Report for Lab4: Multi-cycle CPU

20140843 Taekang Eom

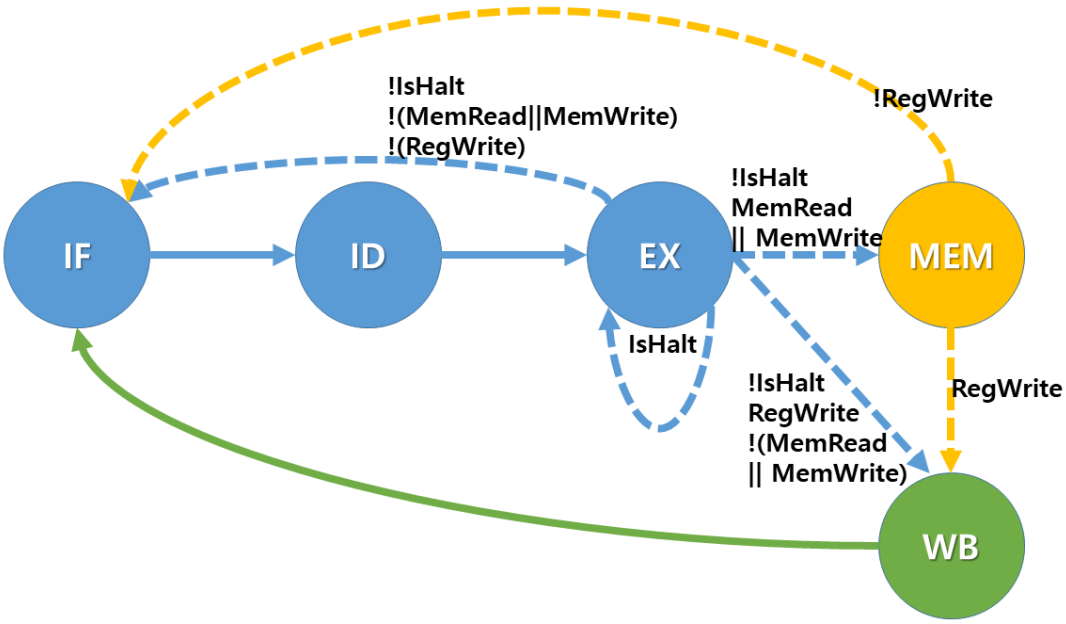
20150384 Eunyoung Hyung

1. Introduction

In this lab, we need to understand the reason of transferring from a single-cycle implementation to a multi-cycle implementation. To do this, we need to design and implement a multi-cycle CPU, which fully support TSC instruction set without RWD, ENI, DSI.

2. Design

1) Finite State Machine Transition Diagram



2) Resource Reuse

We combine all "pc+1" logic with one Adder

```
assign Imm_next_pc = pc + 16'b1;
```

3. Implementation

1) ALU.v is same as the one used in single cycle cpu, so we omit it.

2) opcodes.v

```
`define WORD_SIZE   16
`define NUM_REGS    4
// Opcode
`define ALU_OP  4'd15
`define ADI_OP  4'd4
`define ORI_OP  4'd5
`define LHI_OP  4'd6
`define LWD_OP  4'd7
`define SWD_OP  4'd8
`define BNE_OP  4'd0
`define BEQ_OP  4'd1
`define BGZ_OP  4'd2
`define BLZ_OP  4'd3
`define JMP_OP  4'd9
`define JAL_OP  4'd10
`define JPR_OP  4'd15
`define JRL_OP  4'd15
// ALU Function Codes
`define FUNC_ADD    3'b000
`define FUNC_SUB    3'b001
`define FUNC_AND    3'b010
`define FUNC_ORR    3'b011
`define FUNC_NOT    3'b100
`define FUNC_TCP    3'b101
`define FUNC_SHL    3'b110
`define FUNC_SHR    3'b111
// ALU instruction function codes
`define INST_FUNC_ADD 6'd0
`define INST_FUNC_SUB 6'd1
`define INST_FUNC_AND 6'd2
`define INST_FUNC_ORR 6'd3
`define INST_FUNC_NOT 6'd4
`define INST_FUNC_TCP 6'd5
`define INST_FUNC_SHL 6'd6
`define INST_FUNC_SHR 6'd7
`define INST_FUNC_JPR 6'd25
`define INST_FUNC_JRL 6'd26
`define INST_FUNC_WWD 6'd28
`define INST_FUNC_HALT 6'd29
// State
```

```
`define IF  3'd1
`define ID  3'd2
`define EX  3'd3
`define MEM 3'd4
`define WB  3'd5
```

3) control.v

```
`include "opcodes.v"
module control(
    input wire [`WORD_SIZE-1:0] instruction,
    input wire reset_n,
    input wire clk,
    output reg Jump,
    output reg JALorJALR,//0 if JAL, 1 if JALR(using registers like JPR, JRL)
    output reg Branch,
    output reg MemRead,
    output reg MemtoReg,
    output reg MemWrite,
    output reg PCtoReg,
    output reg ALUSrc,
    output reg RegWrite,
    output reg Halt,
    output reg OpenPort,
    output reg [3:0] ALUOp,
    output reg [1:0] rs1,
    output reg [1:0] rs2,
    output reg [1:0] rd,
    output reg [2:0] State
);
    wire [3:0]opcode;
    wire [5:0]funct;
    assign opcode = instruction[15:12];
    assign funct = instruction[5:0];

    initial begin
        State <= `IF;
    end
    always @(posedge clk) begin
        if(!reset_n) begin
            State <= `IF;
        end
        case(State)
        `IF: State <= `ID;
        `ID: State <= `EX;
        `EX: begin
            if(Halt) begin // Halt
```

```
                State <= `EX;
            end
            if(MemRead || MemWrite) begin // Load or Store
                State = `MEM;
            end
            else if(RegWrite) begin
            // Instruction which does not need memory access
            // but needs to write back in register
            // Ex: R type, I type, Jump(which does write back to Register
file)
                State = `WB;
            end
            else begin
            // Instruction which does not need neither memory access
            // nor write back in register
            // Branch or Jump(which does not write back to Register file)
                State <= `IF;
            end
        end
        `MEM: begin
            if(RegWrite) begin // Load
                    State <= `WB;
            end
            else begin // Store
                State <= `IF;
            end
        end
        `WB: State <= `IF;
        endcase
    end

    always @(*) begin
        if (opcode == 4'd15) begin // if R
            // if ADD, SUB, AND, ORR, NOT, TCP, SHL, SHR
           if (funct < 6'd8) begin
                Jump <= 0;
                JALorJALR <= 1'bx;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                ALUOp <= funct; // funct indicates which operation should be
done
                MemWrite <= 0;
                PCtoReg <= 0;
                ALUSrc <= 0;
                RegWrite <= 1; // rd <-- alu result
                rs1 <= instruction[11:10];
                rs2 <= instruction[09:08];
```

```
            rd <= instruction[07:06];
            Halt <= 0;
            OpenPort <= 0;
        end
        else if (funct == `INST_FUNC_JPR) begin // JPR
            Jump <= 1; // this is jump
            JALorJALR <= 1; // using register
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            ALUOp <= 4'b1000; // C = A
            MemWrite <= 0;
            PCtoReg <= 0; // $pc <-- $rs
            ALUSrc <= 0;
            RegWrite <= 0;
            rs1 <= instruction[11:10];
            rs2 <= 2'bxx;
            rd <= 2'bxx;
            Halt <= 0;
            OpenPort <= 0;
        end
        else if (funct == `INST_FUNC_JRL) begin //JRL
            Jump <= 1; // this is jump
            JALorJALR <= 1; // using register
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            ALUOp <= 4'b1000; // C = A
            MemWrite <= 0;
            PCtoReg <= 1; // $2 <-- $pc $pc <-- $rs
            ALUSrc <= 0;
            RegWrite <= 1;
            rs1 <= instruction[11:10];
            rs2 <= 2'bxx;
            rd <= 2'b10;
            Halt <= 0;
            OpenPort <= 0;
        end
         else if (funct == `INST_FUNC_WWD) begin //WWD
            Jump <= 0; // this is jump
            JALorJALR <= 0; // using register
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            ALUOp <= 4'b1000; // C = A
            MemWrite <= 0;
            PCtoReg <= 0;
            ALUSrc <= 0;
```

```
                RegWrite <= 0;
                rs1 <= instruction[11:10];
                rs2 <= 2'bxx;
                rd <= 2'bxx;
                Halt <= 0;
                OpenPort <= 1;
            end
             else if (funct == `INST_FUNC_HALT) begin //HALT
                Jump <= 1'bx;
                JALorJALR <= 1'bx;
                Branch <= 1'bx;
                MemRead <= 1'bx;
                MemtoReg <= 1'bx;
                ALUOp <= 4'bxxxx;
                MemWrite <= 1'bx;
                PCtoReg <= 1'bx;
                ALUSrc <= 1'bx;
                RegWrite <= 1'bx;
                rs1 <= 2'bxx;
                rs2 <= 2'bxx;
                rd <= 2'bxx;
                Halt <= 1;
                OpenPort <= 0;
            end
            else begin //(RWD, ENI, DSI)
                Jump <= 1'bx;
                JALorJALR <= 1'bx;
                Branch <= 1'bx;
                MemRead <= 1'bx;
                MemtoReg <= 1'bx;
                ALUOp <= 4'bxxxx;
                MemWrite <= 1'bx;
                PCtoReg <= 1'bx;
                ALUSrc <= 1'bx;
                RegWrite <= 1'bx;
                rs1 <= 2'bxx;
                rs2 <= 2'bxx;
                rd <= 2'bxx;
                Halt <= 0;
                OpenPort <= 0;
            end
        end
        else if (opcode < 4'b0100) begin //if Bxx
            Jump <= 0;
            JALorJALR <= 1'bx;
            Branch <= 1;
            MemRead <= 0;
            MemtoReg <= 0;
```

```
            ALUOp <= 4'b0000;
            MemWrite <= 0;
            PCtoReg <= 0;
            ALUSrc <= 1; //{ (offset7)8 ## offset7..0 }
            RegWrite <= 0;
            rs1 <= instruction[11:10];
            rs2 <= instruction[09:08];
            rd <= 2'bxx;
            Halt <= 0;
            OpenPort <= 0;
        end
        else if (opcode < 4'd7) begin //if ADI,ORI,LHI
            Jump <= 0;
            JALorJALR <= 1'bx;
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            MemWrite <= 0;
            PCtoReg <= 0;
            ALUSrc <= 1; // { (imm7)8 ## imm7..0 }
            RegWrite <= 1; // $rt <--
            rs1 <= instruction[11:10];
            rs2 <= 2'bxx;
            rd <= instruction[09:08]; // $rt <--
            Halt <= 0;
            OpenPort <= 0;
            case (opcode)
                `ADI_OP : ALUOp <= 4'b0000; // ADI
                `ORI_OP : ALUOp <= 4'b0011; // ORI
                `LHI_OP : ALUOp <= 4'b1001; // LHI
                default : ALUOp <= 4'b1111;
            endcase
        end
        else if (opcode == `LWD_OP) begin //if LWD
            Jump <= 0;
            JALorJALR <= 1'bx;
            Branch <= 0;
            MemRead <= 1; // M[$rs + ]
            MemtoReg <= 1; // $rt <-- M[]
            ALUOp <= 4'b0000;
            MemWrite <= 0;
            PCtoReg <= 0;
            ALUSrc <= 1; // { (offset7)8 ## offset7..0 }
            RegWrite <= 1; // $rt <--
            rs1 <= instruction[11:10];
            rs2 <= 2'bxx;
            rd <= instruction[09:08];
            Halt <= 0;
```

```
            OpenPort <= 0;
        end
        else if (opcode == `SWD_OP) begin //if SWD
            Jump <= 0;
            JALorJALR <= 1'bx;
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            ALUOp <= 4'b0000;
            MemWrite <= 1; // M[$rs + { (offset7)8 ## offset7..0 }] <-- $rt
            PCtoReg <= 0;
            ALUSrc <= 1; // M[$rs + { (offset7)8 ## offset7..0 }] <-- $rt
            RegWrite <= 0;
            rs1 <= instruction[11:10];
            rs2 <= instruction[09:08];
            rd <= 2'bxx;
            Halt <= 0;
            OpenPort <= 0;
        end
        else if (opcode == `JMP_OP) begin //if JMP
            // $pc <-- $pc15..12 ## target11..0
            Jump <= 1;
            JALorJALR <= 0;
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            ALUOp <= 4'b1010; // C = {A[15:12],B[11:0]};
            MemWrite <= 0;
            PCtoReg <= 0;
            ALUSrc <= 1; // $pc15..12 ## target11..0
            RegWrite <= 0;
            rs1 <= 2'bxx;
            rs2 <= 2'bxx;
            rd <= 2'bxx;
            Halt <= 0;
            OpenPort <= 0;
        end
        else if (opcode == `JAL_OP) begin //if JAL
            // $2 <-- $pc $pc <-- $pc15..12 ## target11..0
            Jump <= 1;
            JALorJALR <= 0;
            Branch <= 0;
            MemRead <= 0;
            MemtoReg <= 0;
            ALUOp <= 4'b1010; //C = {A[15:12],B[11:0]};
            MemWrite <= 0;
            PCtoReg <= 1; // $2 <-- $pc
            ALUSrc <= 1; // $pc15..12 ## target11..0
```

```
            RegWrite <= 1; // $2 <--
            rs1 <= 2'bxx;
            rs2 <= 2'bxx;
            rd <= 2'b10; // $2 <--
            Halt <= 0;
            OpenPort <= 0;
        end
        else begin // when opcode does not belong to any case
            Jump <= 1'bx;
            JALorJALR <= 1'bx;
            Branch <= 1'bx;
            MemRead <= 1'bx;
            MemtoReg <= 1'bx;
            ALUOp <= 4'bxxxx;
            MemWrite <= 1'bx;
            PCtoReg <= 1'bx;
            ALUSrc <= 1'bx;
            RegWrite <= 1'bx;
            rs1 <= 2'bxx;
            rs2 <= 2'bxx;
            rd <= 2'bxx;
            Halt <= 0;
            OpenPort <= 0;
        end
    end
endmodule
```

4) cpu.v

```
`timescale 1ns/1ns
`include "opcodes.v"

module cpu(clk, reset_n, readM, writeM, address, data, num_inst, output_port,
is_halted);
    input clk;
    input reset_n;

    output readM;
    output writeM;
    output [`WORD_SIZE-1:0] address;

    inout [`WORD_SIZE-1:0] data;

    output [`WORD_SIZE-1:0] num_inst;       // number of instruction during
execution (for debuging & testing purpose)
    output [`WORD_SIZE-1:0] output_port;    // this will be used for a "WWD"
instruction
```

```
    output is_halted;

    // TODO : Implement your multi-cycle CPU!
    reg [`WORD_SIZE-1:0] num_inst;
    reg [`WORD_SIZE-1:0] output_port;
    reg [`WORD_SIZE-1:0] address;
    wire [`WORD_SIZE-1:0] data;
    wire [`WORD_SIZE-1:0] immdata;
    reg [`WORD_SIZE-1:0] instruction;
    reg [`WORD_SIZE-1:0] pc;
    reg [`WORD_SIZE-1:0] regFile[`NUM_REGS-1:0];
    reg readM;
    reg writeM;
    reg readData;
    reg is_halted;
    wire reset_n;
    wire [`WORD_SIZE-1:0] Imm_next_pc;
    wire [`WORD_SIZE-1:0] Next_pc;
    wire Jump;
    wire JALorJALR;//0 if JAL, 1 if JALR
   wire Branch;
    wire Branch_cond;
    wire MemRead;
   wire MemtoReg;
   wire MemWrite;
   wire PCtoReg;
   wire ALUSrc;
   wire RegWrite;
    wire Halt;
    wire OpenPort;
    wire [`WORD_SIZE-1:0] ALUResult;
    wire [`WORD_SIZE-1:0] Immediate;
    wire [3:0] ALUOp;
    wire [1:0] rs1;
    wire [1:0] rs2;
    wire [1:0] rd;
    wire [2:0] State;
    //tristate of data
    wire data_write;
    wire [`WORD_SIZE-1:0] data_out;
    assign Imm_next_pc = pc + 16'b1;
    control cont
(.instruction(instruction), .clk(clk), .reset_n(reset_n), .Jump(Jump), .OpenPo
rt(OpenPort), .JALorJALR(JALorJALR), .Branch(Branch), .MemRead(MemRead), .MemW
rite(MemWrite), .MemtoReg(MemtoReg), .PCtoReg(PCtoReg), .ALUSrc(ALUSrc), .RegW
rite(RegWrite), .Halt(Halt), .ALUOp(ALUOp), .rs1(rs1), .rs2(rs2), .rd(rd), .St
ate(State));
```

```
    Immediate_Generator imm
(.instruction(instruction), .Immediate(Immediate));
    ALU alu (.A(((Jump && !JALorJALR) || Branch)? (Imm_next_pc):regFile[rs1]),
     .B(ALUSrc? Immediate:regFile[rs2]), .FuncCode(ALUOp), .C(ALUResult));
    condition cond
(.a(regFile[rs1]), .b(regFile[rs2]), .condcode(instruction[13:12]), .result(Br
anch_cond));

    // if it doesn't jump or branch to the other addresses, pc just increments
    // else we should use ALUResult which has the calculated address.
    assign Next_pc = (Jump || (Branch && Branch_cond))?
ALUResult:(Imm_next_pc);
    // if Load or Store, rt(=rs2) should be taken, else ALUResult is used.
    assign data_out = (MemRead || MemWrite)?  regFile[rs2] : ALUResult;
    // "data" is "inout" type, so when readData is 1, "data" plays as input
else output.
    assign data = (readData)?  16'bz : data_out;

    initial begin
        pc <= 16'b0;
        readM <= 1'b1;
        writeM <= 1'b0;
        address <= 16'b0;
        num_inst <= 16'b0;
        regFile[0] <= 16'b0;
        regFile[1] <= 16'b0;
        regFile[2] <= 16'b0;
        regFile[3] <= 16'b0;
        readData <= 1'b1;
    end

    always @(State) begin
        if(!reset_n) begin
            pc <= 16'b0;
            readM <= 1'b1;
            writeM <= 1'b0;
            address <= 16'b0;
            num_inst <= 16'b0;
            regFile[0] <= 16'b0;
            regFile[1] <= 16'b0;
            regFile[2] <= 16'b0;
            regFile[3] <= 16'b0;
            readData <= 1'b1;
        end
        else begin
            case(State)// In IF, all of works in memory
            `ID: begin
```

```
                instruction <= data; // Decode instruction to generate control
signals
                readM <= 1'b0;
                writeM <= 1'b0;
            end
            `EX: begin // Using control signals, handle each case properly
                is_halted <= Halt;
                if(OpenPort) begin // wwd
                    output_port <= regFile[rs1];
                end
                if(MemRead || MemWrite) begin // load or store
                    if(MemRead) begin // load
                        readData <= 1'b1;
                    end
                    else begin // store
                        readData <= 1'b0;
                    end
                    address <= ALUResult; // set the target address
                    readM <= MemRead; // set signal based on MemRead and
MemWrite
                    writeM <= MemWrite;
                end
                else if(RegWrite) begin // instruction which does not need
memory
                    readM <= MemRead; // set signal based on MemRead and
MemWrite
                    writeM <= MemWrite;
                end
                else begin // when it's "branch" then EX->IF
                // for next instruction
                    readData <= 1'b1;
                    num_inst <= num_inst+1;
                    pc <= Next_pc;
                    address <= Next_pc;
                    readM <= 1'b1;
                    writeM <= 1'b0;
                end
            end
            `MEM: begin
                if(!RegWrite) begin // when it's "store" then MEM-> IF
                // for next instruction
                    readData <= 1'b1;
                    num_inst <= num_inst+1;
                    pc <= Next_pc;
                    address <= Next_pc;
                    readM <= 1'b1;
                    writeM <= 1'b0;
                end
```

```
            end
            `WB: begin // from EX or MEM state
                if(MemRead) begin // Load: From MEM state
                    regFile[rd] <= data;
                end
                else begin // From EX state
                // JRL($2 <-- $pc)
                // JAL($2 <-- $pc $pc <-- $pc15..12 ## target11..0)
                // OR ALU Result
                    regFile[rd] <= (PCtoReg)? (Imm_next_pc) : data_out;
                end
                // for next instruction
                readData <= 1'b1;
                num_inst <= num_inst+1;
                pc <= Next_pc;
                address <= Next_pc;
                readM <= 1'b1;
                writeM <= 1'b0;

            end
            endcase
        end
    end
endmodule

module condition (a,b,condcode,result);
    output reg result;
    input wire [15:0] a;
    input wire [15:0] b;
    input wire [1:0] condcode;
    always @(*) begin
        case (condcode)
            2'b00: result <= (a != b); //BNE
            2'b01: result <= (a == b); // BEQ
            2'b10: result <= (a != 16'b0 && a[15] == 0);  // BGZ (a>0)
            2'b11: result <= (a[15] == 1); // BLZ(a<0)
        endcase
    end
endmodule

module Immediate_Generator (instruction,Immediate);
    output reg [`WORD_SIZE-1:0] Immediate;
    input wire [`WORD_SIZE-1:0] instruction;
    always @(*) begin
        case (instruction[15:12]) // opcode
            // { (offset7)8 ## offset7..0 } or { (imm7)8 ## imm7..0 }
            `BNE_OP, `BEQ_OP, `BGZ_OP, `BLZ_OP, `ADI_OP, `LWD_OP, `SWD_OP:
                Immediate <= {{8{instruction[7]}},instruction[7:0]};
```

```
            // ORI $rt <-- $rs | ( 08 ## imm7..0 )
            `ORI_OP: Immediate <= {8'b0,instruction[7:0]};
            // LHI $rt <-- imm7..0 ## 08
            `LHI_OP: Immediate <= {instruction[7:0],8'b0};
            // JMP JAL $pc15..12 ## target11..0
            `JMP_OP, `JAL_OP: Immediate <= instruction;
            default: Immediate <= 16'b0;
        endcase
    end
endmodule
```

4. Discussion

We found the mistake in the module "condition" of cpu.v which checks whether the branch condition holds or not. We did not consider the sign of the number before, so our cpu was not working well. After figuring it out, we modified that part.

5. Conclusion

We implemented the multi cycle cpu which supports TSC instruction set. We put our datapath in cpu.v and control unit in control.v. Through this lab, we could sufficiently learn about how to make several states and how to make them relate each other. Also, we understand what multi cycle cpu is and how it is more efficient than single cycle CPU.