Computer Architecture ECE 4680

Project 3

WAYNE STATE UNIVERSITY

Group No: 8

Prakash Sharma

Anika Tasnim

Shubh Patel

Instructor: Nabil Sarhan

Date of Submission: 11/14/20

a. Names and student numbers of the team members.

Prakash sharma, Anika Tasnim, Shubh Patel

b. Compilation Status: “Successful” or “Unsuccessful”

Successful

c. Elaboration Status: “Successful” or “Unsuccessful”

Successful

d. Simulation Status: “Simulates correctly” or “Does Not Simulate Correctly”

Simulates correctly

e. A copy of the Verilog design file (yes another copy within the pdf file in addition to the separate .v file in 1)

```
// Author Names: Anika Tasnim, Shubh Patel, Prakash Sharma
// Last Modified Date: 11/13/2020
// Compilation Status: “Successful”
// Elaboration Status: “Successful”
// Simulation Status: “Simulates correctly”

module CPU (clock,PC, IR, ALUOut, MDR, A, B, reg8);
   //opcodes
   parameter R_FORMAT = 6'b000000;
   parameter LW       = 6'b100011;
   parameter LB       = 6'b100000;
   parameter SW       = 6'b101011;
   parameter BEQ      = 6'b000100;
   parameter BNE      = 6'b000101;
   parameter I_FORMAT = 6'b001000;
   parameter J_FORMAT = 6'b000010;
   parameter Jal_FORMAT = 6'b000011;
   parameter Jr_FORMAT = 6'b001000;
   parameter SLT_FORMAT = 6'b101010;
   parameter SLL_FORMAT = 6'b000000;
   parameter SRL_FORMAT = 6'b000010;
   parameter s_inc    = 6'b111111;
   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 [3:0] state; // processor state
   wire [5:0] opcode, shift; //use to get opcode easily
   wire [31:0] SignExtend, PCOffset; //used to get sign extended offset field

   wire [31:0] reg8;
   output [31:0] reg8; //output reg 7 for testing
   assign reg8 = Regs[8]; //output reg 8 (i.e. $t0)

   assign opcode = IR[31:26]; //opcode is upper 6 bits
   //sign extension of lower 16-bits of instruction
   assign SignExtend = {{16{IR[15]}},IR[15:0]};
   assign PCOffset = SignExtend << 2; //PC offset is shifted, displacement X 4
   assign shift = IR[10:6]; //sa

   initial begin   //Load a MIPS test program and data into Memory
   // 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;
      Memory[2] = 32'H20080005;  //  addi $t0,$zero,5// $t0=5
      Memory[3] = 32'Hac08007C;  //  sw $t0, 124($zero) // from add = 7C
      Memory[4] = 32'H8c09007C;  //  lw $t1, 124($zero)// &t1=5 //add = 7C
      Memory[5] = 32'H11280003;  //  beq $t1,$t0,0x000003 (PC=mem[9]X4=36=24(hex))
      Memory[6] = 32'H01284022;  //  sub $t0,$t1,$t0 //$t0=0
      Memory[7] = 32'H01285020;  //  add $t2 $t1 $t0 //$t2=5
      Memory[8] = 32'H0009582A;  //  slt $t3,$zero,$t1 // $t3=1
      Memory[9] = 32'H01294820;  //  add $t1,$t1,$t1 // $t1=A //after beq instruction
      Memory[10] = 32'H01284022;  //  sub $t0,$t1,$t0 //$t0=5
      Memory[11] = 32'H01285020;  //  add $t2 $t1 $t0 //$t2=F
      Memory[12] = 32'H0009582A;  //  slt $t3,$zero,$t1 // $t3=1
      Memory[13] = 32'H000A6080;  //  sll $t4,$t2,2 //$t4= 3C
      Memory[14] = 32'H000C6902;  //  srl $t5,$t4,4 //$t5= 3
      Memory[15] = 32'H08020002;  //  j 0x020002 //PC=18X4=72 = 48 (hex)
      Memory[16] = 32'H000C6902;  //  srl $t5,$t4,4 //$t5= 0
      Memory[17] = 32'H15090003;  //  bne $t0,$t1,0x020003 (PC=mem[21]X4=84 = 54(hex))
//doesn't execute
      Memory[18] = 32'H012C7024;  //  and $t6,$t1,$t4 //$t6 = 8 //from j instruction
      Memory[19] = 32'H012E7827;  //  nor $t7,$t1,$t6 //$t7 = FFFFFFF5
      Memory[20] = 32'H014F4025;  //  or $t0,$t2,$t7  //$t0 = FFFFFFFF
      Memory[21] = 32'H0C020000;  //  jal 0x020000 //22X4=88[mem]
      Memory[22] = 32'H01284022;  //  sub $t0,$t1,$t0 //$t0= B
      Memory[23] = 32'H01285020;  //  add $t2 $t1 $t0 //$t2= 15
      Memory[24] = 32'H81280005;  //  lb $t0,5($t1) (I type) //t1=A, A+5=F
```

```
    Memory[25] = 32'H03E00008;  //  jr $ra (R-type) //PC=memory[22]
    Memory[26] = 32'HAD280010;  //  sw $t0, 10($t1) // $t0 = 10+A = 1A
    Memory[27] = 32'HFFE00008;  //  s_inc // ALUOut = $t1+4 =A+4 = E

  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]; //PC divided by 4 to get memory word address
        PC <= PC + 4;
        state = 2;
      end

      2: begin    //second step: Instruction decode, register fetch, also compute branch address
        A <= Regs[IR[25:21]]; //rs
        B <= Regs[IR[20:16]]; //rt
        ALUOut <= PC + PCOffset;    // compute PC-relative branch target
        if(opcode == LW || opcode == LB || opcode == SW || opcode == I_FORMAT || opcode
== s_inc)
          state = 3;
        if (opcode == R_FORMAT)
          state = 7;
        if (opcode == BEQ)
          state = 9;
        if (opcode == BNE)
          state = 14;
        if (opcode == J_FORMAT)
          state = 10;
        if (opcode == Jal_FORMAT)
          state = 13;
      end

      3: begin    //third step:  Load/Store execution, ALU execution, Branch completion
        ALUOut <= A + SignExtend; //compute effective address
        if (opcode == LW)
          state = 4;
        if (opcode == LB)
          state = 12;
        if (opcode == SW)
          state = 6;
```

```
            if (opcode == I_FORMAT)
                state = 11;
            if (opcode == s_inc)
                state = 15;
        end

        4: begin    //LW
            MDR <= Memory[ALUOut>>2]; // read the memory
            state = 5; // next state
        end

        5: begin     //LW is the only instruction still in execution
            Regs[IR[20:16]] = MDR;      // write the MDR to the register
            state = 1;
        end //complete a LW instruction

        6: begin     //SW
            Regs[16] = A + 4;
            Memory[ALUOut>>2] <= B; // write the memory
            state = 1; // return to state 1
            //store finishes
        end

        7: begin
            case (IR[5:0]) //case for the various R-type instructions
                0: ALUOut = B << shift; //SLL operation
                2: ALUOut = B >> shift; //SRL operation
                8: ALUOut = Regs[31]; //JR operation
                32: ALUOut = A + B; //add operation
                34: ALUOut = A - B; //sub operation
                36: ALUOut = A & B; //AND operation
                37: ALUOut = A | B; //OR operation
                39: ALUOut = ~(A | B); //NOR operation
                42: begin // SLT
                    if (A[31] != B[31]) begin
                        if (A[31] > B[31]) begin
                            ALUOut <= 1;
                        end
                        else begin
                            ALUOut <= 0;
                        end
                    end
                    else begin
                        if (A < B)
                        begin
                            ALUOut <= 1;
```

```
                    end
                else
                    begin
                        ALUOut <= 0;
                    end
                end
            end
        default: ALUOut = A; //other R-type operations
    endcase
    state = 8;
end

8: begin     //R-FORMAT
    Regs[IR[15:11]] <= ALUOut; // write the result
    state = 1;
end

9: begin     //BEQ
    if (A == B)  begin
        PC <= ALUOut; // branch taken--update PC
        state = 1;  //  BEQ finished, return to first state
    end
end

10: begin     // J_FORMAT
    PC = {PC[31:28], IR[25:0], 2'b00};
    PC <=ALUOut;
    state = 1;
end

11: begin   //I_FORMAT
    Regs[IR[20:16]] <= ALUOut; // write the result
    state = 1;
end

12: begin   //LB
    MDR <= Memory[ALUOut]; // read the memory
    state = 5; // next state
end

13: begin //Jal_FORMAT
    Regs[31] = PC + 4; //return address is stored to register 31
    PC = {PC[31:28], IR[25:0], 2'b00};
    PC <=ALUOut;
    state = 1;
end
```

```
        14: begin //BNE
            if (A != B) begin
                PC <= ALUOut; // branch taken--update PC
                state = 1;  //  BEQ finished, return to first state
            end
        end

        15: begin //s_inc
            ALUOut = Regs[16];
            state = 1;
        end
    endcase

  end

endmodule
```

f. A copy of the Verilog testbench (yes another copy within the pdf file in addition to the separate .v file in 2)

```
//Author Names: Anika Tasnim, Prakash Sharma, Shubh Patel
//Last Modified Date: 10/29/2020
//Compilation Status: “Successful”
//Elaboration Status: “Successful”
//Simulation Status: “Simulates correctly”

module cpu_tb;

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

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

  initial begin
    clock = 0;
    repeat (200) //building a waveform with 100 cycles
      begin
        #1 clock = ~clock; //alternate clock signal
      end
    $finish;
  end

endmodule
```

g. The Assembly Test Program in a Well-Documented Form (in MIPS assembly language)

```
addi    $t0, $zero, 5    #$t0=$zero+5 = 5
sw      $t0, 124($zero) #storing $t0=5 in mem[124+$zero] or mem[7C_hex]
lw      $t1, 124($zero) #leading from mem[7C_hex] to $t1, $t1 = 5
beq     $t1,$t0,0x000003 #if $t0=$t1, brach after three memory location
sub     $t0,$t1,$t0 #$t0=0
add     $t2 $t1 $t0 # $t2 = 5
slt     $t3,$zero,$t1 # $t3 = 1
add     $t1,$t1,$t1 # after beq instruction, $t1=$t1+$t1 = 5+5 =A
sub     $t0,$t1,$t0 # $t0 = $t1-$t0 = A-5 = 5
add     $t2 $t1 $t0 # $t2 = $t1+$t0 = A+5 = F
slt     $t3,$zero,$t1 # $t3 = 1, as $zero < A
sll     $t4,$t2,2 # $t2 = F_hex = 1111_2, $t4 = 111100_2 = 3C_hex, F X 4 = 3C
srl     $t5,$t4,4 # $t4 = 3C_hex=111100_2, $t5 = 000011_2 = 3_hex,
j       0x020002 # jump two instruction after
srl     $t5,$t4,4
bne     $t0,$t1,0x020003 #branch 3 instruction after in $t0 != $t1
and     $t6,$t1,$t4 #after j instruction
nor     $t7,$t1,$t6
or      $t0,$t2,$t7 #after bne instruction, $t0 = FFFFFFFF
jal     0x020000 # jump to next instruction
sub     $t0,$t1,$t0 # $t0 = $t1-$t0 = A-FFFFFFFF = B //return address
add     $t2 $t1 $t0 # $t2 = A+B = 15
lb      $t0,5($t1) # loading byte from location (t1=A, A+5=F)
jr      $ra # going back to address of sub instruction
sw      $t0, 10($t1) # storing word in $t0 in mem location (10+A) = 1A
```

The last instruction is the s_inc instruction which consists of two instructions sw and addi. sw $rt, L($rs) and addi $rs, $rs, 4. To execute s_inc in a single instruction we have used an unused opcode of 6 bits 111111. And to execute this in a single instruction we save the $rs in a register after adding 4. So after s_inc instruction the output will be the next register.

s_inc # increased the instruction register after sw instruction, so ALUOut = $t1+4 =A+4 = E

h. All Data Memory Initializations

```
Memory[2] = 32'H20080005;  //  addi $t0,$zero,5// $t0=5

Memory[3] = 32'Hac08007C;  //  sw $t0, 124($zero) // from add = 7C

Memory[4] = 32'H8c09007C;  //  lw $t1, 124($zero)// &t1=5 //add = 7C
```

```
Memory[5] = 32'H11280003;  //  beq $t1,$t0,0x000003 (PC=9X4=36[mem])
Memory[6] = 32'H01284022;  //  sub $t0,$t1,$t0 //$t0=0
Memory[7] = 32'H01285020;  //  add $t2 $t1 $t0 //$t2=5
Memory[8] = 32'H0009582A;  //  slt $t3,$zero,$t1 // $t3=1
Memory[9] = 32'H01294820;  //  add $t1,$t1,$t1 // $t1=A //after beq instruction
Memory[10] = 32'H01284022;  //  sub $t0,$t1,$t0 //$t0=5
Memory[11] = 32'H01285020;  //  add $t2 $t1 $t0 //$t2=F
Memory[12] = 32'H0009582A;  //  slt $t3,$zero,$t1 // $t3=1
Memory[13] = 32'H000A6080;  //  sll $t4,$t2,2 //$t4= 3C
Memory[14] = 32'H000C6902;  //  srl $t5,$t4,4 //$t5= 3
Memory[15] = 32'H08020002;  //  j 0x020002 //PC=18X4=72
Memory[16] = 32'H000C6902;  //  srl $t5,$t4,4 //$t5= 0
Memory[17] = 32'H15090003;  //  bne $t0,$t1,0x020003 (PC=13X4=52[mem])
Memory[18] = 32'H012C7024;  //  and $t6,$t1,$t4 //$t6 = 8 //from j instruction
Memory[19] = 32'H012E7827;  //  nor $t7,$t1,$t6 //$t7 = FFFFFFF5
Memory[20] = 32'H014F4025;  //  or $t0,$t2,$t7  //$t0 = FFFFFFFF
Memory[21] = 32'H0C020000;  //  jal 0x020000 //22X4=88[mem]
Memory[22] = 32'H01284022;  //  sub $t0,$t1,$t0 //$t0= B
Memory[23] = 32'H01285020;  //  add $t2 $t1 $t0 //$t2= 15
Memory[24] = 32'H81280005;  //  lb $t0,5($t1) (I type) //t1=A, A+5=F
Memory[25] = 32'H03E00008;  //  jr $ra (R-type) //PC=memory[22]
Memory[26] = 32'HAD280010;  //  sw $t0, 10($t1) // $t0 = 10+A = 1A
Memory[27] = 32'HFFE00008;  //  s_inc // ALUOut = $t1+4 =A+4 = E
```

i. Detailed Description of the Tested Program

In Moore State Machine, the next state depends on both the current state and opcode. HOwever the corresponding control signals are independent of the opcodes. If the opcode requires additional computation or data storage, the program moves to the next state in the opcode series. We created 15 states to support all the MIPS instructions specified in the project document (add, addi, sub, and, nor, or, slt, sll, srl, beq, bne, j, jal, jr, lw, lb, sw). The program design follows the Moore finite state which allows more flexibility and more states as opposed to mealy machines.

j. Detailed Strategy for Verifying Correctness of the Design

Our way of verifying the correctness of the CPU design was primarily through the Qtspim software. In QtSpim software, our first step was to write an assembly code in MIPS and get an output. Once the results were obtained, we were able to verify them by the states on the waveform. Since our waveforms show expected results we were able to confirm that our design is accurate.

k. Expected Output of the Test Program as Obtained from QtSpim

```
addi $t0,$zero,5// $t0=5

sw $t0, 124($zero) // from add = 7C

lw $t1, 124($zero)// &t1=5 //add = 7C

beq $t1,$t0,0x000003 (PC=9X4=36[mem])

sub $t0,$t1,$t0 //$t0=0

add $t2 $t1 $t0 //$t2=5

slt $t3,$zero,$t1 // $t3=1

add $t1,$t1,$t1 // $t1=A //after beq instruction

sub $t0,$t1,$t0 //$t0=5

add $t2 $t1 $t0 //$t2=F

slt $t3,$zero,$t1 // $t3=1

sll $t4,$t2,2 //$t4= 3C

srl $t5,$t4,4 //$t5= 3

j 0x020002 //PC=18X4=72
```

srl $t5,$t4,4 //$t5= 0

bne $t0,$t1,0x020003 (PC=13X4=52[mem])

and $t6,$t1,$t4 //$t6 = 8 //from j instruction

nor $t7,$t1,$t6 //$t7 = FFFFFFF5

or $t0,$t2,$t7 //$t0 = FFFFFFFF

jal 0x020000 //22X4=88[mem]

sub $t0,$t1,$t0 //$t0= B

add $t2 $t1 $t0 //$t2= 15

lb $t0,5($t1) (I type) //t1=A, A+5=F

jr $ra (R-type) //PC=memory[22]

sw $t0, 10($t1) // $t0 = 10+A = 1A

s_inc // ALUOut = $t1+4 =A+4 = E

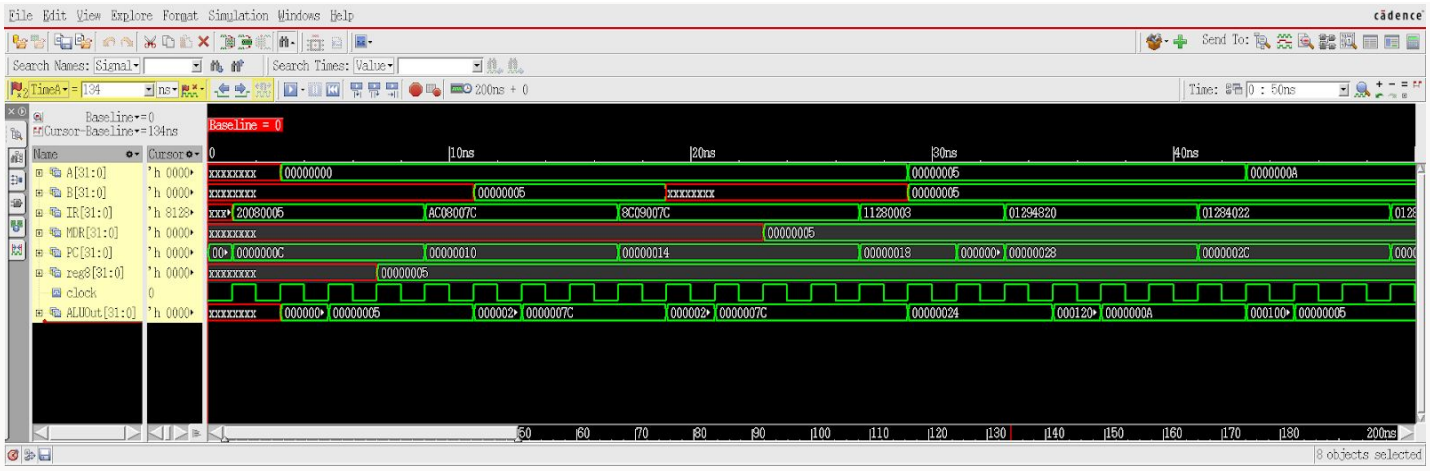
l. Actual Output of the Test Program as Obtained from the Waveforms

The output from the test program obtained from the waveform is the same as the output from the QtSpim.

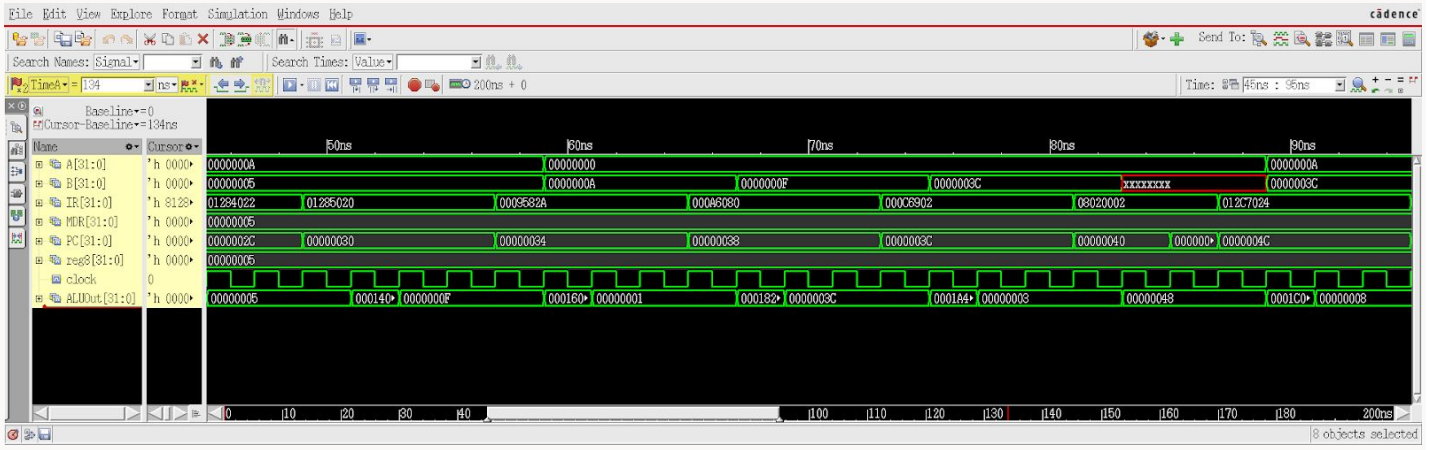
m.Simulation Waveforms.

The result of the instructions is shown below in the waveforms:

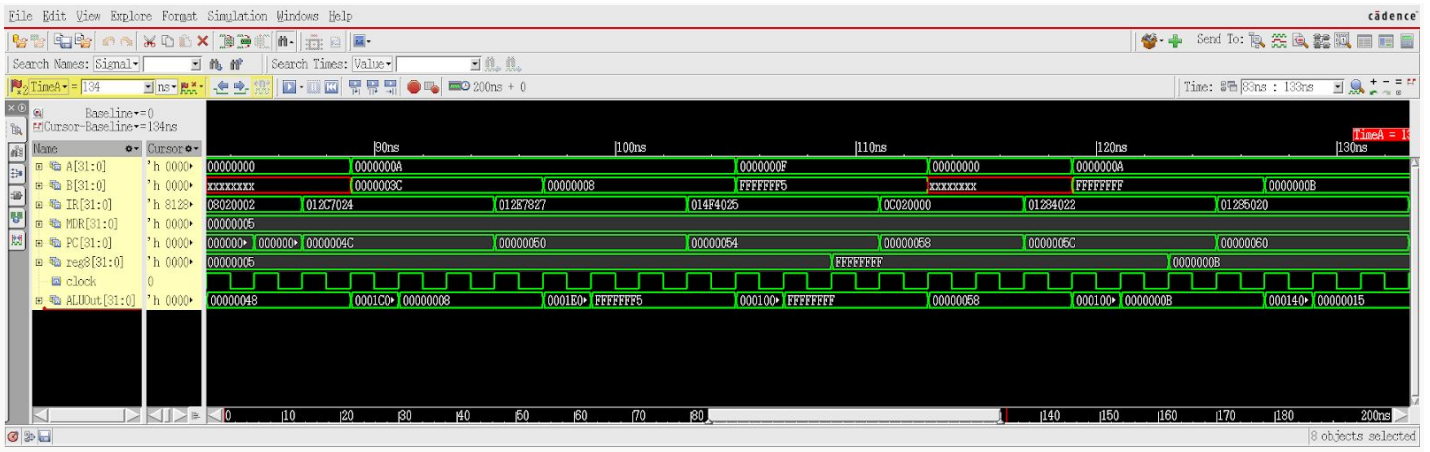
Waveform 1:



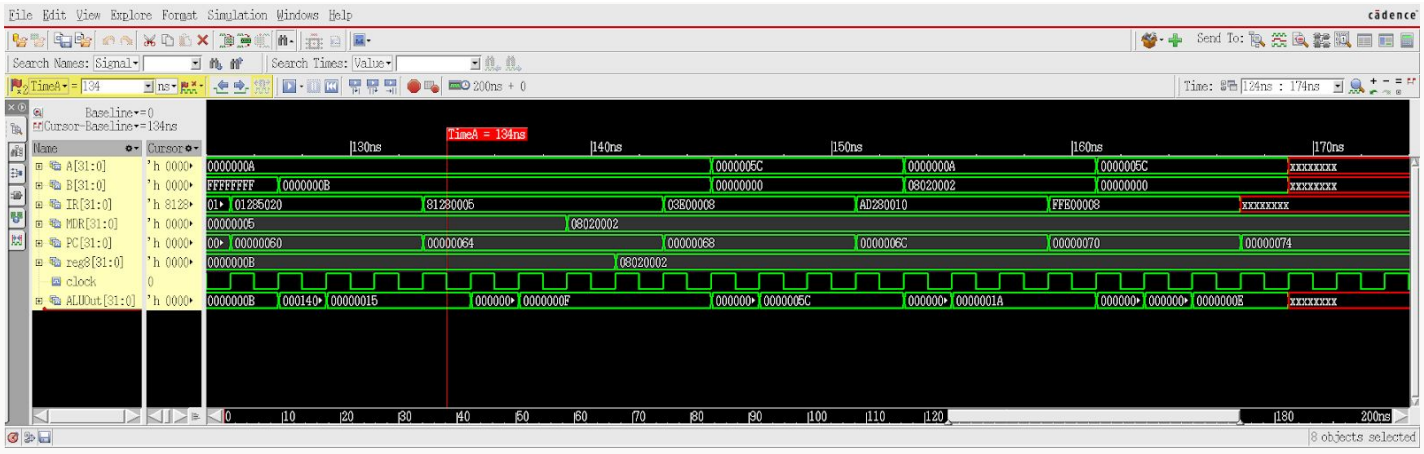
Waveform 2:



Waveform 3:



Waveform: 4


File Edit View Explore Format Simulation Windows Help
cadence
Search Names: Signal
Search Times: Value
Send To:
TimeA = 134
ns
200ns + 0
Time: 124ns : 174ns
Baseline=0
Cursor-Baseline=134ns
TimeA = 134ns
Name
Cursor
130ns
140ns
150ns
160ns
170ns
A[31:0]
B[31:0]
IR[31:0]
MDR[31:0]
PC[31:0]
reg8[31:0]
clock
ALUOut[31:0]
0000000A
0000005C
0000000A
0000005C
FFFFFFFF
0000000B
00000000
08020002
00000000
01285020
81280005
03B00008
AD280010
FFE00008
00000005
08020002
00000060
00000064
00000068
0000006C
00000070
00000074
0000000B
08020002
0000000B
000140
00000015
000000
0000000F
000000
0000005C
000000
0000001A
000000
000000
0000000E
0
10
20
30
40
50
60
70
80
90
100
110
120
180
200ns
8 objects selected