ITI DIGITAL VERIFICATION ENGINEERING PROGRAM

ASSIGNMENT 3

VERILOG

RISC-V SINGLE CYCLE CPU

SUBMITTED TO

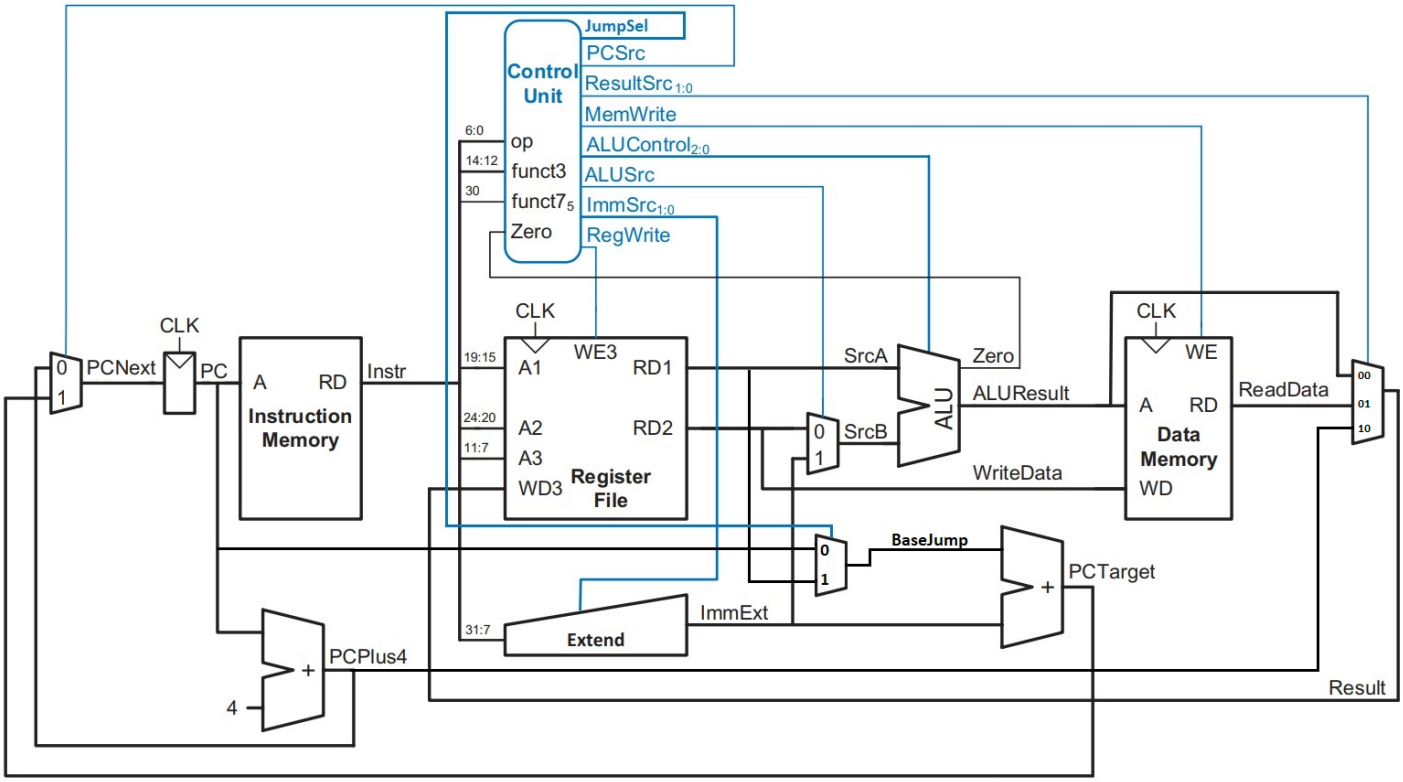
ENG MOHAMED ZAYTOUN

BY

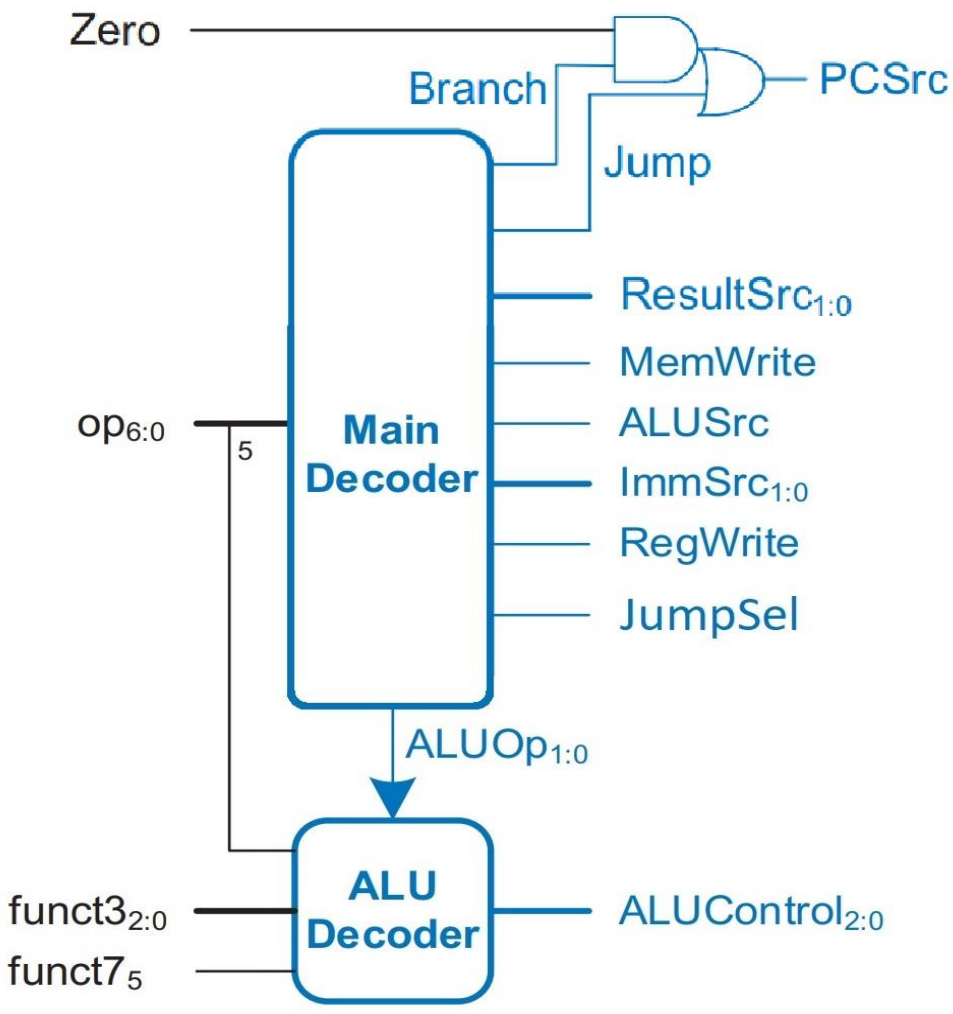
ABDURRAHMAN SHERIF ABDULHAMID

I. Block diagram:

- Design Block Diagram:



- Control Unit Block Diagram



II. Design codes:

- Universal blocks
  - Adder

```
module adder (
    input [31:0] a, b,
    output [31:0] y
);
    assign y = a + b;
endmodule
```

  - 2-1 MUX

```
module mux2 #(parameter WIDTH = 8) (
    input  [WIDTH-1:0] d0, d1,
    input  s,
    output  [WIDTH-1:0] y
);
    assign y = s ? d1 : d0;
endmodule
```

  - 3-1 MUX

```
module mux3 #(parameter WIDTH = 8) (
    input  [WIDTH-1:0] d0, d1, d2,
    input  [1:0] s,
    output  [WIDTH-1:0] y
);
    assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

  - FLOP REGISTER

```
module flopr #(parameter WIDTH = 8) (
    input clk, reset,
    input [WIDTH-1:0] d,
    output reg [WIDTH-1:0] q
);
    always @(posedge clk or posedge reset) begin
        if(reset) begin
            q <= 0;
        end else begin
            q <= d;
        end
    end
endmodule
```

- Control Unit:
  - Main Decoder

```
1
2   module maindec (
3       input  [6:0] op,
4       output [1:0] ResultSrc,
5       output MemWrite,
6       output Branch, ALUSrc,
7       output RegWrite, Jump,
8       output [1:0] ImmSrc,
9       output [1:0] ALUOp,
10      output JumpSel
11  );
12      reg [11:0] controls;
13      assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
14          ResultSrc, Branch, ALUOp, Jump, JumpSel} = controls;
15      always@(*)begin
16          case(op)
17  // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump_JumpSel
18              7'b0000011: controls = 12'b1_00_1_0_01_0_00_0_0; // lw
19              7'b0100011: controls = 12'b0_01_1_1_00_0_00_0_0; // sw
20              7'b0110011: controls = 12'b1_xx_0_0_00_0_10_0_0; // R-type
21              7'b1100011: controls = 12'b0_10_0_0_00_1_01_0_0; // beq/bne*
22              7'b0010011: controls = 12'b1_00_1_0_00_0_10_0_0; // I-type ALU
23              7'b1101111: controls = 12'b1_11_0_0_10_0_00_1_0; // jal
24              7'b1100111: controls = 12'b1_00_0_0_10_0_00_0_1; // jalr *
25
26              default: controls = 11'bx_xx_x_x_xx_x_xx_x; // ???
27          endcase
28      end
29  endmodule
```

  - ALU Decoder

```
1   module aludec (
2       input  opb5,
3       input  [2:0] funct3,
4       input  funct7b5,
5       input  [1:0] ALUOp,
6       output reg [2:0] ALUControl
7   );
8       wire RtypeSub;
9       assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract
10      always@(*)
11          case(ALUOp)
12              2'b00: ALUControl = 3'b000; // addition
13              2'b01: ALUControl = 3'b001; // subtraction
14              default: case(funct3) // R-type or I-type ALU
15                  3'b000: if (RtypeSub)
16                      ALUControl = 3'b001; // sub
17                  else
18                      ALUControl = 3'b000; // add, addi
19                  3'b010: ALUControl = 3'b101; // slt, slti
20                  3'b110: ALUControl = 3'b011; // or, ori
21                  3'b111: ALUControl = 3'b010; // and, andi
22                  default: ALUControl = 3'bxxx; // ???
23              endcase
24          endcase
25      endmodule
```

- Control Unit Wrapper

```
module controller (
    input  [6:0] op,
    input  [2:0] funct3,
    input  funct7b5,
    input  Zero,
    output  [1:0] ResultSrc,
    output  MemWrite,
    output  PCSrc, ALUSrc,
    output  RegWrite, Jump,
    output  [1:0] ImmSrc,
    output  [2:0] ALUControl,
    output  JumpSel
);
    wire [1:0] ALUOp;
    wire Branch;
    maindec md(op, ResultSrc, MemWrite, Branch,
        ALUSrc, RegWrite, Jump, ImmSrc, ALUOp, JumpSel);
    aludec ad(op[5], funct3, funct7b5, ALUOp, ALUControl);
    assign PCSrc = (Branch & Zero & funct3[0]) | Jump | (Branch & !Zero & funct3[0]) | JumpSel;
endmodule
```

- Data Path:

```
module datapath (
    input  clk, reset,
    input  [1:0] ResultSrc,
    input  PCSrc, ALUSrc,
    input  RegWrite,
    input  [1:0] ImmSrc,
    input  [2:0] ALUControl,
    output  Zero,
    output  [31:0] PC,
    input  [31:0] Instr,
    output  [31:0] ALUResult, WriteData,
    input  [31:0] ReadData,
    input JumpSel
);
    wire [31:0] PCNext, PCPlus4, PCTarget;
    wire [31:0] ImmExt;
    wire [31:0] SrcA, SrcB;
    wire [31:0] Result;
    wire [31:0] BaseJump;
// next PC logic
    flopr #(32) pcreg(clk, reset, PCNext, PC);
    adder pcadd4(PC, 32'd4, PCPlus4);
    mux2 #(32) jumpbranch (PC,SrcA , JumpSel, BaseJump);
    adder pcaddbranch(BaseJump, ImmExt, PCTarget);
    mux2 #(32) pcmux(PCPlus4, PCTarget, PCSrc, PCNext);
// register file logic
    regfile rf(clk, RegWrite, Instr[19:15], Instr[24:20],
        Instr[11:7], Result, SrcA, WriteData);
    extend ext(Instr[31:7], ImmSrc, ImmExt);
// ALU logic
    mux2 #(32) srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
    alu alu(SrcA, SrcB, ALUControl, Zero, ALUResult);
    mux3 #(32) resultmux(ALUResult, ReadData, PCPlus4,
        ResultSrc, Result);
endmodule
```

- ALU:

```
module alu (
    input [31:0] SrcA,
    input [31:0] SrcB,
    input [2:0] ALUcontrol,
    output zero,
    output reg [31:0] ALUResult
);

    always @(*) begin
        case (ALUcontrol)
            3'b000: ALUResult = SrcA + SrcB ;        // Addition
            3'b001: ALUResult = SrcA - SrcB ;       // Subtraction
            3'b010: ALUResult = SrcA & SrcB ;      // Bitwise AND
            3'b011: ALUResult = SrcA | SrcB ;      // Bitwise OR
            3'b101: ALUResult = (SrcA < SrcB) ? 1 : 0; // Set if less than (SLT)
            default: ALUResult = 32'bx;         // Default: Zero output
        endcase
    end

    assign zero = (ALUResult == 32'b0) ? 1'b1 : 1'b0;

endmodule

```

- Extend block:

```
module extend (
    input [31:7] instr,
    input [1:0] immsrc,
    output reg [31:0] immext
);

always @(*) begin
    case(immsrc)
        // I-type
        2'b00: immext = {{20{instr[31]}}, instr[31:20]};
        // S-type (stores)
        2'b01: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
        // B-type (branches)
        2'b10: immext = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
        // J-type (jal)
        2'b11: immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
        default: immext = 32'bx; // undefined
    endcase
end

endmodule

```

- Data Memory (RAM):

```
module dmem (
    input clk, we,
    input [31:0] a, wd,
    output [31:0] rd
);
    reg [31:0] RAM [255:0];

    assign rd = RAM[a[7:0]]; // word aligned

    always @(posedge clk) begin
        if (we) RAM[a[7:0]] <= wd;
    end
endmodule
```

- Instruction Memory (ROM):

```
module imem (
    input [31:0] a,
    output [31:0] rd
);
    reg [31:0] ROM[255:0]; // 1KB memory (256 words)

    initial
        $readmemh("riscvtest.txt", ROM);

    assign rd = ROM[a[9:2]]; // word accessible
endmodule

```

- Register File:

```
module regfile (
    input  clk,
    input  we3,
    input  [4:0] a1, a2, a3,
    input  [31:0] wd3,
    output  [31:0] rd1, rd2
);

    reg [31:0] rf [0:31]; // 32 registers

    // Three-ported register file
    // Read two ports combinationaly (A1/RD1, A2/RD2)
    // Write third port on rising edge of clock (A3/WD3/WE3)
    // Register 0 hardwired to 0
    always @(posedge clk) begin
        if (we3)
            rf[a3] <= wd3;
    end

    assign rd1 = (a1 != 0) ? rf[a1] : 32'h0;
    assign rd2 = (a2 != 0) ? rf[a2] : 32'h0;

endmodule
```

- Single cycle RISC-V Processor:

```
module riscvsingle (
    input  clk, reset,
    output  [31:0] PC,
    input  [31:0] Instr,
    output  MemWrite,
    output  [31:0] ALUResult, WriteData,
    input  [31:0] ReadData
);
    wire ALUSrc, RegWrite, Jump, Zero;
    wire [1:0] ResultSrc, ImmSrc;
    wire [2:0] ALUControl;
    wire JumpSel;
    controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
        ResultSrc, MemWrite, PCSrc,
        ALUSrc, RegWrite, Jump,
        ImmSrc, ALUControl, JumpSel);

    datapath dp(clk, reset, ResultSrc, PCSrc,
        ALUSrc, RegWrite,
        ImmSrc, ALUControl,
        Zero, PC, Instr,
        ALUResult, WriteData, ReadData, JumpSel);
endmodule
```

- Top Module Including Memories (Data, Instruction):

```
module top (
    input  clk, reset,
    output  [31:0] WriteData, DataAdr,
    output  MemWrite,
    output [31:0] PC
);
    wire [31:0] Instr, ReadData;
// instantiate processor and memories
    riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite,
        DataAdr, WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

```

III. Test Bench:

- ## Code:

```
module testbench();
    reg clk;
    reg reset;
    wire [31:0] WriteData, DataAdr, PC;
    wire MemWrite;
// instantiate device to be tested
    top dut(clk, reset, WriteData, DataAdr, MemWrite, PC);
// initialize test
    initial
        begin
            reset <= 1; # 22; reset <= 0;
        end
// generate clock to sequence tests
    always
        begin
            clk <= 1; # 5; clk <= 0; # 5;
        end
// check results
    always @(negedge clk)
        begin
            if(MemWrite) begin
                if(DataAdr === 32'h61 & WriteData === 32'h25) begin
                    $display("Simulation succeeded");
                end
                else begin
                    $display("Simulation failed");
                end
            end
            if(PC === 32'h60) $display("PROGRAM COMPLETE");
        end
endmodule
```

- ## Assembly Program:
  - ### Program:

```
main:   addi x2, x0, 5          # x2 = 5                    0       00500113
        addi x3, x0, 12         # x3 = 12                   4       00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8       FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C       0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10      0041F2B3
        add  x5, x5, x4         # x5 = (4 + 7) = 11         14      004282B3
        beq  x5, x7, end        # shouldn't be taken        18      02728863
        bne  x4, x0, around     # should be taken           1C      00021463
        addi x5, x0, 0          # shouldn't happen          20      00000293
around: add  x7, x4, x5         # x7 = (7 + 11) = 18        24      005203B3
        sub  x7, x7, x2         # x7 = (18 - 5) = 13        28      402383B3
        sw   x7, 84(x3)         # [96] = 13                 2C      0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 13            30      06002103
        add  x9, x2, x5         # x9 = (13 + 11) = 24       34      005104B3
        addi x4, x4, 0x7CC      # x4 = 7D3                  38      7CC20213
        jal  x3, end            # jump to end, x3 = 0x40    3C      008001EF
        addi x2, x0, 1          # shouldn't happen          40      00100113
end:    add  x2, x2, x9         # x2 = (13 + 24) = 37       44      00910133
        sw   x2, 0x21(x3)       # mem[61] = 0x25 = 37       48      0221A0A3
done:   beq  x2, x2, done       # infinite loop             4C      00210063
        addi x3, x0, 12         # x3 = 0                    50      00000193
        jalr x2, x3, 60         # jump to 0+8               54      06018167
        add  x9, x2, x5         # x9 = (13 + 11) = 24       58      005104B3
        addi x7, x3, -9         # x7 = (12 - 9) = 3         5C      FF718393
                                                            60
```

- **Code in HEX:**

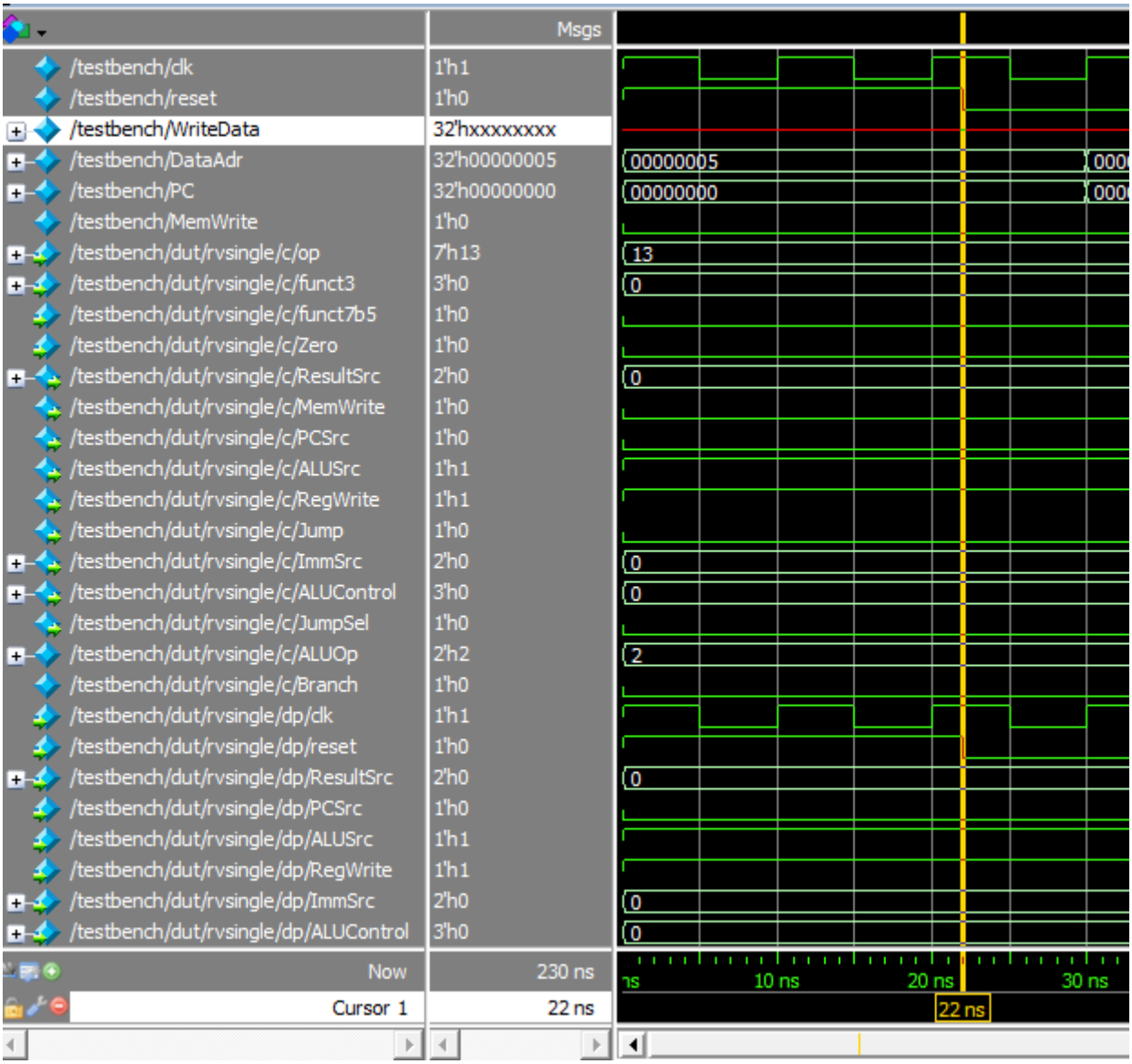
```
00500113
00C00193
FF718393
0023E233
0041F2B3
004282B3
02728863
00021463
00000293
005203B3
402383B3
0471AA23
06002103
005104B3
7CC20213
008001EF
00100113
00910133
0221A0A3
00210063
00000193
06018167
005104B3
FF718393
```

- **Description :**
  - Program manipulates some registers performing different types of instructions.
  - Next, on the last “store instruction” program checks on particular registers after manipulating their values making values manipulation among register serially, which means that the register will never reach this value unless instructions were correctly processed.
  - Finally, test bench checks on the program counter to make sure that we’ve reached the ending point, meaning that all instructions, were correctly performed.

  NOTE: Program will never reach the last program count unless all instructions we’re well handled with no errors.

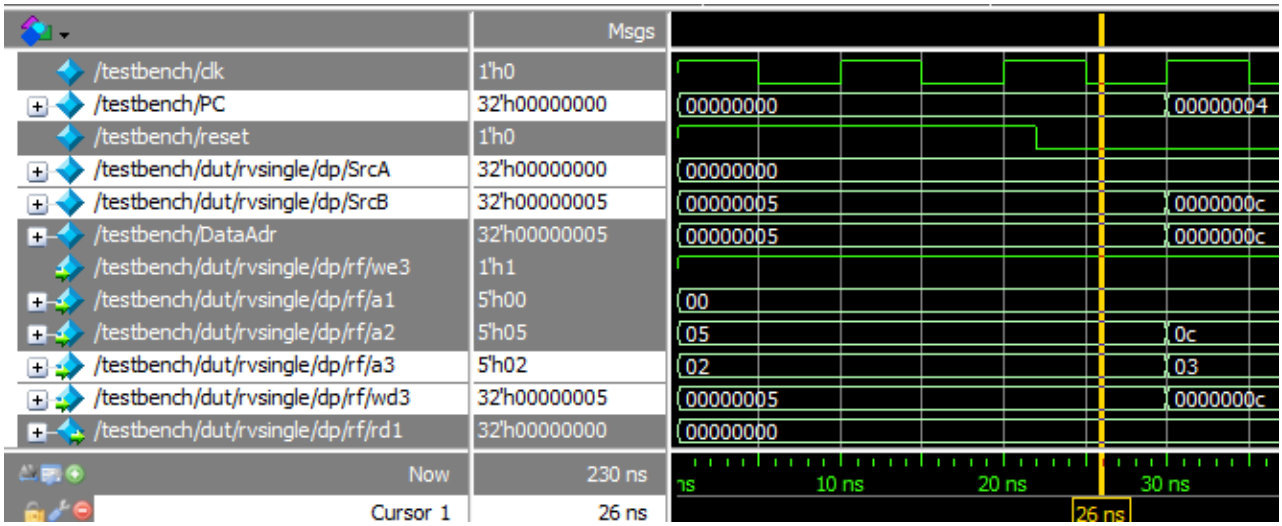
- Snapshots:

1- Reset was set high for 22ns and then toggled.



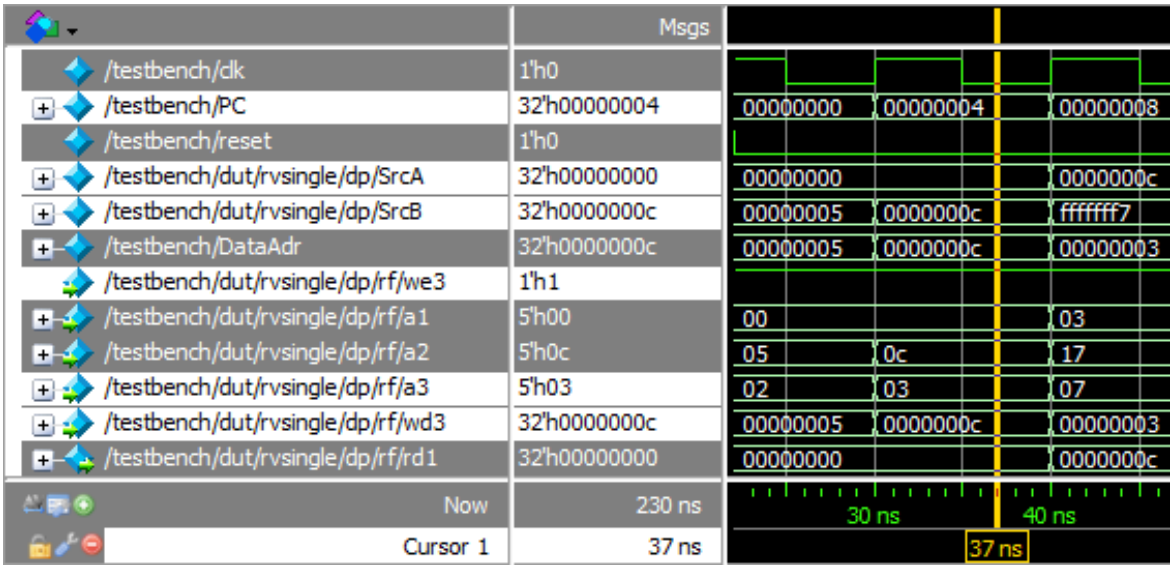
2- addi x2, x0, 5

At PC= 0, write 5 to address 2 in the reg file (x2).



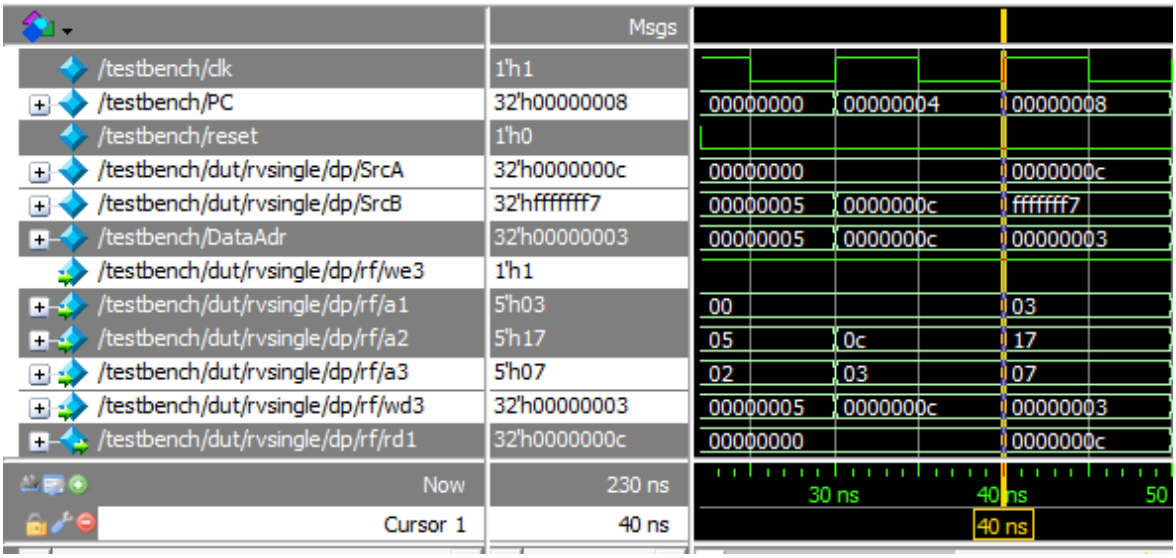
3- addi x3, x0, 12 # x3 = 12

At PC= 4, write 12 (0xC) to address 3 in the reg file (x3).



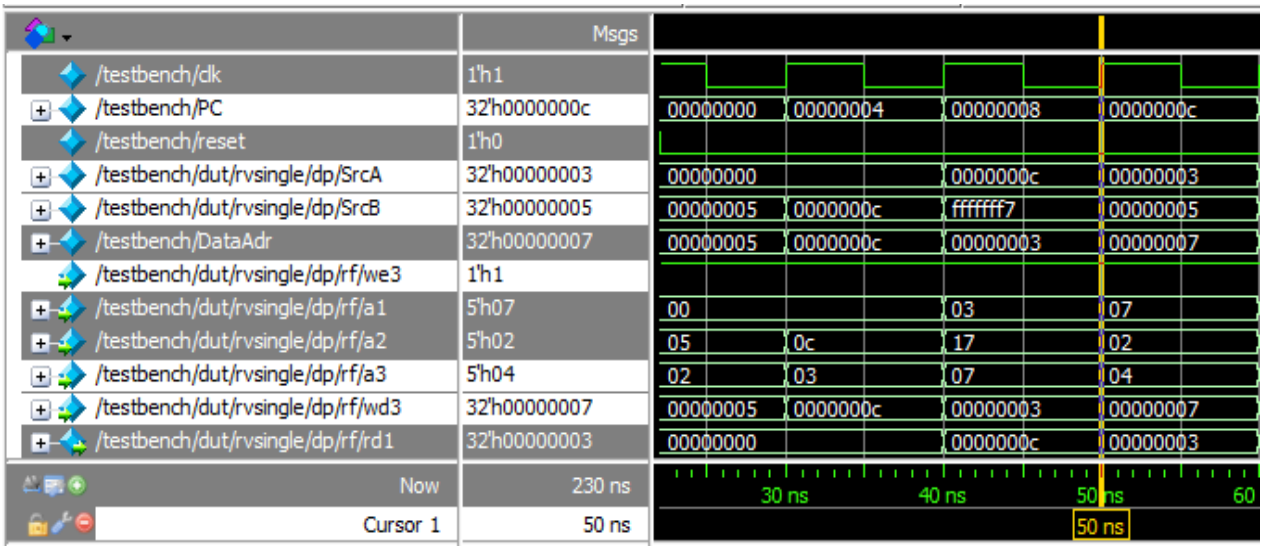
4- addi x7, x3, -9 # x7 = (12 - 9) = 3

At PC= 8, subtract and write 3 to address 7 in the reg file (x3).



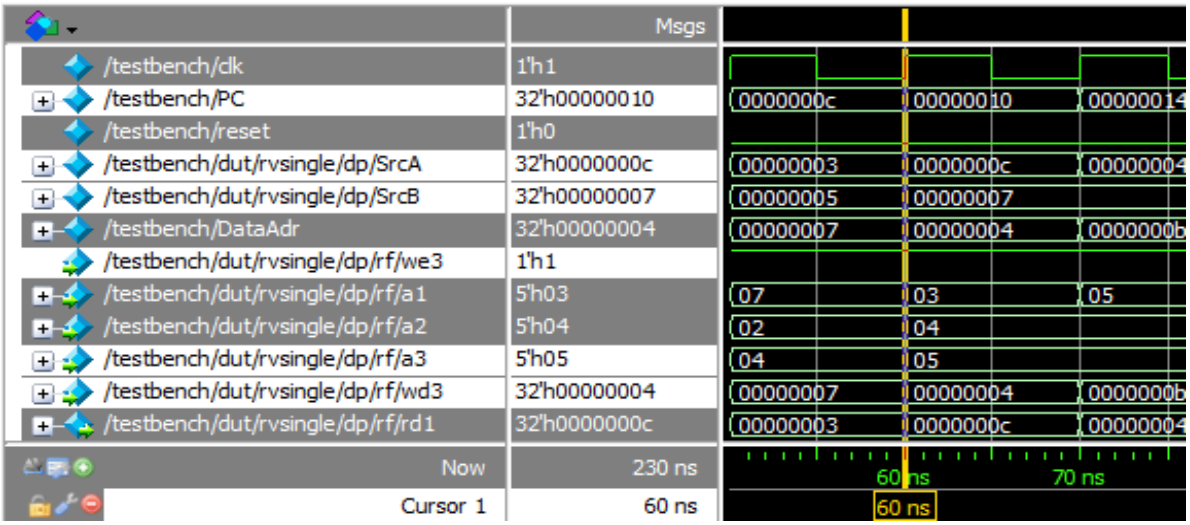
5- or x4, x7, x2 # x4 = (3 OR 5) = 7

At PC= C, perform bitwise 3 OR 5, and write 7 to address 4 in the reg file (x4).



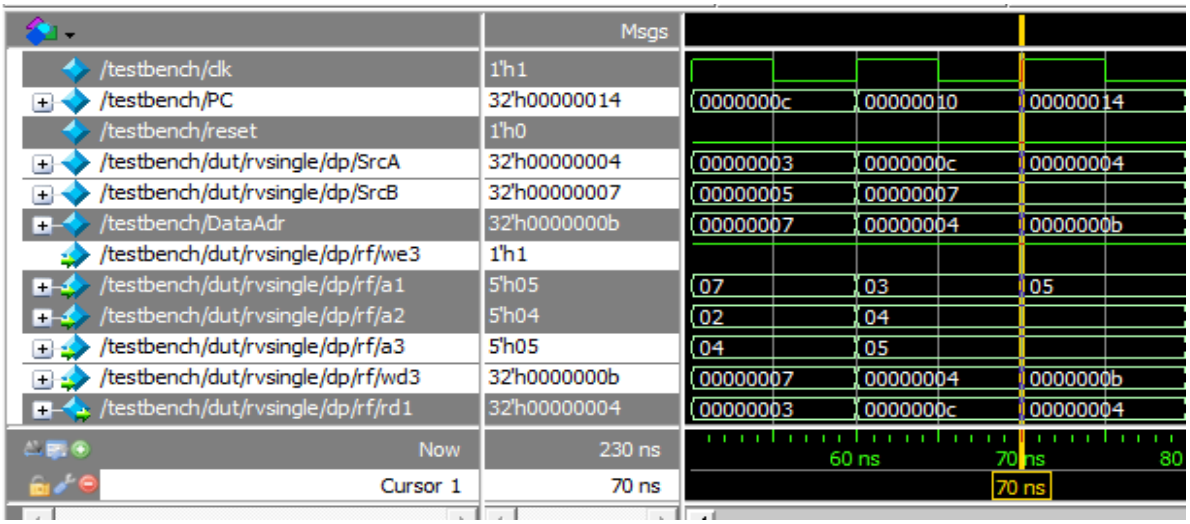
6- and x5, x3, x4 # x5 = (12 AND 7) = 4

At PC= 10, perform bitwise 12 AND 7 and write 4 to address 5 in the reg file (x5).



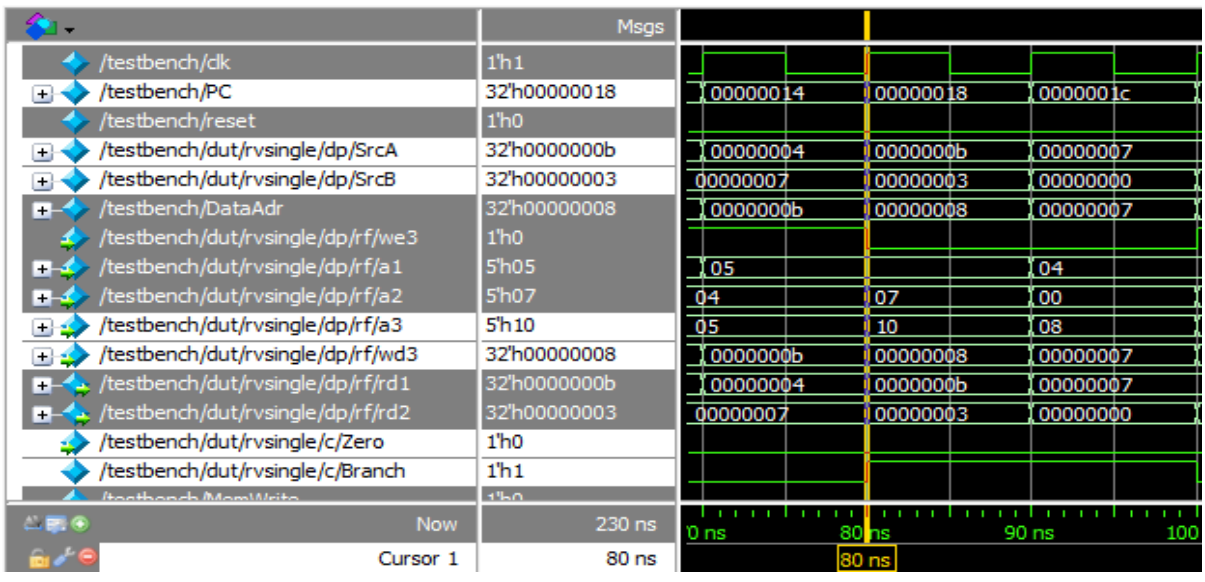
7- add x5, x5, x4 # x5 = (4 + 7) = 11

At PC= 14, perform addition and write 11 (0xB) to address 5 in the reg file (x5).



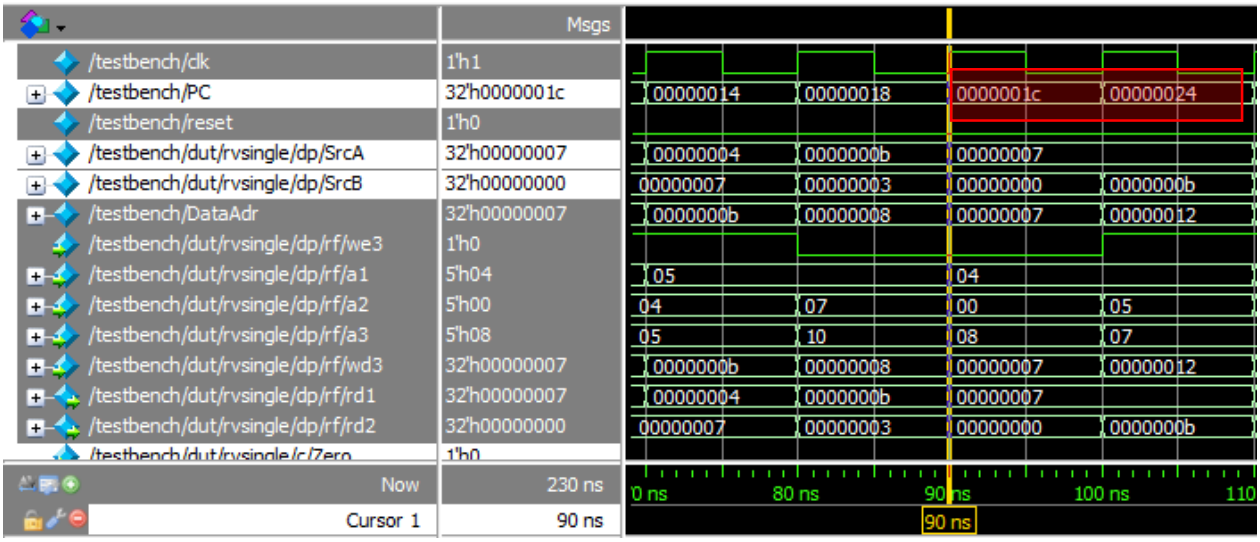
8- beq x5, x7, end # shouldn't be taken

At PC= 18, check if x7(3) == x5(11) then the next PC = 44



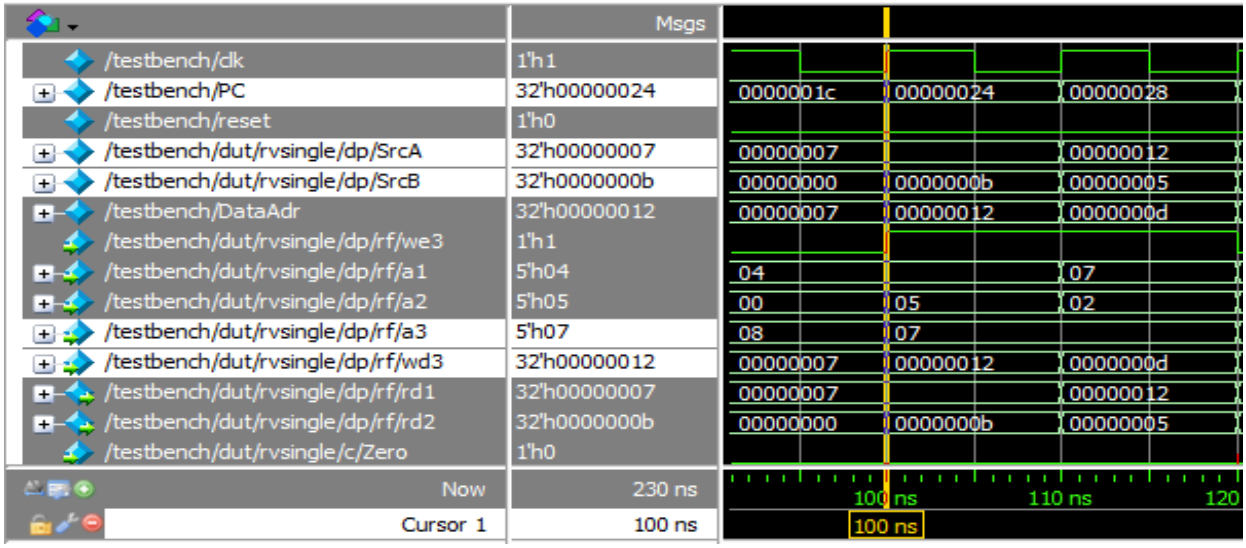
9- bne x4, x0, around # should be taken

At PC= 1C, if x4 ≠ 0 then next PC= 24



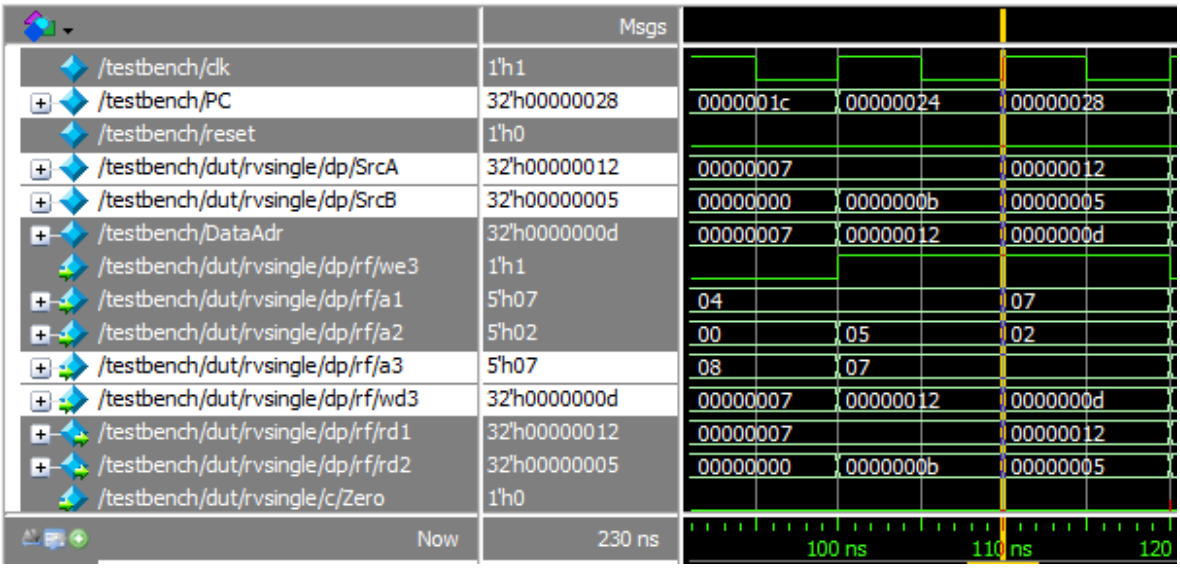
10- around: add x7, x4, x5 # x7 = (7 + 11) = 18

At PC= 24 perform addition and write 18 to address 7 in the reg file (x7).



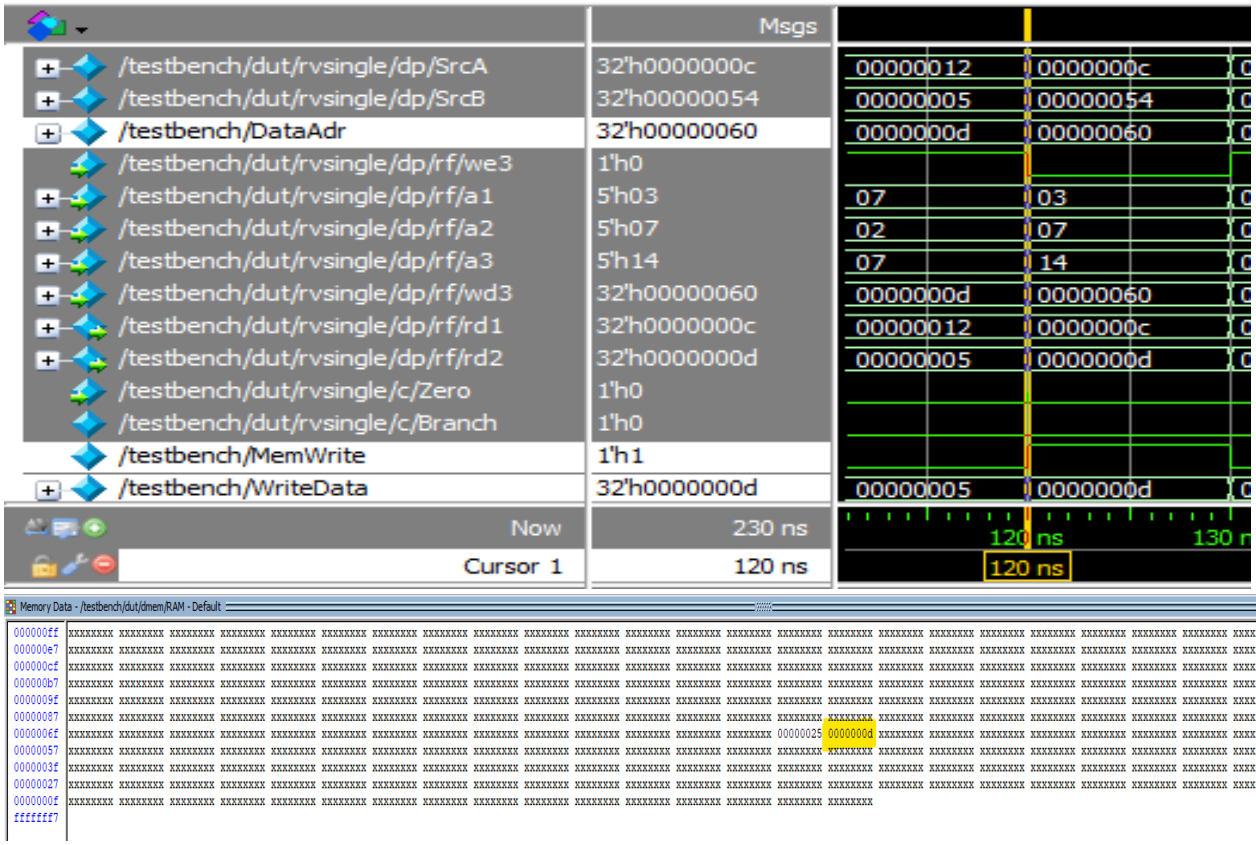
11- sub x7, x7, x2 # x7 = (18 - 5) = 13

At PC= 28 perform subtraction and write 13 (0xD) to address 7 in the reg file (x7).



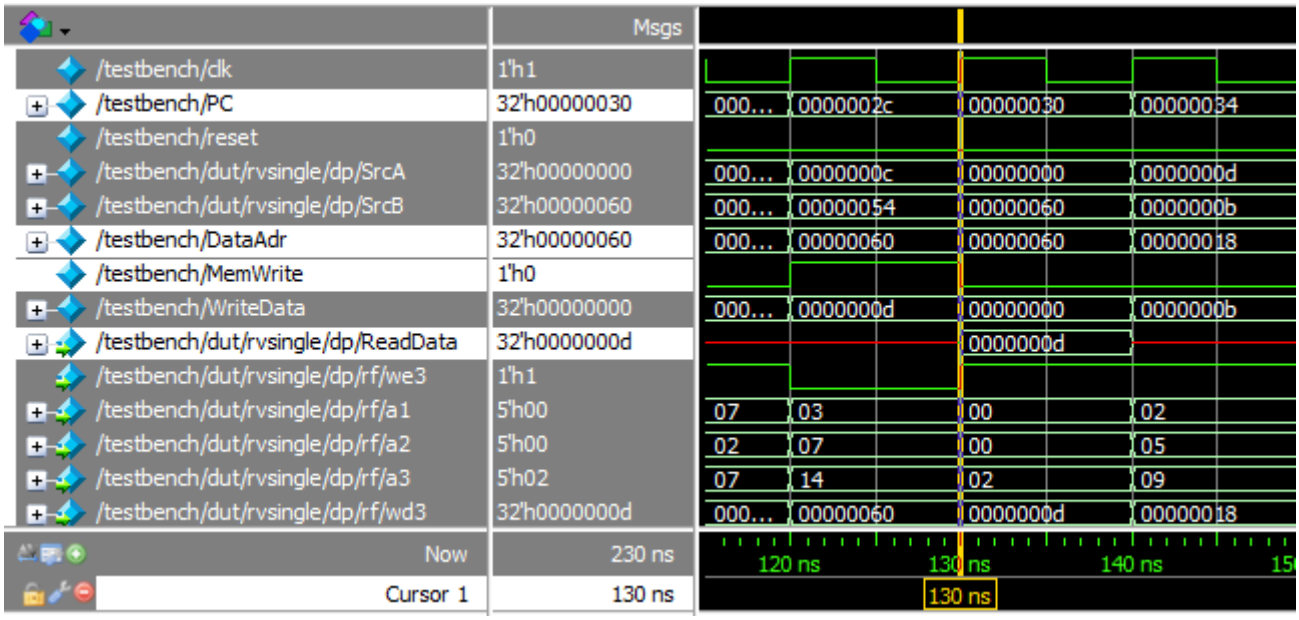
12- sw x7, 84(x3) # [96] = 13

At PC= 2C, store value 0xD in address 96 (0x60) in the data memory



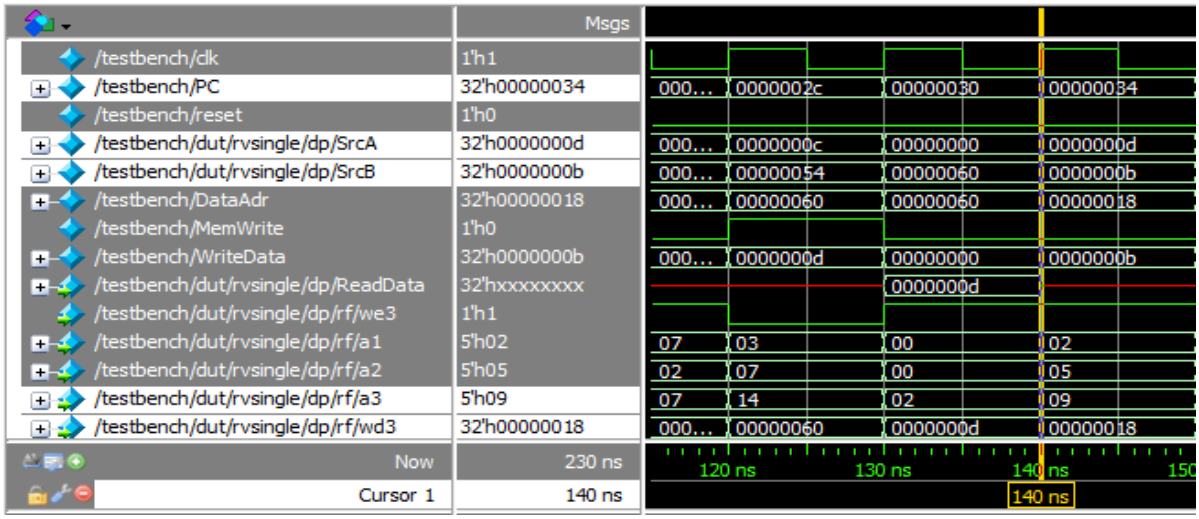
13- lw x2, 96(x0) # x2 = [96] = 13

At PC= 30, load value 0xD from address 96 (0x60) in the data memory to the signal ReadData.



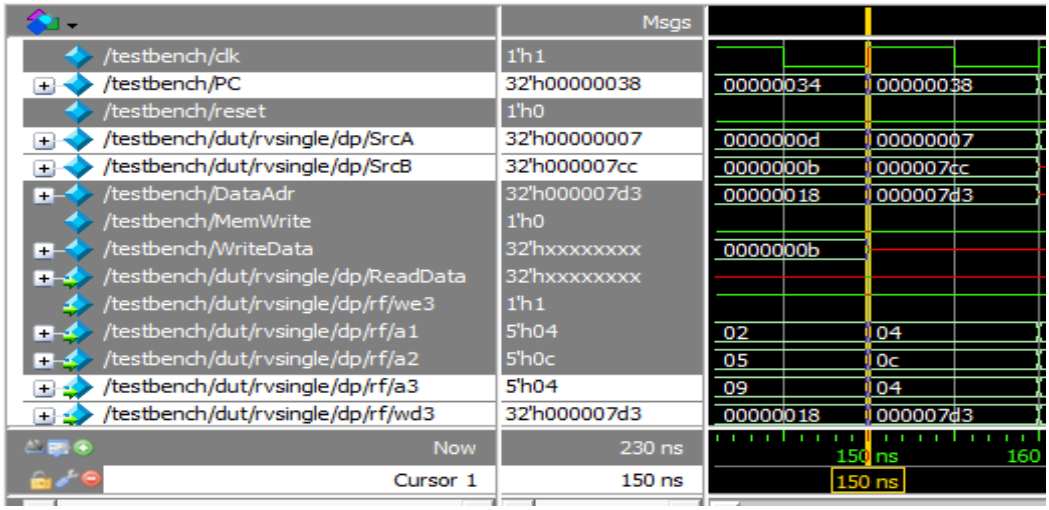
14- add x9, x2, x5 # x9 = (13 + 11) = 24

At PC= 34, perform addition and write 24 to address 9 in the reg file (x9).



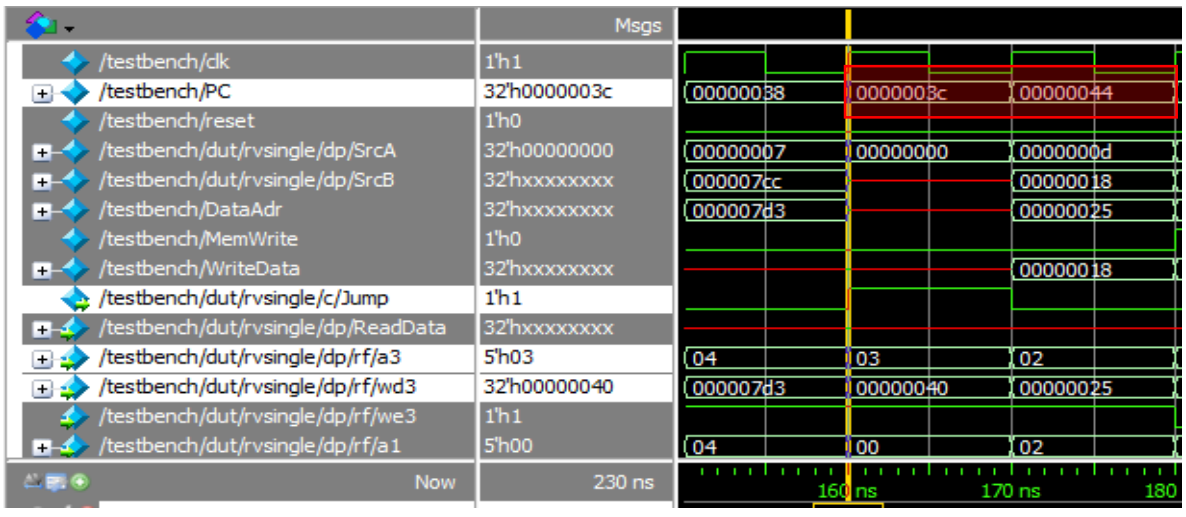
15- addi x4, x4, 0x7CC # x4 = 7D3

At PC= 38, add (0x7CC+ 7) and write 0x7D3 to address 4 in the reg file (x4).



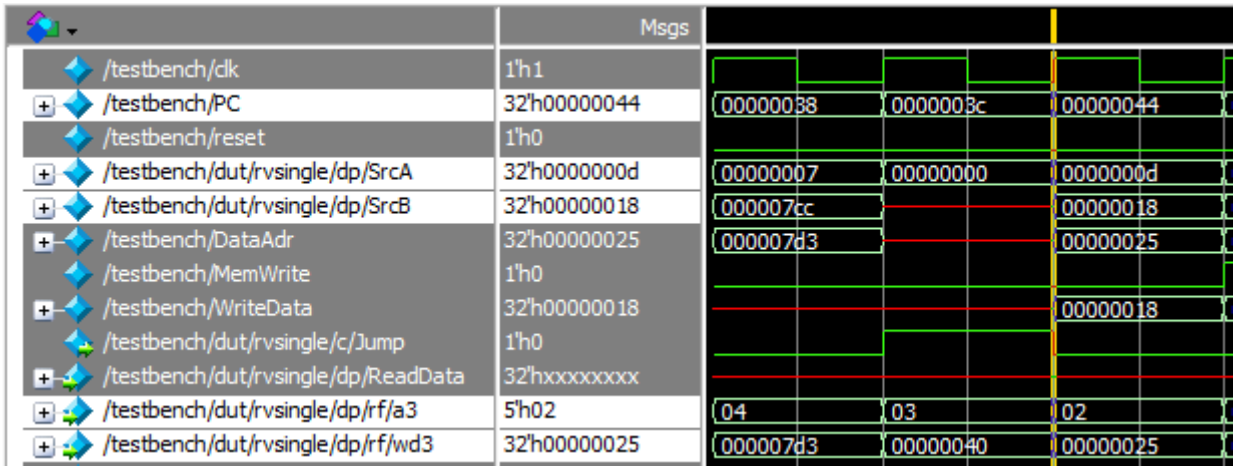
16- jal x3, end # jump to end, x3 = 0x40

At PC= 3C, make next PC= 44 and write the original next PC to address 3 in the reg file (x3).



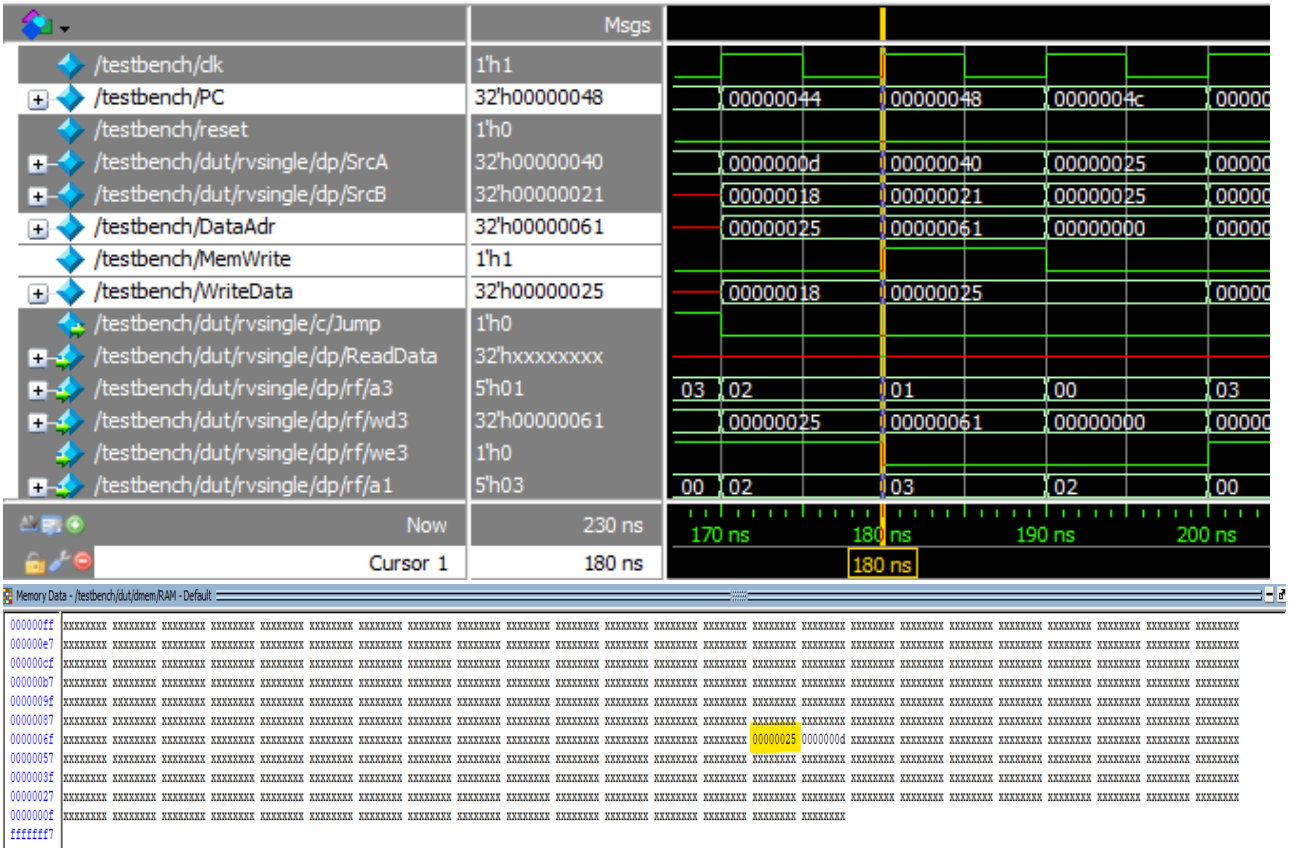
17- end: add x2, x2, x9 # x2 = (13 + 24) = 37

At PC= 44, add 13+24 and write 37 (0x25) to address 2 in the reg rile (x2).



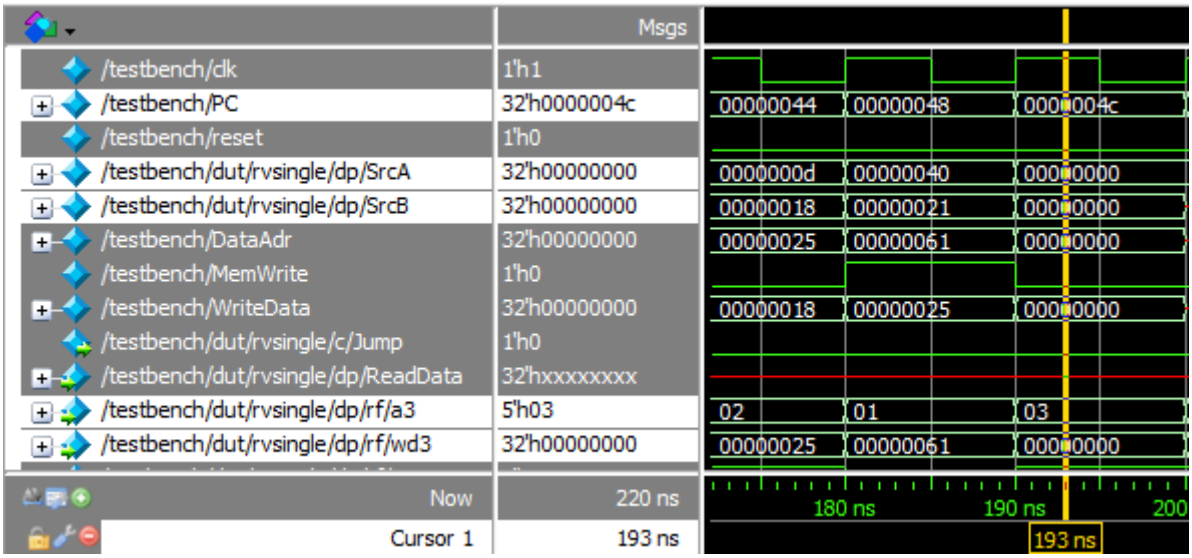
18- sw x2, 0x21(x3) # mem[97] = 0x25 = 37

At PC= 48, store the value 37 (0x25) from (x2) to address 0x61 in data memory.



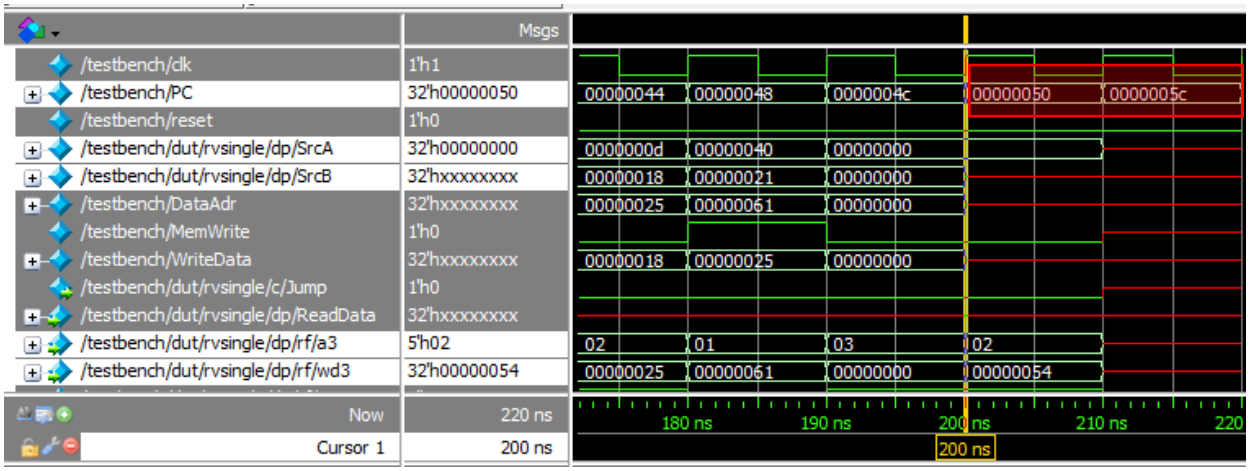
19- addi x3, x0, 0 # x3 = 0

At PC= 4C, write 0 to address 3 in the reg file (x3).



20- jalr x2, x3, 5C # jump to 0+8

At PC=50, set next PC= x3 + 5C = 5C, and write the original next PC= 54 to address 2 in the reg file (x2).



SUCCESSFUL Jump, and store…

21- END OF SIMULATION!

```
VSIM 19> restart
# ** Note: (vsim-8009) Loading existing optimized design _opt2
# Loading work.testbench(fast)
# Loading work.top(fast)
# Loading work.riscvsingle(fast)
# Loading work.controller(fast)
# Loading work.maindec(fast)
# Loading work.aludec(fast)
# Loading work.datapath(fast)
# Loading work.flopr(fast)
# Loading work.adder(fast)
# Loading work.mux2(fast)
# Loading work.regfile(fast)
# Loading work.extend(fast)
# Loading work.alu(fast)
# Loading work.mux3(fast)
# Loading work.imem(fast)
# Loading work.dmem(fast)
VSIM 20> run
# Simulation succeeded
# PROGRAM COMPLETE

VSIM 21>
```