

CSE 224 - Introduction to Digital Systems

Term Project: VerySimpleCPU

Handout (Rev. 5.18)

1 Project Description

This project consists of two parts. In the first part, you are going to design VerySimpleCPU and simulate it on PC. In the second part, you are going to use your VerySimpleCPU design to implement a simple memory-mapped system which has access to switches, push buttons, LEDs on the board and also to a monitor over VGA port of the board.

The block diagram of the design of the first part of the project is given in Figure 1.

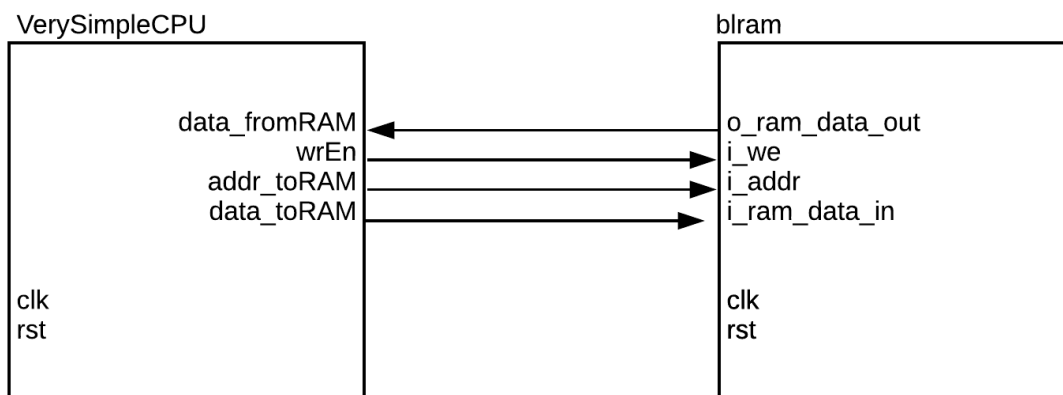


Figure 1: Top-level Block Diagram of Part I of the Project

The block diagram of the design of the second part of the project is given in Figure 2.

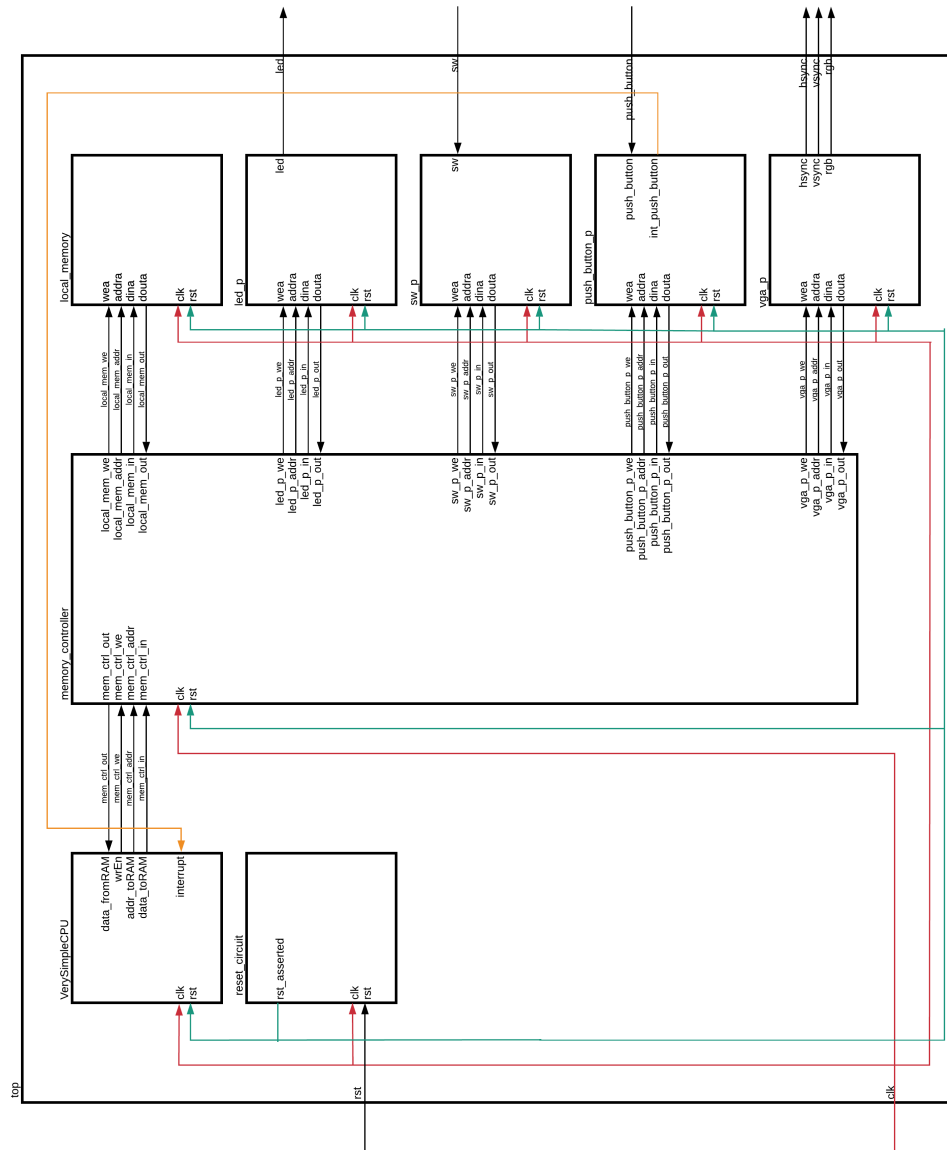
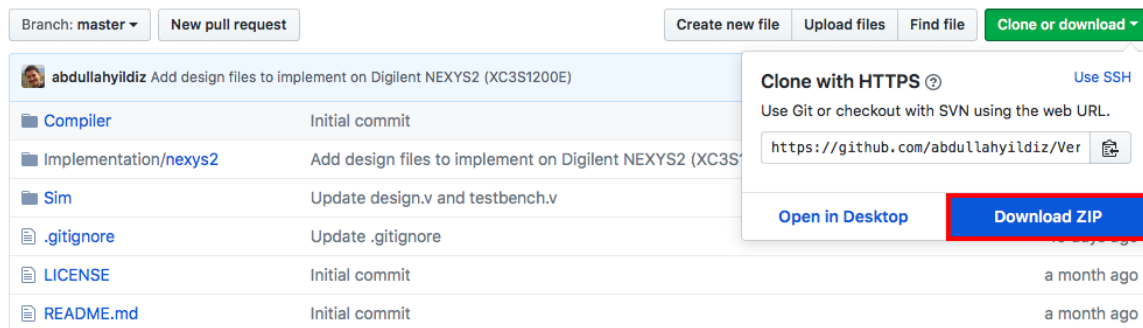


Figure 2: Top-level Block Diagram of Part II of the Project

2 How to Start

Open your web browser and go to [VerySimpleCPU](#) public repository on GitHub. Download the repository to your computer as shown in the following figure.



Extract the ZIP file you downloaded to your desktop folder. Some important files inside the VerySimpleCPU-public folder are as follows:

```

VerySimpleCPU-public
├── Compiler
│   ├── vscompiler.exe --> VerySimpleCPU C compiler
│   └── tests --> includes various C codes
├── Implementation
│   ├── nexys2 --> includes design files for Digilent NEXYS2 (XC3S1200E)
│   └── nexys4ddr --> includes design files for Digilent NEXYS4DDR (XC7A100T)
├── Sim
│   ├── vscpu-sim.exe --> VerySimpleCPU assembler
│   ├── design.v --> VerySimpleCPU black box module
│   └── testbench.v --> VerySimpleCPU testbench and memory module

```

2.1 How to Design VerySimpleCPU

An example VerySimpleCPU design that only understands **ADD** instruction is listed below (You have to implement all the instructions that can be run by VerySimpleCPU later).

```

1 module VerySimpleCPU(clk, rst, data_fromRAM, wrEn, addr_toRAM, data_toRAM);
2

```

```

3 parameter SIZE = 14;
4
5 input clk, rst;
6 input wire [31:0] data_fromRAM;
7 output reg wrEn;
8 output reg [SIZE-1:0] addr_toRAM;
9 output reg [31:0] data_toRAM;
10
11 reg [3:0] state_current, state_next;
12 reg [SIZE-1:0] pc_current, pc_next; // pc: program counter
13 reg [31:0] iw_current, iw_next; // iw: instruction word
14 reg [31:0] r1_current, r1_next;
15 reg [31:0] r2_current, r2_next;
16
17 always@(posedge clk) begin
18     if(rst) begin
19         state_current <= 0;
20         pc_current <= 14'b0;
21         iw_current <= 32'b0;
22         r1_current <= 32'b0;
23         r2_current <= 32'b0;
24     end
25     else begin
26         state_current <= state_next;
27         pc_current <= pc_next;
28         iw_current <= iw_next;
29         r1_current <= r1_next;
30         r2_current <= r2_next;
31     end
32 end
33
34 always@(*) begin
35     state_next = state_current;
36     pc_next = pc_current;
37     iw_next = iw_current;
38     r1_next = r1_current;
39     r2_next = r2_current;
40     wrEn = 0;
41     addr_toRAM = 0;
42     data_toRAM = 0;
43     case(state_current)
44     0: begin
45         pc_next = 0;
46         iw_next = 0;
47         r1_next = 0;
48         r2_next = 0;
49         state_next = 1;
50     end
51     1: begin
52         addr_toRAM = pc_current;
53         state_next = 2;
54     end
55     2: begin
56         iw_next = data_fromRAM;
57         case(data_fromRAM[31:28])
58         {3'b000,1'b0}: begin
59             addr_toRAM = data_fromRAM[27:14];
60             state_next = 3;
61         end
62         default: begin
63             pc_next = pc_current;
64             state_next = 1;
65         end
66     endcase
67 end
68     3: begin
69         r1_next = data_fromRAM;
70         addr_toRAM = iw_current[13:0];
71         state_next = 4;
72     end
73     4: begin
74         case(iw_current[31:28])
75         {3'b000,1'b0}: begin
76             wrEn = 1;
77             addr_toRAM = iw_current[27:14];

```

```

78         data_toRAM = data_fromRAM + r1_current;
79         pc_next = pc_current + 1'b1;
80         state_next = 1;
81     end
82 endcase
83 end
84 endcase
85 end
86
87 endmodule

```

2.2 How to Write Assembly Program for VerySimpleCPU

To test whether your design works, first you need to write a program in assembly language of VerySimpleCPU. This program should include one or more instructions. Each line in a VerySimpleCPU assembly program starts with a number and a colon (:) indicating the memory address for either instruction or data, then an instruction or data follows. For example, a program that reads two numbers from memory and then write their sum to memory could be written as follows:

```

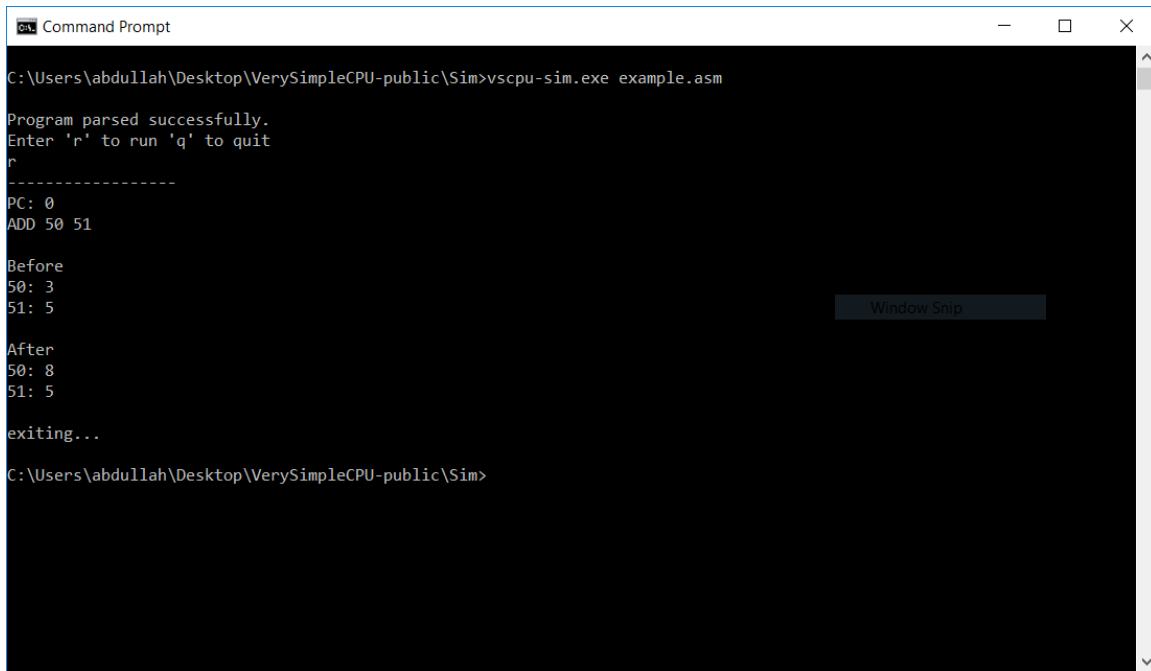
1 0: ADD 50 51 // Read data at 50th and 51st locations in memory, and write their sum
   to 50th location in memory
2 50: 3
3 51: 5

```

2.3 How to generate machine code for VerySimpleCPU

2.3.1 1st method

To generate machine code representation of your assembly program, create a file (e.g., `example.asm`) under Sim folder with the content above, open a Windows command prompt, navigate to the Sim folder and then run `vscpu-sim.exe` as follows:



```

C:\Users\abdullah\Desktop\VerySimpleCPU-public\Sim>vscpu-sim.exe example.asm

Program parsed successfully.
Enter 'r' to run 'q' to quit
r
-----
PC: 0
ADD 50 51

Before
50: 3
51: 5

After
50: 8
51: 5

exiting...

C:\Users\abdullah\Desktop\VerySimpleCPU-public\Sim>

```

Notice that `program.v` is created under the same folder with the content below. This file includes initial memory contents of your design in 32-bit hexadecimal notation.

```

1 memory[0] = 32'hc8033;
2 memory[50] = 32'h3;
3 memory[51] = 32'h5;

```

After that copy the content of `program.v` (which includes the machine code representation of your program) into initial block of blram module within `testbench.v`.

```

1 module blram(clk, rst, i_we, i_addr, i_ram_data_in, o_ram_data_out);
2
3 parameter SIZE = 10, DEPTH = 1024;
4
5 input clk;
6 input rst;
7 input i_we;
8 input [SIZE-1:0] i_addr;
9 input [31:0] i_ram_data_in;
10 output reg [31:0] o_ram_data_out;
11
12 reg [31:0] memory[0:DEPTH-1];
13
14 always @(posedge clk) begin
15     o_ram_data_out <= #1 memory[i_addr[SIZE-1:0]];
16     if (i_we)
17         memory[i_addr[SIZE-1:0]] <= #1 i_ram_data_in;
18 end
19
20 initial begin
21     ///////////////////////////////////
22     // write BRAM content here

```

```

23 //////////////////////////////////////////////////
24 end
25
26
27 endmodule

```

2.3.2 2nd method

You can also generate machine code representation of your assembly program on your web browser. To do that, open your web browser and go to cpu.tc/simulator_py web page. Paste your assembly code into **Input ASM Code** pane and then click on **Simulate** button. After waiting a couple of seconds, **Output Execution** pane will display a full run of simulation and **Output Initial Memory** pane will display initial memory contents of your design in 32-bit hexadecimal notation. After that copy the content of **Output Initial Memory** pane (which includes the machine code representation of your program) into initial block of blram module within **testbench.v**.

Input ASM Code

```

1 0: ADD 50 51
2 50: 3
3 51: 5

```

```

Output Execution
1
2 Starting simulation
3
4 run all mode selected
5 *****
6     current_instruction: ADD 50 51
7     program counter    : 0
8     Memory content before executing instruction
9     mem[ 50 ] : 3
10    mem[ 51 ] : 5
11    Memory content after executing instruction
12    mem[ 50 ] : 8
13    mem[ 51 ] : 5
14 *****
15 Finishing simulation
16 Extracting memory initialization files
17

```

```


Output Initial Memory
1 memory[0] = 32'hc8033;
2 memory[50] = 32'h3;
3 memory[51] = 32'h5;
4

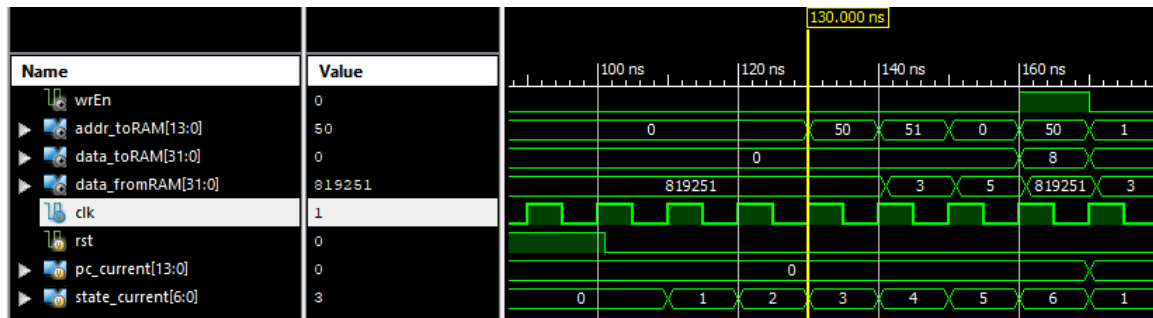
```

2.4 How to Simulate VerySimpleCPU

After you designed VerySimpleCPU and wrote an assembly code to run on it, you have to verify that VerySimpleCPU is functioning correctly. The best way to do that is to perform simulation.

To verify that your VerySimpleCPU design runs properly, create a project in Xilinx ISE


and add `testbench.v` and `design.v` files into this project. Switch to Simulation view ( Simulation) and then run the simulation. You can either eyeball the waveform or check the simulation console whether your VerySimpleCPU design works as expected (make sure that you edited `testbench.v` as needed).



3 Project Part I: Design and Simulation of VerySimpleCPU

In this part, you are asked to design a working VerySimpleCPU and verify it by simulating on PC on two programs.

3.1 Six factorial

You can use the following code to test your VerySimpleCPU design. This code calculates 6! and writes the result which is 720 to 101st address in memory. To do that, first generate the corresponding memory initialization file for this program and then create a project in Xilinx ISE and add testbench.v and design.v (including your VerySimpleCPU design and memory) files into this project. Switch to Simulation view ( Simulation) and then run the simulation. You can either eyeball the waveform or check the simulation console to see whether your VerySimpleCPU design works as expected (make sure that you edited testbench.v as required).

```


1 0: NAND 69 69 // 1's complement of 1
2 1: ADDi 69 1 // 2's complement of 1 [69: -1]
3 2: CPIi 70 100 // loc 1000 -> n
4 3: ADD 100 69 // n - 1
5 4: CP 71 100 // 71 -> n - 1
6 5: LTi 71 1 // n - 1 < 1
7 6: ADDi 70 1 // loc 1000 ++
8 7: BZJ 72 71 // if n-1 > 1 go to loc 2 (72:2) else go to loc 8
9 8: ADD 70 69 // (1000 + n) - 1
10 9: CPI 74 70 // 74 -> # go to loc (1000 + n) - 1
11 10: CP 75 70 // 75:loc (1000 + n) - 1
12 11: ADD 75 69 // [(1000 + n) - 1] --
13 12: CPI 76 75 // 76: # at loc [(1000 + n) - 1] --
14 13: MUL 76 74 // 1.2 --> 2.3 --> 6.4 --> (n-1)!.n
15 14: CP 74 76 // save result
16 15: CP 77 75 // 77: loc loc (1000 + n) - 1
17 16: LTi 77 1001 // check that all n numbers multiplied
18 17: BZJ 73 77 // if 1000 loc stop else continue
19 18: CP 101 76 // copy result to loc 101
20 19: BZJi 20 19 // infinite loop END
21 20: 0
22 69: 1 // [-1]
23 70: 1000 // Stack start #
24 72: 2 // first loop location
25 73: 11 // second loop location
26 100: 6 // INPUT n
27 101: 0 // RESULT
28 999: 1

```

Alp Kaan Erer
RecursiveFactorial

3.2 Maximum of ten numbers

Write a VerySimpleCPU assembly program that finds the maximum of ten numbers which are consecutively stored in memory starting at address 500. Your program will start at zero and write the maximum number to address 600. After writing the assembly program, generate the corresponding memory initialization file for this program and then create a project in Xilinx ISE

and add testbench.v and design.v (including your VerySimpleCPU design and memory) files into this project. Switch to Simulation view ( Simulation) and then run the simulation. You can either eyeball the waveform or check the simulation console to see whether your VerySimpleCPU design works as expected (make sure that you edited testbench.v as required).

4 Design and Implementation of a Memory-Mapped System for VerySimpleCPU

4.1 Memory-Mapping in VerySimpleCPU

A memory-mapped system employs the address space so that both memory and peripherals can be accessed by the processor (i.e., VerySimpleCPU). Hence, when a specific address is accessed by VerySimpleCPU using the same instruction set, it can refer to either the physical memory or peripherals like switch, push button, LED etc. Here, your VerySimpleCPU design will be integrated to an existing memory-mapped system which has access to switch, push button, LED and VGA peripherals. Figure 3 shows which peripherals will be used within this memory-mapped system.

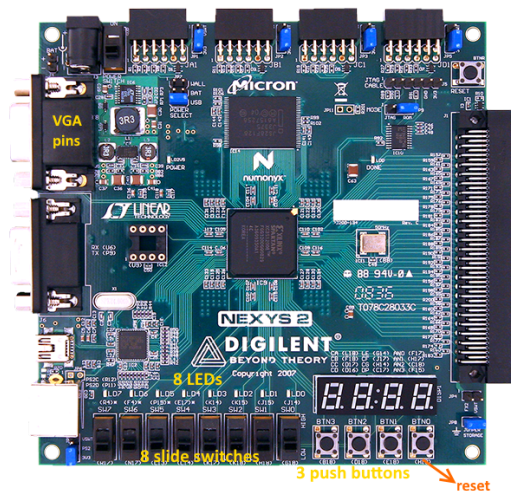


Figure 3: Overview of Memory-Mapped Peripherals on Digilent NEXYS2 Board

Following figure shows the memory map of physical memory and peripherals of this system with their corresponding start and finish addresses.

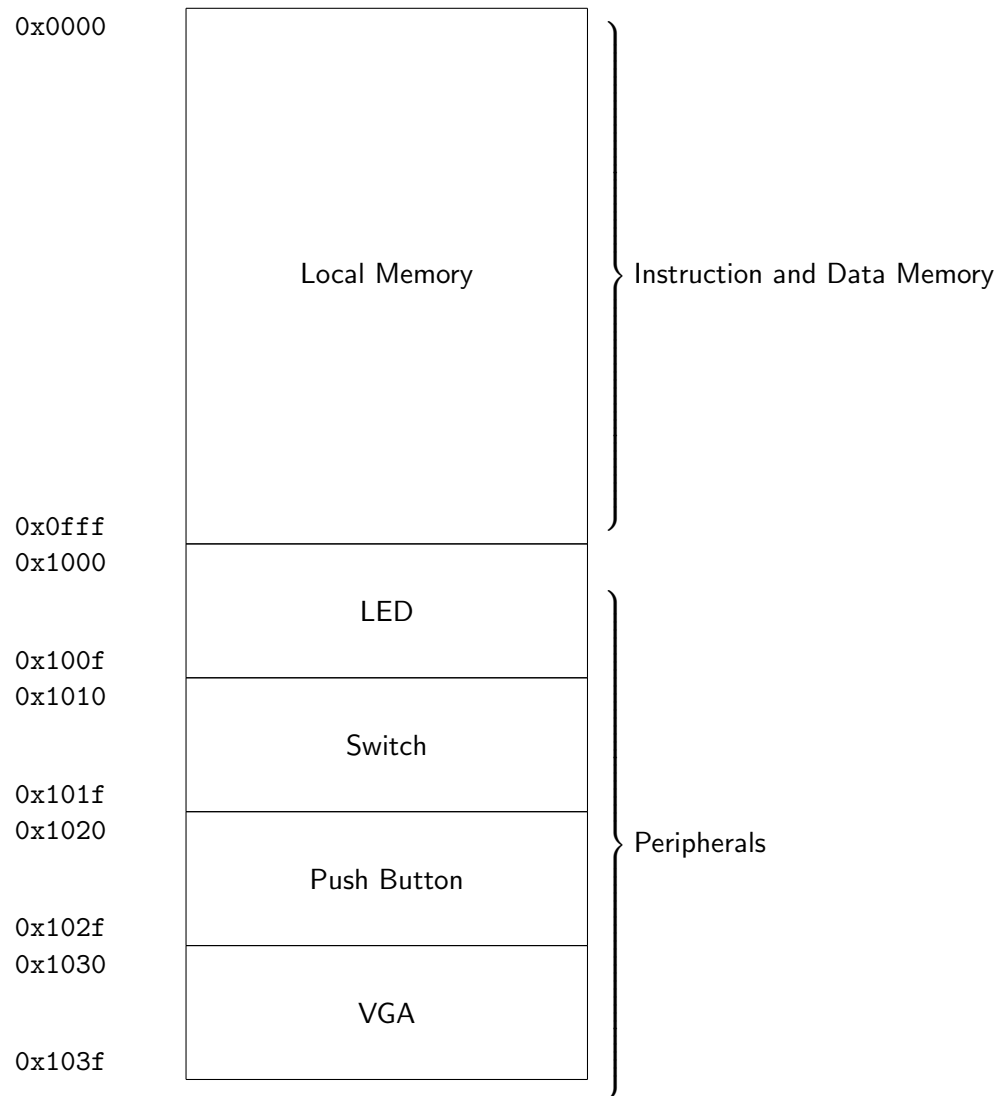


Figure 4: Memory Map of VerySimpleCPU

As shown from the figure above, there are five sections which are accessible by VerySimpleCPU within the memory map of this system. Peripherals like LED, switch, and push button are general-purpose input-output (GPIO) elements and can be easily controlled by accessing their corresponding addresses within memory. However, the case to control a VGA device in a memory-mapped fashion is different than accessing and controlling a GPIO device and it requires a more complex mechanism to interact with. Following sections describe how to access and control these peripherals.

4.2 Design of Local Memory

The first section (0x0000-0x0fff) is reserved for **Local Memory** which is the instruction and data memory of VerySimpleCPU as in part I. Notice that, the size of this section is 4096x32-bit which is exactly the same size as blram used in part I. Hence, each program you write for part I should also work for part II.

The listing below shows how local memory is designed for this memory-mapped system.

```
1 module local_memory(  
2  
3     input clk,  
4     input wea,  
5     input [11:0] addra,  
6     input [31:0] dina,  
7     output [31:0] douta  
8  
9 );  
10  
11 instr_data_memory_v1 inst_instr_data_memory(  
12  
13     .clka(clk), // input clka  
14     .wea(wea), // input [0 : 0] wea  
15     .addra(addra), // input [11 : 0] addra  
16     .dina(dina), // input [31 : 0] dina  
17     .douta(douta) // output [31 : 0] douta  
18  
19 );  
20  
21 endmodule
```

Notice that a module named **instr_data_memory_v1** is instantiated within **local_memory** module. You need to create this module by using the IP CORE Generator & Architecture Wizard of Xilinx ISE (The details are explained later in this chapter).

4.3 Design of GPIO Peripherals

4.3.1 Design of LED Peripheral

The second section (0x1000-0x100f) is reserved for **LED** peripheral. As shown in Figure 5, the size of this section is 16x32-bit which means you can access and drive 512 distinct LEDs from your program.

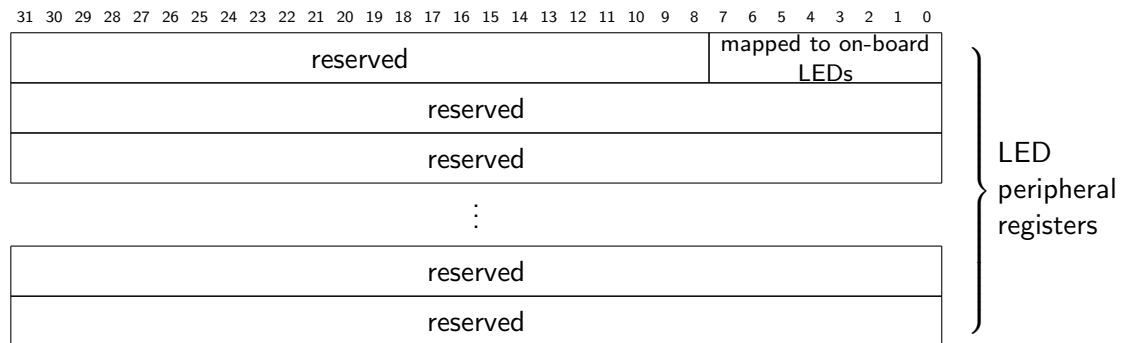


Figure 5: LED Peripheral Registers

The listing below shows how LED peripheral is designed for this memory-mapped system.

```

1 module led_p(
2
3     input clk,
4     input rst,
5     input wea,
6     input [3:0] addra,
7     input [31:0] dina,
8     output reg [31:0] douta,
9     output [7:0] led
10
11 );
12
13 reg [31:0] led_p_r [0:15];
14
15 always@(posedge clk) begin
16     douta <= led_p_r[addra];
17
18     if(rst) begin
19         led_p_r[0] <= 0;
20         led_p_r[1] <= 0;
21         led_p_r[2] <= 0;
22         led_p_r[3] <= 0;
23         led_p_r[4] <= 0;
24         led_p_r[5] <= 0;
25         led_p_r[6] <= 0;
26         led_p_r[7] <= 0;
27         led_p_r[8] <= 0;
28         led_p_r[9] <= 0;
29         led_p_r[10] <= 0;
30         led_p_r[11] <= 0;
31         led_p_r[12] <= 0;
32         led_p_r[13] <= 0;
33         led_p_r[14] <= 0;
34         led_p_r[15] <= 0;
35     end
36     else if(wea) begin
37         led_p_r[addra] <= dina;
38     end
39 end
40
41 assign led = led_p_r[0];
42
43 endmodule

```

Notice that there are 16 32-bit registers (**led_p_r**) dedicated to this peripheral and only the

first eight bits of the first register could access to actual LEDs via **led** port of led_p module. Other registers are reserved for future use.

4.3.2 Design of Switch Peripheral

The third section (0x1010-0x101f) is reserved for **Switch** peripheral. As shown in Figure 6, the size of this section is 16x32-bit which means you can access and read 512 distinct switches from your program.

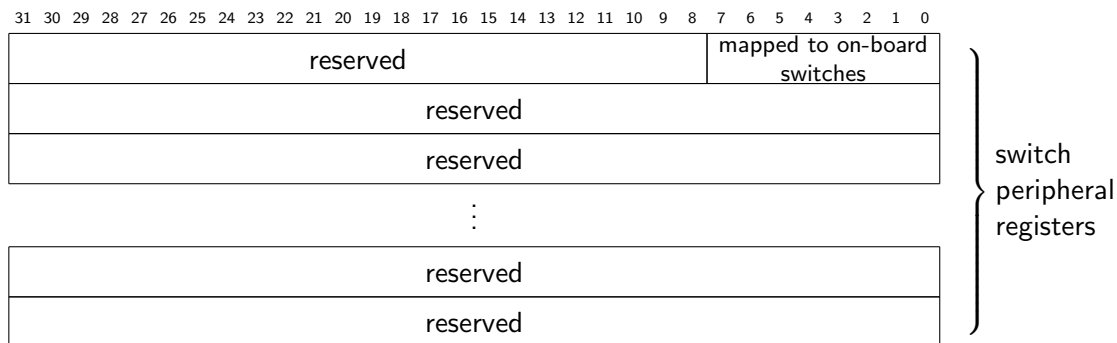


Figure 6: Switch Peripheral Registers

The listing below shows how switch peripheral is designed for this memory-mapped system.

```

1 module sw_p(
2
3     input clk,
4     input rst,
5     input [3:0] addra,
6     output reg [31:0] douta,
7     input [7:0] sw
8
9 );
10
11 reg [31:0] sw_p_r [0:15];
12
13 always@(posedge clk) begin
14
15     douta <= sw_p_r[addra];
16
17     if(rst) begin
18         sw_p_r[0] <= 0;
19         sw_p_r[1] <= 0;
20         sw_p_r[2] <= 0;
21         sw_p_r[3] <= 0;
22         sw_p_r[4] <= 0;
23         sw_p_r[5] <= 0;
24         sw_p_r[6] <= 0;
25         sw_p_r[7] <= 0;
26         sw_p_r[8] <= 0;
27         sw_p_r[9] <= 0;
28         sw_p_r[10] <= 0;
29         sw_p_r[11] <= 0;
30         sw_p_r[12] <= 0;

```



```

31     sw_p_r[13] <= 0;
32     sw_p_r[14] <= 0;
33     sw_p_r[15] <= 0;
34     end
35     else begin
36         sw_p_r[0] <= sw;
37     end
38
39 end
40
41 endmodule

```

Notice that there are 16 32-bit registers (**sw_p_r**) dedicated to this peripheral and only the first eight bits of the first register could access to actual switches via **sw** port of sw_p module. Other registers are reserved for future use.

4.3.3 Design of Push Button Peripheral

The fourth section (0x1020-0x102f) is reserved for **Push Button** peripheral. As shown in Figure 7, the size of this section is 16x32-bit which means you can access and read 512 distinct push buttons from your program.

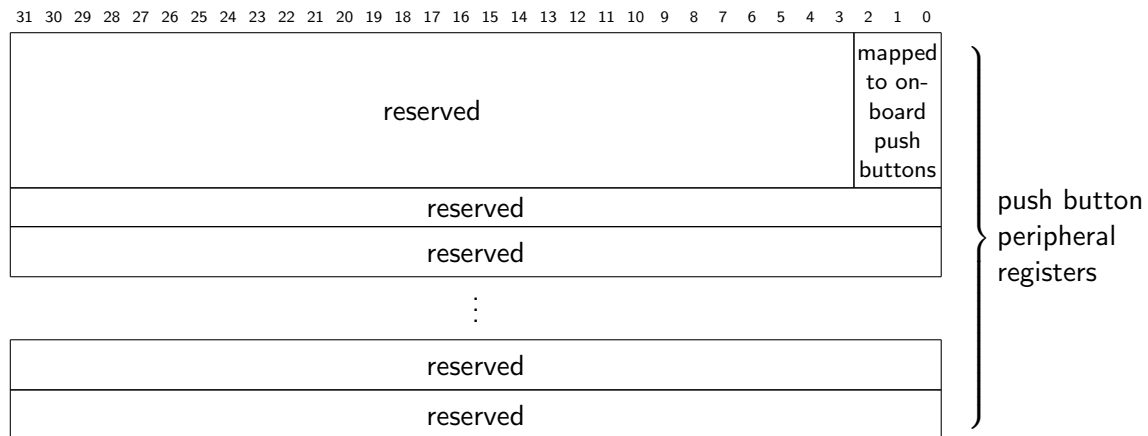


Figure 7: Push Button Peripheral Registers

The listing below shows how push button peripheral is designed for this memory-mapped system.

```

1 module push_button_p(
2
3     input clk,
4     input rst,
5     input [3:0] addra,
6     output reg [31:0] douta,
7     input [2:0] push_button,
8     output int_push_button
9

```

```

10 );
11
12 wire [2:0] db_push_button;
13 reg [31:0] push_button_p_r [0:15];
14
15 always@(posedge clk) begin
16
17     douta <= push_button_p_r[addra];
18
19     if(rst) begin
20         push_button_p_r[0] <= 0;
21         push_button_p_r[1] <= 0;
22         push_button_p_r[2] <= 0;
23         push_button_p_r[3] <= 0;
24         push_button_p_r[4] <= 0;
25         push_button_p_r[5] <= 0;
26         push_button_p_r[6] <= 0;
27         push_button_p_r[7] <= 0;
28         push_button_p_r[8] <= 0;
29         push_button_p_r[9] <= 0;
30         push_button_p_r[10] <= 0;
31         push_button_p_r[11] <= 0;
32         push_button_p_r[12] <= 0;
33         push_button_p_r[13] <= 0;
34         push_button_p_r[14] <= 0;
35         push_button_p_r[15] <= 0;
36     end
37     else begin
38         push_button_p_r[0] <= db_push_button;
39     end
40 end
41
42 debounce_inst_debounce_1(.clk(clk), .reset(rst), .db_in(push_button[0]), .db_out(
43     db_push_button[0]));
44 debounce_inst_debounce_2(.clk(clk), .reset(rst), .db_in(push_button[1]), .db_out(
45     db_push_button[1]));
46 debounce_inst_debounce_3(.clk(clk), .reset(rst), .db_in(push_button[2]), .db_out(
47     db_push_button[2]));
48
49 assign int_push_button = |(db_push_button);
50
51 endmodule

```

Notice that there are 16 32-bit registers (**push_button_p_r**) dedicated to this peripheral and only the first three bits of the first register could access to actual push buttons via **push_button** port of push_button_p module. Other registers are reserved for future use.

push_button_p module is also used to generate interrupts to VerySimpleCPU. That is, each time one of the three push buttons is pressed, an interrupt via **int_push_button signal** is sent to VerySimpleCPU. Each push button input is debounced¹ via **debounce** module in order to create a clean and stable interrupt signal to VerySimpleCPU.

4.4 Design of VGA Peripheral

The fifth section (0x1030-0x103f) is reserved for **VGA** peripheral. As shown in Figure 8, the size of this section is 16x32-bit. In this memory-mapped system, VGA peripheral is capable of

¹A debounce circuit suppresses oscillations due to mechanical and physical issues which may cause incorrect operation within the system. For more information, visit [here](#).

to drive a 640-by-480 screen with 25 MHz pixel rate via a VGA synchronization circuit. This circuit runs along with a text generator circuit which is used to construct a 80-by-30 text screen.

As known, reading/writing from/to VGA peripheral needs a different mechanism than GPIO peripherals like LED, switch, and push button. That is, there is not a one-to-one relationship between registers of VGA peripheral and actual VGA driver circuit. Instead, there must be an additional control mechanism between VGA peripheral registers and VGA driver circuit. In this memory-mapped system, a finite state machine is used for this purpose.

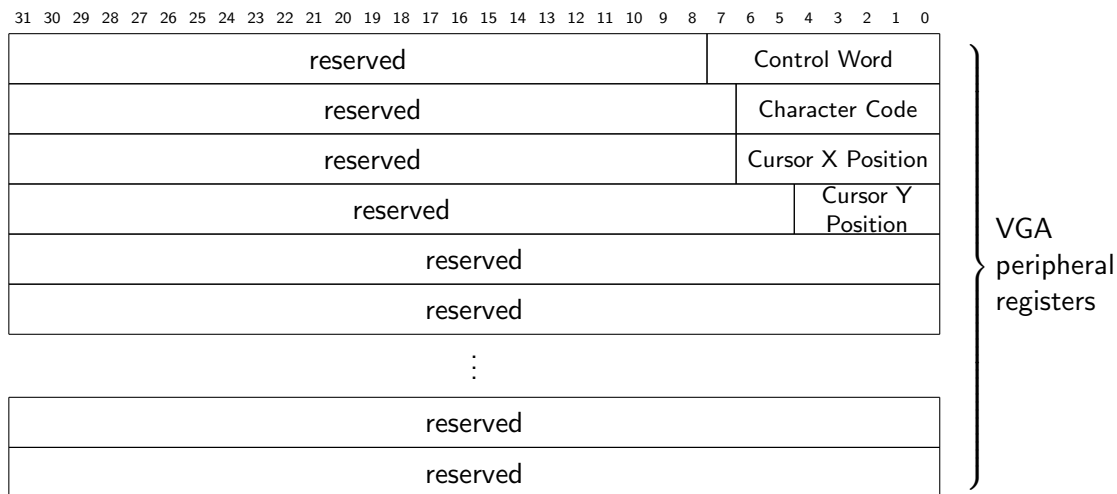


Figure 8: VGA Peripheral Registers

As shown in Figure 8, four peripheral registers are used to control the actual VGA driver circuit over the finite state machine mentioned above. These registers can be explained as follows:

- **Control Word:** controls the state machine along with the other peripheral registers.
 - Reading **1** from this register means that the state machine is idle and can accept commands written to this register.
 - Writing **3** to this register causes the screen to be cleared.
 - Writing **5** to this register causes the screen to display the character whose ASCII code is set within **Character Code** register and the cursor to be moved to the right by one unit.
 - Writing **7** to this register causes the cursor to be located at what is set within **Cursor X Position** and **Cursor Y Position** registers.
- **Character Code:** keeps ASCII code of next character which is to be sent over VGA circuit to the screen (e.g., set here to **97** to display **a** character on the screen).

- **Cursor X Position:** keeps new cursor position in x-axis (between 0 and 80).
- **Cursor Y Position:** keeps new cursor position in y-axis (between 0 and 30).

The listing below shows how VGA peripheral is designed for this memory-mapped system.

```

1 module vga_p
2 (
3
4     input clk,
5     input rst,
6     input wea,
7     input [3:0] addra,
8     input [31:0] dina,
9     output reg [31:0] douta,
10
11     output hsync,
12     output vsync,
13
14     // NEXYS2 has 8 VGA pins,
15     // namely three red, three green, and two blue pins
16     output [7:0] rgb
17     // NEXYS4DDR has 12 VGA pins,
18     // namely four red, four green, and four blue pins
19     // output [11:0] rgb
20
21 );
22
23 wire vga_ctrl_idle;
24 wire [7:0] data_received;
25 reg [31:0] vga_p_r [0:15];
26
27 always@(posedge clk) begin
28
29     douta <= vga_p_r[addra];
30
31     if(rst) begin
32         vga_p_r[0] <= 0;
33         vga_p_r[1] <= 0;
34         vga_p_r[2] <= 0;
35         vga_p_r[3] <= 0;
36         vga_p_r[4] <= 0;
37         vga_p_r[5] <= 0;
38         vga_p_r[6] <= 0;
39         vga_p_r[7] <= 0;
40         vga_p_r[8] <= 0;
41         vga_p_r[9] <= 0;
42         vga_p_r[10] <= 0;
43         vga_p_r[11] <= 0;
44         vga_p_r[12] <= 0;
45         vga_p_r[13] <= 0;
46         vga_p_r[14] <= 0;
47         vga_p_r[15] <= 0;
48     end
49     else if(wea) begin
50         vga_p_r[addra] <= dina;
51     end
52     else begin
53         vga_p_r[0] <= {31'b0, vga_ctrl_idle};
54     end
55 end
56
57
58 text_screen_top text_screen_unit(
59     .clk(clk),
60     .rst(rst),
61     .vga_cmd_word(vga_p_r[0][7:0]),
62     .vga_char_code(vga_p_r[1][6:0]),
63     .vga_cursor_x_pos(vga_p_r[2][6:0]),

```

```
64     .vga_cursor_y_pos(vga_p_r[3][4:0]),  
65     .vga_ctrl_idle(vga_ctrl_idle),  
66     .hsync(hsync),  
67     .vsync(vsync),  
68     .rgb(rgb)  
69 );  
70  
71 endmodule
```

Notice that there are 16 32-bit registers (**vga_p_r**) dedicated to this peripheral and only four of them are used as explained above. Other registers are reserved for future use. Actual access to a VGA monitor is done over **rgb**, **hsync**, and **vsync** ports of vga_p module. **text_screen_top** module includes VGA synchronization and text generator circuits along with the finite state machine which controls what is to be sent to the screen via VGA peripheral registers as explained above.

4.5 Accessing General-Purpose Input-Output (GPIO) Peripherals of Memory-Mapped System

In this section, accessing GPIO peripherals (i.e., LED, switch, push button) of the memory-mapped system from software will be explained.

4.5.1 How to Access GPIO Peripherals within Assembly Code

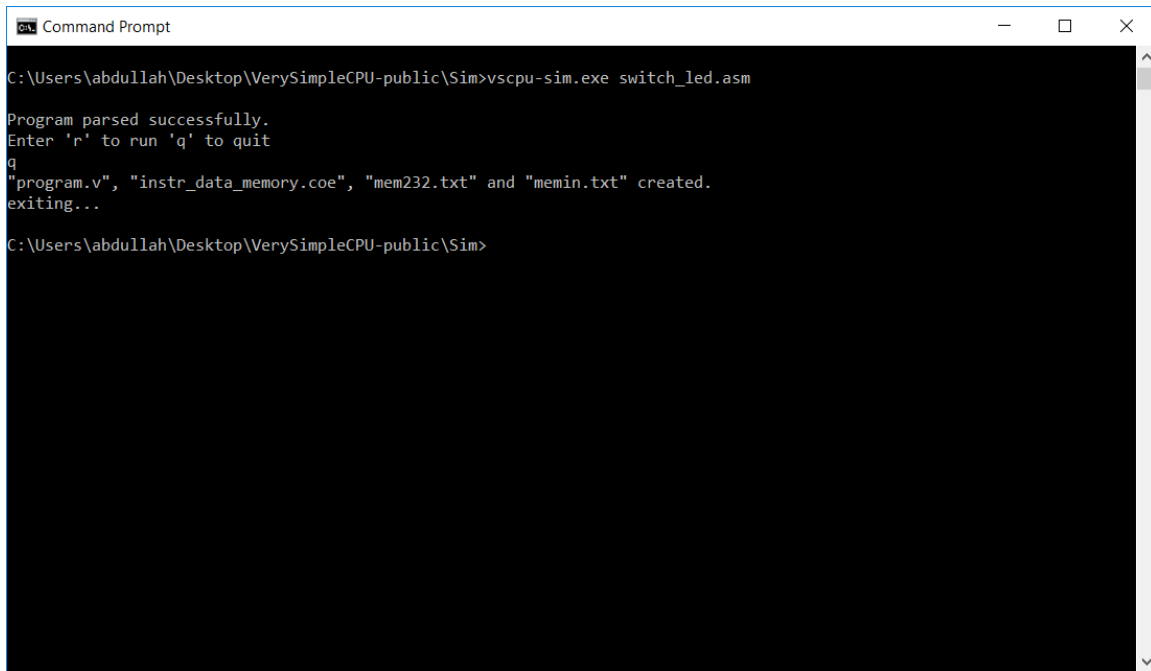
Accessing GPIO peripherals in VerySimpleCPU is the same as reading/writing from/to physical memory. That is, there is no need to add specific instructions to access these peripherals. You can use any instruction within the instruction set of VerySimpleCPU.

The listing below is the test program written in assembly language of VerySimpleCPU which continuously reads from switches and write to LEDs.

```
1 0: CP 4096 4112 // read from switches and write to LEDs
2 1: BZJi 20 0 // jump to the first instruction
3 20: 0
```

In the code above, the first instruction **CP 4096 4112** reads the data at 4112th (0x1010) address and then writes it to 4096th (0x1000) address in memory. Actually, this instruction reads the first register of the switch peripheral and writes it into the first register of the LED peripheral. Hence, the positions of switches (either zero or one) will be displayed on LEDs after executing this instruction. The next instruction **BZJi 20 0** is an unconditional branch instruction which is used to create an infinite loop. Therefore, this code will run forever and each LED will either give light or not depending on the position of its corresponding switch whenever **CP 4096 4112** instruction is executed.

To generate the machine code representation of this program, create a file (e.g., **switch_led.asm**) under Sim folder with the content above, open a Windows command prompt, navigate to the Sim folder and then run **vscpu-sim.exe** as follows:



```
Command Prompt
C:\Users\abdullah\Desktop\VerySimpleCPU-public\Sim>vscpu-sim.exe switch_led.asm

Program parsed successfully.
Enter 'r' to run 'q' to quit
q
"program.v", "instr_data_memory.coe", "mem232.txt" and "memin.txt" created.
exiting...

C:\Users\abdullah\Desktop\VerySimpleCPU-public\Sim>
```

Notice that a file named `instr_data_memory.coe` is created within the existing directory. This file defines the initial content of `instr_data_memory_v1` module which is instantiated by `local_memory` module. You will need this file later while creating your project in Xilinx ISE. The listings below show the content of the first and the last part of `instr_data_memory.coe`.

```
1 memory_initialization_radix=16;
2 memory_initialization_vector=
3 84001010,
4 d0050000,
5 0,
:
:
4095 0,
4096 0,
4097 0,
4098 0;
```

Notice that each line after the first two lines contains a value as a hexadecimal number which corresponds to a specific address of local memory. For example, the third line (hexadecimal **84001010**) is the initial value of the address zero, the fourth line (hexadecimal **d0050000**) is the initial value of the address one and so on. Remember that the local memory has 4096 32-bit storage elements and this file also has exactly 4096 lines (except the first two lines).

4.5.2 How to Access GPIO Peripherals within C code

There is a C compiler available for VerySimpleCPU under Compiler folder named `vscompiler.exe`. You can also see some example C codes under Compiler/tests folder.

The listing below is the test program written in C language which continuously reads from switches and write to LEDs.

```
1 int main(){
2
3     int *led = 4096; // start address of LED peripheral
4     int *sw = 4112; // start address of switch peripheral
5
6     while(1){
7
8         *led = *sw; // read from switches and write to LEDs
9
10    }
11
12 }
```

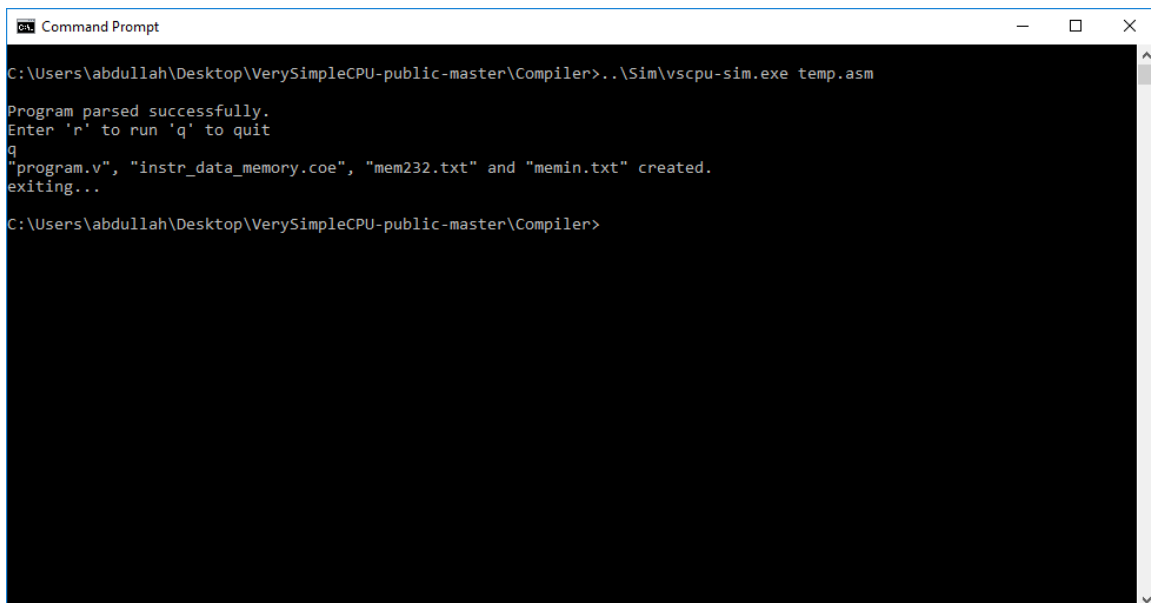
In the code above, there are two pointers **led** and **sw** which keep the start addresses of LED and switch peripherals, respectively. Hence, we can access these peripheral registers by using these pointers. To do that, there is `*led = *sw;` statement within a while loop. Actually, this statement reads the first register of the switch peripheral and writes it into the first register of the LED peripheral. Hence, the positions of switches (either zero or one) will be displayed on LEDs after evaluating this statement. As you can see, this while loop never ends and keeps the code run forever. Therefore, each LED will either give light or not depending on the position of its corresponding switch whenever `*led = *sw;` statement is evaluated.

Since the program is written in C language, first we need to compile this program and generate its corresponding assembly code. To do that, create a file (e.g., `switch_led.c`) under Compiler folder with the content above, open a Windows command prompt, navigate to the Compiler folder and then run `vscompiler.exe` as follows:



```
Command Prompt
C:\Users\abdullah\Desktop\VerySimpleCPU-public-master\Compiler>vscompiler.exe switch_led.c > temp.asm
```

Now we can create the machine code of the program. Without changing the directory, run **vscpu-sim.exe** as follows:



```
Command Prompt
C:\Users\abdullah\Desktop\VerySimpleCPU-public-master\Compiler>..\Sim\vscpu-sim.exe temp.asm
Program parsed successfully.
Enter 'r' to run 'q' to quit
q
"program.v", "instr_data_memory.coe", "mem232.txt" and "memin.txt" created.
exiting...
C:\Users\abdullah\Desktop\VerySimpleCPU-public-master\Compiler>
```

Notice that a file named **instr_data_memory.coe** is created within the existing directory. This file defines the initial content of **instr_data_memory_v1** module which is instantiated by

local_memory module. You will need this file later while creating your project in Xilinx ISE.

4.6 How to Access VGA Peripheral within C code

The listing below is the test program written in C language which displays **Hello World** on screen.

```

1 int main(){
2
3     int *control_word = 4144; // address of VGA control word register
4     int *character_code = 4145; // address of VGA character code register
5     int *cursor_x = 4146; // address of VGA cursor x position
6     int *cursor_y = 4147; // address of VGA cursor y position
7     int *led = 4096; // start address of LED peripheral
8     int *sw = 4112; // start address of switch peripheral
9     int i;
10
11     int char_array[11];
12     char_array[0] = 72; // ASCII code of 'H' character in decimal
13     char_array[1] = 101; // ASCII code of 'e' character in decimal
14     char_array[2] = 108; // ASCII code of 'l' character in decimal
15     char_array[3] = 108; // ASCII code of 'l' character in decimal
16     char_array[4] = 111; // ASCII code of 'o' character in decimal
17     char_array[5] = 32; // ASCII code of ' ' character in decimal
18     char_array[6] = 87; // ASCII code of 'W' character in decimal
19     char_array[7] = 111; // ASCII code of 'o' character in decimal
20     char_array[8] = 114; // ASCII code of 'r' character in decimal
21     char_array[9] = 108; // ASCII code of 'l' character in decimal
22     char_array[10] = 100; // ASCII code of 'd' character in decimal
23
24     while(*control_word == 0){
25         // wait for until VGA control logic is idle
26     }
27     *cursor_x = 1; // set cursor x position to one
28     *cursor_y = 5; // set cursor y position to five
29     *control_word = 7; // locate the cursor at what is set within cursor x and cursor y
        position registers
30
31     for(i=0; i<11; i++){
32         while(*control_word == 0){
33             // wait for until VGA control logic is idle
34         }
35         *character_code = char_array[i]; // set the content of VGA character code
            register as char_array[i]
36         *control_word = 5; // display the character whose ASCII code is set within VGA
            character code register
37     }
38
39     while(1){
40         *led = *sw; // read from switches and write to LEDs
41     }
42
43     return 0;
44 }
45

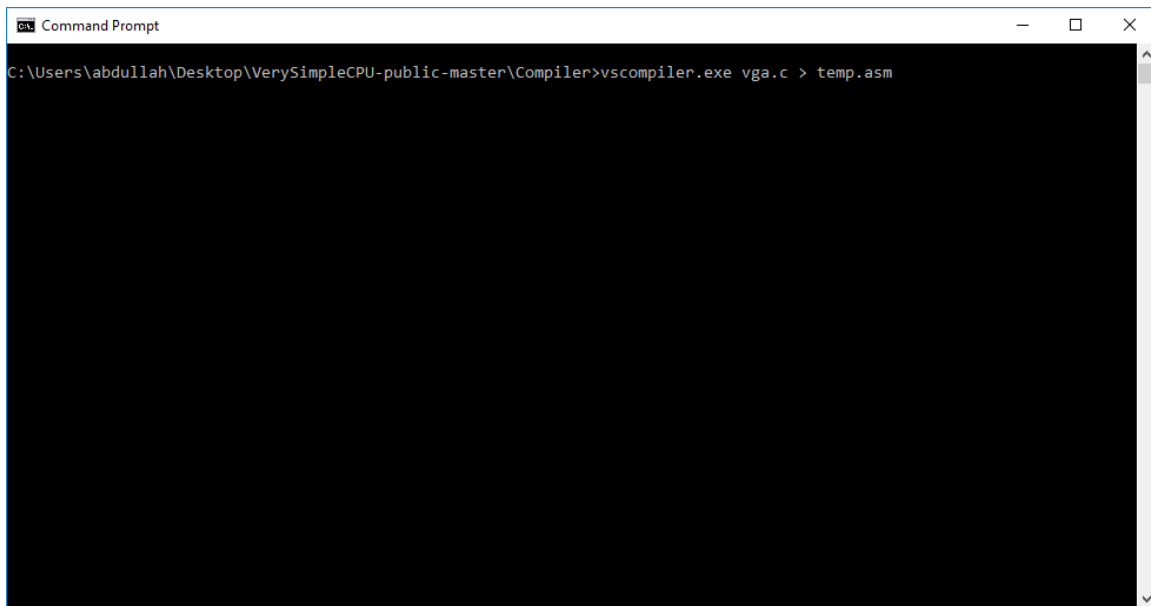
```

In the code above, there are four pointers named **control_word**, **character_code**, **cursor_x**, and **cursor_y** which keep the addresses of the first four registers of the VGA peripheral. Remember that **Control Word** register controls the VGA controller state machine along with the other peripheral registers, **Character Code** register keeps ASCII code of the next character which is to be sent over VGA circuit to the screen, **Cursor X Position** register keeps the new cursor

position in x-axis and **Cursor Y Position** register keeps the new cursor position in y-axis.


Array **char_array** within the code above keeps the ASCII codes of characters within **Hello World** string. After initializing **char_array**, we set the cursor position by giving the respective x and y coordinates to **Cursor X Position** and **Cursor Y Position** registers and set **Control Word** register to seven. Notice that we first check the state machine to be idle with `while(*control_word == 0)` statement before writing to **Control Word** register. To send a character to the screen, again we wait for the state machine be idle with `while(*control_word == 0)` statement. After that we set the content of **Character Code** register to the ASCII code of the corresponding character and **Control Word** register to five. We repeat to evaluate these three statements within a for loop for all the characters within **Hello World** string. Therefore, **Hello World** is displayed on the screen after running this part of the program. The last part of the program includes an infinite loop which continuously read from switches and write to LEDs.

Since the program is written in C language, first we need to compile this program and generate its corresponding assembly code. To do that, create a file (e.g., `vga.c`) under **Compiler** folder with the content above, open a Windows command prompt, navigate to the **Compiler** folder and then run `vscompiler.exe` as follows:



```
Command Prompt
C:\Users\abdullah\Desktop\VerySimpleCPU-public-master\Compiler>vscompiler.exe vga.c > temp.asm
```

Now we can create the machine code of the program. Without changing the directory, run `vscpu-sim.exe` as follows:



```
Command Prompt
C:\Users\abdullah\Desktop\VerySimpleCPU-public-master\Compiler>..\Sim\vscpu-sim.exe temp.asm
Program parsed successfully.
Enter 'r' to run 'q' to quit
q
"program.v", "instr_data_memory.coe", "mem232.txt" and "memin.txt" created.
exiting...
C:\Users\abdullah\Desktop\VerySimpleCPU-public-master\Compiler>
```

Notice that a file named **instr_data_memory.coe** is created within the existing directory. This file defines the initial content of **instr_data_memory_v1** module which is instantiated by **local_memory** module. You will need this file later while creating your project in Xilinx ISE.

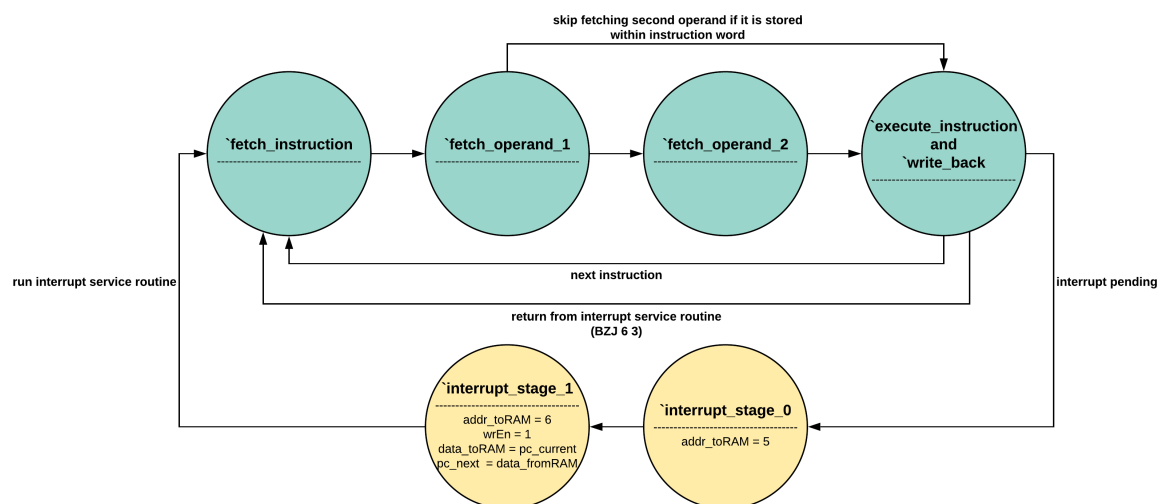
5 Interrupt Handling in VerySimpleCPU

Interrupt is a type of signal which indicates there is an event that should be handled by the processor. An interrupt can be triggered by either hardware or software.

For VerySimpleCPU, there are two special addresses (five and six) to handle interrupts. At the fifth address, we keep the address of interrupt service routine (ISR) to handle the interrupt. At the sixth address, we keep the address of the next instruction to be executed after returning from the interrupt. That is, when an interrupt is taken by VerySimpleCPU, it should take these actions respectively:

- Complete the execution of the current instruction
- Get the ISR address which is stored at fifth address in memory
- Store the address of the next instruction in sixth address in memory
- Run ISR
- Return from ISR
- Continue from where left off

Following figure shows state machine representation of VerySimpleCPU which supports interrupts.



In this memory-mapped system, we will get interrupt to VerySimpleCPU from one of the push buttons on the board. Each time a push button is pressed, an interrupt from push button module should intercept the execution of the running code and ISR should be run appropriately.

The listing below is an example program to show how interrupt could be handled in VerySimpleCPU.


```
1 0: BZJ 2 3 // main code - jump to 10th instruction
2 1: 1
3 2: 10
4 3: 0 // zero
5 4: 12
6 5: 500 // reserved to keep the address of interrupt service routine
7 6: 0 // reserved to keep the address of the next instruction when interrupt occurs
8 10: ADDi 100 1 // main code - arithmetic operation
9 11: ADDi 101 1 // main code - arithmetic operation
10 12: SRL 100 3 // main code - arithmetic operation
11 13: MULi 101 2 // main code - arithmetic operation
12 14: BZJ 4 3 // main code - jump to 12th instruction
13 500: CP 4096 4112 // START of INTERRUPT SERVICE ROUTINE - just read from switches and
    write to LEDs
14 501: BZJ 6 3 // RETURN FROM INTERRUPT
```

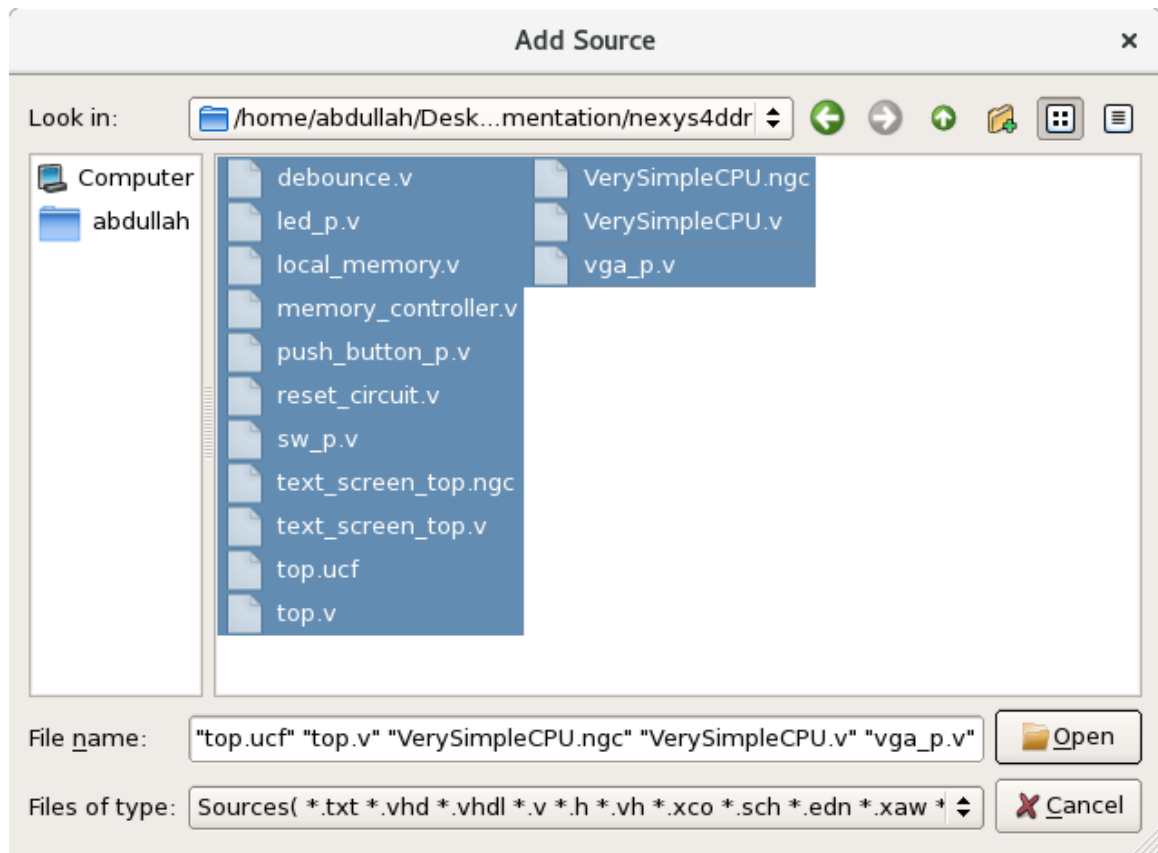
Notice that the fifth address stores the ISR address which is 500. The sixth address will be modified by VerySimpleCPU to store the address of the next instruction when interrupt occurs. In this example program, ISR consists of two instructions, i.e., **CP 4096 4112** and **BZJ 6 3**. The first instruction simply copies the status of switches to LEDs. The second instruction says to return from the interrupt and continue the program where it left off. Since VerySimpleCPU has no any built-in instruction to support returning from interrupt, every ISR should end with **BZJ 6 3** instruction.

6 How to Integrate VerySimpleCPU into the Memory-Mapped System

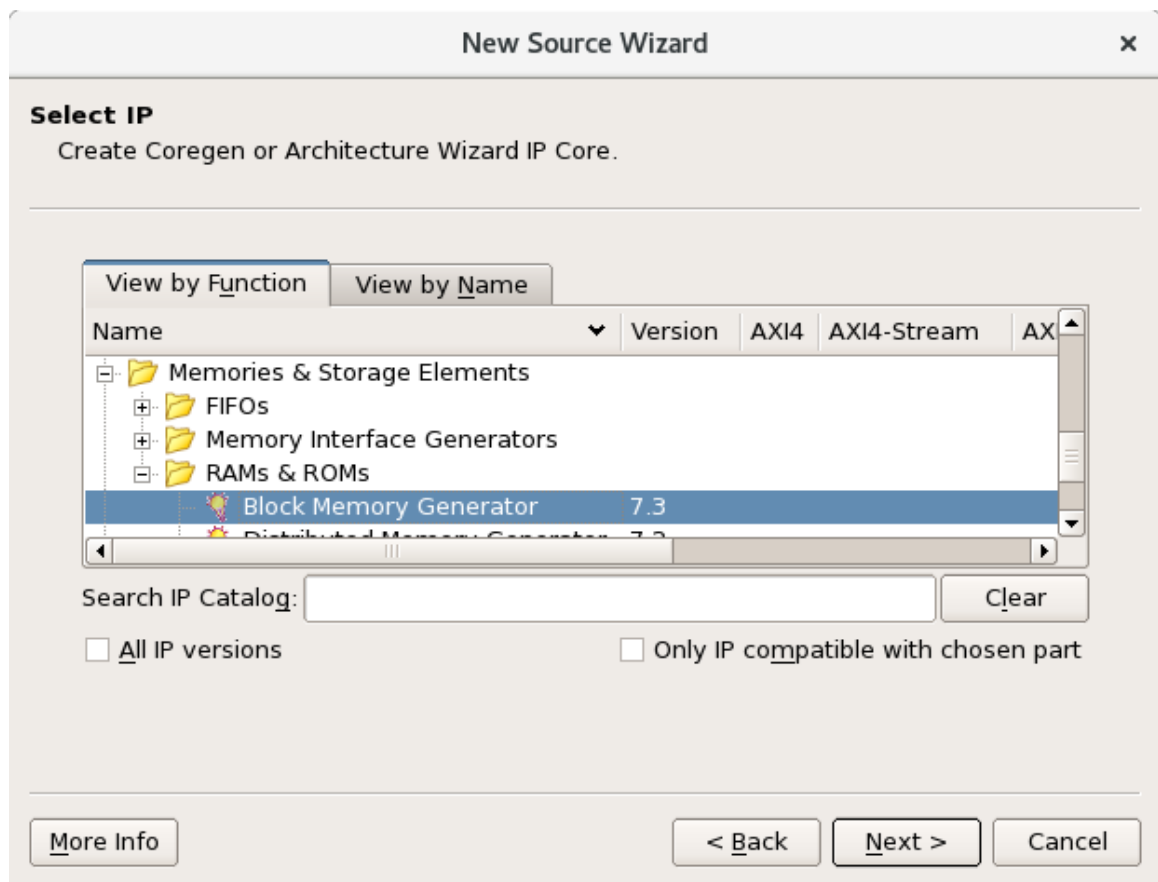
This subsection guides you through the process of implementing a memory-mapped system for VerySimpleCPU.

Perform the following steps:

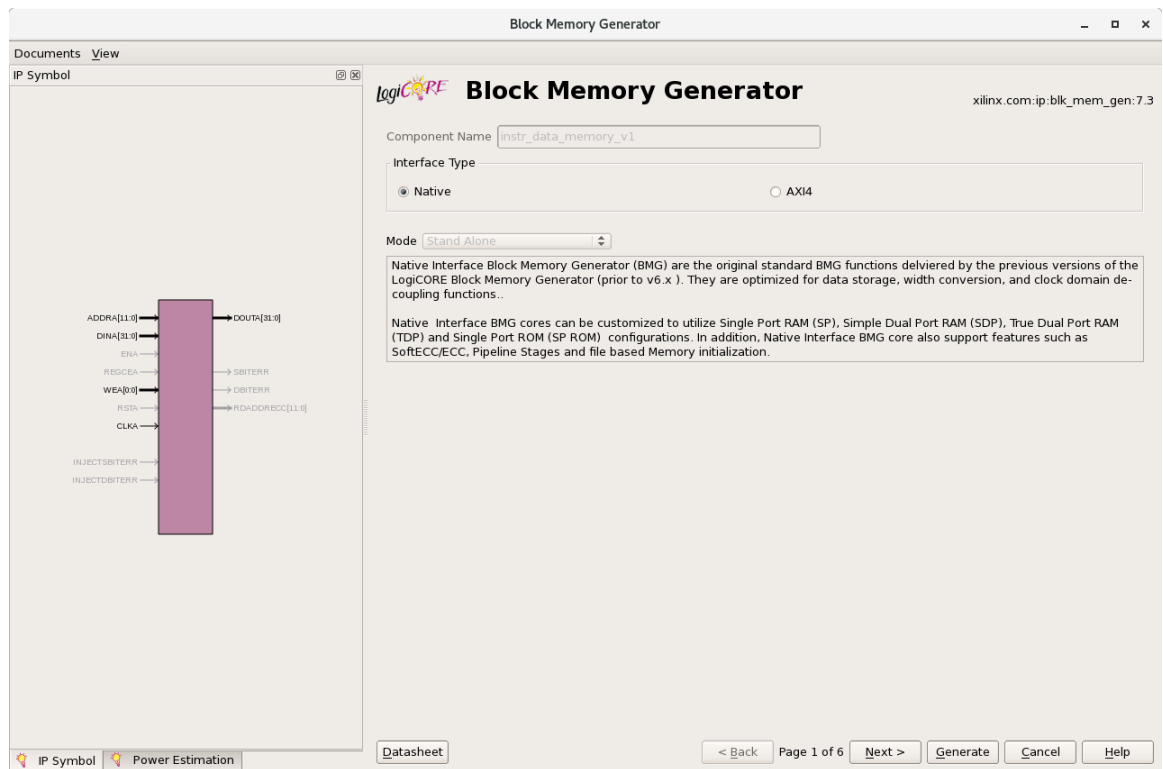
- Invoke Xilinx ISE Design Suite.
- Create a new project under Implementation folder (or in another convenient location).
- Set **Family**, **Device**, and **Package** in Project Settings window according to which board you use (NEXYS2 or NEXYS4).
- Switch to Implementation view ( Implementation)
- Click on **Add Source...** under Project menu and add **all the existing source files (including .ucf file of your FPGA chip)** under either Implementation\nexys2 or Implementation\nexys4ddr folder



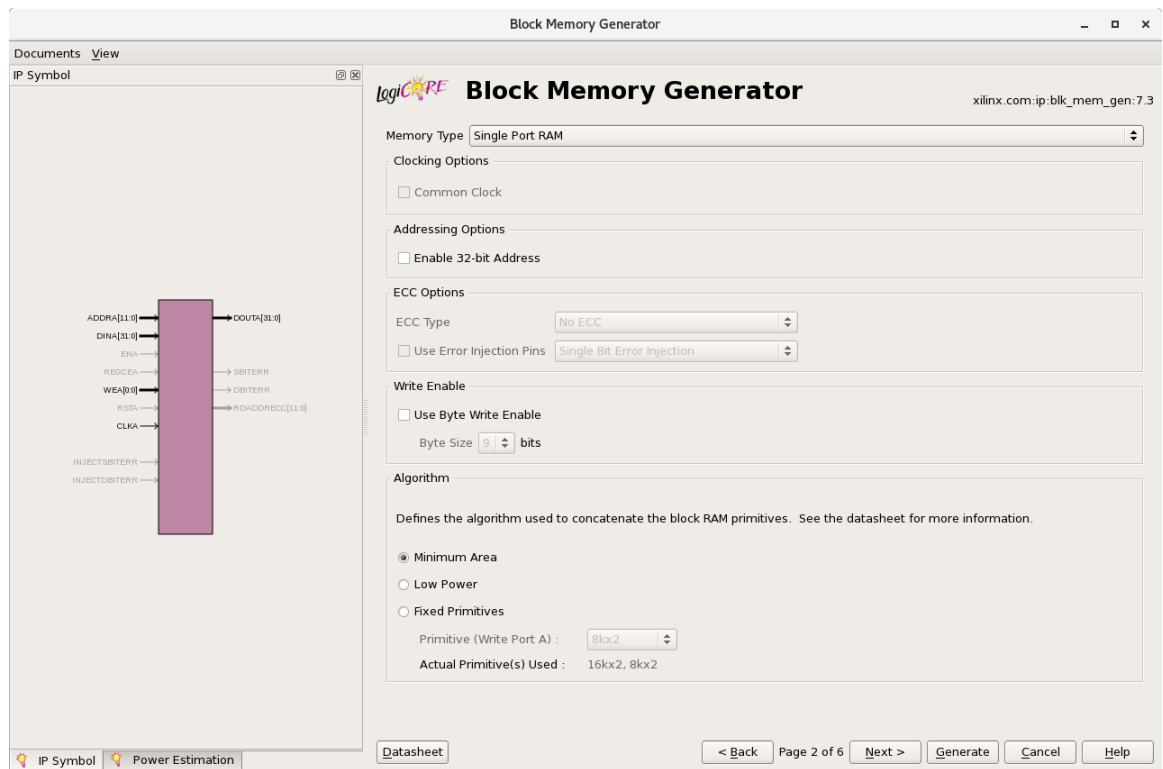
- Click on **New Source...** under Project menu.
- Select **IP (CORE Generator & Architecture Wizard)** and set the file name as **instr_data_memory_v1** in Select Source Type window and then click on Next.
- Select **Block Memory Generator** under **RAMs & ROMs** of **Memories & Storage Elements**. Click on Next and then Finish.



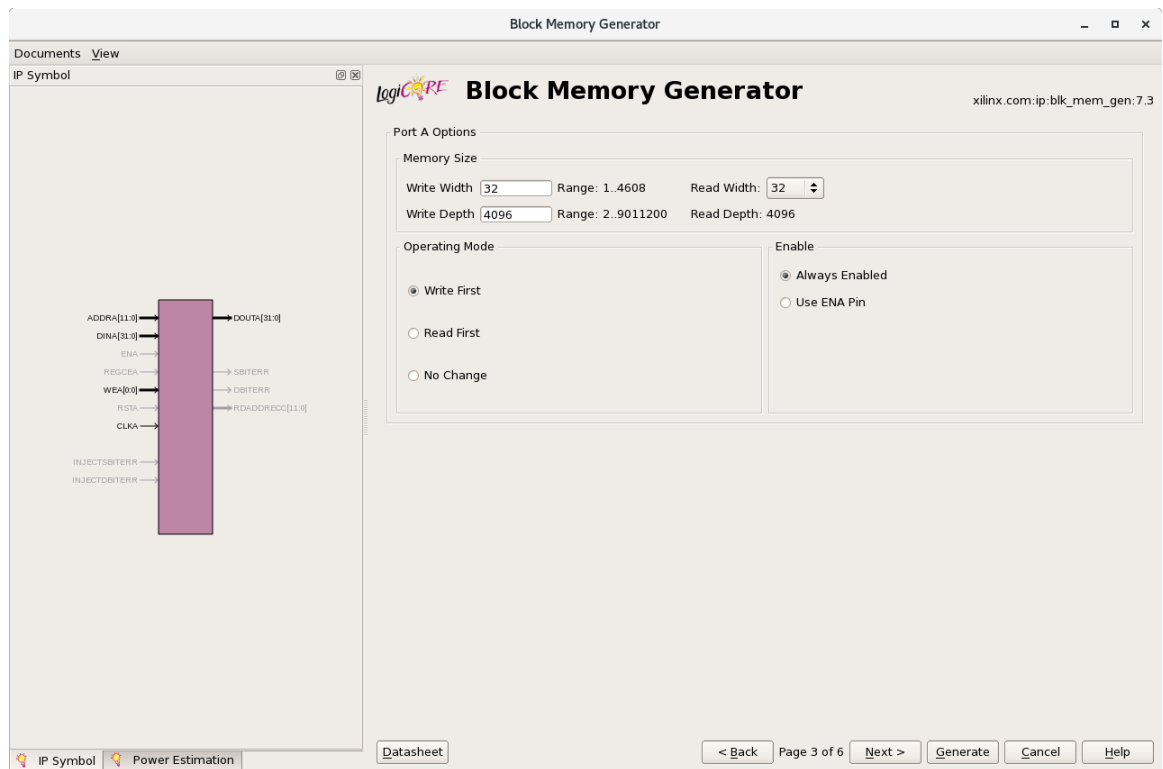
- Skip the first page by clicking on Next.



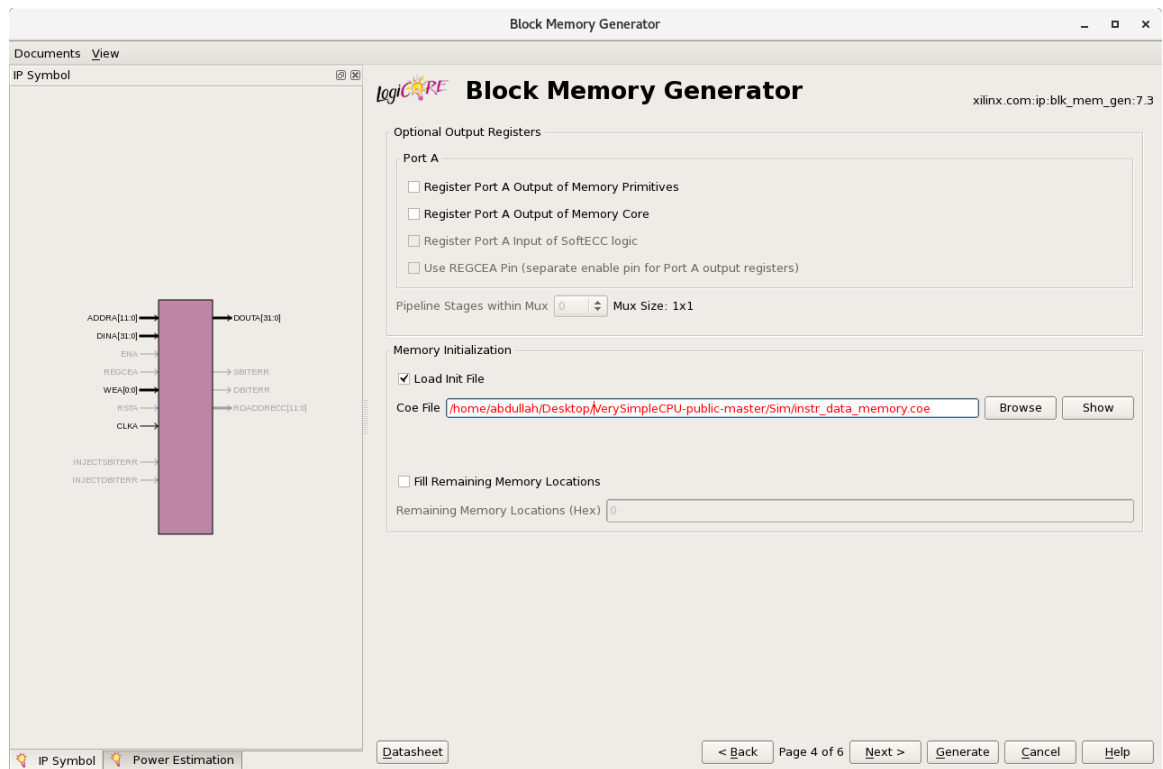
- Select **Single Port RAM** as Memory Type on the second page and click on Next.



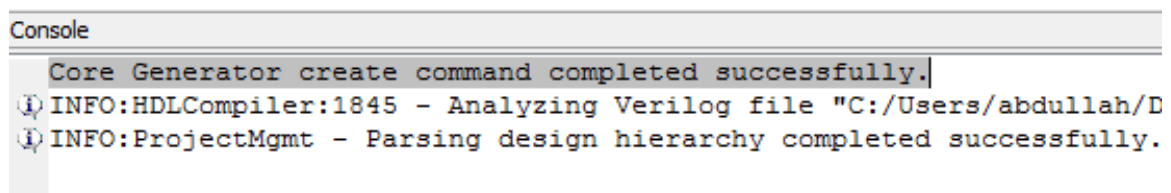
- Set **Write Width** as **32** and **Write Depth** as **4096** on the third page and then click on Next.



- Check the **Load Init File** option on the fourth page and set the path of **Coe File** (**instr_data_memory.coe**) you created before (see subsection 4.5.1 to see how to create a .coe file). Then click on **Generate**.



- Wait until you see a message as "**Core Generator create command completed successfully.**" in console window of Xilinx ISE.



- Finally, synthesize and then implement the design. Generate the programming file, download it to your board. Test your program to verify that whether peripherals are accessed properly.

7 Project Part II: Writing Code for Memory-Mapped System

In this part, you are asked to use VerySimpleCPU you designed in part I to integrate it to the memory-mapped system explained and asked to write programs (either written in assembly language of VerySimpleCPU or C) which run on this memory-mapped system. There will be three problems in total:

1. Software-based Timer
2. Interrupt Controlled Incrementer/Decrementer
3. Sort Numbers and Display Result on VGA Monitor

7.1 Design of a Software-based Timer

You are asked to write a software-based timer code in assembly language of VerySimpleCPU. This software-based timer code will count from a predetermined number (start value of the timer) to zero, and each time the timer reaches zero (i.e., a timeout occurs), the value of the first register of the LED peripheral will be incremented. This process will continue forever.

The increase period of the value of the first register of the LED peripheral will be **250 milliseconds**. Hence, you need to find the start value of the timer. To do this, you need to count the number of cycles per each instruction which are executed periodically. Consider the following example:

```
1 0: CPi 50 51
2 1: ADDi 50 1
3 2: MULi 50 1
4 3: BZJi 20 0
5 20: 1
```

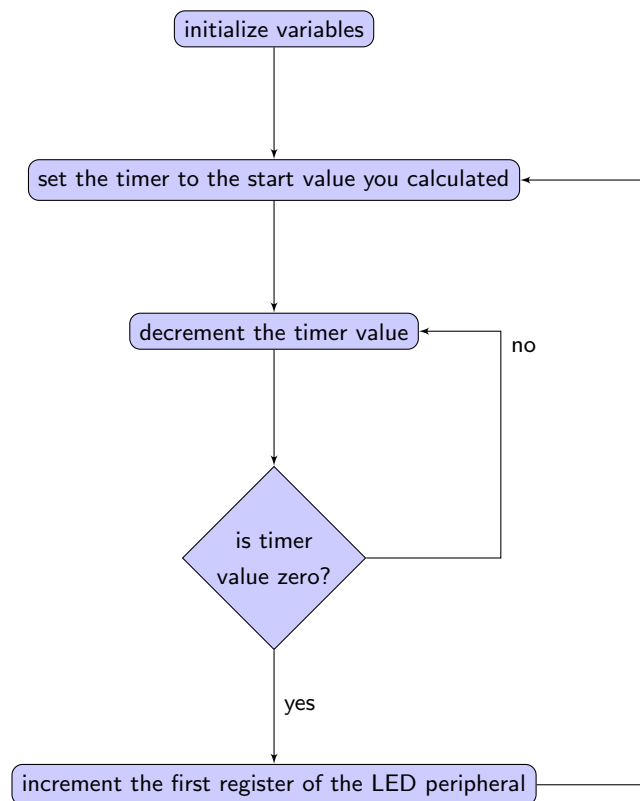
If you look at the code carefully, you will see that the second, third and the fourth instructions form an infinite loop. You can calculate how long it takes to finish the loop before the loop starts over.

You need to know the following information:

- How long does it take to execute each instruction within the loop in terms of clock cycles?
- What is the clock period?

Let's assume that all the three instructions within the loop take four cycles (states) to execute. You know from the lab sessions that the clock period of our design is 20 nanoseconds² assuming a clock frequency of 50 MHz (or 10 nanoseconds³ assuming a clock frequency of 100 MHz). So, each iteration of the loop takes $3 \times 4 \times 20 = 240$ nanoseconds (or $3 \times 4 \times 10 = 120$ nanoseconds) and if you need a 1 second delay before continuing to the next instruction after the loop, you have to run the loop for $1 \text{ second} / 240 \text{ ns} \approx 4166666$ (or $1 \text{ second} / 120 \text{ ns} \approx 8333333$) times (iterations). Similarly, you can calculate the timer start value which corresponds to a 250 milliseconds delay after writing your code.

The figure below shows the flow chart of the software-based timer. You are going to write the corresponding code of this flow chart.



7.2 Interrupt Controlled Incrementer/Decrementer

You are asked to write an interrupt controlled incrementer/decrementer code in assembly language of VerySimpleCPU. This program will include the software-based timer code you wrote in

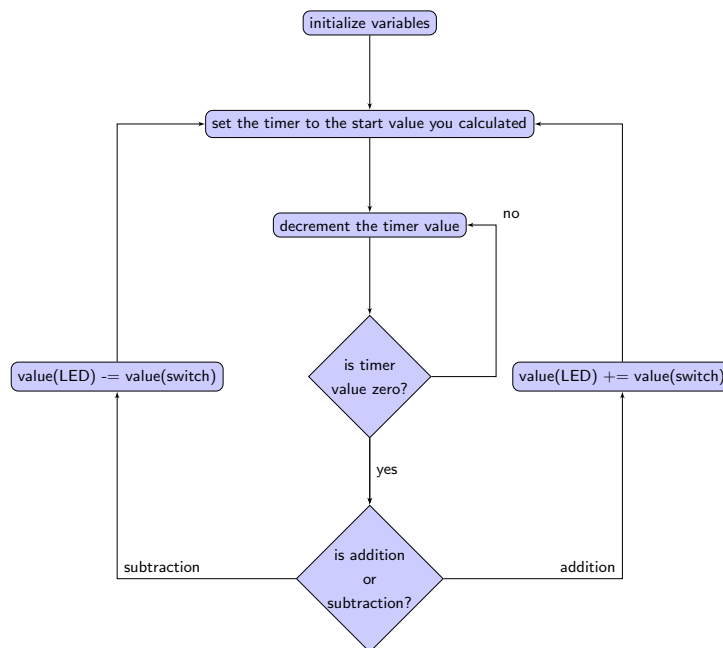
²Digilent NEXYS2 board uses an on-board clock signal whose frequency is 50 MHz.

³Digilent NEXYS4DDR board uses an on-board clock signal whose frequency is 100 MHz.

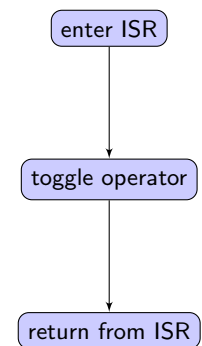
previous part as part of main routine and also another part of code as interrupt service routine (ISR). The main routine will count from the start value of the timer to zero, and each time the timer reaches zero (i.e., a timeout occurs), you will either increment or decrement the value of the first register of the LED peripheral depending on the value of the operator. This process will continue forever. The operator is just a flag which is toggled between one and zero each time an interrupt occurs. If it is one/zero, increment/decrement the value of the first register of the LED peripheral by the amount of the value of the first register of the switch peripheral.

Remember that in this memory-mapped system, we get interrupts to VerySimpleCPU from one of the push buttons on the board.

The figure below shows the flow chart of the interrupt controlled incrementer/decrementer. You are going to write the corresponding code of this flow chart.



(a) main routine



(b) ISR

7.3 Sort Numbers

You are asked to write a C program which sorts elements of an array in ascending order and then display both original and sorted array on monitor over VGA interface. Array will include 5 integers which are between 0 and 9. You can look at `Compiler/tests` folder which includes various examples written in C.