IPA Project Report

Chetanya Goyal - 2021112009 Ajit S - 2021112023

AIM

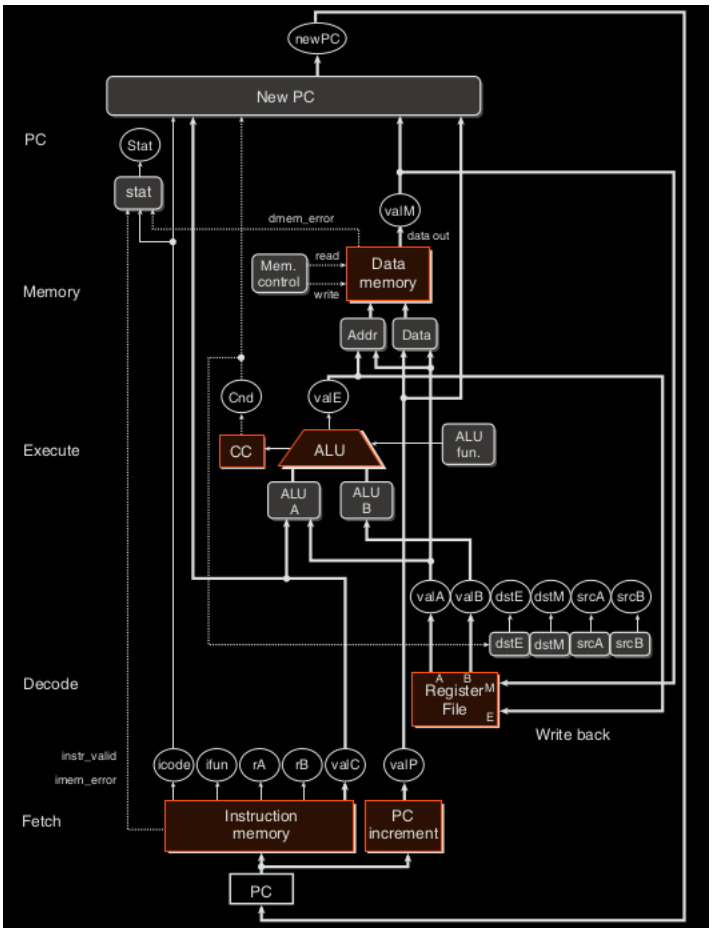
- To create a simple 4 function 64 bit ALU (Add, Subtract, AND, XOR) with an overflow detection
- To use the ALU to make a sequential Y86-64 CISC processor
- To implement pipelining and possibly handle data hazards
- To check the module's functioning with testbenches
- To simulate the testbenches with the GTKWave software

Requirements

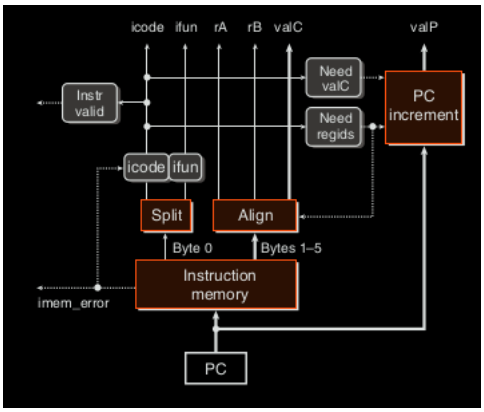
- A 64 - bit Adder block with the ability to add 2 64 bit signed 2's complement numbers
- A 64 - bit Subtractor block with the ability to subtractor 2 64 bit signed 2's complement numbers
- A 64 - bit AND block that performs the bitwise AND of two 64 bit 2's complement numbers

- A 64 - bit XOR block that performs the bitwise XOR of two 64 bit 2’s complement numbers
- Fetch, Decode, Execute, Memory, Writeback and PC_update blocks for the sequential implementation
- Pipeline registers for each stage (i.e. fetch, decode, execute, memory, writeback, PC)

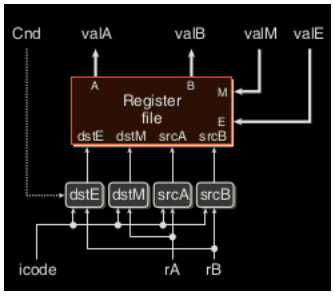
Overview



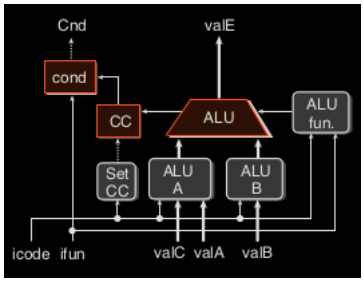
The sequential Y86 implementation follows a simple pipeline architecture consisting of five stages: fetch, decode, execute, memory, and writeback.



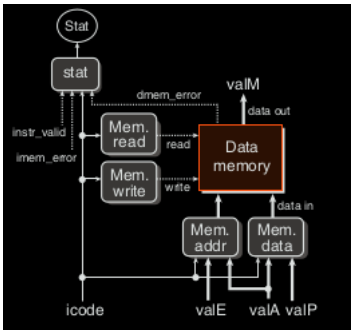
In the *fetch* stage, the instruction address is fetched from memory and passed to the *decode* stage.



In the *decode* stage, the instruction is decoded, and the necessary register values and immediate values are read from the register file and passed on to the *execute* stage.



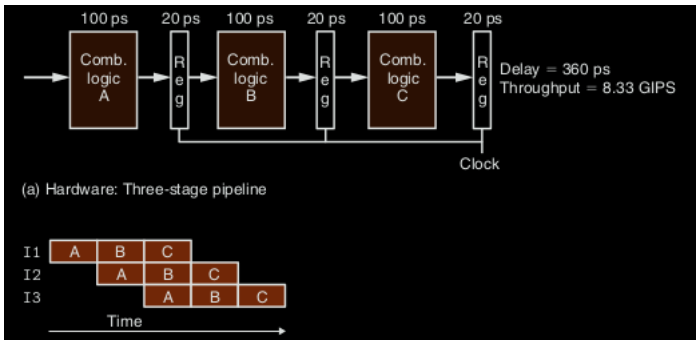
In the *execute* stage, the arithmetic and logical operations are performed, and the condition codes are computed. The computed values are then passed on to the *memory* stage.

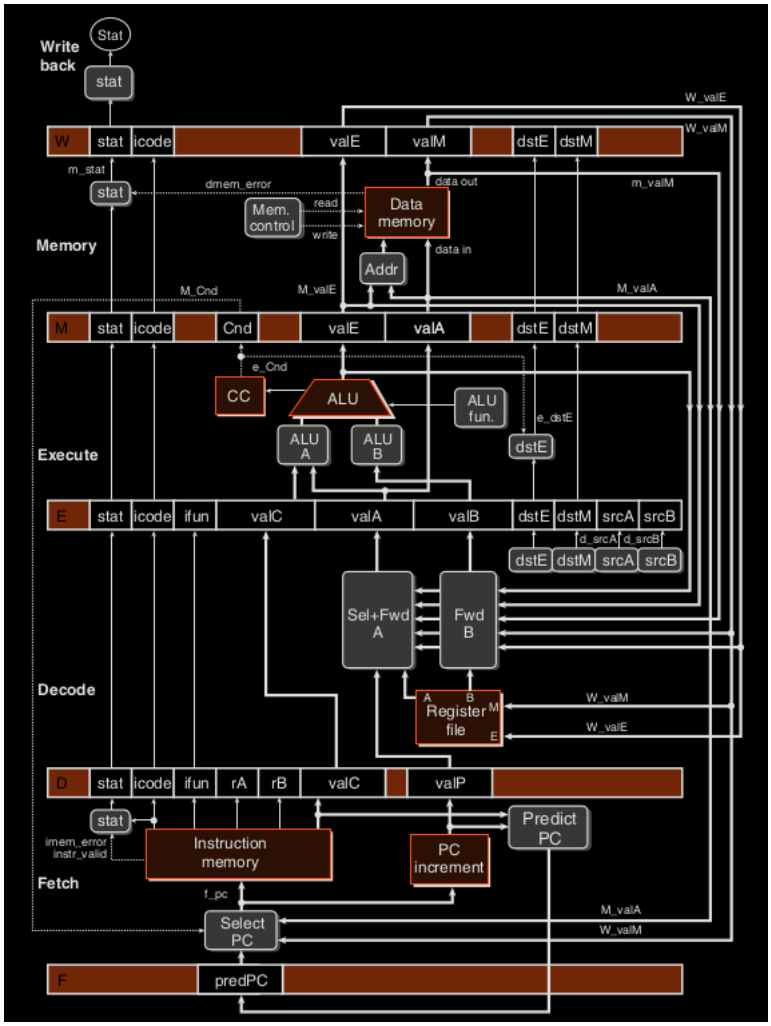


In the *memory* stage, memory operations such as load and store are performed. The data is read from or written to memory and passed on to the *writeback* stage.

In the *writeback* stage, the final result is written to the register file.

A pipelined Y86 processor is a more complex architecture that allows for overlapping of instructions. It also consists of five stages: fetch, decode, execute, memory, and writeback.





In the *fetch* stage, the instruction address is fetched from memory, and the instruction is sent to the *decode* stage.

In the *decode* stage, the instruction is decoded and the necessary register values and immediate values are read from the register file. The decoded instruction is then passed on to the *execute* stage.

In the *execute* stage, the arithmetic and logical operations are performed, and the condition codes are computed. The computed values are then passed on to the *memory* stage.

In the *memory* stage, memory operations such as load and store are performed. The data is read from or written to memory and passed on to the *writeback* stage.

In the *writeback* stage, the final result is written to the register file. However, in a pipelined architecture, the next instruction is already in the fetch stage, waiting to be processed. The pipeline allows for multiple instructions to be processed simultaneously, resulting in a more efficient processor.

Data dependencies can cause problems in pipelining, as some instructions may depend on the results of previous instructions. These dependencies are handled by inserting nops after the required instructions, wherever registers are being written to.

Branch mispredictions are also a problem in pipelining, as the pipeline may have already fetched the next instructions based on an incorrect jump. This is handled by the required control logic change, wherein if the execute cycle's computed condition codes show that the jump branch was mispredicted, it cancels the next fetched instructions and goes to the next instruction which it should have.

The pipelined Y86 processor can also lead to an increase in latency compared to the sequential implementation. This is because in the pipelined architecture, each instruction is divided into multiple stages, and each stage takes a certain amount of time to complete. This overhead in time for each stage, as well as the need for additional pipeline registers and control logic, can increase the overall latency of the processor. Additionally, data dependencies and branch mispredictions can cause pipeline stalls, where the pipeline must wait for previous instructions to complete before continuing, leading to further increases in latency. Despite this increase in latency, the increase in throughput provided by pipelining often outweighs the increased latency, resulting in a faster overall processor.

SEQUENTIAL

The sequential implementation of the Y86 CISC processor involves the use of an ALU to perform operations such as addition, subtraction, bitwise AND, and bitwise XOR. The ALU is used in conjunction with fetch, decode, execute, memory, writeback, and PC_update blocks to create a sequential processor.

For the ALU, a 64-bit adder and subtractor were created using cascaded full and half adders. The AND and XOR operations were performed using 64-bit AND and XOR gates, respectively. The ALU block uses a control input, `select`, which is a 2-bit input that determines which operation is performed.

The functioning of the processor was tested with testbenches and simulated with the GTKWave software.

ADD

- [ ] The `select` input was used a 2 bit input with -
  - [ ] `2'b00` for ADD
  - [ ] `2'b01` for SUB
  - [ ] `2'b10` for AND
  - [ ] `2'b11` for XOR

  - A simple cascade of 64 Full Adder blocks, which were created by cascading two Half Adder blocks with an added OR gate for the Carry-Out bit.

  - For the sake of simplicity, the Adder and the subtractor were, for all intents and purposes, the same design, with only the control bit changing from 0 (for Add functionality) to 1 (for subtract functionality).
  - The `generate` block was used to generate 64 adders and 64 XOR gates, with a for-loop.
  - The overflow bit was taken care of with an XOR of the last two Carry-Outs.

```
module halfadder(ahalf, bhalf, shalf, chalf);

  input ahalf;
  input bhalf;
  output shalf;
  output chalf;
```

```
xor x1(shalf, ahalf, bhalf);
and a1(chalf, ahalf, bhalf);

endmodule

module fulladder(afull, bfull, cfullin, sfull, cfull);

input afull;
input bfull;
input cfullin;
output sfull;
output cfull;
wire x, y, z;

halfadder h1(afull, bfull, x, y);
halfadder h2(cfullin, x, sfull, z);
or o1(cfull, y, z);

endmodule

module ADD(a, b, s, overflow);

input [63:0] a;
input [63:0] b;
output [63:0] s;
output overflow;

wire control;
wire [63:0] xorout;
wire [63:0] carry;

//control 1 for subtraction, 0 for addition

assign control = 1'b0;
genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
        begin
            xor x(xorout[i], b[i], control);
        end
endgenerate

genvar j;
generate
    for (j = 0; j < 64; j = j + 1)
    begin
        if (j == 0)
            fulladder f(a[j], xorout[j], control, s[j], carry[j]);
        else
            fulladder f(a[j], xorout[j], carry[j - 1], s[j], carry[j]);
    end
endgenerate

//assign overflow = (a[3] ^ b[3]) ? 0 : (s[3] ^ a[3]);
xor x4 (overflow, carry[63], carry[62]);

endmodule
```

SUB

- The same circuit as the Adder was used, with only the control bit changing from a 0 (2'b00) to a 1 `(2'b01)`
- The overflow bit was taken care of with an XOR of the last two Carry-Outs.
- The circuit does the following computation in terms of the inputs from the above diagram -
  - if B is negative -

    $A - (-B) = A + B$
  - if B is positive -

    $A - (B) = A - B$

```
module halfadder(ahalf, bhalf, shalf, chalf);

input ahalf;
input bhalf;
output shalf;
output chalf;

xor x1(shalf, ahalf, bhalf);
and a1(chalf, ahalf, bhalf);

endmodule

module fulladder(afull, bfull, cfullin, sfull, cfull);

input afull;
input bfull;
input cfullin;
output sfull;
output cfull;
wire x, y, z;

halfadder h1(afull, bfull, x, y);
halfadder h2(cfullin, x, sfull, z);
or o1(cfull, y, z);

endmodule

module SUB(a, b, d, overflow);

input [63:0] a;
input [63:0] b;
output [63:0] d;
output overflow;

wire control;
wire [63:0] xorout;
wire [63:0] carry;

//control 1 for subtraction, 0 for addition

assign control = 1'b1;
genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
        begin
            xor x(xorout[i], b[i], control);
        end
endgenerate

genvar j;
generate
    for (j = 0; j < 64; j = j + 1)
    begin
        if (j == 0)
            fulladder f(a[j], xorout[j], control, d[j], carry[j]);
        else
            fulladder f(a[j], xorout[j], carry[j - 1], d[j], carry[j]);
    end
endgenerate
//assign overflow = (a[3] ^ b[3]) ? 0 : (d[3] ^ a[3]);

xor x4 (overflow, carry[63], carry[62]);

endmodule
```

AND

- The circuit uses 64 AND gates for the bitwise AND of the 2 input numbers
- The AND gates were generated using the `generate` block.

```
module AND(a, b, out);

// inputs A and B
// output out
```

```
// bitwise AND operation

input [63:0] a;
input [63:0] b;
output [63:0] out;

genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
    begin
        and a(out[i], a[i], b[i]);
    end
endgenerate

endmodule
```

XOR

- The circuit uses 64 XOR gates for the bitwise XOR of the 2 input numbers
- The XOR gates were generated using the `generate` block.

```
module XOR(a, b, out);

// inputs A and B
// output out
// bitwise XOR operation

input [63:0] a;
input [63:0] b;
output [63:0] out;

genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
    begin
        xor x(out[i], a[i], b[i]);
    end
endgenerate

endmodule
```

ALU

- The ALU was made by using the above 4 blocks with the added control bit (named `select` here)

- [ ] The `select` input was used a 2 bit input with -
    - [ ] `2'b00` for ADD
    - [ ] `2'b01` for SUB
    - [ ] `2'b10` for AND
    - [ ] `2'b11` for XOR

Here, `select` is the control input

- The ALU block uses an always statement which triggers the output change at the change of any input (A, B, or $Select$)

```
/*-------------------------------------------------------------------*/
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*                       ALU                                         */
/*                       CHETANYA GOYAL                              */
/*                       AJIT S                                      */
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*-------------------------------------------------------------------*/
```

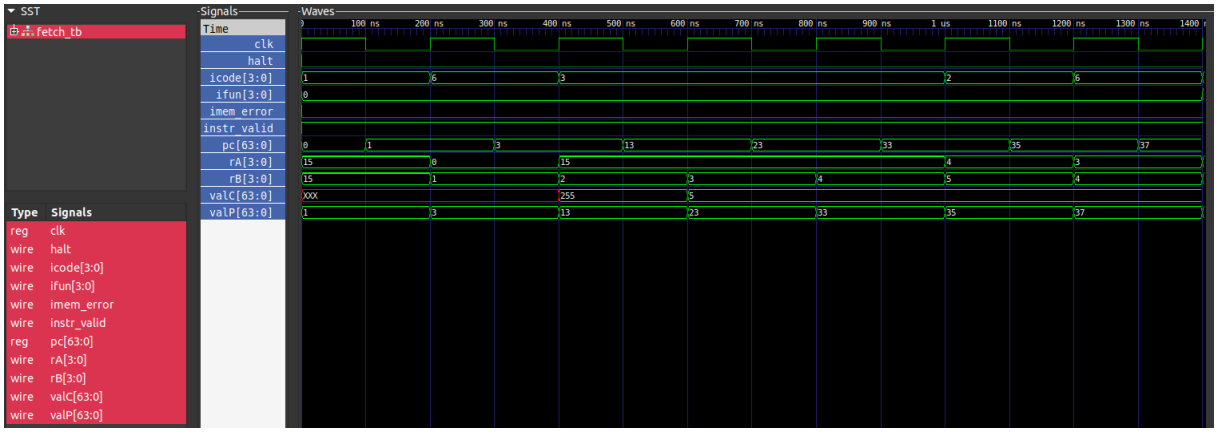
```
module halfadder(ahalf, bhalf, shalf, chalf);

input ahalf;
input bhalf;
output shalf;
output chalf;

xor x1(shalf, ahalf, bhalf);
and a1(chalf, ahalf, bhalf);

endmodule

module fulladder(afull, bfull, cfullin, sfull, cfull);

input afull;
input bfull;
input cfullin;
output sfull;
output cfull;
wire x, y, z;

halfadder h1(afull, bfull, x, y);
halfadder h2(cfullin, x, sfull, z);
or o1(cfull, y, z);

endmodule

module ADD(a, b, s, overflow);

input [63:0] a;
input [63:0] b;
output [63:0] s;
output overflow;

wire control;
wire [63:0] xorout;
wire [63:0] carry;

//control 1 for subtraction, 0 for addition

assign control = 1'b0;
genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
        begin
            xor x(xorout[i], b[i], control);
        end
endgenerate

genvar j;
generate
    for (j = 0; j < 64; j = j + 1)
    begin
        if (j == 0)
            fulladder f(a[j], xorout[j], control, s[j], carry[j]);
        else
            fulladder f(a[j], xorout[j], carry[j - 1], s[j], carry[j]);
    end
endgenerate

//assign overflow = (a[3] ^ b[3]) ? 0 : (s[3] ^ a[3]);
xor x4 (overflow, carry[63], carry[62]);

endmodule


module SUB(a, b, d, overflow);

input [63:0] a;
input [63:0] b;
output [63:0] d;
output overflow;

wire control;
wire [63:0] xorout;
wire [63:0] carry;

//control 1 for subtraction, 0 for addition

assign control = 1'b1;
```

```
genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
        begin
            xor x(xorout[i], b[i], control);
        end
endgenerate

genvar j;
generate
    for (j = 0; j < 64; j = j + 1)
    begin
        if (j == 0)
            fulladder f(a[j], xorout[j], control, d[j], carry[j]);
        else
            fulladder f(a[j], xorout[j], carry[j - 1], d[j], carry[j]);
    end
endgenerate
//assign overflow = (a[3] ^ b[3]) ? 0 : (d[3] ^ a[3]);

xor x4 (overflow, carry[63], carry[62]);

endmodule

module AND(a, b, out);

// inputs A and B
// output out
// bitwise AND operation

input [63:0] a;
input [63:0] b;
output [63:0] out;

genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
    begin
        and a(out[i], a[i], b[i]);
    end
endgenerate

endmodule

module XOR(a, b, out);

// inputs A and B
// output out
// bitwise XOR operation

input [63:0] a;
input [63:0] b;
output [63:0] out;

genvar i;
generate
    for (i = 0; i < 64; i = i + 1)
    begin
        xor x(out[i], a[i], b[i]);
    end
endgenerate

endmodule

module ALU(a, b, s, select, overflow);

input [63:0] a;
input [63:0] b;
input [1:0] select;

output reg [63:0] s;
output reg overflow;

wire signed [63:0] outtempadd;
wire signed [63:0] outtempsub;
wire signed [63:0] outtempand;
wire signed [63:0] outtempxor;
wire signed overflowtempadd, overflowtempsub;

ADD a1(a, b, outtempadd, overflowtempadd);
SUB s1(a, b, outtempsub, overflowtempsub);
```

```
AND and11(a, b, outtempand);
XOR x11(a, b, outtempxor);

always @(*)
begin
    if (select == 2'b00)
    begin
        s = outtempadd; //add
        overflow = overflowtempadd;
    end
    else if (select == 2'b01)
    begin
        s = outtempsub; //sub
        overflow = overflowtempsub;
    end
    else if (select == 2'b10)
    begin
        s = outtempand; //AND
    end
    else if (select == 2'b11)
    begin
        s = outtempxor; //XOR
    end
end

endmodule
```

FETCH

- The FETCH module was implemented as described in class lectures and the reference book.
- It takes the input PC value and finds the value of $icode$, $ifun$, rA, rB, $valC$ and $valP$
- These values are supplied as outputs to the next stage - DECODE
- The computations are positive edge clock triggered
- Additionally, the $halt$ and $imem\_error$ outputs were also decided here.
- It uses an 80 bit instruction as a temporary register and a instruction memory of size 120 $bytes$, declared as a temporary register and coded in little endian (byte-reversed order).
- If the instruction received from the PC values (64 bit) and the $imem\_error$ bit was set to 1 in case of a PC overshoot or overflow and the $instr\_valid$ bit was set to 0 if $icode$ does not correspond to any instruction.



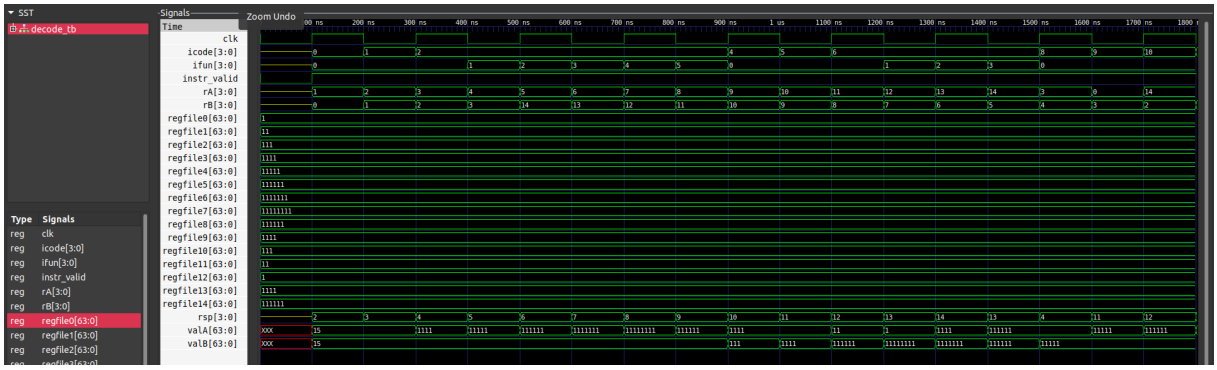

```
module fetch (clk, pc, icode, ifun, rA, rB, valC, valP,halt, instr_valid, imem_error);

input clk;
input [63:0] pc;
output reg [3:0] icode;
output reg [3:0] ifun;
output reg [3:0] rA;
output reg [3:0] rB;
```

```
output reg [63:0] valC;
output reg [63:0] valP;
output reg halt;
output reg instr_valid;
output reg imem_error;
reg [0:79] instr;
reg [0:7] instr_mem [0:119];  // little endian

always @(posedge clk)
begin
    instr_valid = 1'b1;
    imem_error = 1'b0;
    halt = 1'b0;

    if(pc > 119)
    begin
        imem_error = 1'b1;
    end

    instr_mem[0]  = 8'b00010000; // nop instruction pc = pc +1 = 1
    instr_mem[1]  = 8'b01100000; // Opq add
    instr_mem[2]  = 8'b00000001; // rA = 0, rB = 1; pc = pc + 2 = 3

    instr_mem[3]  = 8'b00110000; // irmovq instruction pc = pc + 10 = 13
    instr_mem[4]  = 8'b11110010; // F, rB = 2;
    instr_mem[5]  = 8'b11111111; // 1st byte of V = 255, rest all bytes will be zero
    instr_mem[6]  = 8'b00000000; // 2nd byte
    instr_mem[7]  = 8'b00000000; // 3rd byte
    instr_mem[8]  = 8'b00000000; // 4th byte
    instr_mem[9]  = 8'b00000000; // 5th byte
    instr_mem[10] = 8'b00000000; // 6th byte
    instr_mem[11] = 8'b00000000; // 7th byte
    instr_mem[12] = 8'b00000000; // 8th byte (This completes irmovq)

    instr_mem[13] = 8'b00110000; // irmovq instruction pc = pc + 10 = 23
    instr_mem[14] = 8'b11110011; // F, rB = 3;
    instr_mem[15] = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    instr_mem[16] = 8'b00000000; // 2nd byte
    instr_mem[17] = 8'b00000000; // 3rd byte
    instr_mem[18] = 8'b00000000; // 4th byte
    instr_mem[19] = 8'b00000000; // 5th byte
    instr_mem[20] = 8'b00000000; // 6th byte
    instr_mem[21] = 8'b00000000; // 7th byte
    instr_mem[22] = 8'b00000000; // 8th byte (This completes irmovq)

    instr_mem[23] = 8'b00110000; // irmovq instruction pc = pc + 10 = 33
    instr_mem[24] = 8'b11110100; // F, rB = 4;
    instr_mem[25] = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    instr_mem[26] = 8'b00000000; // 2nd byte
    instr_mem[27] = 8'b00000000; // 3rd byte
    instr_mem[28] = 8'b00000000; // 4th byte
    instr_mem[29] = 8'b00000000; // 5th byte
    instr_mem[30] = 8'b00000000; // 6th byte
    instr_mem[31] = 8'b00000000; // 7th byte
    instr_mem[32] = 8'b00000000; // 8th byte (This completes irmovq)

    instr_mem[33] = 8'b00100000; // rrmovq // pc = pc + 2 = 35
    instr_mem[34] = 8'b01000101; // rA = 4; rB = 5;

    instr_mem[35] = 8'b01100000; // Opq add // pc = pc + 2 = 37
    instr_mem[36] = 8'b00110100; // rA = 3 and rB = 4, final value in rB(4) = 10;

    instr_mem[37] = 8'b00100101; // cmovge // pc = pc + 2 = 39
    instr_mem[38] = 8'b01010110; // rA = 5; rB = 6;

    instr_mem[39] = 8'b01100001; // Opq subq // pc = pc + 2 = 41
    instr_mem[40] = 8'b00110101; // rA = 3, rB = 5; both are equal

    instr_mem[41] = 8'b01110011; //je // pc = pc + 9 = 50
    instr_mem[42] = 8'b00110100; // Dest = 52; 1st byte
    instr_mem[43] = 8'b00000000; // 2nd byte
    instr_mem[44] = 8'b00000000; // 3rd byte
    instr_mem[45] = 8'b00000000; // 4th byte
    instr_mem[46] = 8'b00000000; // 5th byte
    instr_mem[47] = 8'b00000000; // 6th byte
    instr_mem[48] = 8'b00000000; // 7th byte
    instr_mem[49] = 8'b00000000; // 8th byte

    instr_mem[50] = 8'b00010000; // nop
    instr_mem[51] = 8'b00010000; // nop
```

```
    instr_mem[52] = 8'b01100000; // Opq add
    instr_mem[53] = 8'b00110101; // rA = 3; rB = 5;

    instr_mem[54] = 8'b10000000; //8 0 //call
    instr_mem[55] = 8'b00000000; //Dest
    instr_mem[56] = 8'b00000000; //Dest
    instr_mem[57] = 8'b00000000; //Dest
    instr_mem[58] = 8'b00000000; //Dest
    instr_mem[59] = 8'b00000000; //Dest
    instr_mem[60] = 8'b00000000; //Dest
    instr_mem[61] = 8'b00000000; //Dest
    instr_mem[62] = 8'b00000000; //Dest

    instr_mem[63] = 8'b10010000; // 9 0 //ret
    instr_mem[64] = 8'b00000000; // halt

    instr = {instr_mem[pc], instr_mem[pc+1], instr_mem[pc+2], instr_mem[pc+3], instr_mem[pc+4], instr_mem[pc+5], instr_mem[pc+6], instr

    icode = instr[0:3];
    ifun = instr[4:7];

    rA = instr[8:11];
    rB = instr[12:15];

    if (icode == 4'd0)
    begin //halt
        halt = 1;
        valP = pc + 1;
        rA = 4'd15;
        rB = 4'd15;
    end
    else if (icode == 4'd1 || icode == 4'd9)
    begin //nop
        valP = pc + 1;
        rA = 4'd15;
        rB = 4'd15;
    end
    else if (icode == 4'd2 || icode == 4'd6)
    begin //rrmovq or cmovXX or OPq
        valP = pc + 2;
    end
    else if (icode == 4'd3)
    begin //irmovq
        valP = pc + 10;
        valC = {instr_mem[pc+9], instr_mem[pc+8], instr_mem[pc+7], instr_mem[pc+6], instr_mem[pc+5], instr_mem[pc+4], instr_mem[pc+3],
        rA = 4'd15;
    end
    else if (icode == 4'd4 || icode == 4'd5)
    begin //rmmovq or mrmovq
        valP = pc + 10;
        valC = {instr_mem[pc+9], instr_mem[pc+8], instr_mem[pc+7], instr_mem[pc+6], instr_mem[pc+5], instr_mem[pc+4], instr_mem[pc+3],
    end
    else if (icode == 4'd7)
    begin //jxx
        valP = pc + 9;
        valC = {instr_mem[pc+8], instr_mem[pc+7], instr_mem[pc+6], instr_mem[pc+5], instr_mem[pc+4], instr_mem[pc+3], instr_mem[pc+2],
        rA = 4'd15;
        rB = 4'd15;
    end
    else if (icode == 4'd8)
    begin
        valC = {instr_mem[pc+8], instr_mem[pc+7], instr_mem[pc+6], instr_mem[pc+5], instr_mem[pc+4], instr_mem[pc+3], instr_mem[pc+2],
        valP = pc + 9;
    end
    else if (icode == 4'd10 || icode == 4'd11)
    begin
        valP = pc + 2;
        rB = 4'd15;
    end
    else
        instr_valid = 1'b0;

end

endmodule
```

DECODE

- The DECODE cycle takes the input arguments $icode$, $ifun$, rA, rB, $valC$ and $valP$ from the output fo the FETCH cycle.
- The register array was given as a 15×64 dimension (15 for the number of registers and 8 bytes for each word).
- The values $valA$ and $valB$ were taken from this register array, where the addresses for the required array element were given with rA, rB or rsp (for stack pointer).
- These values ($valA$ and $valB$) were the outputs of the cycle

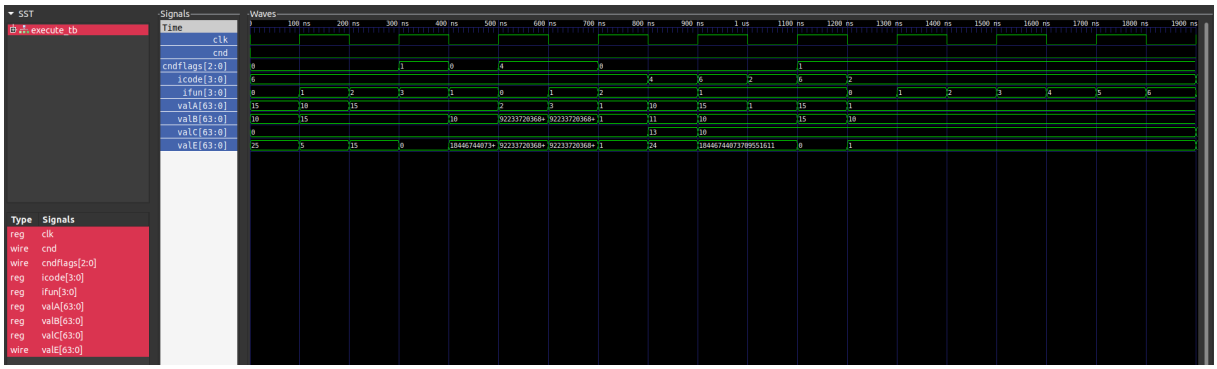



```
module decode (
    clk, valA, valB, icode, ifun, rA, rB, instr_valid,
    regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6,
    regfile7, regfile8, regfile9, regfile10,regfile11,regfile12,regfile13,
    regfile14
);

output reg [63:0] valA;
output reg [63:0] valB;
input [3:0] icode, ifun, rA, rB;
input clk, instr_valid;
input [63:0] regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6, regfile7, regfile8, regfile9, regfile10, regfile11,

reg [63:0] regmem [0:14];

parameter rsp = 4'd4;

always @(*)
begin
    regmem[0] = regfile0;
    regmem[1] = regfile1;
    regmem[2] = regfile2;
    regmem[3] = regfile3;
    regmem[4] = regfile4;
    regmem[5] = regfile5;
    regmem[6] = regfile6;
    regmem[7] = regfile7;
    regmem[8] = regfile8;
    regmem[9] = regfile9;
    regmem[10] = regfile10;
    regmem[11] = regfile11;
    regmem[12] = regfile12;
    regmem[13] = regfile13;
    regmem[14] = regfile14;
    if (icode == 4'd0 || icode == 4'd1)
    begin
        valA = 64'd15;
        valB = 64'd15;
    end
    if (icode == 4'd2)
        valA = regmem[rA];
    if (icode == 4'd4) //rmmovl
    begin
        valA = regmem[rA];
        valB = regmem[rB];
    end
    if (icode == 4'd5) //mrmovl
    begin
        valB = regmem[rB];
    end
```

```
    if (icode == 4'd6)
    begin
        valA = regmem[rA];
        valB = regmem[rB];
    end
    if (icode == 4'd8) //call
        valB = regmem[rsp];
    if (icode == 4'd9) //ret
    begin
        valA = regmem[rsp];
        valB = regmem[rsp];
    end
    if (icode == 4'd10) //pushl
    begin
        valA = regmem[rA];
        valB = regmem[rsp];
    end
    if (icode == 4'd11) //popl
    begin
        valA = regmem[rsp];
        valB = regmem[rsp];
    end
end
endmodule

// halt 00
// nop 10
// rrmovl rA, rB 20rArB
// irmovl V, rB 30FrB V
// rmmovl rA, D(rB) 40rArB D
// mrmovl D(rA), rB 50rArB D
// OP1 rA, rB 6fnrArB
// jXX Dest 7f D
// cmovxx rA, rB 2fnrArB
// call Dest 80 D
// ret 90
// pushl rA A0rAF
// popl rA B0rAF
```

EXECUTE

- The EXECUTE cycle takes $valA,\ valB$ and $valP$ as inputs, and with $icode$ and $ifun$, the output $valE$ is computed.
- The ALU is called as required -
  - if $icode = 0 \text{ or } 1$, do nothing
  - if $icode = 2$, according to the value of $ifun$ the condition codes are looked at and if satisfied, $valA$ is added to 0 (select = 0) and stored in $valE$
  - if $icode = 3$, 0 is added (select = 0) with $valC$ and stored in $valE$
  - if $icode = 4 \text{ or } 5$, $valB$ is added to $valC$ (select = 0) and stored in $valE$
  - if $icode = 6$, the ALU is called according to the value of $ifun$ and $valE$ is given the value $valB \text{ OP } valA$
  - if $icode = 7$, the condition codes are checked and the cnd flag (for the jump instruction) is set
  - if $icode = 8 \text{ or } 10$, select is set to 1 and 8 is subtracted from $valB$ and stored in $valE$
  - if $icode = 9 \text{ or } 11$, select is set to 1 and 8 is added to $valB$ and stored in $valE$




```
////////////////////////////////////////////////
// t = a OP b                                 //
// of = (a < 0 == b < 0) && (t < 0 != a < 0) //
// sf = (t < 0)                               //
// zf = (t == 0)                              //
////////////////////////////////////////////////

`include "ALU.v"
module execute (
    clk, icode, ifun, valC, valA, valB, cndflags, valE, cnd, instr_valid
);

input clk, instr_valid;
input [3:0] icode, ifun;
input [63:0] valA, valB, valC;
output reg [63:0] valE;
output reg [2:0] cndflags; // [zf, sf, of]
output reg cnd;

reg [63:0] aa, bb;
reg [1:0] select;
wire overflow;
wire [63:0] outalu;

ALU alu1(aa, bb, outalu, select, overflow);

initial
begin
    select[0] = ifun[0];
    select[1] = ifun[1];
    aa = valB;
    bb = valA;
    cndflags = 3'b0;
    cnd = 1'b0;
end

always @(*)
begin
    if (icode == 4'd6) //OPq
    begin
        cndflags = 3'b0;
        select[0] = ifun[0];
        select[1] = ifun[1];
        aa = valB;
        bb = valA;
        valE = outalu;
        if (ifun == 4'd0) //addq
        begin
            if (overflow)
                cndflags = 4;
        end
        else if (ifun == 4'd1) //subq
        begin
            if (overflow)
                cndflags = 4;
        end
        if (outalu == 0)
            cndflags = 1;
        else if (outalu < 0 && overflow != 1)
            cndflags = 2;
    end
```

```
    else if (icode == 4'd4 || icode == 4'd5) //rmmovq mrmovq
    begin
        select = 0;
        aa = valB;
        bb = valC;
        valE = outalu;
    end
    else if (icode == 4'd11 || icode == 4'd9) //popq or ret
    begin
        select = 0;
        aa = valB;
        bb = 64'd8;
        valE = outalu;
    end
    else if (icode == 4'd10 || icode == 4'd8) //pushq or call
    begin
        select = 1;
        aa = valB;
        bb = 64'd8;
        valE = outalu;
    end
    else if (icode == 4'd2) //cmovxx
    begin
        select = 0;
        if (ifun == 4'd0) //rrmovq
        begin
            aa = 64'd0;
            bb = valA;
            valE = outalu;
        end
        if (ifun == 4'd1) //cmovle
        begin
            if (cndflags[0] || cndflags[1] ^ cndflags[2])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd2) //cmovl
        begin
            if (cndflags[1] ^ cndflags[2])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd3) //cmove
        begin
            if (cndflags[0])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd4) //cmovne
        begin
            if (cndflags[0] == 0)
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd5) //cmovge
        begin
            if (cndflags[1] == cndflags[2])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd6) //cmovg
        begin
            if (cndflags[0] == 0 && ((cndflags[1] ^ cndflags[2]) == 1))
            begin
                aa = 64'd0;
                bb = valA;
```

```
                valE = outalu;
            end
        end
    end
    else if (icode == 4'd3) //irmovq
    begin
        select = 0;
        aa = 64'd0;
        bb = valC;
        valE = outalu;
    end
    else if (icode == 4'd7) //jXX
    begin
        if (ifun == 4'd0) //jmp
            cnd = 1'b1;
        if (ifun == 4'd1) //jle
            if (cndflags[0] || cndflags[1] ^ cndflags[2])
                cnd = 1'b1;
        if (ifun == 4'd2) //jl
            if (cndflags[1] ^ cndflags[2])
                cnd = 1'b1;
        if (ifun == 4'd3) //je
            if (cndflags[0])
                cnd = 1'b1;
        if (ifun == 4'd4) //jne
            if (cndflags[0] == 0)
                cnd = 1'b1;
        if (ifun == 4'd5) //jge
            if (cndflags[1] == cndflags[2])
                cnd = 1'b1;
        if (ifun == 4'd6) //jg
            if (cndflags[0] == 0 && ((cndflags[1] ^ cndflags[2]) == 1))
                cnd = 1'b1;
    end
end

endmodule
```

MEMORY

- The MEM module takes $valE$ and $valA$ from the EXECUTE module and refers to the main memory for its values. It is also positive edge clocked
- For $icode = 4,\ 8$ and 10, $valA$ is assigned to the main memory address referenced by $valE$ (rmmov, call, push)
- For $icode = 5$ and 11 (mrmov and pop), the value from memory at $valE$ address is written to $valM$
- For the ret instruction, the value from memory address $valA$ is written to $valM$
- Also, the $memdata$ output (8 bytes) is updated with the memory value at address $valE$ at each positive edge of the clock.
- The data memory (main memory) was initialised to a value (7) chosen randomly to avoid don't care states.


```
module mem (clk,
    icode, valE, valA, valM, memdata
);
input clk;
input [3:0] icode;
```

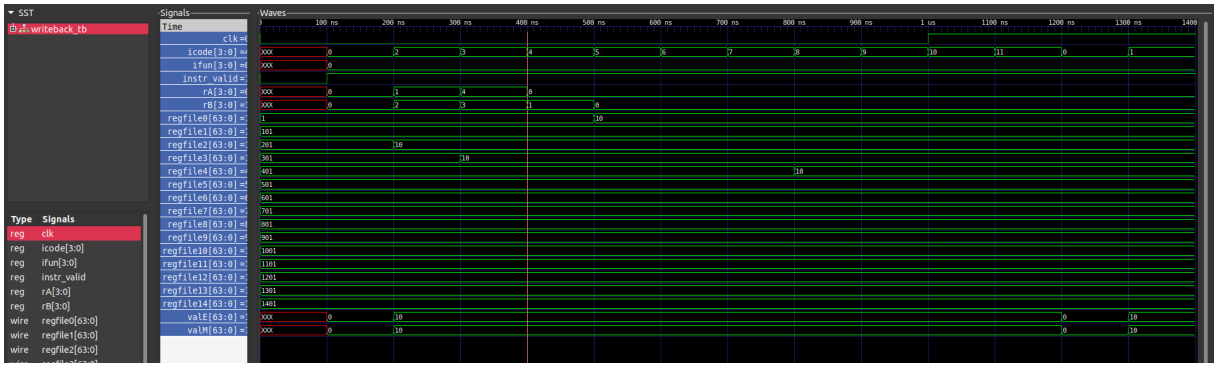
```
input  [63:0] valE, valA;
output reg  [63:0] valM, memdata;

reg [63:0] data[0:1023];

integer i;

initial begin
    for (i = 0; i < 1024; i++)
    begin
         data[i] = 64'b111; //taken randomly
    end
end

always @(posedge clk)
begin
    if (icode == 4'd4)
        data[valE] = valA;
    if (icode == 4'd5)
        valM = data[valE];
    if (icode == 4'd8)
        data[valE] = valA;
    if (icode == 4'd9)
        valM = data[valA];
    if (icode == 4'd10)
        data[valE] = valA;
    if (icode == 4'd11)
        valM = data[valE];

    memdata = data[valE];
end

endmodule
```

WRITEBACK

In Y86 processors, the WRITEBACK module is responsible for writing the final value of a register (if applicable) and setting the condition flags.

The WRITEBACK module is typically the final stage in the Y86 pipeline, and its output is used to update the state of the processor.

- The same register array was used as in the decode stage and this array was written to in the negative edge of the clock.
- The values from this array were written to register outputs as well (15 outputs, each of 1 byte size)
- According to the value of $icode$, either $valE$ or $valM$ is written to the appropriate registers, addressed by rA , rB or rsp
- The writing back happens at the negative edge of the clock so that there is not one more entire clock wasted.




```
module writeback (
    clk, rA, rB, valM, valE, icode, ifun,
    regfile0, regfile1, regfile2, regfile3, regfile4,
    regfile5, regfile6, regfile7, regfile8, regfile9,
    regfile10,regfile11,regfile12,regfile13,regfile14
);

input clk;
```

```
// input instr_valid;
input [63:0] valE, valM;
input [3:0] icode, ifun, rA, rB;
output reg [63:0] regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6, regfile7, regfile8, regfile9, regfile10, regfil

reg [63:0] regmem [0:14];

parameter rsp = 4'd4;

integer i;

initial
begin
    for (i = 0; i < 15; i++)
    begin
        regmem[i] = 100*i + 1;
    end
end
always @(*)
begin
    regfile0 = regmem[0];
    regfile1 = regmem[1];
    regfile2 = regmem[2];
    regfile3 = regmem[3];
    regfile4 = regmem[4];
    regfile5 = regmem[5];
    regfile6 = regmem[6];
    regfile7 = regmem[7];
    regfile8 = regmem[8];
    regfile9 = regmem[9];
    regfile10 = regmem[10];
    regfile11 = regmem[11];
    regfile12 = regmem[12];
    regfile13 = regmem[13];
    regfile14 = regmem[14];
end
always @(negedge clk)
begin
    if (1)
    begin
        if (icode == 4'd0 || icode == 4'd1)
        begin //halt or nop

        end
        if (icode == 4'd2) //cmovxx
            regmem[rB] = valE;
        if (icode == 4'd3) //irmovq
            regmem[rB] = valE;
        if (icode == 4'd5) //mrmovq
            regmem[rA] = valM;
        if (icode == 4'd6) //Opq
            regmem[rB] = valE;
        if (icode == 4'd8) //call
            regmem[rsp] = valE;
        if (icode == 4'd9) //ret
            regmem[rsp] = valE;
        if (icode == 4'd10) //pushq
            regmem[rsp] = valE;
        if (icode == 4'd11) //popq
        begin
            regmem[rsp] = valE;
            regmem[rA] = valM;
        end
    end
end

endmodule
```

PIPELINED

In Y86 processors, pipelining is a technique used to increase the throughput of the processor. It allows multiple instructions to be executed simultaneously, by breaking instructions down into smaller stages that can be executed in parallel.

Each stage operates on a different instruction, allowing multiple instructions to be executed simultaneously. As a result, pipelining can significantly improve the performance of the processor, allowing it to execute more instructions in the same amount of time.

However, pipelining introduces additional complexity and can also introduce hazards, such as data dependencies and control hazards, which must be carefully managed to avoid incorrect results.

There are three main types of pipeline hazards in Y86 processors:

1. Structural hazards **-** These occur when two instructions require the same hardware resources at the same time. For example, if two instructions both require access to the memory at the same time, a structural hazard will occur.
2. Data hazards **-** These occur when an instruction depends on data that is not yet available because it is still being processed by a previous instruction. For example, if an instruction tries to read a register value that is still being updated by a previous instruction, a data hazard will occur.
3. Control hazards - These occur when the pipeline makes a decision based on the outcome of an instruction that has not yet completed. For example, if a branch instruction changes the program counter, but the next instruction has already been fetched, a control hazard will occur.

To avoid these hazards, various techniques are used, such as forwarding, stalling, and branch prediction. These techniques allow the processor to maintain correct operation while still achieving the benefits of pipelining.

F_reg and Fetch

```
//f_reg
module f_reg (
    clk, valP,valC,icode, F_predPC,predPC
);

input clk;
input [3:0] icode;
input [63:0] valC;
input [63:0] valP;
input [63:0] predPC;
output reg [63:0] F_predPC;

always @(posedge clk)
begin
    F_predPC = predPC;
end

endmodule

//fetch
module fetch (clk, f_pc, icode, ifun, rA, rB, valC, valP, halt, instr_valid, imem_error, predPC);

input clk;
input [63:0] f_pc;
output reg [3:0] icode;
output reg [3:0] ifun;
output reg [3:0] rA;
output reg [3:0] rB;
output reg [63:0] valC;
output reg [63:0] valP;
output reg [63:0] predPC;
output reg halt;
output reg instr_valid;
output reg imem_error;
reg [0:79] instr;
reg [0:7] instr_mem [0:119];  // little endian

// always @(valP or valC or icode)
// begin
//     if (icode == 4'd7 || icode == 4'd8)
//         predPC = valC;
//     else
//         predPC = valP;
// end

always @(*)
begin
    instr_valid = 1'b1;
    imem_error = 1'b0;
    halt = 1'b0;

    if(f_pc > 119)
    begin
```

```
        imem_error = 1'b1;
    end


    instr_mem[0]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[1]  = 8'b01100000; // Opq add
    instr_mem[2]  = 8'b00000001; // rA = 0, rB = 1; f_pc = f_pc + 2 = 3
    instr_mem[3]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[4]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[5]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1

    instr_mem[6]  = 8'b00110000; // irmovq instruction f_pc = f_pc + 10 = 13
    instr_mem[7]  = 8'b11110010; // F, rB = 2;
    instr_mem[8]  = 8'b11111111; // 1st byte of V = 255, rest all bytes will be zero
    instr_mem[9]  = 8'b00000000; // 2nd byte
    instr_mem[10]  = 8'b00000000; // 3rd byte
    instr_mem[11]  = 8'b00000000; // 4th byte
    instr_mem[12]  = 8'b00000000; // 5th byte
    instr_mem[13] = 8'b00000000; // 6th byte
    instr_mem[14] = 8'b00000000; // 7th byte
    instr_mem[15] = 8'b00000000; // 8th byte (This completes irmovq)
    instr_mem[16]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[17]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[18]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[19] = 8'b00110000; // irmovq instruction f_pc = f_pc + 10 = 23
    instr_mem[20] = 8'b11110011; // F, rB = 3;
    instr_mem[21] = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    instr_mem[22] = 8'b00000000; // 2nd byte
    instr_mem[23] = 8'b00000000; // 3rd byte
    instr_mem[24] = 8'b00000000; // 4th byte
    instr_mem[25] = 8'b00000000; // 5th byte
    instr_mem[26] = 8'b00000000; // 6th byte
    instr_mem[27] = 8'b00000000; // 7th byte
    instr_mem[28] = 8'b00000000; // 8th byte (This completes irmovq)
    instr_mem[29]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[30]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[31]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[32] = 8'b00110000; // irmovq instruction f_pc = f_pc + 10 = 33
    instr_mem[33] = 8'b11110100; // F, rB = 4;
    instr_mem[34] = 8'b00000101; // 1st byte of V = 5, rest all bytes will be zero
    instr_mem[35] = 8'b00000000; // 2nd byte
    instr_mem[36] = 8'b00000000; // 3rd byte
    instr_mem[37] = 8'b00000000; // 4th byte
    instr_mem[38] = 8'b00000000; // 5th byte
    instr_mem[39] = 8'b00000000; // 6th byte
    instr_mem[40] = 8'b00000000; // 7th byte
    instr_mem[41] = 8'b00000000; // 8th byte (This completes irmovq)
    instr_mem[42]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[43]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[44]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[45] = 8'b00100000; // rrmovq // f_pc = f_pc + 2 = 35
    instr_mem[46] = 8'b01000101; // rA = 4; rB = 5;
    instr_mem[47]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[48]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[49]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[50]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[51]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[52]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[53]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[54]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[55]  = 8'b00010000; // nop instruction f_pc = f_pc +1 = 1
    instr_mem[56] = 8'b01100000; // Opq add // f_pc = f_pc + 2 = 37
    instr_mem[57] = 8'b00110100; // rA = 3 and rB = 4, final value in rB(4) = 10;

    instr_mem[58] = 8'b00100101; // cmovge // f_pc = f_pc + 2 = 39
    instr_mem[59] = 8'b01010110; // rA = 5; rB = 6;

    instr_mem[60] = 8'b01100001; // Opq subq // f_pc = f_pc + 2 = 41
    instr_mem[61] = 8'b00110101; // rA = 3, rB = 5; both are equal
    instr_mem[62] = 8'b00010000; // nop
    instr_mem[63] = 8'b00010000; // nop
    instr_mem[64] = 8'b00010000; // nop
    instr_mem[65] = 8'b00010000; // nop
    instr_mem[66] = 8'b01110011; //je // f_pc = f_pc + 9 = 50   **0011
    instr_mem[67] = 8'd53; // Dest = 52; 1st byte
    instr_mem[68] = 8'b00000000; // 2nd byte
    instr_mem[69] = 8'b00000000; // 3rd byte
    instr_mem[70] = 8'b00000000; // 4th byte
    instr_mem[71] = 8'b00000000; // 5th byte
    instr_mem[72] = 8'b00000000; // 6th byte
    instr_mem[73] = 8'b00000000; // 7th byte
```

```
    instr_mem[74] = 8'b00000000; // 8th byte

    instr_mem[75] = 8'b00010000; // nop
    instr_mem[76] = 8'b00010000; // nop
    instr_mem[77] = 8'b00010000; // nop
    instr_mem[78] = 8'b00010000; // nop
    instr_mem[79] = 8'b00010000; // nop
    instr_mem[80] = 8'b00010000; // nop

    instr_mem[81] = 8'b01100000; // Opq add
    instr_mem[82] = 8'b00110101; // rA = 3; rB = 5;

    instr_mem[83] = 8'b10000000; //8 0 //call
    instr_mem[84] = 8'b00000000; //Dest
    instr_mem[85] = 8'b00000000; //Dest
    instr_mem[86] = 8'b00000000; //Dest
    instr_mem[87] = 8'b00000000; //Dest
    instr_mem[88] = 8'b00000000; //Dest
    instr_mem[89] = 8'b00000000; //Dest
    instr_mem[90] = 8'b00000000; //Dest
    instr_mem[91] = 8'b00000000; //Dest

    // instr_mem[63] = 8'b10010000; // 9 0 //ret
    // instr_mem[81] = 8'b00000000; // halt
    instr_mem[92] = 8'b00000000; // halt



    instr = {instr_mem[f_pc], instr_mem[f_pc+1], instr_mem[f_pc+2], instr_mem[f_pc+3], instr_mem[f_pc+4], instr_mem[f_pc+5], instr_mem[f_pc

    icode = instr[0:3];
    ifun = instr[4:7];

    rA = instr[8:11];
    rB = instr[12:15];

    if (icode == 4'd0)
    begin //halt
        halt = 1;
        valP = f_pc + 1;
        rA = 4'd15;
        rB = 4'd15;
    end
    else if (icode == 4'd1 || icode == 4'd9)
    begin //nop
        valP = f_pc + 1;
        rA = 4'd15;
        rB = 4'd15;
    end
    else if (icode == 4'd2 || icode == 4'd6)
    begin //rrmovq or cmovXX or OPq
        valP = f_pc + 2;
    end
    else if (icode == 4'd3)
    begin //irmovq
        valP = f_pc + 10;
        valC = {instr_mem[f_pc+9], instr_mem[f_pc+8], instr_mem[f_pc+7], instr_mem[f_pc+6], instr_mem[f_pc+5], instr_mem[f_pc+4], instr_mem
        rA = 4'd15;
    end
    else if (icode == 4'd4 || icode == 4'd5)
    begin //rmmovq or mrmovq
        valP = f_pc + 10;
        valC = {instr_mem[f_pc+9], instr_mem[f_pc+8], instr_mem[f_pc+7], instr_mem[f_pc+6], instr_mem[f_pc+5], instr_mem[f_pc+4], instr_mem
    end
    else if (icode == 4'd7)
    begin //jxx
        valP = f_pc + 9;
        // valP = {instr_mem[f_pc+8], instr_mem[f_pc+7], instr_mem[f_pc+6], instr_mem[f_pc+5], instr_mem[f_pc+4], instr_mem[f_pc+3], instr_
        valC = {instr_mem[f_pc+8], instr_mem[f_pc+7], instr_mem[f_pc+6], instr_mem[f_pc+5], instr_mem[f_pc+4], instr_mem[f_pc+3], instr_mem
        rA = 4'd15;
        rB = 4'd15;
    end
    else if (icode == 4'd8)
    begin
        valC = {instr_mem[f_pc+8], instr_mem[f_pc+7], instr_mem[f_pc+6], instr_mem[f_pc+5], instr_mem[f_pc+4], instr_mem[f_pc+3], instr_mem
        valP = f_pc + 9;
    end
    else if (icode == 4'd10 || icode == 4'd11)
    begin
        valP = f_pc + 2;
```

```
        rB = 4'd15;
    end
    else
        instr_valid = 1'b0;

// predict pc
    if (instr[0:3] == 4'd7 || instr[0:3] == 4'd8)
        predPC = valC;
    else
        predPC = valP;
end


endmodule
```

- The changes made in the fetch module are quite evident -
  - The $always$ $@()$ block uses a change in any input instead of being posedge triggered. This has been done because for the posedge case, the output values (e.g. $icode,\ ifun,\ etc$) will only be available at the positive edge of the clock, even if they have been computed before that point.

    This leads to a synchronisation error in the values of FETCH and DECODE, as both of them have their values available at the same positive edge, which makes the pipe lose its “pipe” functionality.

    The same has been done for all stages (excluding register stages).
  - For the case of jumps and call, the value for the predicted PC has been taken as $valC$, despite the state of the condition codes.

    This is to account for the jump prediction and call destination prediction.

    For all other cases, the value for the predicted PC is just $valP$
  - Additionally, the instruction set was modified to include extra $nops$ so that the registers can be written to in time for the next instruction to be fetched.

    This has been done to cancel out any data dependencies and make the pipe work as required.

D_reg and DECODE

```
// D_reg
module d_reg (
    clk, icode, ifun, rA, rB, valC, valP,
    D_icode, D_ifun, D_rA, D_rB, D_valC, D_valP
);

input clk;
input [3:0] icode;
input [3:0] ifun;
input [3:0] rA;
input [3:0] rB;
input [63:0] valC;
input [63:0] valP;
output reg [3:0] D_icode;
output reg [3:0] D_ifun;
output reg [3:0] D_rA;
output reg [3:0] D_rB;
output reg [63:0] D_valC;
output reg [63:0] D_valP;

always @(posedge clk)
begin
    D_icode <= icode;
    D_ifun <= ifun;
    D_rA <= rA;
    D_rB <= rB;
    D_valC <= valC;
    D_valP <= valP;
end
endmodule

// DECODE
module decode (
    clk, valA, valB, icode, ifun, rA, rB, instr_valid, dstE, dstM,
```

```
    regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6,
    regfile7, regfile8, regfile9, regfile10,regfile11,regfile12,regfile13,
    regfile14
);

output reg [63:0] valA;
output reg [63:0] valB;
output reg [3:0] dstE,dstM,srcA,srcB;
input [3:0] icode, ifun, rA, rB;
input clk, instr_valid;
input [63:0] regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6, regfile7, regfile8, regfile9, regfile10, regfile11, regf
reg [63:0] regmem [0:14];

parameter rsp = 4'd4;

always @(*)
begin
    regmem[0] = regfile0;
    regmem[1] = regfile1;
    regmem[2] = regfile2;
    regmem[3] = regfile3;
    regmem[4] = regfile4;
    regmem[5] = regfile5;
    regmem[6] = regfile6;
    regmem[7] = regfile7;
    regmem[8] = regfile8;
    regmem[9] = regfile9;
    regmem[10] = regfile10;
    regmem[11] = regfile11;
    regmem[12] = regfile12;
    regmem[13] = regfile13;
    regmem[14] = regfile14;

// destM
    if(icode == 4'b0101 || icode == 4'b1011)
        dstM = rA;
    else
        dstM = 4'b1111;

// dstE
    if(icode == 4'b0010 || icode == 4'b0011 || icode ==  4'b0110)
        dstE = rB;
    else if(icode == 4'b1000 || icode == 4'b1001 || icode == 4'b1010 || icode == 4'b1011)
        dstE = 4'b0100;
    else
        dstE = 4'b1111;

    if (icode == 4'd0 || icode == 4'd1)
    begin
        valA = 64'd15;
        valB = 64'd15;
    end
    if (icode == 4'd2)
        valA = regmem[rA];
    if (icode == 4'd4) //rmmovl
    begin
        valA = regmem[rA];
        valB = regmem[rB];
    end
    if (icode == 4'd5) //mrmovl
    begin
        valB = regmem[rB];
    end
    if (icode == 4'd6)
    begin
        valA = regmem[rA];
        valB = regmem[rB];
    end
    if (icode == 4'd8) //call
        valB = regmem[rsp];
    if (icode == 4'd9) //ret
    begin
        valA = regmem[rsp];
        valB = regmem[rsp];
    end
    if (icode == 4'd10) //pushl
    begin
        valA = regmem[rA];
        valB = regmem[rsp];
    end
    if (icode == 4'd11) //popl
```

```
    begin
        valA = regmem[rsp];
        valB = regmem[rsp];
    end
end
endmodule

// halt 00
// nop 10
// rrmovl rA, rB 20rArB
// irmovl V, rB 30FrB V
// rmmovl rA, D(rB) 40rArB D
// mrmovl D(rA), rB 50rArB D
// OP1 rA, rB 6fnrArB
// jXX Dest 7f D
// cmovxx rA, rB 2fnrArB
// call Dest 80 D
// ret 90
// pushl rA A0rAF
// popl rA B0rAF
```

- In the DECODE module the only changes made were the addition of cases for $dstE$ and $dstM$, which are required by the subsequent modules for pipe-specific operations

  $dstE$ and $dstM$ are used to keep track of the destination registers for the execution and memory stages, respectively.

  In the execute stage, $dstE$ is used to determine which register is being written to by the current instruction, while in the memory stage, $dstM$ is used to determine which register is being written to by memory operations like $mrmovl$ and $rmmovl$.

  These values are computed by the decode stage and passed on to subsequent stages in the pipeline.

- The D_REG module is just to have non blocking positive edge triggered assignments from the previous module's values to the next modules required values.
- These are clocked at the positive edge for conformity and proper timing of the processor.

E_reg and EXECUTE

```
// E_reg
module e_reg (
    clk, icode, ifun, valA, valB, valC, srcA, srcB, dstE, dstM,
    E_icode, E_ifun, E_valA, E_valB, E_valC, E_srcA, E_srcB, E_dstE, E_dstM
);

input clk;
input [3:0] icode;
input [3:0] ifun;
input [3:0] srcB;
input [3:0] srcA;
input [3:0] dstM;
input [3:0] dstE;
input [63:0] valA;
input [63:0] valB;
input [63:0] valC;
output reg [3:0] E_icode;
output reg [3:0] E_ifun;
output reg [3:0] E_srcB;
output reg [3:0] E_srcA;
output reg [3:0] E_dstM;
output reg [3:0] E_dstE;
output reg [63:0] E_valA;
output reg [63:0] E_valB;
output reg [63:0] E_valC;

always @(posedge clk)
begin
    E_icode <= icode;
    E_ifun <= ifun;
    E_srcB <= srcB;
    E_srcA <= srcA;
    E_dstM <= dstM;
    E_dstE <= dstE;
    E_valA <= valA;
    E_valB <= valB;
    E_valC <= valC;
```

```
end

endmodule

// EXECUTE
////////////////////////////////////////////////
// t = a OP b                                 //
// of = (a < 0 == b < 0) && (t < 0 != a < 0) //
// sf = (t < 0)                               //
// zf = (t == 0)                              //
////////////////////////////////////////////////

`include "ALU.v"
module execute (
    clk, icode, ifun, valC, valA, valB, cndflags, valE, cnd, instr_valid
);

input clk, instr_valid;
input [3:0] icode, ifun;
input [63:0] valA, valB, valC;
output reg [63:0] valE;
output reg [2:0] cndflags; // [zf, sf, of]
output reg cnd;

reg [63:0] aa, bb;
reg [1:0] select;
wire overflow;
wire [63:0] outalu;

ALU alu1(aa, bb, outalu, select, overflow);

initial
begin
    select[0] = ifun[0];
    select[1] = ifun[1];
    aa = valB;
    bb = valA;
    cndflags = 3'b0;
    cnd = 1'b0;
end

always @(*)
begin
    if (icode == 4'd6) //OPq
    begin
        cndflags = 3'b0;
        select[0] = ifun[0];
        select[1] = ifun[1];
        aa = valB;
        bb = valA;
        valE = outalu;
        if (ifun == 4'd0) //addq
        begin
            if (overflow)
                cndflags = 4;
        end
        else if (ifun == 4'd1) //subq
        begin
            if (overflow)
                cndflags = 4;
        end
        if (outalu == 0)
            cndflags = 1;
        else if (outalu < 0 && overflow != 1)
            cndflags = 2;
    end
    else if (icode == 4'd4 || icode == 4'd5) //rmmovq mrmovq
    begin
        select = 0;
        aa = valB;
        bb = valC;
        valE = outalu;
    end
    else if (icode == 4'd11 || icode == 4'd9) //popq or ret
    begin
        select = 0;
        aa = valB;
        bb = 64'd8;
        valE = outalu;
    end
    else if (icode == 4'd10 || icode == 4'd8) //pushq or call
```

```
    begin
        select = 1;
        aa = valB;
        bb = 64'd8;
        valE = outalu;
    end
    else if (icode == 4'd2) //cmovxx
    begin
        select = 0;
        if (ifun == 4'd0) //rrmovq
        begin
            aa = 64'd0;
            bb = valA;
            valE = outalu;
        end
        if (ifun == 4'd1) //cmovle
        begin
            if (cndflags[0] || cndflags[1] ^ cndflags[2])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd2) //cmovl
        begin
            if (cndflags[1] ^ cndflags[2])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd3) //cmove
        begin
            if (cndflags[0])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd4) //cmovne
        begin
            if (cndflags[0] == 0)
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd5) //cmovge
        begin
            if (cndflags[1] == cndflags[2])
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
        if (ifun == 4'd6) //cmovg
        begin
            if (cndflags[0] == 0 && ((cndflags[1] ^ cndflags[2]) == 1))
            begin
                aa = 64'd0;
                bb = valA;
                valE = outalu;
            end
        end
    end
    else if (icode == 4'd3) //irmovq
    begin
        select = 0;
        aa = 64'd0;
        bb = valC;
        valE = outalu;
    end
    else if (icode == 4'd7) //jXX
    begin
    cnd = 1'b0;
        if (ifun == 4'd0) //jmp
```

```
            cnd = 1'b1;
        if (ifun == 4'd1) //jle
            if (cndflags[0] || cndflags[1] ^ cndflags[2])
                cnd = 1'b1;
        if (ifun == 4'd2) //jl
            if (cndflags[1] ^ cndflags[2])
                cnd = 1'b1;
        if (ifun == 4'd3) //je
            if (cndflags[0])
                cnd = 1'b1;
        if (ifun == 4'd4) //jne
            if (cndflags[0] == 0)
                cnd = 1'b1;
        if (ifun == 4'd5) //jge
            if (cndflags[1] == cndflags[2])
                cnd = 1'b1;
        if (ifun == 4'd6) //jg
            if (cndflags[0] == 0 && ((cndflags[1] ^ cndflags[2]) == 1))
                cnd = 1'b1;
    end
end

endmodule
```

- There were no changes in the EXECUTE module which retained the same functionality as the sequential version
- The e_reg module is used to store the output values of the execute stage of the pipeline. It contains registers for the instruction code, instruction function, source A and B, destination M and E, and values A, B, and C.
- These values are passed on to the memory stage of the pipeline. The e_reg module is used to ensure that the output values from the execute stage are not overwritten before they can be used in the memory stage.
- The values are clocked at the positive edge of the clock signal to ensure proper timing of the processor.

M_reg and MEMORY

```
// M_reg

module m_reg (
    clk, icode, valA, valE, dstE, dstM, cnd,
    M_icode, M_valA, M_valE, M_dstE, M_dstM, M_cnd
);

input clk;
input cnd;
input [3:0] icode;
input [3:0] dstE;
input [3:0] dstM;
input [63:0] valE;
input [63:0] valA;
output reg M_cnd;
output reg [3:0] M_icode;
output reg [3:0] M_dstE;
output reg [3:0] M_dstM;
output reg [63:0] M_valE;
output reg [63:0] M_valA;

always @(posedge clk)
begin
    M_cnd  <= cnd;
    M_icode <= icode;
    M_dstE <= dstE;
    M_dstM <= dstM;
    M_valE <= valE;
    M_valA <= valA;
end
endmodule

// MEM
module mem (clk,
    icode, valE, valA, valM, memdata
);
input clk;
input [3:0] icode;
input  [63:0] valE, valA;
```

```
output reg  [63:0] valM, memdata;

reg [63:0] data[0:1023];

integer i;

initial begin
    for (i = 0; i < 1024; i++)
    begin
         data[i] = 64'b111; //taken randomly
    end
end

always @(*)
begin
    if (icode == 4'd4)
        data[valE] = valA;
    if (icode == 4'd5)
        valM = data[valE];
    if (icode == 4'd8)
        data[valE] = valA;
    if (icode == 4'd9)
        valM = data[valA];
    if (icode == 4'd10)
        data[valE] = valA;
    if (icode == 4'd11)
        valM = data[valE];

    memdata = data[valE];
end

endmodule
```

- The MEM module has not been changed except for the $always\ @()$ block, which has been changed to $*$ for synchronisation and consistency's sake.
- In the sequential implementation, the MEM module was positive edge triggered.
- The m_reg module is responsible for holding intermediate values from the memory stage and passing them on to the next stage in the pipeline. It contains the following inputs: clk, $icode$, $valA$, $valE$, $dstE$, $dstM$, $and\ cnd$. It outputs the following signals: M_icode, M_valA, M_valE, M_dstE, M_dstM, and M_cnd.
- During each clock cycle, the module updates its output signals with the values it received as inputs during the previous clock cycle. This is done using non-blocking assignments in a $always\ @(posedge\ clk)$ block.

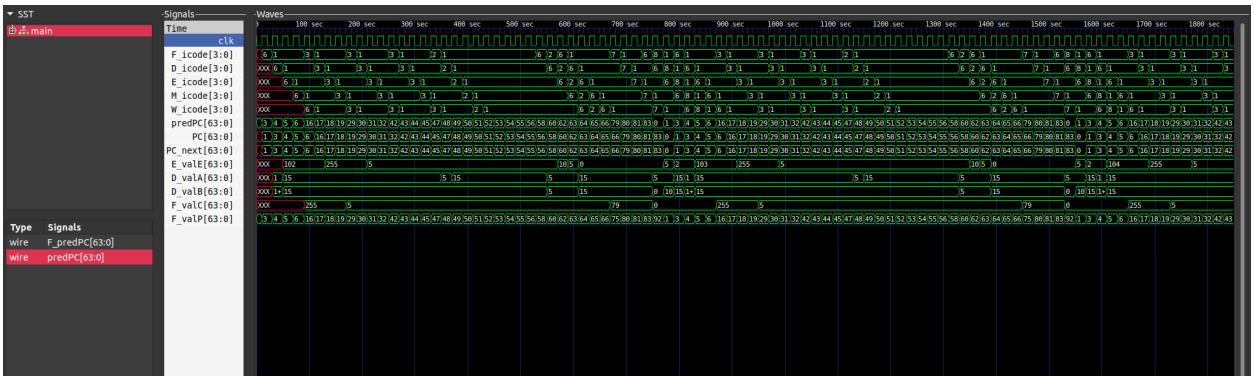
W_reg and WRITEBACK

```
// W_reg
module w_reg (clk, icode, valE, valM, dstE, dstM,W_icode, W_valE, W_valM, W_dstE, W_dstM);

input clk;
input [3:0] icode;
input [3:0] dstE;
input [3:0] dstM;
input [63:0] valE;
input [63:0] valM;
output reg [3:0] W_icode;
output reg [3:0] W_dstE;
output reg [3:0] W_dstM;
output reg [63:0] W_valE;
output reg [63:0] W_valM;

always @(posedge clk)
begin
    W_icode <= icode;
    W_dstE <= dstE;
    W_dstM <= dstM;
    W_valE <= valE;
    W_valM <= valM;
end

endmodule
// WRITEBACK
module writeback (
```

```
    clk, rA, rB, valM, valE, icode,
    regfile0, regfile1, regfile2, regfile3, regfile4,
    regfile5, regfile6, regfile7, regfile8, regfile9,
    regfile10,regfile11,regfile12,regfile13,regfile14
);

input clk;
// input instr_valid;
input [63:0] valE, valM;
input [3:0] icode, rA, rB;
output reg [63:0] regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6, regfile7, regfile8, regfile9, regfile10, regfile11,

reg [63:0] regmem [0:14];

parameter rsp = 4'd4;

integer i;

initial
begin
    for (i = 0; i < 15; i++)
    begin
        regmem[i] = 100*i + 1;
    end
end
always @(*)
begin
    regfile0 = regmem[0];
    regfile1 = regmem[1];
    regfile2 = regmem[2];
    regfile3 = regmem[3];
    regfile4 = regmem[4];
    regfile5 = regmem[5];
    regfile6 = regmem[6];
    regfile7 = regmem[7];
    regfile8 = regmem[8];
    regfile9 = regmem[9];
    regfile10 = regmem[10];
    regfile11 = regmem[11];
    regfile12 = regmem[12];
    regfile13 = regmem[13];
    regfile14 = regmem[14];
end
always @(negedge clk)
begin
    if (1)
    begin
        if (icode == 4'd0 || icode == 4'd1)
        begin //halt or nop

        end
        if (icode == 4'd2) //cmovxx
            regmem[rB] = valE;
        if (icode == 4'd3) //irmovq
            regmem[rB] = valE;
        if (icode == 4'd5) //mrmovq
            regmem[rA] = valM;
        if (icode == 4'd6) //Opq
            regmem[rB] = valE;
        if (icode == 4'd8) //call
            regmem[rsp] = valE;
        if (icode == 4'd9) //ret
            regmem[rsp] = valE;
        if (icode == 4'd10) //pushq
            regmem[rsp] = valE;
        if (icode == 4'd11) //popq
        begin
            regmem[rsp] = valE;
            regmem[rA] = valM;
        end
    end
end

endmodule
```

- The WRITEBACK module was not changed from the sequential implementation and same as before, the register files were written to according to the rA and rB values

- The w_reg module is used to hold intermediate values from the writeback stage and pass them on to the next stage in the pipeline. It contains the following inputs: clk, $icode$, $valE$, $valM$, $dstE$, $dstM$. It outputs the following signals: W_icode, W_valE, W_valM, W_dstE, $and\ W\_dstM$.
- During each clock cycle, the module updates its output signals with the values it received as inputs during the previous clock cycle. This is done using non-blocking assignments in an $always$ @$(posedge\ clk)$ block.
- The w_reg module is used to ensure that the output values from the writeback stage are not overwritten before they can be used in the next stage of the pipeline.



`This is a running version of the instructions in fetch stage which has a` $call$ instruction which has a destination of 0, creating an infinite loop.

It can be noticed that after $icode$ switches to 8 for the first time, PC goes back to 0 and the whole set of instructions computes again

Here, the values of the previous registers have been changed such that the branch is mispredicted.

We can see that even though the jump is taken, the branch misprediction correction works and the next fetched instructions are ignored and the halt is reached properly.

Another test was run with the same jump destination but with the previous values such that the jump should have been taken, and the code runs into an infinite loop, which is what we want, because the jump should have been taken anyway.

Therefore, it has been verified that all required instructions work.

Control Logic

We implemented the bubble logic, which says that if the stage needs bubbles (as mentioned in the book), s_bubble is set to one, which sets s_icode to 1, where s represents the stage (F, D, E, etc)

Bubbles are used in y86 to stall the pipeline and prevent instructions from getting ahead of one another. This is necessary in cases where an instruction needs to wait for a value to become available before it can proceed. By inserting a bubble into the pipeline, the

instruction is effectively stalled until the required value is available. This helps to ensure that instructions are executed in the correct order and that the correct results are produced.

It was tested with the same testbench as the other instructions and the functionality was verified.

Compilation instructions

```
$ iverilog -o <output-file> <input-file>.v <input-file-testbench>.v
$ vvp <output-file>
$ gtkwave <input-file-testbench>.vcd
```

- **example** -

  to compile the ALU script

  ```
  $ iverilog -o output ALU.v ALU_tb.v
  $ vvp output
  $ gtkwave ALU_tb.vcd
  ```

Processor Specifications

1. Clock Frequency

The minimum clock that we could possibly give by adjusting the time scale was $T_{clock} = 1ns$

Therefore, the Clock frequency for the processor is $\frac{1}{T_{clock}} = \frac{1}{10^{-9}} Hz = 1GHz$

2. Memory sizes

- The main memory was declared as a 64×1024 array, making the size of the main memory equal to $8kB$ (8 kilobytes)
- The instruction memory was declared as a 8×120 array, making the size of the instruction memory equal to $0.1kB$ (0.1 kilobyte)
- This implies that the memory was in the style of a harvard processor.

Problems Encountered

- In the sequential implementation, the main problems encountered were with clocking and correcting dont care states, which resulted from incomplete module declarations.

  As verilog does not automatically detect these missing identifier statements, they had to be manually found and accounted for.

  Clocking in terms of writing to the register files in the WRITEBACK module and reading from those files in the DECODE module was a problem.

  These were fixed rather easily by changing clock-edge triggers to any change in input triggers.

- In the Pipelined implementation, the main problems encountered were incomplete module declarations, data dependencies, and unwanted jumps.

  These dependencies were handled by inserting nops after the required instructions, wherever registers are being written to

  The incorrect jumps were handled by the required control logic change, wherein, if the execute cycle's computed condition codes show that the jump branch was mispredicted, it cancels the next 2 fetched instructions and goes to the next instruction which it should have.

  This is also exemplified in the above plots.

  Extensive testing was done for all other instructions and testbenches were implemented for each module. The waveforms will also be attached in the final folder

Testbenches

Fetch

```
`timescale 10ns/1ps

module fetch_tb;

initial
begin
    $dumpfile("fetch_tb.vcd");
    $dumpvars(0, fetch_tb);
end

reg clk;
reg [63:0] pc;
wire [3:0] icode;
wire [3:0] ifun;
wire [3:0] rA;
wire [3:0] rB;
wire [63:0] valC;
wire [63:0] valP;
wire halt;
wire instr_valid;
wire imem_error;

fetch f1
(
    .clk(clk),
    .pc(pc),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP),
    .halt(halt),
    .instr_valid(instr_valid),
    .imem_error(imem_error)
);

initial
    clk = 1;

parameter CLOCK_PER = 10;
always #CLOCK_PER
begin
    clk = ~clk;
end

initial
begin
    pc = 64'd0;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
    #10;
    pc = valP;
```

```
    #10;
    pc = valP;
    #10;
    $finish;
end

initial
begin
    $monitor("pc = %d, icode = %d, ifun = %d, valC = %d, valP = %d, rA = %d, rB = %d", pc, icode, ifun, valC, valP, rA, rB);
end
endmodule
```

Decode

```
`timescale 10ns/1ps
module decode_tb;

initial
begin
    $dumpfile("decode_tb.vcd");
    $dumpvars(0, decode_tb);
end

wire [63:0] valA;
wire [63:0] valB;
reg instr_valid;
reg [3:0] ifun;
reg [3:0] icode;
reg [3:0] rA;
reg [3:0] rB;
reg [3:0] rsp;
reg [63:0] regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6, regfile7, regfile8, regfile9, regfile10, regfile11, regfil
reg clk;

decode uut
(
    clk, valA, valB, icode, ifun, rA, rB, instr_valid,
    regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6,
    regfile7, regfile8, regfile9, regfile10,regfile11,regfile12,regfile13,
    regfile14
);

parameter CLOCK_PER = 10;
initial
begin
    clk = 0;
end
always #CLOCK_PER
begin
    clk = ~clk;
end
initial begin
    regfile0 = 64'd1;
    regfile1 = 64'd11;
    regfile2 = 64'd111;
    regfile3 = 64'd1111;
    regfile4 = 64'd11111;
    regfile5 = 64'd111111;
    regfile6 = 64'd1111111;
    regfile7 = 64'd11111111;
    regfile8 = 64'd111111;
    regfile9 = 64'd1111;
    regfile10 = 64'd111;
    regfile11 = 64'd11;
    regfile12 = 64'd1;
    regfile13 = 64'd1111;
    regfile14 = 64'd111111;
end
initial
begin

    instr_valid = 0;
    icode = 4'bz;  ifun = 4'bz; rA = 4'bz; rB = 4'bz; rsp = 4'bz;
    #10;
    instr_valid = 1'b1;
    icode = 4'd0; ifun = 4'd0; rA = 4'd1; rB = 4'd0; rsp = 4'd2;
    #10;
```

```
    icode = 4'd1; ifun = 4'd0; rA = 4'd2; rB = 4'd1; rsp = 4'd3;
    #10;
    icode = 4'd2; ifun = 4'd0; rA = 4'd3; rB = 4'd2; rsp = 4'd4;
    #10;
    icode = 4'd2; ifun = 4'd1; rA = 4'd4; rB = 4'd3; rsp = 4'd5;
    #10;
    icode = 4'd2; ifun = 4'd2; rA = 4'd5; rB = 4'd14; rsp = 4'd6;
    #10;
    icode = 4'd2; ifun = 4'd3; rA = 4'd6; rB = 4'd13; rsp = 4'd7;
    #10;
    icode = 4'd2; ifun = 4'd4; rA = 4'd7; rB = 4'd12; rsp = 4'd8;
    #10;
    icode = 4'd2; ifun = 4'd5; rA = 4'd8; rB = 4'd11; rsp = 4'd9;
    #10;
    icode = 4'd4; ifun = 4'd0; rA = 4'd9; rB = 4'd10; rsp = 4'd10;
    #10;
    icode = 4'd5; ifun = 4'd0; rA = 4'd10; rB = 4'd9; rsp = 4'd11;
    #10;
    icode = 4'd6; ifun = 4'd0; rA = 4'd11; rB = 4'd8; rsp = 4'd12;
    #10;
    icode = 4'd6; ifun = 4'd1; rA = 4'd12; rB = 4'd7; rsp = 4'd13;
    #10;
    icode = 4'd6; ifun = 4'd2; rA = 4'd13; rB = 4'd6; rsp = 4'd14;
    #10;
    icode = 4'd6; ifun = 4'd3; rA = 4'd14; rB = 4'd5; rsp = 4'd13;
    #10;
    icode = 4'd8; ifun = 4'd0; rA = 4'd3; rB = 4'd4; rsp = 4'd4;
    #10;
    icode = 4'd9; ifun = 4'd0; rA = 4'd0; rB = 4'd3; rsp = 4'd11;
    #10;
    icode = 4'd10; ifun = 4'd0; rA = 4'd14; rB = 4'd2; rsp = 4'd12;
    #10;
    icode = 4'd11; ifun = 4'd0; rA = 4'd14; rB = 4'd1; rsp = 4'd15;
    $finish;
end

initial begin
    $monitor("icode = %d, ifun = %d, R[rA] = %d, R[rB] = %d, R[rsp] = %d, valA = %d, valB = %d", icode, ifun, rA, rB, rsp, valA, valB);
end
endmodule
```

Execute

```
`timescale 10ns/1ps

module execute_tb;

initial begin
    $dumpfile("execute_tb.vcd");
    $dumpvars(0, execute_tb);
end

reg signed clk;
reg signed [3:0] icode;
reg signed [3:0] ifun;
reg signed [63:0] valA;
reg signed [63:0] valB;
reg signed [63:0] valC;              // offset
wire signed [63:0] valE;
wire cnd;
wire [2:0] cndflags;

execute e1(
    .clk(clk),
    .icode(icode),
    .ifun(ifun),
    .valA(valA),
    .valB(valB),
    .valC(valC),
    .valE(valE),
    .cnd(cnd),
    .cndflags(cndflags)
);

parameter CLOCK_PER = 10;
initial
begin
```

```
    clk = 0;
end
always #CLOCK_PER
begin
    clk = ~clk;
end


initial begin
    icode = 4'd6; ifun = 4'd0; valA = 64'd15; valB = 64'd10; valC = 64'd0; #10;
    ifun = 4'd1; valB = 64'd15; valA = 64'd10; #10;
    ifun = 4'd2; valB = 64'd15; valA = 64'd15; #10;
    ifun = 4'd3; valB = 64'd15; valA = 64'd15; #10;
    ifun = 4'd1; valB = 64'd10; valA = 64'd15; #10;
    ifun = 4'd0; valB = 9223372036854775807;  valA = 2; #10;
    ifun = 4'd1; valB = -9223372036854775807; valA = 3; #10;
    ifun = 4'd2; valB = 64'd1; valA = 64'd1; #10;
    icode = 4'd4; valA = 64'd10; valB = 64'd11; valC = 64'd13; #10;
    icode = 4'd6; ifun = 4'd1; valB = 64'd10; valA = 64'd15; valC = 64'd10; #10;
    icode = 4'd2; ifun = 4'd1; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd6; ifun = 4'd1; valB = 64'd15; valA = 64'd15; valC = 64'd10; #10;
    icode = 4'd2; ifun = 4'd0; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd2; ifun = 4'd1; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd2; ifun = 4'd2; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd2; ifun = 4'd3; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd2; ifun = 4'd4; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd2; ifun = 4'd5; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd2; ifun = 4'd6; valB = 64'd10; valA = 64'd1; #10;
    icode = 4'd3; ifun = 4'd0; valC = 64'd1000;
    $finish;
end

initial begin
    $monitor("icode = %d, ifun = %d, valA = %d, valB = %d, out = %d, zf = %d, sf = %d, of = %d", icode, ifun, valA, valB, valE, cndflags[0]
end


endmodule
```

Memory

```
`timescale 10ns/1ps

module mem_tb;

initial begin
    $dumpfile("mem_tb.vcd");
    $dumpvars(0, mem_tb);
end

reg [3:0] icode;
reg [3:0] ifun;
reg signed [63:0] valE;
reg signed [63:0] valA;
reg signed [63:0] valP;
wire signed [63:0] valM;
wire signed [63:0] memdata;

mem m1
(
    .icode(icode),
    .ifun(ifun),
    .valE(valE),
    .valA(valA),
    .valP(valP),
    .valM(valM),
    .memdata(memdata)
);

initial begin
    icode = 4'd0; ifun = 4'd0; valE = 64'd0; valA = 64'd0; valP = 64'd0;
    #10;
    icode = 4'd1; ifun = 4'd0; valA = 64'd10; valE = 64'd10; valP = 64'd10;
    #10;
    icode = 4'd2; ifun = 4'd0;
    #10;
    icode = 4'd3; ifun = 4'd0;
```

```
    #10;
    icode = 4'd4; ifun = 4'd0;
    #10;
    icode = 4'd5; ifun = 4'd0;
    #10;
    icode = 4'd6; ifun = 4'd0;
    #10;
    icode = 4'd7; ifun = 4'd0;
    #10;
    icode = 4'd8; ifun = 4'd0;
    #10;
    icode = 4'd9; ifun = 4'd0;
    #10;
    icode = 4'd10; ifun = 4'd0;
    #10;
    icode = 4'd11; ifun = 4'd0;
end

initial
begin
    $monitor("icode = %d, ifun = %d, valA = %d, valE = %d, valP = %d, memdata = %d, valM = %d", icode, ifun, valA, valE, valP, memdata, val
end
endmodule
```

Writeback

```
`timescale 10ns/1ps

module writeback_tb;

initial begin
    $dumpfile("writeback_tb.vcd");
    $dumpvars(0, writeback_tb);
end

reg clk;
reg instr_valid;
reg signed [63:0] valE;
reg signed [63:0] valM;
reg [3:0] icode;
reg [3:0] ifun;
reg [3:0] rA, rB;
wire signed [63:0] regfile0, regfile1, regfile2, regfile3, regfile4, regfile5, regfile6, regfile7, regfile8, regfile9, regfile10, regfile11


writeback wb1(
    clk, rA, rB, valM, valE, icode, ifun,
    regfile0, regfile1, regfile2, regfile3, regfile4,
    regfile5, regfile6, regfile7, regfile8, regfile9,
    regfile10,regfile11,regfile12,regfile13,regfile14
);

parameter CLOCK_PER = 100;
initial
begin
    clk = 0;
end
always #CLOCK_PER
begin
    clk = ~clk;
end

initial begin
    instr_valid = 0;
    #10;
    instr_valid = 1; valE = 64'd0; valM = 64'd0; icode = 4'd0; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd2; ifun = 4'd0; rA = 4'd1; rB = 4'd2;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd3; ifun = 4'd0; rA = 4'd4; rB = 4'd3;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd4; ifun = 4'd0; rA = 4'd0; rB = 4'd1;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd5; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd6; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
```

```
    valE = 64'd10; valM = 64'd10; icode = 4'd7; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd8; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd9; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd10; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd11; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    instr_valid = 1; valE = 64'd0; valM = 64'd0; icode = 4'd0; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    valE = 64'd10; valM = 64'd10; icode = 4'd1; ifun = 4'd0; rA = 4'd0; rB = 4'd0;
    #10;
    $finish;
end

initial begin
    $monitor("valE = %d, valM = %d, icode = %d, ifun = %d, 0 = %0d,1 = %0d,2 = %0d,3 = %0d,4 = %0d,5 = %0d,6 = %0d,7 = %0d,8 = %0d,9 = %0d,
end
endmodule
```

PC_update

```
`timescale 10ns/1ps

module PCupd_tb;

initial
begin
    $dumpfile("PCupd_tb.vcd");
    $dumpvars(0, PCUpd_tb);
end

reg [3:0] icode;
reg [3:0] ifun;
reg jmpcnd;
reg signed [63:0] valC;
reg signed [63:0] valM;
reg signed [63:0] valP;
wire [63:0] PC;

PCupd p1
(
    .icode(icode),
    .ifun(ifun),
    .jmpcnd(jmpcnd),
    .valC(valC),
    .valM(valM),
    .valP(valP),
    .PC(PC)
);

initial begin
    icode = 4'd0; ifun = 4'd0; jmpcnd = 1'b0; valC = 64'd0; valP = 64'd0; valM = 64'd0;
    #10;
    icode = 4'd0; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd1; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd2; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd3; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd4; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd5; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd6; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd7; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd7; ifun = 4'd0; jmpcnd = 1'b0; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd8; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd9; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
```

```
    #10;
    icode = 4'd10; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
    icode = 4'd11; ifun = 4'd0; jmpcnd = 1'b1; valC = 64'ha; valP = 64'hb; valM = 64'hc;
    #10;
end

initial begin
    $monitor("icode = %d, ifun = %d, valC = %d, valM = %d, valP = %d, PC = %d", icode, ifun, valC, valM, valP, PC);
end
endmodule
```