20180188 서승연 20180373 김현지

Lab4. Multi-Cycle CPU

Introduction

이번 과제의 목표는 Lab3에서의 Single Cycle CPU와 비교하여 개선된 이유 및 방법을 분석하고, 이를 통해 Multi Cycle CPU의 동작원리와 과정을 이해 및 구현하는 것이다.

1. What to learn

a. Multi Cycle vs Single Cycle

Single Cycle CPU는 한 Clock당 하나의 Instruction 밖에 수행하지 못한다. 따라서, Clock 시간이 가장 오래 걸리는 Load Instruction의 시간에 맞춰져야한다. 하지만 이는 효율적이지 않다. Load Instruction이 아닌 다른 Instruction들은 수행하는 시간보다 clock 시간이 길어 그 시간은 낭비가 되기 때문이다.

Multi Cycle CPU는 이를 개선하기 위해, 한 Instruction의 각 State별로 쪼개어 한 Clock당 하나의 State가 수행되도록 한다. 따라서, Clock의 길이는 가장 오래 걸리는 State의 시간에 맞춰야 한다. 이렇게 될 경우, Clock Frequency가 Single Cycle보다 높아지게 되고, Load Instruction이 아닌 다른 Instruction은 각 Instruction이 수행해야하는 State에 대해서만 Clock이 사용되기 때문에, 시간을 절약할 수 있다.

b. What to Design & Implement

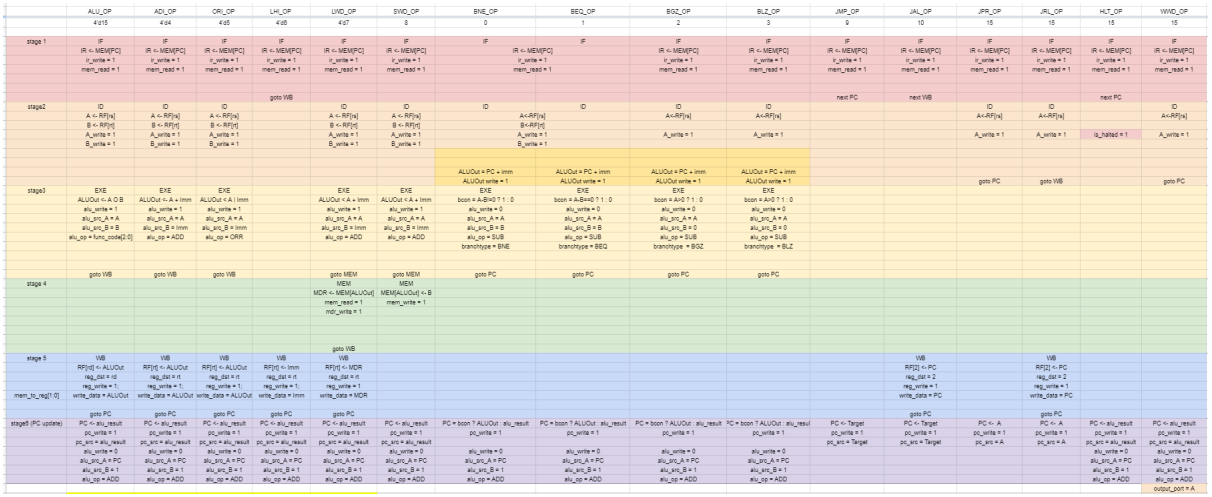
각 Instruction에 대해 어떤 Data Path를 통해 이루어져야하는지를 먼저 파악해야한다. 즉, 어떤 State를 통해 어떻게 Instruction을 수행할 지 디자인해야한다.

Multi Cycle에서는 한 Instruction에서 cycle마다 한 번의 Data Path를 사용하게 되면서 Resource를 Reuse를 할 수 있다. 따라서 Single Cycle에서의 PC Adder를 대신하여 ALU를 사용하거나, Memory도 Instruction Memory와, Data Memory로 구분하지 않고, Timing Difference를 주어 하나의 Single Memory에서 사용하도록 디자인 및 구현해야한다.

그러기 위해서는 각 Resource에서 나온 결과값을 저장해두는 Register (Instruction Register(IR), Memory Data Register(MDR), A, B, ALUOut, PC)가 필요할 것이고, 이 Register에 write하는 여부를 결정하는 signal도 필요할 것이다. 디자인 및 구현시 주의하도록 한다.

Design

1. Instruction Control Flow



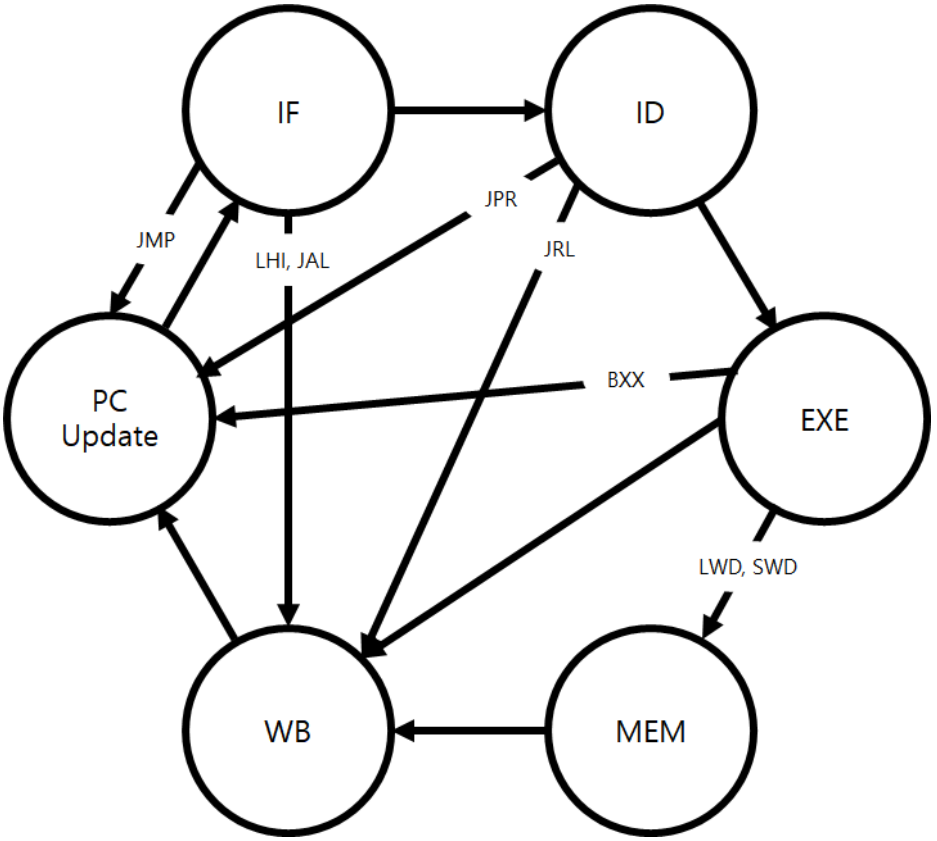
위의 사진은 각 Instruction에 대해 어떤 Flow를 따라 어떻게 계산되는지, Control Unit에서 어떤 Signal이 필요한지 Excel에서 Design하였다.

	IF	ID	EXE	MEM	WB	PCUpdate
ALU	IR<-MEM[PC]	A <- RF[rs] B<-RF[rt]	ALUOut <- A O B		RF[rd] <- ALUOut	PC <- alu_result (PC+1)
ADI, ORI		A <- RF[rs] B<-RF[rt]	ALUOut <- A O Imm		RF[rt] <- ALUOut	PC <- alu_result (PC+1)
LHI					RF[rt] <- Imm	PC <- alu_result (PC+1)
LWD		A <- RF[rs] B<-RF[rt]	ALUOut <- A + Imm	MDR <- MEM[ALUOut]	RF[rt] <- MDR	PC <- alu_result (PC+1)
SWD		A <- RF[rs] B<-RF[rt]	ALUOut <- A + Imm	MEM[ALUOut] <- B		PC <- alu_result (PC+1)
BXX		A <- RF[rs] B<-RF[rt]	bcon? 1 : 0			PC <- bcond ? ALUOut:alu_result
JMP						PC <- Target
JAL					RF[2] <- PC	PC <- Target
JPR		A <- RF[rs] B<-RF[rt]				PC <- A
JRL		A <- RF[rs] B<-RF[rt]			RF[2] <- PC	PC <- A
HLT						PC <- alu_result (PC+1)
WWD		A <- RF[rs] B<-RF[rt]	Output_port <- A			PC <- alu_result (PC+1)

Excel을 간단히 요약하면 위와 같다.

이 작업을 통해 각 Instruction이 어떤 Flow따라 State를 거쳐 수행되는지 파악할 수있었고, control unit에서 어떤 Signal이 필요한 지, 어떤 State로 넘어가야하는지에 대해 쉽게 디자인할 수 있었다. 그 결과는 아래의 Submodule의 control unit part에서 설명할 것 이다.

2. Sequential Control (FSM)

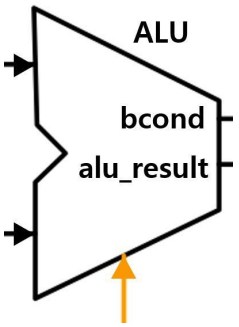


Instruction별 state 이동을 요약한 FSM이다. 이 Finite State Machine은 Mealy Machine이며, State는 총 6개로 구분되어있다. Instruction 종류가 적혀있지 않은 State이동은 표기된 Instruction 외의 모든 Instruction 종류에 대해 Default한 State 이동을 뜻한다. 매 clock 마다 state 이동이 일어난다.

3. Submodules

Submodule은 ALU, ALU Control Unit, ImmGen, Register file, Control Unit의 5개의 submodule과 6개의 Mux들로 구성하였다.,

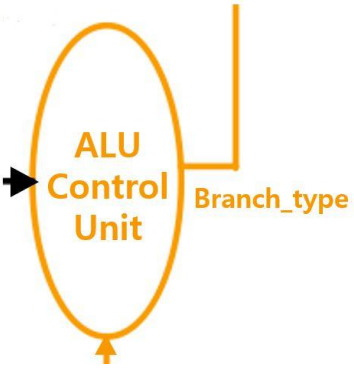
a. ALU



Input : alu_input_1, alu_input_2 (alu operand)
Signal : alu_funccode (수행할 operation을 결정)
branchType (Branch 계산종류를 결정)
Output : alu_result (operation 결과)
bcond (branch 여부)

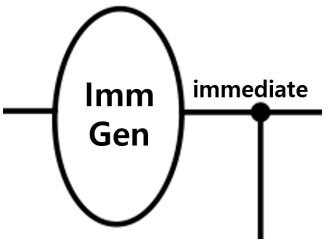
ALU의 경우 Lab3에서 제작하였던 방식을 사용하되, branch의 결과에 따라 bcond 결과를 같이 output하도록 제작하였다.

b. ALU Control Unit



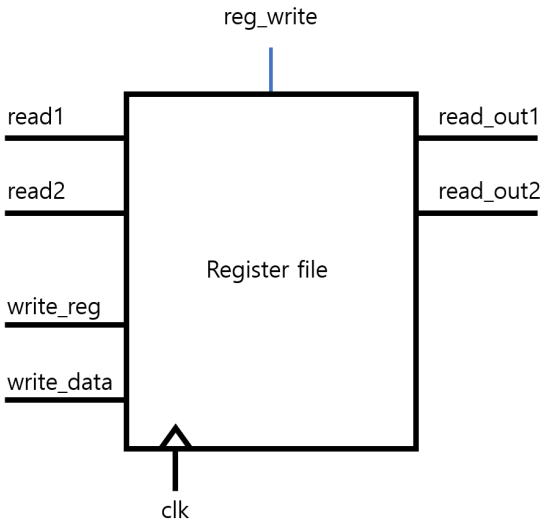
Input : opcode (branch type을 결정)
Signal : alu_op (alu operation을 결정)
Output Signal : branch_type, alu_funccode (alu operation type 결정)

c. ImmGen (immediate generator)



Input : InstReg (Instruction)
Output : immediate (instruction에 따라 immediate를 generate한 결과)

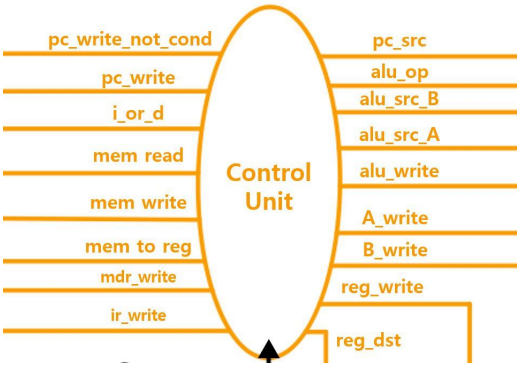
d. Register File



Input : read1, read2 (read할 register)
write_reg (write할 destination register), write_data (write할 data)
Signal : reg_write (write여부를 결정), clk
Output : read_out1, read_out2 (read한 결과)인 로 구성되어 있다.

Register File의 경우 Lab3에서 제작하였던 방식을 사용하되, write reg와 write data값에 각각의 mux output이 들어가게 디자인 하였다.

e. Control Unit



Control Unit에서는 각 instruction에 따라 필요한 control signal이 출력되도록 디자인하였다. Instruction Control Flow Excel 작업을 하며, instruction별로 각 Stage에서 필요한 Control signal을 알게 되었으며, 이는 아래와 같다.

i. Mux Control Signal

먼저 PC, ALU Input1,2, Memory, Register의 Write data, Write Register는 한 input port에 instruction에 따라 다른 값이 들어간다. 이를 제어하기 위한 Mux가 총 6개 필요하고, 이 MUX 제어하는 signal 또한 필요하다.

- PC - pc_src
- ALU Input 1 - alu_src_A
- ALU Input 2 - alu_src_B
- Memory - i_or_d
- Write Data - mem_to_reg
- Write Reg - reg_dst

ii. Register Write Control Signal

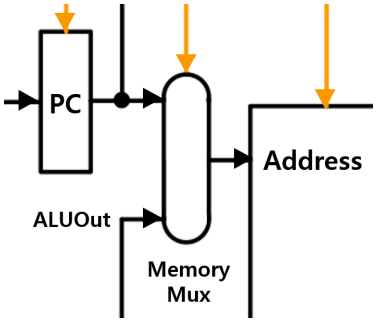
Multicycle에서는 PC, InstReg, MemReg, A, B, ALUOut라는 추가 Register가 생겼고, 이에 Write할 것인지 아닌지 여부를 결정하는 write signal이 필요하다.

- PC - pc_write, pc_write_not_cond
- InstReg - ir_write
- MemReg - mdr_write
- A - A_write
- B - B_write
- ALUOut - alu_write

그 이외의 Signal은 대부분 Single Cycle CPU에서 사용한 Signal으로 자세한 설명은 생략하겠다.

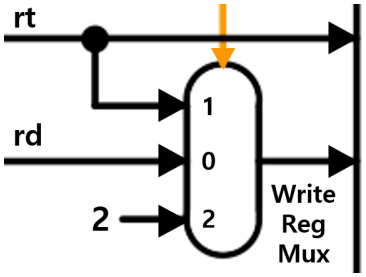
f. Mux

i. Memory Mux



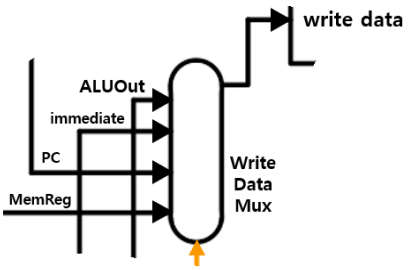
Mux Signal : i_or_d - 0 : PC
- 1 : ALUOut

ii. Write Reg Mux



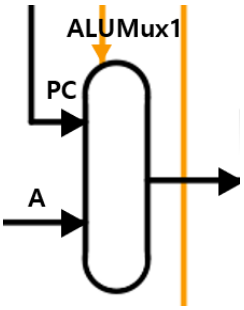
Mux Signal : reg_dst - 0 : rd
- 1 : rt
- 2 : 2

iii. Write Data Mux



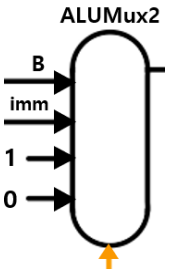
Mux Signal : mem_to_reg - 0 : ALUOut
- 1 : immediate
- 2 : MemReg
- 3 : PC

iv. ALUMux1



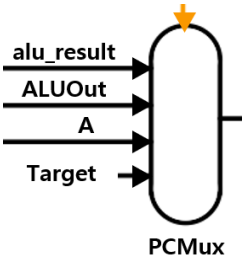
Mux Signal : reg_dst - 0 : A
- 1 : PC

v. ALUMux2



Mux Signal : alu_src_B - 0 : B
- 1 : immediate
- 2 : 1
- 3 : 0

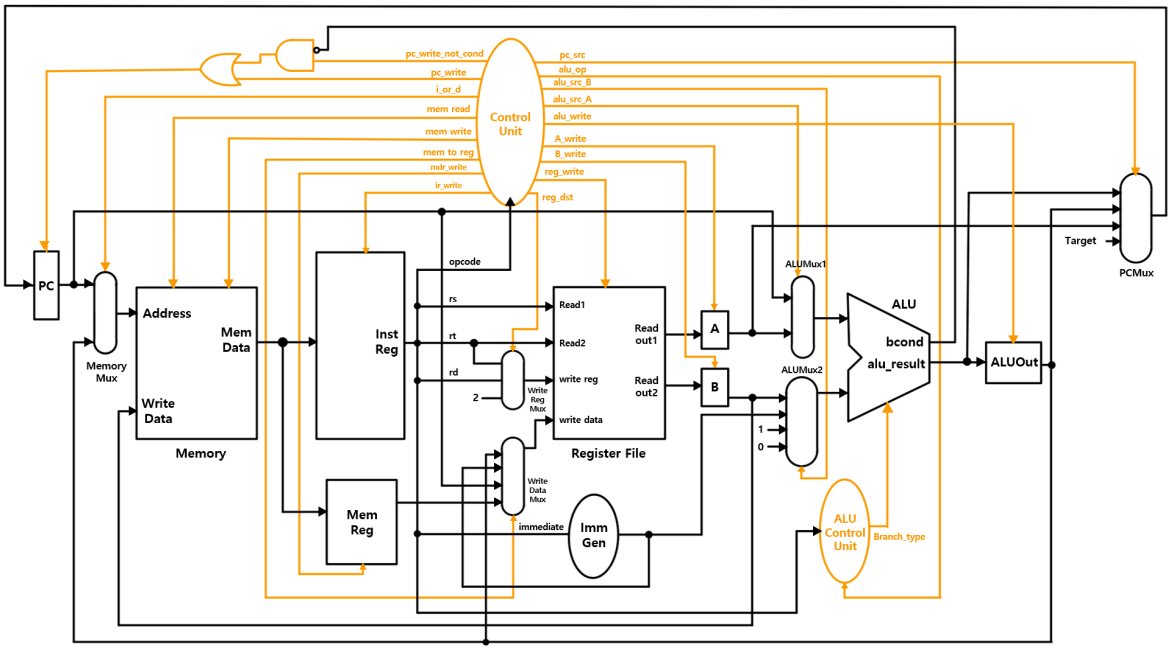
vi. PCMux



Mux Signal : pc_src - 0 : alu_result
- 1 : ALUOut
- 2 : A
- 3 : Target

4. Top-level module (CPU)

Top level module인 cpu의 DataPath를 아래와 같이 디자인하였다.
(위의 각 module 및 mux를 상황에 맞게 연결한 것이다.)



Submodule과 MUX의 Input과 Output을 연결하고, Instruction Decode하는 과정이 top level에서 진행된다. Control unit의 몇몇 signal들을 통해 수행할 기능을 제어하고, 자세한 구현은 아래의 implementation에서 설명하도록 하겠다.

Implementation

1. Control Unit

control unit에서는 IF, ID, EXE, MEM, WB, PCUpdate로 state를 나누고 posedge clk마다 state를 update한다. 각 state에 맞게 control value를 설정한다.

```
always @(posedge clk) begin
  if (!reset_n) state <= 0;
  else state <= next_state;
end
```

위의 always구문은 state update를 담당하며 clk dependent하다. state가 바뀌면 아래의 always 구문에 의해 control value와 next state가 설정된다.

```
always @(*) begin
  case (state)
    `IF : begin
      if(opcode == `HLT_OP && funccode == `INST_FUNC_HLT) is_halted = 1;
      else                                                is_halted = 0;

      pc_write = 0; pc_write_not_cond = 0;
      ir_write = 1; mdr_write = 0; A_write = 0; B_write = 0; alu_write = 0;
      mem_read = 1; mem_write = 0;
      reg_write = 0;
      i_or_d = 0;

      next_state = `ID;
    end
```

IF에서는 memory에서 instruction을 읽어오기 위해 mem_write를 1로 만들어준다. 읽어온 명령어를 instruction register에 저장하기 위해 ir_write를 1로 설정한다. 나머지 signal들을 초기화 시켜준다.

halt instruction을 제외한 나머지 명령어는 ID stage로 이동한다. halt의 경우는 signal을 주어 machine을 중단시킨다.

```
`ID : begin
  is_halted = 0;
  pc_write = 0; pc_write_not_cond = 0;
  ir_write = 0; mdr_write = 0; A_write = 1; B_write = 1; alu_write = 0;
  mem_read = 0; mem_write = 0;
  reg_write = 0;
  i_or_d = 0;

  case (opcode)
    `BNE_OP, `BEQ_OP, `BGZ_OP, `BLZ_OP: begin   //ALUOut = PC + Imm + 1
      alu_src_A = `ALUSrcA_PC;
      alu_src_B = `ALUSrcB_IMM;
      alu_op = `FUNC_ADD;
      alu_write = 1;
    end
  endcase

  // set next state
  case (opcode)
    `LHI_OP : next_state = `WB;
    `JMP_OP : next_state = `PCUpdate;
    `JAL_OP : next_state = `WB;
    `JRL_OP, `JPR_OP, `WWD_OP: begin
      case (funccode)
        `INST_FUNC_JPR : next_state = `PCUpdate;
        `INST_FUNC_JRL : next_state = `WB;
        `INST_FUNC_WWD : next_state = `PCUpdate;
        default : next_state = `EXE;
      endcase
    end
    default : next_state = `EXE;
  endcase
end
```

ID에서는 RF에서 읽은 값을 register에 저장하기 위해서 A_write, B_write를 1로 설정한다. opcode에 따라 next state를 정해준다. branch instruction의 경우는 target address를 ID에서 미리 계산하여 alu register에 저장한다.

```
`EXE : begin
  is_halted = 0;
  pc_write = 0; pc_write_not_cond = 0;
  ir_write = 0; mdr_write = 0; A_write = 0; B_write = 0; alu_write = 0;
  mem_read = 0; mem_write = 0;
  reg_write = 0;
  i_or_d = 0;

  alu_src_A = `ALUSrcA_A;

  case (opcode)
    `BNE_OP, `BEQ_OP, `BGZ_OP, `BLZ_OP: begin
      alu_src_B = (opcode == `BNE_OP || opcode == `BEQ_OP) ? `ALUSrcB_B : `ALUSrcB_0;
      alu_op = `FUNC_SUB;
      alu_write = 0;
    end
    `ALU_OP: begin              //ALUOut = A o B
      alu_src_B = `ALUSrcB_B;
      alu_op = funccode[2:0];
      alu_write = 1;
    end
    default: begin              //ALUOut = A o Imm
      alu_src_B = `ALUSrcB_IMM;
      alu_op = (opcode == `ORI_OP) ? `FUNC_ORR : `FUNC_ADD;
      alu_write = 1;
    end
  endcase

  // set next state
  case (opcode)
    `BNE_OP, `BEQ_OP, `BGZ_OP, `BLZ_OP: next_state = `PCUpdate;
    `LWD_OP, `SWD_OP: next_state = `MEM;
    default: begin
      if(funccode == `INST_FUNC_WWD) next_state = `PCUpdate;
      else next_state = `WB;
    end
  endcase
end
```

EXE 단계에서는 ALU를 이용하여 각 명령어에서 요구하는 연산을 수행한다. 따라서 명령어에 맞는 ALU src와 op을 설정해준다. 필요한 경우 alu_write를 1로 설정해 연산결과를 저장한다.

```
`MEM : begin
  is_halted = 0;
  pc_write = 0; pc_write_not_cond = 0;
  ir_write = 0; mdr_write = 0; A_write = 0; B_write = 0; alu_write = 0;
  mem_read = 0; mem_write = 0;
  reg_write = 0;
  i_or_d = 1;

  case (opcode)
    `LWD_OP : begin
      mem_read = 1;
      mdr_write = 1;
    end
    `SWD_OP : begin
      mem_write = 1;
    end
  endcase

  // set next state
  case (opcode)
    `LWD_OP : next_state = `WB;
    `SWD_OP : next_state = `PCUpdate;
  endcase
end
```

MEM stage를 거치는 명령어는 LWD와 SWD가 있다. LWD의 경우 mem_read와 mdr_write를 1로 설정해 memory 값을 읽고 register에 저장한다. SWD의 경우 mem_write를 1로 설정하여 memory write를 수행한다.

```
`WB : begin
  is_halted = 0;
  pc_write = 0; pc_write_not_cond = 0;
  ir_write = 0; mdr_write = 0; A_write = 0; B_write = 0; alu_write = 0;
  mem_read = 0; mem_write = 0;
  reg_write = 1;
  i_or_d = 0;

  case (opcode)
    `ADI_OP, `ORI_OP: begin reg_dst = `RegDst_RT; mem_to_reg = `MemToReg_ALUOut; end
    `LHI_OP :         begin reg_dst = `RegDst_RT; mem_to_reg = `MemToReg_IMM; end
    `JAL_OP :         begin reg_dst = `RegDst_2;  mem_to_reg = `MemToReg_PC; end
    `LWD_OP :         begin reg_dst = `RegDst_RT; mem_to_reg = `MemToReg_MDR; end
    default: begin
      case(funccode)
        `INST_FUNC_JRL : begin reg_dst = `RegDst_2;  mem_to_reg = `MemToReg_PC; end
        default :        begin reg_dst = `RegDst_RD; mem_to_reg = `MemToReg_ALUOut; end
      endcase
    end
  endcase

  // set next state
  next_state = `PCUpdate;
end
```

WB은 계산 결과나 load한 정보를 register file에 쓰는 stage이다. reg_write를 1로 설정해 register write를 허용한다. 각 명령어에 따라 쓸 레지스터 번호와 data를 설정한다.

```
case (opcode)
  `JMP_OP, `JAL_OP : pc_src = `PCSrc_Target;
  `BNE_OP, `BEQ_OP, `BGZ_OP, `BLZ_OP: begin

    alu_write = 0;
    alu_src_A = `ALUSrcA_PC;
    alu_src_B = `ALUSrcB_1;
    alu_op = `FUNC_ADD;
    if(bcond) begin
      pc_write_not_cond = 0;
      pc_src = `PCSrc_ALUOut;
    end else begin
      pc_write_not_cond = 1;
      pc_src = `PCSrc_ALURes;
    end

  end
  `ADI_OP, `ORI_OP, `LHI_OP, `LWD_OP, `SWD_OP : begin
    alu_write = 0;
    alu_src_A = `ALUSrcA_PC;
    alu_src_B = `ALUSrcB_1;
    alu_op = `FUNC_ADD;
    pc_src = `PCSrc_ALURes;
  end
  `ALU_OP : begin
    case (funccode)
      `INST_FUNC_JPR, `INST_FUNC_JRL: pc_src = `PCSrc_A;
      `INST_FUNC_WWD : begin
        alu_write = 0;
        alu_src_A = `ALUSrcA_PC;
        alu_src_B = `ALUSrcB_1;
        alu_op = `FUNC_ADD;
        pc_src = `PCSrc_ALURes;
      end
      default: begin
        alu_write = 0;
        alu_src_A = `ALUSrcA_PC;
        alu_src_B = `ALUSrcB_1;
        alu_op = `FUNC_ADD;
        pc_src = `PCSrc_ALURes;
      end
    endcase
  end
endcase

// set next state
next_state = `IF;
```

PCUpdate에서는 PC_write를 1로 설정하여 PC에 next PC 값이 써지도록 한다. jump의 경우 target을 사용하고 branch의 경우에는 bcond에 따라 PC+1과 target address 중 선택한다. 그 외의 경우에서는 모두 PC+1로 설정한다.

2. ALU Control Unit

```
always @(*) begin
  case(opcode)
    `BNE_OP : branch_type = `BNE;
    `BEQ_OP : branch_type = `BEQ;
    `BGZ_OP : branch_type = `BGZ;
    `BLZ_OP : branch_type = `BLZ;
  endcase
  alu_funccode = alu_op;
end
```

branch instruction의 type을 구분할 수 있는 변수인 branch_type을 초기화하고 alu_funccode를 설정한다.

설정한 정보들은 ALU unit에서 수행하는 연산과 bcond 설정에 영향을 준다.

3. Submodules

a. ALU

```
always @(*) begin
    case(alu_func_code)
        `FUNC_ADD : alu_result = alu_input_1 + alu_input_2;
        `FUNC_SUB : begin
            alu_result = alu_input_1 - alu_input_2;
            case (branch_type)
                `BNE : bcond = alu_result != 0 ? 1 : 0;
                `BEQ : bcond = alu_result == 0 ? 1 : 0;
                `BGZ : bcond = alu_result > 0 ? 1 : 0;
                `BLZ : bcond = alu_result < 0 ? 1 : 0;
                default : bcond = 0;
            endcase
        end
        `FUNC_AND : alu_result = alu_input_1 & alu_input_2;
        `FUNC_ORR : alu_result = alu_input_1 | alu_input_2;
        `FUNC_NOT : alu_result = ~ alu_input_1;
        `FUNC_TCP : alu_result = ~ alu_input_1 + 1;
        `FUNC_SHL : alu_result = alu_input_1 << 1;
        `FUNC_SHR : alu_result = alu_input_1 >>> 1;
    endcase
end
```

이전 Lab에서 구현한 ALU module과 흡사한 구조를 가진다. branch type에 따라 bcond을 설정하는 부분이 추가되었다. 설정된 bcond는 Control unit에서 next PC 값을 정할 때 사용된다.

b. Register File

4개의 register를 가지고 있으며 이전 과제와 동일하므로 코드는 생략한다.

c. Imm Gen

opcode에 따라 immediate extension을 수행하는 모듈이다. 이전 과제의 cpu.v에서 immediate value를 생성하는 원리와 동일하므로 코드는 생략한다.

4. CPU

```
mux4_1 #(2) WriteRegMux(reg_dst, rd, rt, 2'd2, 2'd0, WriteRegMuxOut);
mux4_1 #(`WORD_SIZE) WriteDataMux(mem_to_reg, ALUOut, immediate, MemReg, PC + `WORD_SIZE'd1, WriteDataMuxOut);
mux2_1 #(`WORD_SIZE) MemoryMux(i_or_d, PC, ALUOut, MemoryMuxOut);
mux2_1 #(`WORD_SIZE) ALUMux1(alu_src_A, A, PC, ALUMux1Out);
mux4_1 #(`WORD_SIZE) ALUMux2(alu_src_B, B, immediate, `WORD_SIZE'd1, `WORD_SIZE'd0, ALUMux2Out);
mux4_1 #(`WORD_SIZE) PCMux(pc_src, alu_result, ALUOut, A, { PC[15:12], target }, PCMuxOut);

imm_gen ImmGen(InstReg, immediate);
alu ALU(ALUMux1Out, ALUMux2Out, alu_funccode, branch_type, alu_result, overflow_flag, bcond);
alu_control_unit ALUControlUnit(clk, funccode, opcode, alu_op, alu_funccode, branch_type);
control_unit ControlUnit (opcode, funccode, bcond, clk, reset_n, is_halted, pc_write_not_cond, pc_write, i_or_d, mem_read, mem_write,
                          mem_to_reg, ir_write, mdr_write, pc_src, alu_op, alu_src_A, alu_src_B, reg_dst, reg_write, A_write, B_write, alu_write);
register_file RegisterFile(read_out1, read_out2, read1, read2, WriteRegMuxOut, WriteDataMuxOut, reg_write, clk);
```

위에서 서술한 module들을 모두 선언하여 만들어준다. 모든 동작은 Control unit의 signal들에 의해 제어된다.

```
always @(posedge clk) begin
    if(!reset_n) begin
        PC <= 0;
        num_inst <= 0;
        InstReg <= 0;
        MemReg <= 0;
        A <= 0;
        B <= 0;
        ALUOut <= 0;
    end
    else begin
        if (A_write) begin A <= read_out1; end
        if (B_write) begin B <= read_out2; end
        if (alu_write) begin ALUOut <= alu_result; end
        if (ir_write) InstReg <= MemData;
        if (mdr_write) MemReg <= MemData;

        if (pc_write || pc_write_not_cond && !bcond) begin  //pc_write
            num_inst <= num_inst + 1;   //new instruction
            PC <= PC_NXT;
        end
    end
end
```

A, B, ALUOut, instruction register, memory register, PC는 모두 write signal이 있을 경우, clk dependent하게 값을 저장한다.

```
always @(*) begin

    opcode = InstReg[15:12];
    rs = InstReg[11:10];
    rt = InstReg[9:8];
    rd = InstReg[7:6];
    funccode = InstReg[5:0];
    target = InstReg[11:0];

    read1 = rs; read2 = rt;

    if(pc_write) begin
        if (opcode == `WWD_OP && funccode == `INST_FUNC_WWD) output_port = A;
    end

    address = MemoryMuxOut;
    MemData = data;

    if (pc_write || pc_write_not_cond && !bcond) PC_NXT = PCMuxOut;

end
```

fetched instruction을 쪼개서 저장한다. (ID) WWD 명령어인 경우 output port의 값을 설정하여 cpu 외부에 알린다.

Discussion

대부분의 Module을 구현하는 것은 SingleCycle을 구현할 때와 거의 유사하여 크게 힘든 부분이 없었다. 하지만 Control Unit을 구현하는 것이 가장 힘들었던것 같다. 처음에 간단히 디자인만 하고 Control unit을 짰더니 세세하게 놓치는 부분이 많아, Excel로 Instruction별 수행해야하는 Flow를 구체적으로 설계하여 Implement해보았다. 처음 간단히 디자인하여 control unit을 구현할 때 state를 IF, ID, EXE, MEM, WB로만 나누고 next state가 IF인 경우에 PC를 계산하여 더해주는 방식을 선택했다. 하지만 Excel로 Flow를 계산해본 결과 해당 방법은 EXE 다음에 IF로 가는 branch instruction의 경우 EXE과정에서 ALU를 2번 사용해야 했다. 이를 해결하기 위해 EXE를 2개로 나누는 방법과 PCUpdate state를 추가하는 방법 중에 좀 더 직관적인 두 번째 방법을 선택했다. 변경된 설계는 branch instruction이 요구하는 1) target address 2) PC+1 3) bcond 계산을 모두 다른 state에서 수행하여 하나의 state에서 동일 resource(ALU)를 두 번 이상 사용하게 되는 상황을 방지하였다.

이외에도 PC Update state를 사용하지 않고 PC+1 연산을 IF에서 수행하는 방법이 있다. 이 방법은 instruction 당 clock 수를 1만큼 줄여주기 때문에 좀 더 효율적이다. 하지만 시간부족으로 구현하지 못해 아쉬웠다.

Conclusion

multi-cycle CPU의 동작 방식을 이해하고 구현하였다.

FSM의 개념과 각 state의 역할, control value들의 역할을 알고 control unit을 구현하였으며, 이외의 다른 Submodule도 역할에 맞게 구현하였다. 또한, 이를 이용하여 cpu 내부의 모든 연산을 제어하였다.

아래는 Lab4에서 TestBench에 주어진 경우를 모두 통과한 모습이다.

```
VSIM 2> run -all
# Clock # 4510
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/intelFPGA_pro/20.4/lab4/cpu_TB.v(151)
#    Time: 451150 ns  Iteration: 2  Instance: /cpu_TB
```