UNIVERSITY OF WATERLOO AQ
Faculty of Engineering
Department of Electrical and Computer Engineering
ECE 621- Computer Organization

Register File and Execute Stage

Group 2
Jordan Ross, Mustafa Faraj
j25ross, mabdulaz
31 October, 2014

ALU Module Code

This section contains the code for our ALU module. It takes in the two source operands as well as an operation control signal (set in the decode stage). It outputs the result of the arithmetic operation as well as set two condition bits (zero and negative).

```
module alu(
	input [31:0] op1, // operand 1 (always from rs)
	input [31:0] op2, // operand 2
	input [5:0] operation, // The arithmatic operation to perform
	input [5:0] shift_amount, // The number of bits to shift
	output reg [31:0] result, // The result based on the operation
	output reg zero, // Indicates if the result of the operation is zero.
	output reg neg // Indicates if the result is negative or not.
);
	// The operations that the ALU supports are:
	// 0: Addition
	// 1: Subtraction
	// 2: Multiply
	// 3: Divide
	// 4: Shift logical left
	// 5: Shift logical right
	// 6: Set on less than
	// 7: And
	// 8: Or
	// 9: Xor
	// 10: Nor
	// 11: SRA
	// 12: Load upper

	always @(op1 or op2 or operation or shift_amount) begin
		case (operation)
			0: result = op1+op2;
			1: result = op1-op2;
			2: result = op1*op2;
			3: begin
				if (op2 != 0) begin
					result = op1/op2;
				end
				// TODO: Otherwise we should trap here
			end
			4: result = op2 << shift_amount;
			5: result = op2 >> shift_amount;
			6: result = op1 < op2 ? 1 : 0;
			7: result = op1 & op2;
			8: result = op1 | op2;
			9: result = op1 ^ op2;
			10: result = ~(op1 | op2);
			11: result = op2 >>> shift_amount;
			12: result = (op2 << 16)&32'hffff0000;
		endcase
		zero = result == 0 ? 1 : 0;
		neg = result[31];
	end
endmodule
```

Branch Resolve Unit

This section contains our logic for selecting whether a branch should be taken or not based on the two condition bits set by the ALU.

```
module branch_resolve(
	input zero,
	input neg,
	input [1:0]branch_type,
	input is_branch,
	output reg branch_taken // Indicates if the branch is taken or not
);
	always @(zero or neg or branch_type or is_branch) begin
		if (is_branch == 1'b1) begin
			case (branch_type)
				0: branch_taken = zero == 0 ? 1 : 0; // BEQ
				1: branch_taken = zero != 0 ? 1 : 0; // BNE
				2: branch_taken = (neg | zero) == 0 ? 1 : 0; // BLEZ
				3: branch_taken = neg != 0 ? 1 : 0; // BGTZ
			endcase
		end else begin
			branch_taken = 0;
		end
	end
endmodule
```

Processor Module Code

This section contains the code for the processor module. The processor module is our top level module where we instantiate our fetch, decode, memory, register file, and ALU so far. This is also where we have our control signals being set. The control is only being done in the decode stage and the control signals that are needed in later stages are latched in pipeline registers. The processor module currently has the IF/ID and ID/IX pipeline registers implemented.

```
module processor(
    input clk,  // The system clock
    input srec_parse    // If the SREC parser is active or not.
);
    // Decoder signals
    wire [4:0]rs;
    wire [4:0]rt;
    wire [4:0]rd;
    wire [4:0]sha;
    wire [5:0]func;
    wire [15:0]immed;
    wire [25:0]target;
    wire [5:0]opcode;
    wire [31:0]pc_out;
    wire [31:0]insn_out;

    // Control signals
    reg stall;
    wire [1:0]insn_access_size;
    wire [1:0]fetch_access_size;
```

```
wire fetch_rw;
wire insn_rw;
reg reg_file_write_enable;

// Address lines
wire [31:0]pc;

// Data lines
wire [31:0]insn_data_out;
wire [31:0]insn_address;

// SREC registers (for helping the parser write to instruction memory)
reg [31:0]srec_address;
reg [31:0]srec_data_in;
reg srec_rw;
reg [1:0]srec_access_size;

// Decode wires
wire [31:0]decode_pc; // the PC for next instruction to be executed
wire [31:0]decode_ir; // the current instruction being decoded
wire [31:0]dec_A;
wire [31:0]dec_B;
wire [4:0]dest_reg;

// Decode control signals
reg dest_reg_sel;
reg dec_illegal_insn;
reg [5:0] dec_alu_op;
reg dec_is_branch;
reg dec_op2_sel;
reg [5:0] dec_shift_amount;
reg [1:0]dec_branch_type; // 0-BEQ, 1-BNE, 2-BLEZ, 3-BGTZ
reg dec_is_jump;

// Execute wires
wire [31:0]exe_pc; // the PC for next instruction to be executed
wire [31:0]exe_ir; // the current instruction being executied
wire [31:0]exe_A;
wire [31:0]exe_B;
  wire [31:0]exe_op2;
wire [31:0]exe_O;
wire [31:0]exe_extended; // The wire the comes for the sign extender
wire exe_zero;
wire [5:0] exe_alu_op;
wire exe_is_branch;
wire exe_op2_sel;
wire [5:0] exe_shift_amount;
wire [31:0] exe_shift_immed;
wire [31:0] exe_shift_target;
wire [31:0] exe_jump_effective_address;
wire [31:0] exe_branch_effective_address;
wire exe_neg;
wire [1:0]exe_branch_type;
wire exe_branch_taken;
wire [31:0]exe_next_pc;
wire [31:0]exe_branch_next_pc;
wire exe_is_jump;
```

```
    // Instantiate mux's for each of the SREC registers to aid the SREC
parser.
    mux_2_1_32_bit srec_insn_address_mux(
        .line0(pc),
        .line1(srec_address),
        .select(srec_parse),
        .output_line(insn_address)
    );
    mux_2_1_1_bit srec_insn_rw_mux(
        .line0(fetch_rw),
        .line1(srec_rw),
        .select(srec_parse),
        .output_line(insn_rw)
    );
    mux_2_1_2_bit srec_insn_access_size_mux(
        .line0(fetch_access_size),
        .line1(srec_access_size),
        .select(srec_parse),
        .output_line(insn_access_size)
    );


    // Instantiate the fetch module
    fetch fetch(
        .clk_in(clk),
        .stall_in(stall),
        .pc_out(pc),
        .rw_out(fetch_rw),
        .access_size_out(fetch_access_size)
    );

    // Instantiate the instruction memory module
    memory insn_memory(
        .data_out(insn_data_out),
        .address(insn_address),
        .data_in(srec_data_in), // We can tie the srec_data_in wire to this
port since we should never be writing to instruction memory unless we are
srec parsing
        .write(insn_rw),
        .clk(clk),
            .access_size(insn_access_size)
    );

    // Instatiate the IF/ID pipeline register to kep the PC and IR
    if_id_pipleline_reg if_id_pipleline_reg(
        .clk(clk),
        .pc_in(pc),
        .ir_in(insn_data_out),
        .pc_out(decode_pc),
        .ir_out(decode_ir)
    );


    // Instantiate a mux for selecting which destination to choose
    mux_2_1_5_bit dest_reg_mux(
```

```
    .line0(rt),
    .line1(rd),
    .select(dest_reg_sel), // TODO: Implement this control singal
    .output_line(dest_reg)
);

// Instantiate the register file
reg_file reg_file(
    .clk(clk),
    .write_enable(reg_file_write_enable),
    .source1(rs),
    .source2(rt),
    .dest(dest_reg),
    .destVal(exe_O), // TODO: Change this to be the output from WB stage
    .s1val(dec_A),
    .s2val(dec_B)
);

// Instantiate the decode module
decode decoder(
    .clk(clk),
    .stall(stall),
    .insn_in(decode_ir),
    .pc_in(pc),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .sha(sha),
    .func(func),
    .immed(immed),
    .target(target),
    .opcode(opcode),
    .pc_out(pc_out),
    .insn_out(insn_out)
);

// Instatiate the ID/IX pipeline register
id_ix_pipleline_reg id_ix_pipleline_reg(
    .clk(clk),
    .pc_in(decode_pc),
    .ir_in(decode_ir),
    .A_in(dec_A),
    .B_in(dec_B),
    .alu_op_in(dec_alu_op),
    .is_branch_in(dec_is_branch),
    .is_jump_in(dec_is_jump),
    .op2_sel_in(dec_op2_sel),
    .shift_amount_in(dec_shift_amount),
    .branch_type_in(dec_branch_type),
    .pc_out(exe_pc),
    .ir_out(exe_ir),
    .A_out(exe_A),
    .B_out(exe_B),
    .alu_op_out(exe_alu_op),
    .is_branch_out(exe_is_branch),
    .is_jump_out(exe_is_jump),
    .op2_sel_out(exe_op2_sel),
```

```
        .shift_amount_out(exe_shift_amount),
        .branch_type_out(exe_branch_type)
    );

    sign_extender sign_extender(
        .in_data(exe_ir[15:0]),
        .out_data(exe_extended)
    );

    assign exe_shift_target = exe_ir[25:0];
    assign exe_jump_effective_address = (exe_pc & 32'hf0000000) |
((exe_shift_target << 2) & 32'h0fffffff);
    assign exe_shift_immed = exe_extended << 2;
    assign exe_branch_effective_address = exe_shift_immed + exe_pc;

    // Instantiate a 32-bit mux for selecting which operand to provide to op2
of the ALU
    mux_2_1_32_bit alu_op2_sel_mux(
        .line0(exe_B),
        .line1(exe_extended),
        .select(exe_op2_sel),
        .output_line(exe_op2)
    );

    alu alu(
        .op1(exe_A), // operand 1 (always from rs)
        .op2(exe_op2), // operand 2
        .operation(exe_alu_op), // The arithmatic operation to perform
        .shift_amount(exe_shift_amount), // The number of bits to shift
        .result(exe_O), // The arithmatic result based on the operation
        .zero(exe_zero), // Indicates if the result of the operation is zero.
        .neg(exe_neg)
    );

    branch_resolve branch_resolve(
      .zero(exe_zero),
            .neg(exe_neg),
            .branch_type(exe_branch_type),
            .is_branch(exe_is_branch),
            .branch_taken(exe_branch_taken) // Indicates if the branch is
taken or not
    );

    mux_2_1_32_bit branch_next_pc_mux(
        .line0(exe_pc),
        .line1(exe_branch_effective_address),
        .select(exe_branch_taken),
        .output_line(exe_branch_next_pc)
    );

    mux_2_1_32_bit jump_next_pc_mux(
        .line0(exe_branch_next_pc),
        .line1(exe_jump_effective_address),
        .select(exe_is_jump),
        .output_line(exe_next_pc)
    );
```

```
// Control
always @(posedge clk) begin
    reg_file_write_enable = 0;

    // ----------- Decode Stage Control Signal Logic ---------------- //
    dec_illegal_insn = 0;
    dest_reg_sel = 0;
    dec_alu_op = 0;
    dec_op2_sel = 0;
    dec_is_branch = 0;
    dec_is_jump = 0;
    dec_branch_type = 0;
    if (((opcode & 6'b111000)>> 3) == 3'h0) begin
        if ((opcode & 6'b000111) == 3'h0) begin
              // We are in the SPECIAL Opcode encoding table
              // This is an R-type instruction
              dest_reg_sel = 0;
              if (func == 6'b100000 || func == 6'b100001) // ADD, ADDU
                    dec_alu_op = 0; // Do an add operation
              else if (func == 6'b100010 || func == 6'b100011)//SUB, SUBU
                    dec_alu_op = 1;
              else if (func == 6'b011000 || func==6'b011001)//MULT, MULTU
                    dec_alu_op = 2;
              else if (func == 6'b011010 || func==6'b011011) // DIV, DIVU
                    dec_alu_op = 3;
              else if (func == 6'b000000) begin // SLL
                    dec_alu_op = 4;
                    dec_shift_amount = sha;
              end
              else if (func == 6'b000010) begin // SLL
                    dec_alu_op = 5;
                    dec_shift_amount = sha;
              end
              else if (func == 6'b101010 || func==6'b101011) // SLT, SLTU
                    dec_alu_op = 6;
             else begin
               dec_illegal_insn = 1;
             end
        end else if (((opcode & 6'b000111)  == 3'd2) || ((opcode &
            6'b000111) == 3'd3)) begin
              // J and JAL instructions
              // This is J-type instruction
              dest_reg_sel = 0;
              dec_is_jump = 1;
              if (opcode == 6'b000011) begin
                    // JAL instruction
                    dec_illegal_insn = 1;
                    // TODO: store the return address into reg[31]
              end
        end else begin
              // BEQ, BNE, BLEZ, BGTZ
              // This is an I-type instruction
              dec_is_branch = 1;
              if (opcode == 6'b000100) begin
                    // BEQ
                    dec_alu_op = 1; // We want to test if the result is z
                                    zero from a subtract
```

```
                dec_branch_type = 0;
            end else if (opcode == 6'b000101) begin
                // BNE
                dec_alu_op = 1; // We want to do a test if the result
                is zero from a subtract
                dec_branch_type = 1;
            end else if (opcode == 6'b000110) begin
                // BLEZ
                // TODO: Test that rt has zero in it
                dec_illegal_insn = 1;
                dec_alu_op = 0;
                dec_branch_type = 2;
            end else begin
                // BGTZ
                // TODO: Test that rt has zero in it
                dec_illegal_insn = 1;
                dec_alu_op = 0;
                dec_branch_type = 3;
            end
        end
    end else if (((opcode & 6'b111000)>> 3) == 3'd1) begin
        // ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI
        // This is an I-type instruction
        case (opcode)
            6'b001000: dec_alu_op = 0;  // ADDI
            6'b001001: dec_alu_op = 0;  // ADDIU
            6'b001010: dec_alu_op = 6;  // SLTI
            6'b001011: dec_alu_op = 6;  // SLTUI
            6'b001100: dec_alu_op = 7;  // ANDI
            6'b001101: dec_alu_op = 8;  // ORI
            6'b001110: dec_alu_op = 9;  // XORI
            6'b001111: dec_alu_op = 12; // LUI
        endcase
        dest_reg_sel = 1;
        dec_op2_sel = 1; // We want op2 to be the immediate in the ALU
    end else if (((opcode & 6'b111000)>> 3) == 3'd3) begin
        dest_reg_sel = 0;
        dec_illegal_insn = 1;
    end else if (((opcode & 6'b111000)>> 3) == 3'd4) begin
        // LB, LH, LWL, LW, LBU, LHU, LWR
        // This is an I-type instruction
        dest_reg_sel = 1;
        dec_op2_sel = 1;
        dec_alu_op = 0; // add the value of rs to the immediate
    end else if (((opcode & 6'b111000)>> 3) == 3'd5) begin
        // SB, SH, SWL, SW, SWR
            // This is an I-type instruction
        dest_reg_sel = 1;
        dec_op2_sel = 1;
        dec_alu_op = 0; //add the value of rs to the immediate
    end else begin
        dec_illegal_insn = 1;
    end

end

endmodule
```

Processor Test Bench

We did not change anything in our test bench actually but the code is shown here again for reference.

```
module processor_tb;

    reg clk;
    reg srec_parse; // enable signal for srec parser.

    // Registers, writes, and variables for parser
    integer fh = 0; // file handler for input
    integer i = 0; // loop variable
    integer data_byte = 0; // variable to keep track of what byte we are on.
    integer data_offset = 0; // keep track of the offset from the data
                               address to write the next byte.
    reg [1:0]nibble_count = 0; // keep track of which nibble is being written
    reg [7:0]rec_type; // record type number
    reg [7:0] byte_count; //byte count for address, data, and checksum
    integer record_code; // A record_code is equivalent to 1 ASCII
                         digit/letter in the .srec file
    reg [31:0] rec_address = 'b0; // the address given by the record.
    integer highest_address = 0; // highest address written by parser
    reg [7:0]rec_data; // A single byte of the data from the record.
    reg [7:0]temp; // a temporary byte used for place holding.
    reg done = 0; // this will set high when we are done parsing the file.
    reg [7:0] file_char = 8'h0A; // Set the initial character from the file
                                   read to be the new line character


    // Instantiate the processor as the unit under test.
    processor processor_uut(.clk(clk), .srec_parse(srec_parse));

    initial begin

        //-------------------------------------------------------
        // Parsing stage of testbench - memory is not valid until
        // after the parser has finished!
        //-------------------------------------------------------

            //SREC file parsing has been omitted from the report for clarity.

        // -------------------------------------------------------
        // Memory is ready to be used after this point!
        // -------------------------------------------------------

        $monitor("Beginning the fetch-decode-execute loop!");
        #100;
        // Deassert stall just read out the pc, rw, and access size.
        processor_uut.stall = 0;


        // -------------------------------------------------------
        // This section prints out the decoded instructions
        // -------------------------------------------------------

        while (processor_uut.pc <= highest_address - 4) begin
            @(posedge clk);
```

```
case(processor_uut.opcode)
    6'd0:  begin //JR, ADD, ADDU, SUB SUBU, DIV, SLT, SLTU, SLL,
                 SRL, SRA, AND, OR, XOR, NOR, NOP
        case(processor_uut.func)
        6'd0: begin //SLL, NOP
            if (processor_uut.sha == 5'b0) $strobe("%h:    %h
             NOP", processor_uut.pc_out, processor_uut.insn_out);
            else $strobe("%h:    %h    sll  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rt,
                processor_uut.sha);
        end
        6'd2: begin //SRL
            $strobe("%h:    %h    srl  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rt,
                processor_uut.sha);
        end
        6'd3: begin //SRA
            $strobe("%h:    %h    sra  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rt,
                processor_uut.sha);
        end
        6'd8: begin //JR
            $strobe("%h:    %h    jr  %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rs);
        end
        6'd26: begin //DIV
            $strobe("%h:    %h    div  %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rs, processor_uut.rt);
        end
        6'd32: begin //ADD
            $strobe("%h:    %h    add  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd33: begin //ADDU
            $strobe("%h:    %h    addu  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd34: begin //SUB
            $strobe("%h:    %h    sub  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd35: begin //SUBU
            $strobe("%h:    %h    subu  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
```

```
        end
        6'd36: begin //AND
            $strobe("%h:    %h    and  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd37: begin //OR
            $strobe("%h:    %h    or  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd38: begin //XOR
            $strobe("%h:    %h    xor  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd39: begin //NOR
            $strobe("%h:    %h    nor  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd42: begin //SLT
            $strobe("%h:    %h    slt  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        6'd43: begin //SLTU
            $strobe("%h:    %h    sltu  %h, %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rd, processor_uut.rs,
                processor_uut.rt);
        end
        default: begin
            $strobe("ERROR! %h:    %h", processor_uut.pc_out,
                processor_uut.insn_out);
            $stop;
        end
        endcase
    end
    6'd1: begin //BLTZ, BGEZ
        case(processor_uut.rt)
        5'd0: begin //BLTZ
            $strobe("%h:    %h    bltz  %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rs, processor_uut.immed);
        end
        5'd1: begin //BGEZ
            $strobe("%h:    %h    bgez  %h, %h",
                processor_uut.pc_out, processor_uut.insn_out,
                processor_uut.rs, processor_uut.immed);
        end
        default: begin
```

```
        $strobe("ERROR! %h:    %h", processor_uut.pc_out,
            processor_uut.insn_out);
        $stop;
    end
    endcase
end
6'd2: begin //J
    //display the destination address of the jump, not the
    offset.
    $strobe("%h:    %h    j  %h",
            processor_uut.pc_out, processor_uut.insn_out,
             {processor_uut.pc_out[31-:4],
             28'b0}+{processor_uut.target,2'b0});
end
6'd3: begin //JAL
    //display the destination address of the jump, not the
    offset.
    $strobe("%h:    %h    jal  %h",
            processor_uut.pc_out, processor_uut.insn_out,
             {processor_uut.pc_out[31-:4],
             28'b0}+{processor_uut.target,2'b0});
end
6'd4: begin //BEQ
    $strobe("%h:    %h    beq  %h, %h, %h",
            processor_uut.pc_out, processor_uut.insn_out,
             processor_uut.rs, processor_uut.rt,
             processor_uut.immed);
end
6'd5: begin //BNE
    $strobe("%h:    %h    bne  %h, %h, %h",
            processor_uut.pc_out, processor_uut.insn_out,
             processor_uut.rs, processor_uut.rt,
             processor_uut.immed);
end
6'd6: begin //BLEZ
    $strobe("%h:    %h    beq  %h, %h",
            processor_uut.pc_out, processor_uut.insn_out,
             processor_uut.rs, processor_uut.immed);
end
6'd7: begin //BGTZ
    $strobe("%h:    %h    bgtz  %h, %h",
            processor_uut.pc_out, processor_uut.insn_out,
             processor_uut.rs, processor_uut.immed);
end
6'd9: begin //ADDIU
    $strobe("%h:    %h    addiu  %h, %h, %d",
            processor_uut.pc_out, processor_uut.insn_out,
             processor_uut.rt, processor_uut.rs,
             processor_uut.immed);
end
6'd10: begin //SLTI
    $strobe("%h:    %h    slti  %h, %h, %h",
            processor_uut.pc_out, processor_uut.insn_out,
             processor_uut.rt, processor_uut.rs,
             processor_uut.immed);
end
6'd13: begin //ORI
```

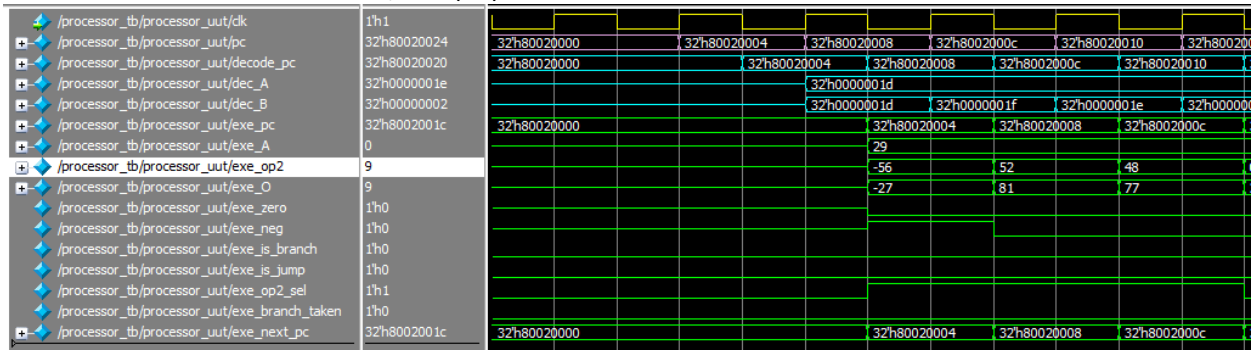
```
            $strobe("%h:    %h    ori  %h, %h, %h",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.rs,
                    processor_uut.immed);
        end
        6'd15: begin //LUI
            $strobe("%h:    %h    lui  %h, %h",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.immed);
        end
        6'd28: begin //MUL
            $strobe("%h:    %h    mul  %h, %h, %h",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rd, processor_uut.rs,
                    processor_uut.rt);
        end
        6'd32: begin //LB
            $strobe("%h:    %h    lb  %h, %d(%h)",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.immed,
                    processor_uut.rs);
        end
        6'd35: begin //LW
            $strobe("%h:    %h    lw  %h, %d(%h)",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.immed,
                    processor_uut.rs);
        end
        6'd36: begin //LBU
            $strobe("%h:    %h    lbu  %h, %d(%h)",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.immed,
                    processor_uut.rs);
        end
        6'd40: begin //SB
            $strobe("%h:    %h    sb  %h, %d(%h)",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.immed,
                    processor_uut.rs);
        end
        6'd43: begin //SW
            $strobe("%h:    %h    sw  %h, %d(%h)",
                    processor_uut.pc_out, processor_uut.insn_out,
                    processor_uut.rt, processor_uut.immed,
                    processor_uut.rs);
        end
        default: begin
            $strobe("ERROR! %h:    %h", processor_uut.pc_out,
                    processor_uut.insn_out);
            $stop;
        end
    endcase
end

#100;
$stop;
```

```
    end

      always begin
            #50 clk = !clk;
      end

    // A function to convert ASCII upper case letters and digits to their
    hexadecimal value.
    function [7:0]atoh;
        input [7:0]aCode;
        begin
            if (aCode >= 8'h30 && aCode <= 8'h39) begin
                atoh = aCode - 8'h30;
            end else if (aCode >= 8'h41 && aCode <= 8'h5A) begin
                atoh = aCode - 8'h37;
            end
        end
    endfunction


endmodule
```

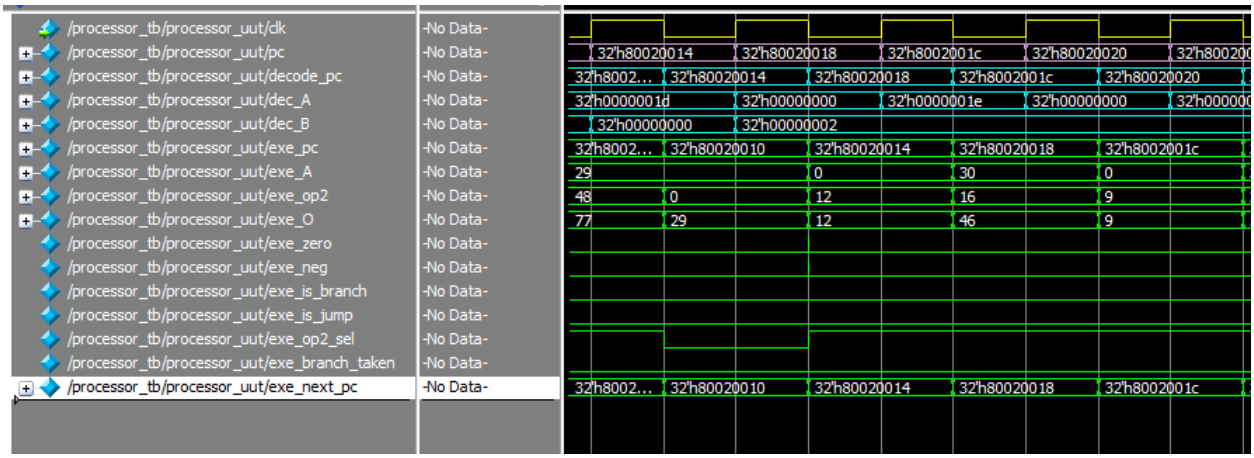
Verification Demonstration

For the verification the setup for us that it is easiest to show is the waveform diagram as the program executes. For readability purposes we have only shown 4 instructions per waveform along with the instruction printout we get from our test bench (this was verified last PD). We show the operands to the ALU, and output from the ALU as decimal to improve readability. The plum colored wave forms are for the fetch stage while the cyan ones are for decode and the green for execute. We executed the bubble sort bench mark in the following diagrams.

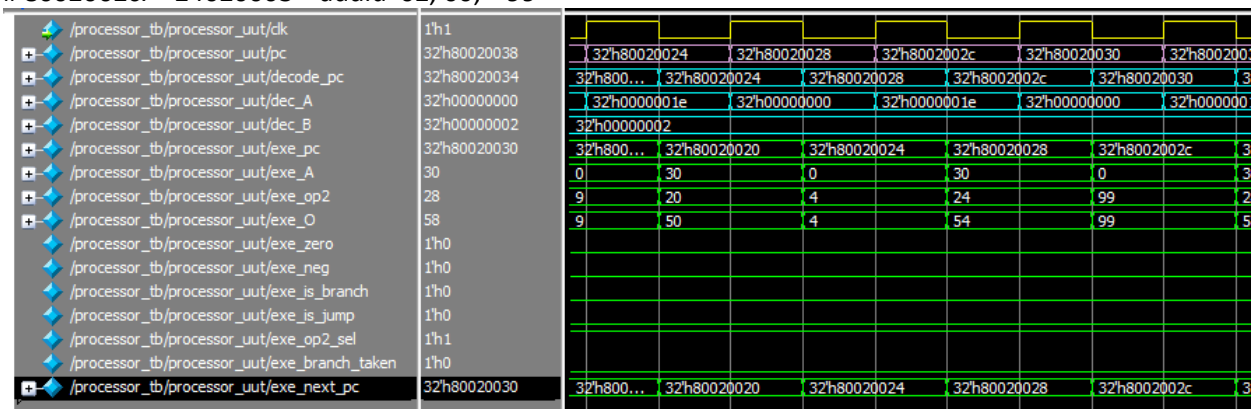
80020004: 27bdffc8 addiu 1d, 1d, 65480
80020008: afbf0034 sw 1f, 52(1d)
8002000c: afbe0030 sw 1e, 48(1d)



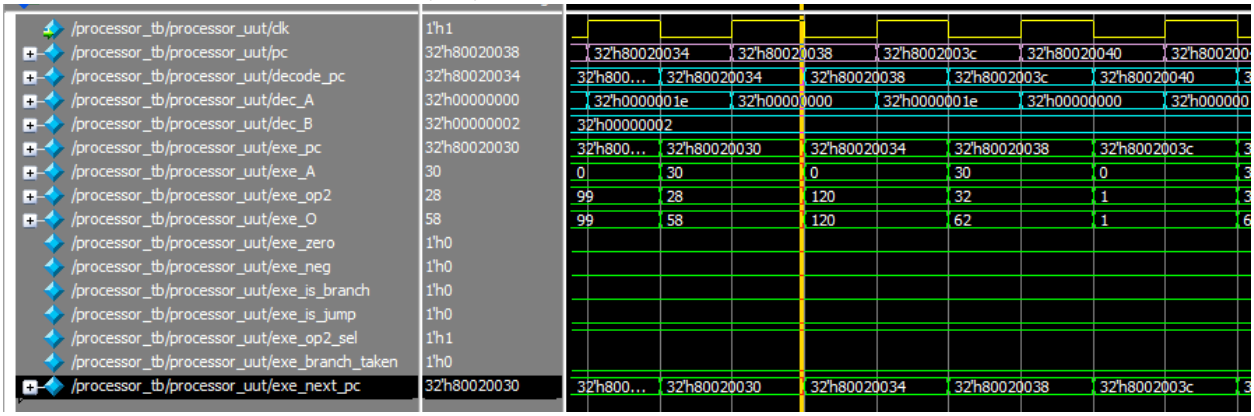
80020010: 03a0f021 addu 1e, 1d, 00
80020014: 2402000c addiu 02, 00, 12
80020018: afc20010 sw 02, 16(1e)
8002001c: 24020009 addiu 02, 00, 9



80020020: afc20014 sw 02, 20(1e)
80020024: 24020004 addiu 02, 00, 4
80020028: afc20018 sw 02, 24(1e)
8002002c: 24020063 addiu 02, 00, 99

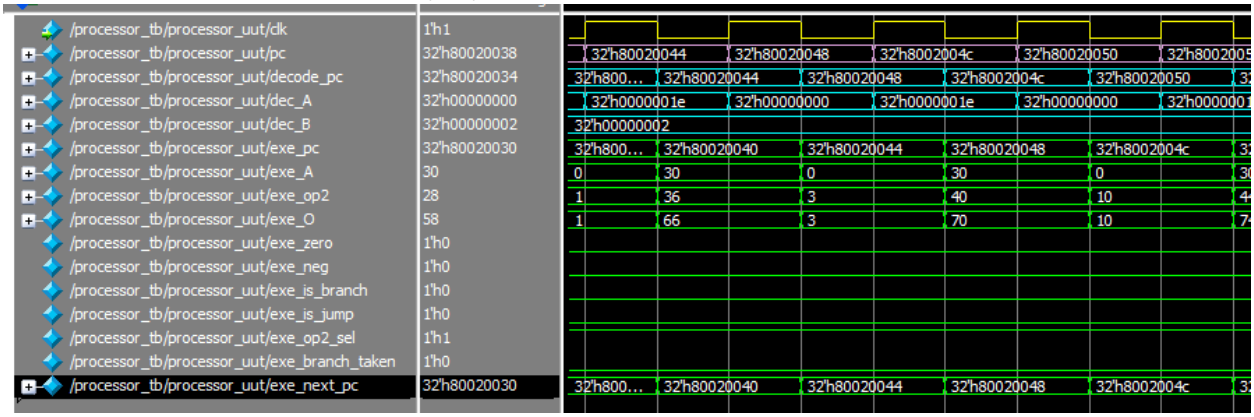


80020030: afc2001c sw 02, 28(1e)
80020034: 24020078 addiu 02, 00, 120
80020038: afc20020 sw 02, 32(1e)
8002003c: 24020001 addiu 02, 00, 1

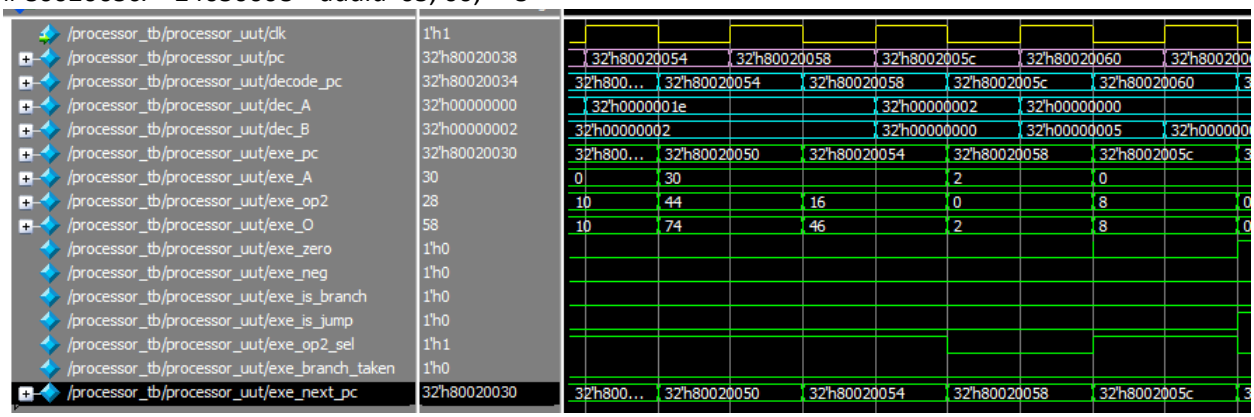


80020040: afc20024 sw 02, 36(1e)

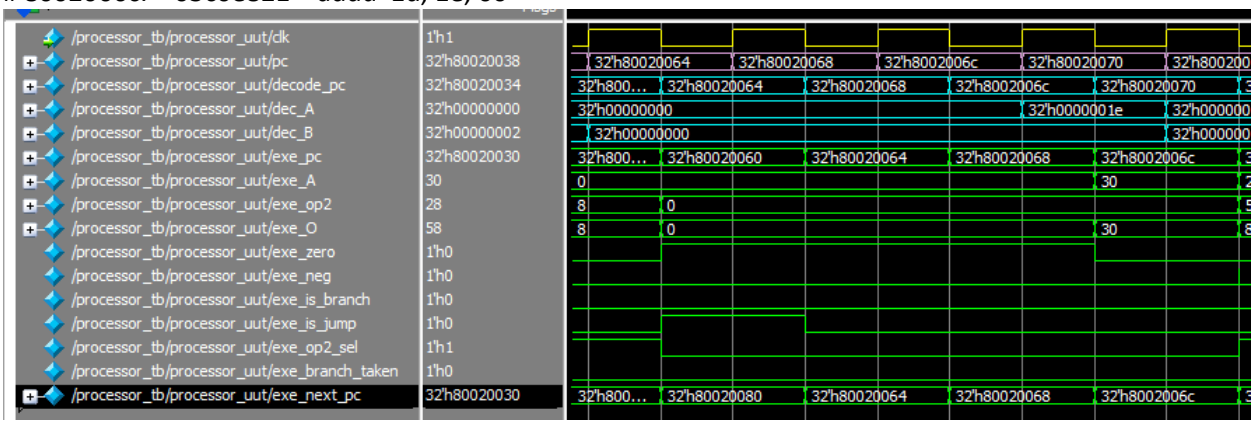
80020044: 24020003 addiu 02, 00, 3
80020048: afc20028 sw 02, 40(1e)
8002004c: 2402000a addiu 02, 00, 10



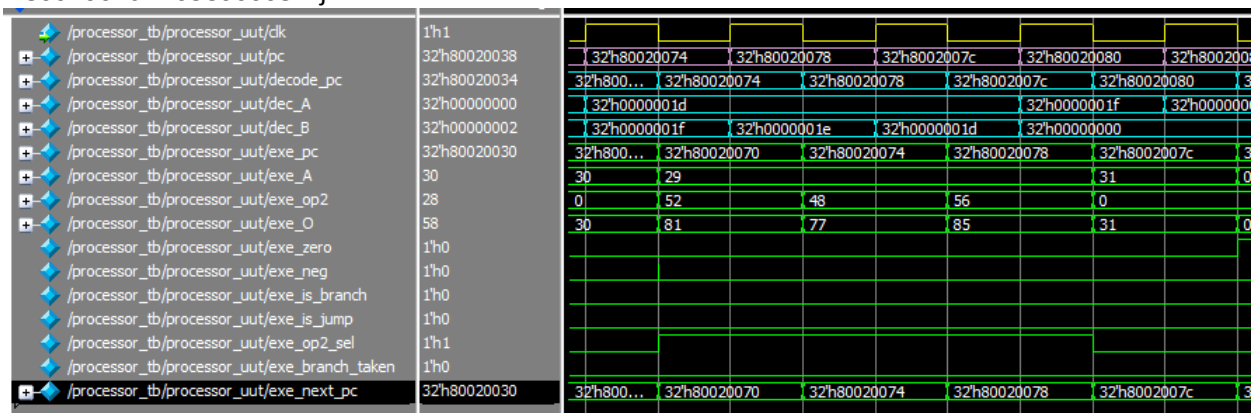
80020050: afc2002c sw 02, 44(1e)
80020054: 27c20010 addiu 02, 1e, 16
80020058: 00402021 addu 04, 02, 00
8002005c: 24050008 addiu 05, 00, 8



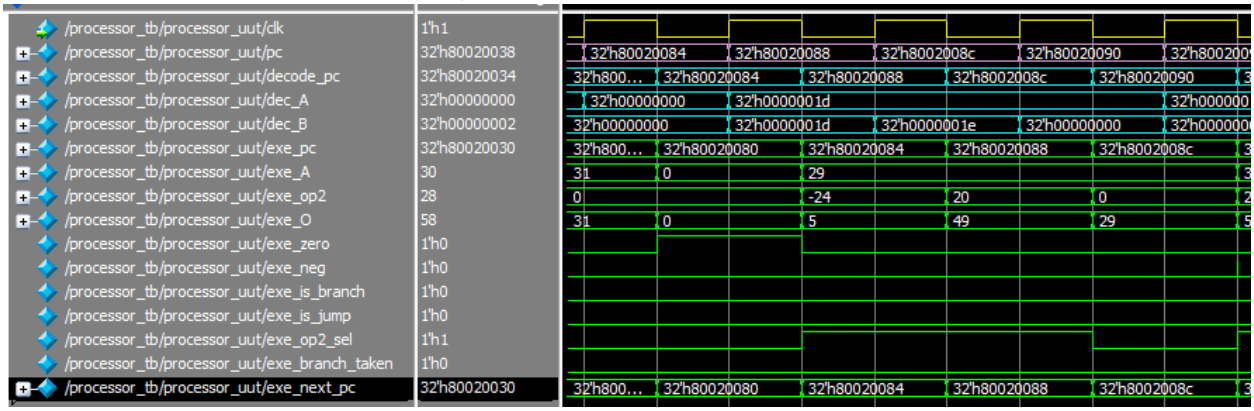
80020060: 0c008020 jal 80020080
80020064: 00000000 NOP
80020068: 00001021 addu 02, 00, 00
8002006c: 03c0e821 addu 1d, 1e, 00



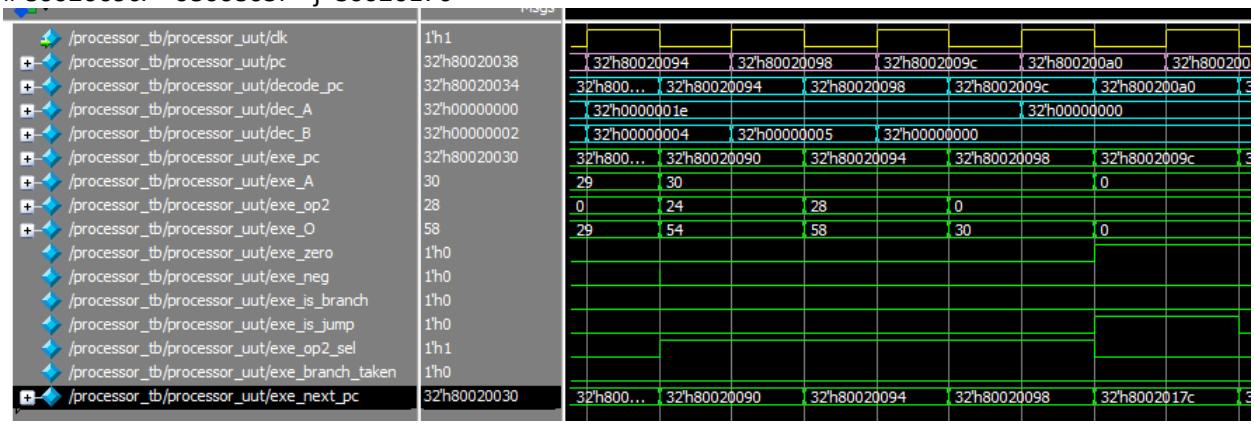
80020070: 8fbf0034 lw 1f, 52(1d)
80020074: 8fbe0030 lw 1e, 48(1d)
80020078: 27bd0038 addiu 1d, 1d, 56
8002007c: 03e00008 jr 1f



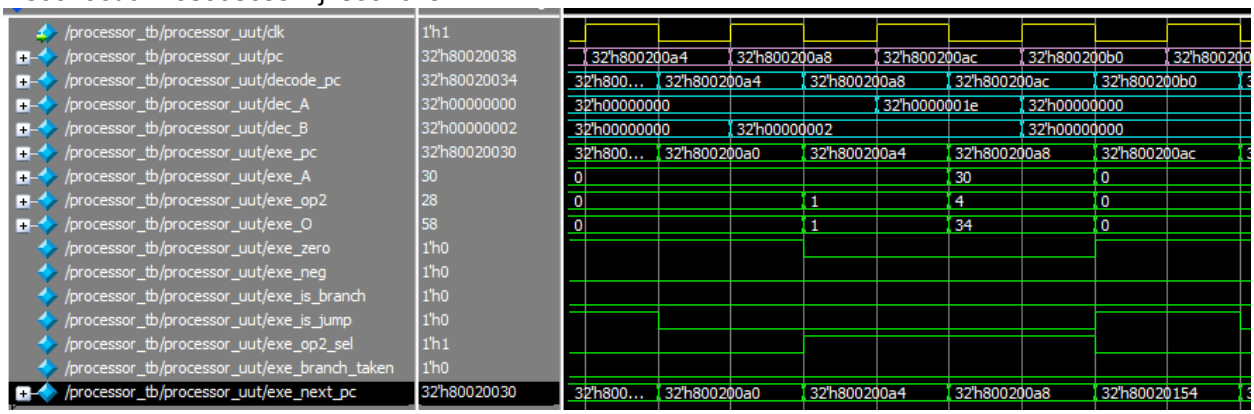
80020080: 00000000 NOP
80020084: 27bdffe8 addiu 1d, 1d, 65512
80020088: afbe0014 sw 1e, 20(1d)
8002008c: 03a0f021 addu 1e, 1d, 00



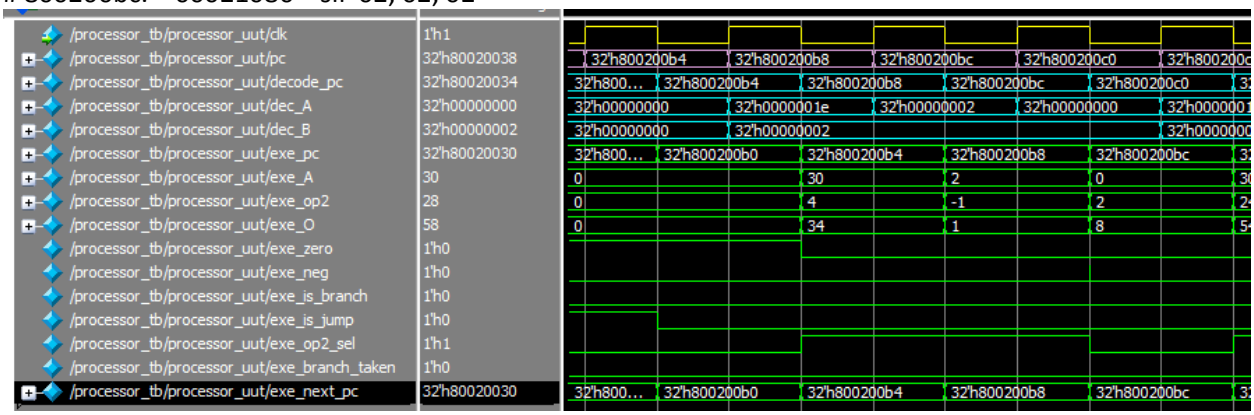
80020090: afc40018 sw 04, 24(1e)
80020094: afc5001c sw 05, 28(1e)
80020098: afc00000 sw 00, 0(1e)
8002009c: 0800805f j 8002017c



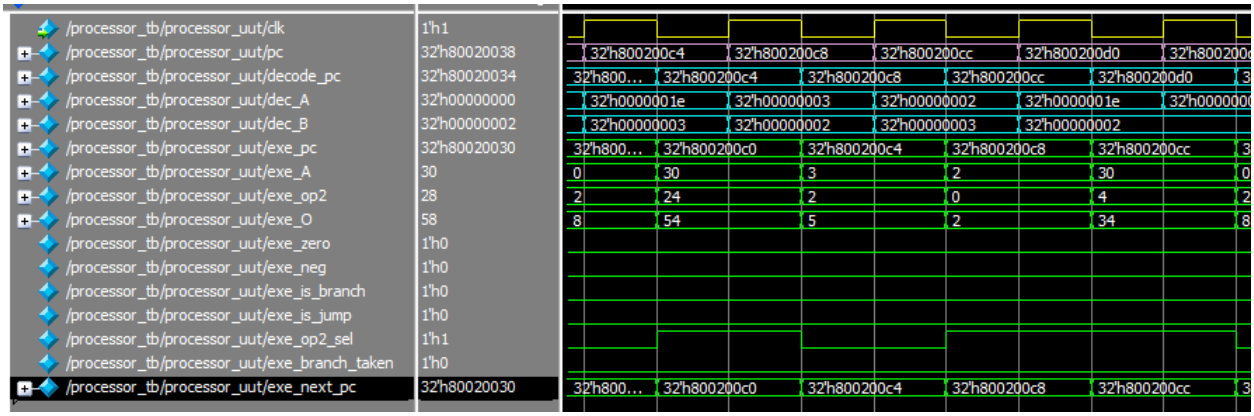
800200a0: 00000000 NOP
800200a4: 24020001 addiu 02, 00, 1
800200a8: afc20004 sw 02, 4(1e)
800200ac: 08008055 j 80020154



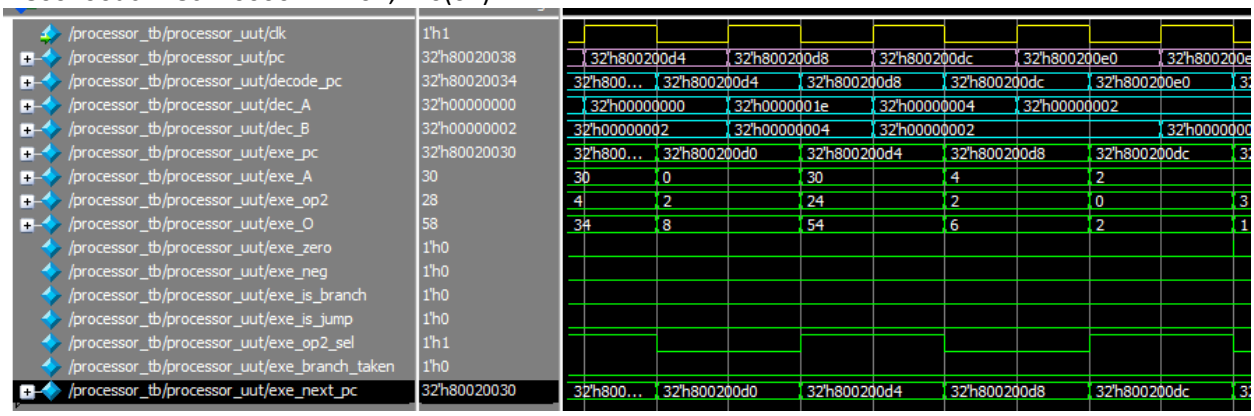
800200b0: 00000000 NOP
800200b4: 8fc20004 lw 02, 4(1e)
800200b8: 2442ffff addiu 02, 02, 65535
800200bc: 00021080 sll 02, 02, 02



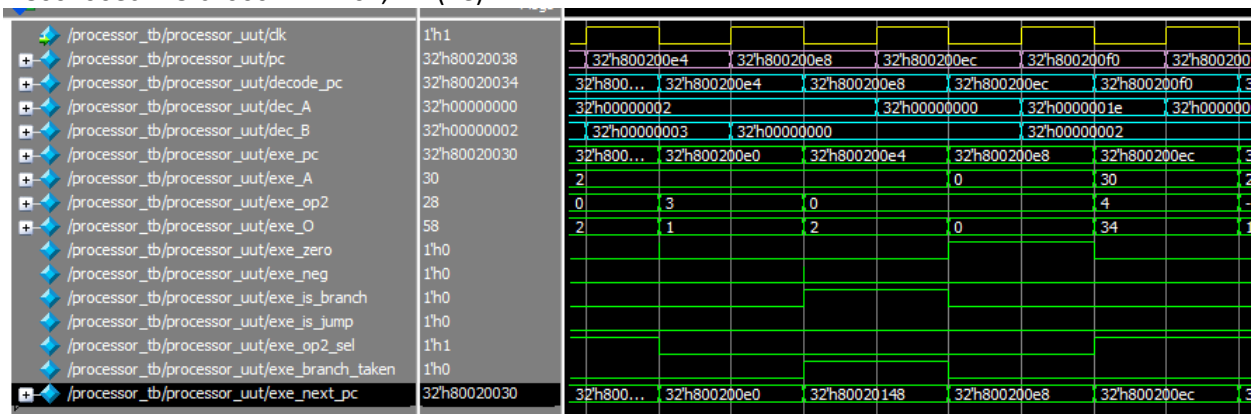
800200c0: 8fc30018 lw 03, 24(1e)
800200c4: 00621021 addu 02, 03, 02
800200c8: 8c430000 lw 03, 0(02)
800200cc: 8fc20004 lw 02, 4(1e)



800200d0: 00021080 sll 02, 02, 02
800200d4: 8fc40018 lw 04, 24(1e)
800200d8: 00821021 addu 02, 04, 02
800200dc: 8c420000 lw 02, 0(02)

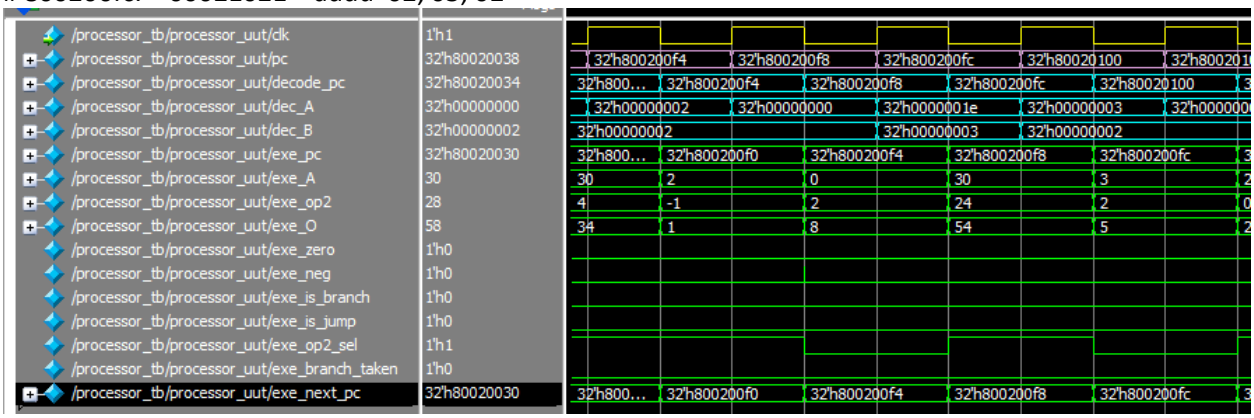


800200e0: 0043102a slt 02, 02, 03
800200e4: 10400019 beq 02, 00, 0019
800200e8: 00000000 NOP
800200ec: 8fc20004 lw 02, 4(1e)

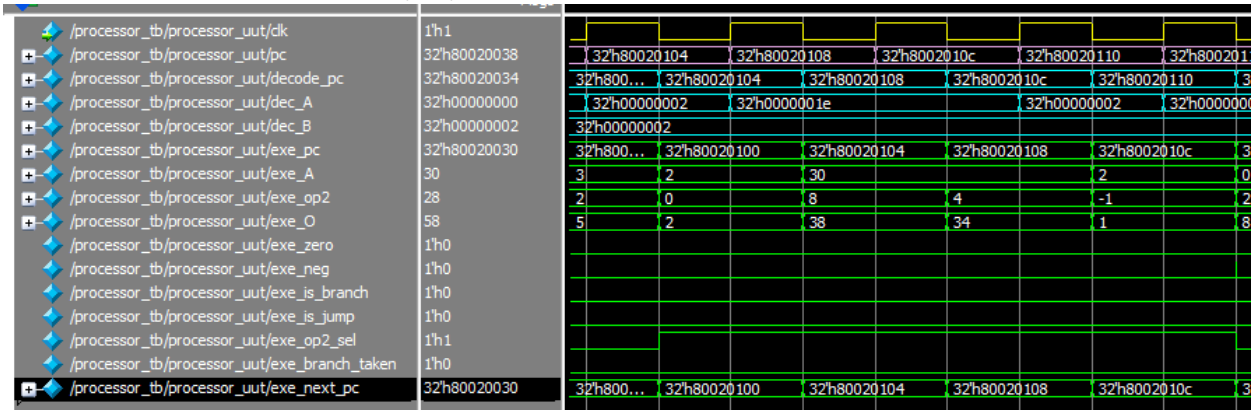


800200f0: 2442ffff addiu 02, 02, 65535
800200f4: 00021080 sll 02, 02, 02
800200f8: 8fc30018 lw 03, 24(1e)

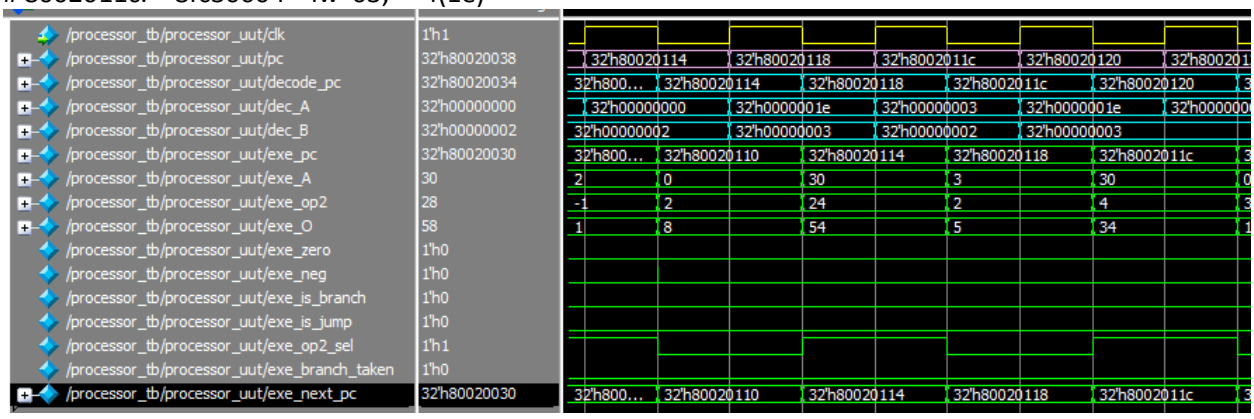
800200fc: 00621021 addu 02, 03, 02



80020100: 8c420000 lw 02, 0(02)
80020104: afc20008 sw 02, 8(1e)
80020108: 8fc20004 lw 02, 4(1e)
8002010c: 2442ffff addiu 02, 02, 65535

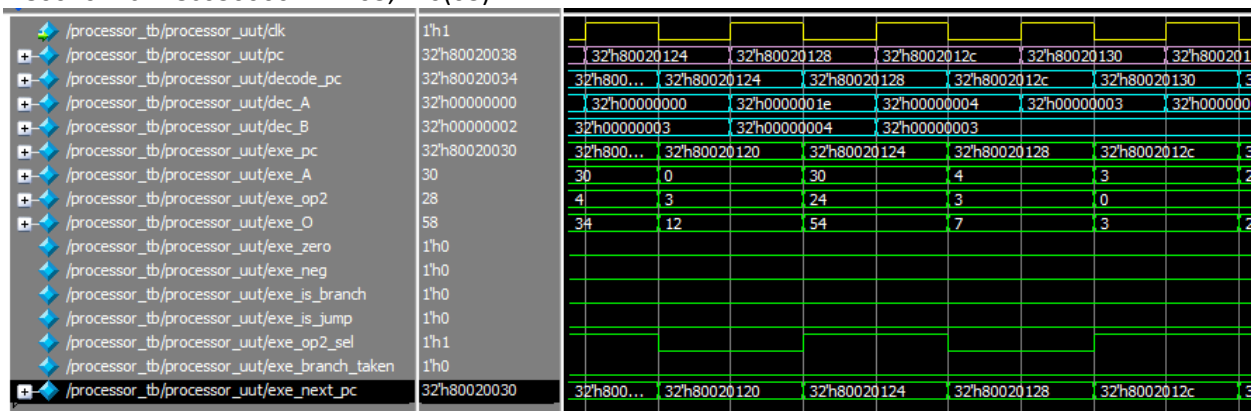


80020110: 00021080 sll 02, 02, 02
80020114: 8fc30018 lw 03, 24(1e)
80020118: 00621021 addu 02, 03, 02
8002011c: 8fc30004 lw 03, 4(1e)

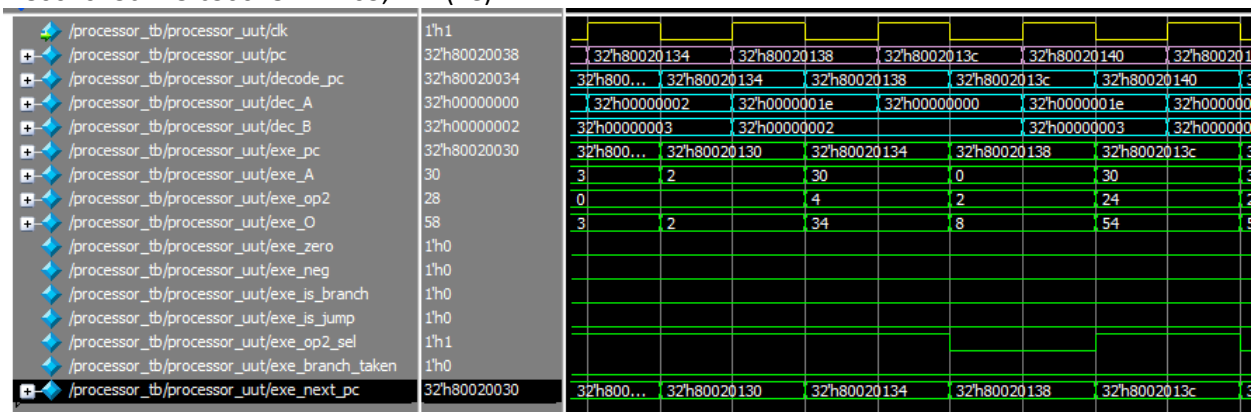


80020120: 00031880 sll 03, 03, 02

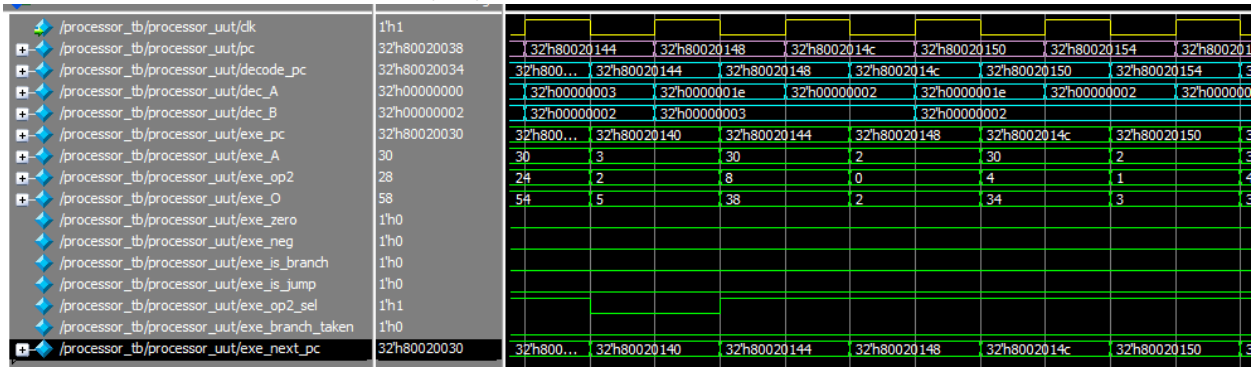
80020124: 8fc40018 lw 04, 24(1e)
80020128: 00831821 addu 03, 04, 03
8002012c: 8c630000 lw 03, 0(03)



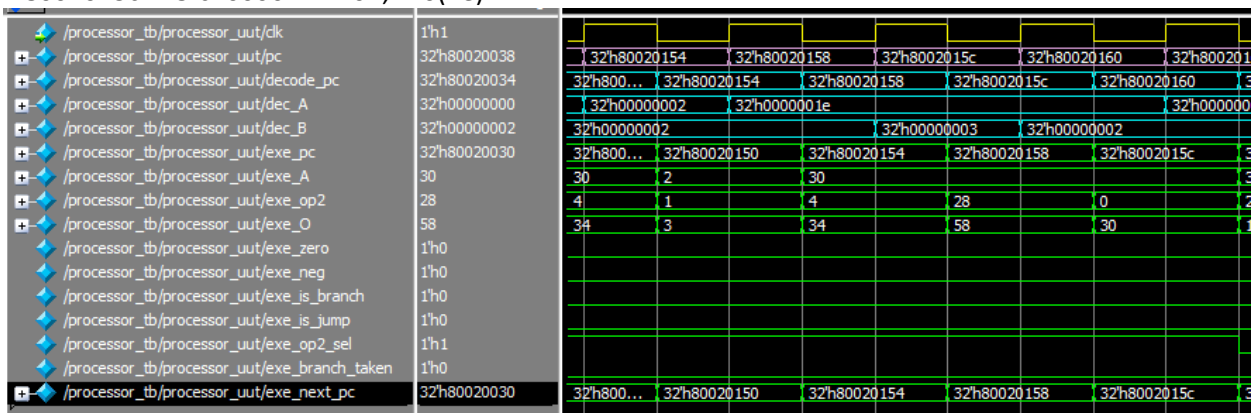
80020130: ac430000 sw 03, 0(02)
80020134: 8fc20004 lw 02, 4(1e)
80020138: 00021080 sll 02, 02, 02
8002013c: 8fc30018 lw 03, 24(1e)



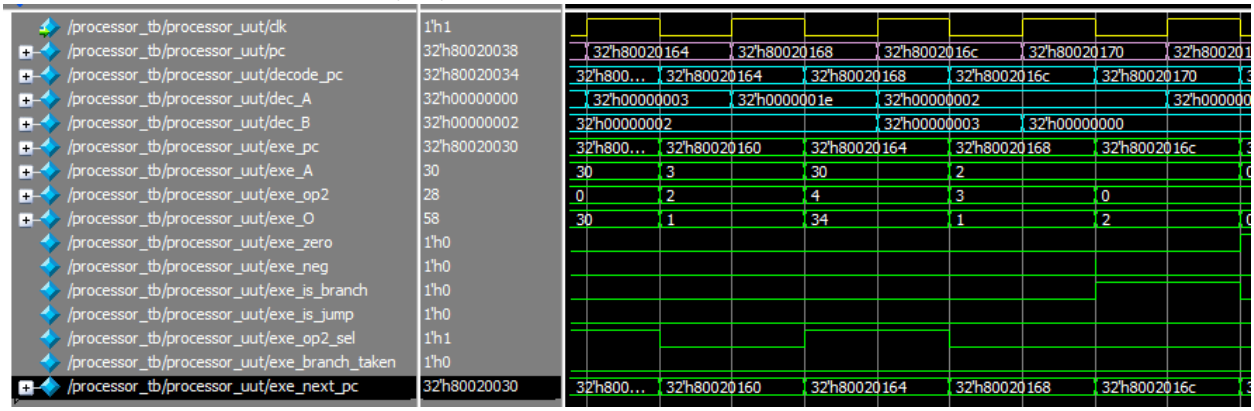
80020140: 00621021 addu 02, 03, 02
80020144: 8fc30008 lw 03, 8(1e)
80020148: ac430000 sw 03, 0(02)
8002014c: 8fc20004 lw 02, 4(1e)
80020150: 24420001 addiu 02, 02, 1



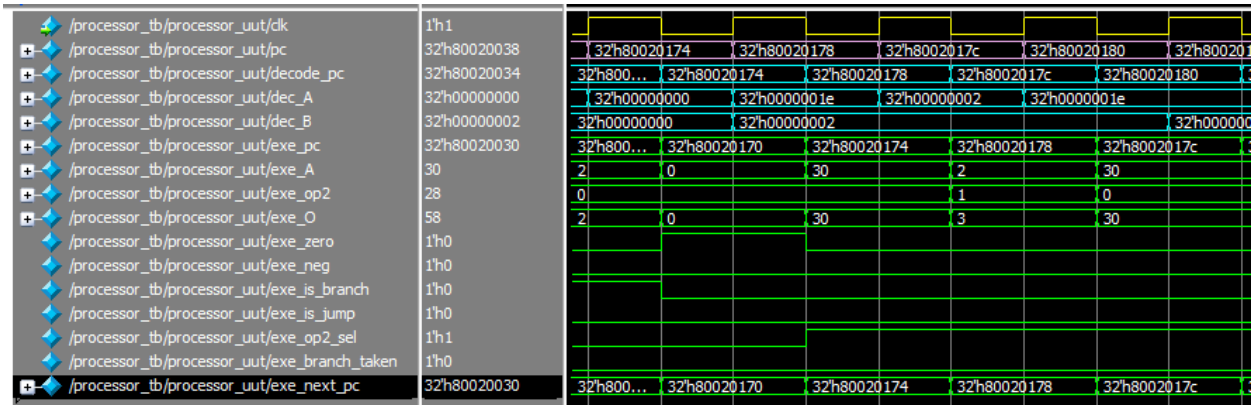
80020150: 24420001 addiu 02, 02, 1
80020154: afc20004 sw 02, 4(1e)
80020158: 8fc3001c lw 03, 28(1e)
8002015c: 8fc20000 lw 02, 0(1e)



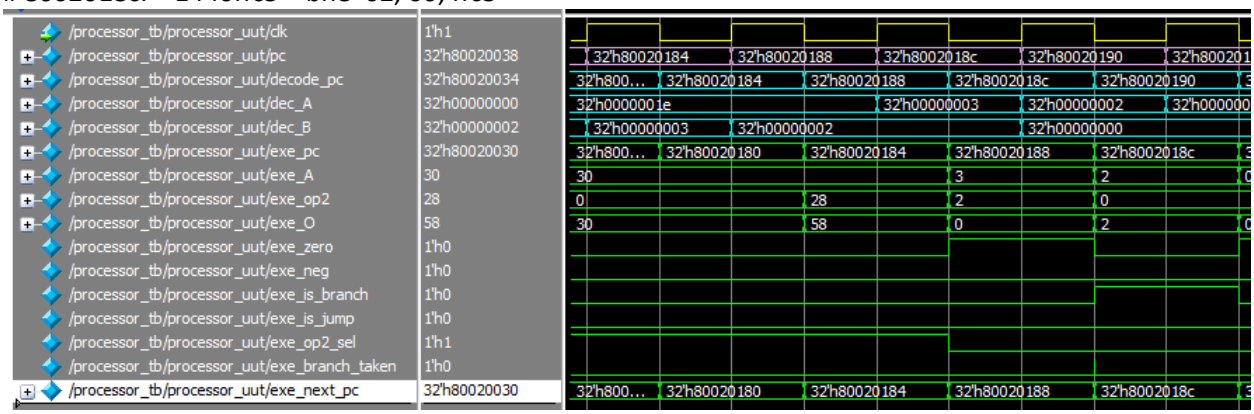
80020164: 8fc20004 lw 02, 4(1e)
80020168: 0043102a slt 02, 02, 03
8002016c: 1440ffd1 bne 02, 00, ffd1



80020170: 00000000 NOP
80020174: 8fc20000 lw 02, 0(1e)
80020178: 24420001 addiu 02, 02, 1
8002017c: afc20000 sw 02, 0(1e)



80020180: 8fc30000 lw 03, 0(1e)
80020184: 8fc2001c lw 02, 28(1e)
80020188: 0062102a slt 02, 03, 02
8002018c: 1440ffc5 bne 02, 00, ffc5



80020190: 00000000 NOP
80020194: 03c0e821 addu 1d, 1e, 00
80020198: 8fbe0014 lw 1e, 20(1d)
8002019c: 27bd0018 addiu 1d, 1d, 24