Semester Project On

"Computer Organization and Architecture"

Submitted By

Group 11

Of Btech IT (2nd sem)

IIT2018040 (Manthan Surkar)

IIT2018042(Aman Joshi)

IIT2018043(Abhishek Manish Singh)

IIT2018051(Avishek)

IIT2018066(Laxman Goliya)

Academic Session 2018-19

Submitted to Dr. Bibhas Ghoshal

Indian Institiute of Information TechnologyAllahabad

Topic - Design of a processor in verilog

Abstract

The following project contains the design and implementation of an 8 – bit processor in the hardware description language Verilog. The motivation for the project was the semester project assignment given to us for completion. This document contains the code, data path and the explanation for the various components of CPU done in accordance with the guidelines given in the question. The practical implementation and designing of a data path lead to a better understanding of the interconnections between the components of a CPU. It gives us the bigger or macro level image rather than individual components or micro level aspects. The whole is bigger than the sum of it parts. We have understood this saying during our project time.

1.0. INTRODUCTION

"Processor is the brain of our computers."

The following implementation is that of an 8-bit processor which has 16-bit instructions. Following is the list of components we used in our processor:

1. Program Counter (PC): A mod 1 counter of size 8bit.
2. Instruction Memory (IMEM): A memory with 16-bit rows, 256 in number. 1 port read.
3. Instruction Register (IR): A 16-bit register.
4. Control Unit (CU): A combinational signal generating block.
5. Register file: Contains 8 registers each of size 8-bit. 2 port read; 1 port write simultaneously.
6. Arithmetic and Logical Unit (ALU): Performs arithmetic operations and generates the result with flags.
7. Data Memory (DMEM): A memory with 8-bit rows, 256 in number.
8. Other implementation specific logics.

1.1. Instruction word specification

The instruction word has been specified exactly as given in the question. Following is the extract from question containing the specification of the instructions

The CPU uses 16-bit instructions that are stored in the IMEM. There are 3 types of instructions:

Type	*15-12*	*11-9*	*8-6*	*5-3*	*2-0*
R (register)	*Opcode*	*Rd*	*Ra*	*Rb*	*Func*
I (immediate)	*Opcode*	*Imm[5:3]*	*Ra*	*Rb*	*Imm[2:0]*
J (jump)	*Opcode [11:7]*	*dont care*		*Addr [6:0]*	

Assume that the register r0 always stores the numeric value 0 (to make comparisons easy), and
Ra, Rb can refer to any register.

The CPU you have to design will have the following instructions:

Instruction	*Opcode/Func*	*Type*	*Operation*
add rd, ra, rb	*0000 / 000*	*R*	*rd <- ra + rb*
sub rd, ra, rb	*0000 / 010*	*R*	*rd <- ra + rb*
and rd, ra, rb	*0000 / 100*	*R*	*rd <- ra AND rb*
or rd, ra, rb	*0000 / 101*	*R*	*rd <- ra OR rb*
addi rb, ra, imm	*0100*	*I*	*rb <- ra + imm*
lw rb, imm(ra)	*1011*	*I*	*rb <- DMEM[ra + imm]*
sw rb, imm(ra)	*1111*	*I*	*DMEM[ra + imm] <- rb*
beq rb, ra, imm	*1000*	*I*	*if (ra == rb) pc <- pc + imm*
j addr	*0010*	*J*	*pc <- addr X 2*

Notes:

1. The imm operand is split into two parts to make the instruction easier to decode - this is just for simplicity.

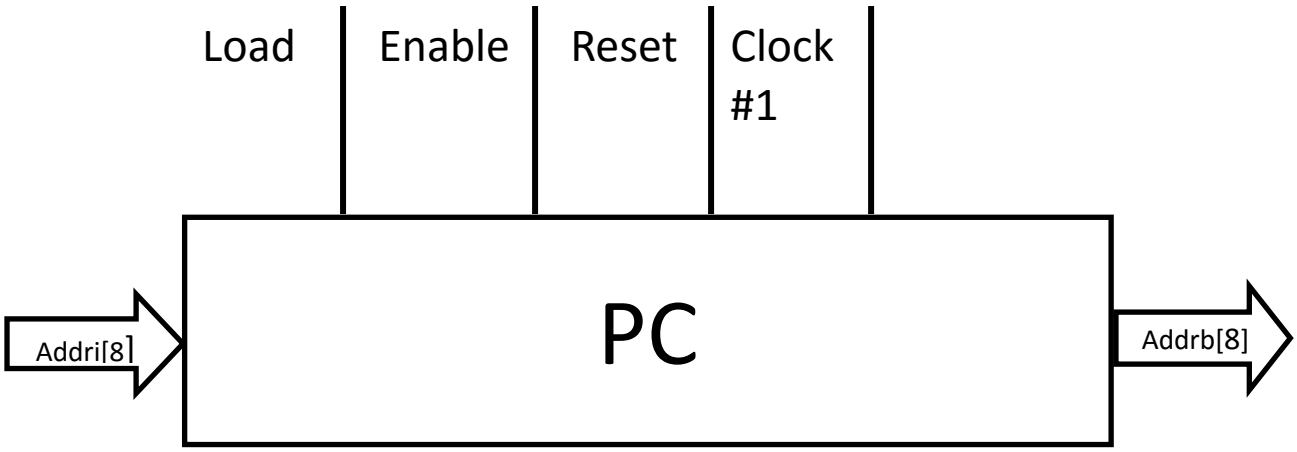
2. For the J instruction, the address is 7 bits which is then extended to 8 bits. This means you can only jump to even numbered addresses. Why? Simplicity.

2. Components of Processor

The following section contains the diagrams, information and codes of different components of processor.

2.1 Program Counter (PC)

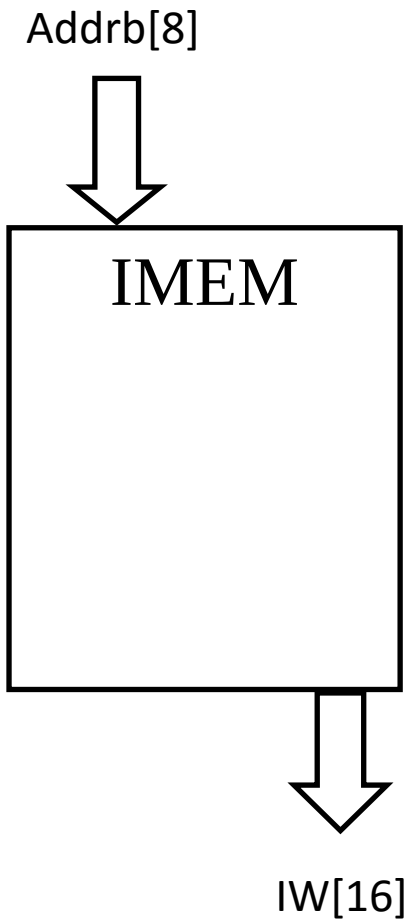
It is an 8-bit register. Under normal operations, will always increment by 1 on every clock cycle, to access the next instruction. In case of a JUMP or BRANCH type of instruction, will go to the value specified by the output of the adder or the immediate operand in the instruction.




```
module pc (addri,load,out, enable, clk, reset);
output [7:0] out; input [7:0]addri;
input enable, clk, reset, load;
reg [7:0] out;

initial out = 0;

always @(posedge clk)
        if (load) out <= addri;
        else if (reset) begin
                out <= 8'b0 ;
        end
        else if (enable) begin
                out <= out + 1;
        end
endmodule
```

2.2 Instruction Memory [IMEM]

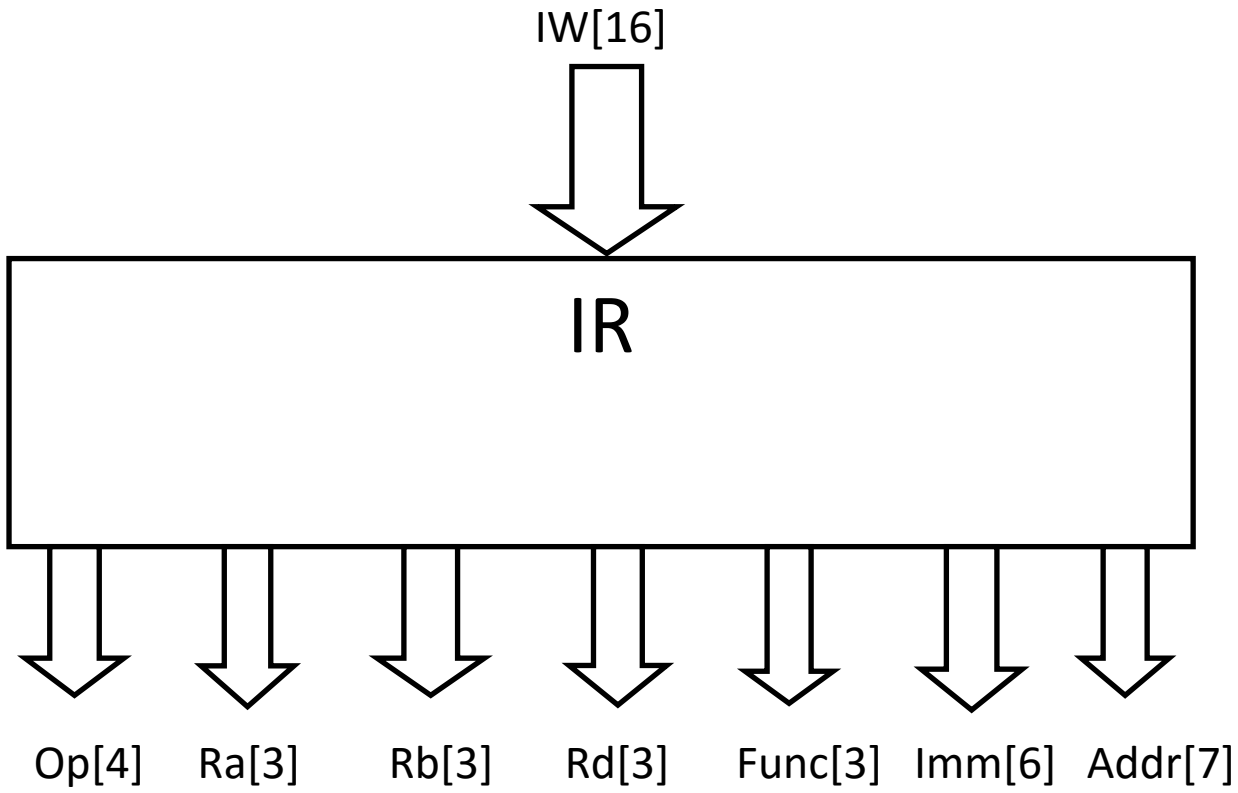
This is a combinational unit - it takes just the address bus (8-bit value) as input, and gives out a 16-bit value that is the instruction to be decoded. The address is provided by the Program Counter (PC). The program is assumed to start from 0x00.




```
module imem (input [7:0] Addr, output [15:0] iw);
reg [15:0] RAM [255:0];

initial begin
$readmemb("memfile.dat",RAM);
end
assign iw = RAM[Addr]; // word aligned
endmodule
```

2.3 Instruction Register [IR]

It is a 16-bit register. Takes input of the word given by IMEM, and gives various components of instruction by assigning appropriate bits.




```
module ir(iw,op,Ra,Rb,Rd,func,imm,addr);
input [15:0] iw; output [3:0]op;
output [2:0] Ra, Rb, Rd, func; output [5:0] imm;
output  [6:0] addr;
assign op = iw[15:12];
assign Ra = iw[8:6];
assign Rb = iw[5:3];
assign Rd = iw[11:9];
assign func = iw[2:0];
assign imm[5:3] = iw[11:9];
assign imm[2:0] = iw[2:0];
```

```
assign addr = iw[6:0];
endmodule
```

2.4 Control Unit [CU]

The unit that actually makes the entire processor work as expected. The input to this is the instruction word, and the output is a set of control signals that decide, for example, whether the register file is to updated, what operation is to be done by the ALU, whether memory read or write is required etc. One way to implement this is to make it a combinational block that will generate the following signals:

MemToReg: does data being written into the register file come from memory or from output of the ALU. 1 if data is from memory else 0.

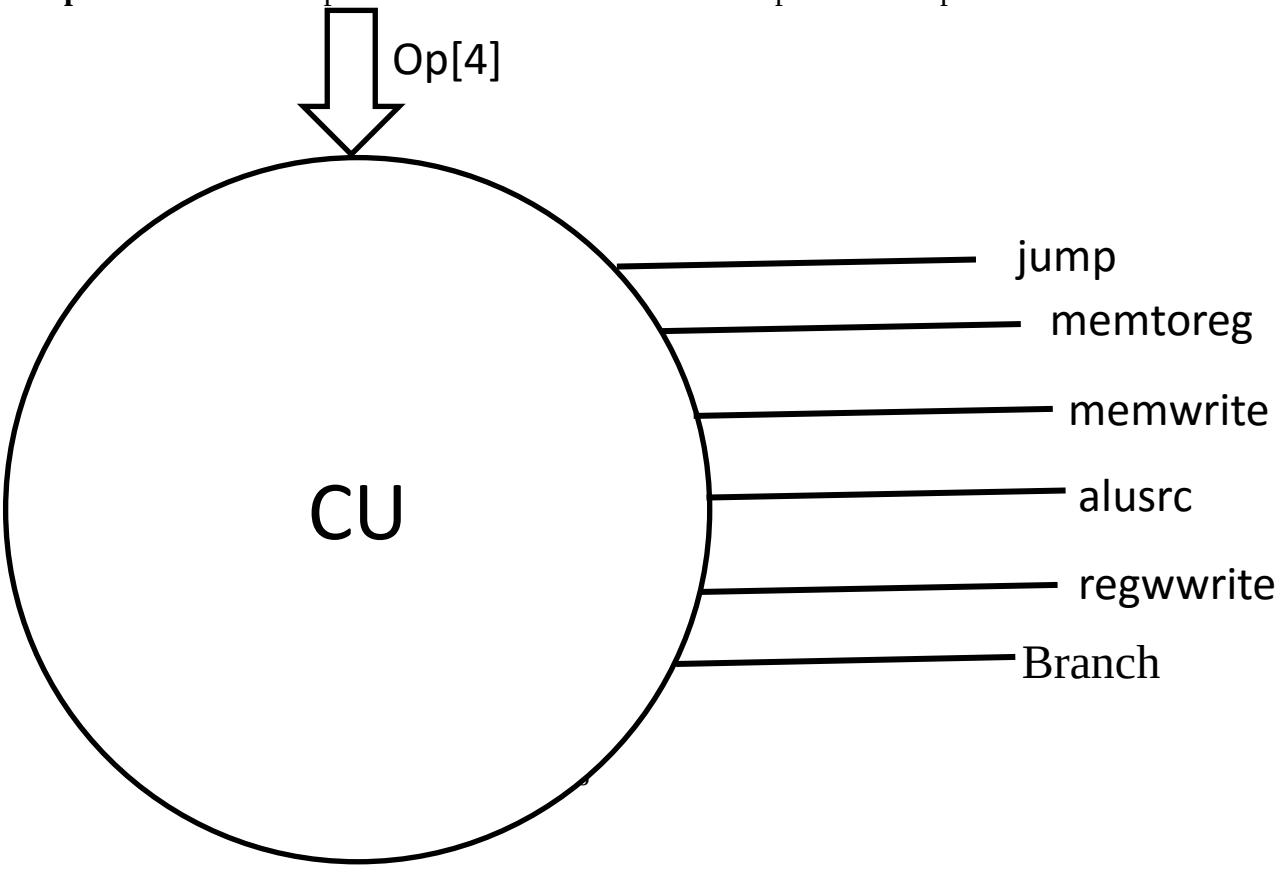
MemWrite: is the present operation going to write into memory?

RegWrite: is an entry in the register file (pointed to by rd) supposed to get an updated value in this instruction?

ALUSrc: is this an immediate operation, or a register operation?

Branch: If the current instruction is a branch, then use this signal as select for a multiplexer to feed the PC.

Jump: Control the PC input MUX and also the immediate operand for input to PC.

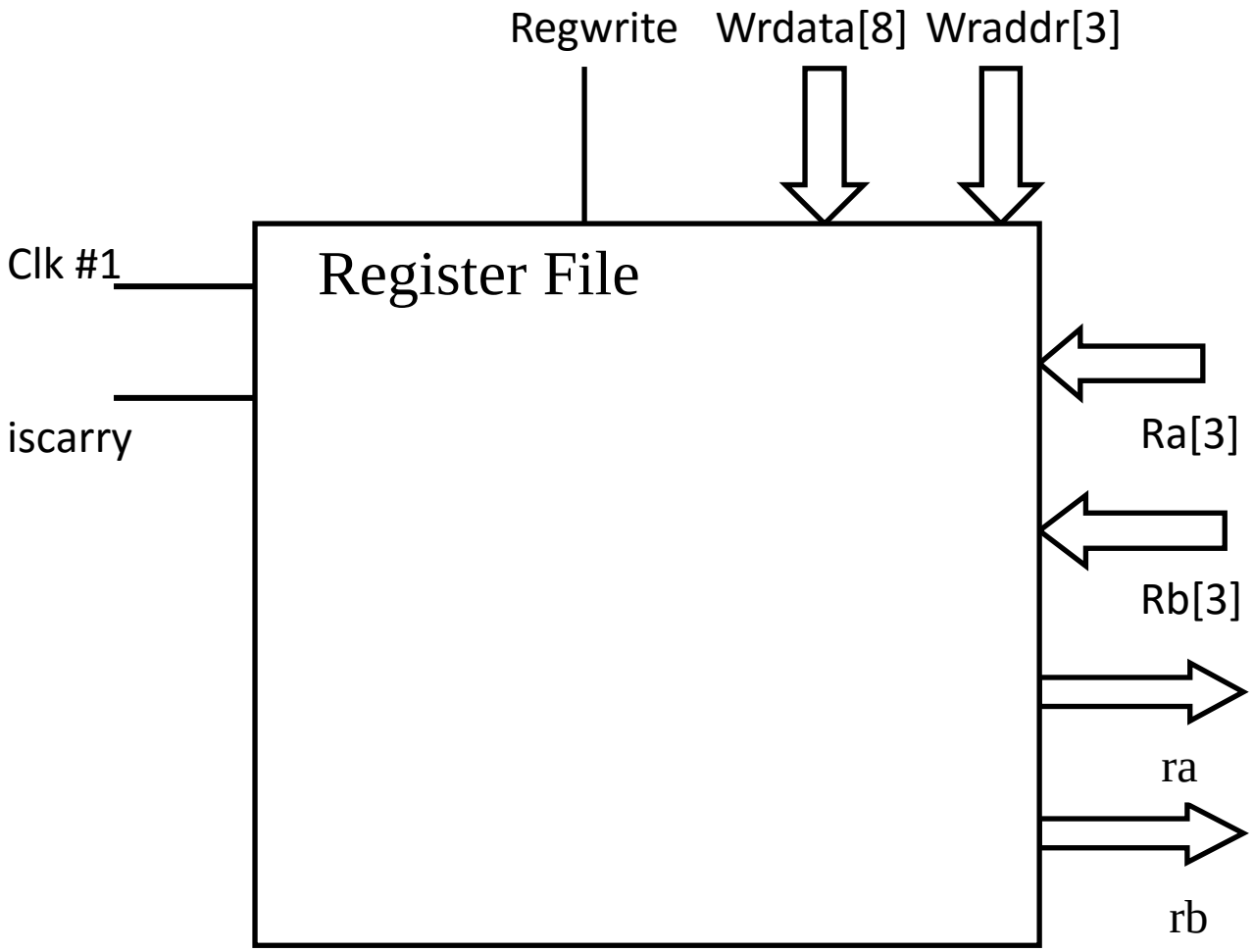


I1	I2	I3	I4	INSTRUCTION	J	Mr	Mw	Rw	Alusrc	bran
0	0	0	0	regtype	0	1	0	1	0	0
0	1	0	0	addi	0	1	0	1	1	0
1	0	1	1	lw	0	0	0	1	1	0
1	1	1	1	sw	0	x	1	0	1	0
1	0	0	0	beq	0	x	0	0	1	1
0	0	1	0	jump	1	x	0	0	x	0

```
module CU(op,jump,memtoreg,memwrite,regwrite,ALUsrc,branch);
input [3:0]op;
output jump,memtoreg,memwrite,regwrite,ALUsrc,branch;
assign jump=op[1]&(~op[0]);
assign memtoreg=op[1];
assign memwrite=op[3]&op[2];
assign regwrite=(~op[3]&~op[1])|(~op[2]&op[0]);
assign ALUsrc=op[2]|op[3];
assign branch=op[3]&(~op[1]);
endmodule
```

2.4 Register file

This is a set of 8 registers, each storing an 8-bit value. There are 2 output values ra and rb, whose values are selected based on the instruction word, as in the instruction definitions given above. There is one port by which data can be written into one of the registers, but an enable signal is required to control whether or not to update the value.



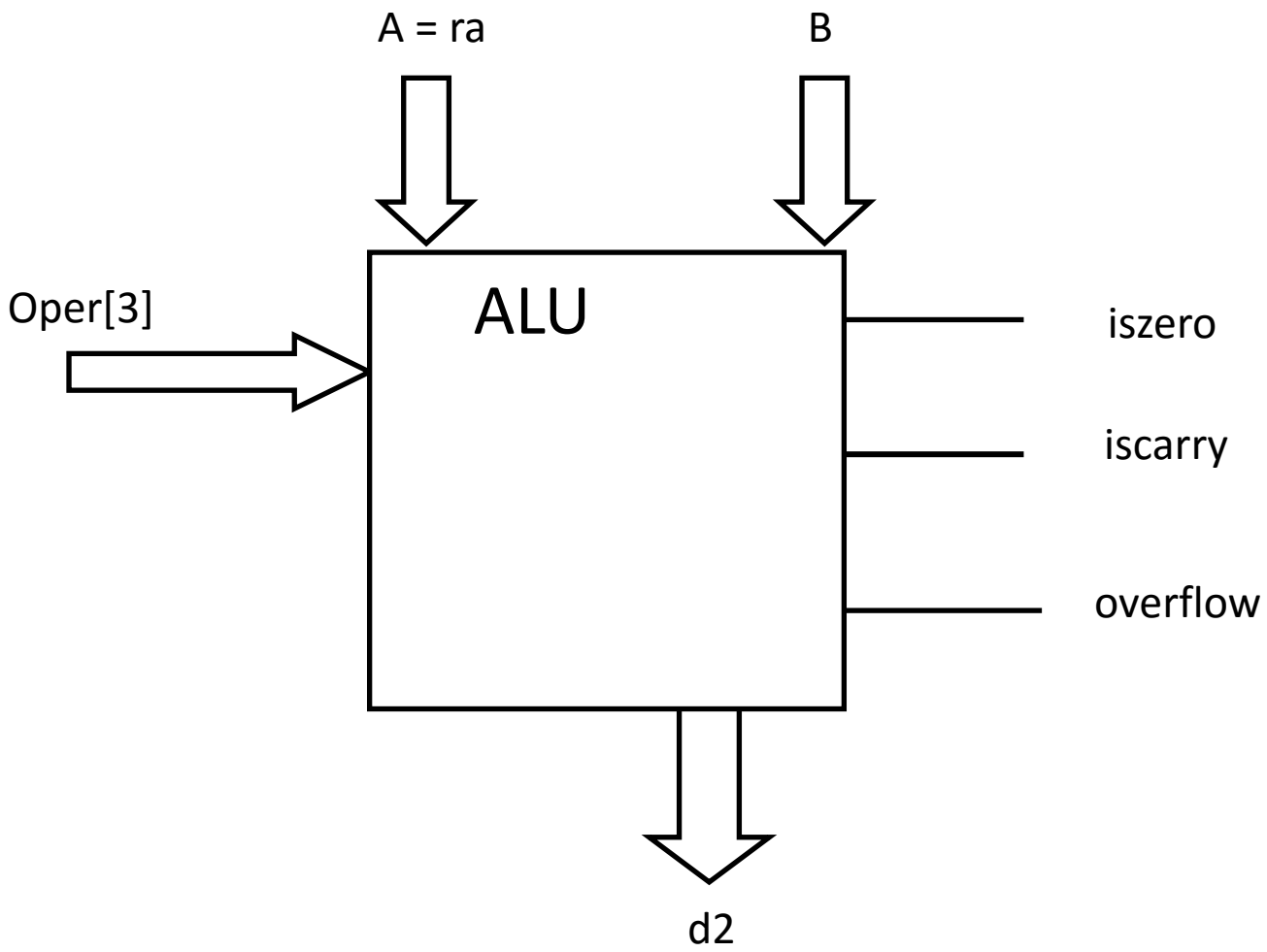
The 0^{th} register is always zero whereas 7^{th} register is used for status information. In the given time frame, we could implement only the carry status update. If there is a carry generated then $7[0] turns 1 else 0.

```
module regfile
 (input clk,
  input regwrite,
  input [2:0] wrAddr,
  input [7:0] wrData,
  input [2:0] rdAddrA,
  output [7:0] rdDataA,
  input [2:0] rdAddrB,
  output [7:0] rdDataB,
  input iscarry
```

```
  );

  reg [7:0] regs[7:0]; integer i;

  initial begin
  for(i = 0; i <8; i=i+1) regs[i] = 0;
  end

  assign rdDataA = regs[rdAddrA];
  assign rdDataB = regs[rdAddrB];

  always @(posedge clk) begin
    if (regwrite)begin regs[wrAddr] = wrData;
          $display("wraddr = %d, w = %b",wrAddr,wrData);
                if  (iscarry) begin regs[7]= 1;
                $display("carry = %b",regs[7]);
                end
                else regs[7] = 0;
          end
                $display("\n--------%b%b----------\n",regs[5],regs[4]);
  end
endmodule
```

2.5 Arithmetic and Logic Unit

The core of the processor - all the actual computations are performed here. As shown in the instruction set, operations such as addition, subtraction and logical operations are all done in this unit. Also, the output of the ALU is used as the address for certain memory related operations.

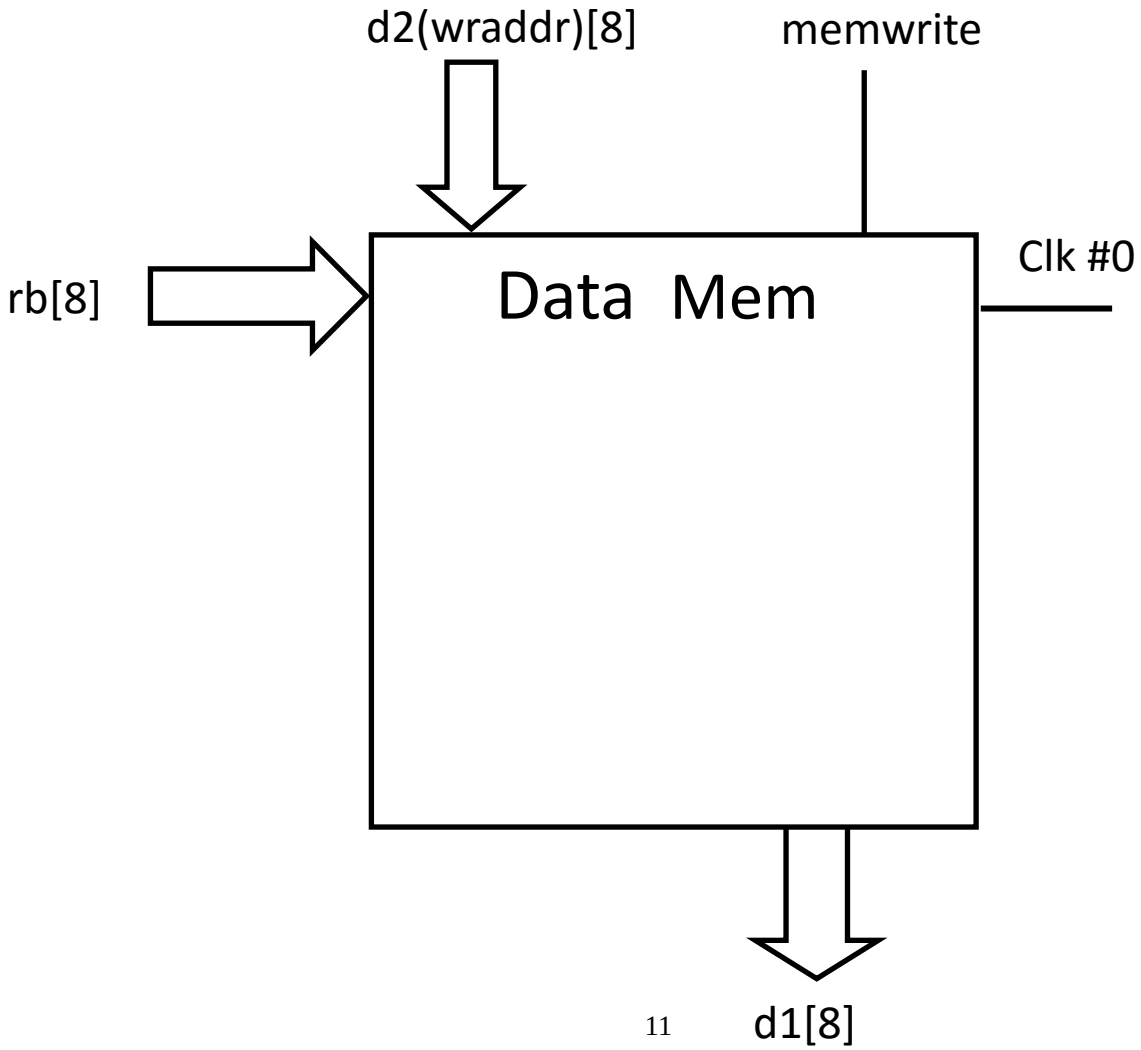



```
module ALU(opr,A,B,result,iszero,overflow,iscarry);
input [2:0] opr; input [7:0]A,B; output iscarry;
output [7:0]result; output iszero,overflow;
reg [7:0] result; reg iszero,overflow,iscarry;
always @(*) begin
if(opr==0)
 result=A+B;
else if(opr==2)
 result=A-B;
else if(opr==4)
 result=A&B;
else if(opr==5)
 result=A|B;
if(result == 0) iszero = 1;
```

```
else iszero = 0;
if((A[7] == B[7]) && A[7]!=result[7]) overflow = 1;
else overflow = 0;
if((~result[7]&A[7])|(A[7]&B[7])|(~result[7]&B[7])) iscarry = 1;
else iscarry = 0;
end
endmodule
```

2.6 Data Memory(DMEM)

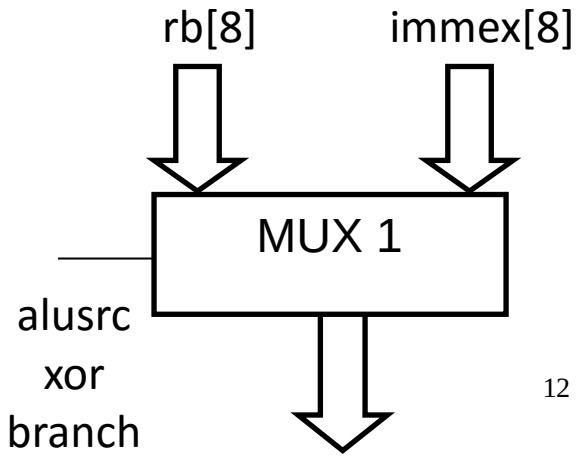
This needs to be a sequential / clocked unit. At any given cycle, you are either reading from it or writing to it, with the address given by the address bus. In case of a lw (load-word) or sw (storeword) instruction, the address is the output of the ALU. The data output of the DMEM will always go into the register file, where it gets stored into a register selected by rd.



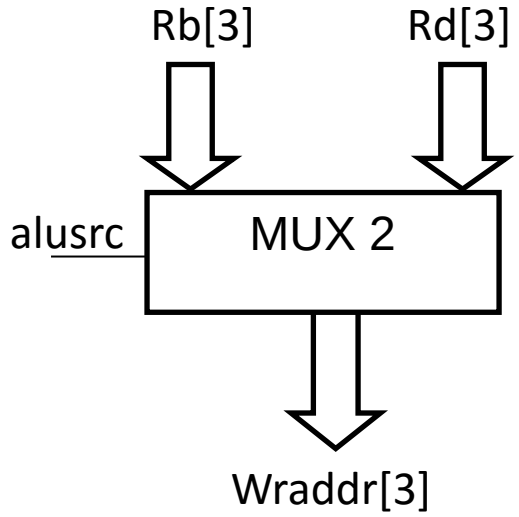
```
module datamemory(clk,memWrite,adr,writedata,outdata);
input clk,memWrite;
input [7:0] adr, writedata;
reg [7:0] dmem [255:0] ;
output reg [7:0]outdata;

initial begin dmem[0] = 31; dmem[1] = 31; end

always@(posedge clk)
begin

        if(memWrite==1) begin
                dmem[adr]= writedata;
                $display("dmemadr = %d; wdat = %d",adr,dmem[adr]);
        end
   else
        outdata = dmem[adr];
end
endmodule
```

2.7 Mux

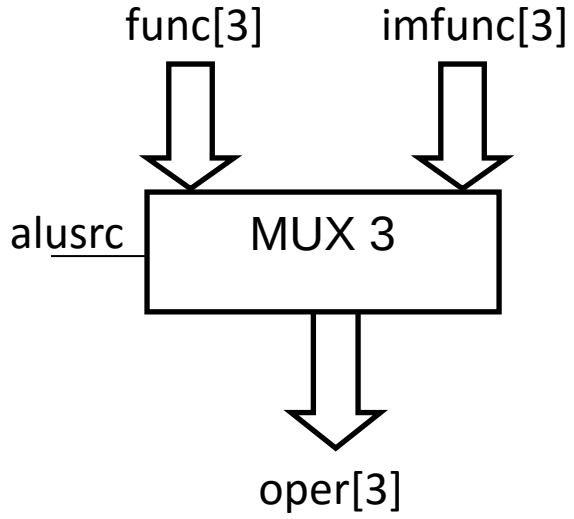
Mux 1: Selects between the second operand of ALU based on alusrc and branch.
If both are 1 or both are 0 then rb else immex(extended immediate).



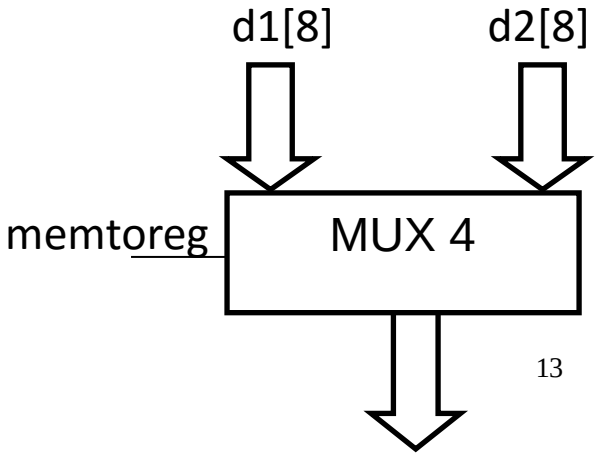
Mux 2: Selects where the data needs to be written in register file. For immediate type it is rb whereas for R type it is rd.



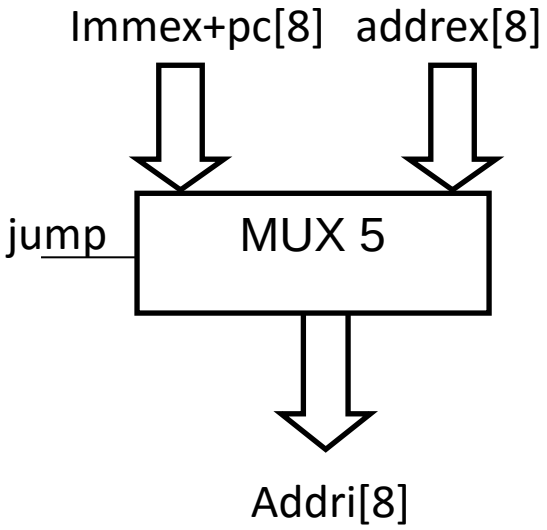
Mux 3: Selects the operation for ALU. If immediate type immfunc is allowed (which is the output of immediate function generator) else the func bit from IR.



Mux 4: Selects the data to be written in register file. If memtoreg is 1 then d1(from memory) is allowed and otherwise d2(from AlU) is allowed.



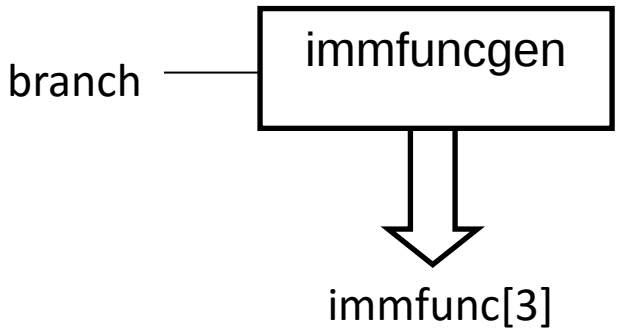
Mux 5: Selects the address to be provided to the PC.



2.8 Other Logics

Immediate function generator

Generates the operation for immediate function. In immediate function only branch requires subtract else all are addition. Therefore, output is 010 else it is 000.

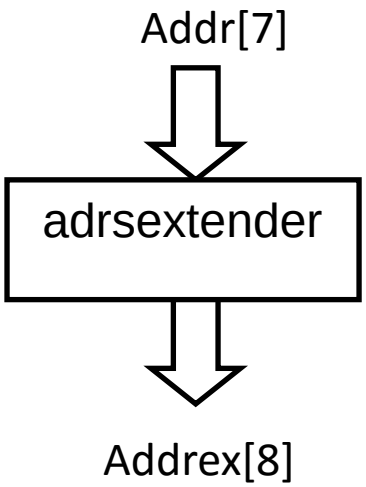



```
module dc1(b,y);
input b;
output [2:0]y;

reg [2:0]y;
```

```
always @(*)
begin
if(~b)   y=3'b000;
else if(b)  y=3'b010;
end

endmodule
```

Adrsextender

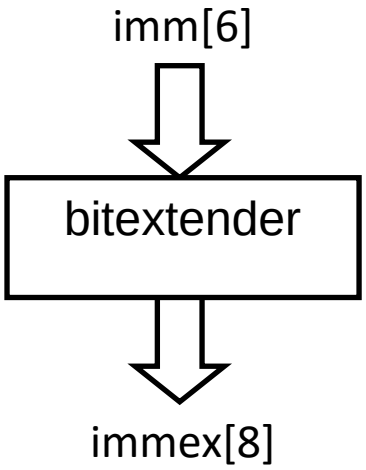
The 7-bit address is extended to 8-bit by logically shifting left by one bit.




```
module adrsExtend(ads,extendedads);

input [6:0] ads;
output [7:0] extendedads;

 assign extendedads[7:1]=ads[6:0];
 assign extendedads[0]=0;



endmodule
```

Immextender

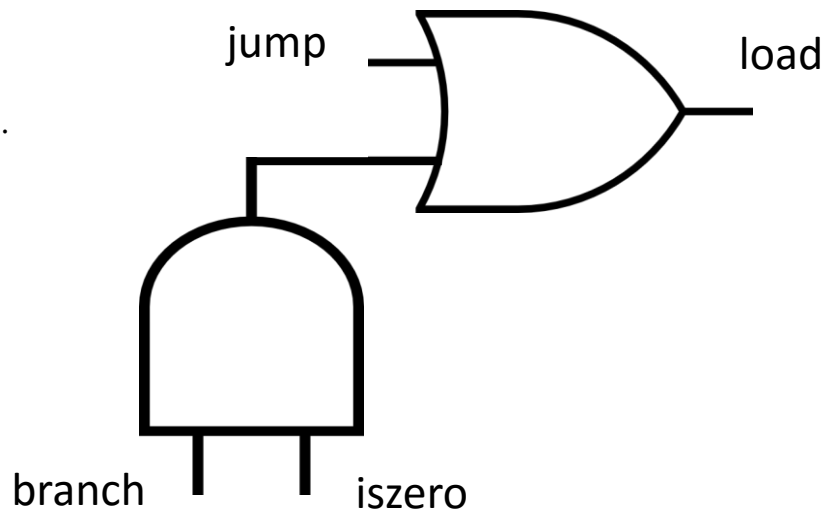
Arithmetically extends the 6-bit immediate data.




```
module bitextender(data,extendedData);

input [5:0] data;
output [7:0] extendedData;

 assign extendedData[5:0]=data[5:0];
 assign extendedData[6]=data[5];
 assign extendedData[7]=data[5];



endmodule
```

The load logic

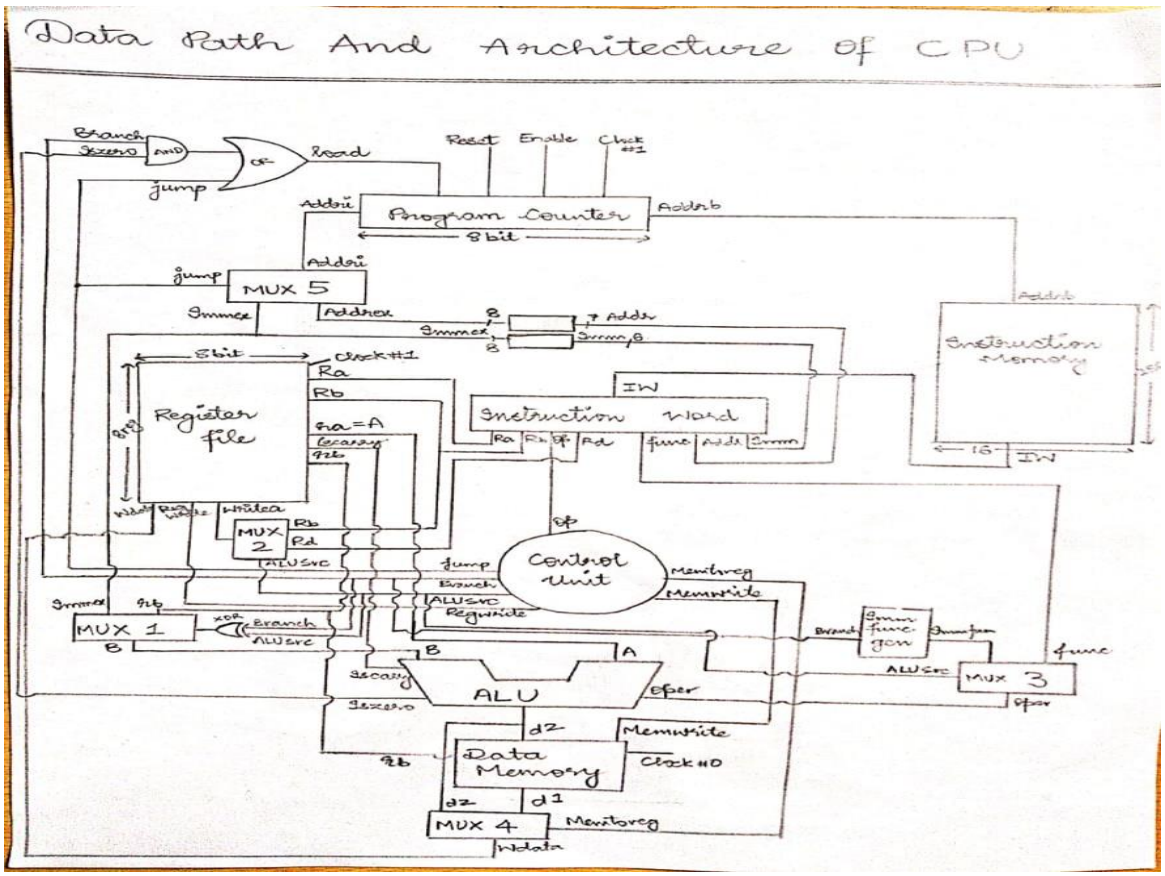
Decides the load bit input to PC.



3. Combining and making the CPU

The Data path

Just connecting the corresponding inputs and outputs in individual components gives us the data path. There are 2 clocks involved. The clock fed to PC and Register file is twice the length of data memory.



Code

```
module cpu;
wire [7:0] addri,addrb,immex,addrex,d2,d1,b,wdata,ra,rb;
wire [2:0] imfunc,func,oper,Ra,Rd,Rb,wraddr,rd1;
wire [3:0] op;
wire [5:0] imm;
wire [6:0] addr;
wire [15:0] iw;
```

```
wire  jump,memtoreg,memwrite,regwrite,alusrc,branch,iszero,overflow,iscarry;
reg clk0,clk1, reset,load,enable;

pc pc0(addri,load,addrb, enable, clk1, reset);

imem imem0(addrb,iw);

ir ir0(iw,op,Ra,Rb,Rd,func,imm,addr);

CU CU0(op,jump,memtoreg,memwrite,regwrite,alusrc,branch);

regfile regfile0(clk1,regwrite,wraddr,wdata,Ra,ra,Rb,rb,iscarry);

ALU ALU0(oper,ra,b,d2,iszero,overflow,iscarry);

datamemory dmem0(clk0,memwrite,d2,rb,d1);

mux3bit21 mux2(Rd,Rb,alusrc,wraddr);

mux3bit21 mux4(func,imfunc,alusrc,oper);

mux8bit21 mux1(rb,immex,alusrc^branch,b);

mux8bit21 mux5(d2,d1,memtoreg,wdata);

mux8bit21 mux6 (addrb+immex,addrex,jump,addri);

bitextender bitex(imm,immex);

adrsExtend adrex(addr,addrex);

dc1 dc0(branch,imfunc);

always @ (*) load = jump | (branch & iszero);
```

```
initial begin
enable = 1; reset = 0; load = 0; clk0 = 0; clk1 = 0;
 $monitor("pc = %d",addrb);
 #700 $finish;
end


always #1 clk0 = ~clk0;
always #2 clk1 = ~clk1;
endmodule
```

That concludes our project.