Project Report

By Rohan Gupta - 2020112022, Anjali Singh - 2020102004

Overview

The goal of the project is to build a processor architecture design based on Y86-64 ISA, by making various design modules using verilog.

Modules

The two modules we are going to build here are sequential and pipelining.

Sequential

The SEQ hardware structure includes the six basic stages: fetch, decode, execute, memory, write back and PC update. The notations icode, ifun indicate components of the instruction bytes; rA, rB indicate components of register specifier byte. The following are the various stages described with their verilog modules.

Fetch

- Here, the program counter register is used as an address.
- The instruction memory reads the bytes of an instruction.
- The PC incrementer is used to compute valP, which is again fetched and used in the fetch module, but we haven't used this module.

Verilog Modules

```
`timescale 1ns / 1ps

module fetch (clk, pc,
                icode, ifun, rA, rB,
                ivalid, ierror, valC, valP, out);

input clk;
input [63:0] pc;

output reg [3:0] icode;
output reg [3:0] ifun;

output reg [3:0] rA;
```

```verilog
output reg [3:0] rB;

output reg ivalid; // instruction validity
output reg ierror; // instruction error

output reg [63:0] valC;
output reg [63:0] valP;

output reg out;

reg [7:0] imem[0:1023]; // instruction memory
reg [0:79] instruct;

always@ (posedge clk) begin
    ierror = 0;
    if(pc > 1023)
    begin
        ierror = 1;
    end

    instruct = {
        mem[pc + 0], mem[pc + 1], mem[pc + 2], mem[pc + 3], mem[pc + 4],
        mem[pc + 5], mem[pc + 6], mem[pc + 7], mem[pc + 8], mem[pc + 9]
    };

    icode = instruct[0:3];
    ifun = instruct[4:7];
    valid = 1'b1; // when icode is invalid, instruction valid = 0

    if (icode == 4'0000) // when halt, icode = 0
    begin
        out = 1; // halt activated
        valP = pc + 64'd1;
    end

    else if (icode == 4'b0001) // for nop, icode = 1
    begin
        valP = pc + 64'd1;
    end
    else if (icode == 4'b0010) // for cmovxx, icode = 2
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valP = pc + 64'd2;
    end

    else if (icode == 4'b0011) // for irmovq, icode = 3
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valC = instruct[16:79];
        valP = pc + 64'd10;
    end

    else if (icode == 4'b0100) // for rmmovq, icode = 4
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valC = instruct[16:79];
        valP = pc + 64'd10;
    end

    else if (icode == 4'b0101) // for mrmovq, icode = 5
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valC = instruct[16:79];
        valP = pc + 64'd10;
    end

    else if (icode == 4'b0110) // for OPq, icode = 6
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valP = pc + 64'd2;
    end

    else if (icode == 4'b0111) // for jxx, icode = 7
    begin
```

```
        valC = instruct[8:71];
        valP = pc + 64'd9;
    end

    else if (icode == 4'b1000) // for call, icode = 8
    begin
        valC = instruct[8:71];
        valP = pc + 64'd9;
    end

    else if (icode == 4'b1001) // for ret, icode = 9
    begin
        valP = pc + 64'd1;
    end

    else if (icode == 4'b1010) // for pushq, icode = 10
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valP = pc + 64'd2;
    end

    else if (icode == 4'b1011) // for popq, icode = 11
    begin
        rA = instruct[8:11];
        rB = instruct[12:15];
        valP = pc + 64'd2;
    end

   else
   begin
       valid = 1'b0; // icode is invalid
   end

end

endmodule
```

Decode/Write Back

- It has two read ports, A and B, using which valA and valB are read simultaneously.
- It also has two write ports, E and M.

Verilog Modules

```
`timescale 1ns/1ps


module registerfile(clk,dstE,dstM,srcA,srcB,valE,valM,valA,valB);

input [3:0]dstE;
input [3:0]dstM;
input [3:0]srcA;
input [3:0]srcB;
input clk;
input [63:0] valE;
input [63:0] valM;

output reg[63:0] valA;
output reg[63:0] valB;


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

parameter rax = 4'h0 ;
parameter rcx = 4'h1 ;
parameter rdx = 4'h2 ;
parameter rbx = 4'h3 ;
parameter rsp = 4'h4 ;
parameter rbp = 4'h5 ;
parameter rsi = 4'h6 ;
parameter rdi = 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 rnone = 4'hF ;

initial
begin
register_file[4'h0] <= 64'h0;
register_file[4'h1] <= 64'h0;
register_file[4'h2] <= 64'h0;
register_file[4'h3] <= 64'h0;
register_file[4'h4] <= 64'h0;
register_file[4'h5] <= 64'h0;
register_file[4'h6] <= 64'h0;
register_file[4'h7] <= 64'h0;
register_file[4'h8] <= 64'h0;
register_file[4'h9] <= 64'h0;
register_file[4'hA] <= 64'h0;
register_file[4'hB] <= 64'h0;
register_file[4'hC] <= 64'h0;
register_file[4'hD] <= 64'h0;
register_file[4'hE] <= 64'h0;
end


always @(*) begin

    if (srcA != rnone) begin
        valA <= register_file[srcA];

    end

    if (srcB != rnone) begin
        valB <= register_file[srcB];

    end

end

always @(negedge(clk)) begin

    if(dstE != rnone) begin
        register_file[dstE] <= valE;
    end

    if(dstM != rnone) begin
        register_file[dstM] <= valM;
    end

end


endmodule




module srcA_logic(icode,rA,srcA);

input[3:0]icode;
input[3:0]rA;

output reg[3:0]srcA;

parameter rsp = 4'h4 ;
parameter rnone = 4'hF ;


always @(icode,rA) begin

    case (icode)
    4'h2, 4'h3, 4'h6, 4'hA:
    begin
        srcA<= rA;
    end
    4'h9,4'hB:
    begin
        srcA <= rsp;
    end
        default: srcA <= rnone;
    endcase

end
```

```
endmodule


module srcB_logic(icode,rB,srcB);


input[3:0]icode;
input[3:0]rB;

output reg[3:0]srcB;

parameter rsp = 4'h4 ;
parameter rnone = 4'hF ;


always @(icode,rB) begin

    case (icode)
    4'h6, 4'h4, 4'h5:
    begin
        srcB<= rB;
    end
    4'hA, 4'hB, 4'h8, 4'h9:
    begin
        srcB <= rsp;
    end
        default: srcB <= rnone;
    endcase

end

endmodule


module dstE_logic(icode,ifun,rB,cnd,dstE);


input[3:0]icode;
input [3:0] ifun;
```

```
input[3:0]rB;
input cnd;
output reg[3:0]dstE;

parameter rsp = 4'h4 ;
parameter rnone = 4'hF ;


always @(icode,ifun,rB,cnd) begin

    case (icode)

    4'h2: begin
                if(ifun == 4'h0)
                    dstE <= rB;
                else
                    begin
                        if(cnd==1'b1)
                            dstE <= rB;
                        else
                            dstE <= rnone;
                    end
    end

    4'h3, 4'h6:
        dstE <= rB;
    4'hA, 4'hB, 4'h8, 4'h9:
        dstE <= rsp;
        default: dstE <= rnone;
    endcase

end

endmodule

module dstM_logic(icode,rA,dstM);


input[3:0]icode;
input[3:0]rA;
```

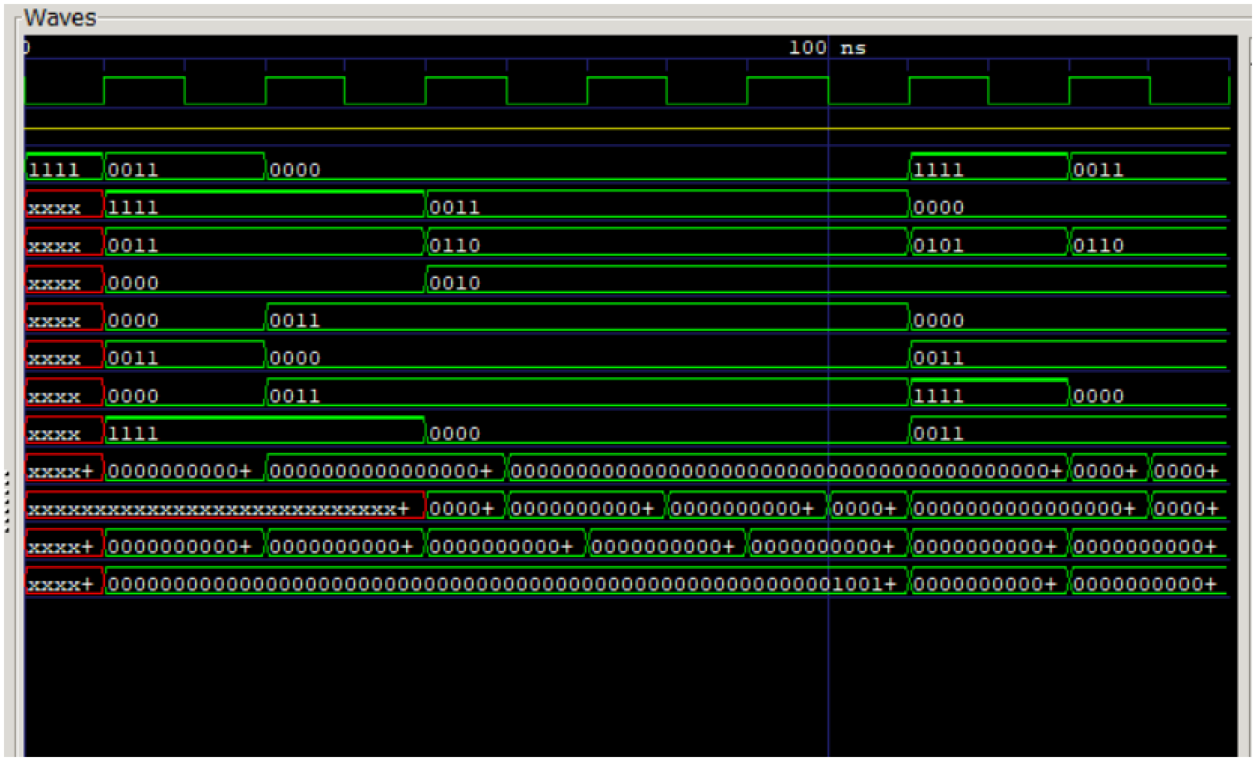
```
output reg[3:0]dstM;


parameter rsp = 4'h4 ;
parameter rnone = 4'hF ;

always @(icode,rA) begin
     case (icode)
     4'h5, 4'h6:
        dstM <= rA;
         default: dstM <= rnone;
     endcase

end
endmodule
```

Testbench

```
`timescale 1ns / 1ps

module decode_test;
    integer k;

    reg [63:0] valE;
    reg [3:0] rA;
    reg [3:0] rB;
    reg [3:0] icode;
    wire [3:0] dstE;
    wire [3:0] dstM;
    wire [3:0] srcA;
    wire [3:0] srcB;
    wire [63:0] valA;
    wire [63:0] valB;
    reg [63:0] valM;
    wire cnd;
    reg [3:0] ifun;
    reg clk;
```

```verilog
    registerfile reg_f(.clk(clk),.dstE(dstE), .dstM(dstM), .srcA(srcA) ,
.srcB(srcB) , .valA(valA) , .valB(valB) , .valM(valM), .valE(valE));
    srcA_logic sA_l(.icode(icode),.rA(rA),.srcA(srcA));
    srcB_logic sB_l(.icode(icode),.rB(rB),.srcB(srcB));
    dstE_logic
dE_l(.icode(icode),.rB(rB),.cnd(cnd),.ifun(ifun),.dstE(dstE));
    dstM_logic dM_l(.icode(icode),.rA(rA),.dstM(dstM));




    initial begin
    clk <= 1'b0;
        $dumpfile("decode_test.vcd");
        $dumpvars(0,decode_test);
        #10;
        // irmovq
        icode <= 4'h3;
        ifun  <= 4'h0;
        rA <= 4'h0;
        rB <= 4'h3;
        valE <= 64'd525;
        valM <= 64'd300;
        #20;

        // moved 64'd525 to register rbx;

        icode <= 4'h3;
        ifun  <= 4'h0;
        rA <= 4'h3;
        rB <= 4'h0;
        valE <= 64'd300;
        valM <= 64'd300;
        #20;

        // moved 64'd300 to register rax;


        icode <= 4'h6;
        ifun  <= 4'h2;
        rA <= 4'h3;
        rB <= 4'h0;
        valE <= 64'd251;
        valM <= 64'd300;
        #20;

        icode <= 4'h6;
        ifun  <= 4'h2;
        rA <= 4'h3;
        rB <= 4'h0;
        valE <= 64'd252;
        valM <= 64'd300;
        #20;
        icode <= 4'h6;
        ifun  <= 4'h2;
        rA <= 4'h3;
        rB <= 4'h0;
        valE <= 64'd253;
        valM <= 64'd300;
        #20;

        //check memory to reg
        icode <= 4'h5;
        ifun  <= 4'h2;
        rA <= 4'h0;
        rB <= 4'h3;
        valE <= 64'd999;
        valM <= 64'd999;

        #20;
        icode <= 4'h6;
        ifun  <= 4'h2;
        rA <= 4'h0;
        rB <= 4'h3;
        valE <= 64'd253;
        valM <= 64'd300;

        #20 $finish;

    end
    always #10 clk = ~clk;


endmodule
```

GTKWave Output



Execute

- It includes the ALU operations, which we made previously.
- The operations and logic used here are AND, SUB, ADD and XOR.

Verilog Modules

Memory

- It is used to either read or write program data.
- There are four blocks: two of which is are control blocks used to generate values for memory address and memory input data and the other two blocks generate the control signals on whether to read or write the program data.

Verilog Modules

```
`timescale 1ns / 1ps

module RAM(memaddr,memdata,read,write,valM,dmemerror);

input[63:0] memaddr;
input[63:0] memdata;
input read;
input write;

reg [63:0] memory[8191:0];

output reg[63:0] valM;
output reg dmemerror;


always @(write,read,memdata,memaddr) begin

    if(read && !write) begin
        valM <= memory[memaddr];
    end

    if(write && !read) begin
        memory[memaddr] <= valM;
    end

    if (memaddr >= 64'd258) begin
        dmemerror <= 1'b1;
    end
    else begin
        dmemerror <= 1'b0;

    end
end

endmodule

module MEM_addr(icode,valE,valA,memaddr);

input[3:0] icode;
input [63:0] valE;
```

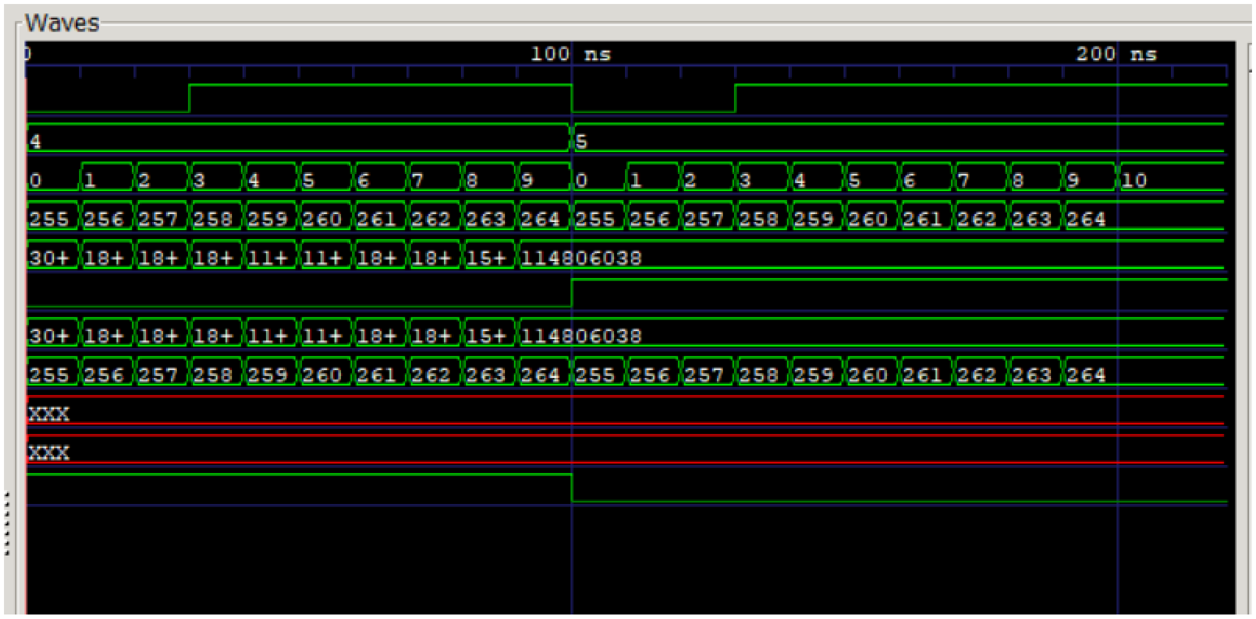
```
input[63:0] valA;

output reg[63:0] memaddr;

always @(icode,valE,valA) begin


    case (icode)
        4'h4, 4'h5, 4'hA, 4'h8:
            memaddr <= valE;
        4'h9, 4'hB:
            memaddr <= valA;
    endcase

end

endmodule

module MEM_data(icode, valA, valP, memdata);

input[3:0] icode;
input [63:0] valA;
input[63:0] valP;

output reg[63:0] memdata;

always @(icode,valA,valP) begin

    case (icode)
        4'h4, 4'hA:
            memdata <= valA;
        4'h8:
            memdata<= valP;

    endcase
end

endmodule

module MEM_read (icode,read);
```

```
input [3:0] icode;
output reg read;

always @(icode) begin


    case (icode)
        4'h5, 4'hB, 4'h9:
            read <= 1'b1;
        default: read <= 1'b0;
    endcase
end
endmodule

module MEM_write (icode,write);

input [3:0] icode;
output reg write;

always @(icode) begin


    case (icode)
        4'h4, 4'hA, 4'h8:
            write <= 1'b1;

        default: write <= 1'b0;
    endcase
end
endmodule
```

Testbench

```
`timescale 1ns / 1ps
module ram_test;

    reg [3:0] icode;
    wire [63:0] memaddr;
    wire [63:0] memdata;
    reg [63:0] valE;
    reg [63:0] valA;
    reg [63:0] valP;
    wire [63:0] valM;
    wire read;
    wire write;

    RAM
ram1(.memaddr(memaddr),.memdata(memdata),.read(read),.write(write),.valM(v
alM),.dmemerror(dmemerror));
    MEM_addr Ma(.icode(icode),.valA(valA),.valE(valE),.memaddr(memaddr));
    MEM_read Mr(.icode(icode),.read(read));
    MEM_write Mw(.icode(icode),.write(write));
    MEM_data Md(.icode(icode),.valA(valA),.valP(valP),.memdata(memdata));


    integer k;
    parameter base_addr = 64'hFF;


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


    for(k=0;k<10;k++)
    begin
        icode <= 4'h4;
        valE  <= k + base_addr;
        valA  <= k+ $random;
        #10;
    end

    for(k=0;k<10;k++)
    begin
        icode <= 4'h5;
        valE  <= k + base_addr;
        #10;
    end

    #20 $finish;
    end



endmodule
```

GTKWave Output



```
4                                                       5
0    1    2    3    4    5    6    7    8    9    0    1    2    3    4    5    6    7    8    9    10
255  256  257  258  259  260  261  262  263  264  255  256  257  258  259  260  261  262  263  264
30+  18+  18+  18+  11+  11+  18+  18+  15+  114806038
30+  18+  18+  18+  11+  11+  18+  18+  15+  114806038
255  256  257  258  259  260  261  262  263  264  255  256  257  258  259  260  261  262  263  264
XXX
XXX
```

PC Update

- The new value of PC computed is either valP, valC or valM.
- The program counter PC is the only SEQ register which is based on the clock.

Verilog Modules

```
`timescale 1ns / 1ps

module pc_update(clk, pc, icode, valC, valM, valP, pc_updated, c);

input clk, c;
input [63:0] valC; // destination address specified by a call or jump
instructions; for call purpose
input [63:0] valM; // return address read from memory; for ret purpose
input [63:0] valP; // address of next instruction; for all other
instructions other than call and ret
                   // used for the computation of a specific value
input [63:0] pc;
input [3:0] icode;

output reg [63:0] pc_updated;

always@(*) begin
    if (icode == 4'b0111) //icode = 7
    begin
        if(c == 1'b1)
        begin
            pc_updated = valC;
        end
        else
            pc_updated = valP;
        end
    end

    else if (icode == 4'b1000) // icode = 8; for call
    begin
        pc_updated = valC;
    end

    else if (icode == 4'b1001) // icode = 9; for ret
    begin
        pc_updated = valM;
    end

    else
    begin
        pc_updated = valP;
    end
end
endmodule
```