Computer Architecture

Lab 2

HOMEWORK: Single cycle MIPS implementation

학번 : 2016707079 이름 : 하상천

1. Show the full modified simple MIPS single cycle ("mips_single.sv") System Verilog code.

방법 1

```
// 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, temp1, temp2;

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

  assign temp1 = branch & zero;
  assign temp2 = aluop[0] & aluop[1] & (~zero);
  assign pcsrc = temp1 | temp2;
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'b000000011; // BNE
      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)
      2'b10: 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
      2'b11: alucontrol <= 3'b110;
      default: alucontrol <= 3'bxxx;
    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
```

방법 2

```
// 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, bne, temp1, temp2;

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

  assign temp1 = branch & zero;
  assign temp2 = bne & (~zero);
  assign pcsrc = temp1 | temp2;
endmodule

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

  logic [9:0] controls;

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

```
  always_comb
    case(op)
      6'b000000: controls <= 10'b1100000010; // RTYPE
      6'b100011: controls <= 10'b1010001000; // LW
      6'b101011: controls <= 10'b0010010000; // SW
      6'b000100: controls <= 10'b0001000001; // BEQ
      6'b000101: controls <= 10'b0000100001; // BNE
      6'b001000: controls <= 10'b1010000000; // ADDI
      6'b000010: controls <= 10'b0000000100; // J
      default:   controls <= 10'bxxxxxxxxxx; // 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, bne)
      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
```

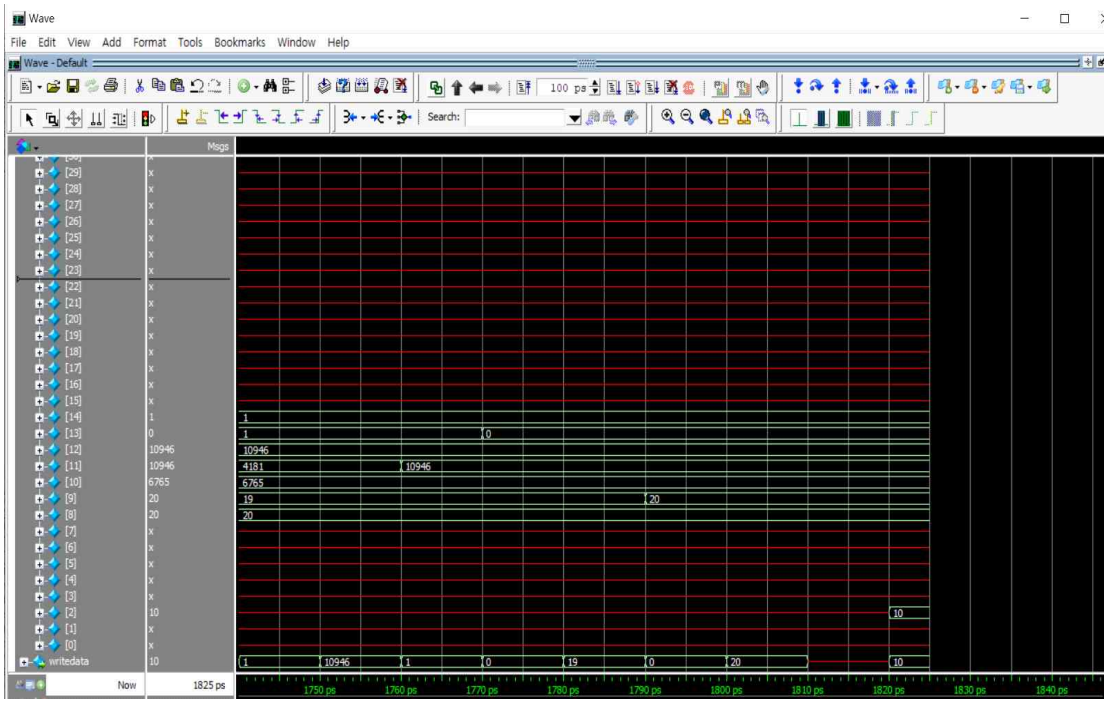
```
  // read two ports combinationally
  // write third port on rising edge of clk
  // register 0 hardwired to 0
  // note: for pipelined processor, write third port
  // on falling edge of clk

  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;
```

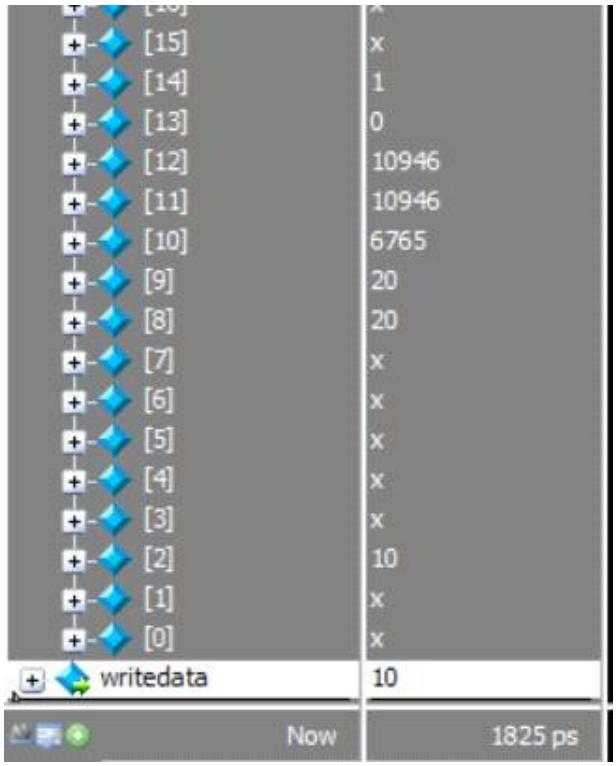
```
endmodule

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 changes that you made to implement the new instruction.

bne과 beq instruction은 ALU에서 뺄셈연산이 이뤄진다는 점은 같지만, zero값이 어떤 값이냐에 따라 branch가 1이 되는지 아닌지에 대한 연산은 반대가 된다.
방법 1에서는 사용하지 않는 aluop 코드 11을 사용했다. branch & zero를 temp1에 저장하고, alu[0] & alu[1] & (~zero)를 temp2에 저장한 다음 pcsrc = temp1 | temp2를 통해서 pcsrc 신호를 제어했다. bne일 때 alu[0], alu[1]이 1이기 때문에 (~zero)가 1이 되면, 즉 rs와 rt가 다르다면 pcsrc가 1이 된다. control 신호로는 regwrite 0, regdst 0 , alusrc 0, branch 0, memwrite 0, memtoreg 0, jump 0, aluop 11을 사용했다. 원래는 pcsrc = branch & zero 였는데 and gate 두개와 or gate 하나와 not gate 하나를 더 사용했다. 이 방법을 사용하면 control 신호를 하나 더 만들지 않기 때문에 메모리나 비용 측면에서 더 효율적이라고 생각한다.
방법 2에서는 bne이라는 control 신호를 하나 더 사용해서 regwrite 0, regdst 0 , alusrc 0, branch 0, bne 1, memwrite 0, memtoreg 0, jump 0, aluop 01을 사용했다. branch & zero를 temp1에 저장하고, bne & (~zero)를 temp2에 저장한 다음 pcsrc = temp1 | temp2를 통해서 pcsrc 신호를 제어했다. 원래는 pcsrc = branch & zero 였는데 and gate 하나와 or gate 하나와 not gate 하나를 더 사용했다.
하지만 두 방법 모두 1825ps에 프로그램이 마무리 되었다. 방법 1이 연산이 더 많기 때문에 더 오래 걸릴 것이라고 생각했지만, 실행결과를 확인해보니 똑같은 시간에 마무리 되었음을 알 수 있었다. beq instruction에서 rs와 rt가 같을 때, bne instruction에서 rs와 rt가 다를 때 program counter + 4 와 immediate를 sign extension하고 4를 곱한 값을 더한 값으로 이동한다.

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


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
writedata
Now
1825 ps


[15] x
[14] 1
[13] 0
[12] 10946
[11] 10946
[10] 6765
[9] 20
[8] 20
[7] x
[6] x
[5] x
[4] x
[3] x
[2] 10
[1] x
[0] x
writedata 10
Now 1825 ps