CS220A: Computer Organisation
Assignment 7

By: Akshat Gupta (200085) and Hrishant Tripathi (210449)

This is the documentation for our implementation of the processor **CSE-BUBBLE**, which follows the Instruction-set Architecture as mentioned in the problem statement. We assume that the processor word size and instruction size are 32 bits and we use the VEDA memory as implemented in Assignment 4. The VEDA memory has two parts: a) Instruction Memory, b) Data Memory.

PDS1 - Decide the registers and their usage protocol.

We have used 32 different registers, each being 32 bits wide. Their usage and labelling are as follows:

Register	Usage
0	Program Counter - Stores the address of the next instruction to be executed.
1	EPC - Stores the address of the instruction that caused an exception
2	Cause - Used to identify the cause of an exception, such as a hardware error or software interrupt.
3	BadVAddr - Stores the virtual address that caused an address-related exception
4	Status - Controls the operating mode and enable/disable interrupts, among other system-level functions
5	IR - Stores the current instruction to be executed
6	r_0 - This register is hardwired to zero at all the times.
7	at - This register is going to be used by the Assembler time to time to implement Pseudo Instructions
8-9	$(v_0 - v_1)$ - This is being used for system calls and system instructions by the user
10-13	$(a_0 - a_3)$ - It is being used to provide arguments for function or system calls by the user
14	gp - Global Pointer
15	sp - Stack Pointer
16	ra - Return Address - Will Store the address of the instruction where we have to return after function exits
17-24	$(t_0 - t_7)$ -Temporary Registers - Will be used to store values just required temporarily.
25-31	$(s_0 - s_6)$ - Stored Registers - Will be used to store values required over multiple functions or modules

Registers 0-5 are system-controlled registers including **PC, EPC, Cause, BadVAddr, Status, IR** while registers 6-31 are user controlled registers.

PDS2 - Decide upon the size for instruction and data memory in VEDA.

For both the instruction and data memory, we will be using a **VEDA memory** containing **256 registers**, each register being **32 bits** wide.

PDS3 - Design the instruction layout for R-, I- and J-type instructions and their respective encoding methodologies.

The instruction layout for R-, I- and J-type instructions can be understood broadly as follows:

R-type:

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6bits

I-type:

opcode	rs	rt	imm/address
6 bits	5 bits	5 bits	16 bits

J-type:

opcode	address
6 bits	26 bits

The ISA CSE-BUBBLE implements the following 27 instructions which are explained below –

Instruction	OPCODE	Operation	Type
add	0	Arithimatic	R
sub	1	Arithimatic	R
subu	2	Arithimatic	R
addu	3	Arithimatic	R
addi	4	Arithimatic	I
addiu	5	Arithimatic	I
and	6	Arithimatic	R
or	7	Arithimatic	R
andi	8	Arithimatic	I
ori	19	Arithimatic	I
sll	10	Arithimatic	I
srl	11	Arithimatic	I
lw	12	Data Transfer	I
sw	13	Data Transfer	I
beq	14	Branching	I
bne	15	Branching	I
bgt	16	Branching	I
bgte	17	Branching	I
ble	18	Branching	I
bleq	19	Branching	I
j	20	Branching	J
jr	21	Branching	R
jal	22	Branching	J
slt	23	Arithimatic	R
slti	24	Arithimatic	I
syscall	25	System	J
nop	26	System	J

PDS4 - Now design and implement an instruction fetch phase where the instruction next to be executed will be stored in the instruction register.

- We have written a Verilog module named **fetch-instructions.v** which is called in the main processor module named **CSE-BUBBLE.v** to initiate the **INSTRUCTION FETCH PHASE**.

- The module is called as follows in the processor:

  ```
  VEDA_instruction fetch(clk, rst, processor[0], 1'b1, 1'b0, 32'b0, instr_output);
  ```

  Here, **clk** is the clock signal, **rst** is the reset signal, **processor[0]** is the Program Counter, **1'b1** is the mode, **1'b0** is the WRITE-ENABLE signal, **32'b0** is the 32-bit 0 input being passed (since we are only reading in this case), and **instr output** is a 32-bit wire which stores the fetched instruction.

- This module reads the instructions the VEDA-Instruction memory at **each positive-edge of the clock** and relays it to the main processor where it can move forward to the following phases like INSTRUCTION-DECODE and INSTRUCTION-EXECUTE. **The instructions are initially stored in the memory at the positive edge of the reset signal**. The VEDA memory only reads when [**mode == 1**] and it writes memory when [**mode == 0 and WRITE ENABLE == 1**].

- The module is programmed as follows:

```
module VEDA_instruction(clk, rst, addr, mode, write_en, write_data, out_wire);

    input clk;                                  // clock signal
    input rst;                                  // reset signal
    input [31:0] addr;                          // input address or PC
    input write_en;                             // write enable signal
    input [31:0] write_data;                    // instruction to be written
    input mode;                                 // mode - either 1 or 0 (fetch or write)

    output signed [31:0] out_wire;              // data/instruction to be outputted or IR

    reg [31:0] veda[0:255];                     // instruction memory
    reg [31:0] out;
    integer i;

    // storing instructions
    always @(posedge rst) begin
    veda[0] <= 32'b000000_00110_00110_11001_00000_000000;
    ...
    ...
    veda[32] <= 32'b011001_00000000000000000000000000;
    end

    always @(posedge clk) begin
        if(mode==0 && write_en==1) begin
            veda[addr] = write_data;
            out = veda[addr];
        end
        else if(mode==1) begin
            out <= veda[addr];
        end
    end

    assign out_wire = out;
endmodule
```

PDS5 - Next design and implement a module for instruction decode to identify which data path element to execute given the opcode of the instruction.

- We have written a module named **instr-decode.v** which is called in the main processor module to decode the instruction which was fetched, and identify which path should be taken going forward.
- This is a completely **combinational** module. It assigns values to various registers like r_s, r_d, r_t etc. based on the instruction format. The instruction format is identified based on the opcode. The module has been implemented as follows:

```
module instruction_decode(clk, instr, opcode, rs, rd, rt, shamt, funct, address_i, address_j, imm);
    input clk;
    input[31:0] instr;
    output [5:0] opcode, funct;
    output [4:0] rs, rt, rd, shamt;
    output [25:0] address_j;
    output [15:0] address_i, imm;
    reg [5:0] functw;
    reg [4:0] rsw, rtw, rdw, shamtw;
    reg [25:0] address_jw;
    reg [15:0] address_iw, immw;
    assign opcode = instr[31:26];
    assign funct = functw;
    assign rs = rsw;
    assign rt = rtw;
    assign rd = rdw;
    assign shamt = shamtw;
    assign address_j = address_jw;
    assign address_i = address_iw;
    assign imm = immw;
    always @(*) begin
        // opcodes for J-type instructions (j, jal)
        if(opcode == 6'b010100 || opcode == 6'b010110) begin
            address_jw <= instr[25:0];
        end
        // opcodes for I-type instructions (addi, addiu, andi, ori, lw, sw, slti, sll, srl,
        beq, bne, bgt, bgte, ble, bleq)
        else if((opcode == 6'b000100) || (opcode == 6'b000101) ||
        (opcode >= 6'b001000 && opcode <= 6'b010011) || (opcode == 6'b011000)) begin
            rsw <= instr[25:21];
            rtw <= instr[20:16];
            address_iw <= instr[15:0];
            immw <= instr[15:0];
        end
        // opcodes for R-type instructions
        else begin
            rsw <= instr[25:21];
            rtw <= instr[20:16];
            rdw <= instr[15:11];
            shamtw <= instr[10:6];
            functw <= instr[5:0];
        end
    end
```

PDS6 - Design and implement the Arithmetic Logic Unit (ALU) in top-down approach to develop different modules for different types of instructions.

- The ALU has been implemented in a separate module named **ALU.v** which is included in the main processor module to access all of its modules.
- Following is the implementation:

```
module ADD (input signed [31:0] A, input signed [31:0] B, output [31:0] out);

assign out = A + B;

endmodule

module ADDI(input signed [31:0] A, input [15:0] imm,  output [31:0] out);

assign out = A + {{16{imm[15]}}, imm};

endmodule

module ADDIU(input unsigned [31:0] A, input [15:0] imm,  output [31:0] out);

assign out = A + {{16{imm[15]}}, imm};

endmodule

module ADDU(input unsigned [31:0] A, input unsigned [31:0] B, output [31:0] out);

assign out = A+B;

endmodule

module SUB(input signed [31:0] A, input signed [31:0] B, output [31:0] diff);

assign diff = A - B;

endmodule

module SUBU(input unsigned [31:0] A, input unsigned [31:0] B, output [31:0] diff);

assign diff = A - B;

endmodule

module AND(input [31:0] A, input [31:0] B, output [31:0] out);

assign out = A&B;

endmodule
```

```
module ANDI(input [31:0] A, input [15:0] imm, output [31:0] out);

assign out = A & {{16{imm[15]}}, imm};

endmodule

module OR(input [31:0] A, input [31:0] B, output [31:0] out);

assign out = A | B;

endmodule

module ORI(input [31:0] A, input [15:0] imm, output [31:0] out);

assign out = A | {{16{imm[15]}}, imm};

endmodule

module SLL(input [31:0] A, input [15:0] sa, output [31:0] out);

assign out = A << sa;

endmodule

module SRL(input [31:0] A, input [15:0] sa, output [31:0] out);

assign out = A >> sa;

endmodule

module SLT(input signed [31:0] A,  input signed [31:0] B, output [31:0] out);

assign out = (A < B) ? 1 : 0;

endmodule

module SLTI(input signed [31:0] A, input signed [15:0] imm, output [31:0] out);

assign out = (A < {{16{imm[15]}}, imm})? 1:0;

endmodule
```

- The complete module is **combinational** as can be seen.

PDS7 - Design and implement the branching operation along the ALU implementation.

- The branching operations have been in implemented in a file named **branching.v** which has been included in the main processor module to use its modules.
- Following is the implementation:

```
module BEQ (input clk,input rst, input signed [31:0] r0,input signed [31:0] r1,
input signed [15:0] offset, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = (r0==r1) ? (pc_in + {{16{offset[15]}}, offset} + 1) : (pc_in + 1) ;

endmodule

module BGT (input clk,input rst, input signed [31:0] r0,input signed [31:0] r1,
input signed [15:0] offset, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = (r0 > r1) ? (pc_in + {{16{offset[15]}}, offset} + 1) : (pc_in + 1) ;

endmodule

module BGTE (input clk,input rst, input signed [31:0] r0,input signed [31:0] r1,
input signed [15:0] offset, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = (r0 >= r1) ? (pc_in + {{16{offset[15]}}, offset} + 1) : (pc_in + 1) ;

endmodule

module BLE (input clk,input rst, input signed [31:0] r0,input signed [31:0] r1,
input signed [15:0] offset, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = (r0 < r1) ? (pc_in + {{16{offset[15]}}, offset} + 1) : (pc_in + 1) ;

endmodule

module BLEQ (input clk,input rst, input signed [31:0] r0,input signed [31:0] r1,
input signed [15:0] offset, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = (r0 <= r1) ? (pc_in + {{16{offset[15]}}, offset} + 1) : (pc_in + 1) ;

endmodule

module BNE (input clk,input rst, input signed [31:0] r0,input signed [31:0] r1,
input signed [15:0] offset, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = (r0 != r1) ? (pc_in + {{16{offset[15]}}, offset} + 1) : (pc_in + 1) ;

endmodule

module J (input clk, input [25:0] target_address, input [31:0] pc_in, output [31:0] pc_out);

  assign pc_out = {{6{target_address[25]}}, target_address};
```

```
endmodule

module JAL(input clk, input [25:0] target_address, input [31:0] pc_in, output [31:0] pc_out,
output [31:0] jal_ra);

  assign pc_out = {pc_in[31:28], target_address, 2'b00};
  assign jal_ra = pc_in + 1;

endmodule

module JR (input clk, input [31:0] r0, output [31:0] pc_out);

  assign pc_out = r0;

endmodule
```

- All the modules are completely **combinational** are assign new values of the program counter based on the logic of the module. The **JAL** module also assigns the value of the next program counter to the **return address** register.

PDS8 - Design the finite state machine for the control signals to execute the processor. Please ensure that every instruction should be executed in single clock cycle. Finally write the test benches to simulate the CSE-BUBBLE.

- We have defined the different states for the FSM based on the operation to be executed, as follows:

```
parameter execute_ADD = 6'b000000, execute_SUB = 6'b000001, execute_SUBU = 6'b000010,
execute_ADDU = 6'b000011,execute_ADDI = 6'b000100, execute_ADDIU = 6'b000101,
execute_AND = 6'b000110, execute_OR = 6'b000111,execute_ANDI = 6'b001000,
execute_ORI = 6'b001001, execute_SLL = 6'b001010,  execute_SRL = 6'b001011,
execute_LW = 6'b001100,    execute_SW = 6'b001101, execute_BEQ = 6'b001110,
execute_BNE = 6'b001111, execute_BGT = 6'b010000, execute_BGTE = 6'b010001,
execute_BLE = 6'b010010, execute_BLEQ = 6'b010011, execute_J = 6'b010100,
execute_JR = 6'b010101,   execute_JAL = 6'b010110,  execute_SLT = 6'b010111,
execute_SLTI = 6'b011000, execute_SYS = 6'b011001, execute_NOP = 6'b011010;
```

- We create the hardware for each possible operation as follows:

```
    wire [31:0] temp_output[0:31];

    ADD add_inst(.A(processor[rs]), .B(processor[rt]), .out(temp_output[0]));
    SUB sub_inst(.A(processor[rs]), .B(processor[rt]), .diff(temp_output[1]));
    SUBU subu_inst(.A(processor[rs]), .B(processor[rt]), .diff(temp_output[2]));
    ADDU addu_inst(.A(processor[rs]), .B(processor[rt]), .out(temp_output[3]));
    ADDI addi_inst(.A(processor[rs]), .imm(imm), .out(temp_output[4]));
    ADDIU addiu_inst(.A(processor[rs]), .imm(imm), .out(temp_output[5]));
    AND and_inst(.A(processor[rs]), .B(processor[rt]), .out(temp_output[6]));
    OR or_inst(.A(processor[rs]), .B(processor[rt]), .out(temp_output[7]));
    ANDI andi_inst(.A(processor[rs]), .imm(imm), .out(temp_output[8]));
```

```
ORI ori_inst( .A(processor[rs]), .imm(imm), .out(temp_output[9]));
SLL sll_inst(.A(processor[rs]), .sa(imm), .out(temp_output[10]));
SRL srl_inst(.A(processor[rs]), .sa(imm), .out(temp_output[11]));
SLT slt_inst( .A(processor[rs]), .B(processor[rt]), .out(temp_output[12]));
SLTI slti_inst( .A(processor[rs]), .imm(imm), .out(temp_output[13]));
BEQ beq_inst(.clk(clk), .rst(rst), .r0(processor[rs]), .r1(processor[rt]), .offset(address_i),
.pc_in(processor[0]), .pc_out(temp_output[14]));
BNE bne_inst(.clk(clk), .rst(rst), .r0(processor[rs]), .r1(processor[rt]), .offset(address_i),
.pc_in(processor[0]), .pc_out(temp_output[15]));
BGT bgt_inst(.clk(clk), .rst(rst), .r0(processor[rs]), .r1(processor[rt]), .offset(address_i),
.pc_in(processor[0]), .pc_out(temp_output[16]));
BGTE bgte_inst(.clk(clk), .rst(rst), .r0(processor[rs]), .r1(processor[rt]), .offset(address_i),
.pc_in(processor[0]), .pc_out(temp_output[17]));
BLE ble_inst(.clk(clk), .rst(rst), .r0(processor[rs]), .r1(processor[rt]), .offset(address_i),
.pc_in(processor[0]), .pc_out(temp_output[18]));
BLEQ bleq_inst(.clk(clk), .rst(rst), .r0(processor[rs]), .r1(processor[rt]), .offset(address_i),
.pc_in(processor[0]), .pc_out(temp_output[19]));
J j_inst(.clk(clk), .target_address(address_j), .pc_in(processor[0]), .pc_out(temp_output[20]));
JAL jal_inst(.clk(clk), .target_address(address_j), .pc_in(processor[0]), .pc_out(temp_output[22]),
.jal_ra(temp_output[24]));
JR jr_inst(.clk(clk), .r0(processor[rs]), .pc_out(temp_output[25]));
```

- We then use the output from the appropriate hardware based on the FSM which executes the instruction on the **negative edge** of clock. Following are a few sections of the FSM:

```
always @(negedge clk) begin
    case(opcode)

    // Following are states for ALU instructions

        execute_ADD: begin
            processor[rd] = temp_output[0];
            processor[0] = processor[0] + 1;  //incrementing the program counter
        end
        execute_SUB: begin
            processor[rd] = temp_output[1];
            processor[0] = processor[0] + 1;
        end
        .
        .
    // Following are states for Data transfer instructions

        execute_LW: begin
            mode = 1'b1;
            data_addr = processor[rs] + {{16{address_i[15]}}, address_i};
            processor[0] = processor[0] + 1;
        end
        execute_SW: begin
            mode = 1'b0;
            data_addr = processor[rs] + {{16{address_i[15]}}, address_i};
            data_in = processor[rt];
            processor[0] = processor[0] + 1;
        end
        .
        .
    // Following are states for Branching instructions

        execute_BNE: begin
            processor[0] <= temp_output[15];
        end
        execute_BGT: begin
            processor[0] <= temp_output[16];
        end

    // Following are states for System instructions

        execute_SYS: begin
            temp_input[0] = processor[8];        // storing $v0
            temp_input[1] = processor[10];       // storing $a0
            processor[0] = processor[0] + 1;
        end

        execute_NOP: begin
            processor[0] <= processor[0] + 1;
        end

        // Default case

        default: begin
            processor[0] <= processor[0] + 1;
        end

    endcase
end
```

- In each state, the ouput is stored in the appropriate register, while the program counter is being updated.
- We have also created a separate module named **system.v** which executes the system instructions like **syscall** or **nop**. It was implemented as follows:

```
module sys_execute(input [5:0] opcode, input [31:0] regv, input [31:0] rega);

    always @(opcode) begin

        // if syscall was not called
        if(opcode != 6'b011001) begin
        end

        // to display int with space
        else if(regv == 32'b000000_00000_00000_00000_00000_000001) begin
            $display("%d ", rega);
        end

        // to display whitespace
        else if(regv == 32'b000000_00000_00000_00000_00000_001011) begin
            $display(" ");
        end

        // to exit program
        else if(regv == 32'b000000_00000_00000_00000_00000_001010 ) begin
            #500
            $finish;
        end

    end

endmodule
```

PDS9 - Develop the MIPS code for Bubble Sort. Then convert it into machine code following the ISA given above.

MIPS code for Bubble Sort:

```
.data

array: .word 5, 4, 3, 2, 1, 0                                  # input array
size:  .word 6                                                 # size of array

.text
main:
    la $s0, array                                              # load address of array into $s0
    lw $s1, size                                               # load size of array into $s1
    addi $s2, $s1, -1                                          # $s2 stores "size - 1" (length of loop)
    li $t0, 0                                                  # loop 1 counter initialization
    li $t1, 0                                                  # loop 2 counter initialization
    li $t6, 0                                                  # printing counter

    Loop:

        sll $t2, $t1, 2                                        # $t2 = 4*i
        add $t2, $s0, $t2                                      # $t2 stores the address of array + 4i
        lw $t3, 0($t2)                                         # $t3 stores arr[i]
        lw $t4, 4($t2)                                         # $t4 stores arr[i+1]
        slt $t5, $t3, $t4                                      # $t5 = 1 if arr[i] < arr[i+1]
        bne $t5, $zero, Increase                               # If $t5 = 1, go to Increase
        sw $t4, 0($t2)                                         # swap operation
        sw $t3, 4($t2)                                         # swap operation

    Increase:

        addi $t1, $t1, 1                                       # increment i
        sub $s3, $s2, $t0                                      # $s3 = n - 1 - j
        bne $t1, $s3, Loop                                     # if i != n-1, go to Loop
        addi $t0, $t0, 1                                       # otherwise increment j
        li $t1, 0                                              # i = 0
        bne $t0, $s2, Loop

    Print:
```

```
    beq $t6, $s1, Exit                                      # if counter != size
    lw $t7, 0($s0)                                          # load from array
    li $v0, 1                                               # system call to print integer
    move $a0, $t7
    syscall
    li $a0, 32                                              # system call to print empty space
    li $v0, 11
    syscall
    addi $s0, $s0, 4                                        # incrementing array
    addi $t6, $t6, 1                                        # incrementing loop counter
    j Print

Exit:

    li $v0, 10                                              # exit the program
    syscall
```

Machine code for the instructions:

```
32'b000000_00110_00110_11001_00000_000000; //  la $s0, array   (add $s0, $zero, $zero) [0]
32'b001100_00110_11010_0000000000000110;   //  lw $s1, size    (lw $s1, 6($zero)) [1]
32'b000100_11010_11011_1111111111111111;   //  addi $s2, $s1, -1   [2]
32'b000000_00110_00110_10001_00000_000000; //  li $t0, 0 (add $t0, $zero, $zero) [3]
32'b000000_00110_00110_10010_00000_000000; //  li $t1, 0 (add $t1, $zero, $zero) [4]
32'b000000_00110_00110_10111_00000_000000; //  li $t6, 0 (add $t6, $zero, $zero) [5]

// instruction here for specifying that we are entering 'Loop' //

32'b001010_10010_10011_0000000000000010;    // sll $t2, $t1, 2 [6]
32'b000000_10011_11001_11011_00000_000000;  // add $t2, $s0, $t2 [7]
32'b001100_10011_10100_0000000000000000;    // lw $t3, 0($t2) [8]
32'b001100_10011_10101_0000000000000001;    // lw $t4, 1($t2) [9]
32'b010111_10100_10101_10110_00000_000000;  // slt $t5, $t3, $t4 [10]
32'b001111_10110_10110_0000000000000010;    // bne $t5, $zero, 2 (address of Increase is 14) [11]
32'b001101_10011_10101_0000000000000000;    // sw $t4, 0($t2) [12]
32'b001101_10011_10100_0000000000000001;    // sw $t3, 1($t2) [13]

// instruction here for specifying that we are entering 'Increase' //

32'b000100_10010_10010_0000000000000001;    // addi $t1, $t1, 1 [14]
32'b000001_11100_11011_10001_00000_000000;  // sub $s3, $s2, $t0 [15]
32'b001111_10010_11100_1111111111110101;    // bne $t1, $s3, -11 (address of Loop is 6) [16]
32'b000100_10001_10001_0000000000000001;    // addi $t0, $t0, 1 [17]
32'b000000_00110_00110_10010_00000_000000;  // li $t1, 0 (add $t1, $zero, $zero) [18]
32'b001111_10001_11011_0000000000001110;    // bne $t0, $s2, -14 (address of Loop is 6) [19]

// instruction here for specifying that we are entering 'Print' //

32'b001110_10111_11010_0000000000001010;    // beq $t6, $s1, 10  (address of Exit is 31) [20]
32'b001100_11001_11000_0000000000000000;    // lw $t7, 0($s0) [21]
32'b000100_00110_01000_0000000000000001;    // li $v0, 1 (addi $v0, $zero, 1) [22]
32'b000000_00110_11000_01010_00000_000000;  // move $a0, $t7 (add $a0, $zero, $t7) [23]
32'b011001_00000000000000000000000000;           [24]
32'b000100_00110_01010_0000000000100000;    // li $a0, 32 (addi $a0, $zero, 32) [25]
```

```
32'b000100_00110_01000_0000000000001011;    // li $v0, 11 (addi $v0, $zero, 11) [26]
32'b011001_00000000000000000000000000;          [27]
32'b000100_11001_11001_0000000000000001;    // addi $s0, $s0, 1 [28]
32'b000100_10111_10111_0000000000000001;    // addi $t6, $t6, 1 [29]
32'b010100_00000000000000000000010100;      // j 20 (address of Print) [30]

// instruction here for specifying that we are entering 'Exit' //

32'b000100_00110_01000_0000000000001010;    // li $v0, 10 (addi $v0, $zero, 10) [31]
// syscall // [32]
```

NOTE: The pseudo-instructions in the MIPS code were converted to basic-instructions in CSE-BUBBLE processor.

PDS10 - Store the machine code in the instruction memory and execute CSE-BUBBLE. Store the output of the bubble sort in the data memory.

- We have stored the machine code for the instructions in the **fetch-instructions.v** module and the data about the array and array-size has been stored in the **fetch-data.v** module.
- The sorted array has been stored in the data memory.
- We simulate the processor through the testbench named **CSE-BUBBLE-tb.v**, as follows:

```
`include "CSE_BUBBLE.v"

module simulator();

    reg clk, rst;
    cse_bubble module_call(.clk(clk), .rst(rst));

    initial begin
        #10
        clk <= 1'b1;
        forever #10 clk <= ~clk;
    end

    initial begin
        rst <= 0;
        #1
        rst <= 1;
    end

    initial begin
        $display("\nSorted Array is: ");
        #5000
        $finish;
    end
endmodule
```

- The following command needs to be executed:

```
    iverilog -o test CSE_BUBBLE_tb.v
    vvp test
```

- We have included **syscall** instructions in the machine code, so that the output of the sorted array can be directly seen after simulation. Therefore, you will be able to see output of the following format:

  ```
  Sorted Array is:
           0

          23

         122

         220

         433

        3432
  ```

- We have provided options for various different arrays that can be used to test the processor, by simply commenting the other examples.
