Assignment-1, Set 2:

Group 43

Name: Shreyas Bindumadhavan

Question:

Design a 5 – stage (Instruction Fetch; Decode; Execute; Memory Access; Write Back) multi-cycle RISC processor that can execute following instructions.

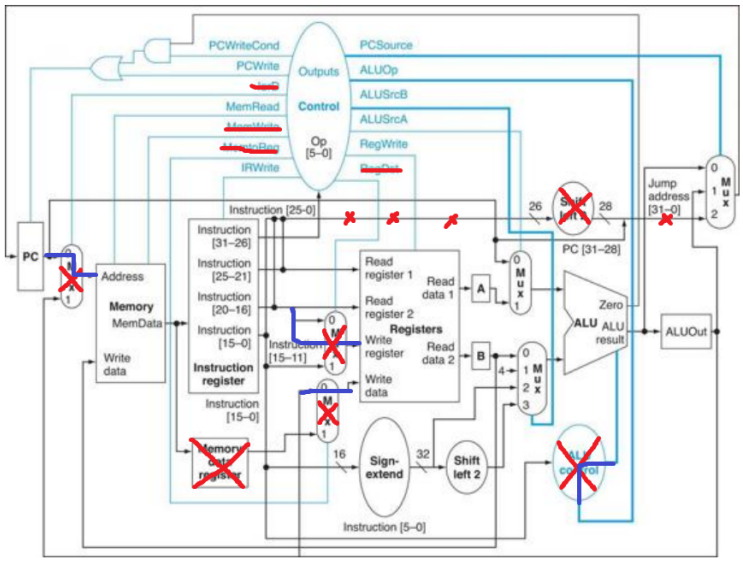
0000 ADDI reg1, reg2, 20

0004 BNE reg3, reg4, 4

Initialize the register file with the following data reg2 = 4BFEA43 reg3 = 11AB0 reg4 = 11AB0

Answer:

Notice that both of these instructions do not need to perform memory write and read, so we can remove the MDR and all control signals associated with it. Since we are not performing any jump instructions, we can get rid of some multiplexers and a Shift Left 2 block. We can also get rid of ALU control – the net Instruction [5:0] is useless since the immediate value is stored here – all information necessary can be gleaned from the opcode which is sent to the main Control unit. Taking all of this into consideration, the updated datapath is as shown below.



The datapath with the necessary changes

We now proceed to create separate modules out of each block on the datapath – and then wire them up together in a top module, which will take the clock signal as the input. The modules are

presented from left to right, mostly in order of execution. All multiplexors have been implemented as 4:1 multiplexor; this means that the lower order multiplexors have some input lines unwired.

(Please note that the module that implements combinational logic in the top-left section of the datapath used to enable writing to the Program Counter has been named **or_and**).

Components of the datapath:

Program Counter:

Increments the program count by 4 once an instruction is completed. This is done by using a combination of control signals to make sure that PC can only be written to once an entire instruction has been completed.

```
module Program_Counter(
    clk,
    in_address,
    write,
    out_address
    );

input clk;
input [31:0] in_address;
input write;
reg [31:0] PC;
output reg[31:0] out_address;

initial begin
    @(posedge clk) out_address <= 32'd0;
end

always @ (posedge clk)
begin
if(write)out_address <= PC;
end

//update internal register with alu computaton mid-cycle for use next cycle
always @ (negedge clk)
begin
if(write)   PC <= in_address;
end

endmodule
```

Instruction Memory:

All the instructions are stored here. Once the PC loads in the instruction address, it accesses the instruction at the specified address and loads it into the instruction register – this is a feature of multicycle instructions designed to preserve the state of the instruction since each instruction takes place over multiple clock cycles. It is controlled by the MemRead control signal that is set during the instruction fetch phase.

```
module Instruction_Memory(
    clk,
    MemRead,
    instrn_address,
    MemOut,
    );
input clk;
input MemRead;
input [31:0] instrn_address;
output reg [31:0] MemOut;
reg [31:0] instrn_mem [0:7];

initial begin
$readmemh("instrn_memory.mem", instrn_mem);
end

always @ (*)
begin
    if(MemRead) MemOut = {instrn_mem[instrn_address+3],instrn_mem[instrn_address+2],instrn_mem[instrn_address+1],instrn_mem[instrn_address]};
end


endmodule
```

Instruction Register:

As described previously, this is a temporary register 32 bits wide used to save the instruction from the instruction memory without loss of state over multiple clock cycles and to prevent timing races.

```
module Instruction_Register(
    clk,
    MemData,
    IRWrite,
    Instruction

    );
input clk;
input [31:0] MemData;
input IRWrite;
reg [31:0] IR;
output wire [31:0] Instruction;

assign Instruction = IR;

always @ (posedge clk)
begin
    IR <= IRWrite? MemData : IR;
end


endmodule
```

Register File:

This is a block of 32 registers visible to the programmer. The RegWrite control signal dictates when the register files can be written into and is typically set during the writeback phase of the multicycle.

We read initial values from a memory file using *readmemh.*

```
module Register_File(
    clk,
    read_reg1,
    read_reg2,
    RegWrite,
    write_reg,
    write_data,
    read_data1,
    read_data2
    );

input clk;
input [4:0] read_reg1;
input [4:0] read_reg2;
input RegWrite;
input [4:0] write_reg;
input [31:0] write_data;


output [31:0] read_data1;
output [31:0] read_data2;

reg [31:0] reg_mem [0:31];

initial begin
$readmemh("reg_memory.mem", reg_mem); //Load initial values
end

assign read_data1 = reg_mem[read_reg1];
assign read_data2 = reg_mem[read_reg2];


// negedge is used to write into the register in the middle of a cycle (any delay could have been chosen - negedge was easiest to write)
always @ (negedge clk)
begin
reg_mem[write_reg] <= RegWrite ? write_data : reg_mem[write_reg];
end
endmodule
```

A and B registers:

These are temporary registers 32 bits wide that the *read_data* outputs from the register file load into for the same reasons outlined earlier.

```verilog
module A_reg(
    clk,
    read_data1,
    alusrc_A
    );

input clk;
input [31:0] read_data1;
output [31:0] alusrc_A;

reg [31:0] A;

assign alusrc_A = A;

// negedge to update midway (before next posedge)through clk cycle
always @ (negedge clk)
begin
    A <= read_data1;
end

endmodule
```

```verilog
module B_reg(
    clk,
    read_data2,
    alusrc_B
    );

input clk;
input [31:0] read_data2;
output [31:0] alusrc_B;

reg [31:0] B;

assign alusrc_B = B;

// negedge to update midway (before next posedge)through clk cycle
always @ (negedge clk)
begin
    B <= read_data2;
end

endmodule
```

ALU:

The arithmetic logic unit – which is taken from our single-cycle implementation. For our purposes, we only require two operations: add (during the ADDI instruction) and subtract (to compute the Zero wire during the BNE instruction). Since both instructions have immediate addressing, we can eliminate the ALU_Control unit altogether and directly use the ALUOp control signal to determine the operation.

```verilog
module Alu_Core(
    A,
    B,
    alu_control,
    result,
    zero
    );

input [31:0] A;
input [31:0] B;
input [1:0] alu_control;
output reg [31:0] result;
output wire zero;

assign zero = !(|result);

always @ (*)
begin
    case(alu_control)
    3'h0: result = A + B;
    3'h1: result = A - B;
    //3'h2: result = A & B;
    //3'h3: result = A | B;
    //3'h4: result = ~(A | B);
    //3'h5: result = (A < B);
    default: result = A + B;
    endcase
end
endmodule
```

ALU Output Register:

This is a temporary register used to store the output of the ALU. During the instruction decode phase, it is used to optimistically calculate and store the branching address from the immediate field regardless of whether or not the instruction branches or not. It is also used during the writeback phase to store the value of the ADDI instruction before writing it to the Register File.

Sign Extension:

Simple combinational unit used to extend the 16-bit immediate field for use in ALU operations.

```
module ALU_Out(
    clk,
    ALU_result,
    WriteData
    );
input clk;
input [31:0] ALU_result;
output wire [31:0] WriteData;
reg [31:0] Alu;

assign  WriteData = Alu;

//negedge so that the temp register is updated before next posedge (any
always @(negedge clk)
begin
    Alu <= ALU_result;
end

endmodule
```

Shift Left 2:

This is used to shift the input to the left by 2 while computing the offset – this is due to the fact that the memory happens to be word addressed and so each instruction has an address terminating with the binary *0*.

```
module Shifter(
    indata,
    shift_amt,
    shift_left,
    outdata
    );

input [31:0] indata;
input [1:0] shift_amt;
input shift_left;
output wire [31:0] outdata;
assign outdata = shift_left ? indata<<shift_amt : indata>>shift_amt;

endmodule
```

Or_and logic:

```verilog
module or_and(
    zero,
    PCWriteCond,
    PCWrite,
    write
    );

input zero, PCWriteCond, PCWrite;
output write;

assign write = (PCWriteCond & !zero) | PCWrite  ;

endmodule
```

This is the combinational logic present on the top left of the datapath that enables the program counter during the BNE instruction (when zero from the ALU is not set and PCWriteCond is set)

Control Unit:

The control logic has been written as a Mealy state machine with 4 states – note that the Memory Access stage is not needed since we are not using load or store instructions and thus, we do not need to add in an extra 5th state for this purpose. The control signals are then computed using logic expressions that use these state as input to decide whether the signal is set or not.

```verilog
module Control(
    clk,
    Op,
    PCWriteCond,
    PCWrite,
    MemRead,
    IRWrite,
    RegWrite,
    PCSource,
    ALUOp,
    ALUSrcB,
    ALUSrcA

    );

input clk;
input [5:0] Op;
output PCWriteCond, PCWrite, MemRead, IRWrite,RegWrite;
output [1:0] PCSource, ALUOp, ALUSrcB, ALUSrcA;


//State register
reg[2:0] state_reg, state_next;
localparam s0 = 0;
localparam s1 = 1;
localparam s2 = 2;
localparam s3 = 3;

localparam BNE = 6'd5; // opcode is 000101
localparam ADDI = 6'd8; // opcode is 001000

initial begin
    @ (posedge clk) state_reg <= s0;
end

always @ (posedge clk)
begin
    state_reg <= state_next;
end
```

```
40
41  // Next state logic
42
43  always @(*)
44  begin
45      case(state_reg)
46      s0: state_next <= s1;
47      s1: state_next <= s2;
48      s2: state_next <= (Op == BNE)? s0 : s3;
49      s3: state_next <= s0;
50      endcase
51  end
52
53  //output logic
54  assign PCWriteCond = (state_reg == s2) & (Op == BNE);
55
56  assign PCWrite = (state_reg == s0);
57
58  assign MemRead = (state_reg == s0);
59
60  assign IRWrite = (state_reg == s0);
61
62  assign RegWrite = ( state_reg == s3 ) & ( Op == ADDI );
63
64  assign PCSource = ( state_reg == s0 ) ? 2'b00 : ((Op==BNE) ? 2'b01 : 2'b10);
65
66  assign ALUOp = ((state_reg == s0)|(state_reg == s1)) ? 2'b00 :((state_reg == s2)&( Op==BNE )) ? 2'b01 : 2'b00;
67
68  assign ALUSrcB = (state_reg == s0) ? 2'b01 : (state_reg == s1) ? 2'b11 : ((state_reg == s2) & (Op == BNE))? 2'b00 : 2'b10;
69
70  assign ALUSrcA = ((state_reg== s0)|(state_reg == s1)) ? 2'b00 :2'b01;
71
72  endmodule
```

The Mealy state machine used to implement the control logic.

Building the top-level module of the processor:

Using the modules described above, we now proceed to connect them all together using wires in the top-level module of the processor that only takes the clock signal as input.

```
1   `include "Program_Counter.v"
2   `include "Instruction_Memory.v"
3   `include "Instruction_Register.v"
4   `include "Register_File.v"
5   `include "Sign_Extension.v"
6   `include "Shifter.v"
7   `include "A_reg.v"
8   `include "B_reg.v"
9   `include "Mult4to1.v"
10  `include "Alu_Core.v"
11  `include "ALU_Out.v"
12  `include "Control.v"
13  `include "or_and.v"
14  module Processor_Top(clk);
15  input clk;
16
17
18  wire [31:0] out_address;
19  wire [31:0] MemData;
20  wire [31:0] Instruction;
21  wire [31:0] read_data1;
22  wire [31:0] read_data2;
23  wire [31:0] bits32_out;
24  wire [31:0] shifted;
25  wire [31:0] A_reg_out;
26  wire [31:0] B_reg_out;
27  wire [31:0] alusrcA_out;
28  wire [31:0] alusrcB_out;
29  wire [31:0] alucore_out;
30  wire zero;
31  wire [31:0] WriteData;
32  wire [31:0] in_address;
33  wire write;
34
35  wire PCWriteCond, PCWrite, MemRead, IRWrite,RegWrite; //pcwritecond
36  wire [1:0] PCSource, ALUOp, ALUSrcB, ALUSrcA;
37
38  Program_Counter pc0(
39      .clk(clk),
40      .in_address(in_address),
41      .write(write),
42      .out_address(out_address)
```

```
    );

or_and or_and0(
    .zero(zero),
    .PCWriteCond(PCWriteCond),
    .PCWrite(PCWrite),
    .write(write)
    );

Instruction_Memory im0(
    .clk(clk),
    .MemRead(MemRead),
    .instrn_address(out_address),
    .MemOut(MemData)
    );

Instruction_Register ir0(
    .clk(clk),
    .MemData(MemData),
    .IRWrite(IRWrite),
    .Instruction(Instruction)
    );

//put control logic unit here

Control ctrl(
    .clk(clk),
    .Op(Instruction[31:26]),
    .PCWriteCond(PCWriteCond),
    .PCWrite(PCWrite),
    .MemRead(MemRead),
    .IRWrite(IRWrite),
    .RegWrite(RegWrite),
    .PCSource(PCSource),
    .ALUOp(ALUOp),
    .ALUSrcB(ALUSrcB),
    .ALUSrcA(ALUSrcA)

    );


```

```
 Register_File rf0(
     .clk(clk),
     .read_reg1(Instruction[25:21]),
     .read_reg2(Instruction[20:16]),
     .RegWrite(RegWrite),
     .write_reg(Instruction[20:16]),
     .write_data(WriteData),
     .read_data1(read_data1),
     .read_data2(read_data2)
     );

 Sign_Extension se0(
     .bits16_in(Instruction[15:0]),
     .bits32_out(bits32_out)
     );

 Shifter sh0(
     .indata(bits32_out),
     .shift_amt(2'd2),
     .shift_left(1'b1),
     .outdata(shifted)
     );

 A_reg a0(
     .clk(clk),
     .read_data1(read_data1),
     .alusrc_A(A_reg_out)
     );

 B_reg b0(
     .clk(clk),
     .read_data2(read_data2),
     .alusrc_B(B_reg_out)
     );

 Mult4to1 alusrcA(
         .in0(out_address),
         .in1(A_reg_out),
         .in2(), //
         .in3(), //
         .sel(ALUSrcA),
         .out(alusrcA_out)
     );

```

```
127
128   Mult4to1 alusrcB(
129           .in0(B_reg_out),
130           .in1(32'd4),
131           .in2(bits32_out), //
132           .in3(shifted),
133           .sel(ALUSrcB),
134           .out(alusrcB_out)
135       );
136
137   Alu_Core alu(
138       .A(alusrcA_out),
139       .B(alusrcB_out),
140       .alu_control(ALUOp),
141       .result(alucore_out),
142       .zero(zero)
143       );
144
145   ALU_Out aluout(
146       .clk(clk),
147       .ALU_result(alucore_out),
148       .WriteData(WriteData)
149       );
150
151   Mult4to1 pcsrc(
152           .in0(alucore_out),
153           .in1(WriteData),
154           .in2(), //
155           .in3(), //
156           .sel(PCSource),
157           .out(in_address)
158       );
159
160   endmodule
```

The top module nightmare (open wires are used in lower-order MUXs)

Writing the testbench:

The testbench for the top module is relatively simple since we only need to provide an input clock to the processor unit under test.

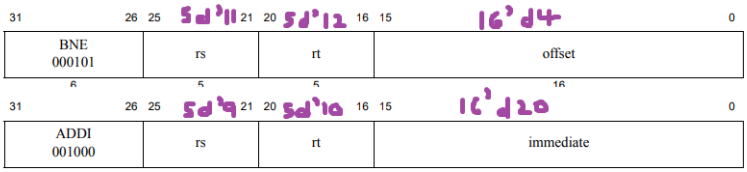
```
1  `include "Processor_Top.v"
2  module top_tb;
3  reg clk;
4  Processor_Top ut0(.clk(clk));
5
6  always begin
7      clk = ~clk;
8      #5;
9  end
10
11 initial begin
12     $dumpfile("ptop.vcd");
13     $dumpvars(0, top_tb);
14     clk = 1'b0;
15     #80;
16     $stop;
17 end
18
19 endmodule
20
21
```

Simulating the processor:

Note that both instructions do not take equal number of clock cycles – the BNE instruction only requires 3 clock cycles to complete whereas the ADDI instruction requires 4 instructions to complete.

I have used mem files to initialise the register and instruction memory using the *readmemh* operation. The mem files are of hexadecimal format.

The formats for the ADDI and BNE instructions are listed below I have used the registers t$1, t$2, t$3, and t$4 which correspond to the register locations 9, 10, 11 and 12.



Converting this to hexadecimal, the instructions are:

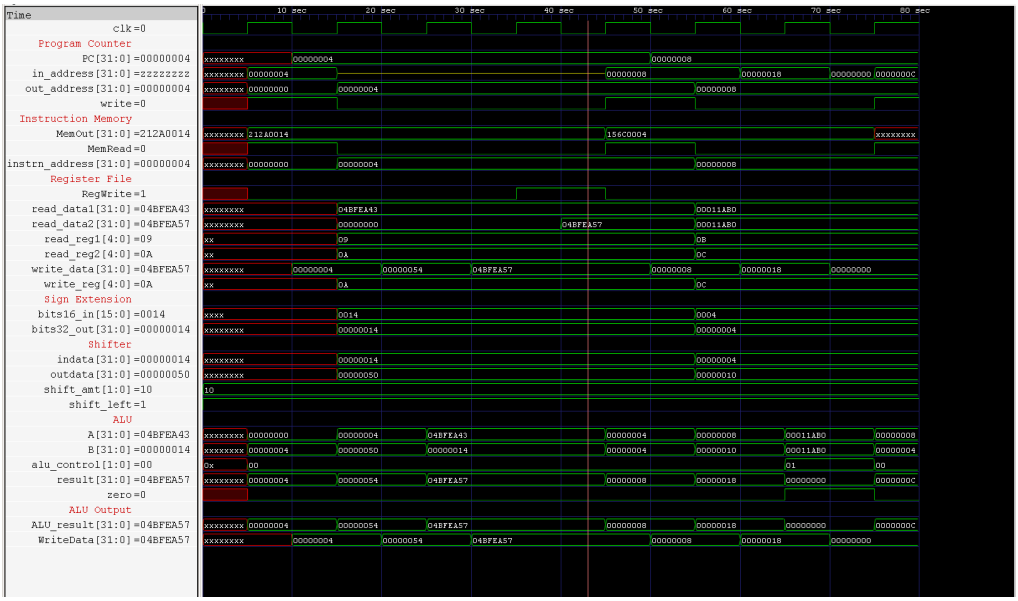
0000: 212A0014

0004: 156C0004

This is stored in the instruction_memory.mem file.

The register memory is also initialised in similar fashion with the values provided in the question.

The waveforms obtained are shown below.



Waveforms obtained for different registers (States are shown at the end of instruction 1)

The output waveforms are as expected: ADDI should produce 4BFEA43 + 14 = 4BFEA57 and the BNE instruction will not branch since both the registers are equal, which is reflected in the program counter not changing. The program completes in seven cycles – 4 for the ADDI instruction and 3 for the BNE instruction.