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 24, 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 24, 2019

Table of Contents

Chapter 1

Memory Mapping

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

1.1 Memory Map

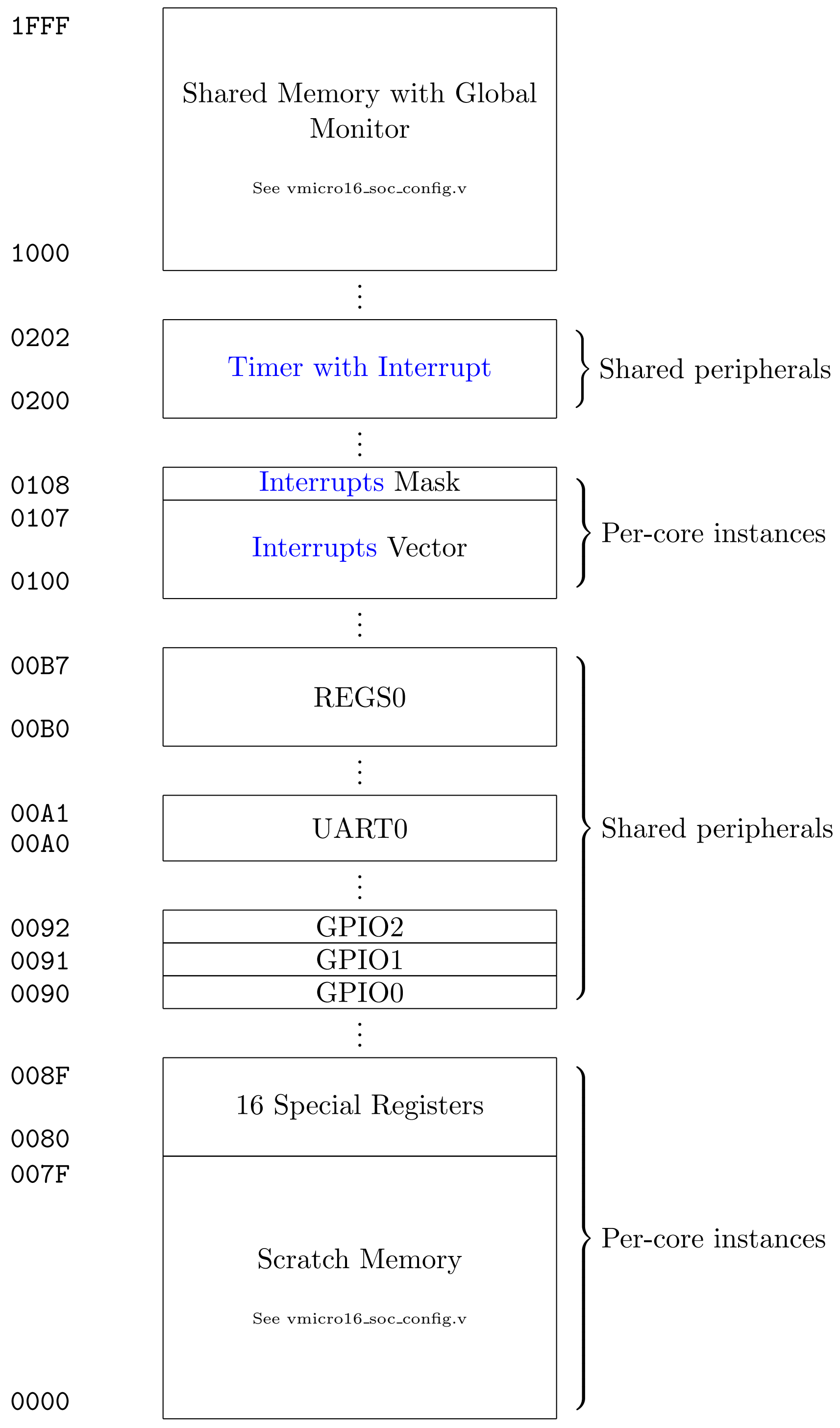


Figure 1.1: Memory map showing addresses of various memory sections.

1.2 Special Registers

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

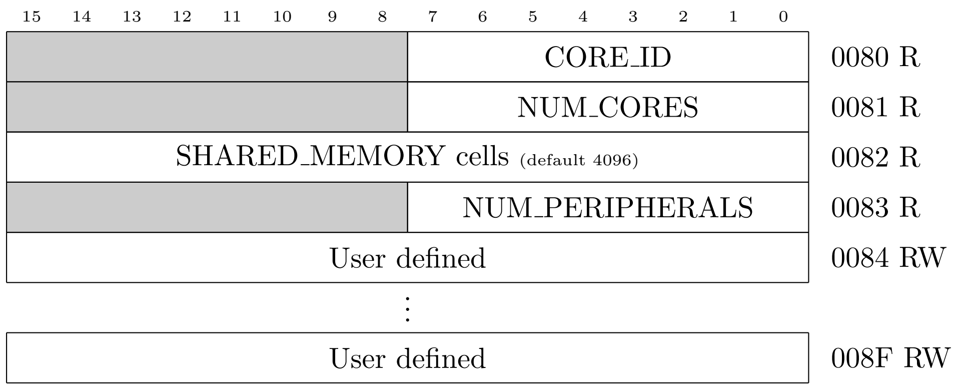


Figure 1.2: Vmicro16 Special Registers layout (0x0080 - 0x008F).

Chapter 2

Interrupts

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

2.1 Why Interrupts?

Interrupts are used to enable asynchronous behaviour within a processor.

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

2.2 Hardware Implementation

2.2.1 Context Switching

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

When saving the state, to avoid clocking 128 bits (8 registers of 16 bits) into another register (which would increase timing delays and logic elements), a dedicated register set for the interrupt mode (`regs_isr`) is multiplexed with the normal mode register set (`regs`). Then depending on the mode (identified by the register **regs_use_int**) the processor can easily switch between the two large states without significantly affecting timing.

The timing diagram in Figure 2.1 visually describes this process.

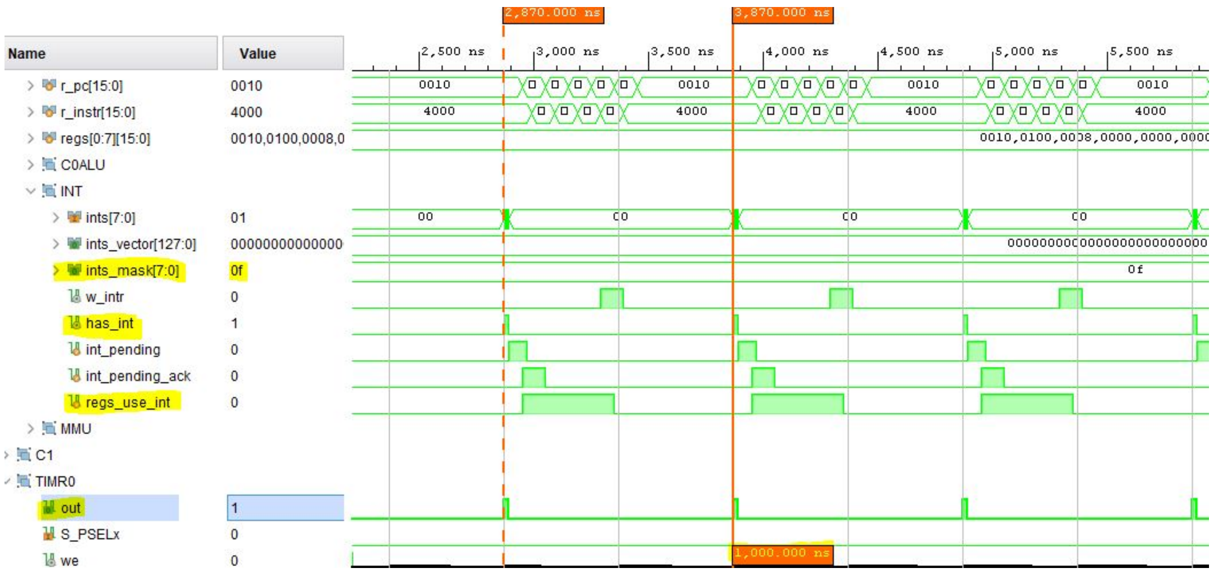


Figure 2.1: Time diagram showing the TIMR0 peripheral emitting a 1us periodic interrupt signal (out) to the processor. The processor acknowledges the interrupt (int_pending_ack) and enters the interrupt mode (regs_use_int) for a period of time. When the interrupt handler reaches the Interrupt Return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

2.3 Software Interface

To enable software to

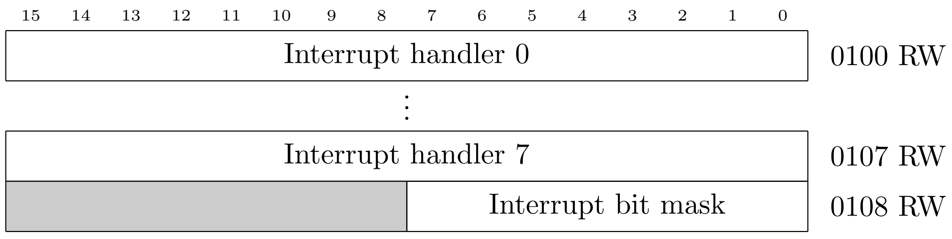


Figure 2.2: The interrupt vector consists of eight 16-bit values that point to memory addresses of the instruction memory to jump to.

2.3.1 Interrupt Vector (0x0100-0x0107)

The interrupt vector is a per-core register that is used to store the addresses of interrupt handlers. An interrupt handler is simply a software function residing in instruction memory that is branched to when a particular interrupt is received.

2.3.2 Interrupt Mask (0x0108)

The interrupt mask is a per-core register that is used to mask/listen specific interrupt sources. This enables processing cores to individually select which interrupts they respond to. This allows for multi-processor designs where each core can be used for a particular interrupt source,

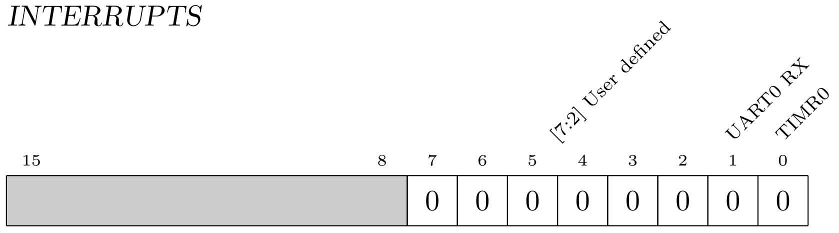


Figure 2.3: Interrupt Mask register (0x0108). Each bit corresponds to an interrupt source. 1 signifies the interrupt is enabled for/visible to the core. Bits [7:2] are left to the designer to assign.

improving the time response to the interrupt for time critical programs. The Interrupt Mask register is an 8-bit read/write register where each bit corresponds to a particular interrupt source and each bit corresponds with the interrupt handler in the interrupt vector.

2.3.3 Software Example

To better understand the usage of the described interrupt registers, a simple software program is described below. The following software program produces a simple and power efficient routine to initialise the interrupt vector and interrupt mask.

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

    // enable all interrupts by writing 0x0f to 0x108
    movi    r0, #0x0F
    sw      r0, r1 + #0x8
    halt                  // enter low power idle state

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

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

2.4 Design Improvements

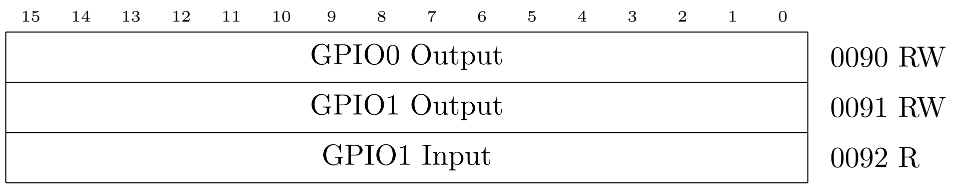
The hardware and software interrupt design have changed throughout the projects cycle. In initial versions of the interrupt implementation, the software program, while waiting for an interrupt, would be in a tight infinite loop (branching to the same instruction). This resulted in the processor using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

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

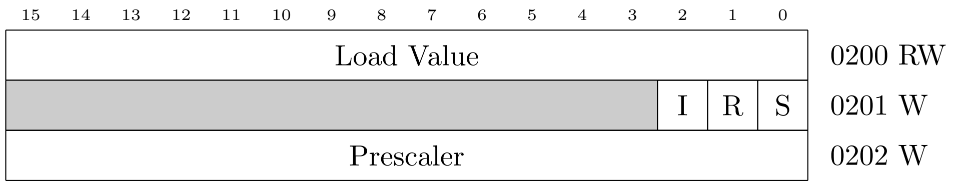
Chapter 3

Peripherals

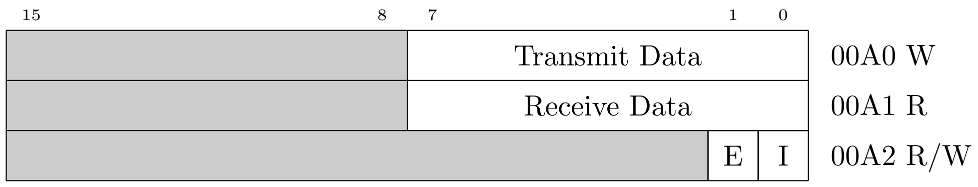
3.1 GPIO Interface



3.2 Timer with Interrupt



3.3 UART Interface



Chapter 4

System-on-Chip Layout

The Vmicro16 processor uses

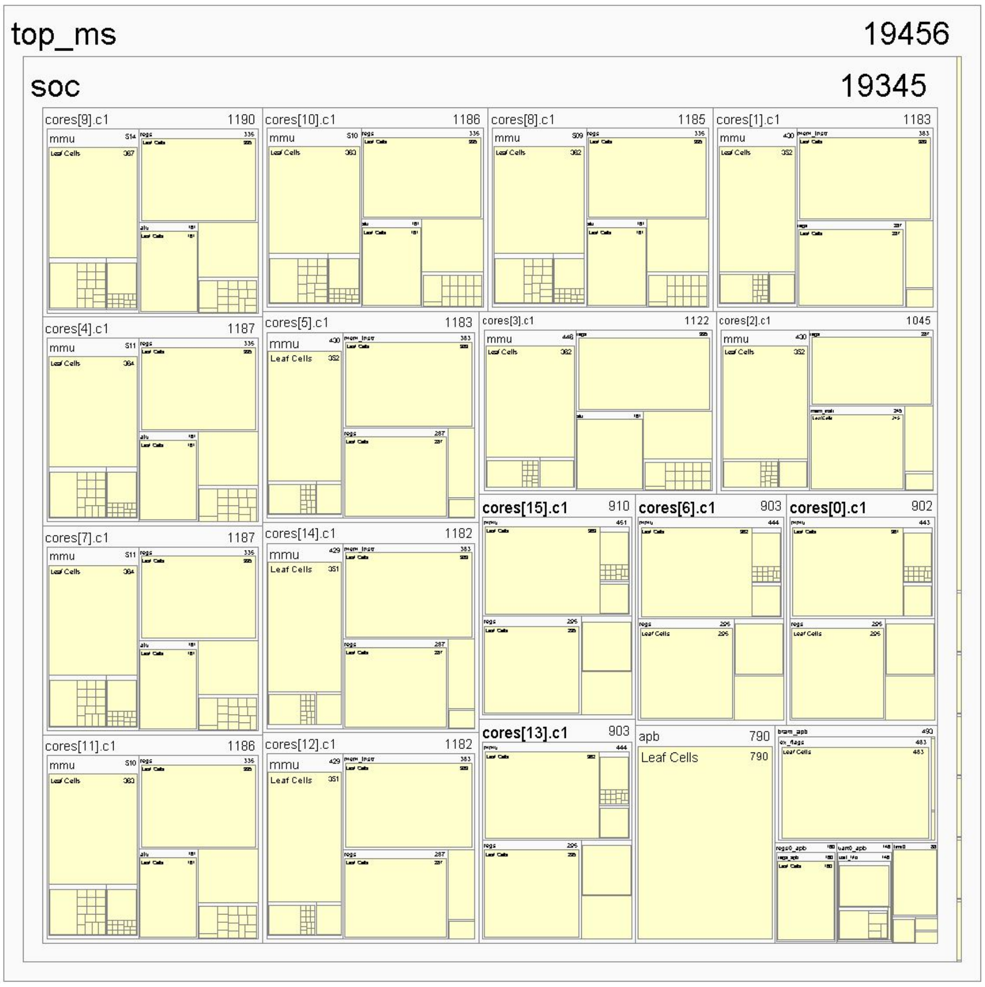


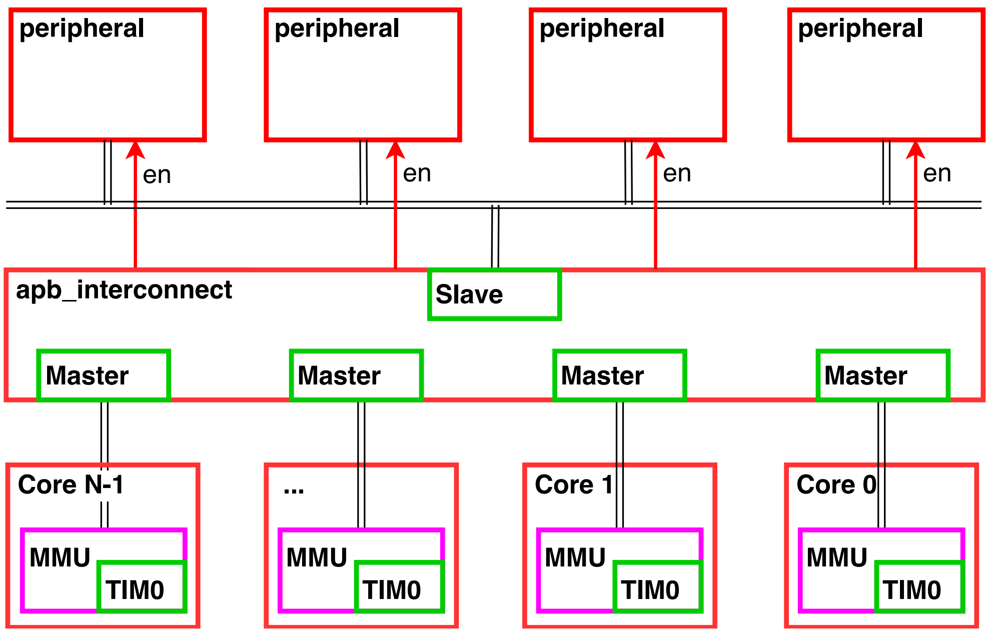
Figure 4.1:

Chapter 5

Interconnect

5.1 Introduction

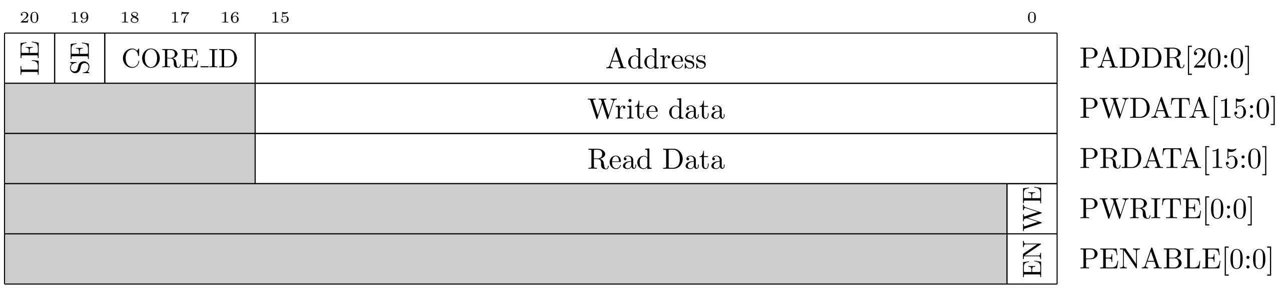
5.2 Overview



5.2.1 Design Considerations

5.3 Interconnect Interface

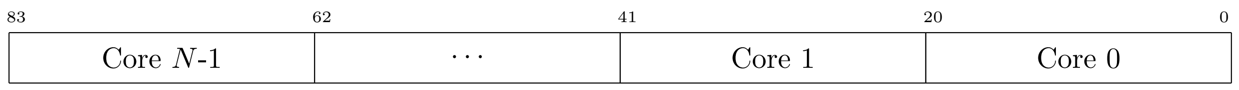
5.3.1 Master to Slave Interface



5.3.2 Variable Core Support

```
input       [MASTER_PORTS*BUS_WIDTH-1:0]  S_PADDR,
input       [MASTER_PORTS-1:0]            S_PWRITE,
input       [MASTER_PORTS-1:0]            S_PSELx,
input       [MASTER_PORTS-1:0]            S_PENABLE,
input       [MASTER_PORTS*DATA_WIDTH-1:0] S_PWDATA,
output reg  [MASTER_PORTS*DATA_WIDTH-1:0] S_PRDATA,
output reg  [MASTER_PORTS-1:0]            S_PREADY,
```

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



5.4 Shared Bus Arbitration

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

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

Chapter 6

Analysis & Results

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 top_ms.v

The top level implementation file is described here.

```verilog
//
//

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

// 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

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

    input clk_en,

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

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

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

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

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

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

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

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

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

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

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

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


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


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
);
    genvar di;
    generate for(di = 0; di < `CORES; di = di + 1) begin : gen_dbug0
        assign dbug0[di] = dbug1[di*8];
    end
    endgenerate

    wire [`CORES-1:0] w_halt;
    assign halt = &w_halt;

    // 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      (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)
    );

    vmicro16_gpio_apb # (
        .BUS_WIDTH  (`APB_WIDTH),
        .DATA_WIDTH (`DATA_WIDTH),
        .PORTS      (`APB_GPIO0_PINS),
        .NAME       ("GPIO0")
    ) gpio0_apb (
        .clk        (clk),
        .reset      (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      (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      (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      (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      (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      (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      (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          (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      (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      (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];
    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)
        always @(posedge clk) begin
            if (w_PSELx[g2])
                bus_core_times[g2] <= bus_core_times[g2] + 1;

            // Core working time
            if (!w_PSELx[g2] && !instr_w_PSELx[g2])
                if (!w_halt[g2])
                    core_work_times[g2] <= core_work_times[g2] + 1;
        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];
            fetch_time_average = fetch_time_average + instr_fetch_times[i2];
            work_time_average  = work_time_average  + core_work_times[i2];
        end

        bus_time_average   = bus_time_average   / `CORES;
        bus_reqs_average   = bus_reqs_average   / `CORES;
        fetch_time_average = fetch_time_average / `CORES;
        work_time_average  = work_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
        // Detect new reqs for each core
        assign bus_core_reqs_real[g3] = w_PSELx[g3] >
                                        bus_core_reqs_last[g3];

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

    end
    endgenerate


    ///////////////////////////////////////////////////
    // Time waiting for instruction fetches
    //   from global  memory
    ///////////////////////////////////////////////////
    reg [15:0] instr_fetch_times [0:`CORES-1];
    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)
            always @(posedge clk)
                if (instr_w_PSELx[g4])
                    instr_fetch_times[g4] <= instr_fetch_times[g4] + 1;
    endgenerate


    `endif // end FORMAL

endmodule
```