

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 30, 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 30, 2019

Table of Contents

1	Interconnect	5
1.1	Introduction	6
1.2	Overview	6
1.2.1	Design Considerations	6
1.3	Peripheral Interconnect Interface	6
1.3.1	Master to Slave Interface	6
1.3.2	Variable Core Support	7
1.4	Shared Bus Arbitration	7
2	Memory Mapping	8
2.1	Memory Map	9
2.2	Special Registers	10
3	Interrupts	11
3.1	Why Interrupts?	11
3.2	Hardware Implementation	11
3.2.1	Context Switching	11
3.3	Software Interface	12
3.3.1	Interrupt Vector (0x0100-0x0107)	12
3.3.2	Interrupt Mask (0x0108)	12
3.3.3	Software Example	13
3.4	Design Improvements	13
4	Peripherals	14
4.1	Watchdog Timer	14
4.2	GPIO Interface	14
4.3	Timer with Interrupt	15
4.4	UART Interface	15
5	System-on-Chip Layout	16
6	Analysis & Results	17
7	Improvements	18
7.1	Foo	18

8 Conclusion	19
8.1 Foo	19
Appendices	20
A Configuration Options	20
A.1 SoC Options	20
A.2 Core Options	20
A.3 Peripheral Options	21
B Code Listing	22
B.1 vmicro16_soc_config.v	22
B.2 top_ms.v	23
B.3 vmicro16_soc.v	25
B.4 vmicro16_periph.v	30
B.5 vmicro16.v	35

Chapter 1

Interconnect

1.1	Introduction	6
1.2	Overview	6
1.2.1	Design Considerations	6
1.3	Peripheral Interconnect Interface	6
1.3.1	Master to Slave Interface	6
1.3.2	Variable Core Support	7
1.4	Shared Bus Arbitration	7

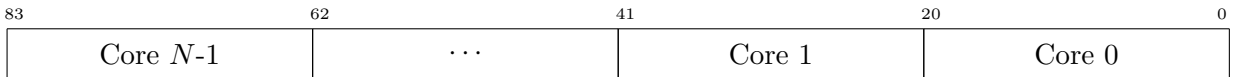
1.3.2 Variable Core Support

```

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 1.1: Variable size inputs and outputs to the interconnect.



1.4 Shared Bus Arbitration

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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 2

Memory Mapping

2.1	Memory Map	9
2.2	Special Registers	10

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

2.1 Memory Map

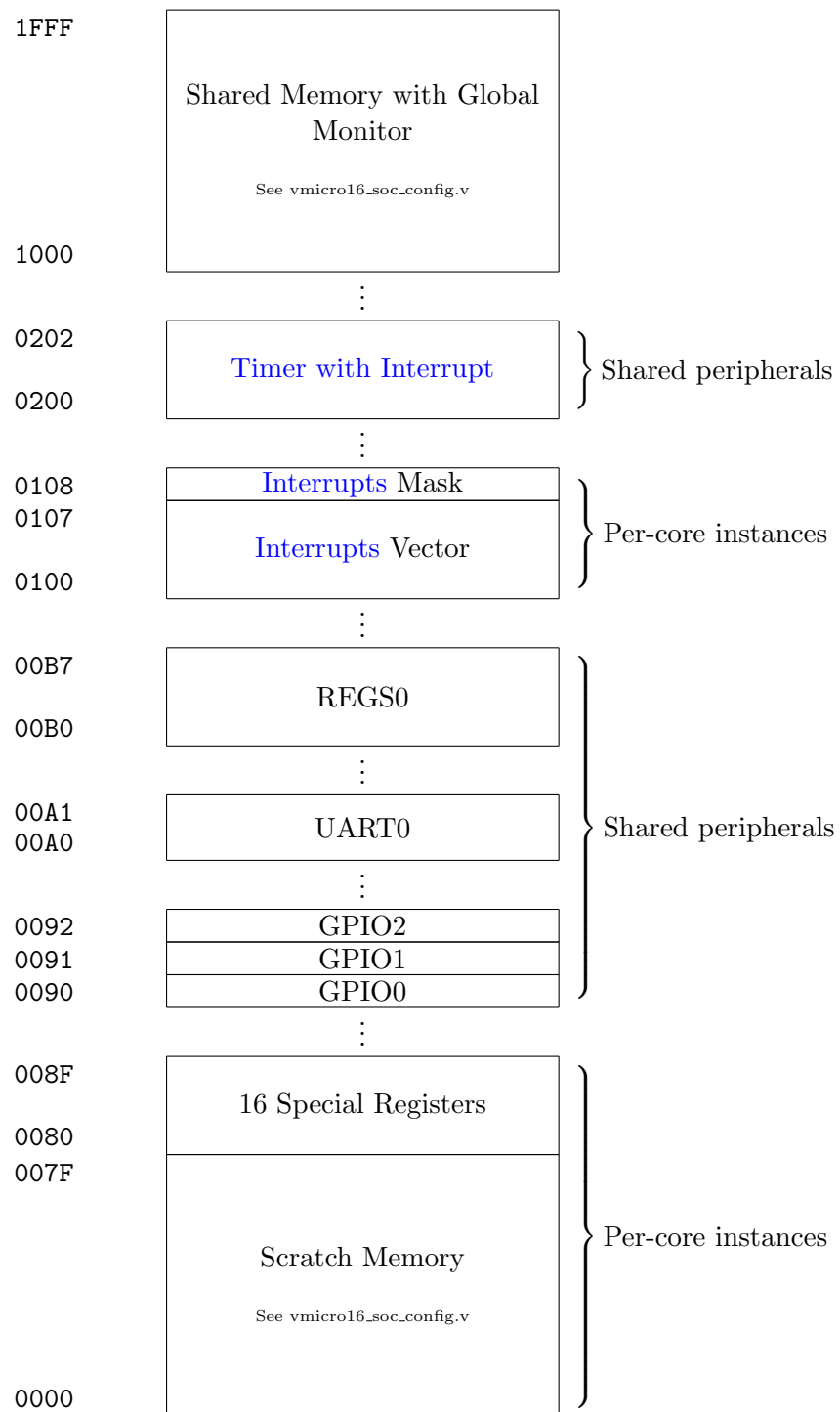


Figure 2.1: Memory map showing addresses of various memory sections.

2.2 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.

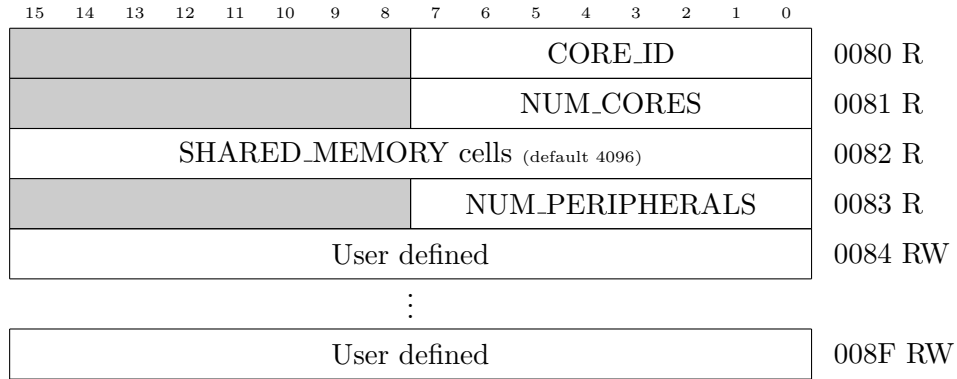


Figure 2.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

Chapter 3

Interrupts

3.1	Why Interrupts?	11
3.2	Hardware Implementation	11
3.2.1	Context Switching	11
3.3	Software Interface	12
3.3.1	Interrupt Vector (0x0100-0x0107)	12
3.3.2	Interrupt Mask (0x0108)	12
3.3.3	Software Example	13
3.4	Design Improvements	13

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

3.1 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.

3.2 Hardware Implementation

3.2.1 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 3.1 visually describes this process.

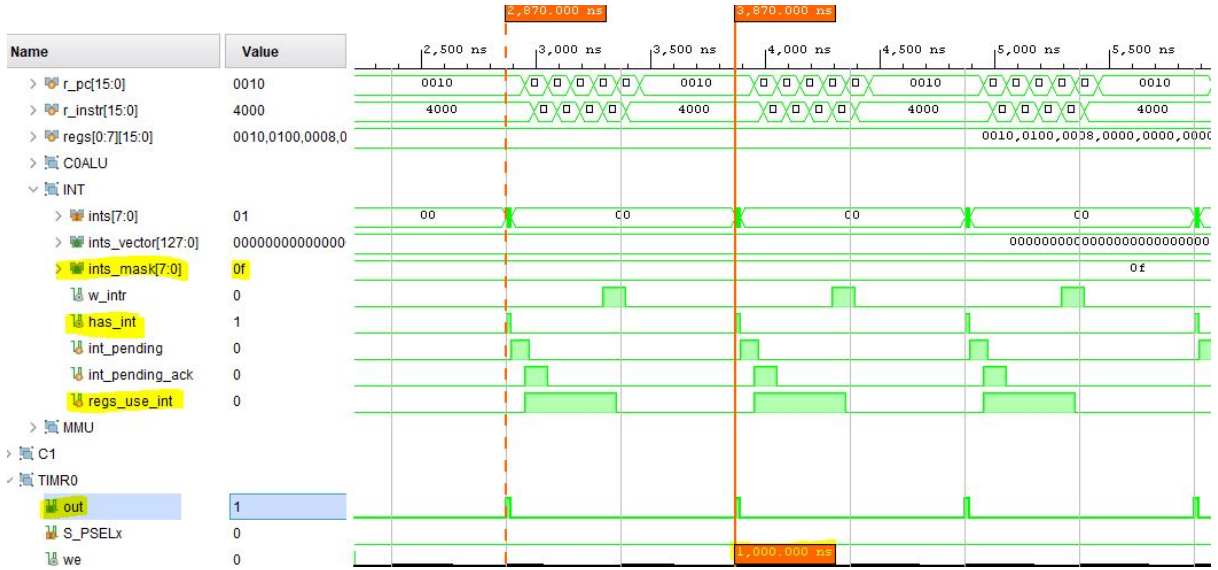


Figure 3.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return instruction (indicated by w_intr) the processor returns to normal mode and restores the normal state.

3.3 Software Interface

To enable software to

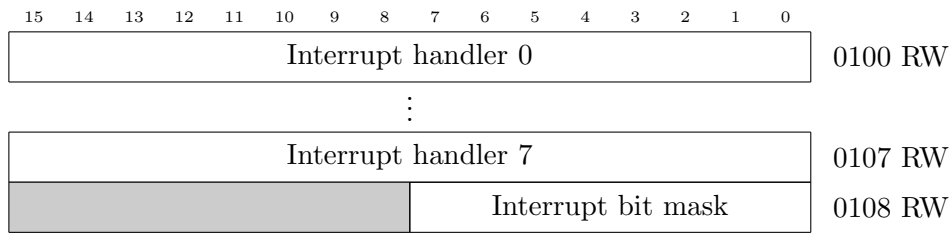


Figure 3.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

3.3.1 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.

3.3.2 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,

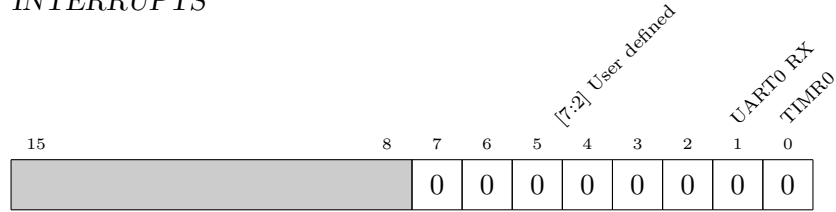


Figure 3.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

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.

3.3.3 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      lshift  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:
18     movi    r0, #0xff        // arbitrary name
19     intr                    // do something
                             // return from interrupt

```

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

3.4 Design Improvements

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

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

Chapter 4

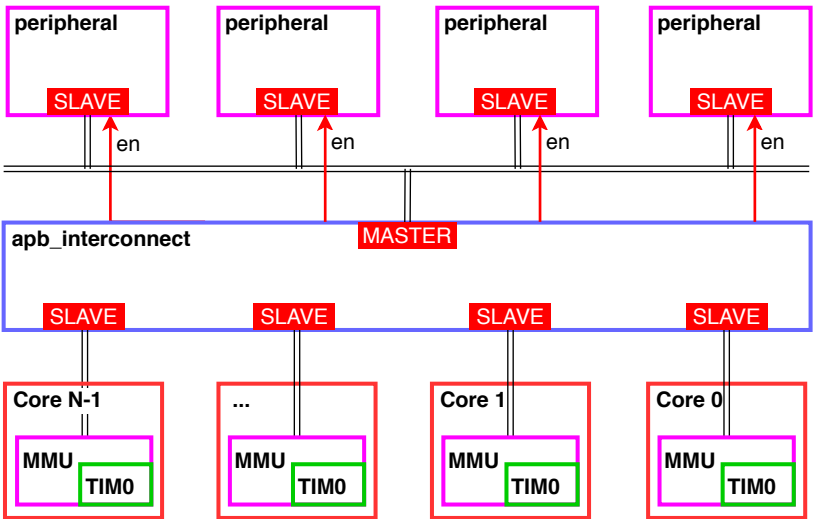
Peripherals

4.1	Watchdog Timer	14
4.2	GPIO Interface	14
4.3	Timer with Interrupt	15
4.4	UART Interface	15

4.1 Watchdog Timer

In any multi-threaded system there exists the possibility for a deadlock – a state where all threads are in a waiting state – and algorithm execution is forever blocked.

A common method of detecting a deadlock is to periodically check that a thread is.



4.2 GPIO Interface

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
GPIO0 Output															0090 RW	
GPIO1 Output															0091 RW	
GPIO1 Input															0092 R	

4.3 Timer with Interrupt

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Load Value																0200 RW
													I	R	S	0201 W
Prescaler																0202 W

4.4 UART Interface

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

Chapter 5

System-on-Chip Layout

The Vmicro16 processor uses



Figure 5.1:

Chapter 6

Analysis & Results

Chapter 7

Improvements

7.1 Foo	18
-------------------	----

7.1 Foo

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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	19
-------------------	----

8.1 Foo

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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	20
A.2 Core Options	20
A.3 Peripheral Options	21

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 GPIO0 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	22
B.2	top_ms.v	23
B.3	vmicro16_soc.v	25
B.4	vmicro16_periph.v	30
B.5	vmicro16.v	35

B.1 vmicro16_soc_config.v

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

```

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

```



```

47 `define DEF_USE_WATCHDOG
48
49 //////////////////////////////////////////////////
50 // Memory mapping
51 //////////////////////////////////////////////////
52 `define APB_WIDTH      (2 + `clog2(`CORES) + `DATA_WIDTH)
53
54 `define APB_PSELX_GPI00 0
55 `define APB_PSELX_UART0 1
56 `define APB_PSELX_REGSO 2
57 `define APB_PSELX_BRAMO 3
58 `define APB_PSELX_GPI01 4
59 `define APB_PSELX_GPI02 5
60 `define APB_PSELX_TIMRO 6
61 `define APB_PSELX_WDOGO 7
62
63 `define APB_GPI00_PINS 8
64 `define APB_GPI01_PINS 16
65 `define APB_GPI02_PINS 8
66
67 // Shared memory words
68 `define APB_BRAMO_CELLS 4096
69
70 //////////////////////////////////////////////////
71 // Memory mapping
72 //////////////////////////////////////////////////
73 // TIMO
74 // Number of scratch memory cells per core
75 `define DEF_MMU_TIMO_CELLS 64
76 `define DEF_MMU_TIMO_S    16'h0000
77 `define DEF_MMU_TIMO_E    16'h007F
78 // SREG
79 `define DEF_MMU_SREG_S    16'h0080
80 `define DEF_MMU_SREG_E    16'h008F
81 // GPI00
82 `define DEF_MMU_GPI00_S   16'h0090
83 `define DEF_MMU_GPI00_E   16'h0090
84 // GPI01
85 `define DEF_MMU_GPI01_S   16'h0091
86 `define DEF_MMU_GPI01_E   16'h0091
87 // GPI02
88 `define DEF_MMU_GPI02_S   16'h0092
89 `define DEF_MMU_GPI02_E   16'h0092
90 // UART0
91 `define DEF_MMU_UART0_S   16'h00A0
92 `define DEF_MMU_UART0_E   16'h00A1
93 // REGSO
94 `define DEF_MMU_REGSO_S   16'h00B0
95 `define DEF_MMU_REGSO_E   16'h00B7
96 // WDOGO
97 `define DEF_MMU_WDOGO_S   16'h00B8
98 `define DEF_MMU_WDOGO_E   16'h00B8
99 // BRAMO
100 `define DEF_MMU_BRAMO_S   16'h1000
101 `define DEF_MMU_BRAMO_E   16'h1fff
102 // TIMRO
103 `define DEF_MMU_TIMRO_S   16'h0200
104 `define DEF_MMU_TIMRO_E   16'h0202
105
106 //////////////////////////////////////////////////
107 // Interrupts
108 //////////////////////////////////////////////////
109 // Enable/disable interrupts
110 // Disabling will free up resources for other features
111 `define DEF_ENABLE_INT
112 // Number of interrupt in signals
113 `define DEF_NUM_INT      8
114 // Default interrupt bitmask (0 = hidden, 1 = enabled)
115 `define DEF_INT_MASK     0
116 // Bit position of the TIMRO interrupt signal
117 `define DEF_INT_TIMRO    0
118 // Interrupt vector memory location
119 `define DEF_MMU_INTSV_S   16'h0100
120 `define DEF_MMU_INTSV_E   16'h0107
121 // Interrupt vector memory location
122 `define DEF_MMU_INTSM_S   16'h0108
123 `define DEF_MMU_INTSM_E   16'h0108
124
125
126 `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
32 // minispartan6+ XC6SLX9
33 module top_ms # (
34     parameter GPIO_PINS = 8
35 ) (
36     input          CLK50,
37     input [3:0]    SW,
38     // UART
39     //input        RXD,
40     output         TXD,
41     // Peripherals
42     output [7:0]   LEDS,
43
44     // SSDs
45     output [6:0]   ssd0,
46     output [6:0]   ssd1,
47     output [6:0]   ssd2,
48     output [6:0]   ssd3,
49     output [6:0]   ssd4,
50     output [6:0]   ssd5
51 );
52 //wire [15:0]      M_PADDR;
53 //wire            M_PWRITE;
54 //wire [5-1:0]    M_PSELx; // not shared
55 //wire            M_PENABLE;
56 //wire [15:0]     M_PWDATA;
57 //wire [15:0]     M_PRDATA; // input to intercon
58 //wire            M_PREADY; // input to intercon
59
60 wire [7:0] gpio0;
61 wire [15:0] gpio1;
62 wire [7:0] gpio2;
63
64 vmicro16_soc soc (
65     .clk      (CLK50),
66     .reset    (~SW[0]),
67
68     // .M_PADDR      (M_PADDR),
69     // .M_PWRITE     (M_PWRITE),
70     // .M_PSELx      (M_PSELx),
71     // .M_PENABLE    (M_PENABLE),
72     // .M_PWDATA     (M_PWDATA),
73     // .M_PRDATA     (M_PRDATA),
74     // .M_PREADY     (M_PREADY),
75
76     .uart_tx (TXD),
77     .gpio0   (LEDS[3:0]),
78     .gpio1   (gpio1),
79     .gpio2   (gpio2),
80
81     .debug0  (LEDS[7:4])
82     // .debug1 (LEDS[7:4])
83 );
84
85 // SSD displays (split across 2 gpio ports 1 and 2)
86 wire [3:0] ssd_chars [0:5];
87 assign ssd_chars[0] = gpio1[3:0];
88 assign ssd_chars[1] = gpio1[7:4];
89 assign ssd_chars[2] = gpio1[11:8];
90 assign ssd_chars[3] = gpio1[15:12];
91 assign ssd_chars[4] = gpio2[3:0];
92 assign ssd_chars[5] = gpio2[7:4];
93 seven_display ssd_0 (.n(ssd_chars[0]), .segments (ssd0));

```

```

94     seven_display ssd_1 (.n(ssd_chars[1]), .segments (ssd1));
95     seven_display ssd_2 (.n(ssd_chars[2]), .segments (ssd2));
96     seven_display ssd_3 (.n(ssd_chars[3]), .segments (ssd3));
97     seven_display ssd_4 (.n(ssd_chars[4]), .segments (ssd4));
98     seven_display ssd_5 (.n(ssd_chars[5]), .segments (ssd5));
99
100 endmodule

```

B.3 vmicro16_soc.v

```

1  //
2  //
3
4  `include "vmicro16_soc_config.v"
5  `include "clog2.v"
6  `include "formal.v"
7
8  module pow_reset # (
9      parameter INIT = 1,
10     parameter N = 8
11 ) (
12     input clk,
13     input reset,
14     output reg resethold
15 );
16     initial resethold = INIT ? (N-1) : 0;
17
18     always @(*)
19         resethold = |hold;
20
21     reg [`clog2(N)-1:0] hold = (N-1);
22     always @(posedge clk)
23         if (reset)
24             hold <= N-1;
25         else
26             if (hold)
27                 hold <= hold - 1;
28 endmodule
29
30 // Vmicro16 multi-core SoC with various peripherals
31 // and interrupts
32 module vmicro16_soc (
33     input clk,
34     input reset,
35
36     //input uart_rx,
37     output uart_tx,
38     output [`APB_GPIO0_PINS-1:0] gpio0,
39     output [`APB_GPIO1_PINS-1:0] gpio1,
40     output [`APB_GPIO2_PINS-1:0] gpio2,
41
42     output halt,
43
44     output [`CORES-1:0] debug0,
45     output [`CORES*8-1:0] debug1
46 );
47     wire [`CORES-1:0] w_halt;
48     assign halt = &w_halt;
49
50     assign debug0 = w_halt;
51
52     // Watchdog reset pulse signal.
53     // Passed to pow_reset to generate a longer reset pulse
54     wire wdreset;
55
56     // soft register reset hold for brams and registers
57     wire soft_reset;
58     `ifdef DEF_GLOBAL_RESET
59         pow_reset # (
60             .INIT (1),
61             .N (8)
62         ) por_inst (
63             .clk (clk),
64             `ifdef DEF_USE_WATCHDOG
65             .reset (reset | wdreset),
66             `else
67             .reset (reset),
68             `endif
69             .resethold (soft_reset)
70         );
71     `else
72         assign soft_reset = 0;
73     `endif
74
75     // Peripherals (master to slave)
76     wire [`APB_WIDTH-1:0] M_PADDR;
77     wire M_PWRITE;
78     wire [`SLAVES-1:0] M_PSELx; // not shared

```

```

79     wire                                M_PENABLE;
80     wire [DATA_WIDTH-1:0]              M_PWDATA;
81     wire [SLAVES*DATA_WIDTH-1:0]       M_PRDATA; // input to intercon
82     wire [SLAVES-1:0]                  M_PREADY; // input
83
84     // Master apb interfaces
85     wire [CORES*APB_WIDTH-1:0]          w_PADDR;
86     wire [CORES-1:0]                    w_PWRITE;
87     wire [CORES-1:0]                    w_PSELx;
88     wire [CORES-1:0]                    w_PENABLE;
89     wire [CORES*DATA_WIDTH-1:0]         w_PWDATA;
90     wire [CORES*DATA_WIDTH-1:0]         w_PRDATA;
91     wire [CORES-1:0]                    w_PREADY;
92
93     // Interrupts
94     `ifdef DEF_ENABLE_INT
95     wire [DEF_NUM_INT-1:0]              ints;
96     wire [DEF_NUM_INT*DATA_WIDTH-1:0]   ints_data;
97     assign ints[7:1] = 0;
98     assign ints_data[DEF_NUM_INT*DATA_WIDTH-1:DATA_WIDTH] =
99         {DEF_NUM_INT*(DATA_WIDTH-1){1'b0}};
100     `endif
101
102     apb_intercon_s # (
103         .MASTER_PORTS    (`CORES),
104         .SLAVE_PORTS      (`SLAVES),
105         .BUS_WIDTH        (`APB_WIDTH),
106         .DATA_WIDTH       (`DATA_WIDTH),
107         .HAS_PSELX_ADDR   (1)
108     ) apb (
109         .clk               (clk),
110         .reset             (soft_reset),
111         // APB master to slave
112         .S_PADDR           (w_PADDR),
113         .S_PWRITE          (w_PWRITE),
114         .S_PSELx           (w_PSELx),
115         .S_PENABLE         (w_PENABLE),
116         .S_PWDATA          (w_PWDATA),
117         .S_PRDATA          (w_PRDATA),
118         .S_PREADY          (w_PREADY),
119         // shared bus
120         .M_PADDR           (M_PADDR),
121         .M_PWRITE          (M_PWRITE),
122         .M_PSELx           (M_PSELx),
123         .M_PENABLE         (M_PENABLE),
124         .M_PWDATA          (M_PWDATA),
125         .M_PRDATA          (M_PRDATA),
126         .M_PREADY         (M_PREADY)
127     );
128
129     `ifdef DEF_USE_WATCHDOG
130     vmicro16_watchdog_apb # (
131         .BUS_WIDTH        (`APB_WIDTH),
132         .NAME              ("WD0G0")
133     ) wd0g0_apb (
134         .clk               (clk),
135         .reset             (),
136         // apb slave to master interface
137         .S_PADDR           (),
138         .S_PWRITE          (M_PWRITE),
139         .S_PSELx           (M_PSELx[`APB_PSELX_WD0G0]),
140         .S_PENABLE         (M_PENABLE),
141         .S_PWDATA          (),
142         .S_PRDATA          (),
143         .S_PREADY         (M_PREADY[`APB_PSELX_WD0G0]),
144
145         .wdreset           (wdreset)
146     );
147     `endif
148
149     vmicro16_gpio_apb # (
150         .BUS_WIDTH        (`APB_WIDTH),
151         .DATA_WIDTH       (`DATA_WIDTH),
152         .PORTS            (`APB_GPIO0_PINS),
153         .NAME              ("GPIO0")
154     ) gpio0_apb (
155         .clk               (clk),
156         .reset             (soft_reset),
157         // apb slave to master interface
158         .S_PADDR           (M_PADDR),
159         .S_PWRITE          (M_PWRITE),
160         .S_PSELx           (M_PSELx[`APB_PSELX_GPIO0]),
161         .S_PENABLE         (M_PENABLE),
162         .S_PWDATA          (M_PWDATA),
163         .S_PRDATA          (M_PRDATA[`APB_PSELX_GPIO0*DATA_WIDTH+:`DATA_WIDTH]),
164         .S_PREADY         (M_PREADY[`APB_PSELX_GPIO0]),
165         .gpio              (gpio0)
166     );
167
168     // GPIO1 for Seven segment displays (16 pin)
169     vmicro16_gpio_apb # (

```

```

170     .BUS_WIDTH    (`APB_WIDTH),
171     .DATA_WIDTH   (`DATA_WIDTH),
172     .PORTS        (`APB_GPIO1_PINS),
173     .NAME          ("GPIO1")
174 ) gpio1_apb (
175     .clk           (clk),
176     .reset         (soft_reset),
177     // apb slave to master interface
178     .S_PADDR       (M_PADDR),
179     .S_PWRITE      (M_PWRITE),
180     .S_PSELx       (M_PSELx[`APB_PSELX_GPIO1]),
181     .S_PENABLE     (M_PENABLE),
182     .S_PWDATA      (M_PWDATA),
183     .S_PRDATA      (M_PRDATA[`APB_PSELX_GPIO1*`DATA_WIDTH +: `DATA_WIDTH]),
184     .S_PREADY      (M_PREADY[`APB_PSELX_GPIO1]),
185     .gpio          (gpio1)
186 );
187
188 // GPIO2 for Seven segment displays (8 pin)
189 vmicro16_gpio_apb # (
190     .BUS_WIDTH    (`APB_WIDTH),
191     .DATA_WIDTH   (`DATA_WIDTH),
192     .PORTS        (`APB_GPIO2_PINS),
193     .NAME          ("GPIO2")
194 ) gpio2_apb (
195     .clk           (clk),
196     .reset         (soft_reset),
197     // apb slave to master interface
198     .S_PADDR       (M_PADDR),
199     .S_PWRITE      (M_PWRITE),
200     .S_PSELx       (M_PSELx[`APB_PSELX_GPIO2]),
201     .S_PENABLE     (M_PENABLE),
202     .S_PWDATA      (M_PWDATA),
203     .S_PRDATA      (M_PRDATA[`APB_PSELX_GPIO2*`DATA_WIDTH +: `DATA_WIDTH]),
204     .S_PREADY      (M_PREADY[`APB_PSELX_GPIO2]),
205     .gpio          (gpio2)
206 );
207
208 apb_uart_tx # (
209     .DATA_WIDTH   (8),
210     .ADDR_EXP     (4) //2^4 = 16 FIFO words
211 ) uart0_apb (
212     .clk           (clk),
213     .reset         (soft_reset),
214     // apb slave to master interface
215     .S_PADDR       (M_PADDR),
216     .S_PWRITE      (M_PWRITE),
217     .S_PSELx       (M_PSELx[`APB_PSELX_UART0]),
218     .S_PENABLE     (M_PENABLE),
219     .S_PWDATA      (M_PWDATA),
220     .S_PRDATA      (M_PRDATA[`APB_PSELX_UART0*`DATA_WIDTH +: `DATA_WIDTH]),
221     .S_PREADY      (M_PREADY[`APB_PSELX_UART0]),
222     // uart wires
223     .tx_wire       (uart_tx),
224     .rx_wire       (uart_rx)
225 );
226
227 timer_apb timr0 (
228     .clk           (clk),
229     .reset         (soft_reset),
230     // apb slave to master interface
231     .S_PADDR       (M_PADDR),
232     .S_PWRITE      (M_PWRITE),
233     .S_PSELx       (M_PSELx[`APB_PSELX_TIMR0]),
234     .S_PENABLE     (M_PENABLE),
235     .S_PWDATA      (M_PWDATA),
236     .S_PRDATA      (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
237     .S_PREADY      (M_PREADY[`APB_PSELX_TIMR0])
238     //
239     `ifdef DEF_ENABLE_INT
240     ,.out           (ints [`DEF_INT_TIMR0]),
241     .int_data       (ints_data[`DEF_INT_TIMR0*`DATA_WIDTH +: `DATA_WIDTH])
242     `endif
243 );
244
245 // Shared register set for system-on-chip info
246 // R0 = number of cores
247 vmicro16_regs_apb # (
248     .BUS_WIDTH    (`APB_WIDTH),
249     .DATA_WIDTH   (`DATA_WIDTH),
250     .CELL_DEPTH   (8),
251     .PARAM_DEFAULTS_R0  (`CORES),
252     .PARAM_DEFAULTS_R1  (`SLAVES)
253 ) regs0_apb (
254     .clk           (clk),
255     .reset         (soft_reset),
256     // apb slave to master interface
257     .S_PADDR       (M_PADDR),
258     .S_PWRITE      (M_PWRITE),
259     .S_PSELx       (M_PSELx[`APB_PSELX_REGS0]),
260     .S_PENABLE     (M_PENABLE),

```

```

261     .S_PWDATA    (M_PWDATA),
262     .S_PRDATA    (M_PRDATA[~APB_PSELX_REGSO*`DATA_WIDTH +: `DATA_WIDTH]),
263     .S_PREADY    (M_PREADY[~APB_PSELX_REGSO])
264 );
265
266 vmicro16_bram_ex_apb # (
267     .BUS_WIDTH    (`APB_WIDTH),
268     .MEM_WIDTH    (`DATA_WIDTH),
269     .MEM_DEPTH    (`APB_BRAMO_CELLS),
270     .CORE_ID_BITS (`clog2(`CORES))
271 ) bram_apb (
272     .clk           (clk),
273     .reset         (soft_reset),
274     // apb slave to master interface
275     .S_PADDR       (M_PADDR),
276     .S_PWRITE      (M_PWRITE),
277     .S_PSELx       (M_PSELx[~APB_PSELX_BRAMO]),
278     .S_PENABLE     (M_PENABLE),
279     .S_PWDATA      (M_PWDATA),
280     .S_PRDATA      (M_PRDATA[~APB_PSELX_BRAMO*`DATA_WIDTH +: `DATA_WIDTH]),
281     .S_PREADY      (M_PREADY[~APB_PSELX_BRAMO])
282 );
283
284 // There must be atleast 1 core
285 `static_assert(`CORES > 0)
286 `static_assert(`DEF_MEM_INSTR_DEPTH > 0)
287 `static_assert(`DEF_MMU_TIMO_CELLS > 0)
288
289 // Single instruction memory
290 `ifndef DEF_CORE_HAS_INSTR_MEM
291 // slave input/outputs from interconnect
292 wire [~APB_WIDTH-1:0] instr_M_PADDR;
293 wire [1:0] instr_M_PWRITE;
294 wire [1:0] instr_M_PSELx; // not shared
295 wire [1:0] instr_M_PENABLE;
296 wire [~DATA_WIDTH-1:0] instr_M_PWDATA;
297 wire [1*`DATA_WIDTH-1:0] instr_M_PRDATA; // slave response
298 wire [1:0] instr_M_PREADY; // slave response
299
300 // Master apb interfaces
301 wire [~CORES*`APB_WIDTH-1:0] instr_w_PADDR;
302 wire [~CORES-1:0] instr_w_PWRITE;
303 wire [~CORES-1:0] instr_w_PSELx;
304 wire [~CORES-1:0] instr_w_PENABLE;
305 wire [~CORES*`DATA_WIDTH-1:0] instr_w_PWDATA;
306 wire [~CORES*`DATA_WIDTH-1:0] instr_w_PRDATA;
307 wire [~CORES-1:0] instr_w_PREADY;
308
309 vmicro16_bram_apb # (
310     .BUS_WIDTH    (`APB_WIDTH),
311     .MEM_WIDTH    (`DATA_WIDTH),
312     .MEM_DEPTH    (`DEF_MEM_INSTR_DEPTH),
313     .USE_INITS    (1),
314     .NAME         ("INSTR_ROM_G")
315 ) instr_rom_apb (
316     .clk           (clk),
317     .reset         (soft_reset),
318     .S_PADDR       (instr_M_PADDR),
319     .S_PWRITE      (),
320     .S_PSELx       (instr_M_PSELx),
321     .S_PENABLE     (instr_M_PENABLE),
322     .S_PWDATA      (),
323     .S_PRDATA      (instr_M_PRDATA),
324     .S_PREADY      (instr_M_PREADY)
325 );
326
327 apb_intercon_s # (
328     .MASTER_PORTS (`CORES),
329     .SLAVE_PORTS   (1),
330     .BUS_WIDTH     (`APB_WIDTH),
331     .DATA_WIDTH    (`DATA_WIDTH),
332     .HAS_PSELX_ADDR (0)
333 ) apb_instr_intercon (
334     .clk           (clk),
335     .reset         (soft_reset),
336     // APB master from cores
337     // master
338     .S_PADDR       (instr_w_PADDR),
339     .S_PWRITE      (instr_w_PWRITE),
340     .S_PSELx       (instr_w_PSELx),
341     .S_PENABLE     (instr_w_PENABLE),
342     .S_PWDATA      (instr_w_PWDATA),
343     .S_PRDATA      (instr_w_PRDATA),
344     .S_PREADY      (instr_w_PREADY),
345     // shared bus slaves
346     // slave outputs
347     .M_PADDR       (instr_M_PADDR),
348     .M_PWRITE      (instr_M_PWRITE),
349     .M_PSELx       (instr_M_PSELx),
350     .M_PENABLE     (instr_M_PENABLE),

```

```

352     .M_PWDATA    (instr_M_PWDATA),
353     .M_PRDATA    (instr_M_PRDATA),
354     .M_PREADY    (instr_M_PREADY)
355 );
356 `endif
357
358 genvar i;
359 generate for(i = 0; i < `CORES; i = i + 1) begin : cores
360
361     vmicro16_core # (
362         .CORE_ID        (i),
363         .DATA_WIDTH      (`DATA_WIDTH),
364
365         .MEM_INSTR_DEPTH (`DEF_MEM_INSTR_DEPTH),
366         .MEM_SCRATCH_DEPTH (`DEF_MMU_TIMO_CELLS)
367     ) c1 (
368         .clk            (clk),
369         .reset          (soft_reset),
370
371         // debug
372         .halt           (w_halt[i]),
373
374         // interrupts
375         .ints           (ints),
376         .ints_data      (ints_data),
377
378         // Output master port 1
379         .w_PADDR        (w_PADDR  [`APB_WIDTH*i +: `APB_WIDTH] ),
380         .w_PWRITE       (w_PWRITE [i] ),
381         .w_PSELx        (w_PSELx  [i] ),
382         .w_PENABLE      (w_PENABLE [i] ),
383         .w_PWDATA       (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
384         .w_PRDATA       (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
385         .w_PREADY       (w_PREADY  [i] )
386     )
387 `ifndef DEF_CORE_HAS_INSTR_MEM
388     // APB instruction rom
389     , // Output master port 2
390     .w2_PADDR        (instr_w_PADDR  [`APB_WIDTH*i +: `APB_WIDTH] ),
391     // .w2_PWRITE      (instr_w_PWRITE [i] ),
392     .w2_PSELx        (instr_w_PSELx  [i] ),
393     .w2_PENABLE      (instr_w_PENABLE [i] ),
394     // .w2_PWDATA      (instr_w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
395     .w2_PRDATA       (instr_w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
396     .w2_PREADY       (instr_w_PREADY  [i] )
397 `endif
398 );
399 end
400 endgenerate
401
402
403 ///////////////////////////////////////////////////
404 // Formal Verification
405 ///////////////////////////////////////////////////
406 `ifndef FORMAL
407 wire all_halted = &w_halt;
408 ///////////////////////////////////////////////////
409 // Count number of clocks each core is spending on
410 // bus transactions
411 ///////////////////////////////////////////////////
412 reg [15:0] bus_core_times    [0:`CORES-1];
413 reg [15:0] core_work_times   [0:`CORES-1];
414 reg [15:0] instr_fetch_times [0:`CORES-1];
415 integer i2;
416 initial
417     for(i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
418         bus_core_times[i2] = 0;
419         core_work_times[i2] = 0;
420     end
421
422 // total bus time
423 generate
424     genvar g2;
425     for (g2 = 0; g2 < `CORES; g2 = g2 + 1) begin : formal_for_times
426         always @(posedge clk) begin
427             if (w_PSELx[g2])
428                 bus_core_times[g2] <= bus_core_times[g2] + 1;
429
430             // Core working time
431             `ifndef DEF_CORE_HAS_INSTR_MEM
432                 if (!w_PSELx[g2] && !instr_w_PSELx[g2])
433                     if (!w_PSELx[g2])
434                         if (!w_halt[g2])
435                             core_work_times[g2] <= core_work_times[g2] + 1;
436             end
437         end
438     end
439 endgenerate
440
441 endgenerate
442

```

```

443 reg [15:0] bus_time_average = 0;
444 reg [15:0] bus_reqs_average = 0;
445 reg [15:0] fetch_time_average = 0;
446 reg [15:0] work_time_average = 0;
447 //
448 always @(all_halted) begin
449     for (i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
450         bus_time_average = bus_time_average + bus_core_times[i2];
451         bus_reqs_average = bus_reqs_average + bus_core_reqs_count[i2];
452         work_time_average = work_time_average + core_work_times[i2];
453         fetch_time_average = fetch_time_average + instr_fetch_times[i2];
454     end
455
456     bus_time_average = bus_time_average / `CORES;
457     bus_reqs_average = bus_reqs_average / `CORES;
458     work_time_average = work_time_average / `CORES;
459     fetch_time_average = fetch_time_average / `CORES;
460 end
461
462 //////////////////////////////////////////////////
463 // Count number of bus requests per core
464 //////////////////////////////////////////////////
465 // 1 clock delay of w_PSELx
466 reg [`CORES-1:0] bus_core_reqs_last;
467 // rising edges of each
468 wire [`CORES-1:0] bus_core_reqs_real;
469 // storage for counters for each core
470 reg [15:0] bus_core_reqs_count [0:`CORES-1];
471 initial
472     for(i2 = 0; i2 < `CORES; i2 = i2 + 1)
473         bus_core_reqs_count[i2] = 0;
474
475 // 1 clk delay to detect rising edge
476 always @(posedge clk)
477     bus_core_reqs_last <= w_PSELx;
478
479 generate
480     genvar g3;
481     for (g3 = 0; g3 < `CORES; g3 = g3 + 1) begin : formal_for_reqs
482         // Detect new reqs for each core
483         assign bus_core_reqs_real[g3] = w_PSELx[g3] >
484
485
486         always @(posedge clk)
487             if (bus_core_reqs_real[g3])
488                 bus_core_reqs_count[g3] <= bus_core_reqs_count[g3] + 1;
489     end
490 endgenerate
491
492
493 `ifndef DEF_CORE_HAS_INSTR_MEM
494     //////////////////////////////////////////////////
495     // Time waiting for instruction fetches
496     // from global memory
497     //////////////////////////////////////////////////
498     integer i3;
499     initial
500         for(i3 = 0; i3 < `CORES; i3 = i3 + 1)
501             instr_fetch_times[i3] = 0;
502
503     // total bus time
504     // Instruction fetches occur on the w2 master port
505     generate
506         genvar g4;
507         for (g4 = 0; g4 < `CORES; g4 = g4 + 1) begin : formal_for_fetch_times
508             always @(posedge clk)
509                 if (instr_w_PSELx[g4])
510                     instr_fetch_times[g4] <= instr_fetch_times[g4] + 1;
511             end
512         endgenerate
513     `endif
514 `endif
515
516 `endif // end FORMAL
517
518 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 `include "formal.v"

```



```

5
6 // Simple watchdog peripheral
7 module vmicro16_watchdog_apb # (
8     parameter BUS_WIDTH = 16,
9     parameter NAME       = "WD",
10    parameter CLK_HZ      = 50_000_000
11 ) (
12     input clk,
13     input reset,
14
15     // APB Slave to master interface
16     input [0:0]          S_PADDR, // not used (optimised out)
17     input                S_PWRITE,
18     input                S_PSELx,
19     input                S_PENABLE,
20     input [0:0]          S_PWDATA,
21
22     // prdata not used
23     output [0:0]         S_PRDATA,
24     output               S_PREADY,
25
26     // watchdog reset, active high
27     output reg           wdreset
28 );
29 //assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
30 assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
31 wire we        = (S_PSELx & S_PENABLE & S_PWRITE);
32
33 // countdown timer
34 reg [`clog2(CLK_HZ)-1:0] timer = CLK_HZ;
35
36 wire w_wdreset = (timer == 0);
37
38 // infer a register to aid timing
39 initial wdreset = 0;
40 always @(posedge clk)
41     wdreset <= w_wdreset;
42
43 always @(posedge clk)
44     if (we) begin
45         $display($time, "\t\t%s <= RESET", NAME);
46         timer <= CLK_HZ;
47     end else begin
48         timer <= timer - 1;
49     end
50 endmodule
51
52 module timer_apb # (
53     parameter CLK_HZ = 50_000_000
54 ) (
55     input clk,
56     input reset,
57     input clk_en,
58
59     // 0 16-bit value R/W
60     // 1 16-bit control R b0 = start, b1 = reset
61     // 2 16-bit prescaler
62     input [1:0]          S_PADDR,
63
64     input                S_PWRITE,
65     input                S_PSELx,
66     input                S_PENABLE,
67     input [DATA_WIDTH-1:0] S_PWDATA,
68
69     output reg [DATA_WIDTH-1:0] S_PRDATA,
70     output               S_PREADY,
71
72     output out,
73     output [DATA_WIDTH-1:0] int_data
74 );
75 //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hFOFO : 16'h0000;
76 assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
77 wire en        = (S_PSELx & S_PENABLE);
78 wire we        = (en & S_PWRITE);
79
80 reg [DATA_WIDTH-1:0] r_counter = 0;
81 reg [DATA_WIDTH-1:0] r_load = 0;
82 reg [DATA_WIDTH-1:0] r_pres = 0;
83 reg [DATA_WIDTH-1:0] r_ctrl = 0;
84
85 localparam CTRL_START = 0;
86 localparam CTRL_RESET = 1;
87 localparam CTRL_INT   = 2;
88
89 localparam ADDR_LOAD = 2'b00;
90 localparam ADDR_CTRL = 2'b01;
91 localparam ADDR_PRE = 2'b10;
92
93 always @(*) begin
94     S_PRDATA = 0;
95

```

```

97         if (en)
98             case(S_PADDR)
99                 ADDR_LOAD: S_PRDATA = r_counter;
100                 ADDR_CTRL: S_PRDATA = r_ctrl;
101                 //ADDR_CTRL: S_PRDATA = r_pres;
102                 default:    S_PRDATA = 0;
103             endcase
104         end
105         // prescaler counts from r_pres to 0, emitting a stb signal
106         // to enable the r_counter step
107         reg [`DATA_WIDTH-1:0] r_pres_counter = 0;
108         wire counter_en = (r_pres_counter == 0);
109         always @(posedge clk)
110             if (r_pres_counter == 0)
111                 r_pres_counter <= r_pres;
112             else
113                 r_pres_counter <= r_pres_counter - 1;
114
115         always @(posedge clk)
116             if (we)
117                 case(S_PADDR)
118                     // Write to the load register:
119                     // Set load register
120                     // Set counter register
121                     ADDR_LOAD: begin
122                         r_load      <= S_PWDATA;
123                         r_counter   <= S_PWDATA;
124                         $display($time, "\t\ttimr0: WRITE LOAD: %h", S_PWDATA);
125                     end
126                     ADDR_CTRL: begin
127                         r_ctrl      <= S_PWDATA;
128                         $display($time, "\t\ttimr0: WRITE CTRL: %h", S_PWDATA);
129                     end
130                     ADDRPres: begin
131                         r_pres      <= S_PWDATA;
132                         $display($time, "\t\ttimr0: WRITE PRES: %h", S_PWDATA);
133                     end
134                 endcase
135             else
136                 if (r_ctrl[CTRL_START]) begin
137                     if (r_counter == 0)
138                         r_counter <= r_load;
139                     else if(counter_en)
140                         r_counter <= r_counter -1;
141                     end else if (r_ctrl[CTRL_RESET])
142                         r_counter <= r_load;
143
144                 // generate the output pulse when r_counter == 0
145                 // out = (counter reached zero && counter started)
146                 assign out     = (r_counter == 0) && r_ctrl[CTRL_START]; // && r_ctrl[CTRL_INT];
147                 assign int_data = {`DATA_WIDTH{1'b1}};
148             endmodule
149
150         // APB wrapped vmicro16_bram
151         module vmicro16_bram_apb # (
152             parameter BUS_WIDTH   = 16,
153             parameter MEM_WIDTH   = 16,
154             parameter MEM_DEPTH   = 64,
155             parameter APB_PADDR   = 0,
156             parameter USE_INITS   = 0,
157             parameter NAME        = "BRAM",
158             parameter CORE_ID     = 0
159         ) (
160             input clk,
161             input reset,
162             // APB Slave to master interface
163             input  [`clog2(MEM_DEPTH)-1:0] S_PADDR,
164             input                               S_PWRITE,
165             input                               S_PSELx,
166             input                               S_PENABLE,
167             input [BUS_WIDTH-1:0]            S_PWDATA,
168
169             output [BUS_WIDTH-1:0]           S_PRDATA,
170             output                               S_PREADY
171         );
172         wire [MEM_WIDTH-1:0] mem_out;
173
174         assign S_PRDATA = (S_PSELx & S_PENABLE) ? mem_out : 16'h0000;
175         assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
176         assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
177
178         always @(*)
179             if (S_PSELx && S_PENABLE)
180                 $display($time, "\t\t%s => %h", NAME, mem_out);
181
182         always @(posedge clk)
183             if (we)
184                 $display($time, "\t\t%s[%h] <= %h", NAME,
185                     S_PADDR, S_PWDATA);
186

```

```

187     vmicro16_bram # (
188         .MEM_WIDTH  (MEM_WIDTH),
189         .MEM_DEPTH  (MEM_DEPTH),
190         .NAME        (NAME),
191         .USE_INITS   (1),
192         .CORE_ID     (-1)
193     ) bram_apb (
194         .clk         (clk),
195         .reset       (reset),
196
197         .mem_addr    (S_PADDR),
198         .mem_in      (S_PWDATA),
199         .mem_we      (we),
200         .mem_out     (mem_out)
201     );
202 endmodule
203
204 // Shared memory with hardware monitor (LWEX/SWEX)
205 module vmicro16_bram_ex_apb # (
206     parameter BUS_WIDTH  = 16,
207     parameter MEM_WIDTH  = 16,
208     parameter MEM_DEPTH  = 64,
209     parameter CORE_ID_BITS = 3,
210     parameter SWEX_SUCCESS = 16'h0000,
211     parameter SWEX_FAIL   = 16'h0001
212 ) (
213     input clk,
214     input reset,
215
216     // /19 /18 /16 /15 0/
217     // | LWEX | SWEX | 3 bit CORE_ID | S_PADDR |
218     input  [`APB_WIDTH-1:0] S_PADDR,
219
220     input S_PWRITE,
221     input S_PSELx,
222     input S_PENABLE,
223     input [MEM_WIDTH-1:0] S_PWDATA,
224
225     output reg [MEM_WIDTH-1:0] S_PRDATA,
226     output S_PREADY
227 );
228 // exclusive flag checks
229 wire [MEM_WIDTH-1:0] mem_out;
230 reg swex_success = 0;
231
232 localparam ADDR_BITS = `clog2(MEM_DEPTH);
233
234 // hack to create a 1 clock delay to S_PREADY
235 // for bram to be ready
236 reg cdelay = 1;
237 always @(posedge clk)
238     if (S_PSELx)
239         cdelay <= 0;
240     else
241         cdelay <= 1;
242
243 //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hFOFO : 16'h0000;
244 assign S_PREADY = (S_PSELx & S_PENABLE & (!cdelay)) ? 1'b1 : 1'b0;
245 assign we = (S_PSELx & S_PENABLE & S_PWRITE);
246 wire en = (S_PSELx & S_PENABLE);
247
248 // Similar to:
249 // http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204f/Cihbghef.html
250
251 // mem_wd is the CORE_ID sent in bits [18:16]
252 localparam TOP_BIT_INDEX = `APB_WIDTH - 1;
253 localparam PADDR_CORE_ID_MSB = TOP_BIT_INDEX - 2;
254 localparam PADDR_CORE_ID_LSB = PADDR_CORE_ID_MSB - (CORE_ID_BITS-1);
255
256 // [LWEX, CORE_ID, mem_addr] from S_PADDR
257 wire lwex = S_PADDR[TOP_BIT_INDEX];
258 wire swex = S_PADDR[TOP_BIT_INDEX-1];
259 wire [CORE_ID_BITS-1:0] core_id = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
260 // CORE_ID to write to ex_flags register
261 wire [ADDR_BITS-1:0] mem_addr = S_PADDR[ADDR_BITS-1:0];
262
263 wire [CORE_ID_BITS:0] ex_flags_read;
264 wire is_locked = |ex_flags_read;
265 wire is_locked_self = is_locked && (core_id == (ex_flags_read-1));
266
267 // Check exclusive access flags
268 always @(*) begin
269     swex_success = 0;
270     if (en)
271         // bug!
272         if (!swex && !lwex)
273             swex_success = 1;
274         else if (swex)
275             if (is_locked && !is_locked_self)
276                 // someone else has locked it
277                 swex_success = 0;

```

```

278         else if (is_locked && is_locked_self)
279             swex_success = 1;
280     end
281
282     always @(*)
283         if (swex)
284             if (swex_success)
285                 S_PRDATA = SWEX_SUCCESS;
286             else
287                 S_PRDATA = SWEX_FAIL;
288             else
289                 S_PRDATA = mem_out;
290
291     wire reg_we = en && ((lwex && !is_locked)
292         || (swex && swex_success));
293
294     reg [CORE_ID_BITS:0] reg_wd;
295     always @(*) begin
296         reg_wd = {{CORE_ID_BITS}{1'b0}};
297
298         if (en)
299             // if wanting to lock the addr
300             if (lwex)
301                 // and not already locked
302                 if (!is_locked) begin
303                     reg_wd = (core_id + 1);
304                 end
305             else if (swex)
306                 if (is_locked && is_locked_self)
307                     reg_wd = {{CORE_ID_BITS}{1'b0}};
308     end
309
310     // Exclusive flag for each memory cell
311     vmicro16_bram # (
312         .MEM_WIDTH  (CORE_ID_BITS + 1),
313         .MEM_DEPTH  (MEM_DEPTH),
314         .USE_INITS  (0),
315         .NAME       ("rexram")
316     ) ram_exflags (
317         .clk        (clk),
318         .reset      (reset),
319
320         .mem_addr   (mem_addr),
321         .mem_in     (reg_wd),
322         .mem_we     (reg_we),
323         .mem_out    (ex_flags_read)
324     );
325
326     always @(*)
327         if (S_PSELx && S_PENABLE)
328             $display($time, "\t\tBRAMex[%h] READ %h\tCORE: %h", mem_addr, mem_out, S_PADDR[16 +: CORE_ID_BITS]);
329
330     always @(posedge clk)
331         if (we)
332             $display($time, "\t\tBRAMex[%h] WRITE %h\tCORE: %h", mem_addr, S_PWDATA, S_PADDR[16 +: CORE_ID_BITS]);
333
334     vmicro16_bram # (
335         .MEM_WIDTH  (MEM_WIDTH),
336         .MEM_DEPTH  (MEM_DEPTH),
337         .USE_INITS  (0),
338         .NAME       ("BRAMexinst")
339     ) bram_apb (
340         .clk        (clk),
341         .reset      (reset),
342
343         .mem_addr   (mem_addr),
344         .mem_in     (S_PWDATA),
345         .mem_we     (we && swex_success),
346         .mem_out    (mem_out)
347     );
348 endmodule
349
350 // Simple APB memory-mapped register set
351 module vmicro16_regs_apb # (
352     parameter BUS_WIDTH      = 16,
353     parameter DATA_WIDTH   = 16,
354     parameter CELL_DEPTH    = 8,
355     parameter PARAM_DEFAULTS_R0 = 0,
356     parameter PARAM_DEFAULTS_R1 = 0
357 ) (
358     input clk,
359     input reset,
360     // APB Slave to master interface
361     input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
362     input S_PWRITE,
363     input S_PSELx,
364     input S_PENABLE,
365     input [DATA_WIDTH-1:0] S_PWDATA,
366
367     output [DATA_WIDTH-1:0] S_PRDATA,
368     output S_PREADY

```

```

369 );
370 wire [DATA_WIDTH-1:0] rd1;
371
372 assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1 : 16'h0000;
373 assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
374 assign reg_we = (S_PSELx & S_PENABLE & S_PWRITE);
375
376 always @(*)
377     if (reg_we)
378         $display($time, "\t\tREGS_APB[%h] <= %h",
379                 S_PADDR, S_PWDATA);
380
381 always @(*)
382     rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))
383
384 vmicro16_regs # (
385     .CELL_DEPTH (CELL_DEPTH),
386     .CELL_WIDTH (DATA_WIDTH),
387     .PARAM_DEFAULTS_RO (PARAM_DEFAULTS_RO),
388     .PARAM_DEFAULTS_R1 (PARAM_DEFAULTS_R1)
389 ) regs_apb (
390     .clk (clk),
391     .reset (reset),
392     // port 1
393     .rs1 (S_PADDR),
394     .rd1 (rd1),
395     .we (reg_we),
396     .ws1 (S_PADDR),
397     .wd (S_PWDATA)
398     // port 2 unconnected
399     // .rs2 (),
400     // .rd2 ()
401 );
402 endmodule
403
404 // Simple GPIO write only peripheral
405 module vmicro16_gpio_apb # (
406     parameter BUS_WIDTH = 16,
407     parameter DATA_WIDTH = 16,
408     parameter PORTS = 8,
409     parameter NAME = "GPIO"
410 ) (
411     input clk,
412     input reset,
413     // APB Slave to master interface
414     input [0:0] S_PADDR, // not used (optimised out)
415     input S_PWRITE,
416     input S_PSELx,
417     input S_PENABLE,
418     input [DATA_WIDTH-1:0] S_PWDATA,
419
420     output [DATA_WIDTH-1:0] S_PRDATA,
421     output S_PREADY,
422     output reg [PORTS-1:0] gpio
423 );
424 assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
425 assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
426 assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);
427
428 always @(posedge clk)
429     if (reset)
430         gpio <= 0;
431     else if (ports_we) begin
432         $display($time, "\t\t%s <= %h", NAME, S_PWDATA[PORTS-1:0]);
433         gpio <= S_PWDATA[PORTS-1:0];
434     end
435 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         `ifndef TEST_SW
55             $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\verilog_memh.txt", mem);
56         `endif
57
58         `define TEST_ASM
59         `ifndef TEST_ASM
60             $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\asm.s.hex", mem);
61         `endif
62
63         //`define TEST_COND
64         `ifndef 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         `ifndef 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         `ifndef 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         `ifndef 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        `ifndef 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     `ifndef ALL_TEST
111     // Standard all test
112     // REGSO
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     // TIMO
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     // BRAMO
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     `ifndef TEST_BRAM
156     // 2 core BRAMO 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
164
165 always @(posedge clk) begin
166     // synchronous WRITE_FIRST (page 13)
167     if (mem_we) begin
168         mem[mem_addr] <= mem_in;
169         $display($time, "\t\t%s[%h] <= %h",
170             NAME, mem_addr, mem_in);
171     end else
172         mem_out <= mem[mem_addr];
173 end
174
175 // TODO: Reset impl = every clock while reset is asserted, clear each cell
176 // one at a time, mem[i++] <= 0
177 endmodule
178
179
180 module vmicro16_core_mmu # (
181     parameter MEM_WIDTH = 16,
182     parameter MEM_DEPTH = 64,
183
184     parameter CORE_ID = 3'h0,
185     parameter CORE_ID_BITS = `clog2(`CORES)
186 ) (
187     input clk,
188     input reset,
189
190     input req,
191     output busy,
192
193     // From core
194     input [MEM_WIDTH-1:0] mmu_addr,
195     input [MEM_WIDTH-1:0] mmu_in,
196     input mmu_we,

```



```

197     input                                mmu_lwex,
198     input                                mmu_swex,
199     output reg [MEM_WIDTH-1:0] mmu_out,
200
201     // interrupts
202     output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
203     output reg [`DEF_NUM_INT-1:0] ints_mask,
204
205     // TO APB interconnect
206     output reg [`APB_WIDTH-1:0] M_PADDR,
207     output reg M_PWRITE,
208     output reg M_PSELx,
209     output reg M_PENABLE,
210     output reg [MEM_WIDTH-1:0] M_PWDATA,
211     // from interconnect
212     input [MEM_WIDTH-1:0] M_PRDATA,
213     input M_PREADY
214 );
215 localparam MMU_STATE_T1 = 0;
216 localparam MMU_STATE_T2 = 1;
217 localparam MMU_STATE_T3 = 2;
218 reg [1:0] mmu_state = MMU_STATE_T1;
219
220 reg [MEM_WIDTH-1:0] per_out = 0;
221 wire [MEM_WIDTH-1:0] tim0_out;
222
223 assign busy = req || (mmu_state == MMU_STATE_T2);
224
225 // more luts than below but easier
226 wire tim0_en = (mmu_addr >= `DEF_MMU_TIMO_S)
227               && (mmu_addr <= `DEF_MMU_TIMO_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             `ifdef FIX_T3
315                 MMU_STATE_T2: begin
316                     M_PENABLE <= 1;
317
318                     if (M_PREADY == 1'b1) begin
319                         mmu_state <= MMU_STATE_T3;
320                     end
321                 end
322             end
323                 MMU_STATE_T3: begin
324                     // Slave has output a ready signal (finished)
325                     M_PENABLE <= 0;
326                     M_PADDR <= 0;
327                     M_PWDATA <= 0;
328                     M_PSELx <= 0;
329                     M_PWRITE <= 0;
330                     // Clock the peripheral output into a reg,
331                     // to output on the next clock cycle
332                     per_out <= M_PRDATA;
333
334                     mmu_state <= MMU_STATE_T1;
335                 end
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             end
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_BRAMO_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 TIMO 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        ("TIMO")
384     ) TIMO (
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\tl %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 module vmicro16_core # (
765     parameter DATA_WIDTH = 16,
766     parameter MEM_INSTR_DEPTH = 64,
767     parameter MEM_SCRATCH_DEPTH = 64,
768     parameter MEM_WIDTH = 16,
769     parameter CORE_ID = 3'h0
770 ) (
771     input clk,
772     input reset,
773     output [7:0] debug,
774     output halt,
775
776     // interrupt sources
777     input [ `DEF_NUM_INT-1:0 ] ints,
778     input [ `DEF_NUM_INT*DATA_WIDTH-1:0 ] ints_data,
779     output [ `DEF_NUM_INT-1:0 ] ints_ack,
780
781     // APB master to slave interface (apb_intercon)
782     output [ `APB_WIDTH-1:0 ] w_PADDR,
783     output w_PWRITE,
784     output w_PSELx,
785     output w_PENABLE,
786     output [DATA_WIDTH-1:0] w_PWDATA,
787     input [DATA_WIDTH-1:0] w_PRDATA,
788     input w_PREADY
789
790     `ifndef DEF_CORE_HAS_INSTR_MEM
791     , // APB master interface to slave instruction memory
792     output reg [ `APB_WIDTH-1:0 ] w2_PADDR,
793     output reg w2_PWRITE,
794     output reg w2_PSELx,
795     output reg w2_PENABLE,
796     output reg [DATA_WIDTH-1:0] w2_PWDATA,
797     input [DATA_WIDTH-1:0] w2_PRDATA,
798     input w2_PREADY
799     `endif
800 );
801     localparam STATE_IF = 0;
802     localparam STATE_R1 = 1;
803     localparam STATE_R2 = 2;
804     localparam STATE_ME = 3;
805     localparam STATE_WB = 4;
806     localparam STATE_FE = 5;
807     localparam STATE_IDLE = 6;
808     localparam STATE_HALT = 7;
809     reg [2:0] r_state = STATE_IF;
810
811     reg [DATA_WIDTH-1:0] r_pc = 16'h0000;
812     reg [DATA_WIDTH-1:0] r_pc_saved = 16'h0000;
813     reg [DATA_WIDTH-1:0] r_instr = 16'h0000;
814     wire [DATA_WIDTH-1:0] w_mem_instr_out;
815     wire w_halt;
816
817     assign debug = {7'h00, w_halt};
818     assign halt = w_halt;
819
820     wire [4:0] r_instr_opcode;
821     wire [4:0] r_instr_alu_op;
822     wire [2:0] r_instr_rsd;
823     wire [2:0] r_instr_rsa;
824     reg [DATA_WIDTH-1:0] r_instr_rdd = 0;
825     reg [DATA_WIDTH-1:0] r_instr_rda = 0;
826     wire [3:0] r_instr_imm4;
827     wire [7:0] r_instr_imm8;
828     wire [4:0] r_instr_simm5;
829     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             end
923             // Restore state
924             r_pc <= r_pc_saved;

```



```

925         regs_use_int    <= 0;
926         int_pending_ack <= 0;
927     end else
928     `endif
929     if (w_branching) begin
930         $display($time, "\tC%02h: branching to %h", CORE_ID, r_instr_rdd);
931         r_pc    <= r_instr_rdd;
932
933         `ifdef DEF_ENABLE_INT
934             int_pending_ack <= 0;
935         `endif
936     end else if (r_pc < (MEM_INSTR_DEPTH-1)) begin
937         // normal increment
938         // pc <= pc + 1
939         r_pc    <= r_pc + 1;
940
941         `ifdef DEF_ENABLE_INT
942             int_pending_ack <= 0;
943         `endif
944     end
945 end // end r_state == STATE_WB
946 else if (r_state == STATE_HALT) begin
947     `ifdef DEF_ENABLE_INT
948         // Only an interrupt can return from halt
949         // duplicate code form STATE_ME!
950         if (int_pending) begin
951             $display($time, "\tC%02h: Jumping to ISR: %h", CORE_ID, ints_vector[0 +: `DATA_WIDTH]);
952             // TODO: check bounds
953             // Save state
954             r_pc_saved    <= r_pc; // + 1; HALT = stay with same PC
955             regs_use_int  <= 1;
956             int_pending_ack <= 1;
957             // Jump to ISR
958             r_pc    <= ints_vector[0 +: `DATA_WIDTH];
959         end else if (w_intr) begin
960             $display($time, "\tC%02h: Returning from ISR: %h", CORE_ID, r_pc_saved);
961             r_pc    <= r_pc_saved;
962             regs_use_int <= 0;
963             int_pending_ack <= 0;
964         end
965     `endif
966 end
967
968 `ifndef DEF_CORE_HAS_INSTR_MEM
969     initial w2_PSELx    = 0;
970     initial w2_PENABLE = 0;
971     initial w2_PADDR    = 0;
972 `endif
973
974 // cpu state machine
975 always @(posedge clk)
976     if (reset) begin
977         r_state    <= STATE_IF;
978         r_instr    <= 0;
979         r_mem_scratch_req <= 0;
980         r_instr_rdd <= 0;
981         r_instr_rda <= 0;
982     end
983     else begin
984
985     `ifndef DEF_CORE_HAS_INSTR_MEM
986         if (r_state == STATE_IF) begin
987             r_instr <= w_mem_instr_out;
988
989             $display("");
990             $display($time, "\tC%02h: PC: %h", CORE_ID, r_pc);
991             $display($time, "\tC%02h: INSTR: %h", CORE_ID, w_mem_instr_out);
992
993             r_state <= STATE_R1;
994         end
995     `else
996         // wait for global instruction rom to give us our instruction
997         if (r_state == STATE_IF) begin
998             // wait for ready signal
999             if (!w2_PREADY) begin
1000                 w2_PSELx    <= 1;
1001                 w2_PWRITE    <= 0;
1002                 w2_PENABLE    <= 1;
1003                 w2_PWDATA    <= 0;
1004                 w2_PADDR    <= r_pc;
1005             end else begin
1006                 w2_PSELx    <= 0;
1007                 w2_PWRITE    <= 0;
1008                 w2_PENABLE    <= 0;
1009                 w2_PWDATA    <= 0;
1010
1011                 r_instr <= w2_PRDATA;
1012
1013                 $display("");
1014                 $display($time, "\tC%02h: PC: %h", CORE_ID, r_pc);
1015                 $display($time, "\tC%02h: INSTR: %h", CORE_ID, w2_PRDATA);

```



```

1016
1017         r_state <= STATE_R1;
1018     end
1019 end
1020 `endif
1021
1022     else if (r_state == STATE_R1) begin
1023         if (w_halt) begin
1024             $display("");
1025             $display("");
1026             $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
1027             r_state <= STATE_HALT;
1028         end else begin
1029             // primary operand
1030             r_instr_rdd <= r_reg_rd1;
1031             r_state <= STATE_R2;
1032         end
1033     end
1034     else if (r_state == STATE_R2) begin
1035         // Choose secondary operand (register or immediate)
1036         if (r_instr_has_imm8) r_instr_rda <= r_instr_imm8;
1037         else if (r_instr_has_imm4) r_instr_rda <= r_reg_rd1 + r_instr_imm4;
1038         else r_instr_rda <= r_reg_rd1;
1039
1040         if (r_instr_has_mem) begin
1041             r_state <= STATE_ME;
1042             // Pulse req
1043             r_mem_scratch_req <= 1;
1044         end else
1045             r_state <= STATE_WB;
1046     end
1047     else if (r_state == STATE_ME) begin
1048         // Pulse req
1049         r_mem_scratch_req <= 0;
1050         // Wait for MMU to finish
1051         if (!r_mem_scratch_busy)
1052             r_state <= STATE_WB;
1053     end
1054     else if (r_state == STATE_WB) begin
1055         if (r_instr_has_cmp) begin
1056             $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
1057             r_cmp_flags <= r_alu_out[3:0];
1058         end
1059         r_state <= STATE_FE;
1060     end
1061     else if (r_state == STATE_FE)
1062         r_state <= STATE_IF;
1063     else if (r_state == STATE_HALT) begin
1064         `ifdef DEF_ENABLE_INT
1065             if (int_pending) begin
1066                 r_state <= STATE_FE;
1067             end
1068         end
1069     end
1070 end
1071 end
1072
1073 `ifndef DEF_CORE_HAS_INSTR_MEM
1074 // Instruction ROM
1075 (* rom_style = "distributed" *)
1076 vmicro16_bram # (
1077     .MEM_WIDTH      (DATA_WIDTH),
1078     .MEM_DEPTH      (MEM_INSTR_DEPTH),
1079     .CORE_ID        (CORE_ID),
1080     .USE_INITS      (1),
1081     .NAME            ("INSTR_MEM")
1082 ) mem_instr (
1083     .clk             (clk),
1084     .reset           (reset),
1085     // port 1
1086     .mem_addr        (r_pc),
1087     .mem_in          (0),
1088     .mem_we          (1'b0), // ROM
1089     .mem_out         (w_mem_instr_out)
1090 );
1091 `endif
1092
1093 // MMU
1094 vmicro16_core_mmu # (
1095     .MEM_WIDTH      (DATA_WIDTH),
1096     .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
1097     .CORE_ID        (CORE_ID)
1098 ) mmu (
1099     .clk             (clk),
1100     .reset           (reset),
1101     .req             (r_mem_scratch_req),
1102     .busy            (r_mem_scratch_busy),
1103     // interrupts
1104     .ints_vector     (ints_vector),
1105     .ints_mask       (ints_mask),
1106     // port 1

```

```

1107     .mmu_addr      (r_mem_scratch_addr),
1108     .mmu_in       (r_mem_scratch_in),
1109     .mmu_we       (r_mem_scratch_we),
1110     .mmu_lwex     (r_instr_has_lwex),
1111     .mmu_swex     (r_instr_has_swex),
1112     .mmu_out      (r_mem_scratch_out),
1113     // APB maste  r to slave
1114     .M_PADDR      (w_PADDR),
1115     .M_PWRITE     (w_PWRITE),
1116     .M_PSELx      (w_PSELx),
1117     .M_PENABLE    (w_PENABLE),
1118     .M_PWDATA     (w_PWDATA),
1119     .M_PRDATA     (w_PRDATA),
1120     .M_PREADY     (w_PREADY)
1121 );
1122
1123 // Instruction decoder
1124 vmicro16_dec dec (
1125     // input
1126     .instr         (r_instr),
1127     // output async
1128     .opcode        (),
1129     .rd            (r_instr_rsd),
1130     .ra            (r_instr_rsa),
1131     .imm4          (r_instr_imm4),
1132     .imm8          (r_instr_imm8),
1133     .imm12         (),
1134     .simm5         (r_instr_simm5),
1135     .alu_op        (r_instr_alu_op),
1136     .has_imm4      (r_instr_has_imm4),
1137     .has_imm8      (r_instr_has_imm8),
1138     .has_we        (r_instr_has_we),
1139     .has_br        (r_instr_has_br),
1140     .has_cmp       (r_instr_has_cmp),
1141     .has_mem       (r_instr_has_mem),
1142     .has_mem_we    (r_instr_has_mem_we),
1143     .halt          (w_halt),
1144     .intr          (w_intr),
1145     .has_lwex      (r_instr_has_lwex),
1146     .has_swex      (r_instr_has_swex)
1147 );
1148
1149 // Software registers
1150 vmicro16_regs # (
1151     .CORE_ID      (CORE_ID),
1152     .CELL_WIDTH   (`DATA_WIDTH)
1153 ) regs (
1154     .clk          (clk),
1155     .reset        (reset),
1156     // async port 0
1157     .rs1          (r_reg_rs1),
1158     .rd1          (r_reg_rd1_s),
1159     // async port 1
1160     // .rs2        (),
1161     // .rd2        (),
1162     // write port
1163     .we           (r_reg_we && ~regs_use_int),
1164     .ws1          (r_instr_rsd),
1165     .wd           (r_reg_wd)
1166 );
1167
1168 // Interrupt replacement registers
1169 `ifndef DEF_ENABLE_INT
1170 vmicro16_regs # (
1171     .CORE_ID      (CORE_ID),
1172     .CELL_WIDTH   (`DATA_WIDTH),
1173     .DEBUG_NAME   ("REGSINT")
1174 ) regs_intr (
1175     .clk          (clk),
1176     .reset        (reset),
1177     // async port 0
1178     .rs1          (r_reg_rs1),
1179     .rd1          (r_reg_rd1_i),
1180     // async port 1
1181     // .rs2        (),
1182     // .rd2        (),
1183     // write port
1184     .we           (r_reg_we && regs_use_int),
1185     .ws1          (r_instr_rsd),
1186     .wd           (r_reg_wd)
1187 );
1188 `endif
1189
1190 // ALU
1191 vmicro16_alu # (
1192     .CORE_ID      (CORE_ID)
1193 ) alu (
1194     .op           (r_instr_alu_op),
1195     .a            (r_instr_rdd),
1196     .b            (r_instr_rda),
1197     .flags        (r_cmp_flags),

```

```
1198         // async output
1199         .c          (r_alu_out)
1200     );
1201
1202     branch branch_check (
1203         .flags      (r_cmp_flags),
1204         .cond       (r_instr_imm8),
1205         .en         (w_branch_en)
1206     );
1207
1208 endmodule
```