**ECE 4680**

**Project Report**

**Team Number: 1**

**Team Members**: David Baron-Vega (gf7068), Daniel Forta (he8407), Alexandros Toghe (hd3653)

**Compilation Status**:

Successful

**Elaboration Status**:

Successful

**Simulation Status**:

Simulates correctly

**A copy of the Verilog design file** (yes another copy within the pdf file in addition to the separate .v file)

CPU Code:

// Author Names: Daniel Forta (he8407), Alex Toghe (hd3653), David Baron-Vega (gf7068)

// Creation date: 3/5/2024

// Last Date Modified: 3/27/2024

// Compilation Status: Successful

// Elaboration Status: Successful

// Simulation Status: Simulates correctly

module CPU (clock,PC, IR, ALUOut, MDR, A, B, reg8, reg9);

parameter R\_FORMAT = 6'b000000;

parameter LW = 6'b100011;

parameter SW = 6'b101011;

parameter BEQ = 6'b000100;

parameter BNE = 6'b000101;

parameter ADDI = 6'b001000;

parameter JM = 6'b111011; //Value seems to be unused

// other opcodes go here

//....

input clock; //the clock is an external input

//Make these datapath registers available outside the module in order to do the testing

output PC, IR, ALUOut, MDR, A, B;

reg[31:0] PC, IR, ALUOut, MDR, A, B;

// The architecturally visible registers and scratch registers for implementation

reg [31:0] Regs[0:31], Memory [0:1023];

reg [2:0] state; // processor state

wire [5:0] opcode; //use to get opcode easily

wire [31:0] SignExtend, PCOffset; //used to get sign extended offset field

assign opcode = IR[31:26]; //opcode is upper 6 bits

assign SignExtend = {{16{IR[15]}},IR[15:0]}; //sign extension of lower 16-bits of instruction

assign PCOffset = SignExtend << 2; //PC offset is shifted

wire [31:0] reg8;

output [31:0] reg8; //output reg 8 for testing

assign reg8 = Regs[8]; //output reg 8 (i.e. $t0)

wire [31:0] reg9;

output [31:0] reg9; //output reg 9 for testing

assign reg9 = Regs[9]; //output reg 9 (i.e. $t1)

initial begin //Load a MIPS test program and data into Memory

//MIPS code starting with addi 8

Memory[2] = 32'h20080008; // addi $t0, $zero, 8

Memory[3] = 32'hac08007c; // sw $t0, 124($zero)

Memory[4] = 32'h8c09007c; // lw $t1, 124($zero)

Memory[5] = 32'h01094820; // add $t1, $t0, $t1

Memory[6] = 32'h11090002; // beq $t0, $t1, NOSWAP

Memory[7] = 32'h01094039; // swap $t0, $t1

Memory[8] = 32'h01094822; // sub $t1, $t0, $t1

Memory[9] = 32'h15090001; // NOSWAP: bne $t0, $t1, NONOR

Memory[10] = 32'h01094827; // nor $t1, $t0, $t1

Memory[11] = 32'hec09007c; // NONOR: jm mem

//MIPS code starting with addi 0

//Memory[2] = 32'h20080000; // addi $t0, $zero, 0

//Memory[3] = 32'hac08007c; // sw $t0, 124($zero)

//Memory[4] = 32'h8c09007c; // lw $t1, 124($zero)

//Memory[5] = 32'h01094820; // add $t1, $t0, $t1

//Memory[6] = 32'h11090002; // beq $t0, $t1, NOSWAP

//Memory[7] = 32'h01094039; // swap $t0, $t1

//Memory[8] = 32'h01094822; // sub $t1, $t0, $t1

//Memory[9] = 32'h15090001; // NOSWAP: bne $t0, $t1, NONOR

//Memory[10] = 32'h01094827; // nor $t1, $t0, $t1

//Memory[11] = 32'hec09007c; // NONOR: jm mem

end

initial begin // set the PC to 8 and start the control in state 1 to start fetch instructions from Memory[2] (byte 8)

PC = 8;

state = 1;

end

always @(posedge clock) begin

//make R0 0

//short-cut way to make sure R0 is always 0

Regs[0] = 0;

case (state) //action depends on the state

1: begin //first step: fetch the instruction, increment PC, go to next state

IR <= Memory[PC>>2]; //changed

PC <= PC + 4; //changed

state = 2; //next state

end

2: begin //second step: Instruction decode, register fetch

A <= Regs[IR[25:21]]; //rs register

state = 3;

end

3: begin //third step: Second fetch, also compute branch address

B <= Regs[IR[20:16]]; //rt register

ALUOut <= PC + PCOffset; // compute PC-relative branch target

state = 4;

end

4: begin //fourth step: Load/Store execution, ALU execution, Branch completion

state = 5; // default next state

if (opcode == R\_FORMAT)

case (IR[5:0]) //case for the various R-type instructions

32: ALUOut = A + B; //add operation

34: ALUOut = A - B; //sub operation

39: ALUOut = ~(A | B); //nor operation

57: begin //57 = (0x38 + 1)

A <= B; //Non-blocking setting A = B

B <= A; //Non-blocking setting B = A

end

// other function fields for R-Format instructions go here

//

//

default: ALUOut = A; //other R-type operations

endcase

else if ((opcode == LW) | (opcode==SW) | (opcode==JM))

ALUOut <= A + SignExtend; //compute effective address

else if (opcode == BEQ) begin

if (A==B)

PC <= ALUOut; // branch taken--update PC

state = 1; // BEQ finished, return to first state

end

else if (opcode == BNE) begin

if (A!=B)

PC <= ALUOut; // branch taken--update PC

state = 1; // BEQ finished, return to first state

end

else if (opcode == ADDI)

ALUOut <= A + SignExtend;

// implementations of other instructions (such bne, addi, etc.) as go here

// else if ...

// ...

// else if ...

// ...

end

5: begin //fifth step

if (opcode == R\_FORMAT) begin //ALU Operation (SPLIT INTO 2 STEPS)

if (IR[5:0] == 57) begin //if SWAP operation value

Regs[IR[25:21]] <= A; //rs = A which is now equal to original //rt (from B)

state = 6;

end

else begin

Regs[IR[15:11]] <= ALUOut; // write the result

state = 1;

end

end //R-type finishes

else if ((opcode == LW) | (opcode == JM)) begin // load instruction

MDR <= Memory[ALUOut>>2]; // read the memory

state = 6; // next state

end

else if (opcode == SW) begin

Memory[ALUOut>>2] <= B; // write the memory

state = 1; // return to state 1

end //store finishes

else if (opcode == ADDI) begin

Regs[IR[20:16]] <= ALUOut; // write the result to rt

state = 1;

end

// implementations of other instructions (such as addi, etc.) go here

// else if ...

// ...

// else if ...

// ...

end

6: begin

if (opcode == R\_FORMAT) begin //ALU Operation (SPLIT INTO 2 STEPS) (only swap R //format makes it here)

Regs[IR[20:16]] <= B; //rt = B which is now equal to original //rs (from A)

state = 1;

end //R-type finishes

if (opcode == LW) begin

Regs[IR[20:16]] = MDR; // write the MDR to the register

state = 1;

end

else if (opcode == JM) begin

PC = MDR; // write the MDR to the PC

state = 1;

end

end

endcase

end // always

endmodule

TestBench Code:

// Author Names: Daniel Forta (he8407), Alex Toghe (hd3653), David Baron-Vega (gf7068)

// Creation date: 3/5/2024

// Last Date Modified: 3/27/2024

module cpu\_tb;

wire[31:0] PC, IR, ALUOut, MDR, A, B, reg8, reg9;

reg clock;

CPU cpu1 (clock,PC, IR, ALUOut, MDR, A, B, reg8, reg9);// Instantiate CPU module

initial begin

clock = 0;

repeat (104) // 2\*51 + padding needed

begin

#10 clock = ~clock; //alternate clock signal

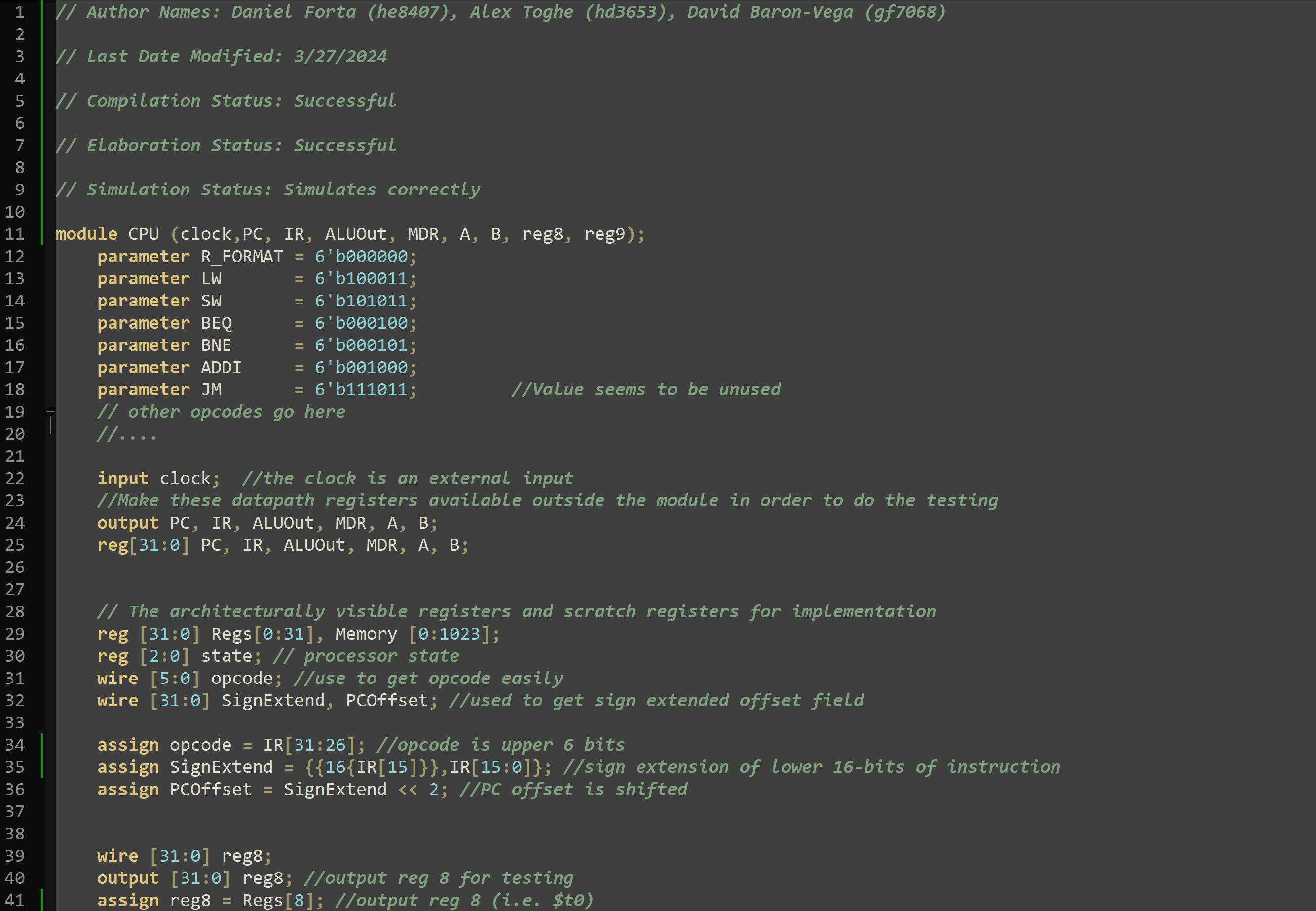
end

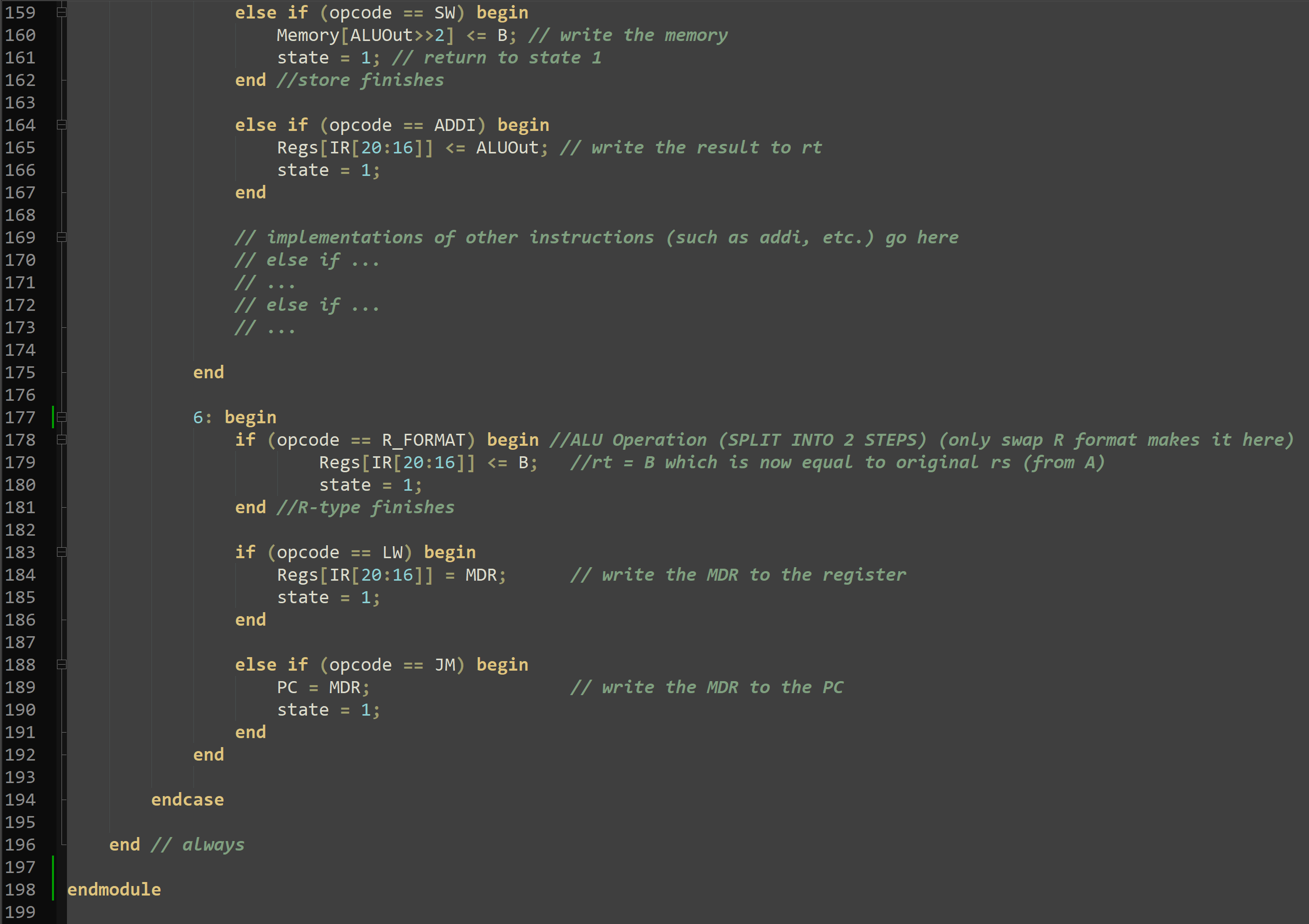
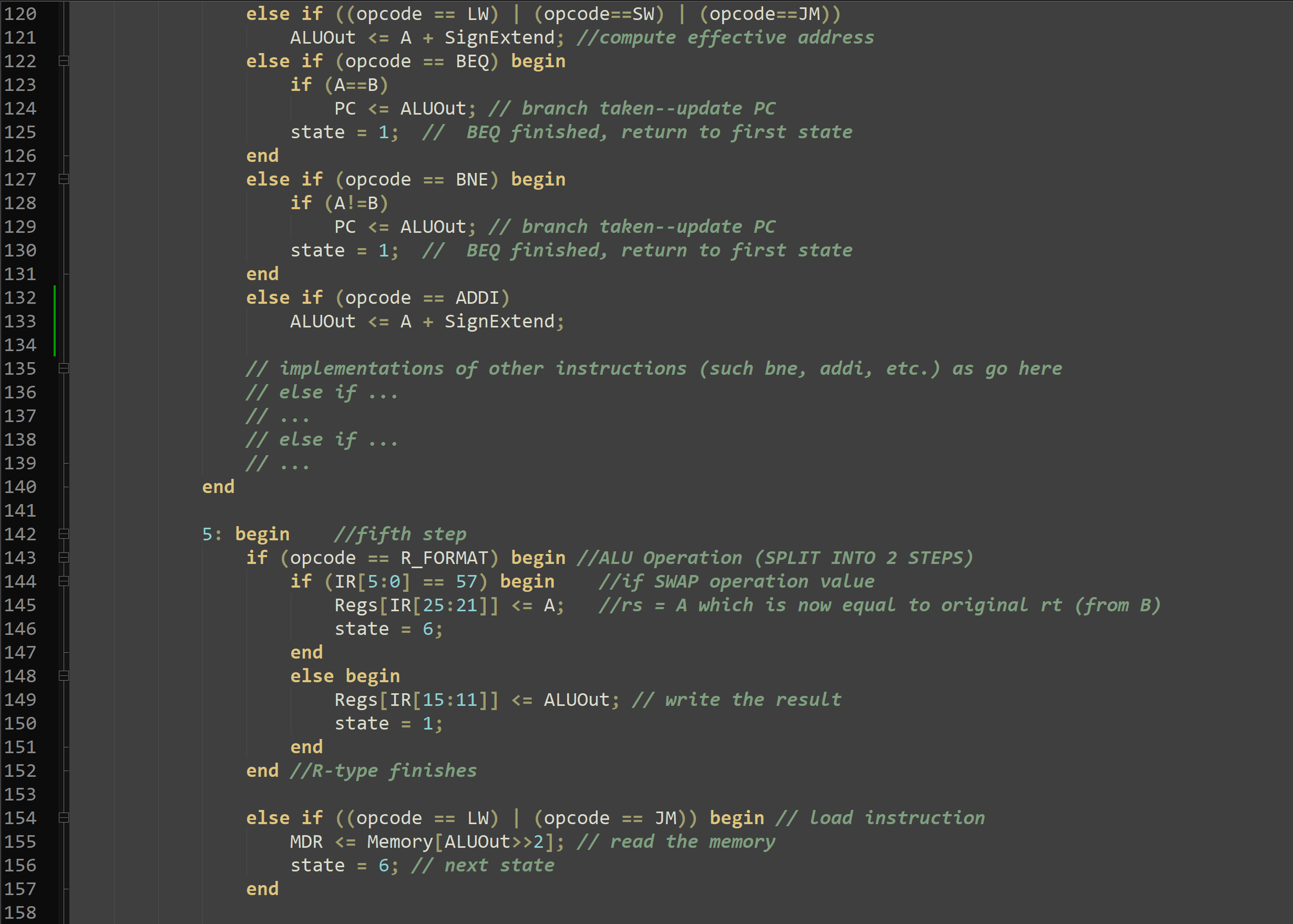
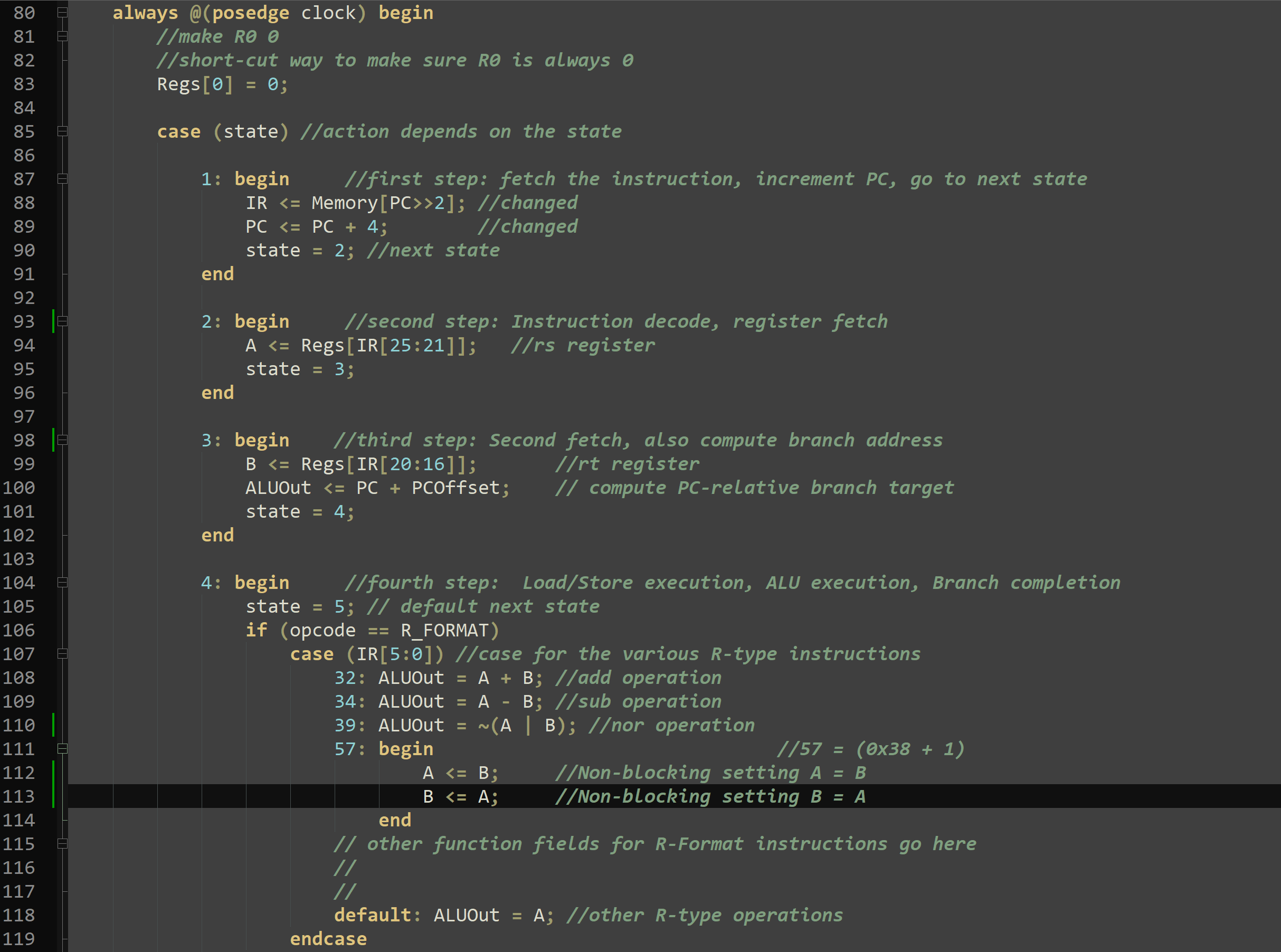
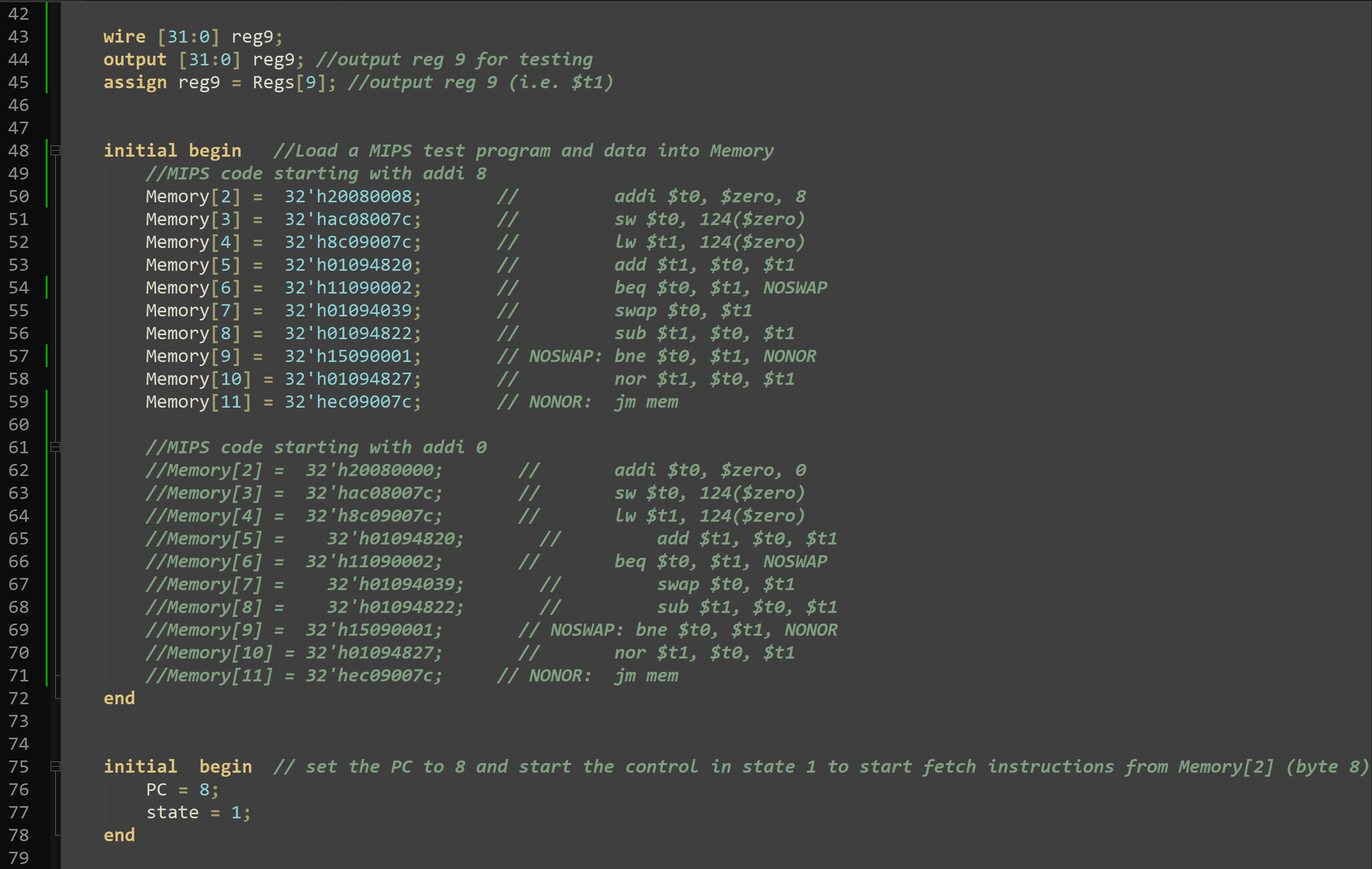
$finish;

end

endmodule

Insert snapshot here:





**A copy of the Verilog testbench** (yes another copy within the pdf file in addition to the separate .v file)

module cpu\_tb;

wire[31:0] PC, IR, ALUOut, MDR, A, B, reg8, reg9;

reg clock;

CPU cpu1 (clock,PC, IR, ALUOut, MDR, A, B, reg8, reg9);// Instantiate CPU module

initial begin

clock = 0;

repeat (104) // 2\*52 needed

begin

#10 clock = ~clock; //alternate clock signal

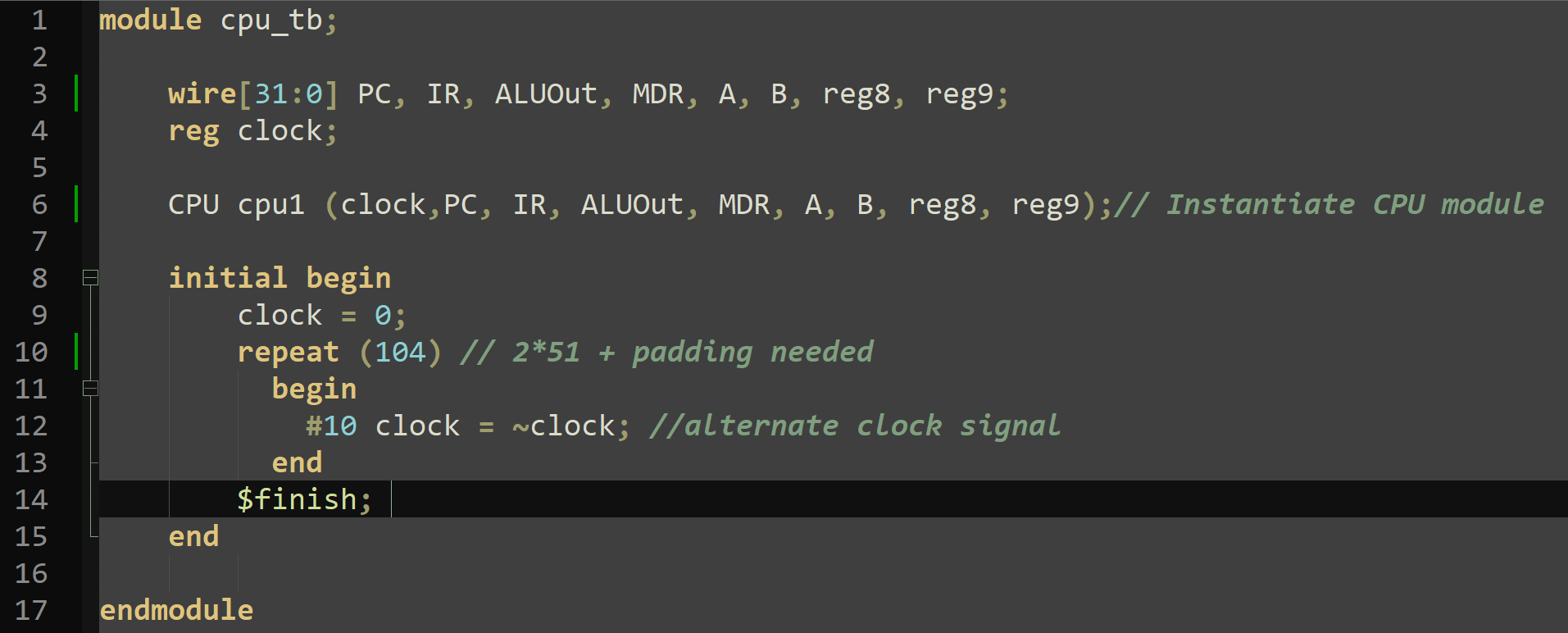
end

$finish;

end

endmodule

Insert snapshot here:



**Simulation Waveforms**

Insert snapshot here





**Detailed Analysis of the Waveforms in Verifying the Correctness of the Design:**

The first simulation picture is a case where the starting number is 8. The addi 8 runs at ~100ns, making $t0 = 8. At ~200ns the sw stores 8 in memory and at ~300ns the lw loads that value into $t1. At ~400ns the add adds 8 to $t1 making it equal to 16 (0x10). At ~500ns the code checks if it should run a branch, but $t0 and $t1 are different so it does not branch. At ~600ns the code swaps $t1 and $t0 making $t0 = 16 and $t1 = 8. At ~700ns the code subtracts 8 from 16 and stores it in $t1 keeping the value at 8. At ~800ns the code branches due to $t1 and $t0 being different and skips the nor operation (branches to NONOR). At ~900ns the code sets the PC to the value stores in memory, 8. The simulation then attempts to restart the code as 8 is the first MIPS instruction step in memory, but we did not include enough cycles for it to loop.

The second simulation picture is a case where the starting number is 0. The addi 0 runs at ~100ns, making $t0 = 0. At ~200ns the sw stores 0 in memory and at ~300ns the lw loads that value into $t1. At ~400ns the add adds 0 to $t1 making it equal to 0. At ~500ns the code checks if it should run a branch and since $t0 equals $t1 it branches to NOSWAP (bne). At ~600ns the code checks if it should branch past the nor operation but does not branch as the values are equal. At ~700ns the code performs the nor operation on $t0 and $t1 and stores the value (0xFFFFFFFF) in $t1. At ~800ns the code sets the PC to the value stores in memory, 0. This simulation does not attempt to run again as the value 0 is before our MIPS instructions stored in memory.

**Part 1**: Summary of Test Results for Each Instruction in Your **Test Program**

Run 1: start with addi 8 (beq does not run, bne runs)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MIPS Instruction** in Your Test Program (show the MIPS Code for each instruction in the order of execution) | **Machine Language Representation** of the Instruction (in Hexadecimal) | **Desired Effect on Registers or Memory**  (for example $t0 – 2 or Mem[20] = 20) | **Actual Effect according to Actual Simulation (Waveforms)**  (for example $t0 – 2 or Mem[20] = 20 | **Did it Run Correctly? (Yes/No)** |
| *Example:*  *addi $t0,$zero,7* | *32’H20080007* | *$t0 = 7* | *$t0 = 7* | *Yes* |
| 1st Instruction:  addi $t0, $zero, 8 | 32'h20080008 | $t0 = 8 | $t0 = 8 | Yes |
| 2nd Instruction:  sw $t0, 124($zero) | 32'hac08007c | Memory[$zero + 124] = $t0 | Memory[$zero + 124] = $t0 | Yes |
| 3rd Instruction:  lw $t1, 124($zero) | 32'h8c09007c | $t1 = Memory[$zero + 124] | $t1 = Memory[$zero + 124] | Yes |
| 4th Instruction:  add $t1, $t0, $t1 | 32'h01094820 | $t1 = $t0 + $t1 = 16 | $t1 = $t0 + $t1 = 16 | Yes |
| 5th Instruction:  beq $t0, $t1, NOSWAP | 32'h11090002 | Not equal, no effect | Not equal, no effect | Yes |
| 6th Instruction:  swap $t0, $t1 | 32'h01094039 | $t0 = $t1 = 16  $t1 = $t0 = 8 | $t0 = $t1 = 16  $t1 = $t0 = 8 | Yes |
| 7th Instruction:  sub $t1, $t0, $t1 | 32'h01094822 | $t1 = $t0 - $t1 = 8 | $t1 = $t0 - $t1 = 8 | Yes |
| 8th Instruction:  bne $t0, $t1, NONOR | 32'h15090001 | Branch to 10th instruction | Branch to 10th instruction | Yes |
| 9th Instruction:  nor $t1, $t0, $t1 | 32'h01094827 | None in this run, skipped by branch | None in this run, skipped by branch | Yes |
| 10th Instruction:  jm mem | 32'hec09007c | PC = Memory[$zero + 124] = 8 | PC = Memory[$zero + 124] = 8 | Yes |

Run 2: start with addi 0 (beq runs, bne does not runs)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MIPS Instruction** in Your Test Program (show the MIPS Code for each instruction in the order of execution) | **Machine Language Representation** of the Instruction (in Hexadecimal) | **Desired Effect on Registers or Memory**  (for example $t0 – 2 or Mem[20] = 20) | **Actual Effect according to Actual Simulation (Waveforms)**  (for example $t0 – 2 or Mem[20] = 20 | **Did it Run Correctly? (Yes/No)** |
| *Example:*  *addi $t0,$zero,7* | *32’H20080007* | *$t0 = 7* | *$t0 = 7* | *Yes* |
| 1st Instruction:  addi $t0, $zero, 0 | 32'h20080000 | $t0 = 0 | $t0 = 0 | Yes |
| 2nd Instruction:  sw $t0, 124($zero) | 32'hac08007c | Memory[$zero + 124] = $t0 | Memory[$zero + 124] = $t0 | Yes |
| 3rd Instruction:  lw $t1, 124($zero) | 32'h8c09007c | $t1 = Memory[$zero + 124] | $t1 = Memory[$zero + 124] | Yes |
| 4th Instruction:  add $t1, $t0, $t1 | 32'h01094820 | $t1 = $t0 + $t1 = 0 | $t1 = $t0 + $t1 = 0 | Yes |
| 5th Instruction:  beq $t0, $t1, NOSWAP | 32'h11090002 | Branch to 8th instruction | Branch to 8th instruction | Yes |
| 6th Instruction:  swap $t0, $t1 | 32'h01094039 | None in this run, skipped by branch | None in this run, skipped by branch | Yes |
| 7th Instruction:  sub $t1, $t0, $t1 | 32'h01094822 | None in this run, skipped by branch | None in this run, skipped by branch | Yes |
| 8th Instruction:  bne $t0, $t1, NONOR | 32'h15090001 | Registers are equal, no effect | Registers are equal, no effect | Yes |
| 9th Instruction:  nor $t1, $t0, $t1 | 32'h01094827 | $t1 = $t0 ~| $t1 = 0xFFFFFFFF | $t1 = $t0 ~| $t1 = 0xFFFFFFFF | Yes |
| 10th Instruction:  jm mem | 32'hec09007c | PC = Memory[$zero + 124] = 0 | PC = Memory[$zero + 124] = 0 | Yes |

Overall Summary of the Project:

The objective of this project was to modify a default 32-bit multi-cycle MIPS CPU (based on the provided lab3\_cpu.v file) to meet specific design requirements. The custom CPU made needed to incorporate a modified instruction execution design with only one read and write port and to support various additional instructions, including further arithmetic capabilities, as well as more custom instructions that required us to expand beyond the MIPS instruction set’s default capabilities and supported functions. The functionality avaiable to the final version of the project are: add (addition), sub (subtraction), addi (immediate addition), nor (bitwise nor), lw (load word from memory), sw (store word in memory), beq (branch if registers are equal), bne (branch if registers are unequal), jm (set the PC to the calue stored in memory), and swap (swap provided registers).

The key changes necessary for the specified implementation were primarily fulfilled by changing the behavioral finite-state-machine Verilog design given to us. Writing a custom sample MIPS program was also needed for testing and validation, which was achieved by working with and analyzing the QTSpim compiler. The jm and swap instructions are not standard MIPS instructions and were therefore programmed with functionality by hand and the corresponding machine code for the MIPS instructions were also converted by hand.

Conclusion: Our project met all requirements by modifying the default MIPS multi-cycle CPU template provided to us. The custom CPU design file now supports a the full wider range of operations required and demonstrates the capability for the functionality of the design to further increased in the future. It could potentially be made more pipeline-able by further granulating the instruction execution into smaller discrete events. The programmed changes fulfilled the project criteria and improved our understanding of basic MIPS architecture design.