CS220: Computer Organisation

Assignment Report

Assignment 7: CSE-BUBBLE

Aniket Suhas Borkar 210135
Vikas Yadav 211166

20th March 2023 (PDS1, PDS2, PDS3, PDS4, PDS5)

PDS1:

We plan to use 32 registers, each having a width of 32 bits (1 word) :-

Register	Number	Use
$zero	0	Constant 0
$v0-$v1	2-3	Return Value of function
$a0-$a3	4–7	Arguments to functions
$t0–$t7	8–15	Temporary Registers
$s0–$s7	16–23	Saved Registers
$t8–$t9	24–25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

PDS2:

We plan to use instruction memory of size 256 words (32 bits each).
We plan to use data memory of 256 words (32 bits each).

We have created two separate modules for the data and instruction memories. This was done for simplification, and to ensure that the data and instructions remain separate and are not mixed or overwritten.

PDS3:

We plan to use the instruction formats borrowed from the MIPS instruction set architecture. The instruction layout and encoding strategies for the R, I and J-type instructions are as detailed below.

Instruction layout for R-type instruction:

- It is 32 bits or 4 bytes wide
- Bit 31 to bit 26 is used to store operation code for the instruction.
- Bit 25 to bit 21 is used to store the number of the register that is the first source operand
- Bit 20 to bit 16 is used to store the number of the register that is the second source operand
- Bit 15 to bit 11 is used to store the number of the register that is the destination
- Bit 10 to bit 6 is used to store the shift amount
- Bit 5 to bit 0 is used to store function code

Instruction layout for I-type instruction:

- It is 32 bits or 4 bytes wide
- Bit 31 to bit 26 is used to store operation code for the instruction.
- Bit 25 to bit 21 is used to store the number of the register that stores the base address
- Bit 20 to bit 16 is used to store the number of the register that is the destination
- Bit 15 to bit 0 is used to store the memory address that is to be loaded

Instruction layout for J-type instruction:

- It is 32 bits or 4 bytes wide
- Bit 31 to bit 26 is used to store operation code for the instruction.
- Bit 25 to bit 0 is used to store the offset from the current instructions

Opcode and function for instruction used in our design

- Add: opcode=0 function code=32
- Addu: opcode=0 function code=33
- Sub : opcode=0 function code=34
- Subu : opcode=0 function code=35
- And: opcode=0 function code=36
- Or : opcode=0 function code=37
- Nor : opcode=0 function code=39
- Slt : opcode=0 function code=42
- Sll : opcode=0 function code=0
- Srl : opcode=0 function code=2
- Jr : opcode=0 function code=8
- J : opcode=2
- Jal : opcode=3
- beq : opcode=4
- bneq : opcode=5
- Stli : opcode=10

- Addi : opcode=8
- Addiu : opcode=9
- Andi : opcode=12
- Ori : opcode=13
- Sw : opcode=43
- Lw : opcode=35
- Bgt : opcode=63
- Bgte : opcode=62
- Ble : opcode=61
- Bleq : opcode=60

PDS4:

We have designed and implemented instruction fetching in an implicit manner. The instruction memory has been instantiated in a module named **top_module.** We pass the address (in terms of program counter **PC**) of the instruction to be fetched and the instruction memory returns the instruction to be executed which is carried in a 32 bit wire named **instruction**.

```
wire [7:0] PC_pruned;
assign PC_pruned = PC[7:0];

// instantiate the instruction memory
inst_mem inst(
    .address(PC_pruned),
    .write_enable(zero),
    .write_address(zero8),
    .write_data(zero32),
    .instr_out(instruction)
);
```

PDS5:

We have implemented the instruction decode inside the **control** module.
Here we first identify whether the instruction is an R-type, I-type or J-type by checking the most significant 6 bits of the instruction.
After identifying the type, we further decode the exact type of instruction by checking the exact value of opcode and function code.

27th March 2023 (PDS6, PDS7)

PDS6:

Internally the ALU has various modules which perform different types of instructions such as add, and, or etc.
Then we have an output mux (output_mux) which selects the correct output for ALU from the outputs of various modules (as described above) based on the operation code received from the control.

The ALU operation codes are as follows:

Operation code	Operation
0	add
1	or
2	and
3	nor
4	Set less than
5	Set less than or equal to
6	Equal to
7	Shift left logical
8	Shift right logical
9	Not equal to
10	Greater than
11	Greater than or equal to
default	Output ‘0’

Subtract is implemented by sending the 2’s complement of the second operand with operation code for add (0).

PDS7:

Branching is handled by appropriately updating the program counter.
The design consists of a module named **PC_selector_mux** in the **top** module which handles how the program counter is to be updated. It contains adders and a mux. The adders generate the values PC+1 and PC+1+increment. The mux chooses the appropriate value which is to be set as the next PC value.
where

- PC : is the program counter
- PC_cntrl : holds data which states whether we have to branch(1) , jump(3) , jr(2) or simply update the program counter by 1 (0). It is the control line of the mux.
- Immediate : stores the offset in the branch instruction along with sign
- rs_reg_data: stores the value in the register in jr instruction
- ALUflag : represents whether the given comparison statement (i.e, the branching condition) is true(1) or false(0).
- PC_next : stores the value of the updated program counter

```
// instantiate the mux which handles PC updation (branch, jump, jr, PC+1)
PC_selector_mux PC_updater(
    .PC(PC),
    .sel(PC_cntrl),
    .Immediate(Immediate),
    .Reg_val(rs_reg_data),
    .ALU_flag(ALUflag),
    .out(PC_next)
);
```

```
wire [31:0] PC_plus_1;
assign PC_plus_1 = PC + 1;
wire [31:0] branch_address;
assign branch_address = PC + 1 + Immediate;
wire [31:0] jump_address;
assign jump_address = Immediate;

always @(*) begin
    case (sel)
        0: out <= PC_plus_1;
        1: begin
            if(ALU_flag) begin
                out <= branch_address;
            end
            else begin
                out <= PC_plus_1;
            end
        end
        2: out <= Reg_val;
        3: out <= jump_address;
    endcase
end
```

3rd April 2023 (PDS8)

PDS8:

This is an extension of the PDS5, and control signals have been implemented in the **control** module. Depending on the type of the instruction we send the appropriate operands and operation code to be performed to the ALU. We also send appropriate control signals which are used to decide how the PC's next value is to be chosen (branch, jump, PC+1). We decide whether the value to be stored into registers comes from data memory or registers. Finally, the control signals for Memory write and Register write operations are also controlled by this module.

Further details corresponding to how each specific instruction affects the control signals is included in the comments which accompany the code.

10th April 2023 (PDS9, PDS10)

PDS9:

MIPS code of bubble sort is submitted as a part of the codebase along with this document. The above code is converted to machine code using the **ISA** specified and stored in instruction memory as shown in the image for PSD10.

PDS10:

Machine code is stored in instruction memory as follows:

```
initial begin
    for (i = 0; i < 255 ; i = i + 1) begin
        memory[i] = 0;
    end

    //code for bubble sort
    memory[0] = {6'b001000 , 5'd0 , 5'd23 , 16'd0}; //addi
    memory[1] = {6'b001000 , 5'd0 , 5'd16 , 16'd0}; //addi
    memory[2] = {6'b001000 , 5'd0 , 5'd22 , 16'd9}; //addi
    memory[3] = {6'b001000 , 5'd0 , 5'd17 , 16'd0}; //addi
    memory[4] = {6'b000000 , 5'd17 , 5'd23 , 5'd15 , 5'd0 , 6'd32}; //add
    memory[5] = {6'b100011 , 5'd15 , 5'd8, 16'd0}; //lw
    memory[6] = {6'b100011 , 5'd15 , 5'd9, 16'd1}; //lw
    memory[7] = {6'b000000 , 5'd8  , 5'd9, 5'd10 , 5'd0, 6'b101010}; //slt
    memory[8] = {6'b000101 , 5'd10 , 5'd0, 16'd2}; //bne
    memory[9] = {6'b101011 , 5'd15 , 5'd9, 16'd0}; //sw
    memory[10] = {6'b101011 , 5'd15 , 5'd8, 16'd1}; //sw
    memory[11] = {6'b001000 , 5'd17 , 5'd17 , 16'd1}; //addi
    memory[12] = {6'b000000 , 5'd22 , 5'd16 , 5'd21 , 5'd0 , 6'd34}; //sub
    memory[13] = {6'b000101 , 5'd17 , 5'd21, 16'd65526}; //bne
    memory[14] = {6'b001000 , 5'd16 , 5'd16 , 16'd1}; //addi
    memory[15] = {6'b001000 , 5'd0 , 5'd17 , 16'd0} ;//addi
    memory[16] = {6'b000101 , 5'd16 , 5'd22, 16'd65523}; //bne
end
```

The bubble sort is executed by simply instantiating the **top_module** in a testbench with clk and reset as the only input to the module.

The output of the bubble sort is stored in the memory module from 0 to 9th index in the case in which we are sorting 10 numbers.

The changes can be viewed by **$monitor** for the used memory indexes.

```
initial begin
    // monitor allows us to see how the array changes as it is sorted
    $monitor("%d, %d, %d, %d, %d, %d, %d, %d, %d, %d", memory[0], memory[1], memory[2], memory[3], memory[4], memory[5], memory[6], memory[7], memory[8], memory[9]);
end
```

For example, when the input array is
[431, 4, 1, 3, 0, 341, 2, 5, -9, 64],
we get the following output on the command line.

The first line shows the unsorted array and the last line shows the final sorted array.

```
hunter@HUNTER-F15:~/Desktop/CS220/final$ vvp bsort.vvp
VCD info: dumpfile newtb.vcd opened for output.
      431,         4,         1,         3,         0,       341,         2,         5,        -9,        64
        4,         4,         1,         3,         0,       341,         2,         5,        -9,        64
        4,       431,         1,         3,         0,       341,         2,         5,        -9,        64
        4,         1,         1,         3,         0,       341,         2,         5,        -9,        64
        4,         1,       431,         3,         0,       341,         2,         5,        -9,        64
        4,         1,         3,         3,         0,       341,         2,         5,        -9,        64
        4,         1,         3,       431,         0,       341,         2,         5,        -9,        64
        4,         1,         3,         0,         0,       341,         2,         5,        -9,        64
        4,         1,         3,         0,       431,       341,         2,         5,        -9,        64
        4,         1,         3,         0,       341,       341,         2,         5,        -9,        64
        4,         1,         3,         0,       341,       431,         2,         5,        -9,        64
        4,         1,         3,         0,       341,         2,         2,         5,        -9,        64
        4,         1,         3,         0,       341,         2,       431,         5,        -9,        64
        4,         1,         3,         0,       341,         2,         5,         5,        -9,        64
        4,         1,         3,         0,       341,         2,         5,       431,        -9,        64
        4,         1,         3,         0,       341,         2,         5,        -9,        -9,        64
        4,         1,         3,         0,       341,         2,         5,        -9,       431,        64
        4,         1,         3,         0,       341,         2,         5,        -9,        64,        64
        4,         1,         3,         0,       341,         2,         5,        -9,        64,       431
        1,         1,         3,         0,       341,         2,         5,        -9,        64,       431
        1,         4,         3,         0,       341,         2,         5,        -9,        64,       431
        1,         3,         3,         0,       341,         2,         5,        -9,        64,       431
        1,         3,         4,         0,       341,         2,         5,        -9,        64,       431
        1,         3,         0,         0,       341,         2,         5,        -9,        64,       431
        1,         3,         0,         4,       341,         2,         5,        -9,        64,       431
        1,         3,         0,         4,         2,         2,         5,        -9,        64,       431
        1,         3,         0,         4,         2,       341,         5,        -9,        64,       431
        1,         3,         0,         4,         2,         5,         5,        -9,        64,       431
        1,         3,         0,         4,         2,         5,       341,        -9,        64,       431
        1,         3,         0,         4,         2,         5,        -9,        -9,        64,       431
        1,         3,         0,         4,         2,         5,        -9,       341,        64,       431
        1,         3,         0,         4,         2,         5,        -9,        64,        64,       431
        1,         3,         0,         4,         2,         5,        -9,        64,       341,       431
        1,         0,         0,         4,         2,         5,        -9,        64,       341,       431
        1,         0,         3,         4,         2,         5,        -9,        64,       341,       431
        1,         0,         3,         2,         2,         5,        -9,        64,       341,       431
        1,         0,         3,         2,         4,         5,        -9,        64,       341,       431
        1,         0,         3,         2,         4,        -9,        -9,        64,       341,       431
        1,         0,         3,         2,         4,        -9,         5,        64,       341,       431
        0,         0,         3,         2,         4,        -9,         5,        64,       341,       431
        0,         1,         3,         2,         4,        -9,         5,        64,       341,       431
        0,         1,         2,         2,         4,        -9,         5,        64,       341,       431
        0,         1,         2,         3,         4,        -9,         5,        64,       341,       431
        0,         1,         2,         3,        -9,        -9,         5,        64,       341,       431
        0,         1,         2,         3,        -9,         4,         5,        64,       341,       431
        0,         1,         2,        -9,        -9,         4,         5,        64,       341,       431
        0,         1,         2,        -9,         3,         4,         5,        64,       341,       431
        0,         1,        -9,        -9,         3,         4,         5,        64,       341,       431
        0,         1,        -9,         2,         3,         4,         5,        64,       341,       431
        0,        -9,        -9,         2,         3,         4,         5,        64,       341,       431
        0,        -9,         1,         2,         3,         4,         5,        64,       341,       431
       -9,        -9,         1,         2,         3,         4,         5,        64,       341,       431
       -9,         0,         1,         2,         3,         4,         5,        64,       341,       431
```

***Note**: We have used $monitor which displays the data as soon as it detects change in any of the values to be displayed . therefore you can see an intermediate extra line during each swap operation.