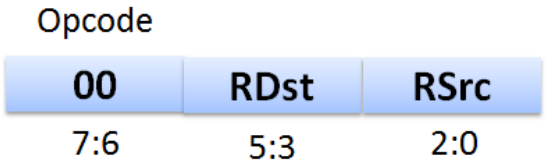
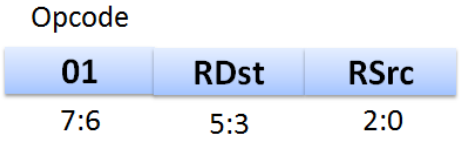
Birla Institute of Technology and Science – Pilani, Hyderabad Campus
Second Semester 2019-20
CS F342: Computer Architecture Assignment (20 Marks)

1. (a) Implement 4-stage pipelined processor in Verilog. This processor supports data transfer (mov), addition (add) and Unconditional Jump (J) instructions only. The processor should implement forwarding to resolve data hazards. The processor has Reset, CLK as inputs and no outputs. The processor has instruction fetch unit, register file (with 8 8-bit registers), Execution and Writeback unit. Read and write operations on Register file can happen simultaneously and should be independent of CLK. The processor also contains three pipelined registers IF/ID, ID/EX and EX/WB. When reset is activated the PC, IF/ID, ID/EX, EX/WB registers are initialized to 0, the instruction memory and registerfile get loaded by **predefined values**. When the instruction unit starts fetching the first instruction the pipeline registers contain unknown values. When the second instruction is being fetched in IF unit, the IF/ID registers will hold the instruction code for first instruction. When the third instruction is being fetched by IF unit, the IF/ID register contains the instruction code of second instruction, ID/EX register contains information related to first instruction and so on. (Assume 8-bit PC. Also Assume Address and Data size as 8-bits)
The instruction and its **8-bit instruction format** are shown below:
mov DestinationReg, SourceReg (Moves data in register specified by register number in Rsrc field to a register specified by register number in RDst field. Opcode for mov is 00)



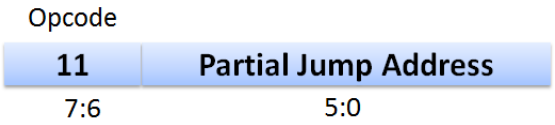
Example usage: mov R2, R0 (R2←R0)

add DestinationReg, SourceReg (adds data in register specified by register number in Rsrc field to data in register specified by register number in RDst field. Result is stored in register specified by register number in RDst field. Opcode for add is 01)



Example usage: add R2, R0 (R2←R2+ R0)

j L1 (Jumps to an address generated by appending 2 MSB bits of PC+1 to the data specified in instruction field (5:0). Opcode for j is 11)



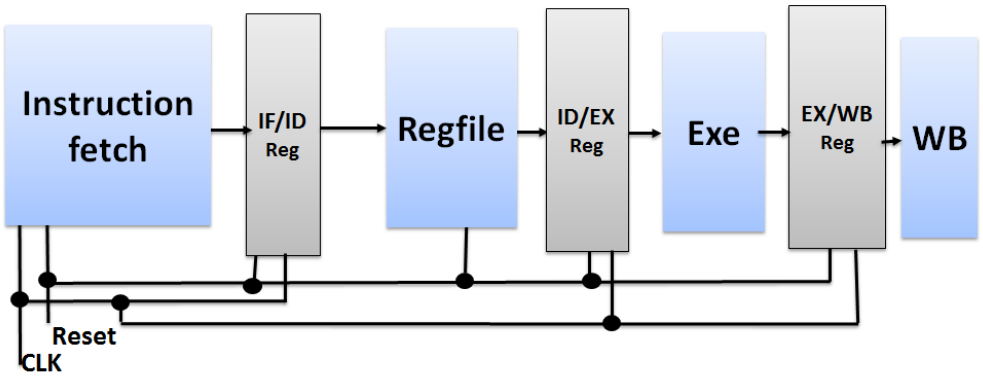
Example usage: j L1 (Jump address is calculated using Pseudo direct addressing)

Assume the register file contains 8 registers (R0-R7) each register can hold 8-bit data. On reset register file should get initialized such that R0 = 0, R1 = 1, R2 = 2, R3 = 3 …etc. On reset assume that the instruction memory gets initialized with four instructions.

```
        mov Rx, Ry
        add Ry, Rx
        add Rz, Ry
        j L1
        mov Rx, Rz
L1:     add Rx, Rz
```

Where x, y, z are related to last 3 digits of your ID No.
If ID number: 20XXXXXXABCH, then x =A mod 8 (A%8),
y= (B+2) mod 8 ((B+2)%8),
z= (C+3) mod 8 ((C+3)%8),

A partial block level representation of 4-stage pipelined processor is shown below. **Please note that for registerfile implementation, both read and write are independent of CLK**. Write operation depends on control signal.



As part of the assignment three files should be submitted in zipped folder.

1. PDF version of this Document with all the Questions below answered with file name as **IDNO_NAME.pdf**.

2. Design Verilog Files for all the Sub-modules (instruction fetch, Register file, forwarding unit).

3. Design Verilog file for the main processor.

The name of the zipped folder should be in the format IDNO_NAME.zip

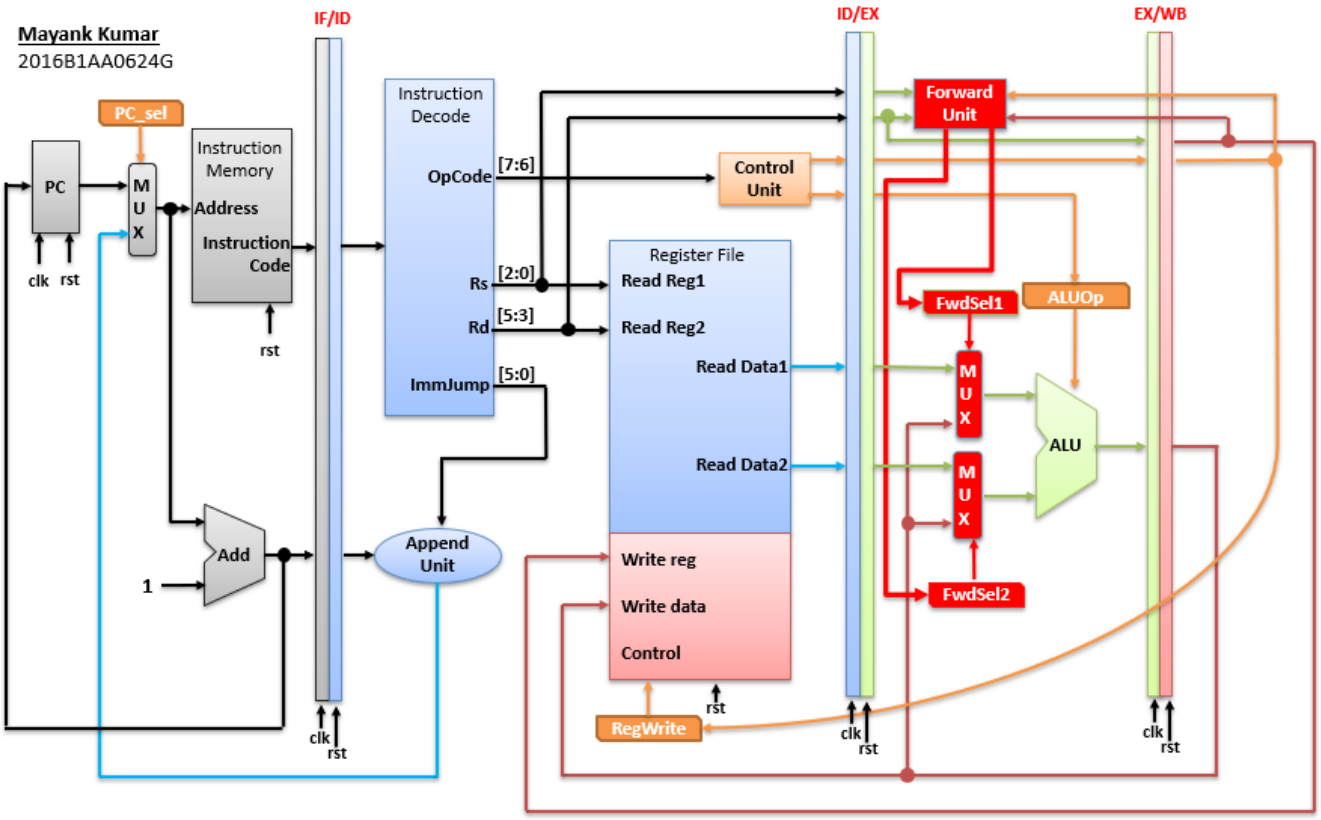
The due date for submission is 25-April-2020, 5:00 PM.

Name: Mayank Kumar **ID No:** 2016B1AA0624G

Questions Related to Assignment

1. **Draw the complete Datapath and show control signals of the 4-stage pipelined processor. A sample Datapath for 5-stage pipelined MIPS processor has been discussed in class. A ppt named Assignmenthelp.ppt contains this 5-stage processor and is uploaded in CMS. You can modify this according to your specification.**

Answer:



2. List the control signals used and also the values of control signals for different instructions.

Answer:

Instructions	**Control Signals**					
	ALUOp	**PC_Sel**	**Reg_Write**			
mov	0	0	1			
add	1	0	1			
j	X	1	0			

3. Implement the Instruction Fetch block. Copy the <u>image</u> of Verilog code of the Instruction fetch block here

Answer:

```
//*************************************INSTRUCTION MEMORY(USED BY IF)*************************************
module Instruction_Memory(
    input [7:0] PC,
    input reset,
    output [7:0] Instruction_Code
);
reg [7:0] Mem [80:0];  //Byte addresssable memory 36 locations

//For normal memory read we use the following statement
assign Instruction_Code= Mem[PC];

//Handling reset condition
always @(reset)
    begin

    if(reset == 0)//reset logic is 0
        begin


            //Instructions given by Assignment
            Mem[0] = 8'b00110010; //mov r6, r2
            Mem[1] = 8'b01010110; //add r2, r6
            Mem[2] = 8'b01100010; //add r4, r2
            Mem[3] = 8'b11000101; //j L1(or 5)
            Mem[4] = 8'b00110100; //mov r6, r4
            Mem[5] = 8'b01110100; //add r6, r4
            //Mem[6] = 8'b11000000; //j 0

        end
    end
endmodule
```

```
//*************************************PROGRAM COUNTER*************************************
module ProgramCounter(
input [7:0] PC_in,
input clk,
input reset,
output reg [7:0] PC_out
);

    always@(posedge clk, negedge reset)
        begin
        if(reset == 0)
            PC_out = 0;
        else
            PC_out = PC_in;
        end



endmodule
```

4. Implement the Register File and copy the <u>image</u> of Verilog code of Register file unit here.

Answer:

```
//**************************************REGISTER FILE**************************************
module Register_File(
    input [2:0] Read_Reg_Num_1,
    input [2:0] Read_Reg_Num_2,
    input [2:0] Write_Reg_Num,
    input [7:0] Write_Data,
    input RegWrite,
    input reset,

    output [7:0] Read_Data_1,
    output [7:0] Read_Data_2
);

    reg [7:0] RegMemory [7:0];

    //Handling reset condition
    always @(negedge reset)
        begin

        if(reset == 0)//reset logic is 0
            begin
            RegMemory[0] = 0;
            RegMemory[1] = 1;
            RegMemory[2] = 2;
            RegMemory[3] = 3;
            RegMemory[4] = 4;
```

```
        if(reset == 0)//reset logic is 0
            begin
            RegMemory[0] = 0;
            RegMemory[1] = 1;
            RegMemory[2] = 2;
            RegMemory[3] = 3;
            RegMemory[4] = 4;
            RegMemory[5] = 5;
            RegMemory[6] = 6;
            RegMemory[7] = 7;
            end
        end

    assign Read_Data_1 = RegMemory[Read_Reg_Num_1];
    assign Read_Data_2 = RegMemory[Read_Reg_Num_2];


    always @(RegWrite, Write_Reg_Num, Write_Data)
        begin
        if(RegWrite == 1)
            RegMemory[Write_Reg_Num] = Write_Data;

        end

endmodule
```

5. **Determine the condition that can be used to detect data hazard?**

Answer: A Data Hazard can be detected by checking the condition that
For Rd: (ID_EX_Rd == EX_WB_Rd AND (EX_WB_RegWrite == 1)
For Rs: (ID_EX_Rs == EX_WB_Rd AND (EX_WB_RegWrite == 1)

6. Implement the forwarding unit and copy the <u>image</u> of Verilog code of forwarding unit here.

Answer:

```
//***********************************FORWARDING UNIT**********************************
module ForwardUnit(
input [2:0] Rin1,
input [2:0] Rin2,
input [2:0] RDest,
input PL_EX_WB_RegWrite,
output reg forwardMuxSel1,
output reg forwardMuxSel2
);
    always@(*)
        begin
        if((Rin1 == RDest) && (PL_EX_WB_RegWrite == 1))
            forwardMuxSel1= 1;//if Rs register and output register of EX are same,then forawrding is required
        else
            forwardMuxSel1= 0;
        if((Rin2 == RDest) && (PL_EX_WB_RegWrite == 1))
            forwardMuxSel2= 1;//if Rd register and output register of EX are same, then forawrding is required
        else
            forwardMuxSel2= 0;
        end


endmodule
```

7. Implement complete processor in Verilog (using all the Datapath blocks). Copy the <u>image</u> of Verilog code of the processor here. (Use comments to describe your Verilog implementation)

Answer:

```
//***********************************MAIN PROCESSOR IMPLEMENTATION**********************************
module processor(
    input clk,
    input reset
);

    //Variable definitions

    //Pipeline registers are named as PL_STAGE1_STAGE2_NAME,
    //STAGE1, STAGE2 refer to the two stages the pipeline is in between and NAME refers to what value it should contain

    //In case of connecting wires, they are named as STAGE_NAME

    //For the PC
    wire [7:0] PC_Addr;
    wire [7:0] PC_Mux_out, PC_add, PC_append;

    wire [7:0] Ins_Code;//Output code directly from Instruction Memory

    //IF ---> ID

    reg [7:0] PL_IF_ID_PC_1;//PC+1 value stored
    reg [7:0] PL_IF_ID_Ins_code;//Code stored in the IF/ID pipeline register

    //In ID stage, there will be a Rd, Rs, imm(for jump) and control signals generated.
    wire [2:0] ID_Rs, ID_Rd;
    wire [5:0] ID_jmp_imm;
    wire [1:0] ID_OpCode;

    wire [7:0] ID_RegFile_Rs, ID_RegFile_Rd;//Outputs of the register file
```

```
//These are generated immediately by the control unit, and have to be stored to move it forward
wire ALUOp;
wire RegWrite;


//ID ---> EX

reg PL_ID_EX_ALUOp;
reg PL_ID_EX_RegWrite;
reg [2:0] PL_ID_EX_Rs;
reg [2:0] PL_ID_EX_Rd;
reg [7:0] PL_ID_EX_RegFile_Rs;
reg [7:0] PL_ID_EX_RegFile_Rd;

wire [7:0] EX_ALUin1;
wire [7:0] EX_ALUin2;

wire [7:0] EX_ALU_Out;
wire ALU_Zero_val;

//For forawrding unit
wire FwdSel1, FwdSel2;

//EX ---> WB
reg PL_EX_WB_RegWrite;
reg [2:0] PL_EX_WB_Rd;
reg [7:0] PL_EX_WB_ALU_Out;
```

```
//Defining DataBlocks here


//--------IF STAGE
//Get the address of the next instruction to be fetched
ProgramCounter PC(PC_add,clk,reset, PC_Addr);
PCAdder PC_PC1(PC_Mux_out, PC_add);
PCAppender PC_Jmp(PL_IF_ID_PC_1, ID_jmp_imm, PC_append);
mux_2x1 PC_Mux(PC_append, PC_Addr, RegWrite, PC_Mux_out);


//Put it in IM, to get code
Instruction_Memory IM(PC_Mux_out, reset, Ins_Code);


//--------ID STAGE
Instruction_Decode ID(PL_IF_ID_Ins_code, ID_Rd, ID_Rs, ID_jmp_imm, ID_OpCode);
ControlUnit CU(ID_OpCode, ALUOp, RegWrite);
Register_File RF(ID_Rs, ID_Rd, PL_EX_WB_Rd, PL_EX_WB_ALU_Out, PL_EX_WB_RegWrite, reset,ID_RegFile_Rs, ID_RegFile_Rd );


//--------EX STAGE
ForwardUnit Forward(PL_ID_EX_Rs, PL_ID_EX_Rd, PL_EX_WB_Rd,PL_EX_WB_RegWrite, FwdSel1, FwdSel2);
mux_2x1 FwdMux1(PL_ID_EX_RegFile_Rs,PL_EX_WB_ALU_Out,FwdSel1,EX_ALUin1);
mux_2x1 FwdMux2(PL_ID_EX_RegFile_Rd,PL_EX_WB_ALU_Out,FwdSel2,EX_ALUin2);
ALU EX(EX_ALUin1, EX_ALUin2, PL_ID_EX_ALUOp, ALU_Zero_val, EX_ALU_Out);


//--------WB STAGE
//No datablock
```

```
//Order is backwards to ensure correct values go to the next stage
//---------------EX_WB PIPELINE
always@(posedge clk, negedge reset)
begin
    if(reset == 0)
        begin
        PL_EX_WB_RegWrite = 0;
        PL_EX_WB_ALU_Out = 0;
        PL_EX_WB_Rd = 0;
        //Assign everything zero here
        end
    else
        begin
        PL_EX_WB_RegWrite = PL_ID_EX_RegWrite;
        PL_EX_WB_ALU_Out = EX_ALU_Out;
        PL_EX_WB_Rd = PL_ID_EX_Rd;
        end
end
```

```
    //---------------ID_EX PIPELINE
    always@(posedge clk, negedge reset)
    begin
        if(reset == 0)
            begin
            PL_ID_EX_ALUOp = 0;
            PL_ID_EX_RegWrite = 0;
            PL_ID_EX_Rs = 0;
            PL_ID_EX_Rd = 0;
            PL_ID_EX_RegFile_Rs = 0;
            PL_ID_EX_RegFile_Rd = 0;
            //Assign everything zero here
            end
        else
            begin
            PL_ID_EX_ALUOp = ALUOp;
            PL_ID_EX_RegWrite = RegWrite;
            PL_ID_EX_Rs = ID_Rs;
            PL_ID_EX_Rd = ID_Rd;
            PL_ID_EX_RegFile_Rs = ID_RegFile_Rs;
            PL_ID_EX_RegFile_Rd = ID_RegFile_Rd;
            end
    end
```

```
    //---------------IF_ID PIPELINE
    always@(posedge clk, negedge reset)
    begin
        if(reset == 0)
            begin
            //Assign everything zero here
            PL_IF_ID_PC_1 = 0;
            PL_IF_ID_Ins_code = 0;

            end
        else
            begin
            PL_IF_ID_Ins_code = Ins_Code;
            PL_IF_ID_PC_1 = PC_add;
            end
    end

endmodule
```

MISCELLANEUOS BLOCKS

```
//*************************************CONTROL UNIT*************************************
module ControlUnit(input [1:0] OpCode,
output ALUOp,
//output jmp,
output RegWrite
);

assign ALUOp = OpCode[0];
assign RegWrite = !OpCode[1];
//assign jmp = OpCode[0] & OpCode[1];
endmodule

//*************************************INSTRUCTION DECODE*************************************
module Instruction_Decode(input [7:0] Ins_code,
output [2:0] ID_Rd,
output [2:0] ID_Rs,
output [5:0] ID_jmp_imm,
output [1:0] ID_OpCode);

assign ID_OpCode = Ins_code[7:6];
assign ID_Rd = Ins_code[5:3];
assign ID_Rs = Ins_code[2:0];
assign ID_jmp_imm = Ins_code[5:0];

endmodule
```

```
//************************************8 BIT MUX*************************************
module mux_2x1(
  input [7:0] in1,in2,
  input select,
  output reg [7:0] out);

  always@(in1, in2, select)
  begin
    if(select == 1'b0)
      out = in1;
    else
      out = in2;
  end
endmodule

//************************************PC ADDER*************************************
module PCAdder(input [7:0] PC,
               output [7:0] PCAdd);

assign PCAdd = PC + 1;

endmodule

//************************************PC APPENDER*************************************
module PCAppender(input [7:0] PC,
                  input [5:0] PC_append_val,
                  output [7:0] PC_append);

assign PC_append = {PC[7:6], PC_append_val};
endmodule
```

```
//************************************ALU*************************************
module ALU(
    input signed [7:0]A,
    input signed [7:0]B,
    input control,
    output reg zero_val,
    output reg signed [7:0]out
    );

    always@(A or B or control)
    begin
    case(control)

        0: out = A;
        1: out = A+B;

    endcase
    end


    always@(out)
    begin
        if(out == 0)
            zero_val = 1;
        else
            zero_val = 0;
    end
endmodule
```

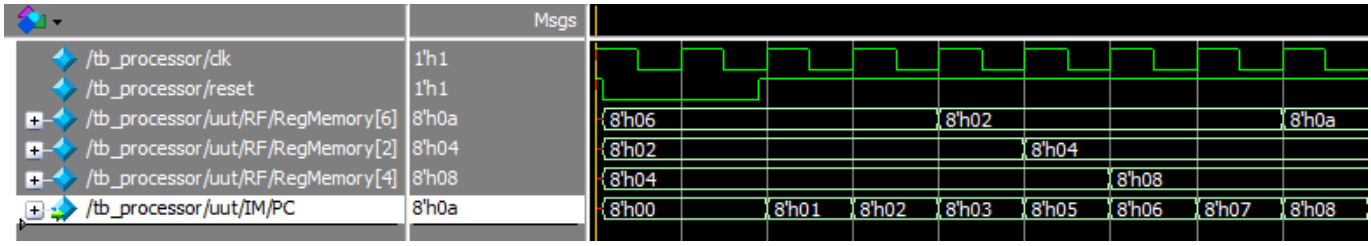
8. Test the processor design by generating the appropriate clock and reset. Copy the <u>image</u> of your testbench code here.

Answer:

```
module tb_processor;
reg clk;
reg reset;


    processor uut(clk, reset);
    //Get the clock up
    initial
    begin
        clk = 1;
        forever #5 clk = ~clk;
    end


    //reset a few times
    initial
    begin
            reset = 1;
        #1  reset = 0;
        #18  reset = 1;
        #400 reset = 0;
        #10  reset = 1;
    end


endmodule
```

9. Verify if the register file is getting updated according to the set of instructions (mentioned earlier).

Copy verified **<u>Register file</u>** waveform here (show only the Registers that get updated, CLK, and RESET):



Unrelated Questions

What were the problems you faced during the implementation of the processor?

Answer: Getting the order of moving values into the pipeline correct was a major problem I faced. It was hard to debug also. I also faced a few minor bugs which I managed to solve eventually.

Did you implement the processor on your own? If you took help from someone whose help did you take? Which part of the design did you take help for?

Answer: Yes, I implemented the complete processor on my own.
I asked Chetan sir a few doubts about implementing the jump instruction.

Honor Code Declaration by student:

- My answers to the above questions are my own work.
- I have not shared the codes/answers written by me with any other students. (I might have helped clear doubts of other students).
- I have not copied other's code/answers to improve my results. (I might have got some doubts cleared from other students).

Name: Mayank Kumar **Date:** 21-Apr-2020
ID No.: 2016B1AA0624G