     );
739 endmodule
740
741 // flags = 4 bit r_cmp_flags register
742 // cond  = 8 bit VMICRO16_OP_BR_? value. See vmicro16_isa.v
743 module branch (
744     input [3:0] flags,
745     input [7:0] cond,
746     output reg  en
747 );
748     always @(*)
749         case (cond)
750             `VMICRO16_OP_BR_U:  en = 1; `VMICRO16_OP_BR_U:  en = 1;
751             `VMICRO16_OP_BR_E:  en = (flags[`VMICRO16_SFLAG_Z] == 1);
752             `VMICRO16_OP_BR_NE: en = (flags[`VMICRO16_SFLAG_Z] == 0);
753             `VMICRO16_OP_BR_G:  en = (flags[`VMICRO16_SFLAG_Z] == 0) &&
754                                      (flags[`VMICRO16_SFLAG_N] == flags[`VMICRO16_SFLAG_V]);
755             `VMICRO16_OP_BR_L:  en = (flags[`VMICRO16_SFLAG_Z] != flags[`VMICRO16_SFLAG_N]);
756             `VMICRO16_OP_BR_GE: en = (flags[`VMICRO16_SFLAG_Z] == flags[`VMICRO16_SFLAG_N]);
757             `VMICRO16_OP_BR_LE: en = (flags[`VMICRO16_SFLAG_Z] == 1) ||
758                                      (flags[`VMICRO16_SFLAG_N] != flags[`VMICRO16_SFLAG_V]);
759             default:            en = 0;
760         endcase
761 endmodule
762
763
764
765 module vmicro16_core # (
766     parameter DATA_WIDTH        = 16,
767     parameter MEM_INSTR_DEPTH   = 64,
768     parameter MEM_SCRATCH_DEPTH = 64,
769     parameter MEM_WIDTH         = 16,
770
771     parameter CORE_ID           = 3'h0
772 ) (
```

```
773       input         clk,
774       input         reset,
775
776       output [7:0] dbug,
777
778       output        halt,
779
780       // interrupt sources
781       input  [`DEF_NUM_INT-1:0]              ints,
782       input  [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_data,
783       output [`DEF_NUM_INT-1:0]              ints_ack,
784
785       // APB master to slave interface (apb_intercon)
786       output  [`APB_WIDTH-1:0]     w_PADDR,
787       output                       w_PWRITE,
788       output                       w_PSELx,
789       output                       w_PENABLE,
790       output  [DATA_WIDTH-1:0]     w_PWDATA,
791       input   [DATA_WIDTH-1:0]     w_PRDATA,
792       input                        w_PREADY
793
794   `ifndef DEF_CORE_HAS_INSTR_MEM
795       , // APB master interface to slave instruction memory
796       output reg [`APB_WIDTH-1:0]     w2_PADDR,
797       output reg                      w2_PWRITE,
798       output reg                      w2_PSELx,
799       output reg                      w2_PENABLE,
800       output reg [DATA_WIDTH-1:0]     w2_PWDATA,
801       input      [DATA_WIDTH-1:0]     w2_PRDATA,
802       input                           w2_PREADY
803   `endif
804   );
805       localparam STATE_IF = 0;
806       localparam STATE_R1 = 1;
807       localparam STATE_R2 = 2;
808       localparam STATE_ME = 3;
809       localparam STATE_WB = 4;
810       localparam STATE_FE = 5;
811       localparam STATE_IDLE = 6;
812       localparam STATE_HALT = 7;
813       reg  [2:0] r_state = STATE_IF;
814
815       reg  [DATA_WIDTH-1:0] r_pc          = 16'h0000;
816       reg  [DATA_WIDTH-1:0] r_pc_saved    = 16'h0000;
817       reg  [DATA_WIDTH-1:0] r_instr       = 16'h0000;
818       wire [DATA_WIDTH-1:0] w_mem_instr_out;
819       wire                  w_halt;
820
821       assign dbug = {7'h00, w_halt};
822       assign halt = w_halt;
823
824       wire [4:0]            r_instr_opcode;
825       wire [4:0]            r_instr_alu_op;
826       wire [2:0]            r_instr_rsd;
827       wire [2:0]            r_instr_rsa;
828       reg  [DATA_WIDTH-1:0] r_instr_rdd = 0;
829       reg  [DATA_WIDTH-1:0] r_instr_rda = 0;
830       wire [3:0]            r_instr_imm4;
831       wire [7:0]            r_instr_imm8;
832       wire [4:0]            r_instr_simm5;
833       wire                  r_instr_has_imm4;
834       wire                  r_instr_has_imm8;
835       wire                  r_instr_has_we;
836       wire                  r_instr_has_br;
837       wire                  r_instr_has_cmp;
838       wire                  r_instr_has_mem;
839       wire                  r_instr_has_mem_we;
840       wire                  r_instr_halt;
841       wire                  r_instr_has_lwex;
842       wire                  r_instr_has_swex;
843
844       wire [DATA_WIDTH-1:0] r_alu_out;
845
846       wire [DATA_WIDTH-1:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
847       wire [DATA_WIDTH-1:0] r_mem_scratch_in   = r_instr_rdd;
848       wire [DATA_WIDTH-1:0] r_mem_scratch_out;
849       wire                  r_mem_scratch_we   = r_instr_has_mem_we && (r_state == STATE_ME);
850       reg                   r_mem_scratch_req  = 0;
851       wire                  r_mem_scratch_busy;
852
853       reg  [2:0]            r_reg_rs1 = 0;
854       wire [DATA_WIDTH-1:0] r_reg_rd1_s;
855       wire [DATA_WIDTH-1:0] r_reg_rd1_i;
856       wire [DATA_WIDTH-1:0] r_reg_rd1 = regs_use_int ? r_reg_rd1_i : r_reg_rd1_s;
857       //wire [15:0] r_reg_rd2;
858       wire [DATA_WIDTH-1:0] r_reg_wd = (r_instr_has_mem) ? r_mem_scratch_out : r_alu_out;
859       wire                  r_reg_we = r_instr_has_we && (r_state == STATE_WB);
860
861       // branching
862       wire        w_intr;
863       wire        w_branch_en;
```

```
864     wire        w_branching   = r_instr_has_br && w_branch_en;
865     reg  [3:0]  r_cmp_flags   = 4'h00; // N, Z, C, V
866
867     always @(r_cmp_flags)
868         $display($time, "\tC%02h:\tALU CMP: %b", CORE_ID, r_cmp_flags);
869
870     // 2 cycle register fetch
871     always @(*) begin
872         r_reg_rs1 = 0;
873         if (r_state == STATE_R1)
874             r_reg_rs1 = r_instr_rsd;
875         else if (r_state == STATE_R2)
876             r_reg_rs1 = r_instr_rsa;
877         else
878             r_reg_rs1 = 3'h0;
879     end
880
881     reg regs_use_int = 0;
882     `ifdef DEF_ENABLE_INT
883     wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_vector;
884     wire [`DEF_NUM_INT-1:0]             ints_mask;
885     wire                                has_int = ints & ints_mask;
886     reg int_pending = 0;
887     reg int_pending_ack = 0;
888     always @(posedge clk)
889         if (int_pending_ack)
890             // We've now branched to the isr
891             int_pending <= 0;
892         else if (has_int)
893             // Notify fsm to switch to the ints_vector at the last stage
894             int_pending <= 1;
895         else if (w_intr)
896             // Return to Interrupt instruction called,
897             //   so we've finished with the interrupt
898             int_pending <= 0;
899     `endif
900
901     // Next program counter logic
902     reg [`DATA_WIDTH-1:0] next_pc = 0;
903     always @(posedge clk)
904         if (reset)
905             r_pc <= 0;
906         else if (r_state == STATE_WB) begin
907             `ifdef DEF_ENABLE_INT
908             if (int_pending) begin
909                 $display($time, "\tC%02h: Jumping to ISR: %h",
910                     CORE_ID,
911                     ints_vector[0 +: `DATA_WIDTH]);
912                 // TODO: check bounds
913                 // Save state
914                 r_pc_saved      <= r_pc + 1;
915                 regs_use_int    <= 1;
916                 int_pending_ack <= 1;
917                 // Jump to ISR
918                 r_pc            <= ints_vector[0 +: `DATA_WIDTH];
919             end else if (w_intr) begin
920                 $display($time, "\tC%02h: Returning from ISR: %h",
921                     CORE_ID, r_pc_saved);
922
923                 // Restore state
924                 r_pc            <= r_pc_saved;
925                 regs_use_int    <= 0;
926                 int_pending_ack <= 0;
927
928                 `ifndef DEF_CORE_HAS_INSTR_MEM
929                     w2_PADDR <= r_pc_saved;
930                 `endif
931             end else
932             `endif
933             if (w_branching) begin
934                 $display($time, "\tC%02h: branching to %h", CORE_ID, r_instr_rdd);
935                 r_pc            <= r_instr_rdd;
936
937                 `ifndef DEF_CORE_HAS_INSTR_MEM
938                     w2_PADDR <= r_instr_rdd;
939                 `endif
940
941                 `ifdef DEF_ENABLE_INT
942                     int_pending_ack <= 0;
943                 `endif
944             end else if (r_pc < (MEM_INSTR_DEPTH-1)) begin
945                 // normal increment
946                 // pc <= pc + 1
947                 r_pc            <= r_pc + 1;
948
949                 `ifndef DEF_CORE_HAS_INSTR_MEM
950                     // setup t1, t2 of apb
951                     w2_PADDR <= r_pc + 1;
952                 `endif
953
954                 `ifdef DEF_ENABLE_INT
```

```
955                     int_pending_ack <= 0;
956                 `endif
957             end
958         end // end r_state == STATE_WB
959         else if (r_state == STATE_HALT) begin
960             `ifdef DEF_ENABLE_INT
961             // Only an interrupt can return from halt
962             // duplicate code form STATE_ME!
963             if (int_pending) begin
964                 $display($time, "\tC%02h: Jumping to ISR: %h", CORE_ID, ints_vector[0 +: `DATA_WIDTH]);
965                 // TODO: check bounds
966                 // Save state
967                 r_pc_saved      <= r_pc;// + 1; HALT = stay with same PC
968                 regs_use_int    <= 1;
969                 int_pending_ack <= 1;
970                 // Jump to ISR
971                 r_pc            <= ints_vector[0 +: `DATA_WIDTH];
972                 r_state         <= STATE_FE;
973             end else if (w_intr) begin
974                 $display($time, "\tC%02h: Returning from ISR: %h", CORE_ID, r_pc_saved);
975                 r_pc            <= r_pc_saved;
976                 regs_use_int    <= 0;
977                 int_pending_ack <= 0;
978             end
979             `endif
980         end
981 
982 `ifndef DEF_CORE_HAS_INSTR_MEM
983     initial w2_PSELx   = 0;
984     initial w2_PENABLE = 0;
985     initial w2_PADDR   = 0;
986 `endif
987 
988     // cpu state machine
989     always @(posedge clk)
990         if (reset) begin
991             r_state           <= STATE_IF;
992             r_instr           <= 0;
993             r_mem_scratch_req <= 0;
994             r_instr_rdd       <= 0;
995             r_instr_rda       <= 0;
996         end
997         else begin
998 
999 `ifdef DEF_CORE_HAS_INSTR_MEM
1000             if (r_state == STATE_IF) begin
1001                     r_instr <= w_mem_instr_out;
1002 
1003                     $display("");
1004                     $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
1005                     $display($time, "\tC%02h: INSTR: %h", CORE_ID, w_mem_instr_out);
1006 
1007                     r_state <= STATE_R1;
1008             end
1009 `else
1010             // wait for global instruction rom to give us our instruction
1011             if (r_state == STATE_IF) begin
1012                 // wait for ready signal
1013                 if (!w2_PREADY) begin
1014                     w2_PSELx   <= 1;
1015                     w2_PWRITE  <= 0;
1016                     w2_PENABLE <= 1;
1017                     w2_PWDATA  <= 0;
1018                     w2_PADDR   <= r_pc;
1019                 end else begin
1020                     w2_PSELx   <= 0;
1021                     w2_PWRITE  <= 0;
1022                     w2_PENABLE <= 0;
1023                     w2_PWDATA  <= 0;
1024 
1025                     r_instr <= w2_PRDATA;
1026 
1027                     $display("");
1028                     $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
1029                     $display($time, "\tC%02h: INSTR: %h", CORE_ID, w2_PRDATA);
1030 
1031                     r_state <= STATE_R1;
1032                 end
1033             end
1034 `endif
1035 
1036             else if (r_state == STATE_R1) begin
1037                 if (w_halt) begin
1038                     $display("");
1039                     $display("");
1040                     $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
1041                     r_state <= STATE_HALT;
1042                 end else begin
1043                     // primary operand
1044                     r_instr_rdd <= r_reg_rd1;
1045                     r_state     <= STATE_R2;
```

```
1046                end
1047            end
1048            else if (r_state == STATE_R2) begin
1049                // Choose secondary operand (register or immediate)
1050                if      (r_instr_has_imm8)  r_instr_rda <= r_instr_imm8;
1051                else if (r_instr_has_imm4)  r_instr_rda <= r_reg_rd1 + r_instr_imm4;
1052                else                        r_instr_rda <= r_reg_rd1;
1053
1054                if (r_instr_has_mem) begin
1055                    r_state           <= STATE_ME;
1056                    // Pulse req
1057                    r_mem_scratch_req <= 1;
1058                end else
1059                    r_state <= STATE_WB;
1060            end
1061            else if (r_state == STATE_ME) begin
1062                // Pulse req
1063                r_mem_scratch_req <= 0;
1064                // Wait for MMU to finish
1065                if (!r_mem_scratch_busy)
1066                    r_state <= STATE_WB;
1067            end
1068            else if (r_state == STATE_WB) begin
1069                if (r_instr_has_cmp) begin
1070                    $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
1071                    r_cmp_flags <= r_alu_out[3:0];
1072                end
1073
1074                r_state <= STATE_FE;
1075            end
1076            else if (r_state == STATE_FE)
1077                r_state <= STATE_IF;
1078        end
1079
1080    `ifdef DEF_CORE_HAS_INSTR_MEM
1081        // Instruction ROM
1082        (* rom_style = "distributed" *)
1083        vmicro16_bram # (
1084            .MEM_WIDTH      (DATA_WIDTH),
1085            .MEM_DEPTH      (MEM_INSTR_DEPTH),
1086            .CORE_ID        (CORE_ID),
1087            .USE_INITS      (1),
1088            .NAME           ("INSTR_MEM")
1089        ) mem_instr (
1090            .clk            (clk),
1091            .reset          (reset),
1092            // port 1
1093            .mem_addr       (r_pc),
1094            .mem_in         (0),
1095            .mem_we         (1'b0),  // ROM
1096            .mem_out        (w_mem_instr_out)
1097        );
1098    `endif
1099
1100        // MMU
1101        vmicro16_core_mmu # (
1102            .MEM_WIDTH      (DATA_WIDTH),
1103            .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
1104            .CORE_ID        (CORE_ID)
1105        ) mmu (
1106            .clk            (clk),
1107            .reset          (reset),
1108            .req            (r_mem_scratch_req),
1109            .busy           (r_mem_scratch_busy),
1110            // interrupts
1111            .ints_vector    (ints_vector),
1112            .ints_mask      (ints_mask),
1113            // port 1
1114            .mmu_addr       (r_mem_scratch_addr),
1115            .mmu_in         (r_mem_scratch_in),
1116            .mmu_we         (r_mem_scratch_we),
1117            .mmu_lwex       (r_instr_has_lwex),
1118            .mmu_swex       (r_instr_has_swex),
1119            .mmu_out        (r_mem_scratch_out),
1120            // APB maste    r to slave
1121            .M_PADDR        (w_PADDR),
1122            .M_PWRITE       (w_PWRITE),
1123            .M_PSELx        (w_PSELx),
1124            .M_PENABLE      (w_PENABLE),
1125            .M_PWDATA       (w_PWDATA),
1126            .M_PRDATA       (w_PRDATA),
1127            .M_PREADY       (w_PREADY)
1128        );
1129
1130        // Instruction decoder
1131        vmicro16_dec dec (
1132            // input
1133            .instr          (r_instr),
1134            // output async
1135            .opcode         (),
1136            .rd             (r_instr_rsd),
```

```
1137            .ra             (r_instr_rsa),
1138            .imm4           (r_instr_imm4),
1139            .imm8           (r_instr_imm8),
1140            .imm12          (),
1141            .simm5          (r_instr_simm5),
1142            .alu_op         (r_instr_alu_op),
1143            .has_imm4       (r_instr_has_imm4),
1144            .has_imm8       (r_instr_has_imm8),
1145            .has_we         (r_instr_has_we),
1146            .has_br         (r_instr_has_br),
1147            .has_cmp        (r_instr_has_cmp),
1148            .has_mem        (r_instr_has_mem),
1149            .has_mem_we     (r_instr_has_mem_we),
1150            .halt           (w_halt),
1151            .intr           (w_intr),
1152            .has_lwex       (r_instr_has_lwex),
1153            .has_swex       (r_instr_has_swex)
1154        );
1155
1156        // Software registers
1157        vmicro16_regs # (
1158            .CORE_ID    (CORE_ID),
1159            .CELL_WIDTH (`DATA_WIDTH)
1160        ) regs (
1161            .clk        (clk),
1162            .reset      (reset),
1163            // async port 0
1164            .rs1        (r_reg_rs1),
1165            .rd1        (r_reg_rd1_s),
1166            // async port 1
1167            //.rs2        (),
1168            //.rd2        (),
1169            // write port
1170            .we         (r_reg_we && ~regs_use_int),
1171            .ws1        (r_instr_rsd),
1172            .wd         (r_reg_wd)
1173        );
1174
1175        // Interrupt replacement registers
1176        `ifdef DEF_ENABLE_INT
1177        vmicro16_regs # (
1178            .CORE_ID    (CORE_ID),
1179            .CELL_WIDTH (`DATA_WIDTH),
1180            .DEBUG_NAME ("REGSINT")
1181        ) regs_intr (
1182            .clk        (clk),
1183            .reset      (reset),
1184            // async port 0
1185            .rs1        (r_reg_rs1),
1186            .rd1        (r_reg_rd1_i),
1187            // async port 1
1188            //.rs2        (),
1189            //.rd2        (),
1190            // write port
1191            .we         (r_reg_we && regs_use_int),
1192            .ws1        (r_instr_rsd),
1193            .wd         (r_reg_wd)
1194        );
1195        `endif
1196
1197        // ALU
1198        vmicro16_alu # (
1199            .CORE_ID(CORE_ID)
1200        ) alu (
1201            .op         (r_instr_alu_op),
1202            .a          (r_instr_rdd),
1203            .b          (r_instr_rda),
1204            .flags      (r_cmp_flags),
1205            // async output
1206            .c          (r_alu_out)
1207        );
1208
1209        branch branch_check (
1210            .flags      (r_cmp_flags),
1211            .cond       (r_instr_imm8),
1212            .en         (w_branch_en)
1213        );
1214
1215    endmodule
```