IPA Project Report

Eshan Gupta, 2019102044

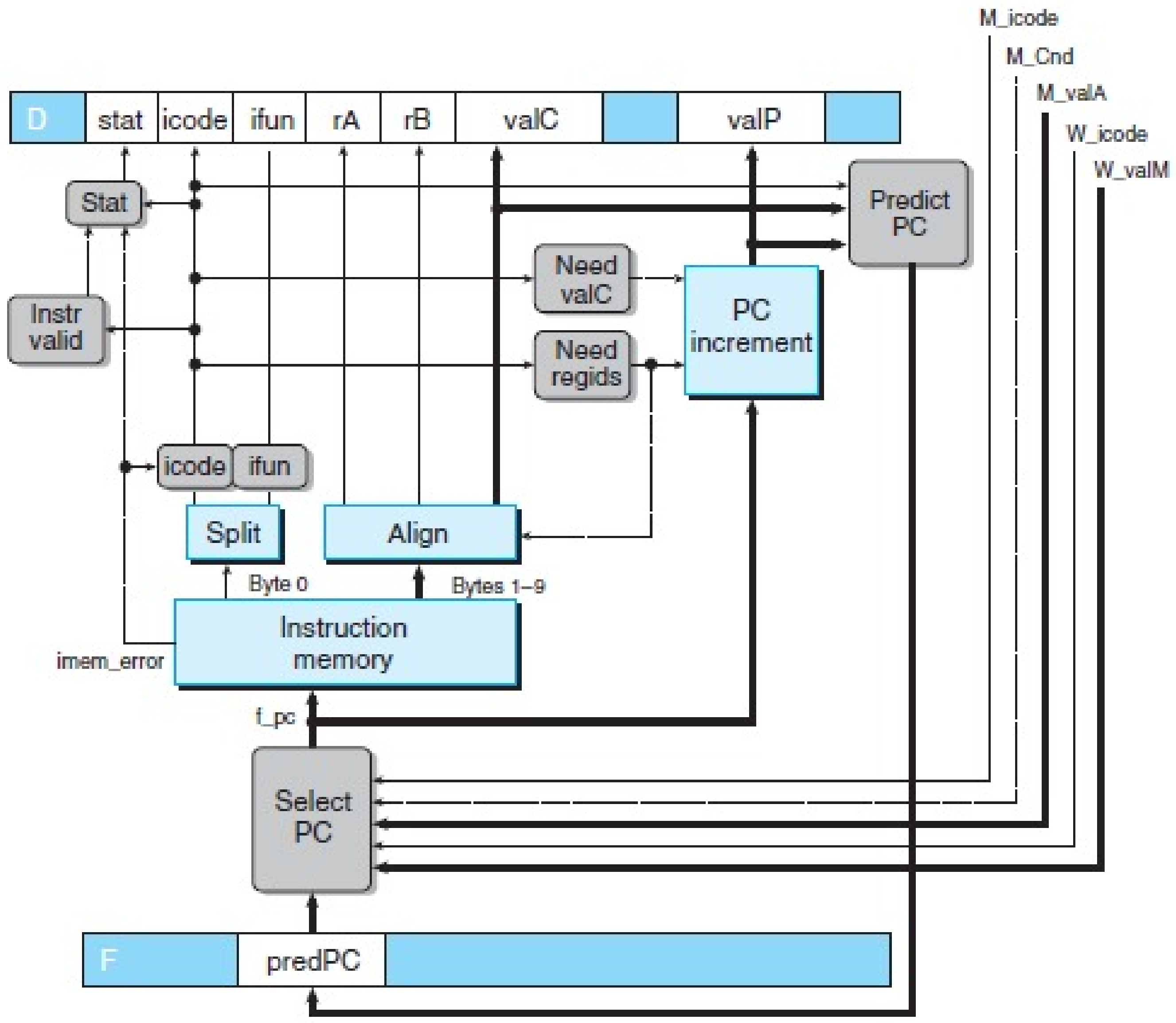
$10^{th} March, 2021$

Processor Architecture with 5 Stage Pipelined based on the structure of Y86 Processor

The processor architecture can be broadly divided into 4 modules described below

Fetch Module -

In the fetch stage, foremost step is to fetch the instruction from the memory address that is currently stored in the PC (program counter) and stored into the instruction register. Now, when this step ends, the PC points to the next instruction to be read in the next cycle and the cycle continues like this to fetch all the instructions. The memory hardware unit reads 10 bytes in a clock cycle from memory as per the sequence defined above, using the PC as the address of the first byte which is byte 0. Now, we pass this instructions to be split into 2 4-bits amounts and is interpreted as instruction byte. As per the diagram below, the instruction and function codes are then computed by the control logic blocks labelled "icode" and "ifun" as equaling either the values read from memory or, if the instruction address is invalid, the values corresponding to a nop instruction. Now, we can observe two variables named NeedRegids and NeedValC. NeedRegids instruction includes a register specific byte. NeedValC instruction includes a constant word. Now, the remaining 9 bytes encode a mix of the register specifier byte and constant word. Now, as seen in the figure, a hardware module *Align* converts these bytes into register fields and constant terms. When the computed signal NeedRegids is 1, byte 1 is split into register specifiers rA and rB. If NeedRegids is 0, all register specifiers are set to 0xF (RNONE), indicating that this instruction does not specify any registers. *Align* module is used to generate a constant word that is denoted in the diagram by valC, whose value depends upon the value of NeedRegids. The last module used in Fetch module is known as *PC Incrementer* that is used to generate signal named valP, which is based upon the other values from other units.



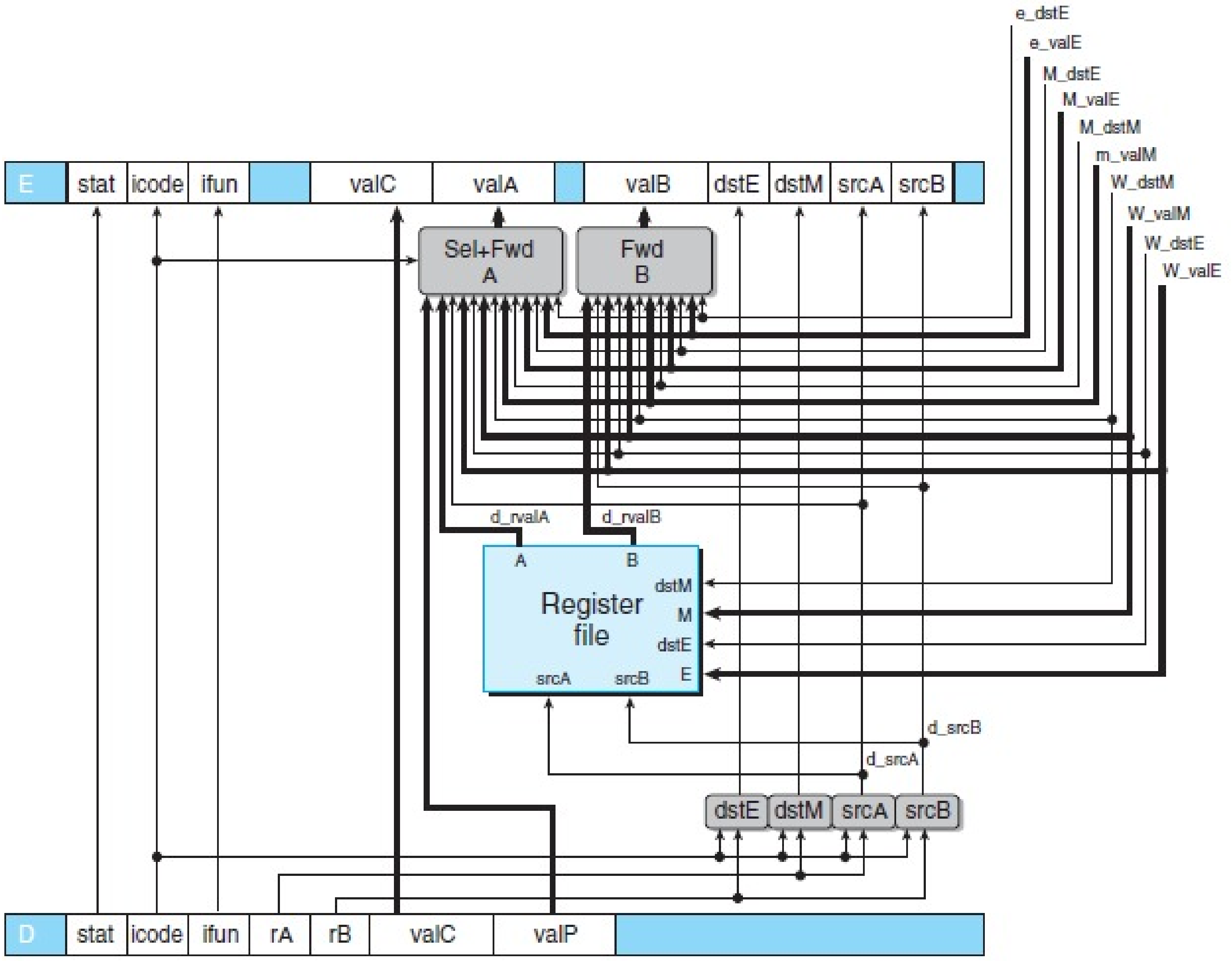
The Verilog code used for the same module is given below -

```
    // To give set of instructions to our processor as input module
InstructionMemory(f_pc,f_ibyte,f_ibytes,imem_error);
    input [63:0] f_pc;
    output reg[71:0] f_ibytes;
    output reg[7:0] f_ibyte;
    output reg imem_error;
    reg [7:0] instruction_mem [2047:0];

    initial
        begin
            $readmemh("./instructions.mem", instruction_mem);
            $display("init");
        end
    always @(f_pc)
    begin
        $display(instruction_mem[f_pc]);
        f_ibyte <= instruction_mem[f_pc];
        f_ibytes[71:64] <= instruction_mem[f_pc+1];
        f_ibytes[63:56] <= instruction_mem[f_pc+2];
        f_ibytes[55:48] <= instruction_mem[f_pc+3];
        f_ibytes[47:40] <= instruction_mem[f_pc+4];
        f_ibytes[39:32] <= instruction_mem[f_pc+5];
        f_ibytes[31:24] <= instruction_mem[f_pc+6];
        f_ibytes[23:16] <= instruction_mem[f_pc+7];
        f_ibytes[15:8] <= instruction_mem[f_pc+8];
        f_ibytes[7:0] <= instruction_mem[f_pc+9];

        imem_error <= (f_pc < 64'd0 || f_pc > 64'd2047 ) ? 1'b1:1'b0;
    end

endmodule
// Splitting module to generate icode and ifun

module split(f_ibyte, f_icode, f_ifun);
input [7:0] f_ibyte;
output [3:0] f_icode;
output [3:0] f_ifun;
assign f_icode = f_ibyte[7:4];
assign f_ifun = f_ibyte[3:0];
endmodule

// This module helps to extract the required 9 bit word in order to proceed

module align(f_ibytes, need_regids, rA, rB, valC);
input [71:0] f_ibytes;
input need_regids;
output [ 3:0] rA;
output [ 3:0] rB;
output [63:0] valC;
assign rA = f_ibytes[71:68];
assign rB = f_ibytes[67:64];
assign valC = need_regids ? f_ibytes[63:0] : f_ibytes[71:8];
endmodule

// This module increments the count of PC as per our requirements

module pc_increment(pc, need_regids, need_valC, valP);
input [63:0] pc;
input need_regids;
input need_valC;
output [63:0] valP;
```

```
assign valP = pc + 1 + 8*need_valC + need_regids;
endmodule
```

Decode and Writeback Module -

The decode module is used to interpret the encoded instruction that is provided in the instruction register at a given clock cycle. The block named register file in the diagram is used to get the values from the registers used in the instructions which is to be decoded. It can be seen in the diagram below that the register file has 4 ports (entry and exit marked with bold arrows). Now, each port holds two properties namely an address and a data connection. The register ID is stored in Address Connection while the data connection contains the set of 64 wires (64 bit) for output or the input word. The ports A and B have address inputs as srcA and srcB. Ports E and M have address inputs dstE and dstM. The destination where valE is stored is held by the register ID dstE, similarly for valM its held by dstM. $Sel + FwdA$ merges the valP signal with valA signal for later stages in order to reduce the amount of state in the pipeline register and also implements the forwarding logic for source operand valA. $FwdB$ implements the forwarding logic for source operand valB.



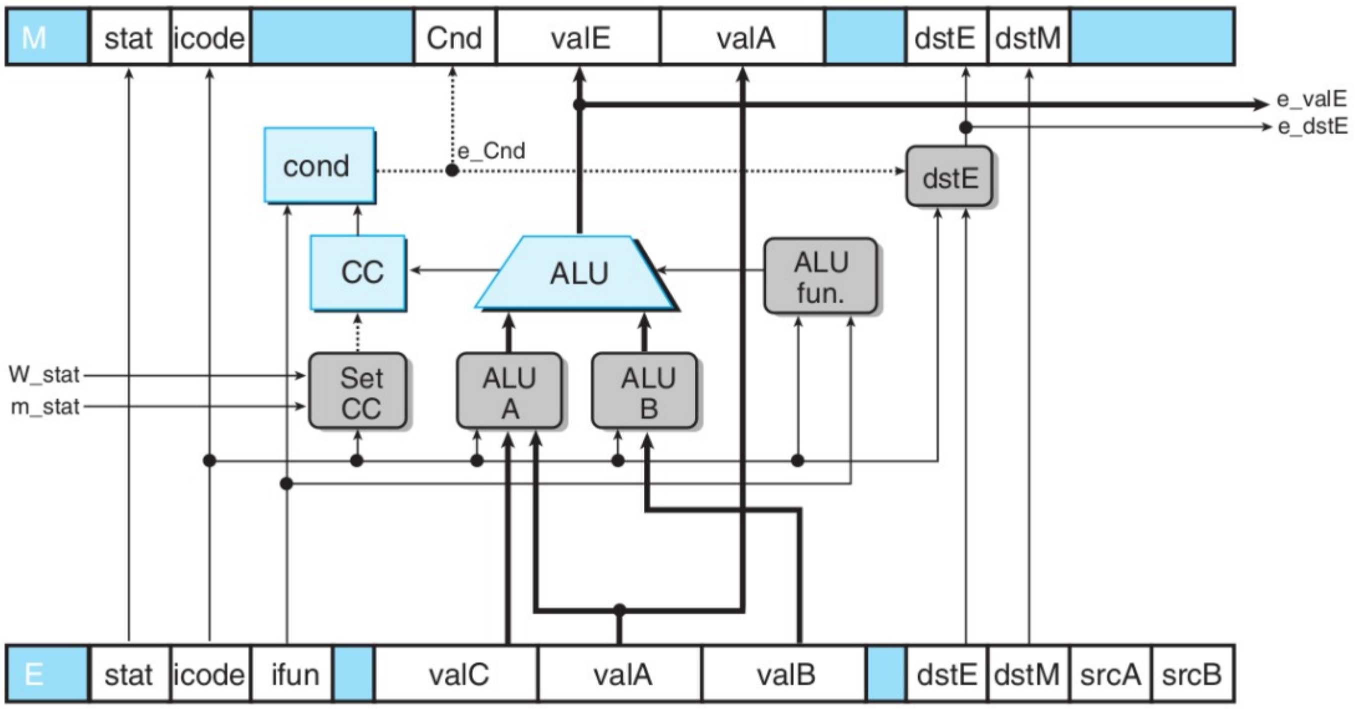
The Verilog code used for the same module is given below -

```
// This module is used to create a clocked register with enable signal
// and synchronous reset
 module cenrreg(out, in, enable, reset, resetval, clock);
 parameter width = 8;
 output [width-1:0] out;
 reg [width-1:0] out;
 input [width-1:0] in;
 input enable;
 input reset;
 input [width-1:0] resetval;
 input clock;

 always @(posedge clock) begin
    if (reset)
        out <= resetval;
    else if (enable)
        out <= in;
    end
 endmodule


 // This module creates Pipeline register. Uses reset signal to inject bubble
 // When bubbling, must specify value that will be loaded
 module pipreg(out, in, stall, bubble, bubbleval, clock);

 parameter width = 8;
 output [width-1:0] out;
 input [width-1:0] in;
 input stall, bubble;
 input [width-1:0] bubbleval;
 input clock;

 cenrreg #(width) r(out, in, ~stall, bubble, bubbleval, clock);
 endmodule


 // This module is used to create Register file
 module regfile(dstE, valE, dstM, valM, srcA, valA, srcB, valB, reset, clock,
 rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi,
 r8, r9, r10, r11, r12, r13, r14);
 input [ 3:0] dstE;
 input [63:0] valE;
 input [ 3:0] dstM;
 input [63:0] valM;
 input [ 3:0] srcA;
 output [63:0] valA;
 input [ 3:0] srcB;
 output [63:0] valB;
 // Reset is used to set all registers to 0
 input reset;
 input clock;
 output [63:0] rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi,
 r8, r9, r10, r11, r12, r13, r14;

 // Define names for registers used in HCL code

parameter RRAX = 4'h0;
parameter RRCX = 4'h1;
parameter RRDX = 4'h2;
parameter RRBX = 4'h3;
parameter RRSP = 4'h4;
parameter RRBP = 4'h5;
parameter RRSI = 4'h6;
parameter RRDI = 4'h7;
parameter R8 = 4'h8;
parameter R9 = 4'h9;
parameter R10 = 4'ha;
parameter R11 = 4'hb;
parameter R12 = 4'hc;
parameter R13 = 4'hd;
parameter R14 = 4'he;
parameter RRNONE = 4'hf;
parameter BIT0 = 64'b0;

 // Input data for each register is defined here
 wire [63:0] rax_dat, rcx_dat, rdx_dat, rbx_dat,
 rsp_dat, rbp_dat, rsi_dat, rdi_dat,
 r8_dat, r9_dat, r10_dat, r11_dat,

 r12_dat, r13_dat, r14_dat;

 // Input write controls for each register is defined here
 wire rax_wrt, rcx_wrt, rdx_wrt, rbx_wrt,
 rsp_wrt, rbp_wrt, rsi_wrt, rdi_wrt,
 r8_wrt, r9_wrt, r10_wrt, r11_wrt,
 r12_wrt, r13_wrt, r14_wrt;


 reg temp = 1'b0;
 cenrreg #(64) rax_reg(rax, rax_dat, rax_wrt, temp, BIT0, clock);
 cenrreg #(64) rcx_reg(rcx, rcx_dat, rcx_wrt, temp, BIT0, clock);
 cenrreg #(64) rdx_reg(rdx, rdx_dat, rdx_wrt, temp, BIT0, clock);
 cenrreg #(64) rbx_reg(rbx, rbx_dat, rbx_wrt, temp, BIT0, clock);
 cenrreg #(64) rsp_reg(rsp, rsp_dat, rsp_wrt, temp, BIT0, clock);
 cenrreg #(64) rbp_reg(rbp, rbp_dat, rbp_wrt, temp, BIT0, clock);
 cenrreg #(64) rsi_reg(rsi, rsi_dat, rsi_wrt, temp, BIT0, clock);
 cenrreg #(64) rdi_reg(rdi, rdi_dat, rdi_wrt, temp, BIT0, clock);
 cenrreg #(64) r8_reg(r8, r8_dat, r8_wrt, temp, BIT0, clock);
 cenrreg #(64) r9_reg(r9, r9_dat, r9_wrt, temp, BIT0, clock);
 cenrreg #(64) r10_reg(r10, r10_dat, r10_wrt, temp, BIT0, clock);
 cenrreg #(64) r11_reg(r11, r11_dat, r11_wrt, temp, BIT0, clock);
 cenrreg #(64) r12_reg(r12, r12_dat, r12_wrt, temp, BIT0, clock);
 cenrreg #(64) r13_reg(r13, r13_dat, r13_wrt, temp, BIT0, clock);
 cenrreg #(64) r14_reg(r14, r14_dat, r14_wrt, temp, BIT0, clock);

 // Reads occur like combinational logic here
 assign valA =
 srcA == RRAX ? rax :
 srcA == RRCX ? rcx :
 srcA == RRDX ? rdx :
 srcA == RRBX ? rbx :
 srcA == RRSP ? rsp :
 srcA == RRBP ? rbp :
 srcA == RRSI ? rsi :
 srcA == RRDI ? rdi :
 srcA == R8 ? r8 :
 srcA == R9 ? r9 :
 srcA == R10 ? r10 :
 srcA == R11 ? r11 :
 srcA == R12 ? r12 :
 srcA == R13 ? r13 :
 srcA == R14 ? r14 :
 0;

 assign valB =
 srcB == RRAX ? rax :
 srcB == RRCX ? rcx :
 srcB == RRDX ? rdx :
 srcB == RRBX ? rbx :

 srcB == RRSP ? rsp :
 srcB == RRBP ? rbp :
 srcB == RRSI ? rsi :
 srcB == RRDI ? rdi :
 srcB == R8 ? r8 :
 srcB == R9 ? r9 :
 srcB == R10 ? r10 :
 srcB == R11 ? r11 :
 srcB == R12 ? r12 :
 srcB == R13 ? r13 :
 srcB == R14 ? r14 :
```

```
0;

assign rax_dat = dstM == RRAX ? valM : valE;
assign rcx_dat = dstM == RRCX ? valM : valE;
assign rdx_dat = dstM == RRDX ? valM : valE;
assign rbx_dat = dstM == RRBX ? valM : valE;
assign rsp_dat = dstM == RRSP ? valM : valE;
assign rbp_dat = dstM == RRBP ? valM : valE;
assign rsi_dat = dstM == RRSI ? valM : valE;
assign rdi_dat = dstM == RRDI ? valM : valE;
assign r8_dat = dstM == R8 ? valM : valE;
assign r9_dat = dstM == R9 ? valM : valE;
assign r10_dat = dstM == R10 ? valM : valE;
assign r11_dat = dstM == R11 ? valM : valE;
assign r12_dat = dstM == R12 ? valM : valE;
assign r13_dat = dstM == R13 ? valM : valE;
assign r14_dat = dstM == R14 ? valM : valE;

assign rax_wrt = dstM == RRAX | dstE == RRAX;
assign rcx_wrt = dstM == RRCX | dstE == RRCX;
assign rdx_wrt = dstM == RRDX | dstE == RRDX;
assign rbx_wrt = dstM == RRBX | dstE == RRBX;
assign rsp_wrt = dstM == RRSP | dstE == RRSP;
assign rbp_wrt = dstM == RRBP | dstE == RRBP;
assign rsi_wrt = dstM == RRSI | dstE == RRSI;
assign rdi_wrt = dstM == RRDI | dstE == RRDI;
assign r8_wrt = dstM == R8 | dstE == R8;
assign r9_wrt = dstM == R9 | dstE == R9;
assign r10_wrt = dstM == R10 | dstE == R10;
assign r11_wrt = dstM == R11 | dstE == R11;
assign r12_wrt = dstM == R12 | dstE == R12;
assign r13_wrt = dstM == R13 | dstE == R13;
assign r14_wrt = dstM == R14 | dstE == R14;

endmodule
```

Execute -

The execute module is used to carry out the functional operations which are implemented by *ALU*. This module is used to send the decoded data as control signals to the functional units and then the required instructions are carried out before being transferred to the ALU for all the mathematical and logical operations and the result is written back into the register. The resulting output is either sent to the main memory or to output unit. As seen in the diagram *ALU* takes two inputs and the resulting output is sent to valE. In this stage, the foremost step is to compute ALU for every instruction given in clock cycle. Now, another module used in this stage is condition code register. When ALU runs, it generates 3 signals that decide the action of the condition codes.They are zero, symbol or overflow. setcc signal controls whether or not the condition code register should be updated and has signal mstat and Wstat as inputs which detect cases where an instruction causing an exception is passing through later pipeline stages. We use ahrdware module called *Cond* to decide if conditional branch or data transfer should take place. It generates Cnd signal used in the next PC logic and to set dstE with conditonal moves.



The verilog code for this module is as follows -

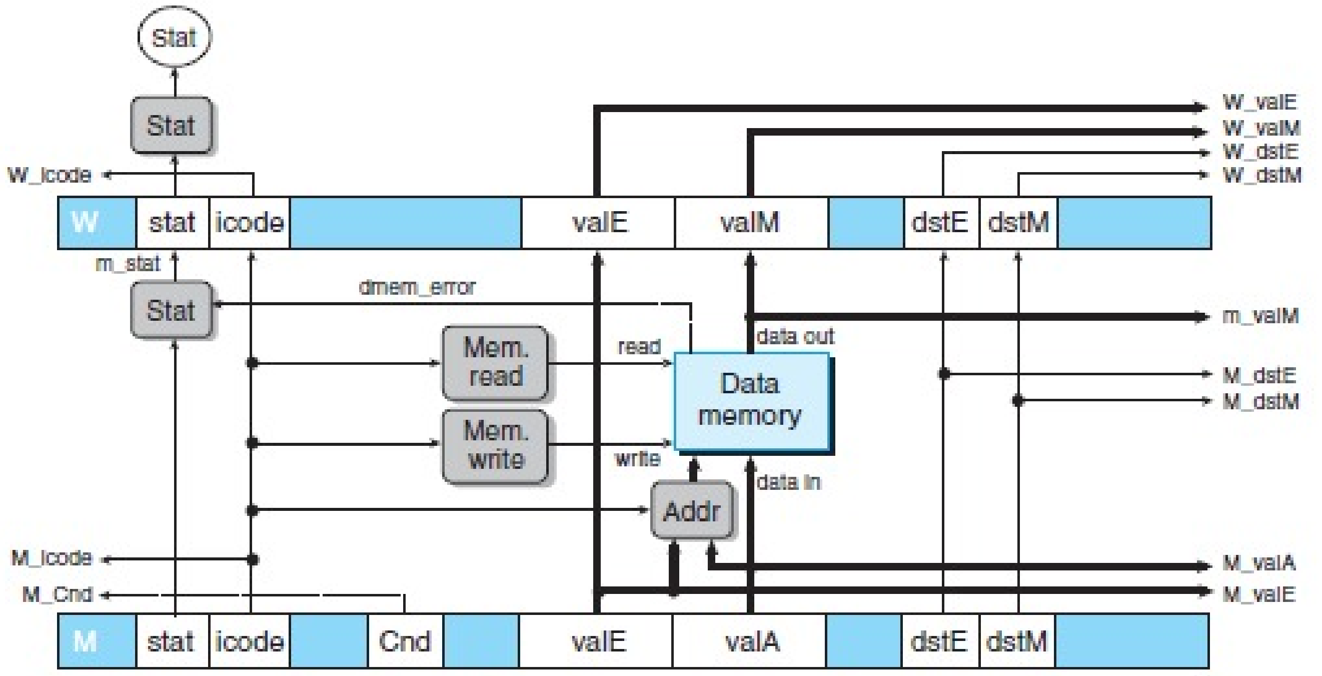
```
 // This module creates Pipeline register.
 //Uses reset signal to inject bubble When bubbling,
 //must specify value that will be loaded
module pipreg1(out, in, stall,bubble, bubbleval, clock);

 parameter width = 8;
 output reg [width-1:0] out;
 input [width-1:0] in;
 input stall,bubble;
 input [width-1:0] bubbleval;
 input clock;

 initial begin
     out <= bubbleval;
 end
 always @(posedge clock) begin
    if (!stall && !bubble)
        out <= in;
    else if (!stall && bubble)
        out <= bubbleval;
    end
 endmodule
```

```
module pipreg2(out, in, stall, bubbleval, clock);

 parameter width = 8;
 output reg [width-1:0] out;
 input [width-1:0] in;
 input stall;
 input [width-1:0] bubbleval;
 input clock;

 initial begin
     out <= bubbleval;
 end
 always @(posedge clock) begin
    if (!stall)
        out <= in;
    end

 endmodule




// This module creates ALU used in our processor
module alu(aluA, aluB, alufun, valE, new_cc);
 // Data inputs, ALU function, Data Output
input [63:0] aluA, aluB;
input [ 3:0] alufun;
output [63:0] valE;
output [ 2:0] new_cc;

parameter ALUADD = 4'h0;
parameter ALUSUB = 4'h1;
parameter ALUAND = 4'h2;
parameter ALUXOR = 4'h3;


assign valE =
    alufun == ALUSUB ? aluB - aluA :
    alufun == ALUAND ? aluB & aluA :
    alufun == ALUXOR ? aluB ^ aluA :
    aluB + aluA;
assign new_cc[2] = (valE == 0); // ZF
assign new_cc[1] = valE[63]; // SF
assign new_cc[0] = // OF
    alufun == ALUADD ?
        (aluA[63] == aluB[63]) & (aluA[63] != valE[63]) :
```

```
    alufun == ALUSUB ?
        (~aluA[63] == aluB[63]) & (aluB[63] != valE[63]) :
    0;
endmodule

// Condition code register is defined here
module cc(cc, new_cc, set_cc, reset, clock);
output[2:0] cc;
input [2:0] new_cc;
input set_cc;
input reset;
input clock;

pipreg2 #(3) c(cc, new_cc, ~set_cc, 3'b100, clock);
endmodule

// branch condition logic is defined here
module cond(ifun, cc, Cnd);
input [3:0] ifun;
input [2:0] cc;
output Cnd;

wire zf = cc[2];
wire sf = cc[1];
wire of = cc[0];

// Jump & move conditions are defined here
parameter C_YES = 4'h0;
parameter C_LE = 4'h1;
parameter C_L = 4'h2;
parameter C_E = 4'h3;
parameter C_NE = 4'h4;
parameter C_GE = 4'h5;
parameter C_G = 4'h6;

assign Cnd =
(ifun == C_YES) | //
(ifun == C_LE & ((sf^of)|zf)) | // less than equal to condition
(ifun == C_L & (sf^of)) | // less than condition
(ifun == C_E & zf) | // equal to condition
(ifun == C_NE & ~zf) | // not equal to condition
(ifun == C_GE & (~sf^of)) | // greater than equal to condition
(ifun == C_G & (~sf^of)&~zf); // greater than condition

endmodule
```

Memory Module -

This is the final module used while developing this procoessor design. In this stage, memory operands and read and written from/to memory. When the read operation is performed, valM is generated. valE and valM are addresses for memory read and write. The selection of data sources between valP and valA is performed by $sel + FwdA$ in the decode stage.



The verilog code for this module is as follows -

```
// The module defines the memory block of the processor
module data_memory(mem_addr,M_valA,mem_read,mem_write,mem_data,dmem_error);

input [63:0] mem_addr;
input [63:0] M_valA;
input mem_read;
input mem_write;
output reg [63:0] mem_data;
output reg dmem_error;
reg [63:0] mem [8191:0];

initial begin
    dmem_error <= 0;
end

always @ (mem_addr,M_valA,mem_read,mem_write) begin
    if (mem_write && !mem_read)
        mem[mem_addr] = M_valA;
    if (!mem_write && mem_read)
        mem_data = mem[mem_addr];
    end
endmodule
```

The following instructions are supported by the processor -

1. Instruction encodings range between 1 and 10 bytes.
2. halt
3. nop
4. rrmovq rA, rB
5. irmovq V, rB
6. rmmovq rA, D(rB)
7. mrmovq D(rB), rA
8. Opq rA, rB
9. jXX Dest
10. cmovXX rA, rB
11. call Dest
12. ret
13. pushq rA
14. popq rA

The codes for HCF for 2 numbers are as follows

C++ code for HCF of two nos. -

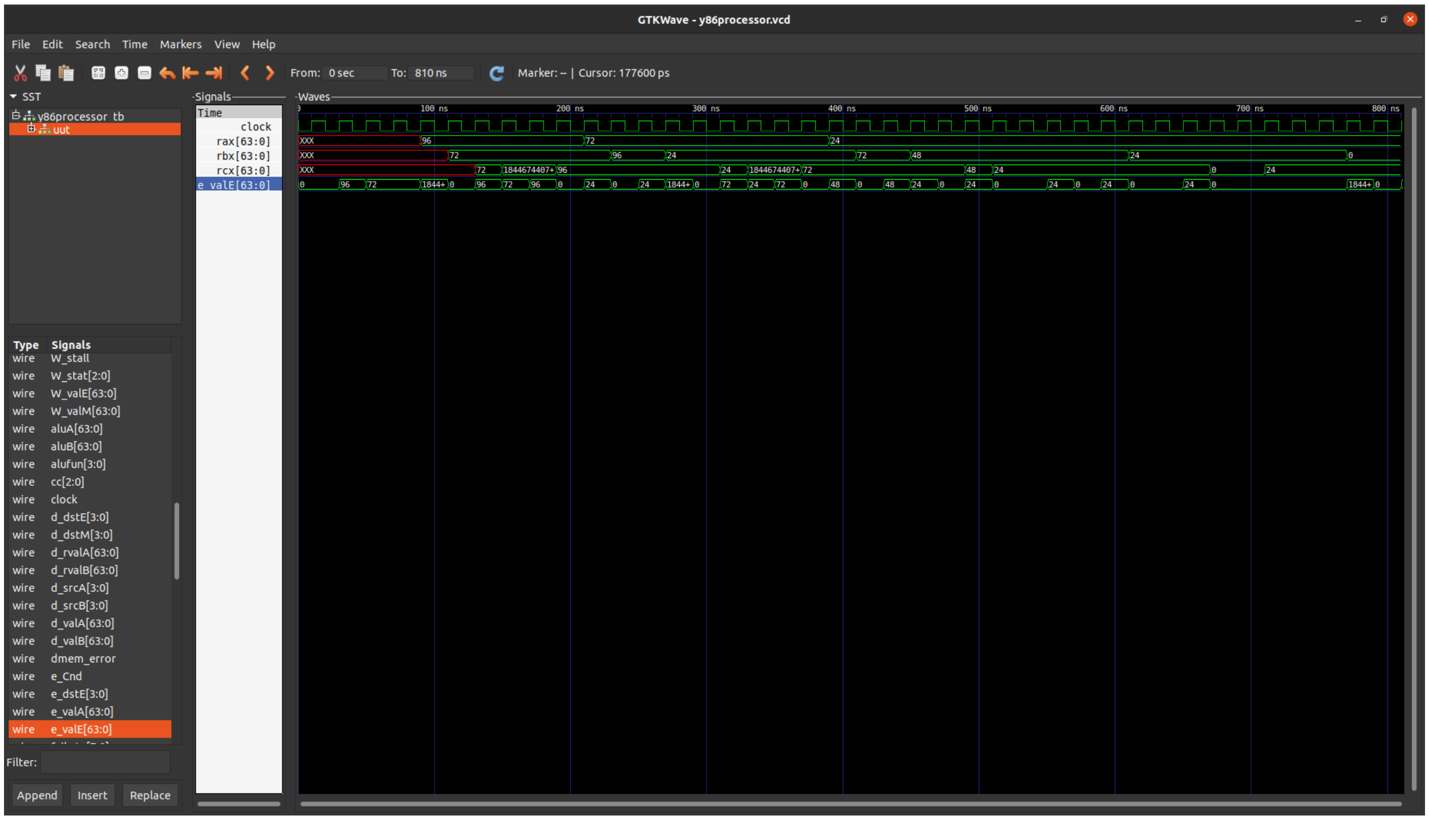
```
#include<bits/stdc++.h>
using namespace std;
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main()
{
    int a=96,b=72;
    cout<<gcd(96,72)<<endl;
}
```

Encoded instructions are as follows -

```
30
F0
00
00
00
00
00
00
00
60
30
F3
00
00
00
00
00
00
00
48
20
31
61
01
72
00
00
00
00
00
00
00
36
76
00
00
00
00
00
00
00
2B
00
61
03
70
```

```
00
00
00
00
00
00
00
14
20
01
20
30
20
13
70
00
00
00
00
00
00
00
2B
```

GTKWave Output



Instructions to run the code in ubuntu -

```
1 Extract all the files to your system
2 Open the folder where your files are in the terminal
3 Run the command iverilog y86processor_tb.v Y86Processor.v
4 Run the command ./a.out
5 Run the command gtkwave y86processor.vcd to get the desired gtkwave output
```