IPA PROJECT MID REPORT

Gokulraj R - 2020102042

Ruhul Ameen S - 2020102031

Sequential Design

this is implemented in a sequential basis with 6 blocks as discussed below

Fetch

It reads bytes of instructions from the memory using the input of PC address. It also splits out icode and ifun from the instruction. according to that, it gives the register specifiers rA and rB and also calculates valP which gives the new PC address for the next the sequential process of next instruction.

- Source code

```
`timescale 1ns / 1ps

module fetch (
    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
);

reg instr_valid;
reg imem_error;
reg hlt;

reg [7:0] instr_mem[0:1023]; // 1024 bytes
/* reg [0:8191] instr_mem; // same 1024 bytes */
/* reg [7:0] instr_mem[1023:0]; // 1024 elements of 8-bit each */

reg [0:79] instr_read; // 10 bytes
/* reg [7:0] instr_read[9:0]; */
/* reg [7:0] instr_read[0:9]; */

initial
```

```
begin

    instr_mem[0]=8'b00110000; //3 0
    instr_mem[1]=8'b00000000; //F rB=0
    instr_mem[2]=8'b00000000;
    instr_mem[3]=8'b00000000;
    instr_mem[4]=8'b00000000;
    instr_mem[5]=8'b00000000;
    instr_mem[6]=8'b00000000;
    instr_mem[7]=8'b00000000;
    instr_mem[8]=8'b00000000;
    instr_mem[9]=8'b00000000; //V=0

end

genvar i;

always @ (posedge clk)
begin
    imem_error = 0;
    // check for memory error
    if (PC > 1023)
    begin
        imem_error = 1;
    end

    /* generate for (i = 0; i < 10; i = i + 1) */
    /*     begin */
    /*         instr_read[i] = instr_mem[PC + i]; */
    /*     end */
    /* endgenerate */

    instr_read = {
        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_mem[PC+7],
        instr_mem[PC+8],
        instr_mem[PC+9]
```

```
    };

/* instr_read[0] = instr_mem[PC]; */
/* instr_read[8] = instr_mem[PC + 1]; */
/* instr_read[16] = instr_mem[PC + 2]; */
/* instr_read[24] = instr_mem[PC + 3]; */
/* instr_read[32] = instr_mem[PC + 4]; */
/* instr_read[40] = instr_mem[PC + 5]; */
/* instr_read[48] = instr_mem[PC + 6]; */
/* instr_read[56] = instr_mem[PC + 7]; */
/* instr_read[64] = instr_mem[PC + 8]; */
/* instr_read[72] = instr_mem[PC + 9]; */

/* instr_read[0] = instr_mem[PC]; */
/* instr_read[1] = instr_mem[PC + 1]; */
/* instr_read[2] = instr_mem[PC + 2]; */
/* instr_read[3] = instr_mem[PC + 3]; */
/* instr_read[4] = instr_mem[PC + 4]; */
/* instr_read[5] = instr_mem[PC + 5]; */
/* instr_read[6] = instr_mem[PC + 6]; */
/* instr_read[7] = instr_mem[PC + 7]; */
/* instr_read[8] = instr_mem[PC + 8]; */
/* instr_read[9] = instr_mem[PC + 9]; */

// check the other methods of doing this

/* instr_read = instr_mem[PC : PC+9]; */
/* instr_read = instr_mem[PC : PC+79]; */

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

instr_valid = 1'b1; // assume that the instr is valid

case (icode)
    4'b0000:begin // halt
        hlt = 1;
        valP = PC + 64'd1;
    end
    4'b0001:begin // nop
        valP = PC + 64'd1;
    end
    4'b0010:begin // cmovXX
```

```
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valP = PC + 64'd2;
        end
        4'b0011:begin // irmovq
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valC = instr_read[16:79];
            valP = PC + 64'd10;
        end
        4'b0100:begin // rmmovq
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valC = instr_read[16:79];
            valP = PC + 64'd10;
        end
        4'b0101:begin // mrmovq
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valC = instr_read[16:79];
            valP = PC + 64'd10;
        end
        4'b0110:begin // OPq
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valP = PC + 64'd2;
        end
        4'b0111:begin // jXX
            valC = instr_read[8:71];
            valP = PC + 64'd9;
        end
        4'b1000:begin // call
            valC = instr_read[8:71];
            valP = PC + 64'd9;
        end
        4'b1001:begin // ret
            valP = PC + 64'd1;
        end
        4'b1010:begin // pushq
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valP = PC + 64'd2;
        end
```

```
        4'b1011:begin // popq
            rA = instr_read[8:11];
            rB = instr_read[12:15];
            valP = PC + 64'd2;
        end
        default:begin
            instr_valid = 1'b0;
        end
    endcase

end

endmodule
```

Decode

It gets the values valA and valB which was stored in registers using operands rA and rB.

- Source code

```
`timescale 1ns / 1ps

module decode (
    input clk,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    output reg [63:0] valA,
    output reg [63:0] valB,

    output reg [63:0] registers[14:0]

);

initial
begin
    registers[0] = 64'd0;
    registers[1] = 64'd1;
    registers[2] = 64'd2;
    registers[3] = 64'd3;
    registers[4] = 64'd4;
    registers[5] = 64'd5;
    registers[6] = 64'd6;
    registers[7] = 64'd7;
    registers[8] = 64'd8;
```

```
    registers[9] = 64'd9;
    registers[10] = 64'd10;
    registers[11] = 64'd11;
    registers[12] = 64'd12;
    registers[13] = 64'd13;
    registers[14] = 64'd14;
end

always @ (*) // why??
begin
    case(icode)
        4'b0010:begin // cmovXX
            valA = registers[rA];
        end
        4'b0010:begin // rmmovq
            valA = registers[rA];
            valB = registers[rB];
        end
        4'b0010:begin // mrmovq
            valB = registers[rB];
        end
        4'b0010:begin // OPq
            valA = registers[rA];
            valB = registers[rB];
        end
        4'b0010:begin //call
            valB = registers[4]; // 4 is %rsp
        end
        4'b0010:begin // ret
            valA = registers[4]; // 4 is %rsp
            valB = registers[4];
        end
        4'b0010:begin // pushq
            valA = registers[rA];
            valB = registers[4];
        end
        4'b0010:begin // popq
            valA = registers[4];
            valB = registers[4];
        end

endmodule
```

Execute

it performs the logic and arithmetic operations if icode denote OPq functions with the help of Alu built to perform such actions. it also computes the move/jump flag in case of other functions by setting condition code and also responsible for change in stack pointer.

- Source code

```
`timescale 1ns / 1ps

`include "ALU/alu/alu.v"

module execute (
    input Clk,
    input [3:0] icode,
    input [3:0] ifun,
    input [63:0] valA,
    input [63:0] valB,
    input [63:0] valC,
    output reg ZF,
    output reg SF,
    output reg OF,
    output reg cnd,
    output reg [63:0] valE
);

always @(*) begin

    if (icode==4'b0110 && Clk==1) begin
        ZF=(out==0);
        SF=(out<0);
        OF=(A<0==B<0)&&(out<0!=A<0);
    end
end

initial begin
    ZF=0;
    SF=0;
    OF=0;

end
```

```
reg [1:0] control_input;
reg signed [63:0] A;
reg signed [63:0] B;
wire signed [63:0] out;
wire overfow;
reg signed [63:0] out_last;
alu64bit ALU1(
    control_input,
    A,
    B,
    out,
    overflow
);


initial begin
    control_input=0;
    A=64'd0;
    B=64'd0;
end

always @(*)
begin
    if (Clk==1) begin
        case (icode)
            4'b0010:begin
                case(ifun)
                    4'b0000:begin//rrmovq
                        cnd=1;
                    end
                    4'b0001:begin//cmovle
                        if((SF^OF)|ZF)
                        begin
                            cnd=1;
                        end
                    end
                    4'b0010:begin//cmovl
                        if ((SF^OF)) begin
                            cnd=1;
                        end
                    end
                    4'b0011:begin//cmove
                        if (ZF) begin
```

```
                    cnd=1;
                end
            end
            4'b0100:begin//cmovne
                if (~ZF) begin
                    cnd=1;
                end
            end
            4'b0101:begin//cmovge
                if ((~(SF^OF))) begin
                    cnd=1;
                end
            end
            4'b0110:begin//cmovg
                if ((~(SF^OF))&(~ZF)) begin
                    cnd=1;
                end
            end
        endcase
        valE=valA+valB;
    end
    4'b0011:begin//irmovq
        valE=valC+valB;
    end
    4'b0100:begin//rmmovq
        valE=valC+valB;
    end
    4'b0101:begin//mrmovq
        valE=valC+valB;
    end
    4'b0110:begin

        case (ifun)
            4'b0000:begin//addq
                A=valA;
                B=valB;
                control_input=0;

            end
            4'b0001:begin//subq
                A=valA;
                B=valB;
                control_input=1;
```

```
                end
                4'b0010:begin//andq
                    A=valA;
                    B=valB;
                    control_input=2;
                end
                4'b0011:begin//xorq
                    A=valA;
                    B=valB;
                    control_input=3;
                end
            endcase
            assign out_last=out;
            valE=out_last;
        end
        4'b0111:begin
            case(ifun)
                4'b0000:begin//jmp
                    cnd=1;
                end
                4'b0001:begin//jle
                    if((SF^OF)|ZF)
                    begin
                        cnd=1;
                    end
                end
                4'b0010:begin//jl
                    if ((SF^OF)) begin
                        cnd=1;
                    end
                end
                4'b0011:begin//je
                    if (ZF) begin
                        cnd=1;
                    end
                end
                4'b0100:begin//jne
                    if (~ZF) begin
                        cnd=1;
                    end
                end
                4'b0101:begin//jge
                    if ((~(SF^OF))) begin
```

```
                            cnd=1;
                        end
                    end
                    4'b0110:begin//jg
                        if ((~(SF^OF))&(~ZF)) begin
                            cnd=1;
                        end
                    end
                endcase
            end
            4'b1000:begin//call
                valE=valB+(-64'd8);
            end
            4'b1001:begin//ret
                valE=valB+64'd8;
            end
            4'b1010:begin//pushq
                valE=valB+(-64'd8);
            end
            4'b1011:begin//popq
                valE=valB+64'd8;
            end

        endcase
    end

end

endmodule
```

memory

reads or writes the data from the memory for particular functions like rmmovq,mrmovq, etc.

- Source Code

```
timescale 1ns / 1ps

module memory(
    input clk,
    input [3:0] icode,
    input [63:0] valA,
    input [63:0] valB,
    input [63:0] valE,
    input [63:0] valP,
```

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

/* reg [63:0] data_memory[0:1023]; */
reg [63:0] data_memory[1023:0];

always @ (*)
begin
    case(icode)
        4'b0100:begin // rmmovq
            data_memory[valE] = valA;
        end
        4'b0101:begin // mrmovq
            valM = data_memory[valE]
        end
        4'b1000:begin // call
            data_memory[valE] = valP;
        end
        4'b1001:begin // ret
            valM = data_memory[valA];
        end
        4'b1010:begin // pushq
            data_memory[valE] = valA;
        end
        4'b1000:begin // popq
            valM = data_memory[valE]
        end
    endcase

    /* datamem = data_memory[valE]; // why?? */
end

endmodule
```

Write back

writes the computed values from ALU or data output of the memory block in the register file.

- Source code

```
timescale 1ns / 1ps

module write_back (
   input clk,
```

```
   input cnd,
   input [3:0] icode,
   input [3:0] rA,
   input [3:0] rB,

   input valE,
   input valM,

   /* output reg [63:0] registers[0:14] */
   output reg [63:0] reg0,
   output reg [63:0] reg1,
   output reg [63:0] reg2,
   output reg [63:0] reg3,
   output reg [63:0] reg4,
   output reg [63:0] reg5,
   output reg [63:0] reg6,
   output reg [63:0] reg7,
   output reg [63:0] reg8,
   output reg [63:0] reg9,
   output reg [63:0] reg10,
   output reg [63:0] reg11,
   output reg [63:0] reg12,
   output reg [63:0] reg13,
   output reg [63:0] reg14
);

reg [63:0] registers[0:14];
/* reg [63:0] registers[14:0]; */

always @ (negedge clk)
begin
   case(icode)
       4'b0010:begin // cmovXX
           if(cnd == 1'b1)
           begin
               registers[rB] = valE;
           end
       end
       4'b0011:begin // irmovq
           registers[rB] = valE;
       end
       4'b0101:begin // mrmovq
           registers[rA] = valM;
```

```
        end
        4'b0110:begin // OPq
            registers[rB] = valE;
        end
        4'b1000:begin // call
            registers[4] = valE; // 4 is %rsp
        end
        4'b1001:begin // ret
            registers[4] = valE; // 4 is %rsp
        end
        4'b1010:begin // pushq
            registers[4] = valE; // 4 is %rsp
        end
        4'b1011:begin // popq
            registers[4] = valE; // 4 is %rsp
            registers[rA] = valM;
        end
    endcase

    reg0 = registers[0];
    reg1 = registers[1];
    reg2 = registers[2];
    reg3 = registers[3];
    reg4 = registers[4];
    reg5 = registers[5];
    reg6 = registers[6];
    reg7 = registers[7];
    reg8 = registers[8];
    reg9 = registers[9];
    reg10 = registers[10];
    reg11 = registers[11];
    reg12 = registers[12];
    reg13 = registers[13];
    reg14 = registers[14];

end
```

PC update

PC address is set to valP value to execute the next instruction in the instruction memory.

- Source code

```
`timescale 1ns / 1ps
```

```
module pc_update (
    input clk,
    input [63:0] PC,
    input cnd,
    input [63:0] valC,
    input [63:0] valP,
    input [63:0] valM,

    output reg [63:0] newPC
);

always @ (*)
begin
    case(icode)
        4'b0111:begin // jXX
            if(cnd == 1'b1)
                newPC = valC;
            else
                newPC = valP;
        end
        4'b1000:begin // call
            newPC = valC;
        end
        4'b0111:begin // ret
            newPC = valM;
        end
        default:begin
            newPC = valP;
        end
    endcase
end

endmodule
```

Sequential implementation using all the blocks

It calls the all the blocks sequentially from fetch to PC update to perform the instruction sets pointed by the PC address.

- Source code

```
timescale 1ns / 1ps

`include "fetch.v"
`include "decode.v"
```

```
`include "execute.v"
`include "write_back.v"
`include "memory.v"
`include "pc_update.v"

module seq;

reg clk;
reg [63:0] PC;

reg [2:0] stat; // check this

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 instr_valid;
wire imem_error;

wire [63:0] valA;
wire [63:0] valB;
wire [63:0] valE;
wire [63:0] valM;

wire cnd;
wire [63:0] newPC;

wire [63:0] reg0;
wire [63:0] reg1;
wire [63:0] reg2;
wire [63:0] reg3;
wire [63:0] reg4;
wire [63:0] reg5;
wire [63:0] reg6;
wire [63:0] reg7;
wire [63:0] reg8;
wire [63:0] reg9;
wire [63:0] reg10;
wire [63:0] reg11;
wire [63:0] reg12;
```

```
wire [63:0] reg13;
wire [63:0] reg14;

wire SF;
wire ZF;
wire OF;

fetch fetch1 (
    .clk(clk),
    .PC(PC),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP),
    .instr_valid(instr_valid),
    .imem_error(imem_error),
    .hlt(hlt)
);

decode decode1(
    .clk(clk),
    .icode(icode),
    .rA(rA),
    .rB(rB),
    /* .cnd(cnd), */
    .valA(valA),
    .valB(valB),
    .reg0(reg0),
    .reg1(reg1),
    .reg2(reg2),
    .reg3(reg3),
    .reg4(reg4),
    .reg5(reg5),
    .reg6(reg6),
    .reg7(reg7),
    .reg8(reg8),
    .reg9(reg9),
    .reg10(reg10),
    .reg11(reg11),
    .reg12(reg12),
    .reg13(reg13),
```

```
    .reg14(reg14)
);

execute execute1(
    .clk(clk),
    .icode(icode),
    .ifun(ifun),
    .valA(valA),
    .valB(valB),
    .valC(valC),
    .SF(SF),
    .ZF(ZF),
    .OF(OF),
    .cnd(cnd),
    .valE(valE)
);

memory memory1(
    .clk(clk),
    .icode(icode),
    .valA(valA),
    .valB(valB),
    .valE(valE),
    .valP(valP),
    .valM(valM),
);

write_back write_back1(
    .clk(clk),
    .cnd(cnd),
    .icode(icode),
    .rA(rA),
    .rB(rB),
    .valE(valE),
    .valM(valM),
    .reg0(reg0),
    .reg1(reg1),
    .reg2(reg2),
    .reg3(reg3),
    .reg4(reg4),
    .reg5(reg5),
    .reg6(reg6),
    .reg7(reg7),
```

```
    .reg8(reg8),
    .reg9(reg9),
    .reg10(reg10),
    .reg11(reg11),
    .reg12(reg12),
    .reg13(reg13),
    .reg14(reg14)
);

pc_update pc_update1(
    .clk(clk),
    .PC(PC),
    .cnd(cnd),
    .icode(icode),
    .valC(valC),
    .valP(valP),
    .valM(valM),
    .newPC(newPC)
);

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

    $monitor("clk=%d icode=%b ifun=%b rA=%b rB=%b valA=%d valB=%d valC=%d
valE=%d valM=%d instr_valid=%d mem_err=%d cnd=%d halt=%d 0=%d 1=%d 2=%d 3=%d
4=%d 5=%d 6=%d 7=%d 8=%d 9=%d 10=%d 11=%d 12=%d 13=%d
14=%d\n",clk,icode,ifun,rA,rB,valA,valB,valC,valE,valM,instr_valid,imem_erro
r,cnd,stat[2],reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,
reg12,reg13,reg14);

    clk = 0;

    PC = 64'd0;

    stat[0] = 1;
    stat[1] = 0;
    stat[2] = 0;
end

always #5 clk = ~clk;
```

```
always @ (*)
begin
    PC = newPC;

    if (hlt)
    begin
        stat[2] = hlt;
        stat[1] = 1'b0;
        stat[0] = 1'b0;
    end

    else if (instr_valid)
    begin
        stat[1] = instr_valid;
        stat[2] = 1'b0;
        stat[0] = 1'b0;
    end

    else
    begin
        stat[0] = 1;
        stat[1] = 0;
        stat[2] = 0;
    end

    if (stat[2] == 1'b1)
    begin
        $finish;
    end
end



endmodule
```