Lab 2

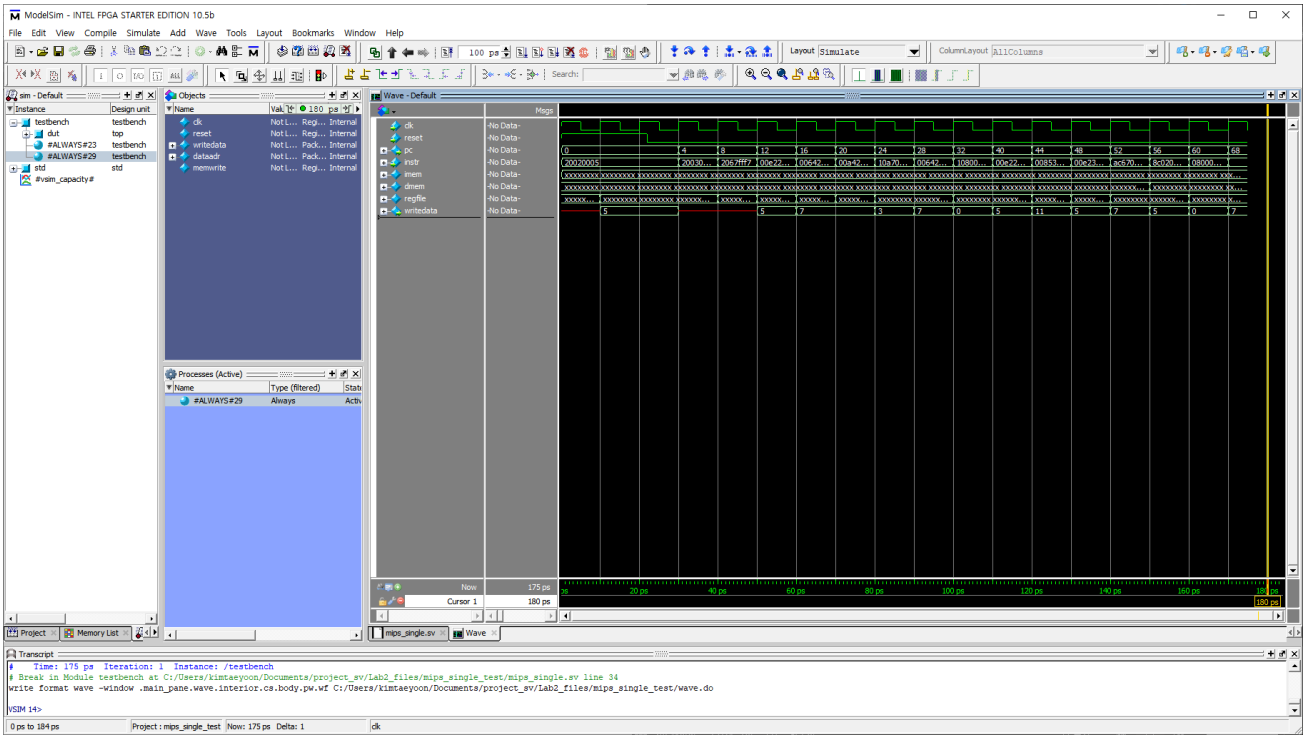
MIPS Assembly Programming

Date : 2021.04/09
NAME : 김태윤
Student Number : 2019707033

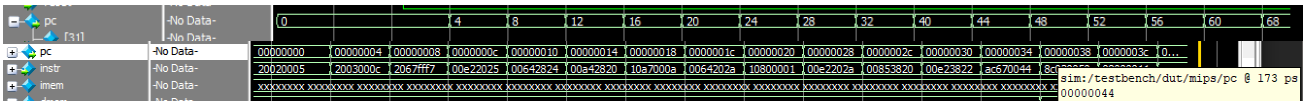
Contents

1. Simulation of MIPS Single Cycle Processor

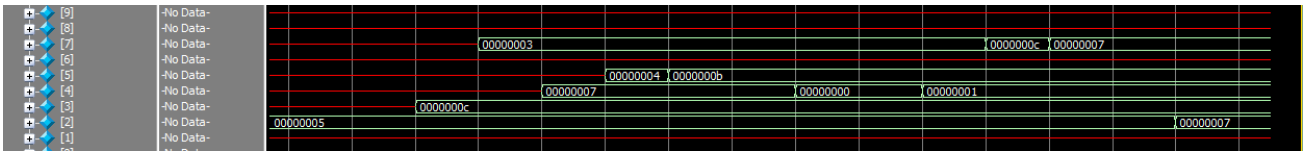
A. Result of Simulation



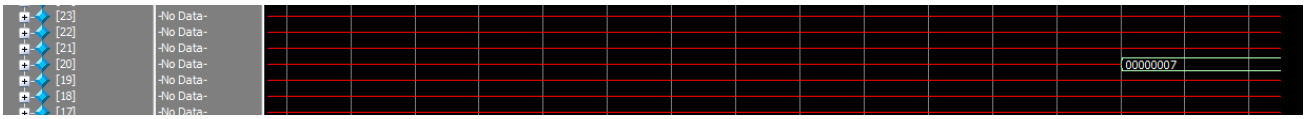
Full shot of Simulation layout on my computer.



We can see program counter was working well like guide.



Also, Register file was working well.



Dmem, which have to be 7 to make interrupt to process, was working well too.

2. Simulation of Fibonacci

A. Result of Simulation

- Modification

```
// check results
always @(negedge clk)
  begin
    if(memwrite) begin
      if(dataadr === 84 & writedata === 7) begin
        $display("Simulation succeeded");
        $stop;
      end else if (dataadr !== 80) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

->

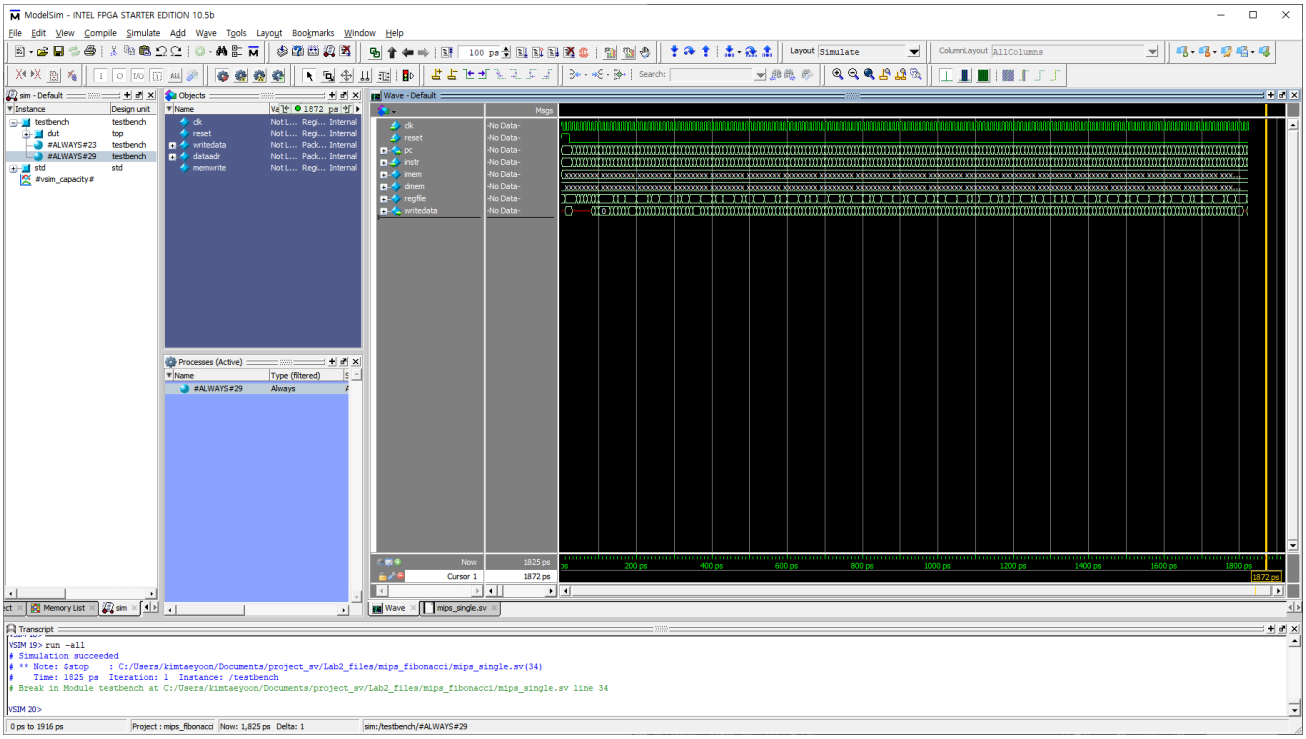
```
// check results
always @(negedge clk)
  begin
    if(memwrite) begin
      if(dataadr === 84 & writedata === 10) begin
        $display("Simulation succeeded");
        $stop;
      end else if (dataadr !== 80) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

```
module imem(input  logic [5:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("testfile.dat",RAM);

  assign rd = RAM[a]; // word aligned
endmodule
```

->

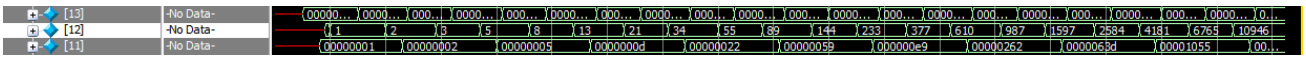
```
module imem(input  logic [5:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
      $readmemh("fibonacci.dat",RAM);

  assign rd = RAM[a]; // word aligned
endmodule
```

- Simulation



With wave.do, setting up Simulation's environment was very comfort.

```
VSIM 19> run -all
# Simulation succeeded
# ** Note: $stop    : C:/Use
#    Time: 1825 ps  Iteratic
# Break in Module testbench

VSIM 20>
```



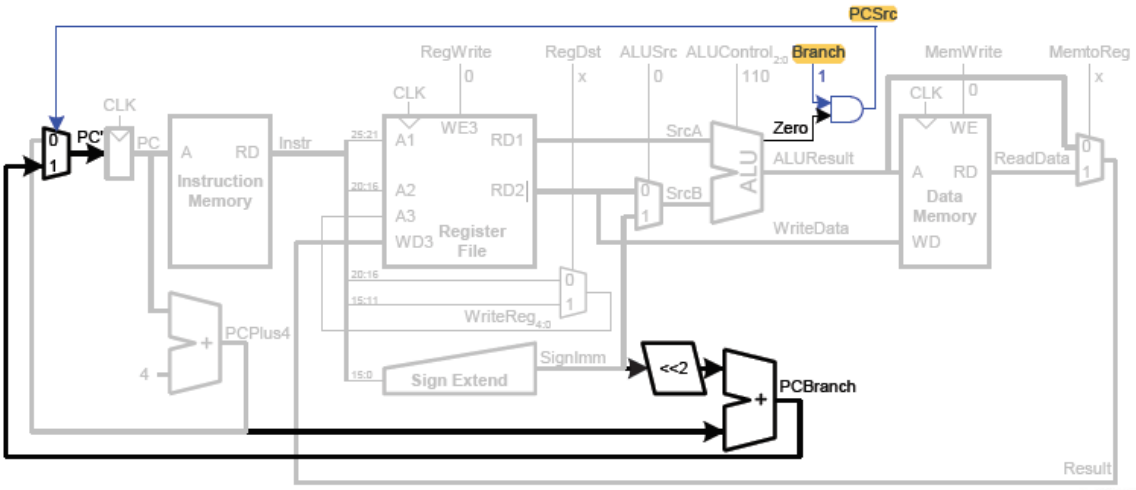
As we can see from $12, Fibonacci processing was working well. {1, 2, 3, 5, 8, 13, 21, 34, 55, ..., 10946} was generated.

Also, 10 was generated from "writedata", so simulations was terminated successfully.

3. HOMEWORK; Extend Simple MIPS Single Cycle Processor

A. Implementing "bne"

First, we have to know what we have to change for implement "bne". Also, we have to understand how beq, bne works.



In above datapath, it features how beq instruction works. On the top of this datapath, PCSrc(Program Counter source) is generated by Control signal(Branch) and Zero register's & operations's result.

Maybe, to implement bne, we can make it to modify that operation parts a little bit. BEQ(branch equal) works when equal, and BNE(Branch not equal) works when not equal. And it's each opcode is 000100, 000101. I think, I can implement bne just with opcode's lsb and operating part of branch and zero.

1. Show the modifications on the simple MIPS single cycle (“mips_single.sv”) System Verilog code.

```
// mips.sv
// From Section 7.6 of Digital Design & Computer Architecture
// Updated to SystemVerilog 26 July 2011 David_Harris@hmc.edu
// ---------------- ---------------- ---------------- ----------------
module testbench();

  logic          clk;
  logic          reset;

  logic [31:0] writedata, dataadr;
  logic          memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);

  // 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 === 84 & writedata === 10) begin
          $display("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display("Simulation failed");
          $stop;
        end
```

```
        end
      end
endmodule


// ---------------- ---------------- ---------------- ----------------
module top(input    logic              clk, reset,
                output logic [31:0] writedata, dataadr,
                output logic              memwrite);

  logic [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr,
                writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);
endmodule


// ---------------- ---------------- ---------------- ----------------
module dmem(input    logic              clk, we,
                input    logic [31:0] a, wd,
                output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule


// ---------------- ---------------- ---------------- ----------------
module imem(input    logic [5:0] a,
                output logic [31:0] rd);

  logic [31:0] RAM[63:0];
```

```
  initial
      $readmemh("fibonacci_bne.dat",RAM);

  assign rd = RAM[a]; // word aligned
endmodule


// ---------------- ---------------- ---------------- ----------------
module mips(input   logic          clk, reset,
            output logic [31:0] pc,
            input   logic [31:0] instr,
            output logic          memwrite,
            output logic [31:0] aluout, writedata,
            input   logic [31:0] readdata);

  logic        memtoreg, alusrc, regdst,
               regwrite, jump, pcsrc, zero;
  logic [2:0] alucontrol;

  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata);
endmodule


// ---------------- ---------------- ---------------- ----------------
module controller(input   logic [5:0] op, funct,
                  input   logic        zero,
                  output logic        memtoreg, memwrite,
                  output logic        pcsrc, alusrc,
                  output logic        regdst, regwrite,
                  output logic        jump,
                  output logic [2:0] alucontrol);
```

```
  logic [1:0] aluop;
  logic       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump, aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = op[0] ? (branch & ~zero) : (branch & zero); //      ----      (Changed)
                 // Beq is already working.
                 // BEQ's op is 000100,
                 // BNE's op is 000101.
                 // As well, maybe we can use difference of lsb.
                 // So if lsb is 0,   -> for beq
                 // And if lsb is 1, -> for bne
endmodule


// ---------------- ---------------- ---------------- ----------------
module maindec(input   logic [5:0] op,
               output logic        memtoreg, memwrite,
               output logic        branch, alusrc,
               output logic        regdst, regwrite,
               output logic        jump,
               output logic [1:0] aluop);

  logic [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b000101: controls <= 9'b000100001; // BNE      ----      (Changed)
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
```

```
    endcase
endmodule


// ---------------- ---------------- ---------------- ----------------
module aludec(input   logic [5:0] funct,
              input   logic [1:0] aluop,
              output logic [2:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b010;   // add (for lw/sw/addi)
      2'b01: alucontrol <= 3'b110;   // sub (for beq)
      default: case(funct)           // R-type instructions
          6'b100000: alucontrol <= 3'b010; // add
          6'b100010: alucontrol <= 3'b110; // sub
          6'b100100: alucontrol <= 3'b000; // and
          6'b100101: alucontrol <= 3'b001; // or
          6'b101010: alucontrol <= 3'b111; // slt
          default:   alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule


// ---------------- ---------------- ---------------- ----------------
module datapath(input   logic          clk, reset,
                input   logic          memtoreg, pcsrc,
                input   logic          alusrc, regdst,
                input   logic          regwrite, jump,
                input   logic [2:0]    alucontrol,
                output logic           zero,
                output logic [31:0] pc,
                input   logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input   logic [31:0] readdata);

  logic [4:0]   writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
```

```
  logic [31:0] srca, srcb;
  logic [31:0] result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
                 writereg, result, srca, writedata);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb, alucontrol, aluout, zero);
endmodule


// ---------------- ---------------- ---------------- ----------------
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clk
  // register 0 hardwired to 0
  // note: for pipelined processor, write third port
```

```
  // on falling edge of clk

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule


// ---------------- ---------------- ---------------- ----------------
module adder(input   logic [31:0] a, b,
                     output logic [31:0] y);

  assign y = a + b;
endmodule


// ---------------- ---------------- ---------------- ----------------
module sl2(input   logic [31:0] a,
                   output logic [31:0] y);

  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule


// ---------------- ---------------- ---------------- ----------------
module signext(input   logic [15:0] a,
                       output logic [31:0] y);

  assign y = {{16{a[15]}}, a};
endmodule


// ---------------- ---------------- ---------------- ----------------
module flopr #(parameter WIDTH = 8)
                      (input   logic                  clk, reset,
                       input   logic [WIDTH-1:0] d,
                       output logic [WIDTH-1:0] q);
```

```
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule


// ---------------- ---------------- ---------------- ----------------
module mux2 #(parameter WIDTH = 8)
             (input   logic [WIDTH-1:0] d0, d1,
              input   logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule


// ---------------- ---------------- ---------------- ----------------
module alu(input   logic [31:0] a, b,
           input   logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;

  assign condinvb = alucontrol[2] ? ~b : b;
  assign sum = a + condinvb + alucontrol[2];

  always_comb
    case (alucontrol[1:0])
      2'b00: result = a & b;
      2'b01: result = a | b;
      2'b10: result = sum;
      2'b11: result = sum[31];
    endcase

  assign zero = (result == 32'b0);
endmodule
```

2. Indicate the changed that you made to implement the new instruction.

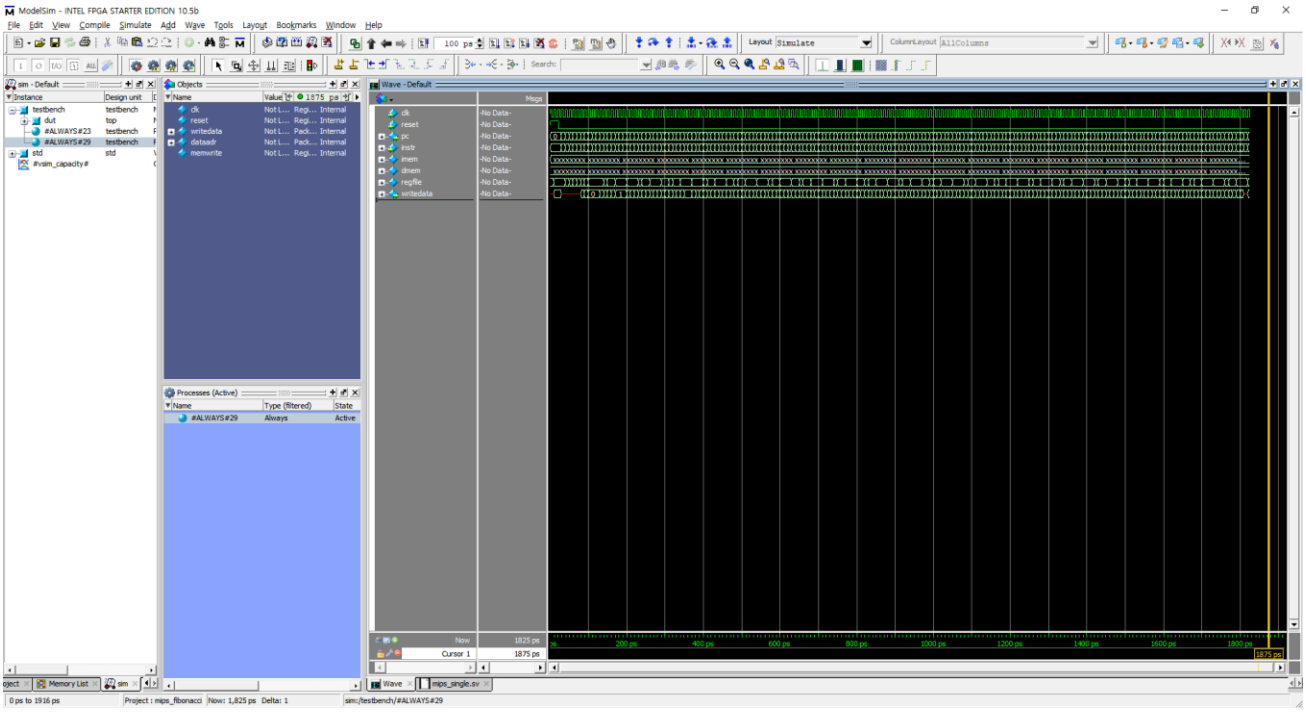
```
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

  logic [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b000101: controls <= 9'b000100001; // BNE    ----    (Changed)
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

I added "6'b000101: controls <= 9'b000100001. It means when opcode is 000101, push 000100001 to control signal so bne works well.

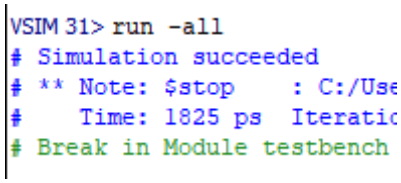
```
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump, aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = op[0] ? (branch & ~zero) : (branch & zero); //    ----    (Changed)
                // Beq is already working.
                // BEQ's op is 000100,
                // BNE's op is 000101.
                // As well, maybe we can use difference of lsb.
                // So if lsb is 0,  -> for beq
                // And if lsb is 1, -> for bne
endmodule
```

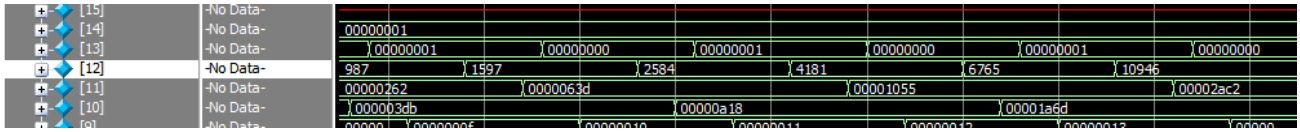
BEQ's opcode is 000100. BNE's opcode is 000101. Just lsb is different. So I think that using ternary operation is the best way. When op[0](LSB) is 1, pcsrc goes branch & ~zero, and op[0](LSB) is 0, pcsrc goes branch & zero. It works well.

3. Show the output wave simulation that contains f[21]=10946 in register $12 and the value 10 in writedata variable.

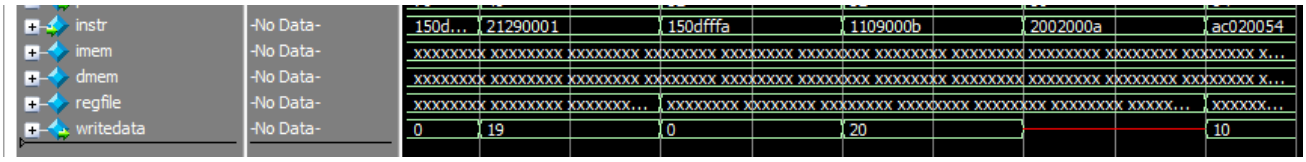


Full of my Simulation layout(Fibonacci_bne)





We can see ..., {1597, 2584, 4181, 6765, 10946}. It seems works very well.



We can see writedata's last value is 10, so simulation termintated successfully.