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

Name: Ben David Lancaster

Date: July 30, 2019

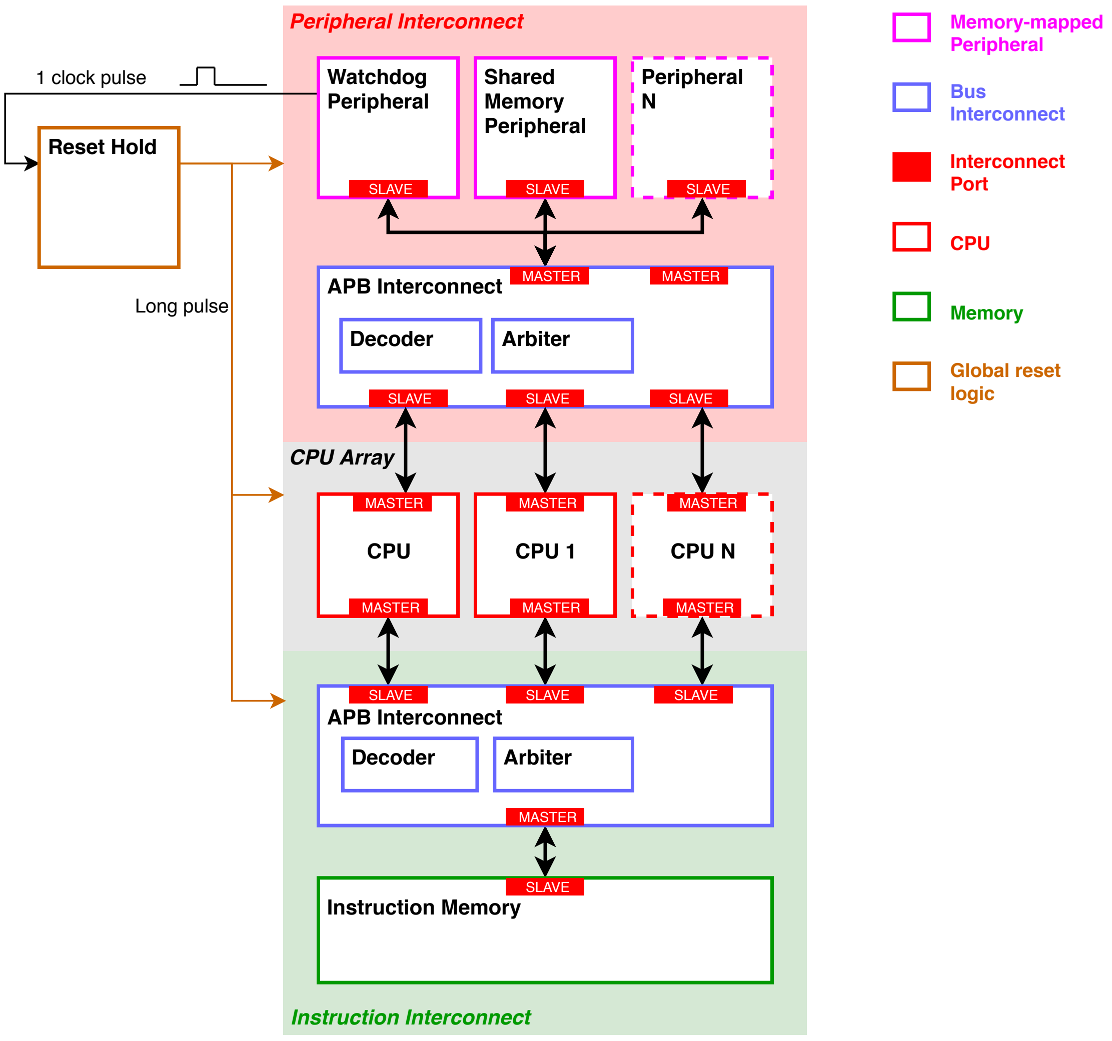
Table of Contents

Chapter 1

Interconnect

1.1 Introduction

1.2 Overview



1.2.1 Design Considerations

1.3 Peripheral Interconnect Interface

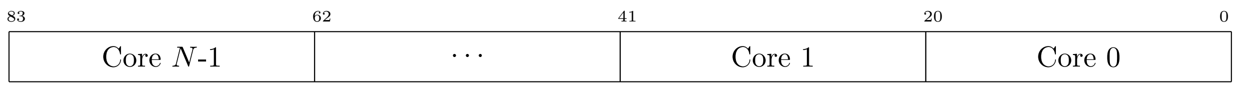
1.3.1 Master to Slave Interface

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

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

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

Chapter 2

Memory Mapping

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

2.1 Memory Map

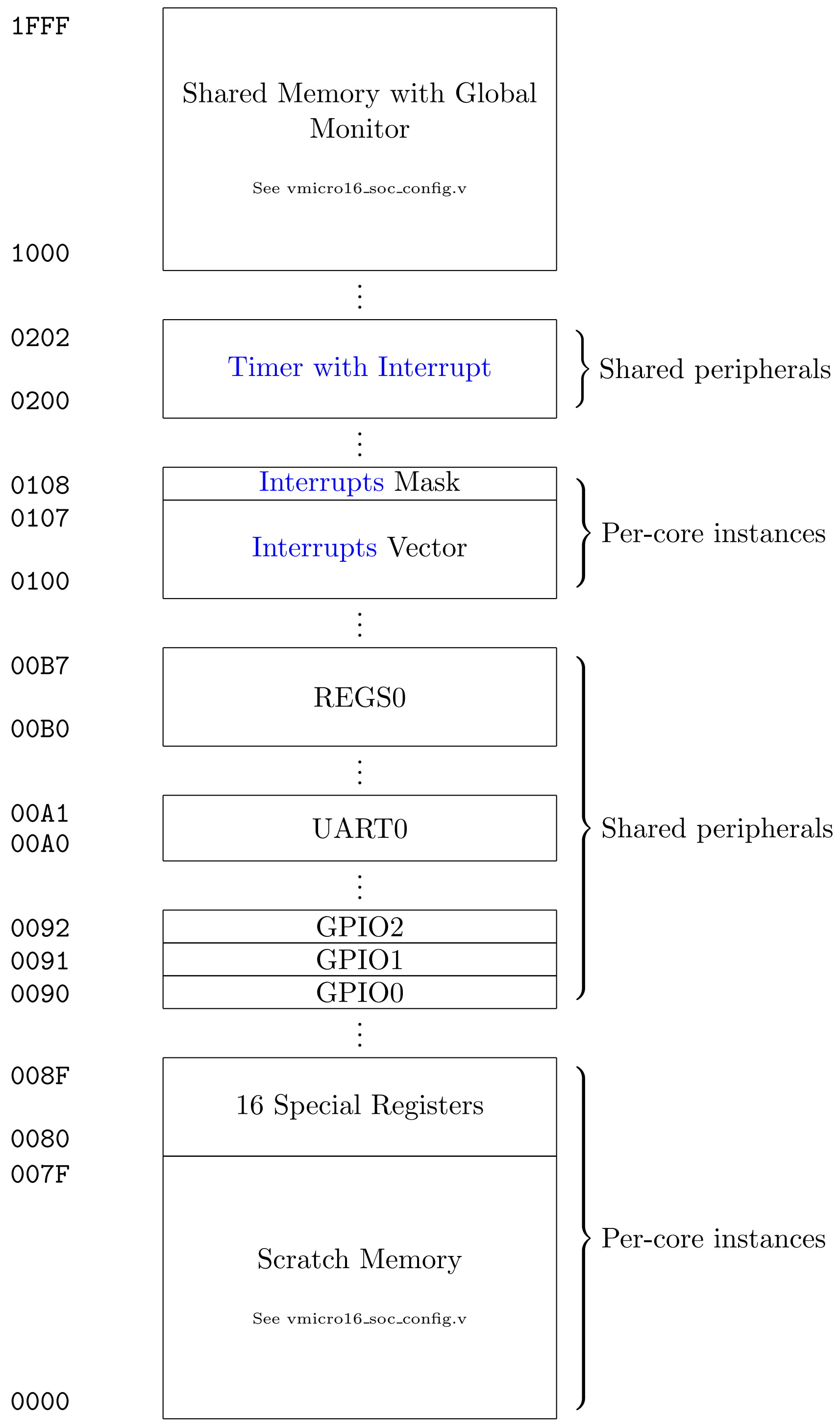


Figure 2.1: Memory map showing addresses of various memory sections.

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.

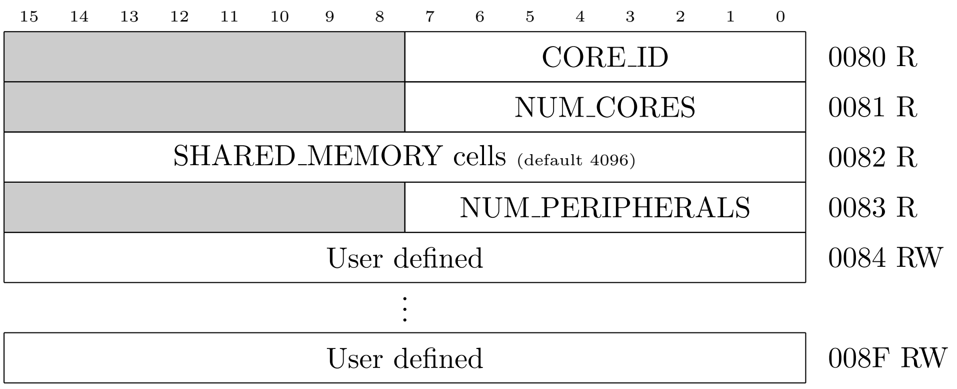


Figure 2.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

Chapter 3

Interrupts

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.

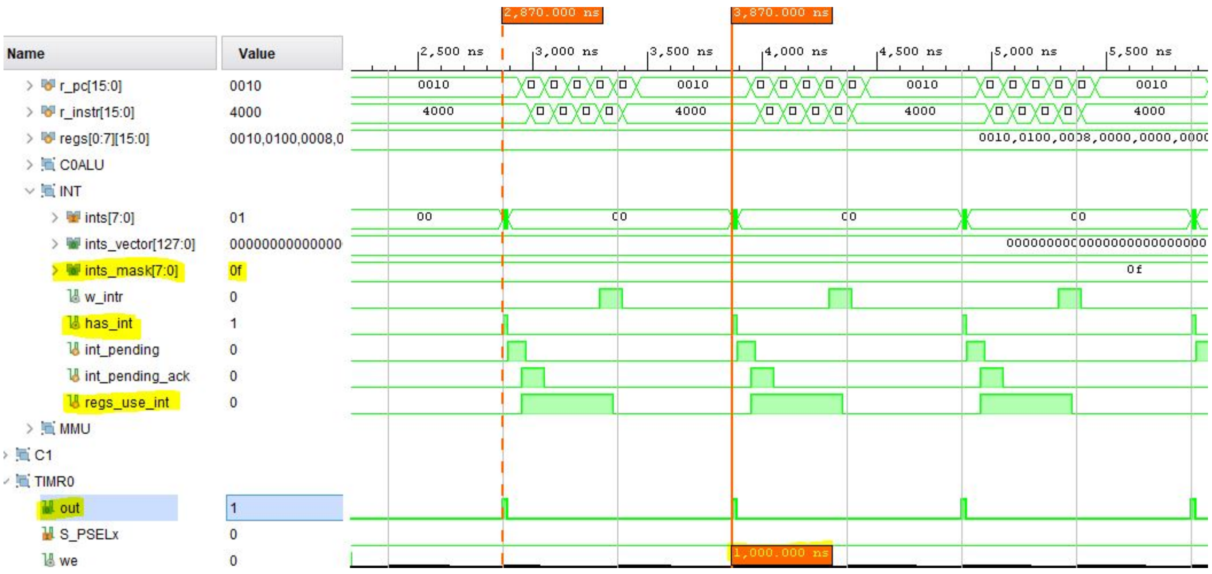


Figure 3.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return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

3.3 Software Interface

To enable software to

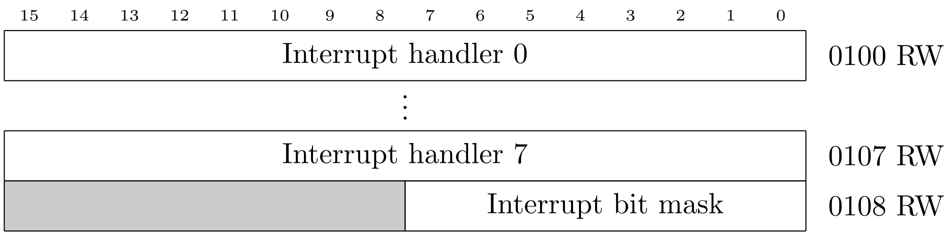


Figure 3.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

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,

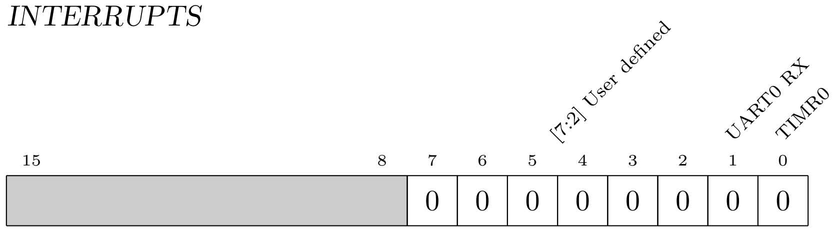


Figure 3.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

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.

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

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

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

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

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.

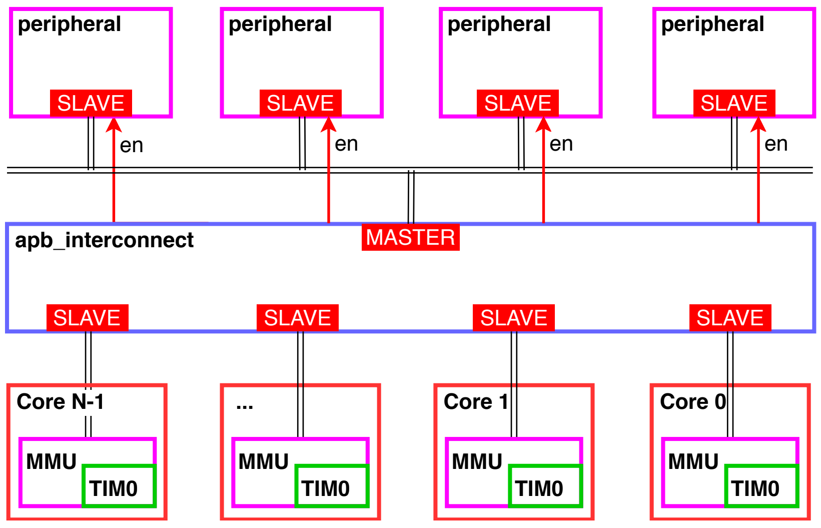
Chapter 4

Peripherals

4.1 Watchdog Timer

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

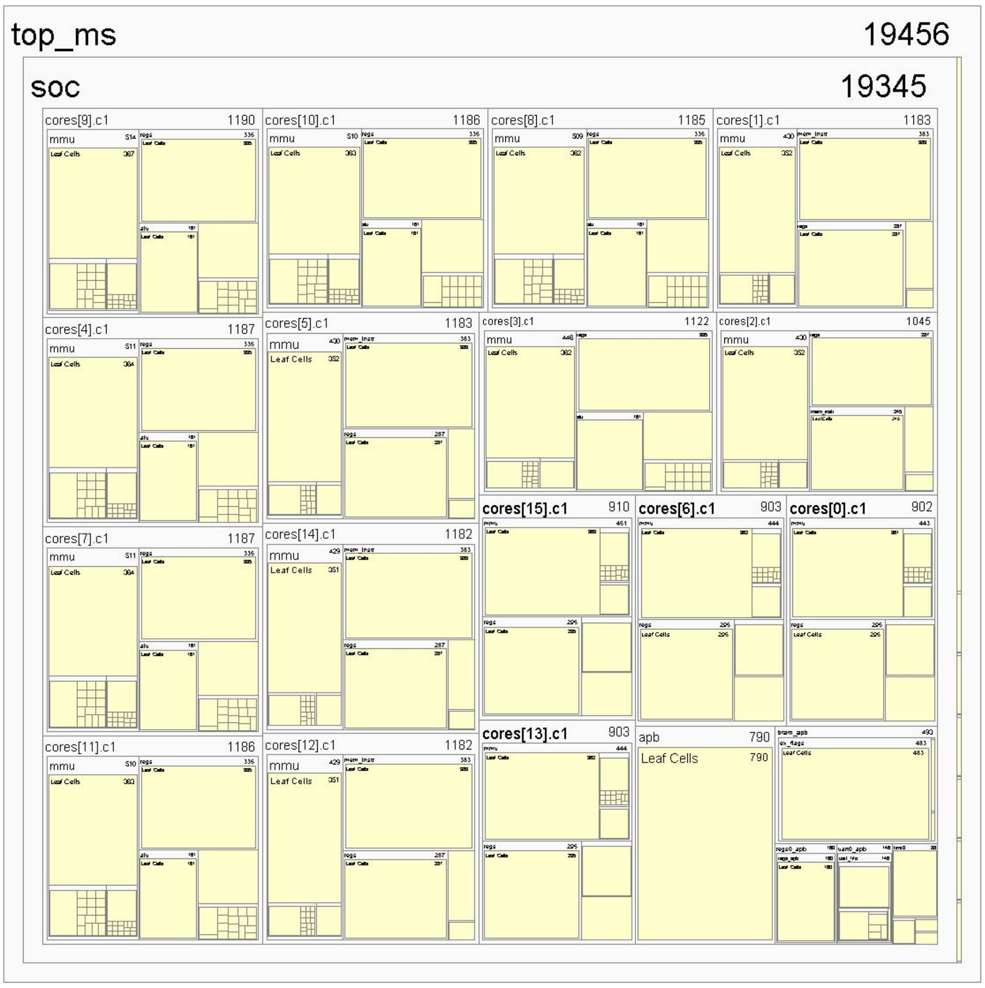


Figure 5.1:

Chapter 6

Analysis & Results

Chapter 7

Improvements

7.1 Foo

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

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

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

Chapter 8

Conclusion

8.1 Foo

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

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

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

Appendix A

Configuration Options

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

A.1 SoC Options

Macro	Default	Purpose
CORES	4	Number of CPU cores in the SoC
SLAVES	7	Number of peripherals

Table A.1: SoC Configuration Options

A.2 Core Options

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

Table A.2: Core Options

A.3 Peripheral Options

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

Table A.3: Peripheral Options

Appendix B

Code Listing

B.1 vmicro16_soc_config.v

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

```verilog
// Configuration defines for the vmicro16_soc and vmicro16 cpu.

`ifndef VMICRO16_SOC_CONFIG_H
`define VMICRO16_SOC_CONFIG_H

`include "clog2.v"

`define FORMAL

`define CORES           2
`define SLAVES          8

////////////////////////////////////////////////////////////
// Core parameters
///////////////////////////////////////////////////////////
// Per core instruction memory
//  Set this to give each core its own instruction memory cache
//`define DEF_CORE_HAS_INSTR_MEM

// Top level data width for registers, memory cells, bus widths
`define DATA_WIDTH      16

// Set this to use a workaround for the MMU's APB T2 clock
//`define FIX_T3

// Instruction memory (read only)
//   Must be large enough to support software program.
`ifdef DEF_CORE_HAS_INSTR_MEM
    // 4096 16-bit words global
    `define DEF_MEM_INSTR_DEPTH 4096
`else
    // 64 16-bit words per core
    `define DEF_MEM_INSTR_DEPTH 64
`endif

// Scratch memory (read/write) on each core.
//   See `DEF_MMU_TIM0_* defines for info.
`define DEF_MEM_SCRATCH_DEPTH 64

// Enables hardware multiplier and mult rr instruction
`define DEF_ALU_HW_MULT 1

// Enables global reset (requires more luts)
`define DEF_GLOBAL_RESET

// Enable a watch dog timer to reset the soc if threadlocked
```

```
`define DEF_USE_WATCHDOG

/////////////////////////////////////////////////////////
// Memory mapping
/////////////////////////////////////////////////////////
`define APB_WIDTH       (2 + `clog2(`CORES) + `DATA_WIDTH)

`define APB_PSELX_GPIO0 0
`define APB_PSELX_UART0 1
`define APB_PSELX_REGS0 2
`define APB_PSELX_BRAM0 3
`define APB_PSELX_GPIO1 4
`define APB_PSELX_GPIO2 5
`define APB_PSELX_TIMR0 6
`define APB_PSELX_WDOG0 7

`define APB_GPIO0_PINS  8
`define APB_GPIO1_PINS  16
`define APB_GPIO2_PINS  8

// Shared memory words
`define APB_BRAM0_CELLS 4096

/////////////////////////////////////////////////////////
// Memory mapping
/////////////////////////////////////////////////////////
// TIM0
// Number of scratch memory cells per core
`define DEF_MMU_TIM0_CELLS  64
`define DEF_MMU_TIM0_S      16'h0000
`define DEF_MMU_TIM0_E      16'h007F
// SREG
`define DEF_MMU_SREG_S      16'h0080
`define DEF_MMU_SREG_E      16'h008F
// GPIO0
`define DEF_MMU_GPIO0_S     16'h0090
`define DEF_MMU_GPIO0_E     16'h0090
// GPIO1
`define DEF_MMU_GPIO1_S     16'h0091
`define DEF_MMU_GPIO1_E     16'h0091
// GPIO2
`define DEF_MMU_GPIO2_S     16'h0092
`define DEF_MMU_GPIO2_E     16'h0092
// UART0
`define DEF_MMU_UART0_S     16'h00A0
`define DEF_MMU_UART0_E     16'h00A1
// REGS0
`define DEF_MMU_REGS0_S     16'h00B0
`define DEF_MMU_REGS0_E     16'h00B7
// WDOG0
`define DEF_MMU_WDOG0_S     16'h00B8
`define DEF_MMU_WDOG0_E     16'h00B8
// BRAM0
`define DEF_MMU_BRAM0_S     16'h1000
`define DEF_MMU_BRAM0_E     16'h1fff
// TIMR0
`define DEF_MMU_TIMR0_S     16'h0200
`define DEF_MMU_TIMR0_E     16'h0202

/////////////////////////////////////////////////////////
// Interrupts
/////////////////////////////////////////////////////////
// Enable/disable interrupts
//   Disabling will free up resources for other features
`define DEF_ENABLE_INT
// Number of interrupt in signals
`define DEF_NUM_INT     8
// Default interrupt bitmask (0 = hidden, 1 = enabled)
`define DEF_INT_MASK    0
// Bit position of the TIMR0 interrupt signal
`define DEF_INT_TIMR0   0
// Interrupt vector memory location
`define DEF_MMU_INTSV_S     16'h0100
`define DEF_MMU_INTSV_E     16'h0107
// Interrupt vector memory location
`define DEF_MMU_INTSM_S     16'h0108
`define DEF_MMU_INTSM_E     16'h0108


`endif
```

B.2 top_ms.v

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

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

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


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

    // SSDs
    output [6:0] ssd0,
    output [6:0] ssd1,
    output [6:0] ssd2,
    output [6:0] ssd3,
    output [6:0] ssd4,
    output [6:0] ssd5
);
    //wire [15:0]         M_PADDR;
    //wire                M_PWRITE;
    //wire [5-1:0]        M_PSELx;  // not shared
    //wire                M_PENABLE;
    //wire [15:0]         M_PWDATA;
    //wire [15:0]         M_PRDATA; // input to intercon
    //wire                M_PREADY; // input to intercon

    wire [7:0]  gpio0;
    wire [15:0] gpio1;
    wire [7:0]  gpio2;

    vmicro16_soc soc (
        .clk     (CLK50),
        .reset   (~SW[0]),

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

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

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

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

endmodule
```

B.3 vmicro16_soc.v

```
//
//

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

module pow_reset # (
    parameter INIT  = 1,
    parameter N     = 8
) (
    input       clk,
    input       reset,
    output reg  resethold
);
    initial resethold = INIT ? (N-1) : 0;

    always @(*)
        resethold = |hold;

    reg [`clog2(N)-1:0] hold = (N-1);
    always @(posedge clk)
        if (reset)
            hold <= N-1;
        else
            if (hold)
                hold <= hold - 1;
endmodule

// Vmicro16 multi-core SoC with various peripherals
// and interrupts
module vmicro16_soc (
    input clk,
    input reset,

    //input  uart_rx,
    output                          uart_tx,
    output [`APB_GPIO0_PINS-1:0]    gpio0,
    output [`APB_GPIO1_PINS-1:0]    gpio1,
    output [`APB_GPIO2_PINS-1:0]    gpio2,

    output                          halt,

    output      [`CORES-1:0]        dbug0,
    output      [`CORES*8-1:0]      dbug1
);
    wire [`CORES-1:0] w_halt;
    assign halt = &w_halt;

    assign dbug0 = w_halt;

    // Watchdog reset pulse signal.
    //   Passed to pow_reset to generate a longer reset pulse
    wire wdreset;

    // soft register reset hold for brams and registers
    wire soft_reset;
    `ifdef DEF_GLOBAL_RESET
        pow_reset # (
            .INIT       (1),
            .N          (8)
        ) por_inst (
            .clk        (clk),
            `ifdef DEF_USE_WATCHDOG
            .reset      (reset | wdreset),
            `else
            .reset      (reset),
            `endif
            .resethold  (soft_reset)
        );
    `else
        assign soft_reset = 0;
    `endif

    // Peripherals (master to slave)
     wire [`APB_WIDTH-1:0]          M_PADDR;
     wire                           M_PWRITE;
     wire [`SLAVES-1:0]             M_PSELx;  // not shared
     wire                          M_PENABLE;
     wire [`DATA_WIDTH-1:0]        M_PWDATA;
     wire [`SLAVES*`DATA_WIDTH-1:0] M_PRDATA; // input to intercon
     wire [`SLAVES-1:0]            M_PREADY; // input

    // Master apb interfaces
     wire [`CORES*`APB_WIDTH-1:0]   w_PADDR;
     wire [`CORES-1:0]              w_PWRITE;
     wire [`CORES-1:0]              w_PSELx;
     wire [`CORES-1:0]              w_PENABLE;
     wire [`CORES*`DATA_WIDTH-1:0]  w_PWDATA;
     wire [`CORES*`DATA_WIDTH-1:0]  w_PRDATA;
     wire [`CORES-1:0]              w_PREADY;

    // Interrupts
`ifdef DEF_ENABLE_INT
    wire [`DEF_NUM_INT-1:0]              ints;
    wire [`DEF_NUM_INT*`DATA_WIDTH-1:0]  ints_data;
    assign ints[7:1] = 0;
    assign ints_data[`DEF_NUM_INT*`DATA_WIDTH-1:`DATA_WIDTH] =
                {`DEF_NUM_INT*(`DATA_WIDTH-1){1'b0}};
`endif

    apb_intercon_s # (
        .MASTER_PORTS   (`CORES),
        .SLAVE_PORTS    (`SLAVES),
        .BUS_WIDTH      (`APB_WIDTH),
        .DATA_WIDTH     (`DATA_WIDTH),
        .HAS_PSELX_ADDR (1)
    ) apb (
        .clk        (clk),
        .reset      (soft_reset),
        // APB master to slave
        .S_PADDR    (w_PADDR),
        .S_PWRITE   (w_PWRITE),
        .S_PSELx    (w_PSELx),
        .S_PENABLE  (w_PENABLE),
        .S_PWDATA   (w_PWDATA),
        .S_PRDATA   (w_PRDATA),
        .S_PREADY   (w_PREADY),
        // shared bus
        .M_PADDR    (M_PADDR),
        .M_PWRITE   (M_PWRITE),
        .M_PSELx    (M_PSELx),
        .M_PENABLE  (M_PENABLE),
        .M_PWDATA   (M_PWDATA),
        .M_PRDATA   (M_PRDATA),
        .M_PREADY   (M_PREADY)
    );

`ifdef DEF_USE_WATCHDOG
    vmicro16_watchdog_apb # (
        .BUS_WIDTH  (`APB_WIDTH),
        .NAME       ("WDOGO")
    ) wdog0_apb (
        .clk        (clk),
        .reset      (),
        // apb slave to master interface
        .S_PADDR    (),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_WDOGO]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (),
        .S_PRDATA   (),
        .S_PREADY   (M_PREADY[`APB_PSELX_WDOGO]),

        .wdreset    (wdreset)
    );
`endif

    vmicro16_gpio_apb # (
        .BUS_WIDTH  (`APB_WIDTH),
        .DATA_WIDTH (`DATA_WIDTH),
        .PORTS      (`APB_GPIO0_PINS),
        .NAME       ("GPIO0")
    ) gpio0_apb (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_GPIO0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO0*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_GPIO0]),
        .gpio       (gpio0)
    );

    // GPIO1 for Seven segment displays (16 pin)
    vmicro16_gpio_apb # (
        .BUS_WIDTH  (`APB_WIDTH),
        .DATA_WIDTH (`DATA_WIDTH),
        .PORTS      (`APB_GPIO1_PINS),
        .NAME       ("GPIO1")
    ) gpio1_apb (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_GPIO1]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO1*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_GPIO1]),
        .gpio       (gpio1)
    );

    // GPIO2 for Seven segment displays (8 pin)
    vmicro16_gpio_apb # (
        .BUS_WIDTH  (`APB_WIDTH),
        .DATA_WIDTH (`DATA_WIDTH),
        .PORTS      (`APB_GPIO2_PINS),
        .NAME       ("GPIO2")
    ) gpio2_apb (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_GPIO2]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_GPIO2*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_GPIO2]),
        .gpio       (gpio2)
    );

    apb_uart_tx # (
        .DATA_WIDTH (8),
        .ADDR_EXP   (4) //2^^4 = 16 FIFO words
    ) uart0_apb (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_UART0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_UART0*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_UART0]),
        // uart wires
        .tx_wire    (uart_tx),
        .rx_wire    (uart_rx)
    );

    timer_apb timr0 (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_TIMR0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_TIMR0*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_TIMR0])
        //
        `ifdef DEF_ENABLE_INT
        ,.out       (ints      [`DEF_INT_TIMR0]),
         .int_data  (ints_data[`DEF_INT_TIMR0*`DATA_WIDTH +: `DATA_WIDTH])
        `endif
    );

    // Shared register set for system-on-chip info
    // R0 = number of cores
    vmicro16_regs_apb # (
        .BUS_WIDTH          (`APB_WIDTH),
        .DATA_WIDTH         (`DATA_WIDTH),
        .CELL_DEPTH         (8),
        .PARAM_DEFAULTS_R0  (`CORES),
        .PARAM_DEFAULTS_R1  (`SLAVES)
    ) regs0_apb (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_REGS0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_REGS0*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_REGS0])
    );

    vmicro16_bram_ex_apb # (
        .BUS_WIDTH    (`APB_WIDTH),
        .MEM_WIDTH    (`DATA_WIDTH),
        .MEM_DEPTH    (`APB_BRAM0_CELLS),
        .CORE_ID_BITS (`clog2(`CORES))
    ) bram_apb (
        .clk        (clk),
        .reset      (soft_reset),
        // apb slave to master interface
        .S_PADDR    (M_PADDR),
        .S_PWRITE   (M_PWRITE),
        .S_PSELx    (M_PSELx[`APB_PSELX_BRAM0]),
        .S_PENABLE  (M_PENABLE),
        .S_PWDATA   (M_PWDATA),
        .S_PRDATA   (M_PRDATA[`APB_PSELX_BRAM0*`DATA_WIDTH +: `DATA_WIDTH]),
        .S_PREADY   (M_PREADY[`APB_PSELX_BRAM0])
    );

    // There must be atleast 1 core
    `static_assert(`CORES > 0)
    `static_assert(`DEF_MEM_INSTR_DEPTH > 0)
    `static_assert(`DEF_MMU_TIM0_CELLS > 0)


    // Single instruction memory
`ifndef DEF_CORE_HAS_INSTR_MEM
    // slave input/outputs from interconnect
    wire [`APB_WIDTH-1:0]          instr_M_PADDR;
    wire                           instr_M_PWRITE;
    wire [1-1:0]                   instr_M_PSELx;  // not shared
    wire                           instr_M_PENABLE;
    wire [`DATA_WIDTH-1:0]         instr_M_PWDATA;
    wire [1*`DATA_WIDTH-1:0]       instr_M_PRDATA; // slave response
    wire [1-1:0]                   instr_M_PREADY; // slave response

    // Master apb interfaces
    wire [`CORES*`APB_WIDTH-1:0]   instr_w_PADDR;
    wire [`CORES-1:0]              instr_w_PWRITE;
    wire [`CORES-1:0]              instr_w_PSELx;
    wire [`CORES-1:0]              instr_w_PENABLE;
    wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PWDATA;
    wire [`CORES*`DATA_WIDTH-1:0]  instr_w_PRDATA;
    wire [`CORES-1:0]              instr_w_PREADY;

    vmicro16_bram_apb # (
        .BUS_WIDTH      (`APB_WIDTH),
        .MEM_WIDTH      (`DATA_WIDTH),
        .MEM_DEPTH      (`DEF_MEM_INSTR_DEPTH),
        .USE_INITS      (1),
        .NAME           ("INSTR_ROM_G")
    ) instr_rom_apb (
        .clk            (clk),
        .reset          (soft_reset),
        .S_PADDR        (instr_M_PADDR),
        .S_PWRITE       (),
        .S_PSELx        (instr_M_PSELx),
        .S_PENABLE      (instr_M_PENABLE),
        .S_PWDATA       (),
        .S_PRDATA       (instr_M_PRDATA),
        .S_PREADY       (instr_M_PREADY)
    );

    apb_intercon_s # (
        .MASTER_PORTS   (`CORES),
        .SLAVE_PORTS    (1),
        .BUS_WIDTH      (`APB_WIDTH),
        .DATA_WIDTH     (`DATA_WIDTH),
        .HAS_PSELX_ADDR (0)
    ) apb_instr_intercon (
        .clk        (clk),
        .reset      (soft_reset),
        // APB master from cores
        // master
        .S_PADDR    (instr_w_PADDR),
        .S_PWRITE   (instr_w_PWRITE),
        .S_PSELx    (instr_w_PSELx),
        .S_PENABLE  (instr_w_PENABLE),
        .S_PWDATA   (instr_w_PWDATA),
        .S_PRDATA   (instr_w_PRDATA),
        .S_PREADY   (instr_w_PREADY),
        // shared bus slaves
        // slave outputs
        .M_PADDR    (instr_M_PADDR),
        .M_PWRITE   (instr_M_PWRITE),
        .M_PSELx    (instr_M_PSELx),
        .M_PENABLE  (instr_M_PENABLE),
        .M_PWDATA   (instr_M_PWDATA),
        .M_PRDATA   (instr_M_PRDATA),
        .M_PREADY   (instr_M_PREADY)
    );
`endif

    genvar i;
    generate for(i = 0; i < `CORES; i = i + 1) begin : cores

        vmicro16_core # (
            .CORE_ID            (i),
            .DATA_WIDTH         (`DATA_WIDTH),

            .MEM_INSTR_DEPTH    (`DEF_MEM_INSTR_DEPTH),
            .MEM_SCRATCH_DEPTH  (`DEF_MMU_TIM0_CELLS)
        ) c1 (
            .clk        (clk),
            .reset      (soft_reset),

            // debug
            .halt       (w_halt[i]),

            // interrupts
            .ints       (ints),
            .ints_data  (ints_data),

            // Output master port 1
            .w_PADDR    (w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
            .w_PWRITE   (w_PWRITE  [i]                           ),
            .w_PSELx    (w_PSELx   [i]                           ),
            .w_PENABLE  (w_PENABLE [i]                           ),
            .w_PWDATA   (w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w_PRDATA   (w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w_PREADY   (w_PREADY  [i]                           )

`ifndef DEF_CORE_HAS_INSTR_MEM
            // APB instruction rom
            , // Output master port 2
            .w2_PADDR   (instr_w_PADDR   [`APB_WIDTH*i +: `APB_WIDTH]  ),
            //.w2_PWRITE  (instr_w_PWRITE  [i]                           ),
            .w2_PSELx   (instr_w_PSELx   [i]                           ),
            .w2_PENABLE (instr_w_PENABLE [i]                           ),
            //.w2_PWDATA  (instr_w_PWDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w2_PRDATA  (instr_w_PRDATA  [`DATA_WIDTH*i +: `DATA_WIDTH]),
            .w2_PREADY  (instr_w_PREADY  [i]                           )
`endif
        );
    end
    endgenerate


    ////////////////////////////////////////////////////
    // Formal Verification
    ////////////////////////////////////////////////////
    `ifdef FORMAL
    wire all_halted = &w_halt;
    ////////////////////////////////////////////////////
    // Count number of clocks each core is spending on
    //   bus transactions
    ////////////////////////////////////////////////////
    reg [15:0] bus_core_times       [0:`CORES-1];
    reg [15:0] core_work_times      [0:`CORES-1];
    reg [15:0] instr_fetch_times    [0:`CORES-1];
    integer i2;
    initial
        for(i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
            bus_core_times[i2] = 0;
            core_work_times[i2] = 0;
        end

    // total bus time
    generate
        genvar g2;
        for (g2 = 0; g2 < `CORES; g2 = g2 + 1) begin : formal_for_times
                        always @(posedge clk) begin
                                if (w_PSELx[g2])
                                        bus_core_times[g2] <= bus_core_times[g2] + 1;

                                // Core working time
                                `ifndef DEF_CORE_HAS_INSTR_MEM
                                        if (!w_PSELx[g2] && !instr_w_PSELx[g2])
                                `else
                                        if (!w_PSELx[g2])
                                `endif
                                                if (!w_halt[g2])
                                                        core_work_times[g2] <= core_work_times[g2] + 1;

                        end
                end
    endgenerate

    reg [15:0] bus_time_average = 0;
    reg [15:0] bus_reqs_average = 0;
    reg [15:0] fetch_time_average = 0;
    reg [15:0] work_time_average = 0;
    //
    always @(all_halted) begin
        for (i2 = 0; i2 < `CORES; i2 = i2 + 1) begin
            bus_time_average   = bus_time_average   + bus_core_times[i2];
            bus_reqs_average   = bus_reqs_average   + bus_core_reqs_count[i2];
            work_time_average  = work_time_average  + core_work_times[i2];
            fetch_time_average = fetch_time_average + instr_fetch_times[i2];
        end

        bus_time_average   = bus_time_average   / `CORES;
        bus_reqs_average   = bus_reqs_average   / `CORES;
        work_time_average  = work_time_average  / `CORES;
        fetch_time_average = fetch_time_average / `CORES;
    end

    ///////////////////////////////////////////////////
    // Count number of bus requests per core
    ///////////////////////////////////////////////////
    // 1 clock delay of w_PSELx
    reg [`CORES-1:0] bus_core_reqs_last;
    // rising edges of each
    wire [`CORES-1:0] bus_core_reqs_real;
    // storage for counters for each core
    reg [15:0] bus_core_reqs_count [0:`CORES-1];
    initial
        for(i2 = 0; i2 < `CORES; i2 = i2 + 1)
            bus_core_reqs_count[i2] = 0;

    // 1 clk delay to detect rising edge
    always @(posedge clk)
        bus_core_reqs_last <= w_PSELx;

    generate
        genvar g3;
                          for (g3 = 0; g3 < `CORES; g3 = g3 + 1) begin : formal_for_reqs
                          // Detect new reqs for each core
                          assign bus_core_reqs_real[g3] = w_PSELx[g3] >
                                                                                                                bus_core_rec

                          always @(posedge clk)
                                        if (bus_core_reqs_real[g3])
                                                 bus_core_reqs_count[g3] <= bus_core_reqs_count[g3] + 1;

                 end
    endgenerate


    `ifndef DEF_CORE_HAS_INSTR_MEM
        ///////////////////////////////////////////////////
        // Time waiting for instruction fetches
        //   from global  memory
        ///////////////////////////////////////////////////
        integer i3;
        initial
            for(i3 = 0; i3 < `CORES; i3 = i3 + 1)
                instr_fetch_times[i3] = 0;

        // total bus time
        // Instruction fetches occur on the w2 master port
        generate
            genvar g4;
            for (g4 = 0; g4 < `CORES; g4 = g4 + 1) begin : formal_for_fetch_times
                always @(posedge clk)
                    if (instr_w_PSELx[g4])
                        instr_fetch_times[g4] <= instr_fetch_times[g4] + 1;
            end
        endgenerate
    `endif


    `endif // end FORMAL

endmodule
```

B.4 vmicro16_periph.v

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

```
// Vmicro16 peripheral modules

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

```

// Simple watchdog peripheral
module vmicro16_watchdog_apb # (
    parameter BUS_WIDTH  = 16,
    parameter NAME       = "WD",
    parameter CLK_HZ     = 50_000_000
) (
    input clk,
    input reset,

    // APB Slave to master interface
    input  [0:0]                    S_PADDR, // not used (optimised out)
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [0:0]                    S_PWDATA,

    // prdata not used
    output [0:0]                    S_PRDATA,
    output                          S_PREADY,

    // watchdog reset, active high
    output reg                      wdreset
);
    //assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    wire   we       = (S_PSELx & S_PENABLE & S_PWRITE);

    // countdown timer
    reg [`clog2(CLK_HZ)-1:0] timer = CLK_HZ;

    wire w_wdreset = (timer == 0);

    // infer a register to aid timing
    initial wdreset = 0;
    always @(posedge clk)
        wdreset <= w_wdreset;

    always @(posedge clk)
        if (we) begin
            $display($time, "\t\%s <= RESET", NAME);
            timer <= CLK_HZ;
        end else begin
            timer <= timer - 1;
        end
endmodule

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

    input clk_en,

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

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

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

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

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

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

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

    always @(*) begin
        S_PRDATA = 0;
```

```verilog
        if (en)
            case(S_PADDR)
                ADDR_LOAD: S_PRDATA = r_counter;
                ADDR_CTRL: S_PRDATA = r_ctrl;
                //ADDR_CTRL: S_PRDATA = r_pres;
                default:   S_PRDATA = 0;
            endcase
    end

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

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

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

// APB wrapped vmicro16_bram
module vmicro16_bram_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter APB_PADDR    = 0,
    parameter USE_INITS    = 0,
    parameter NAME         = "BRAM",
    parameter CORE_ID      = 0
) (
    input clk,
    input reset,
    // APB Slave to master interface
    input  [`clog2(MEM_DEPTH)-1:0]  S_PADDR,
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [BUS_WIDTH-1:0]          S_PWDATA,

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

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

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

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

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .NAME       (NAME),
        .USE_INITS  (1),
        .CORE_ID    (-1)
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

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

// Shared memory with hardware monitor (LWEX/SWEX)
module vmicro16_bram_ex_apb # (
    parameter BUS_WIDTH    = 16,
    parameter MEM_WIDTH    = 16,
    parameter MEM_DEPTH    = 64,
    parameter CORE_ID_BITS = 3,
    parameter SWEX_SUCCESS = 16'h0000,
    parameter SWEX_FAIL    = 16'h0001
) (
    input clk,
    input reset,

    // |19    |18    |16             |15          0|
    // | LWEX | SWEX | 3 bit CORE_ID |     S_PADDR |
    input  [`APB_WIDTH-1:0]         S_PADDR,

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

    output reg [MEM_WIDTH-1:0]      S_PRDATA,
    output                          S_PREADY
);
    // exclusive flag checks
    wire [MEM_WIDTH-1:0] mem_out;
    reg                  swex_success = 0;

    localparam ADDR_BITS = `clog2(MEM_DEPTH);

    // hack to create a 1 clock delay to S_PREADY
    // for bram to be ready
    reg cdelay = 1;
    always @(posedge clk)
        if (S_PSELx)
            cdelay <= 0;
        else
            cdelay <= 1;

    //assign S_PRDATA = (S_PSELx & S_PENABLE) ? swex_success ? 16'hF0F0 : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE & (!cdelay)) ? 1'b1       : 1'b0;
    assign we       = (S_PSELx & S_PENABLE & S_PWRITE);
    wire   en       = (S_PSELx & S_PENABLE);

    // Similar to:
    //   http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204f/Cihbghef.html

    // mem_wd is the CORE_ID sent in bits [18:16]
    localparam TOP_BIT_INDEX         = `APB_WIDTH -1;
    localparam PADDR_CORE_ID_MSB     = TOP_BIT_INDEX - 2;
    localparam PADDR_CORE_ID_LSB     = PADDR_CORE_ID_MSB - (CORE_ID_BITS-1);

    // [LWEX, CORE_ID, mem_addr] from S_PADDR
    wire                     lwex        = S_PADDR[TOP_BIT_INDEX];
    wire                     swex        = S_PADDR[TOP_BIT_INDEX-1];
    wire [CORE_ID_BITS-1:0] core_id     = S_PADDR[PADDR_CORE_ID_MSB:PADDR_CORE_ID_LSB];
    // CORE_ID to write to ex_flags register
    wire [ADDR_BITS-1:0]    mem_addr    = S_PADDR[ADDR_BITS-1:0];

    wire [CORE_ID_BITS:0]   ex_flags_read;
    wire                     is_locked      = |ex_flags_read;
    wire                     is_locked_self = is_locked && (core_id == (ex_flags_read-1));

    // Check exclusive access flags
    always @(*) begin
        swex_success = 0;
        if (en)
            // bug!
            if (!swex && !lwex)
                swex_success = 1;
            else if (swex)
                if (is_locked && !is_locked_self)
                    // someone else has locked it
                    swex_success = 0;
                else if (is_locked && is_locked_self)
                    swex_success = 1;
    end

    always @(*)
        if (swex)
            if (swex_success)
                S_PRDATA = SWEX_SUCCESS;
            else
                S_PRDATA = SWEX_FAIL;
        else
            S_PRDATA = mem_out;

    wire reg_we = en && ((lwex && !is_locked)
                      || (swex && swex_success));

    reg  [CORE_ID_BITS:0] reg_wd;
    always @(*) begin
        reg_wd = {{CORE_ID_BITS}{1'b0}};

        if (en)
            // if wanting to lock the addr
            if (lwex)
                // and not already locked
                if (!is_locked) begin
                    reg_wd = (core_id + 1);
                end
            else if (swex)
                if (is_locked && is_locked_self)
                    reg_wd = {{CORE_ID_BITS}{1'b0}};
    end

    // Exclusive flag for each memory cell
    vmicro16_bram # (
        .MEM_WIDTH  (CORE_ID_BITS + 1),
        .MEM_DEPTH  (MEM_DEPTH),
        .USE_INITS  (0),
        .NAME       ("rexram")
    ) ram_exflags (
        .clk        (clk),
        .reset      (reset),

        .mem_addr   (mem_addr),
        .mem_in     (reg_wd),
        .mem_we     (reg_we),
        .mem_out    (ex_flags_read)
    );

    always @(*)
        if (S_PSELx && S_PENABLE)
            $display($time, "\t\tBRAMex[%h] READ %h\tCORE: %h", mem_addr, mem_out, S_PADDR[16 +: CORE_ID_BITS]);

    always @(posedge clk)
        if (we)
            $display($time, "\t\tBRAMex[%h] WRITE %h\tCORE: %h", mem_addr, S_PWDATA, S_PADDR[16 +: CORE_ID_BITS]);

    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .USE_INITS  (0),
        .NAME       ("BRAMexinst")
    ) bram_apb (
        .clk        (clk),
        .reset      (reset),

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

// Simple APB memory-mapped register set
module vmicro16_regs_apb # (
    parameter BUS_WIDTH         = 16,
    parameter DATA_WIDTH        = 16,
    parameter CELL_DEPTH        = 8,
    parameter PARAM_DEFAULTS_R0 = 0,
    parameter PARAM_DEFAULTS_R1 = 0
) (
    input clk,
    input reset,
    // APB Slave to master interface
    input  [`clog2(CELL_DEPTH)-1:0] S_PADDR,
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [DATA_WIDTH-1:0]         S_PWDATA,

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

```verilog
);
    wire [DATA_WIDTH-1:0] rd1;

    assign S_PRDATA = (S_PSELx & S_PENABLE) ? rd1  : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    assign reg_we   = (S_PSELx & S_PENABLE & S_PWRITE);

    always @(*)
        if (reg_we)
            $display($time, "\t\tREGS_APB[%h] <= %h",
                S_PADDR, S_PWDATA);

    always @(*)
        `rassert(reg_we == (S_PSELx & S_PENABLE & S_PWRITE))

    vmicro16_regs # (
        .CELL_DEPTH         (CELL_DEPTH),
        .CELL_WIDTH         (DATA_WIDTH),
        .PARAM_DEFAULTS_R0  (PARAM_DEFAULTS_R0),
        .PARAM_DEFAULTS_R1  (PARAM_DEFAULTS_R1)
    ) regs_apb (
        .clk    (clk),
        .reset  (reset),
        // port 1
        .rs1    (S_PADDR),
        .rd1    (rd1),
        .we     (reg_we),
        .ws1    (S_PADDR),
        .wd     (S_PWDATA)
        // port 2 unconnected
        //.rs2    (),
        //.rd2    ()
    );
endmodule

// Simple GPIO write only peripheral
module vmicro16_gpio_apb # (
    parameter BUS_WIDTH  = 16,
    parameter DATA_WIDTH = 16,
    parameter PORTS      = 8,
    parameter NAME       = "GPIO"
) (
    input clk,
    input reset,
    // APB Slave to master interface
    input  [0:0]                    S_PADDR, // not used (optimised out)
    input                           S_PWRITE,
    input                           S_PSELx,
    input                           S_PENABLE,
    input  [DATA_WIDTH-1:0]         S_PWDATA,

    output [DATA_WIDTH-1:0]         S_PRDATA,
    output                          S_PREADY,
    output reg [PORTS-1:0]          gpio
);
    assign S_PRDATA = (S_PSELx & S_PENABLE) ? gpio : 16'h0000;
    assign S_PREADY = (S_PSELx & S_PENABLE) ? 1'b1 : 1'b0;
    assign ports_we = (S_PSELx & S_PENABLE & S_PWRITE);

    always @(posedge clk)
        if (reset)
            gpio <= 0;
        else if (ports_we) begin
            $display($time, "\t\%s <= %h", NAME, S_PWDATA[PORTS-1:0]);
            gpio <= S_PWDATA[PORTS-1:0];
        end
endmodule
```

B.5 vmicro16.v

Vmicro16 CPU core module.

```verilog
// This file contains multiple modules.
//   Verilator likes 1 file for each module
/* verilator lint_off DECLFILENAME */
/* verilator lint_off UNUSED */
/* verilator lint_off BLKSEQ */
/* verilator lint_off WIDTH */

// Include Vmicro16 ISA containing definitions for the bits
`include "vmicro16_isa.v"

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


```

```

// This module aims to be a SYNCHRONOUS, WRITE_FIRST BLOCK RAM
//   https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
//   https://www.xilinx.com/support/documentation/user_guides/ug383.pdf
//   https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf
module vmicro16_bram # (
    parameter MEM_WIDTH     = 16,
    parameter MEM_DEPTH     = 64,
    parameter CORE_ID       = 0,
    parameter USE_INITS     = 0,
    parameter PARAM_DEFAULTS_R0 = 0,
    parameter PARAM_DEFAULTS_R1 = 0,
    parameter PARAM_DEFAULTS_R2 = 0,
    parameter PARAM_DEFAULTS_R3 = 0,
    parameter NAME          = "BRAM"
) (
    input clk,
    input reset,

    input      [`clog2(MEM_DEPTH)-1:0] mem_addr,
    input      [MEM_WIDTH-1:0]         mem_in,
    input                              mem_we,
    output reg [MEM_WIDTH-1:0]         mem_out
);
    // memory vector
    (* ram_style = "block" *)
    reg [MEM_WIDTH-1:0] mem [0:MEM_DEPTH-1];

    // not synthesizable
    integer i;
    initial begin
        for (i = 0; i < MEM_DEPTH; i = i + 1) mem[i] = 0;
        mem[0] = PARAM_DEFAULTS_R0;
        mem[1] = PARAM_DEFAULTS_R1;
        mem[2] = PARAM_DEFAULTS_R2;
        mem[3] = PARAM_DEFAULTS_R3;

        if (USE_INITS) begin
            //`define TEST_SW
            `ifdef TEST_SW
            $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\verilog_memh.txt", mem);
            `endif

            `define TEST_ASM
            `ifdef TEST_ASM
            $readmemh("E:\\Projects\\uni\\vmicro16\\sw\\asm.s.hex", mem);
            `endif

            //`define TEST_COND
            `ifdef TEST_COND
            mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
            mem[0] = {`VMICRO16_OP_MOVI,    3'h7, 8'hC0}; // lock
            `endif

            //`define TEST_CMP
            `ifdef TEST_CMP
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0A};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h0B};
            mem[2] = {`VMICRO16_OP_CMP,     3'h1, 3'h0, 5'h1};
            `endif

            //`define TEST_LWEX
            `ifdef TEST_LWEX
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC5};
            mem[1] = {`VMICRO16_OP_SW,      3'h0, 3'h0, 5'h1};
            mem[2] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h1};
            mem[3] = {`VMICRO16_OP_LWEX,    3'h2, 3'h0, 5'h1};
            mem[4] = {`VMICRO16_OP_SWEX,    3'h3, 3'h0, 5'h1};
            `endif

            //`define TEST_MULTICORE
            `ifdef TEST_MULTICORE
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h80};
            mem[4] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
            mem[5] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[6] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[7] = {`VMICRO16_OP_MOVI,    3'h1, 8'h33};
            mem[8] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
            mem[9] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h0};
            `endif

            //`define TEST_BR
            `ifdef TEST_BR
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h0};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h3, 8'h3};
            mem[2] = {`VMICRO16_OP_MOVI,    3'h1, 8'h2};
            mem[3] = {`VMICRO16_OP_ARITH_U, 3'h0, 3'h1, 5'b11111};
            mem[4] = {`VMICRO16_OP_BR,      3'h3, `VMICRO16_OP_BR_U};
            mem[5] = {`VMICRO16_OP_MOVI,    3'h0, 8'hFF};
            `endif

            //`define ALL_TEST
            `ifdef ALL_TEST
            // Standard all test
            // REGS0
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'h81};
            mem[1] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0}; // MMU[0x81] = 6
            mem[2] = {`VMICRO16_OP_SW,      3'h2, 3'h0, 5'h1}; // MMU[0x82] = 6
            // GPIO0
            mem[3] = {`VMICRO16_OP_MOVI,    3'h0, 8'h90};
            mem[4] = {`VMICRO16_OP_MOVI,    3'h1, 8'hD};
            mem[5] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[6] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
            // TIM0
            mem[7] = {`VMICRO16_OP_MOVI,    3'h0, 8'h07};
            mem[8] = {`VMICRO16_OP_LW,      3'h3, 3'h0, 5'h03};
            // UART0
            mem[9]  = {`VMICRO16_OP_MOVI,    3'h0, 8'hA0};      // UART0
            mem[10] = {`VMICRO16_OP_MOVI,    3'h1, 8'h41};      // ascii A
            mem[11] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[12] = {`VMICRO16_OP_MOVI,    3'h1, 8'h42}; // ascii B
            mem[13] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[14] = {`VMICRO16_OP_MOVI,    3'h1, 8'h43}; // ascii C
            mem[15] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[16] = {`VMICRO16_OP_MOVI,    3'h1, 8'h44}; // ascii D
            mem[17] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[18] = {`VMICRO16_OP_MOVI,    3'h1, 8'h45}; // ascii D
            mem[19] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[20] = {`VMICRO16_OP_MOVI,    3'h1, 8'h46}; // ascii E
            mem[21] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            // BRAM0
            mem[22] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
            mem[23] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
            mem[24] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
            mem[25] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
            // GPIO1 (SSD 24-bit port)
            mem[26] = {`VMICRO16_OP_MOVI,    3'h0, 8'h91};
            mem[27] = {`VMICRO16_OP_MOVI,    3'h1, 8'h12};
            mem[28] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            mem[29] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h0};
            // GPIO2
            mem[30] = {`VMICRO16_OP_MOVI,    3'h0, 8'h92};
            mem[31] = {`VMICRO16_OP_MOVI,    3'h1, 8'h56};
            mem[32] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h0};
            `endif

            //`define TEST_BRAM
            `ifdef TEST_BRAM
            // 2 core BRAM0 test
            mem[0] = {`VMICRO16_OP_MOVI,    3'h0, 8'hC0};
            mem[1] = {`VMICRO16_OP_MOVI,    3'h1, 8'hA};
            mem[2] = {`VMICRO16_OP_SW,      3'h1, 3'h0, 5'h5};
            mem[3] = {`VMICRO16_OP_LW,      3'h2, 3'h0, 5'h5};
            `endif
        end
    end

    always @(posedge clk) begin
        // synchronous WRITE_FIRST (page 13)
        if (mem_we) begin
            mem[mem_addr] <= mem_in;
            $display($time, "\t\t%s[%h] <= %h",
                    NAME, mem_addr, mem_in);
        end else
            mem_out <= mem[mem_addr];
    end

    // TODO: Reset impl = every clock while reset is asserted, clear each cell
    //       one at a time, mem[i++] <= 0
endmodule


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

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

    input  req,
    output busy,

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

    // interrupts
    output reg [`DATA_WIDTH*`DEF_NUM_INT-1:0] ints_vector,
    output reg [`DEF_NUM_INT-1:0]             ints_mask,

    // TO APB interconnect
    output reg [`APB_WIDTH-1:0]  M_PADDR,
    output reg                   M_PWRITE,
    output reg                   M_PSELx,
    output reg                   M_PENABLE,
    output reg [MEM_WIDTH-1:0]   M_PWDATA,
    // from interconnect
    input      [MEM_WIDTH-1:0]   M_PRDATA,
    input                        M_PREADY
);
    localparam MMU_STATE_T1  = 0;
    localparam MMU_STATE_T2  = 1;
    localparam MMU_STATE_T3  = 2;
    reg [1:0]  mmu_state      = MMU_STATE_T1;

    reg  [MEM_WIDTH-1:0] per_out = 0;
    wire [MEM_WIDTH-1:0] tim0_out;

    assign busy = req || (mmu_state == MMU_STATE_T2);

    // more luts than below but easier
    wire tim0_en = (mmu_addr >= `DEF_MMU_TIM0_S)
                && (mmu_addr <= `DEF_MMU_TIM0_E);
    wire sreg_en = (mmu_addr >= `DEF_MMU_SREG_S)
                && (mmu_addr <= `DEF_MMU_SREG_E);
    wire intv_en = (mmu_addr >= `DEF_MMU_INTSV_S)
                && (mmu_addr <= `DEF_MMU_INTSV_E);
    wire intm_en = (mmu_addr >= `DEF_MMU_INTSM_S)
                && (mmu_addr <= `DEF_MMU_INTSM_E);

    //wire tim0_en = ~mmu_addr[12] && ~mmu_addr[9] && ~mmu_addr[7];
    //wire sreg_en = mmu_addr[7] && ~mmu_addr[4] && ~mmu_addr[5];
    //wire intv_en = mmu_addr[8] && ~mmu_addr[3];
    //wire intm_en = mmu_addr[8] && mmu_addr[3];

    wire apb_en    = !(|{tim0_en, sreg_en, intv_en, intm_en});
    wire tim0_we   = (tim0_en && mmu_we);
    wire intv_we   = (intv_en && mmu_we);
    wire intm_we   = (intm_en && mmu_we);

    // Special register selects
    localparam SPECIAL_REGS = 8;
    wire [MEM_WIDTH-1:0] sr_val;

    // Interrupt vector and mask
    initial ints_vector = 0;
    initial ints_mask   = 0;
    wire [2:0] intv_addr = mmu_addr[`clog2(`DEF_NUM_INT)-1:0];
    always @(posedge clk)
        if (intv_we)
            ints_vector[intv_addr*`DATA_WIDTH +: `DATA_WIDTH] <= mmu_in;

    always @(posedge clk)
        if (intm_we)
            ints_mask <= mmu_in;


    always @(ints_vector)
        $display($time,
                "\tC%d\t\tints_vector W: | %h %h %h %h | %h %h %h %h |",
                 CORE_ID,
            ints_vector[0*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[1*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[2*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[3*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[4*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[5*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[6*`DATA_WIDTH +: `DATA_WIDTH],
            ints_vector[7*`DATA_WIDTH +: `DATA_WIDTH]
            );

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

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

    // APB master to slave interface
    always @(posedge clk)
        if (reset) begin
            mmu_state <= MMU_STATE_T1;
            M_PENABLE <= 0;
            M_PADDR   <= 0;
            M_PWDATA  <= 0;
            M_PSELx   <= 0;
            M_PWRITE  <= 0;
        end
        else
            casex (mmu_state)
                MMU_STATE_T1: begin
                    if (req && apb_en) begin
                        M_PADDR   <= {mmu_lwex,
                                      mmu_swex,
                                      CORE_ID[CORE_ID_BITS-1:0],
                                      mmu_addr[MEM_WIDTH-1:0]};

                        M_PWDATA  <= mmu_in;
                        M_PSELx   <= 1;
                        M_PWRITE  <= mmu_we;

                        mmu_state <= MMU_STATE_T2;
                    end
                end

                `ifdef FIX_T3
                    MMU_STATE_T2: begin
                        M_PENABLE <= 1;

                        if (M_PREADY == 1'b1) begin
                            mmu_state <= MMU_STATE_T3;
                        end
                    end

                    MMU_STATE_T3: begin
                        // Slave has output a ready signal (finished)
                        M_PENABLE <= 0;
                        M_PADDR   <= 0;
                        M_PWDATA  <= 0;
                        M_PSELx   <= 0;
                        M_PWRITE  <= 0;
                        // Clock the peripheral output into a reg,
                        //   to output on the next clock cycle
                        per_out   <= M_PRDATA;

                        mmu_state <= MMU_STATE_T1;
                    end
                `else
                    // No FIX_T3
                    MMU_STATE_T2: begin
                        if (M_PREADY == 1'b1) begin
                            M_PENABLE <= 0;
                            M_PADDR   <= 0;
                            M_PWDATA  <= 0;
                            M_PSELx   <= 0;
                            M_PWRITE  <= 0;
                            // Clock the peripheral output into a reg,
                            //   to output on the next clock cycle
                            per_out   <= M_PRDATA;

                            mmu_state <= MMU_STATE_T1;
                        end else begin
                            M_PENABLE <= 1;
                        end
                    end
                `endif
            endcase

    (* ram_style = "block" *)
    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (SPECIAL_REGS),
        .USE_INITS  (0),
        .PARAM_DEFAULTS_R0  (CORE_ID),
        .PARAM_DEFAULTS_R1  (`CORES),
        .PARAM_DEFAULTS_R2  (`APB_BRAM0_CELLS),
        .PARAM_DEFAULTS_R3  (`SLAVES),
        .NAME       ("ram_sr")
    ) ram_sr (
        .clk        (clk),
        .reset      (reset),
        .mem_addr   (mmu_addr[`clog2(SPECIAL_REGS)-1:0]),
        .mem_in     (),
        .mem_we     (),
        .mem_out    (sr_val)
    );

    // Each M core has a TIM0 scratch memory
    (* ram_style = "block" *)
    vmicro16_bram # (
        .MEM_WIDTH  (MEM_WIDTH),
        .MEM_DEPTH  (MEM_DEPTH),
        .USE_INITS  (0),
        .NAME       ("TIM0")
    ) TIM0 (
        .clk        (clk),
        .reset      (reset),
        .mem_addr   (mmu_addr[7:0]),
        .mem_in     (mmu_in),
        .mem_we     (tim0_we),
        .mem_out    (tim0_out)
    );
endmodule



module vmicro16_regs # (
    parameter CELL_WIDTH        = 16,
    parameter CELL_DEPTH        = 8,
    parameter CELL_SEL_BITS     = `clog2(CELL_DEPTH),
    parameter CELL_DEFAULTS     = 0,
    parameter DEBUG_NAME        = "",
    parameter CORE_ID           = 0,
    parameter PARAM_DEFAULTS_R0 = 16'h0000,
    parameter PARAM_DEFAULTS_R1 = 16'h0000
) (
    input clk,
    input reset,
    // Dual port register reads
    input      [CELL_SEL_BITS-1:0]  rs1, // port 1
    output     [CELL_WIDTH-1   :0]  rd1,
    //input       [CELL_SEL_BITS-1:0]  rs2, // port 2
    //output      [CELL_WIDTH-1   :0]  rd2,
    // EX/WB final stage write back
    input                           we,
    input [CELL_SEL_BITS-1:0]       ws1,
    input [CELL_WIDTH-1:0]          wd
);
    (* ram_style = "distributed" *)
    reg [CELL_WIDTH-1:0] regs [0:CELL_DEPTH-1] /*verilator public_flat*/;

    // Initialise registers with default values
    //   Really only used for special registers used by the soc
    // TODO: How to do this on reset?
    integer i;
    initial
        if (CELL_DEFAULTS)
            $readmemh(CELL_DEFAULTS, regs);
        else begin
            for(i = 0; i < CELL_DEPTH; i = i + 1)
                regs[i] = 0;
            regs[0] = PARAM_DEFAULTS_R0;
            regs[1] = PARAM_DEFAULTS_R1;
            end

    `ifdef ICARUS
        always @(regs)
            $display($time, "\tC%02h\t\t| %h %h %h %h | %h %h %h %h |",
                CORE_ID,
                regs[0], regs[1], regs[2], regs[3],
                regs[4], regs[5], regs[6], regs[7]);
    `endif

    always @(posedge clk)
        if (reset) begin
            for(i = 0; i < CELL_DEPTH; i = i + 1)
                regs[i] <= 0;
            regs[0] <= PARAM_DEFAULTS_R0;
            regs[1] <= PARAM_DEFAULTS_R1;
        end
        else if (we) begin
            $display($time, "\tC%02h: REGS #%s: Writing %h to reg[%d]",
                CORE_ID, DEBUG_NAME, wd, ws1);

            // Perform the write
            regs[ws1] <= wd;
        end

    // sync writes, async reads
    assign rd1 = regs[rs1];
    //assign rd2 = regs[rs2];
endmodule

module vmicro16_dec # (
    parameter INSTR_WIDTH    = 16,
    parameter INSTR_OP_WIDTH = 5,
    parameter INSTR_RS_WIDTH = 3,
    parameter ALU_OP_WIDTH   = 5
) (
    //input clk,   // not used yet (all combinational)
    //input reset, // not used yet (all combinational)

    input  [INSTR_WIDTH-1:0]    instr,

    output [INSTR_OP_WIDTH-1:0] opcode,
    output [INSTR_RS_WIDTH-1:0] rd,
    output [INSTR_RS_WIDTH-1:0] ra,
    output [3:0]                imm4,
    output [7:0]                imm8,
    output [11:0]               imm12,
    output [4:0]                simm5,

    // This can be freely increased without affecting the isa
    output reg [ALU_OP_WIDTH-1:0] alu_op,

    output reg has_imm4,
    output reg has_imm8,
    output reg has_imm12,
    output reg has_we,
    output reg has_br,
    output reg has_mem,
    output reg has_mem_we,
    output reg has_cmp,

    output halt,
    output intr,

    output reg has_lwex,
    output reg has_swex

    // TODO: Use to identify bad instruction and
    //       raise exceptions
    //,output     is_bad
);
    assign opcode = instr[15:11];
    assign rd     = instr[10:8];
    assign ra     = instr[7:5];
    assign imm4   = instr[3:0];
    assign imm8   = instr[7:0];
    assign imm12  = instr[11:0];
    assign simm5  = instr[4:0];

    // exme_op
    always @(*) case (opcode)
        `VMICRO16_OP_SPCL: casez(instr[11:0])
            `VMICRO16_OP_SPCL_NOP,
            `VMICRO16_OP_SPCL_HALT,
            `VMICRO16_OP_SPCL_INTR:   alu_op = `VMICRO16_ALU_NOP;
            default:                  alu_op = `VMICRO16_ALU_NOP; endcase

        `VMICRO16_OP_LW:              alu_op = `VMICRO16_ALU_LW;
        `VMICRO16_OP_SW:              alu_op = `VMICRO16_ALU_SW;
        `VMICRO16_OP_LWEX:            alu_op = `VMICRO16_ALU_LW;
        `VMICRO16_OP_SWEX:            alu_op = `VMICRO16_ALU_SW;

        `VMICRO16_OP_MOV:             alu_op = `VMICRO16_ALU_MOV;
        `VMICRO16_OP_MOVI:            alu_op = `VMICRO16_ALU_MOVI;

        `VMICRO16_OP_BR:              alu_op = `VMICRO16_ALU_BR;
        `VMICRO16_OP_MULT:            alu_op = `VMICRO16_ALU_MULT;

        `VMICRO16_OP_CMP:             alu_op = `VMICRO16_ALU_CMP;
        `VMICRO16_OP_SETC:            alu_op = `VMICRO16_ALU_SETC;

        `VMICRO16_OP_BIT:     casez (simm5)
            `VMICRO16_OP_BIT_OR:      alu_op = `VMICRO16_ALU_BIT_OR;
            `VMICRO16_OP_BIT_XOR:     alu_op = `VMICRO16_ALU_BIT_XOR;
            `VMICRO16_OP_BIT_AND:     alu_op = `VMICRO16_ALU_BIT_AND;
            `VMICRO16_OP_BIT_NOT:     alu_op = `VMICRO16_ALU_BIT_NOT;
            `VMICRO16_OP_BIT_LSHFT:   alu_op = `VMICRO16_ALU_BIT_LSHFT;
            `VMICRO16_OP_BIT_RSHFT:   alu_op = `VMICRO16_ALU_BIT_RSHFT;
            default:                  alu_op = `VMICRO16_ALU_BAD; endcase

        `VMICRO16_OP_ARITH_U:     casez (simm5)
            `VMICRO16_OP_ARITH_UADD:  alu_op = `VMICRO16_ALU_ARITH_UADD;
            `VMICRO16_OP_ARITH_USUB:  alu_op = `VMICRO16_ALU_ARITH_USUB;
            `VMICRO16_OP_ARITH_UADDI: alu_op = `VMICRO16_ALU_ARITH_UADDI;
            default:                  alu_op = `VMICRO16_ALU_BAD; endcase

        `VMICRO16_OP_ARITH_S:     casez (simm5)
            `VMICRO16_OP_ARITH_SADD:  alu_op = `VMICRO16_ALU_ARITH_SADD;
            `VMICRO16_OP_ARITH_SSUB:  alu_op = `VMICRO16_ALU_ARITH_SSUB;
            `VMICRO16_OP_ARITH_SSUBI: alu_op = `VMICRO16_ALU_ARITH_SSUBI;
            default:                  alu_op = `VMICRO16_ALU_BAD; endcase

        default: begin
                                      alu_op = `VMICRO16_ALU_NOP;
            $display($time, "\tDEC: unknown opcode: %h ... NOPPING", opcode);
        end
    endcase

    // Special opcodes
    //assign nop  == ((opcode == `VMICRO16_OP_SPCL) & (~instr[0]));
    assign halt = ((opcode == `VMICRO16_OP_SPCL) &   instr[0]);
    assign intr = ((opcode == `VMICRO16_OP_SPCL) &   instr[1]);

    // Register writes
    always @(*) case (opcode)
        `VMICRO16_OP_LWEX,
        `VMICRO16_OP_SWEX,
        `VMICRO16_OP_LW,
        `VMICRO16_OP_MOV,
        `VMICRO16_OP_MOVI,
        //`VMICRO16_OP_MOVI_L,
        `VMICRO16_OP_ARITH_U,
        `VMICRO16_OP_ARITH_S,
        `VMICRO16_OP_SETC,
        `VMICRO16_OP_BIT,
        `VMICRO16_OP_MULT:      has_we = 1'b1;
        default:                has_we = 1'b0;
    endcase

    // Contains 4-bit immediate
    always @(*)
        if( ((opcode == `VMICRO16_OP_ARITH_U) && (simm5[4] == 0)) ||
            ((opcode == `VMICRO16_OP_ARITH_S) && (simm5[4] == 0)) )
            has_imm4 = 1'b1;
        else
            has_imm4 = 1'b0;

    // Contains 8-bit immediate
    always @(*) case (opcode)
        `VMICRO16_OP_MOVI,
        `VMICRO16_OP_BR:        has_imm8 = 1'b1;
        default:                has_imm8 = 1'b0;
    endcase

    //// Contains 12-bit immediate
    //always @(*) case (opcode)
    //    `VMICRO16_OP_MOVI_L:    has_imm12 = 1'b1;
    //    default:                has_imm12 = 1'b0;
    //endcase

    // Will branch the pc
    always @(*) case (opcode)
        `VMICRO16_OP_BR:    has_br = 1'b1;
        default:            has_br = 1'b0;
    endcase

    // Requires external memory
    always @(*) case (opcode)
        `VMICRO16_OP_LW,
        `VMICRO16_OP_SW,
        `VMICRO16_OP_LWEX,
        `VMICRO16_OP_SWEX:  has_mem = 1'b1;
        default:            has_mem = 1'b0;
    endcase

    // Requires external memory write
    always @(*) case (opcode)
        `VMICRO16_OP_SW,
        `VMICRO16_OP_SWEX:  has_mem_we = 1'b1;
        default:            has_mem_we = 1'b0;
    endcase

    // Affects status registers (cmp instructions)
    always @(*) case (opcode)
        `VMICRO16_OP_CMP:   has_cmp = 1'b1;
        default:            has_cmp = 1'b0;
    endcase

    // Performs exclusive checks
    always @(*) case (opcode)
        `VMICRO16_OP_LWEX:   has_lwex = 1'b1;
        default:             has_lwex = 1'b0;
    endcase

    always @(*) case (opcode)
        `VMICRO16_OP_SWEX:   has_swex = 1'b1;
        default:             has_swex = 1'b0;
    endcase
endmodule


module vmicro16_alu # (
    parameter OP_WIDTH   = 5,
    parameter DATA_WIDTH = 16,
    parameter CORE_ID    = 0
) (
    // input clk, // TODO: make clocked

    input       [OP_WIDTH-1:0]   op,
    input      [DATA_WIDTH-1:0] a, // rs1/dst
    input      [DATA_WIDTH-1:0] b, // rs2
    input      [3:0]            flags,
    output reg [DATA_WIDTH-1:0] c
);
    localparam TOP_BIT = (DATA_WIDTH-1);
    // 17-bit register
    reg [DATA_WIDTH:0] cmp_tmp = 0; // = {carry, [15:0]}
    wire r_setc;

    always @(*) begin
                        cmp_tmp = 0;
                        case (op)
        // branch/nop, output nothing
        `VMICRO16_ALU_BR,
        `VMICRO16_ALU_NOP:          c = {DATA_WIDTH{1'b0}};
        // load/store addresses (use value in rd2)
        `VMICRO16_ALU_LW,
        `VMICRO16_ALU_SW:           c = b;
        // bitwise operations
        `VMICRO16_ALU_BIT_OR:       c = a | b;
        `VMICRO16_ALU_BIT_XOR:      c = a ^ b;
        `VMICRO16_ALU_BIT_AND:      c = a & b;
        `VMICRO16_ALU_BIT_NOT:      c = ~(b);
        `VMICRO16_ALU_BIT_LSHFT:    c = a << b;
        `VMICRO16_ALU_BIT_RSHFT:    c = a >> b;

        `VMICRO16_ALU_MOV:          c = b;
        `VMICRO16_ALU_MOVI:         c = b;
        `VMICRO16_ALU_MOVI_L:       c = b;

        `VMICRO16_ALU_ARITH_UADD:   c = a + b;
        `VMICRO16_ALU_ARITH_USUB:   c = a - b;
        // TODO: ALU should have simm5 as input
        `VMICRO16_ALU_ARITH_UADDI:  c = a + b;

        `ifdef DEF_ALU_HW_MULT
            `VMICRO16_ALU_MULT:  c = a * b;
        `endif

        `VMICRO16_ALU_ARITH_SADD:   c = $signed(a) + $signed(b);
        `VMICRO16_ALU_ARITH_SSUB:   c = $signed(a) - $signed(b);
        // TODO: ALU should have simm5 as input
        `VMICRO16_ALU_ARITH_SSUBI:  c = $signed(a) - $signed(b);

        `VMICRO16_ALU_CMP: begin
            // TODO: Do a-b in 17-bit register
            //       Set zero, overflow, carry, signed bits in result
            cmp_tmp = a - b;
            c = 0;

            // N   Negative condition code flag
            // Z   Zero condition code flag
            // C   Carry condition code flag
            // V   Overflow condition code flag
            c[`VMICRO16_SFLAG_N] = cmp_tmp[TOP_BIT];
            c[`VMICRO16_SFLAG_Z] = (cmp_tmp == 0);
            c[`VMICRO16_SFLAG_C] = 0; //cmp_tmp[TOP_BIT+1]; // not used

            // Overflow flag
            // https://stackoverflow.com/questions/30957188/
            // https://github.com/bendl/prco304/blob/master/prco_core/rtl/prco_alu.v#L50
            case(cmp_tmp[TOP_BIT+1:TOP_BIT])
                2'b01:   c[`VMICRO16_SFLAG_V] = 1;
                2'b10:   c[`VMICRO16_SFLAG_V] = 1;
                default: c[`VMICRO16_SFLAG_V] = 0;
            endcase

            $display($time, "\tC%02h: ALU CMP: %h %h = %h = %b", CORE_ID, a, b, cmp_tmp, c[3:0]);
        end

        `VMICRO16_ALU_SETC: c = { {15{1'b0}}, r_setc };

        // TODO: Parameterise
        default: begin
            $display($time, "\tALU: unknown op: %h", op);
            c       = 0;
            cmp_tmp = 0;
        end
                endcase
                end

    branch setc_check (
        .flags      (flags),
        .cond       (b[7:0]),
        .en         (r_setc)
    );
endmodule

// flags = 4 bit r_cmp_flags register
// cond  = 8 bit VMICRO16_OP_BR_? value. See vmicro16_isa.v
```

```
module branch (
    input [3:0] flags,
    input [7:0] cond,
    output reg  en
);
    always @(*)
        case (cond)
            `VMICRO16_OP_BR_U:  en = 1; `VMICRO16_OP_BR_U:  en = 1;
            `VMICRO16_OP_BR_E:  en = (flags[`VMICRO16_SFLAG_Z] == 1);
            `VMICRO16_OP_BR_NE: en = (flags[`VMICRO16_SFLAG_Z] == 0);
            `VMICRO16_OP_BR_G:  en = (flags[`VMICRO16_SFLAG_Z] == 0) &&
                                     (flags[`VMICRO16_SFLAG_N] == flags[`VMICRO16_SFLAG_V]);
            `VMICRO16_OP_BR_L:  en = (flags[`VMICRO16_SFLAG_Z] != flags[`VMICRO16_SFLAG_N]);
            `VMICRO16_OP_BR_GE: en = (flags[`VMICRO16_SFLAG_Z] == flags[`VMICRO16_SFLAG_N]);
            `VMICRO16_OP_BR_LE: en = (flags[`VMICRO16_SFLAG_Z] == 1) ||
                                     (flags[`VMICRO16_SFLAG_N] != flags[`VMICRO16_SFLAG_V]);
            default:            en = 0;
        endcase
endmodule



module vmicro16_core # (
    parameter DATA_WIDTH        = 16,
    parameter MEM_INSTR_DEPTH   = 64,
    parameter MEM_SCRATCH_DEPTH = 64,
    parameter MEM_WIDTH         = 16,

    parameter CORE_ID           = 3'h0
) (
    input        clk,
    input        reset,

    output [7:0] dbug,

    output       halt,

    // interrupt sources
    input   [`DEF_NUM_INT-1:0]             ints,
    input   [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_data,
    output  [`DEF_NUM_INT-1:0]             ints_ack,

    // APB master to slave interface (apb_intercon)
    output   [`APB_WIDTH-1:0]    w_PADDR,
    output                       w_PWRITE,
    output                       w_PSELx,
    output                       w_PENABLE,
    output   [DATA_WIDTH-1:0]    w_PWDATA,
    input    [DATA_WIDTH-1:0]    w_PRDATA,
    input                        w_PREADY

`ifndef DEF_CORE_HAS_INSTR_MEM
    , // APB master interface to slave instruction memory
    output reg [`APB_WIDTH-1:0]    w2_PADDR,
    output reg                     w2_PWRITE,
    output reg                     w2_PSELx,
    output reg                     w2_PENABLE,
    output reg [DATA_WIDTH-1:0]    w2_PWDATA,
    input      [DATA_WIDTH-1:0]    w2_PRDATA,
    input                          w2_PREADY
`endif
);
    localparam STATE_IF = 0;
    localparam STATE_R1 = 1;
    localparam STATE_R2 = 2;
    localparam STATE_ME = 3;
    localparam STATE_WB = 4;
    localparam STATE_FE = 5;
    localparam STATE_IDLE = 6;
    localparam STATE_HALT = 7;
    reg  [2:0] r_state = STATE_IF;

    reg  [DATA_WIDTH-1:0] r_pc           = 16'h0000;
    reg  [DATA_WIDTH-1:0] r_pc_saved     = 16'h0000;
    reg  [DATA_WIDTH-1:0] r_instr        = 16'h0000;
    wire [DATA_WIDTH-1:0] w_mem_instr_out;
    wire                  w_halt;

    assign dbug = {7'h00, w_halt};
    assign halt = w_halt;

    wire [4:0]            r_instr_opcode;
    wire [4:0]            r_instr_alu_op;
    wire [2:0]            r_instr_rsd;
    wire [2:0]            r_instr_rsa;
    reg  [DATA_WIDTH-1:0] r_instr_rdd = 0;
    reg  [DATA_WIDTH-1:0] r_instr_rda = 0;
    wire [3:0]            r_instr_imm4;
    wire [7:0]            r_instr_imm8;
    wire [4:0]            r_instr_simm5;
    wire                  r_instr_has_imm4;
    wire                 r_instr_has_imm8;
    wire                 r_instr_has_we;
    wire                 r_instr_has_br;
    wire                 r_instr_has_cmp;
    wire                 r_instr_has_mem;
    wire                 r_instr_has_mem_we;
    wire                 r_instr_halt;
    wire                 r_instr_has_lwex;
    wire                 r_instr_has_swex;

    wire [DATA_WIDTH-1:0] r_alu_out;

    wire [DATA_WIDTH-1:0] r_mem_scratch_addr = $signed(r_alu_out) + $signed(r_instr_simm5);
    wire [DATA_WIDTH-1:0] r_mem_scratch_in   = r_instr_rdd;
    wire [DATA_WIDTH-1:0] r_mem_scratch_out;
    wire                 r_mem_scratch_we   = r_instr_has_mem_we && (r_state == STATE_ME);
    reg                  r_mem_scratch_req  = 0;
    wire                 r_mem_scratch_busy;

    reg  [2:0]           r_reg_rs1 = 0;
    wire [DATA_WIDTH-1:0] r_reg_rd1_s;
    wire [DATA_WIDTH-1:0] r_reg_rd1_i;
    wire [DATA_WIDTH-1:0] r_reg_rd1 = regs_use_int ? r_reg_rd1_i : r_reg_rd1_s;
    //wire [15:0] r_reg_rd2;
    wire [DATA_WIDTH-1:0] r_reg_wd = (r_instr_has_mem) ? r_mem_scratch_out : r_alu_out;
    wire                 r_reg_we = r_instr_has_we && (r_state == STATE_WB);

    // branching
    wire        w_intr;
    wire        w_branch_en;
    wire        w_branching   = r_instr_has_br && w_branch_en;
    reg  [3:0]  r_cmp_flags   = 4'h00; // N, Z, C, V

    always @(r_cmp_flags)
        $display($time, "\tC%02h:\tALU CMP: %b", CORE_ID, r_cmp_flags);

    // 2 cycle register fetch
    always @(*) begin
        r_reg_rs1 = 0;
        if (r_state == STATE_R1)
            r_reg_rs1 = r_instr_rsd;
        else if (r_state == STATE_R2)
            r_reg_rs1 = r_instr_rsa;
        else
            r_reg_rs1 = 3'h0;
    end

    reg regs_use_int = 0;
    `ifdef DEF_ENABLE_INT
    wire [`DEF_NUM_INT*`DATA_WIDTH-1:0] ints_vector;
    wire [`DEF_NUM_INT-1:0]             ints_mask;
    wire                                has_int = ints & ints_mask;
    reg int_pending = 0;
    reg int_pending_ack = 0;
    always @(posedge clk)
        if (int_pending_ack)
            // We've now branched to the isr
            int_pending <= 0;
        else if (has_int)
            // Notify fsm to switch to the ints_vector at the last stage
            int_pending <= 1;
        else if (w_intr)
            // Return to Interrupt instruction called,
            //   so we've finished with the interrupt
            int_pending <= 0;
    `endif

    // Next program counter logic
    reg [`DATA_WIDTH-1:0] next_pc = 0;
    always @(posedge clk)
        if (reset)
            r_pc <= 0;
        else if (r_state == STATE_WB) begin
            `ifdef DEF_ENABLE_INT
            if (int_pending) begin
                $display($time, "\tC%02h: Jumping to ISR: %h",
                    CORE_ID,
                    ints_vector[0 +: `DATA_WIDTH]);
                // TODO: check bounds
                // Save state
                r_pc_saved      <= r_pc + 1;
                regs_use_int    <= 1;
                int_pending_ack <= 1;
                // Jump to ISR
                r_pc            <= ints_vector[0 +: `DATA_WIDTH];
            end else if (w_intr) begin
                $display($time, "\tC%02h: Returning from ISR: %h",
                    CORE_ID, r_pc_saved);

                // Restore state
                r_pc            <= r_pc_saved;
                regs_use_int    <= 0;
                int_pending_ack <= 0;
            end else
            `endif
            if (w_branching) begin
                $display($time, "\tC%02h: branching to %h", CORE_ID, r_instr_rdd);
                r_pc            <= r_instr_rdd;

                `ifdef DEF_ENABLE_INT
                    int_pending_ack <= 0;
                `endif
            end else if (r_pc < (MEM_INSTR_DEPTH-1)) begin
                // normal increment
                // pc <= pc + 1
                r_pc            <= r_pc + 1;

                `ifdef DEF_ENABLE_INT
                    int_pending_ack <= 0;
                `endif
            end
        end // end r_state == STATE_WB
        else if (r_state == STATE_HALT) begin
            `ifdef DEF_ENABLE_INT
            // Only an interrupt can return from halt
            // duplicate code form STATE_ME!
            if (int_pending) begin
                $display($time, "\tC%02h: Jumping to ISR: %h", CORE_ID, ints_vector[0 +: `DATA_WIDTH]);
                // TODO: check bounds
                // Save state
                r_pc_saved      <= r_pc;// + 1; HALT = stay with same PC
                regs_use_int    <= 1;
                int_pending_ack <= 1;
                // Jump to ISR
                r_pc            <= ints_vector[0 +: `DATA_WIDTH];
            end else if (w_intr) begin
                $display($time, "\tC%02h: Returning from ISR: %h", CORE_ID, r_pc_saved);
                r_pc            <= r_pc_saved;
                regs_use_int    <= 0;
                int_pending_ack <= 0;
            end
            `endif
        end

`ifndef DEF_CORE_HAS_INSTR_MEM
    initial w2_PSELx   = 0;
    initial w2_PENABLE = 0;
    initial w2_PADDR   = 0;
`endif

    // cpu state machine
    always @(posedge clk)
        if (reset) begin
            r_state           <= STATE_IF;
            r_instr           <= 0;
            r_mem_scratch_req <= 0;
            r_instr_rdd       <= 0;
            r_instr_rda       <= 0;
        end
        else begin

`ifdef DEF_CORE_HAS_INSTR_MEM
            if (r_state == STATE_IF) begin
                    r_instr <= w_mem_instr_out;

                    $display("");
                    $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
                    $display($time, "\tC%02h: INSTR: %h", CORE_ID, w_mem_instr_out);

                    r_state <= STATE_R1;
            end
`else
            // wait for global instruction rom to give us our instruction
            if (r_state == STATE_IF) begin
                // wait for ready signal
                if (!w2_PREADY) begin
                    w2_PSELx   <= 1;
                    w2_PWRITE  <= 0;
                    w2_PENABLE <= 1;
                    w2_PWDATA  <= 0;
                    w2_PADDR   <= r_pc;
                end else begin
                    w2_PSELx   <= 0;
                    w2_PWRITE  <= 0;
                    w2_PENABLE <= 0;
                    w2_PWDATA  <= 0;

                    r_instr <= w2_PRDATA;

                    $display("");
                    $display($time, "\tC%02h: PC: %h",    CORE_ID, r_pc);
                    $display($time, "\tC%02h: INSTR: %h", CORE_ID, w2_PRDATA);

                    r_state <= STATE_R1;
                end
            end
`endif

            else if (r_state == STATE_R1) begin
                if (w_halt) begin
                    $display("");
                    $display("");
                    $display($time, "\tC%02h: PC: %h HALT", CORE_ID, r_pc);
                    r_state <= STATE_HALT;
                end else begin
                    // primary operand
                    r_instr_rdd <= r_reg_rd1;
                    r_state     <= STATE_R2;
                end
            end
            else if (r_state == STATE_R2) begin
                // Choose secondary operand (register or immediate)
                if      (r_instr_has_imm8)  r_instr_rda <= r_instr_imm8;
                else if (r_instr_has_imm4)  r_instr_rda <= r_reg_rd1 + r_instr_imm4;
                else                        r_instr_rda <= r_reg_rd1;

                if (r_instr_has_mem) begin
                    r_state           <= STATE_ME;
                    // Pulse req
                    r_mem_scratch_req <= 1;
                end else
                    r_state <= STATE_WB;
            end
            else if (r_state == STATE_ME) begin
                // Pulse req
                r_mem_scratch_req <= 0;
                // Wait for MMU to finish
                if (!r_mem_scratch_busy)
                    r_state <= STATE_WB;
            end
            else if (r_state == STATE_WB) begin
                if (r_instr_has_cmp) begin
                    $display($time, "\tC%02h: CMP: %h", CORE_ID, r_alu_out[3:0]);
                    r_cmp_flags <= r_alu_out[3:0];
                end

                r_state <= STATE_FE;
            end
            else if (r_state == STATE_FE)
                r_state <= STATE_IF;
            else if (r_state == STATE_HALT) begin
                `ifdef DEF_ENABLE_INT
                    if (int_pending) begin
                        r_state <= STATE_FE;
                    end
                `endif
            end
        end

`ifdef DEF_CORE_HAS_INSTR_MEM
    // Instruction ROM
    (* rom_style = "distributed" *)
    vmicro16_bram # (
        .MEM_WIDTH      (DATA_WIDTH),
        .MEM_DEPTH      (MEM_INSTR_DEPTH),
        .CORE_ID        (CORE_ID),
        .USE_INITS      (1),
        .NAME           ("INSTR_MEM")
    ) mem_instr (
        .clk            (clk),
        .reset          (reset),
        // port 1
        .mem_addr       (r_pc),
        .mem_in         (0),
        .mem_we         (1'b0),  // ROM
        .mem_out        (w_mem_instr_out)
    );
`endif

    // MMU
    vmicro16_core_mmu # (
        .MEM_WIDTH      (DATA_WIDTH),
        .MEM_DEPTH      (MEM_SCRATCH_DEPTH),
        .CORE_ID        (CORE_ID)
    ) mmu (
        .clk            (clk),
        .reset          (reset),
        .req            (r_mem_scratch_req),
        .busy           (r_mem_scratch_busy),
        // interrupts
        .ints_vector    (ints_vector),
        .ints_mask      (ints_mask),
        // port 1
        .mmu_addr       (r_mem_scratch_addr),
        .mmu_in         (r_mem_scratch_in),
        .mmu_we         (r_mem_scratch_we),
        .mmu_lwex       (r_instr_has_lwex),
        .mmu_swex       (r_instr_has_swex),
        .mmu_out        (r_mem_scratch_out),
        // APB maste    r to slave
        .M_PADDR        (w_PADDR),
        .M_PWRITE       (w_PWRITE),
        .M_PSELx        (w_PSELx),
        .M_PENABLE      (w_PENABLE),
        .M_PWDATA       (w_PWDATA),
        .M_PRDATA       (w_PRDATA),
        .M_PREADY       (w_PREADY)
    );

    // Instruction decoder
    vmicro16_dec dec (
        // input
        .instr          (r_instr),
        // output async
        .opcode         (),
        .rd             (r_instr_rsd),
        .ra             (r_instr_rsa),
        .imm4           (r_instr_imm4),
        .imm8           (r_instr_imm8),
        .imm12          (),
        .simm5          (r_instr_simm5),
        .alu_op         (r_instr_alu_op),
        .has_imm4       (r_instr_has_imm4),
        .has_imm8       (r_instr_has_imm8),
        .has_we         (r_instr_has_we),
        .has_br         (r_instr_has_br),
        .has_cmp        (r_instr_has_cmp),
        .has_mem        (r_instr_has_mem),
        .has_mem_we     (r_instr_has_mem_we),
        .halt           (w_halt),
        .intr           (w_intr),
        .has_lwex       (r_instr_has_lwex),
        .has_swex       (r_instr_has_swex)
    );

    // Software registers
    vmicro16_regs # (
        .CORE_ID    (CORE_ID),
        .CELL_WIDTH (`DATA_WIDTH)
    ) regs (
        .clk        (clk),
        .reset      (reset),
        // async port 0
        .rs1        (r_reg_rs1),
        .rd1        (r_reg_rd1_s),
        // async port 1
        //.rs2        (),
        //.rd2        (),
        // write port
        .we         (r_reg_we && ~regs_use_int),
        .ws1        (r_instr_rsd),
        .wd         (r_reg_wd)
    );

    // Interrupt replacement registers
    `ifdef DEF_ENABLE_INT
    vmicro16_regs # (
        .CORE_ID    (CORE_ID),
        .CELL_WIDTH (`DATA_WIDTH),
        .DEBUG_NAME ("REGSINT")
    ) regs_intr (
        .clk        (clk),
        .reset      (reset),
        // async port 0
        .rs1        (r_reg_rs1),
        .rd1        (r_reg_rd1_i),
        // async port 1
        //.rs2        (),
        //.rd2        (),
        // write port
        .we         (r_reg_we && regs_use_int),
        .ws1        (r_instr_rsd),
        .wd         (r_reg_wd)
    );
    `endif

    // ALU
    vmicro16_alu # (
        .CORE_ID(CORE_ID)
    ) alu (
        .op         (r_instr_alu_op),
        .a          (r_instr_rdd),
        .b          (r_instr_rda),
        .flags      (r_cmp_flags),
        // async output
        .c          (r_alu_out)
    );

    branch branch_check (
        .flags      (r_cmp_flags),
        .cond       (r_instr_imm8),
        .en         (w_branch_en)
    );

endmodule
```