Preliminary Design Report

b) The assembly language equivalent of the machine codes given in the imem module.

|  |  |  |
| --- | --- | --- |
| Location | Machine Instruction | Assembly Expression |
| 8'h00 | 0x20020005 | addi $v0, $0, 5 |
| 8'h04 | 0x2003000c | addi $v1, $0, 12 |
| 8'h08 | 0x2067fff7 | addi $a3, $v1, -9 |
| 8'h0c | 0x00e22025 | or $a0, $a3, $v0 |
| 8'h10 | 0x00642824 | and $a1, $v1, $a0 |
| 8'h14 | 0x00a42820 | add $a1, $a1, $a0 |
| 8'h18 | 0x10a7000a | beq $a1, $a3, 10 |
| 8'h1c | 0x0064202a | slt $a0, $v1, $a0 |
| 8'h20 | 0x10800001 | beq $a0, $0, 1 |
| 8'h24 | 0x20050000 | addi $a1, $0, 0 |
| 8'h28 | 0x00e2202a | slt, $a0, $a3, $v0 |
| 8'h2c | 0x00853820 | add $a3, $a0, $a1 |
| 8'h30 | 0x00e23822 | sub $a3, $a3, $v0 |
| 8'h34 | 0xac670044 | sw $a3, 0x0044($v1) |
| 8'h38 | 0x8c020050 | lw $v0, 0x0050($0) |
| 8'h3c | 0x08000011 | j 0x0000011 |
| 8'h40 | 0x20020001 | addi $v0, $0, 0x0001 |
| 8'h44 | 0xac020054 | sw $v0, 0x0054($0) |
| 8'h48 | 0x08000012 | j 48 (0x0000012) |

c) Register Transfer Level (RTL) expressions for each of the new instructions that you are adding:

New instructions will be added:

New: “jalm”, “push”

First one is jalm:

jalm: IM[PC]

RF[rt]  PC + 4

PC  DM[RF[rs] + SignExt(immed)]

push: *IM[PC]*

*DM[RF[29] - 4]🡨 RF[rt]*

*RF[29] 🡨 RF[29] – 4*

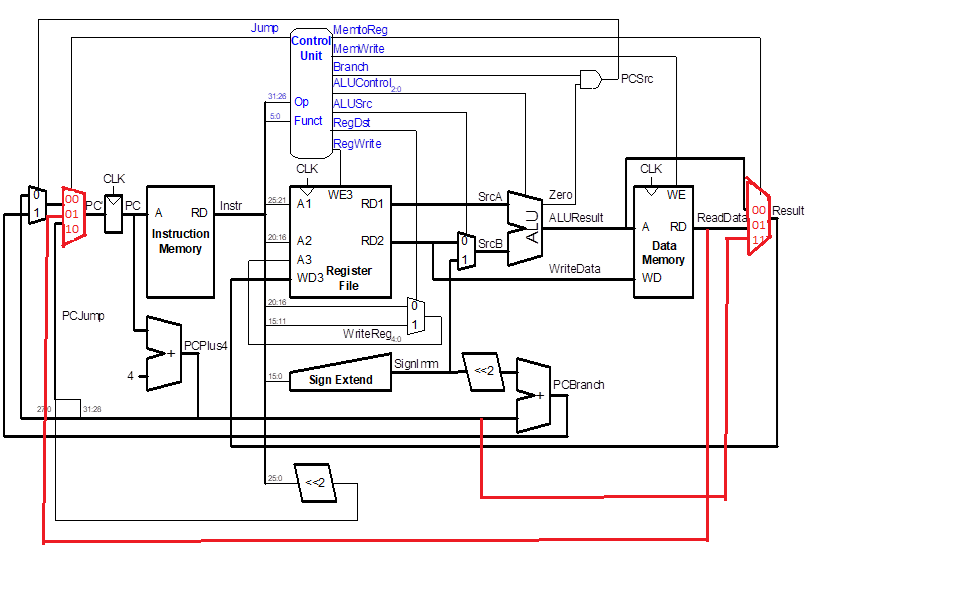
*PC 🡨 PC + 4*

d)

Modifications that needed to be done:

For jalm instruction which is I-type instruction, we need to make changes in both datapath and the control unit of the MIPS single-cycle architecture. For the changes in the datapath:

To be able to add jalm instruction, for the RTL expression RF[rt]  PC + 4, we need to connect PC + 4 to the WriteData port of the Register File with using the multiplexer called MemToReg. But it already takes 2 inputs, therefore, we need to make it 3to1 multiplexer. Namely, it needs to take 2 bit data, let’s choose 11 for the PC + 4 input. Now our MemToReg mux is has 2-bit control signal. Other change need to be done for the RTL expression PC DM[RF[rs] + SignExt(immed)]. We need to take DataMemory value from DM’s ReadData and connect it with the Jump Mux. Now, we also need to make it 2 bit and we gave 01 for this option.

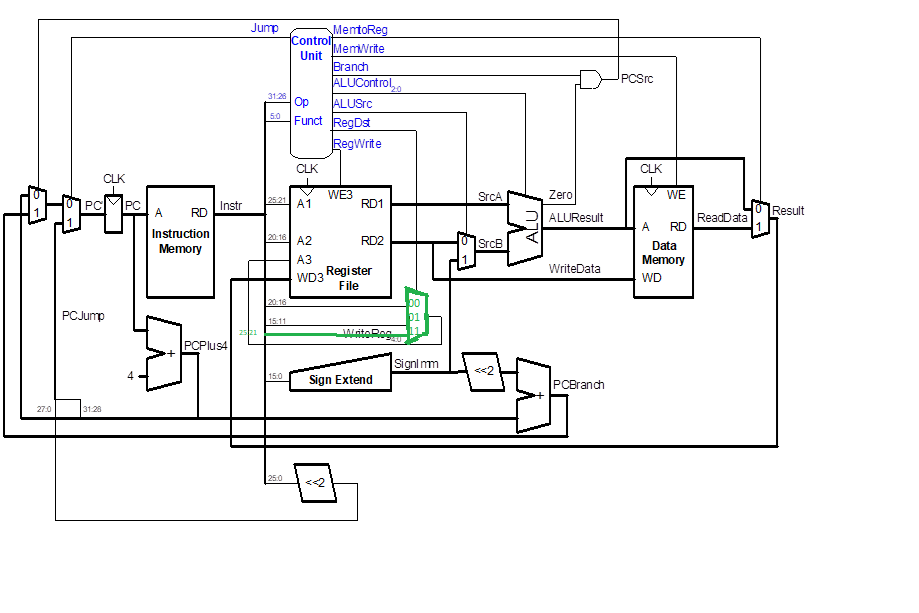


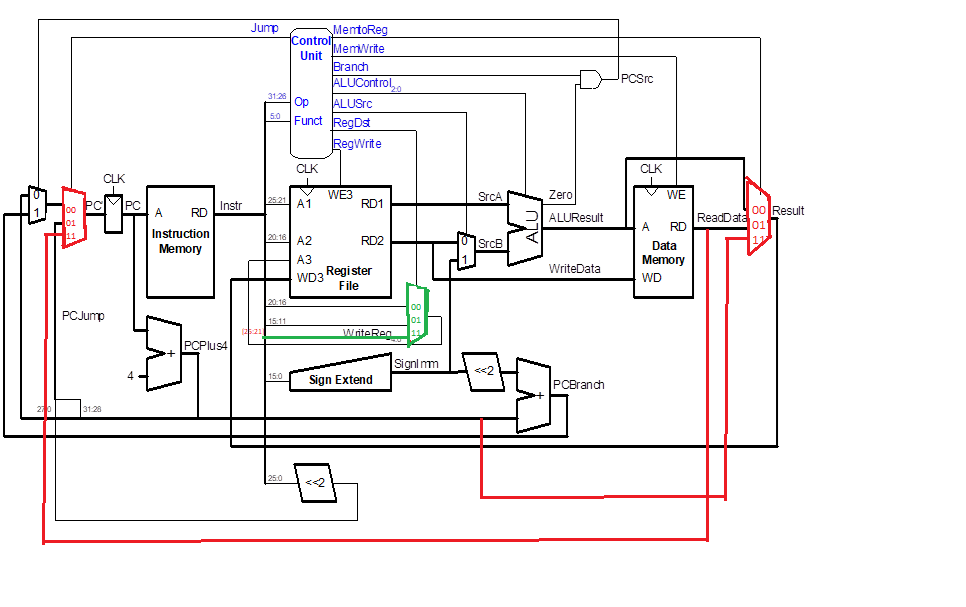
For the Push instruction

* For *RF[rs] 🡨 RF[rs] – 4, we need to choose rs as the destination register. Normally,*

*control statement RegDst chooses destination between 20:16 for rt and 15:11 for rd in R type. But we need to add rs (Inst [25-21]), to the RegDst mux. Thus, RegDst mux becomes a 3-to-1 MUX. Thus, we expand its control signal to a 2-bits.Instruction like RF[29] – 4 can be done with the ALU thus, no change is necessary.*

*Single-cycle map for push:*

**

Combination of the changes that made to adapt jalm and push functions 

Finalized version of the modification that made.

To sum up, there had been added 3 mux to sustain following expression:

* *RF[rs] 🡨 RF[rs] – 4 2 bit RegDst to make rs as a destination register.*
* RF[rt]  PC + 4 2 bit MemToReg to save PC+4 to the destination register rt.
* PC DM[RF[rs] + SignExt(immed)] we need 2 bit jump instruction to save readData value to the PC

e) Default control Unit:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** |  | **Reg**  **Write** | **RegDst** | **ALUSrc** | **Branch** | **MemWrite** | **Mem**  **toReg** | **ALUOp** | **Jump** |
| R-type |  | 1 | *1* | 0 | 0 | 0 | *0* | 10 | *0* |
| lw |  | 1 | *0* | 1 | 0 | 0 | *1* | 00 | *0* |
| sw |  | 0 | *0* | 1 | 0 | 1 | *0* | 00 | *0* |
| beq |  | 0 | *0* | 0 | 1 | 0 | *0* | 01 | *0* |
| addi |  | 1 | *0* | 1 | 0 | 0 | *0* | 00 | *0* |
| j |  | *0* | *0* | *0* | *0* | *0* | *0* | *00* | *1* |

This is default control unit given in our maindec:

assign {regwrite, regdst, alusrc, branch, memwrite,

memtoreg, aluop, jump} = controls;

6'b000000: controls <= 9'b110000100; // R-type

6'b100011: controls <= 9'b1010 01000; // LW

6'b101011: controls <= 9'b0010 10000; // SW

6'b000100: controls <= 9'b0001 00010; // BEQ

6'b001000: controls <= 9'b1010 00000; // ADDI

6'b000010: controls <= 9'b0000 00001; // J

default: controls <= 9'bxxxxxxxxx; // illegal op

Default control unit variables in the maindec which includes given instructions

Note: There is no don’t care in here for some of them because of the fact that this is the implementation in the maindec module of the given single-cycle.

Note2: I gave 2 default default control table first one in in the code, second one with don’t cares.

We need to put 1 or 0 for don’t care’s in the implementation. But these values don’t care because they are not change the result of the datapath.

Table with don’t cares:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** |  | **Reg**  **Write** | **RegDst** | **ALUSrc** | **Branch** | **MemWrite** | **Mem**  **toReg** | **ALUOp** | **Jump** |
| R-type |  | 1 | *1* | 0 | 0 | 0 | *0* | 10 | *0* |
| lw |  | 1 | *0* | 1 | 0 | 0 | *1* | 00 | *0* |
| sw |  | 0 | *X* | 1 | 0 | 1 | *X* | 00 | *0* |
| beq |  | 0 | *X* | 0 | 1 | 0 | *X* | 01 | *0* |
| addi |  | 1 | *0* | 1 | 0 | 0 | *0* | 00 | *0* |
| j |  | *0* | *X* | *X* | *X* | *0* | *X* | *XX* | *1* |

Modified Control Unit Table:

Main decoder Code after the modifications:

logic [12:0] controls;

assign {regwrite, regdst, alusrc, branch, memwrite,

memtoreg, aluop, jump } = controls;

always\_comb

case(op)

6'b000000: controls <= 13'b101000001000; // R-type

6'b100011: controls <= 13'b100100010000; // LW

6'b101011: controls <= 13'b000101000000; // SW

6'b000100: controls <= 13'b000010000100; // BEQ

6'b001000: controls <= 13'b100100000000; // ADDI

6'b000010: controls <= 13'b000000000001; // J

6'b010101: controls <= 13'b100100110011; // Jalm

6'b000111: controls <= 13'b011101000100; // push

default: controls <= 13'bxxxxxxxxxxxx; // illegal op

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** |  | **Reg**  **Write** | **RegDst** | **ALUSrc** | **Branch** | **MemWrite** | **Mem**  **toReg** | **ALUOp** | **Jump** |
| R-type |  | 1 | *01* | 0 | 0 | 0 | *00* | 10 | *00* |
| lw |  | 1 | *00* | 1 | 0 | 0 | *01* | 00 | *00* |
| sw |  | 0 | *00* | 1 | 0 | 1 | *00* | 00 | *00* |
| beq |  | 0 | *00* | 0 | 1 | 0 | *00* | 01 | *00* |
| addi |  | 1 | *00* | 1 | 0 | 0 | *00* | 00 | *00* |
| j |  | *0* | *00* | *0* | *0* | *0* | *00* | *00* | *01* |
| jalm |  | *1* | *00* | *1* | *0* | *0* | *11* | *00* | *11* |
| push |  | *0* | *11* | *1* | *0* | *1* | *00* | *01* | *00* |

This is table that after the implementation of the given changes.

No change is necessary for the ALU decoder as described in the RTL, we can handle theses instruction with the our default ALU decoder.

PART F

* #Mips instruction for: push $a1

.text

add $t1, $0, 4

sub $sp , $sp , $t1

sw $a1, 0($sp)

* #Mips instruction for: jalm $t5, 40($t1)

.text

#jalm $t5, 40($t1)

lui $at, 0x0040

ori $t1, $at, 0x0000

#add 4 to the given addres which is the immediate in the jalm $t5, 4($t1)

addi $t2, $0, 40

add $t3, $t2, $t1 # which is the adddress that we will go in this case it is the 28 + $t1

#in the jalm instruction 0x00400000 + 40 = 0x00400028 40 has converted to hex

jal ege #using jal we have saved pc +4 to the $ra

ege:

add $t5, $0, $ra # with jal we saved pc + 4 to the $ra now we will make # RF[rt]  PC + 4

#adding the $ra to the our destination register $t5

jr $t3 # as we showed 0x00400028 is the address that we will go PC 

# DM[RF[rs] + SignExt(immed)]

#these are the test cases

addi $t4, $0, 0

addi $t4, $t4, 10

addi $t4, $t4, 10

addi $t4, $t4, 10 #0x00400028 this is our address we will reach here

addi $t4, $t4, 10

addi $t4, $t4, 10

* All code given instructions jalm and push added and the instuctions in the instruction memory

.text

#jalm $t5, 40($t1) is below

lui $at, 0x0040

ori $t1, $at, 0x0000

#add 4 to the given addres which is the immediate in the jalm $t5, 4($t1)

addi $t2, $0, 40

add $t3, $t2, $t1 # which is the adddress that we will go in this case it is the 28 + $t1

#in the jalm instruction 0x00400000 + 40 = 0x00400028 40 has converted to hex

jal ege #using jal we have saved pc +4 to the $ra

ege:

add $t5, $0, $ra # with jal we saved pc + 4 to the $ra now we will make #RF[rt]  PC + 4

#adding the $ra to the our destination register $t5

jr $t3 # as we showed 0x00400028 is the address that we will go PC  DM[RF[rs] + SignExt(immed)]

#these are the test cases

addi $t4, $0, 0

addi $t4, $t4, 10

addi $t4, $t4, 10

addi $t4, $t4, 10 #0x00400028 this is our address we will reach here

addi $t4, $t4, 10

addi $t4, $t4, 10

#Mips instruction for: push $a1

add $t1, $0, 4

sub $sp , $sp , $t1

sw $a1, 0($sp)

#after this instruction has executed

addi $sp, $sp, 4

#given code in the instruction memory

addi $v0, $0, 5

addi $v1, $0, 12

addi $a3, $v1, -9

or $a0, $a3, $v0

and $a1, $v1, $a0

add $a1, $a1, $a0

beq $a1, $a3, goHere2

slt $a0, $v1, $a0

beq $a0, $0, goHere

addi $a1, $0, 0

goHere: slt, $a0, $a3, $v0

add $a3, $a0, $a1

sub $a3, $a3, $v0

sw $a3, 68($v1)

lw $v0, 80($0)

j goHere2

addi $v0, $0, 1

goHere2: sw $v0, 84($0)

itself: j itself

PART G

Modules that need to be modifies:

module mips: some variables need to be made 2 bits

module mips (input logic clk, reset,

output logic[31:0] pc,

input logic[31:0] instr,

output logic memwrite,

output logic[31:0] aluout, writedata,

input logic[31:0] readdata,

output logic[31:0] writedata2);

logic [1:0] memtoreg; logic pcsrc, zero, alusrc;

logic [1:0] regdst;

logic regwrite, look;

logic [1:0] jump;

logic [2:0] alucontrol;

controller c (instr[31:26], instr[5:0], zero, memtoreg, memwrite, pcsrc,

alusrc, regdst, regwrite, jump, alucontrol, look);

datapath dp (look, clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,

alucontrol, zero, pc, instr, aluout, writedata, readdata, writedata2);

endmodule

// I made memtoreg 2 bit to be able to choose among aluresult, readdata and pcplus4

// I made regdest 2 bit to be able to choose among rt, rd, and rs fields

// I made jump 2 bit to be able to choose between jump target, pc, and the readdata value which brings our next Pc address

// I add look to be able to choose between rt and rs in the reg file according to the wanted source

module controller: I made same bit extensions in the controller module.

module controller(input logic[5:0] op, funct,

input logic zero,

output logic [1:0] memtoreg,

output logic memwrite,

output logic pcsrc, alusrc,

output logic [1:0] regdst,

output logic regwrite,

output logic [1:0] jump,

output logic[2:0] alucontrol, output logic look);

logic [1:0] aluop;

logic branch;

maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,

jump, aluop, look);

aludec ad (funct, aluop, alucontrol);

assign pcsrc = branch & zero;

endmodule

// I made memtoreg 2 bit to be able to choose among aluresult, readdata and pcplus4

// I made regdest 2 bit to be able to choose among rt, rd, and rs fields

// I made jump 2 bit to be able to choose between jump target, pc, and the readdata value which brings our next Pc address

module maindec:

module maindec (input logic[5:0] op,

output logic [1:0] memtoreg,

output logic memwrite, branch,

output logic alusrc,

output logic [1:0] regdst,

output logic regwrite,

output logic [1:0] jump,

output logic[1:0] aluop , output logic look);

logic [12:0] controls;

assign {regwrite, regdst, alusrc, branch, memwrite,

memtoreg, aluop, jump, look} = controls;

always\_comb

case(op)

6'b000000: controls <= 13'b1010000010000; // R-type

6'b100011: controls <= 13'b1001000100000; // LW

6'b101011: controls <= 13'b0001010000000; // SW

6'b000100: controls <= 13'b0000100001000; // BEQ

6'b001000: controls <= 13'b1001000000000; // ADDI

6'b000010: controls <= 13'b0000000000010; // J

6'b010101: controls <= 13'b1001001100110; // Jalm

6'b000111: controls <= 13'b0111010001001; // push

default: controls <= 13'bxxxxxxxxxxxx; // illegal op

endcase

endmodule

I made extended our controller unit to a 13 bits, I gave new opcodes for jalm and push

I gave new value to regDst, jump and memtoReg. These are determined according to my control value table.

module datapath:

module datapath (input logic look, clk, reset,

input logic [1:0] memtoreg,

input logic pcsrc, alusrc,

input logic [1:0] regdst,

input logic regwrite,

input logic [1:0] jump,

input logic[2:0] alucontrol,

output logic zero,

output logic[31:0] pc,

input logic[31:0] instr,

output logic[31:0] aluout, writedata,

input logic[31:0] readdata,

output logic[31:0] writedata2);

logic [4:0] writereg;

logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;

logic [31:0] signimm, signimmsh, srca, srcb, result;

// next PC logic

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcplus4);

sl2 immsh(signimm, signimmsh);

adder pcadd2(pcplus4, signimmsh, pcbranch);

mux2 #(32) pcbrmux(pcplus4, pcbranch, pcsrc,

pcnextbr);

mux3 #(32) pcmux(pcnextbr, {pcplus4[31:28],

instr[25:0], 2'b00}, readdata , jump, pcnext);

// register file logic

regfile rf (clk, regwrite, instr[25:21], instr[20:16], writereg, look,

result, srca, writedata, writedata2);

mux3 #(5) wrmux (instr[20:16], instr[15:11], instr[25:21] , regdst, writereg);

mux3 #(32) resmux (aluout, readdata, pcplus4, memtoreg, result);

signext se (instr[15:0], signimm);

// ALU logic

mux2 #(32) srcbmux (writedata, signimm, alusrc, srcb);

alu alu (srca, srcb, alucontrol, aluout, zero);

endmodule

I have extended 3 2x1 mux to 3x1 mux as I showed in the datapath.

One 3x1 for memtoreg mux bit to be able to choose among aluresult, readdata and pcplus4

One 3x1 for regdest mux bit to be able to choose among rt, rd, and rs fields

One 3x1 for jump mux bit to be able to choose among jump target, pc, and the readdata value which brings our next Pc address

module mux3:

module mux3 #(parameter WIDTH = 8)

(input logic[WIDTH-1:0] d0, d1, d2,

input logic [1:0] s,

output logic[WIDTH-1:0] y);

assign y = s[1] ? (s[1] ? d2 : d2) : (s[0] ? d1 : d0);

endmodule

I have added this 3mux and used this for both 3 mux memtoreg, regDst, jump

Module alu: needed to be written

module alu(input logic [31:0] a, b,

input logic [2:0] alucont,

output logic [31:0] result,

output logic zero);

// details of the model need to be

// filled in by you, the designer !

always @(\*)

begin// (1)

if(alucont == 3'b010) //add

begin

result = a + b;

end

else if(alucont == 3'b110) //sub

begin

result = a - b;

end

else if(alucont == 3'b000) // and

begin

result = a & b;

end

else if(alucont == 3'b001) // or

begin

result = a | b;

end

else if(alucont == 3'b111) //slt

begin

if(a < b)

begin

result = 32'b1;

end

else

begin

result = 32'b0;

end

end

else if(alucont == 3'bxxx) //we don't know

begin

end

if(result == 32'b0)

begin

zero = 1;

end

else

begin

zero = 0;

end

end // (1)

endmodule

* Hex code for the instructions written by me. These are created to implement push and jalm their opcode specified by me in the maindec module.

This is hex code of the jalm instruction

8'h00: instr = 32'h54030034; //hex for jalm

This is hex code for the push instruction:

8'h08: instr = 32'h1fa30000;

module imem ( input logic [5:0] addr, output logic [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM

always\_comb

case ({addr,2'b00}) // word-aligned fetch

// address instruction

// ------- -----------

8'h00: instr = 32'h20020005; // disassemble, by hand

8'h04: instr = 32'h2003000c; // or with a program,

8'h08: instr = 32'h2067fff7; // to find out what

8'h0c: instr = 32'h00e22025; // this program does!

8'h10: instr = 32'h00642824;

8'h14: instr = 32'h00a42820;

8'h18: instr = 32'h10a7000a;

8'h1c: instr = 32'h0064202a;

8'h20: instr = 32'h10800001;

8'h24: instr = 32'h20050000;

8'h28: instr = 32'h00e2202a;

8'h2c: instr = 32'h00853820;

8'h30: instr = 32'h00e23822;

8'h34: instr = 32'hac670044;

8'h38: instr = 32'h8c020050;

8'h3c: instr = 32'h08000011;

8'h40: instr = 32'h20020001;

8'h44: instr = 32'hac020054;

8'h48: instr = 32'h54030034; //jalm instruction

8'h4c: instr = 32'h1fa30000; //push instruction

8'h50: instr = 32'h08000012; // j 48, so it will loop here\*/

default: instr = {32{1'bx}}; // unknown address

endcase

endmodule

I have added jalm and push hex codes into the imem module