|  |
| --- |
| **PROJECT: 32-BIT SINGLE CYCLE PROCESSOR**  **DEVELOPMENT AND TESTING** |



|  |  |
| --- | --- |
| **Project: 32-bit Single Cycle Processor Development and Testing** | |
| **Authors:** | **Barak Barclay** |
| **Recipient:** | **Dr. Dahua Qin** |
| **Course:** | **ECE 4480-002** |
| **Date:** | **04/17/2018** |

# Introduction

A single cycle processor was designed and implemented previously with Verilog. The Verilog models should be able to be simulated in ModelSim without any modifications. Only a small portion of the MIPS ISA was implemented.

# Learn and Familiarize with the Design

The Verilog file was split in one file per module. After that, comments on what the modules did were added. Below is a list of the comments that were added:

* The ALU module gets register data from registers module, gets func and shamt bits from the Instruction\_Memory, and outputs to Data\_Memory module.
* The ALU\_Control module converts 6 bit funct to 4 bit ALUfunc.
* The branch\_add module is used for branching instructions.
* The Branch\_logic module sssigns Branchsel a value based on Branch, bne, and zero.
* The Data\_Memory module gets values from registers module, main\_control module, and ALU module. It outputs memout.
* The five\_bit\_mux module does the input and output to show the generic names are used.
* The Instruction\_Memory module outputs instructions that correspond to the address given by the PC module.
* The jump\_calc module concatenates the 26 bits from the instruction word after it outputs 12 control actions.
* The main\_control module gets the most significant 6 bits of the instruction word to output values for the 12 control actions.
* The PC module is a register that makes the next instruction address for each clock cycle.
* The PC\_Adder module increases the PC by 4.
* The processor module calls all other modules.
* The registers module gets input register values and outputs data registers
* The shift\_left\_two module performs the shift left 2 on signex before giving the value to the Branch Adder module.
* The sign\_extend module is used for sign extended instructions.
* The test module is the original test bench that’s not really a test bench.
* The thirtytwo\_bit\_mux module does the input and output to show the generic names are used.

# Model Improvements

All modules of the design should be fully synthesizable. The instructions written in the instruction memory needed to be re-written in a test bench. The data memory also needs to be initialized in a test bench. Memory[10] is a misaligned location in the data memory module. Data memory is 32 bits wide and ALUOut needs to be word-aligned. It has a problem with memory addressing definition.

# Write a test bench to test one instruction of each type from the implemented R, I, and J instructions

**Original Instruction Memory simulation:**

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

memory[8] = 32'b00000000000000100001110000000000; //3. sll $3, $2, 16

memory[12] = 32'b00000000000000110010011000000010; //4. srl $4, $3, 24

memory[16] = 32'b00100100000001010000000000001111; //5. addui $5, $0, 17

memory[20] = 32'b00000000100001010011000000100011; //6. subu $6, $4, $5

memory[24] = 32'b00000000110001000011100000100001; //7. addu $7, $6, $4

memory[28] = 32'b00000000111001100100000000100100; //8. and $8, $7, $6

memory[32] = 32'b00000000111001100100100000100110; //9. xor $9, $7, $6

memory[36] = 32'b00000000001010010101000000101010; //10. slt $10, $1, $9

memory[40] = 32'b00010101010000000000000000000010; //11. bne $10, $0, xxx,

// next instruction is 52 = PC+4 + 8

memory[52] = 32'b00000000111010010101100000100101; //12 or $11, $7, $9

memory[56] = 32'b00001100000000000000000000010010; //13 jal XXXX

// I'll jump C0 72 = 1001000

memory[72] = 32'b00110001011011000000000000001010; //14. andi $12, $11, 1010

memory[76] = 32'b00000011111000000000000000001000; //15. jr $31

// next instruction is 60 [56+4] from $31

memory[60] = 32'b00001000000000000000000000010100; //16. j zzzz

// next instruction is 80 = 1010000

memory[80] = 32'b10001101100011010000000000000000; //17. lw $13, 0($12)

memory[84] = 32'b00111001101011100000000000001110; //18. xori $14, $13, 1110

memory[88] = 32'b00110101101011110000000000001110; //19. ori $15, $13, 1110

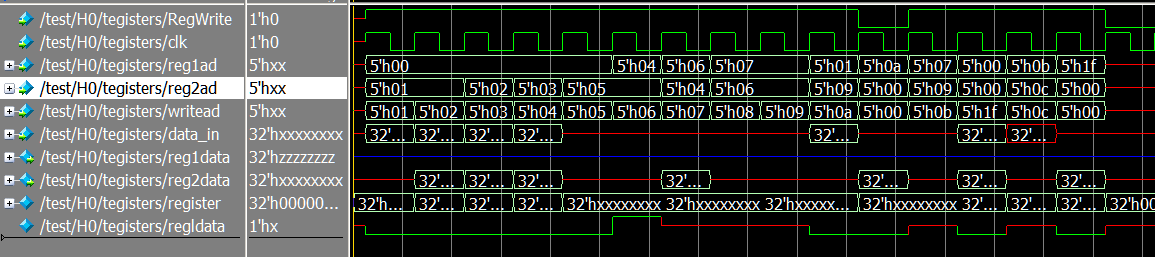
memory[92] = 32'b00000000001010011000000000101011; //20. sltu $16, $1, $9

memory[96] = 32'b00101000001100010000000000000000; //21. slti $17, $1, 0

memory[100] = 32'b00101100001100100000000000000000; //22. sltui $18, s1, 0

memory[104] = 32'b00010010000100100000000000000010; //23. beq $16,$18, yyyy

// next instruction is +12 (4+8)

memory[116] = 32'b10101101100010110000000000000100; //24. sw $11, 4(12) 

**R-Type Instruction (and):**

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

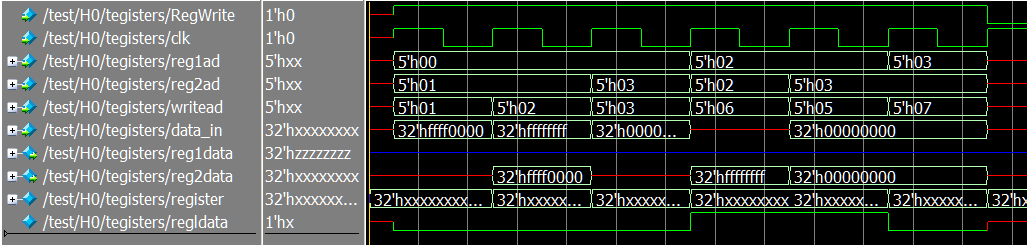
memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

memory[12] = 32'b00000000010000100011000000100100; //4. and $6, $2, $2

memory[16] = 32'b00000000010000110010100000100100; //5. and $5, $2, $3

memory[20] = 32'b00000000011000110011100000100100; //6. and $7, $3, $3

Below shows the simulated code. After the forth-positive edge of the clock reg1add = 2/3, reg2add = 2/3, and RegWrite = 1, writeadd = 6/5/7, and the ALU must have performed the and because regin = 0000…. The results should be stored in registers[6/5/7].



**I-Type Instruction (slti):**

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

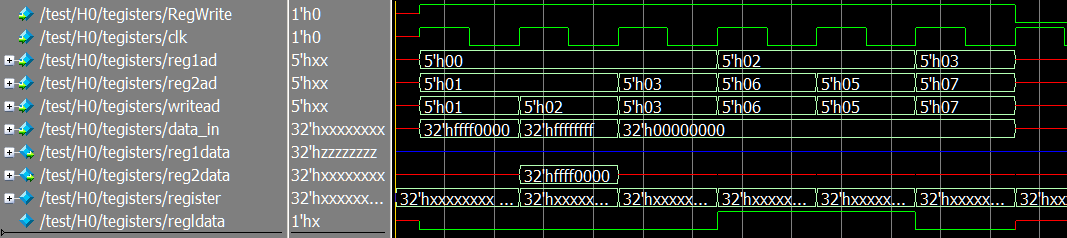
memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

memory[12] = 32'b00101000010001100011000000000000; //4. slti $6, $2, 1

memory[16] = 32'b00101000010001010010100000000000; //5. slti $5, $2, 1

memory[20] = 32'b00101000011001110011111111111111; //6. slti $7, $3, 1

Below shows the simulated code. After the forth-positive edge of the clock reg1add = 2/3, reg2add = 6/5/7, and RegWrite = 1, writeadd = 6/5/7, and the ALU must have performed the and because regin = 0000…. The results should be stored in registers[5/6/7].



**J-Type Instruction (j):**

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

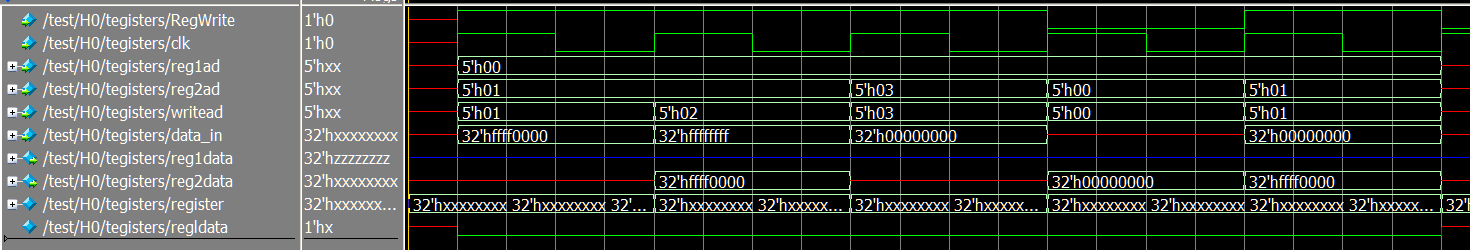
memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

memory[12] = 32'b00001000000000000000000000000101; //4. j mem[20]

// next instruction is 20 = PC+ 8

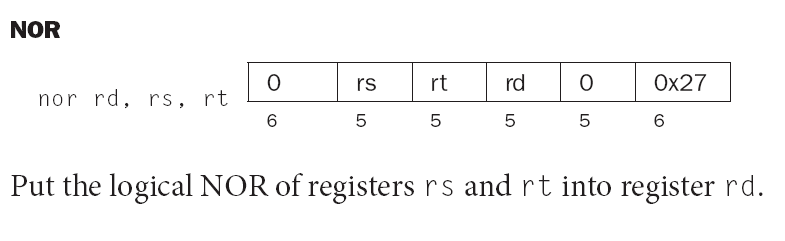
memory[20] = 32'b00111100000000010000000000000000; //5. lui $1, 0

Below shows the simulated code. After the forth-positive edge of the clock reg1add = 0, reg2add = 0/1, and RegWrite = 1, writeadd = 0/1, and the ALU must have performed the and because regin = 0000…. The PC jumps to memory[20].

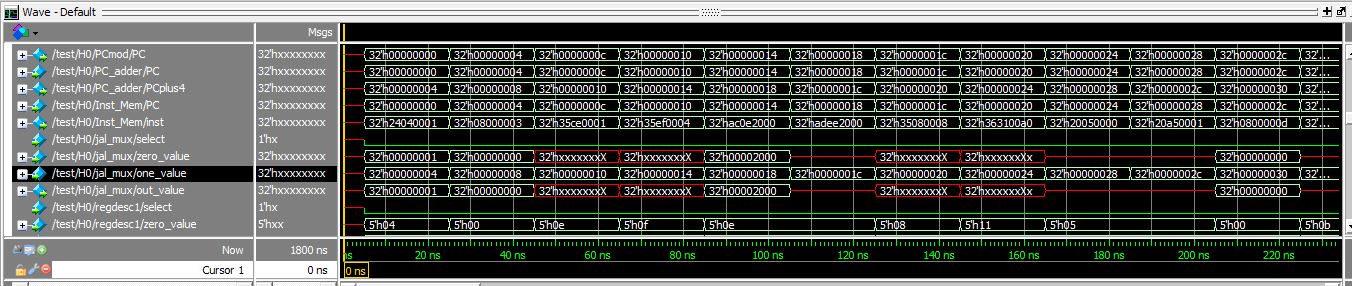


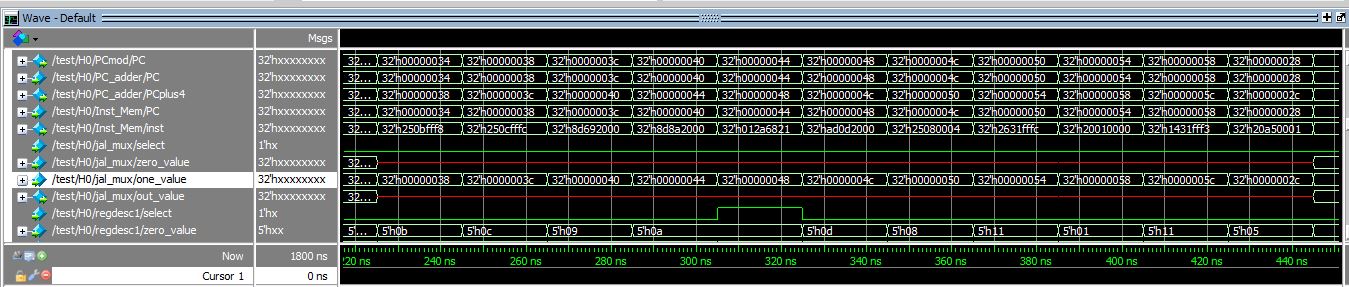
# Implement one instruction from MIPS ISA that is not yet implemented in the design

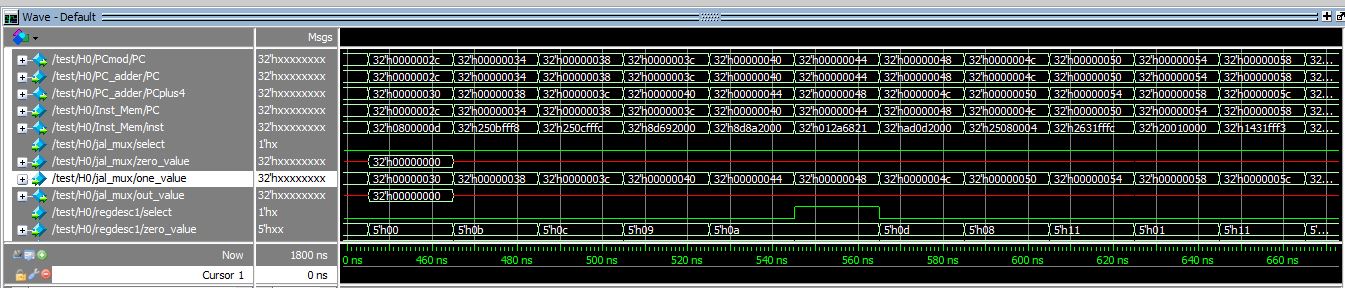
**Implementing and testing the nor instruction:**

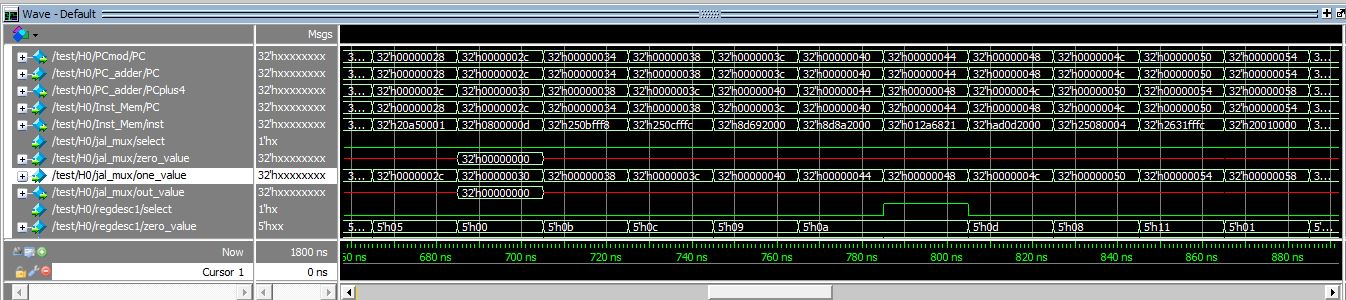


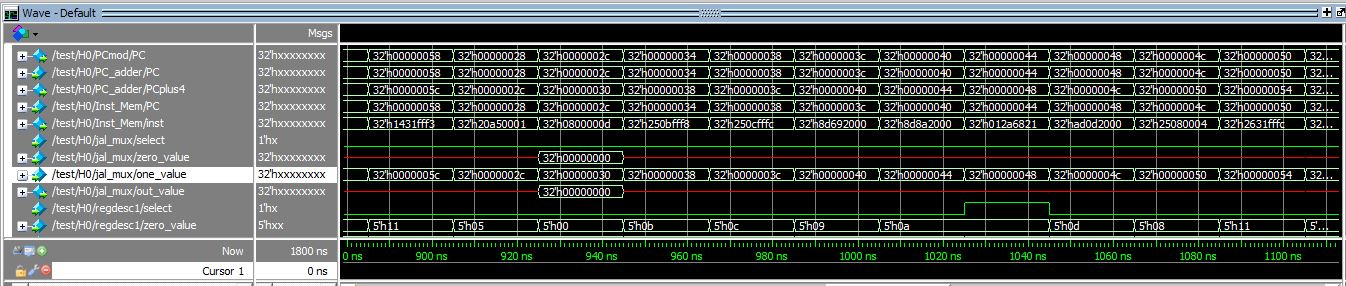
A few modules had to be added to in order to implement the nor instruction:



****

****

****

****

# Appendix (Code)

// Gets register data from registers module.

// Gets func and shamt bits from Instruction\_Memory.

// Outputs to Data\_Memory module.

module ALU (

input[31:0] reg1data, ALU2,

input[3:0] ALUfunc,

input[4:0] shamt,

output reg [31:0] ALUOut,

output reg zero, overflow);

reg[5:0] i;

always @ (\*)

begin

case (ALUfunc)

4'b0010: ALUOut = reg1data + ALU2; // add, addui, lw, sw

4'b0110: ALUOut = reg1data - ALU2; // sub, bne,beq

4'b0000: ALUOut = reg1data & ALU2; // and, andi

4'b0001: ALUOut = reg1data | ALU2; // or , ori

4'b1001: ALUOut = reg1data ^ ALU2; // xor, xori

4'b1010: ALUOut = ALU2 << shamt; // sll slci

4'b1100: ALUOut = ALU2 >> shamt; // srl

4'b0111: begin // slt,- SIGNED #5

if (reg1data[31] == ALU2[31])

begin

if (reg1data < ALU2) ALUOut =1;

else ALUOut = 0;

end

else

begin

if (reg1data > ALU2) ALUOut =1;

else ALUOut = 0;

end

end

4'b1101: ALUOut = ALU2 << 16; // lui

4'b1110: begin // sltu, sltui - UNSIGNED #s

if (reg1data < ALU2) ALUOut = 1;

else ALUOut = 0;

end

4'b1011: begin // sra

ALUOut=ALU2;

for (i=0; i<=31; i=i+1)

begin

if (i < shamt) ALUOut = {ALUOut[31],ALUOut[31:1]};

end

end

default: ALUOut = 0;

endcase

if (ALUOut == 0) zero = 1; else zero = 0;

end

endmodule

// Converts 6 bit funct to 4 bit ALUfunc

module ALU\_Control(

input[3:0] ALUOp,

input[5:0] funct,

output reg jr,

output reg[3:0] ALUfunc);

always @ (ALUOp, funct, ALUfunc)

begin

jr = 0; // default

case (ALUOp)

4'b0010: // R-type instructions

begin

case (funct) // look at function code

6'b100001: ALUfunc = 4'b0010; // addu

6'b100011: ALUfunc = 4'b0110; // subu

6'b100100: ALUfunc = 4'b0000; // and

6'b100101: ALUfunc = 4'b0001; // or

6'b100110: ALUfunc = 4'b1001; // xor

6'b000000: ALUfunc = 4'b1010; // sll

6'b000011: ALUfunc = 4'b1011; // sra

6'b000010: ALUfunc = 4'b1100; // srl

6'b101010: ALUfunc = 4'b0111; // slt

6'b101011: ALUfunc = 4'b1110; // sltu

6'b001000: begin ALUfunc = 4'b0010; jr = 1; end // jr

endcase

end

4'b0011: ALUfunc = 4'b1101; //lui - sll 16

4'b0000: ALUfunc = 4'b0010; //addui, lw, sw - add

4'b0101: ALUfunc = 4'b0000; //andi - AND

4'b0110: ALUfunc = 4'b0001; //ori

4'b0111: ALUfunc = 4'b1001; //xori

4'b0100: ALUfunc = 4'b0111; //slti - SIGNED Slt

4'b0001: ALUfunc = 4'b0110; //beq, bne - subtract

4'b1001: ALUfunc = 4'b1110; //sltui - subtract & set bit

endcase

end

endmodule

always @ \*

begin

case (M0.inst[31:26])

6'b000000 : case (M0.insta[5:0])

6'b100001:Instr\_s = "ADDU";

6'b100011:Instr\_s = "SUBU";

6'b100100:Instr\_s = "AND ";

6'b100101:Instr\_s = "OR ";

6'b100010:Instr\_s = "NOR ";

6'b100110:Instr\_s = "XOR ";

6'b000000:Instr\_s = "SLL ";

6'b000011:Instr\_s = "SRA ";

6'b101010:Instr\_s = "SRL ";

6'b100001:Instr\_s = "SLT ";

6'b101011:Instr\_s = "SLTU";

6'b001000:Instr\_s = "JR ";

6'b100111:Instr\_s = "MLUT";

default :Instr\_s = "idle";

endcase

6'b001111:Instr\_s = "LUI ";

6'b001001:Instr\_s = "ADDUI";

6'b001100:Instr\_s = "ANDI ";

6'b001101:Instr\_s = "ORI ";

6'b011101:Instr\_s = "XORI ";

6'b001010:Instr\_s = "SLTI ";

6'b001011:Instr\_s = "SLTIU";

6'b000100:Instr\_s = "BEQ ";

6'b000101:Instr\_s = "BNE ";

6'b100011:Instr\_s = "LW ";

6'b000010:Instr\_s = "JUMP ";

6'b000011:Instr\_s = "JAL ";

default :Instr\_s = "IDLE ";

endcase

// Used for branching instructions.

module branch\_add (

input[31:0] PCplus4, Boff,

output[31:0] BAout);

assign BAout = PCplus4 + Boff;

endmodule

// Assigns Branchsel a value based on Branch, bne, and zero.

module Branch\_logic(

input Branch, bne, zero,

output Branchsel);

assign Branchsel = (Branch & zero) | (bne & ~zero);

endmodule

// Gets values from registers module, main\_control module, and ALU module.

// Outputs memout.

module Data\_Memory (

input clk, MemRead, MemWrite,

input [31:0] ALUOut, reg2data,

output reg [31:0] memout);

reg [31:0] memory[199:0];

// Lines below are commented out because they would be implemented in a testbench

// initial begin

// memory[8] = 32'b11110000111100001111000011110000;

// memory[10] = 32'b00000000000000000000000000000101; //Must be word addressed

//end

always @ (ALUOut, MemRead)

begin

if (MemRead == 1) memout = memory[ALUOut];

end

always @(posedge clk)

begin

if (MemWrite ==1) memory[ALUOut] <= reg2data;

end

endmodule

module evenBetterTB();

reg reset, clk;

reg [31:0] instrmem[29:0];

reg [5:0] opcode\_lui, opcode\_ori;

reg [4:0] rs, rtl, rd;

reg [15:0] immvalue\_upper, immvalue\_lower;

reg [31:0] temp\_value;

integer i, j, fp;

initial begin

fp = $fopen("mem.bin");

immvalue\_upper = 0;

immvalue\_lower = 0;

$display(^First Second Or`ed);

for(i = 0 ; i <=28; i = i + 2)

begin

opcode\_lui = 6'b001111; // Fixed opcode for LUI instructions

opcode\_orti = 6'b001101; // Fixed opcode for ORI instructions

rs = 5'b00000;

rtl = 1 + ($urandom) % (10); // Random register address for addui

immvalue\_upper = 1 + ($urandom (immvalue\_upper)) % (100); // Random data for register 1

immvalue\_lower = 1 + ($urandom (immvalue\_upper)) % (100); // Random data for register 2

temp\_value = immvalue\_upper;

// Form 30 instructions

// 15 LUI and 15 ORI instructions

for (j=0;j<2;j = j+1)

begin

if(j == 0) instrmem[i+j] = (opcode\_lui, rs, rtl, immvalue\_upper); // Create 1st LUI instruction

else if(j == 1) instrmem[i+j] = {opcode\_ori, rtl, rtl, immvale\_lower); // Create 2nd ORI instruction

$fdisplayb{fb.instrmem[i+j]); //Write the instrcution to file

end

$display("%h %h %h", temp\_value << 16, immvalue\_lower, temp\_value << 16 | immvalue\_lower);

end

$fclose(fb);

#10 reset = 1; clk = 1;

#10 clk = 0; reset = 0;

End

// It does the input and output to show the generic names are used.

module five\_bit\_mux(input select,

input[4:0] zero\_value, one\_value,

output reg [4:0] out\_value);

always @ (select, zero\_value, one\_value)

begin

if (select == 0) out\_value=zero\_value;

if (select == 1) out\_value= one\_value;

end

endmodule

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

memory[12] = 32'b00101000010001100011000000000000; //4. slti $6, $2, 1

memory[16] = 32'b00101000010001010010100000000000; //5. slti $5, $2, 1

memory[20] = 32'b00101000011001110011111111111111; //6. slti $7, $3, 1

// Outputs instructions that correspond to the address given by the PC module.

module Instruction\_Memory (

input [31:0] PC,

output reg [31:0] inst);

// reg [7:0] memory[199:0];

reg[31:0] memory[199:0];

initial begin

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

memory[8] = 32'b00000000000000100001110000000000; //3. sll $3, $2, 16

memory[12] = 32'b00000000000000110010011000000010; //4. srl $4, $3, 24

memory[16] = 32'b00100100000001010000000000001111; //5. addui $5, $0, 17

memory[20] = 32'b00000000100001010011000000100011; //6. subu $6, $4, $5

memory[24] = 32'b00000000110001000011100000100001; //7. addu $7, $6, $4

memory[28] = 32'b00000000111001100100000000100100; //8. and $8, $7, $6

memory[32] = 32'b00000000111001100100100000100110; //9. xor $9, $7, $6

memory[36] = 32'b00000000001010010101000000101010; //10. slt $10, $1, $9

memory[40] = 32'b00010101010000000000000000000010; //11. bne $10, $0, xxx,

// next instruction is 52 = PC+4 + 8

memory[52] = 32'b00000000111010010101100000100101; //12 or $11, $7, $9

memory[56] = 32'b00001100000000000000000000010010; //13 jal XXXX

// I'll jump C0 72 = 1001000

memory[72] = 32'b00110001011011000000000000001010; //14. andi $12, $11, 1010

memory[76] = 32'b00000011111000000000000000001000; //15. jr $31

// next instruction is 60 [56+4] from $31

memory[60] = 32'b00001000000000000000000000010100; //16. j zzzz

// next instruction is 80 = 1010000

memory[80] = 32'b10001101100011010000000000000000; //17. lw $13, 0($12)

memory[84] = 32'b00111001101011100000000000001110; //18. xori $14, $13, 1110

memory[88] = 32'b00110101101011110000000000001110; //19. ori $15, $13, 1110

memory[92] = 32'b00000000001010011000000000101011; //20. sltu $16, $1, $9

memory[96] = 32'b00101000001100010000000000000000; //21. slti $17, $1, 0

memory[100] = 32'b00101100001100100000000000000000; //22. sltui $18, s1, 0

memory[104] = 32'b00010010000100100000000000000010; //23. beq $16,$18, yyyy

// next instruction is +12 (4+8)

memory[116] = 32'b10101101100010110000000000000100; //24. sw $11, 4(12)

end

always @ (PC)

//inst = {memory[PC],memory[PC+1],memory[PC+2],memory[PC+3]);

inst = memory[PC];

endmodule

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

memory[12] = 32'b00001000000000000000000000000101; //4. j mem[20]

// next instruction is 20 = PC+ 8

memory[20] = 32'b00111100000000010000000000000000; //5. lui $1, 0

// Concatenates the 26 bits from the instruction word after it outpus 12 control actions.

module jump\_calc (input[3:0] PCbits,

input[25:0] instbits,

output [31:0] JumpAdrs);

assign JumpAdrs = {PCbits, instbits, 2'b00};

endmodule

// Gets the most significant 6 bits of the instruction word to output values for the 12 control actions.

module main\_control (input[5:0] inst,

output reg[3:0] ALUOp,

output reg ALUSrc, RegDst, RegWrite, MemWrite, MemtoReg,

MemRead, Branch, bne, Jump, jal, ui);

always @ (inst)

begin

case (inst)

6'b000000: // R-type instruction

begin

ALUOp=4'b0010; ALUSrc=0; RegDst=1; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=0;

end

6'b001111: // lui instruction

begin

ALUOp=4'b0011; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=0;

end

6'b001001: // addui instruction

begin

ALUOp=4'b0000; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=1;

end

6'b001100: // andi instruction

begin

ALUOp=4'b0101; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=1;

end

6'b001101: // ori instruction

begin

ALUOp=4'b0110; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=1;

end

6'b001110: // xori instruction

begin

ALUOp=4'b0111; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=1;

end

6'b001010: // slti instruction

begin

ALUOp=4'b0100; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=0;

end

6'b001011: // sltiu instruction

begin

ALUOp=4'b1001; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=1;

end

6'b000100: // beg instruction

begin

ALUOp=4'b0001; ALUSrc=0; RegDst=0; RegWrite=0; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=1; bne=0; Jump=0; jal=0; ui=0;

end

6'b000101: // bne instruction

begin

ALUOp=4'b0001; ALUSrc=0; RegDst=0; RegWrite=0; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=1; Jump=0; jal=0; ui=0;

end

6'b100011: // lu instruction

begin

ALUOp=4'b0000; ALUSrc=1; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=1; MemRead=1; Branch=0; bne=0; Jump=0; jal=0; ui=0;

end

6'b101011: // sw instruction

begin

ALUOp=4'b0000; ALUSrc=1; RegDst=0; RegWrite=0; MemWrite=1;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=0;

end

6'b000010: // j instruction

begin

ALUOp=4'b0000; ALUSrc=0; RegDst=0; RegWrite=0; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=1; jal=0; ui=0;

end

6'b000011: // jal instruction

begin

ALUOp=4'b0000; ALUSrc=0; RegDst=0; RegWrite=1; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=1; jal=1; ui=0;

end

default:

begin

ALUOp=4'b0000; ALUSrc=0; RegDst=0; RegWrite=0; MemWrite=0;

MemtoReg=0; MemRead=0; Branch=0; bne=0; Jump=0; jal=0; ui=0;

end

endcase

end

endmodule

// The PC is a register that makes the next instruction address for each clock cycle.

module PC (input clk, reset,

input[31:0] newPC,

output reg[31:0] PC);

always @(posedge clk or posedge reset)

begin

if (reset == 1)

PC<= 0; // or whereever the first instruction is

else

PC<= newPC;

end

endmodule

// If a branching instruction is not called, the PC is increased by 4.

module PC\_adder(

input [31:0] PC,

output [31:0] PCplus4);

assign PCplus4 = PC + 4;

endmodule

// Calls all other modules.

module processor(

input reset, clk);

wire[3:0] ALUOp;

wire[3:0] ALUfunc;

wire[4:0] regad, Rmuxout, ra;

wire[31:0] PC, PCplus4, newPC, inst, JumpAdrs,

Mmuxout, regin, signex, Boff, ALU2,

BAOut, BMuxout, JMuxout, memout,

ALUOut, reg2data, reg1data;

wire RegWrite;

assign ra = 5'b11111;

PC PCmod (clk, reset, newPC, PC);

PC\_adder PC\_adder (PC, PCplus4);

Instruction\_Memory Inst\_Mem (PC, inst);

thirtytwo\_bit\_mux jal\_mux (.select(jal), .zero\_value(Mmuxout),

.one\_value(PCplus4), .out\_value(regin));

five\_bit\_mux regdesc1(.select(RegDst), .zero\_value(inst[20:16]),

.one\_value(inst[15:11]), .out\_value(Rmuxout));

five\_bit\_mux regdest2(.select(jal), .zero\_value(Rmuxout),

.one\_value(ra), .out\_value(regad));

registers tegisters(RegWrite, clk, inst[25:21], inst[20:16], regad,

regin, reg1data, reg2data);

jump\_calc j\_calc (PCplus4[31:28],inst[25:0],JumpAdrs);

main\_control main\_control (inst[31:26], ALUOp, ALUSrc, RegDst,

RegWrite, MemWrite, MemtoReg, MemRead,

Branch, bne, Jump, jal, ui);

sign\_extend sign\_ex (ui, inst[15:0], signex);

shift\_left\_two SL2 (signex, Boff);

branch\_add Branch\_add (PCplus4, Boff, BAOut);

thirtytwo\_bit\_mux ALUSrcMux (.select(ALUSrc), .zero\_value(reg2data),

.one\_value(signex), .out\_value(ALU2));

ALU ALU (reg1data, ALU2, ALUfunc, inst[10:6], ALUOut, zero, overflow);

ALU\_Control ALU\_cont (ALUOp, inst[5:0], jr, ALUfunc);

Branch\_logic BL (Branch, bne, zero, Branchsel);

thirtytwo\_bit\_mux Bmux (.select(Branchsel), .zero\_value(PCplus4),

.one\_value(BAOut), .out\_value(BMuxout));

thirtytwo\_bit\_mux Jmux (.select(Jump), .zero\_value(BMuxout),

.one\_value(JumpAdrs), .out\_value(JMuxout));

thirtytwo\_bit\_mux JRmux (.select(jr), .zero\_value(JMuxout),

.one\_value(reg1data), .out\_value(newPC));

Data\_Memory DM (clk, MemRead, Memwrite, ALUOut, reg2data, memout);

thirtytwo\_bit\_mux Mmux (.select(MemtoReg), .zero\_value(ALUOut),

.one\_value(memout), .out\_value(Mmuxout));

Endmodule

memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

memory[12] = 32'b00000000010000100011000000100100; //4. and $6, $2, $2

memory[16] = 32'b00000000010000110010100000100100; //5. and $5, $2, $3

memory[20] = 32'b00000000011000110011100000100100; //6. and $7, $3, $3

// Gets input register values and outputs data registers

module registers(input RegWrite, clk,

input[4:0] reg1ad, reg2ad, writead,

input[31:0] data\_in, output wire [31:0] reg1data, reg2data);

reg[31:0] register[31:0];

initial begin register[0] = 0; end // $zero

assign regldata = register[reg1ad]; // read

assign reg2data = register[reg2ad]; // write

always @ (posedge clk)

begin

if (RegWrite== 1) register[writead] <= data\_in;

end

endmodule

// Performs the shift left 2 on signex before giving the value to the Branch Adder module.

module shift\_left\_two (

input[31:0] signex,

output[31:0] Boff);

assign Boff = signex << 2;

endmodule

// Used for sign extended instructions.

module sign\_extend(

input ui,

input[15:0] inst,

output reg [31:0] signex);

always @ (ui, inst)

begin

if (ui == 1) signex = {16'b0000000000000000, inst};

else begin

if (inst[15] == 1) signex={16'b1111111111111111, inst};

if (inst[15] == 0) signex={16'b0000000000000000, inst};

end

end

endmodule

// Load Instruction Memory from Test Bench

initial begin

M0.Inst\_Mem.memory[0] = 32'b00111100000000011111111111111111; //1. lui $1, 1

M0.Inst\_Mem.memory[4] = 32'b00000000000000010001010000000011; //2. sra $2, $1, 16

M0.Inst\_Mem.memory[8] = 32'b00111100000000110000000000000000; //3. lui $3, 0

M0.Inst\_Mem.memory[12] = 32'b00000000010000100011000000100100; //4. and $6, $2, $2

M0.Inst\_Mem.memory[16] = 32'b00000000010000110010100000100100; //5. and $5, $2, $3

M0.Inst\_Mem.memory[20] = 32'b00000000011000110011100000100100; //6. and $7, $3, $3

end

// Load Data Memory from Test Bench

// Data Memory is 32 bits wide ALUOut needs to be word aligned. memory[10] is misaligned location.

initial begin

M0.DM.memory[0] = 32'hDEADBEEF;

M0.DM.memory[1] = 32'hBADDBADD;

end

reg reset, clk;

initial begin

#5 reset = 1; clk = 0;

#100 reset = 0; clk = 0;

#100 reset = 0; clk = 0;

forever #10 clk = ~clk;

end

//

//processor M0 (reset, clk);

// Original test module.

module test;

reg reset, clk;

initial begin

#5 reset = 1; clk = 1;

#10 clk = 0; reset = 0;

forever #10 clk = ~clk;

end

processor H0 (reset, clk);

endmodule

// It does the input and output to show the generic names are used.

module thirtytwo\_bit\_mux(input select,

input[31:0] zero\_value, one\_value,

output reg [31:0] out\_value);

always @ (select, zero\_value, one\_value)

begin

if (select ==0) out\_value = zero\_value;

if (select ==1) out\_value = one\_value;

end

endmodule