20180188 서승연 20180373 김현지

Lab3. Single-Cycle CPU

Introduction

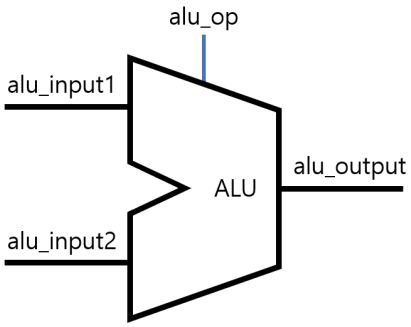
이번 과제의 목표는 Single Cycle CPU의 동작원리와 과정을 이해하고 구현하는 것이다. CPU 구성요소인 register file과 ALU를 직접 구현하여 연결하며 cpu 동작을 관리하는 control unit 또한 직접 구현해본다. memory에서 instruction을 읽고 decoding하여 실행하고 memory와 register에 write하는 전 과정을 구현한다.

Design

1. Submodules

Submodule은 ALU, Register file, Control unit의 세 module로 구성하였다.

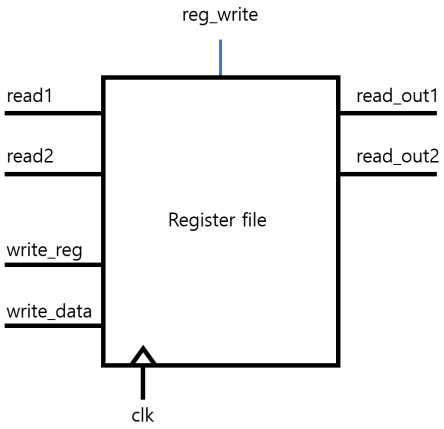
a. ALU



ALU의 Input은 alu operand input인 2개의 alu_input_1, alu_input_2와 operator를 결정하는 alu_op로 이루어져 있고, Output은 그 operation 결과인 alu_output이 있다.

ALU의 경우 Lab1에서 제작하였던 방식을 사용하되, 당시에 모듈화 연습을 위해 세분화된 ALU를 제작했기 때문에 이를 하나의 모듈로 다시 디자인하였다.

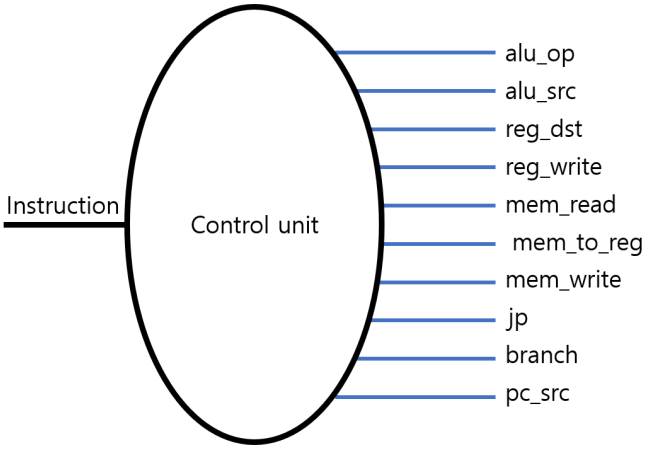
b. Register File



Register File의 Input은 read할 register인 2개의 read1, read2와 write할 register인 write_reg, write할 data인 write_data, write여부를 결정하는 control signal인 reg_write와 clk으로 구성되어 있고, Output은 read한 결과인 read_out1, read_out_2로 구성되어 있다. Register File에는 4개의 Register data를 담고 있으며, input과 output에 따라 그 data를 읽거나 쓰게 된다.

Register File에서 read는 combinational하게 구현하여 항상 출력되도록 하고, write는 reg_write에 의해 control되므로, clk dependent하게 구현하여 control signal에 맞게 write되도록 디자인 하였다.

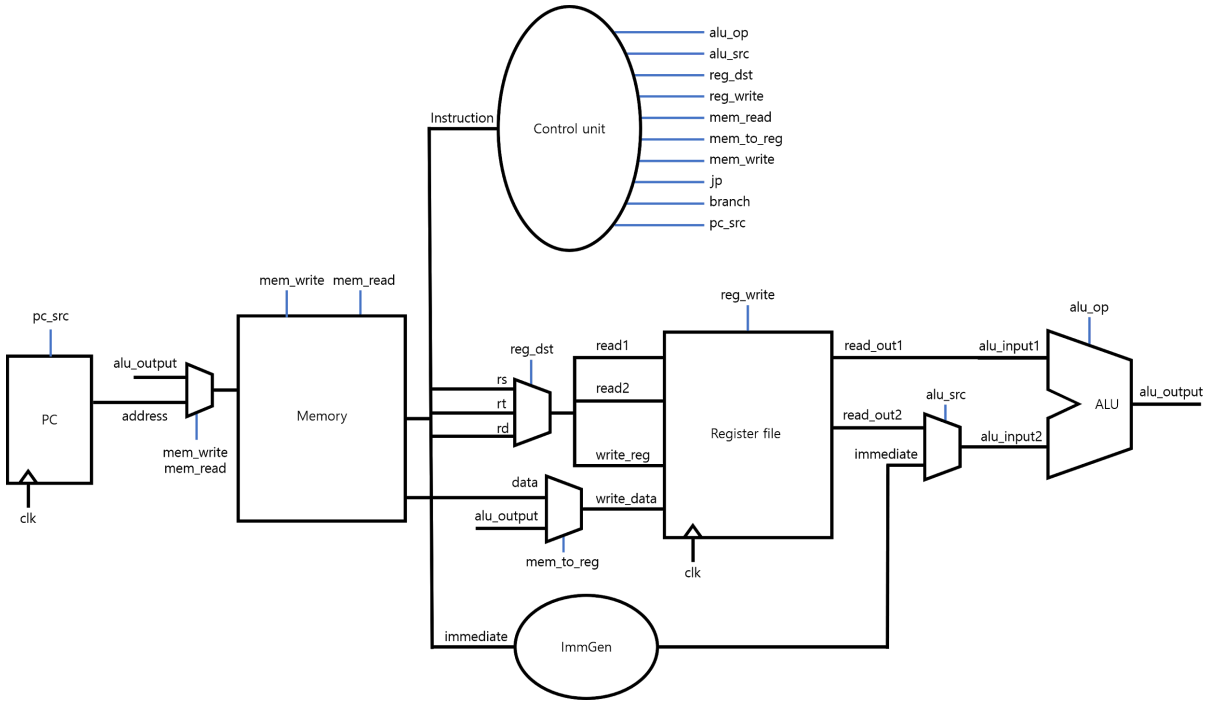
c. Control unit



Control unit에서는 각 instruction에 따라 필요한 control signal이 출력되도록 디자인하였다.

2. Top-level module (CPU)

Top level module인 cpu는 다음과 같이 디자인하였다.



세 Submodule에서 수행하는 기능을 제외한 나머지 기능(PC, ImmGen 등)은 전부 top level에서 진행된다. Control unit의 signal들을 통해 수행할 기능을 제어하고, 자세한 구현은 아래의 implement에서 설명하도록 하겠다.

Implementation

1. Submodules

a. ALU

Lab1에서 구현한 ALU module의 형식대로 구현하였다. 따라서 코드는 생략하였다. 필수적으로 구현해야하는 opcode외에 앞으로의 구현에 있어서 추가적으로 구현된 Function은 다음과 같다.

- FUNC_ID1 : input1을 출력한다.
- FUNC_ID1 : input2를 출력한다.
- FUCN_ZRO : 0을 출력한다.

해당 function에 대해서는 opcode.v에 추가로 define하였고, 이에 따라 funccode의 사이즈가 3에서 4로 증가하였다. 이 function들은 write data값이 alu_output으로 부터 나올때 어떤 연산 결과가 아닌 특정 value들(PC, Imm)로 설정되는 경우 쉽게 구현하거나, Branch Instruction의 결과를 쉽게 계산하기 위해 추가하였다.

b. Register File

```
initial begin
    for (i = 0; i < `NUM_REGS; i = i + 1) register[i] = 0;
end

always @(*) begin
    read_out1 = register[read1];
    read_out2 = register[read2];
end

always @(posedge clk) begin
    if (reg_write) register[write_reg] <= write_data;
end
```

4개의 register를 가지고 있으며 register number를 input으로 받아 read/write한다. Register Write는 clk에 따라 진행되며, reg_write control에 따라 write_reg에 해당 data의 write 여부를 결정한다.

c. Control Unit

Control Unit에서 다룰 control signal은 아래와 같다.

- alu_op: alu에서 수행할 연산을 결정한다.
- alu_src: alu_input2 값을 결정한다.
    - 1이면 immediate를 넣는다 (I-type)
    - 0이면 register read data를 넣는다.
    (추가로, J instruc의 경우 alu_op control에 따라 input에 PC를 넣는다.)
- reg_dst: register file의 write_reg 값을 결정한다.
    - 0이면 write_reg = rt이다.
    - 1이면 write_reg = rd이다.
    (추가로, J instruc의 경우 jp control에 따라 write_reg를 2로 설정한다.)
- reg_write: register file write 여부를 결정한다.
- mem_read: memory read 여부를 결정한다.
- mem_to_reg: memory에서 읽은 값을 register에 쓸 지 여부를 결정한다.
- mem_write: memory write 여부를 결정한다.
- jp: Jump instruction의 경우 1로 설정된다.
- branch: Branch condition을 설정한다. (0: NE, 1: EQ, 2: GZ, 3:)
- pc_src: PC_NXT 값을 설정한다. (0: DEF, 1: IMM, 2: TAR, 3: REG)

Opcode에 따른 control signal의 결과이다.

	alu op	alu src	reg dst	reg write	mem read	mem to_reg	mem write	jp	branch	pc src
ALU	Func	0	1	1	0	0	0	0	x	DEF
ADI	ADD	1	0	1	0	0	0	0	x	DEF
ORI	ORR	1	0	1	0	0	0	0	x	DEF
LHI	ID2	1	0	1	0	0	0	0	x	DEF
LWD	ADD	1	0	1	1	1	0	0	x	DEF
SWD	ADD	1	0	0	0	0	1	0	x	DEF
BEQ	SUB	0	0	0	0	0	0	0	BRANCH_EQ	IMM
BNE	SUB	0	0	0	0	0	0	0	BRANCH_NE	IMM
BGZ	ID1	0	0	0	0	0	0	0	BRANCH_GZ	IMM
BLZ	ID1	0	0	0	0	0	0	0	BRANCH_LZ	IMM
JMP	x	0	0	0	0	0	0	1	x	TAR
JAL	ID2	0	1	1	0	0	0	1	x	TAR
JPR	x	0	0	0	0	0	0	1	x	REG
JRL	ID2	0	1	1	0	0	0	1	x	REG

2. CPU modules

a. Instruction & Memory Data Fetch

```
always @(posedge inputReady) begin
    if(!instrDataFetch) begin
        instr <= data;
        instrDataFetch <= 1;
    end
    if(!memoryDataFetch) begin
        memory <= data;
        memoryDataFetch <= 1;
    end
end
```

```
always @(negedge clk) begin
    memoryDataFetch <= 0;
end

always @(posedge clk) begin
    instrDataFetch <= 0;
```

cpu module의 inout port인 data에 담겨있는 정보를 읽는다. Instruction data fetch, memory data fetch가 한 clk에 모두 진행되어야하므로, flag를 이용해 이를 구별한다.

posedge clk에서는 instruction fetch를 실행하고, negedge clk에서는 memory data fetch가 실행되도록 하였다.

b. Instruction Decode

```
always @(*) begin
    if(instrDataFetch) begin

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

        // immediate extension
        case (opcode)
            `ORI_OP : immediate = { 8'b0, instr[7:0] };
            `LHI_OP : immediate = instr[7:0] << 8;
            default : immediate = { {8{instr[7]}}, instr[7:0] };
        endcase
        target = instr[11:0];
```

Instruction Fetch후, 해당 instruction을 decode한다. bit 수는 TSC manual을 참고하였다. 또한 opcode에 따라 case를 나누어 ImmGen한 값을 immediate에 저장하였다.

```
read1 = rs;
read2 = rt;
if (reg_dst) begin
    if (jp)  write_reg = 2;      // JAL, JRL
    else     write_reg = rd;     // ALU
end else     write_reg = rt;
```

decode한 결과 나온 register number에 대해 이를 control에 따라 register file의 input으로 적절히 넣어준다. reg_dst가 0인 경우 write_reg가 rt가 되며, reg_dst가 1인 경우 일반적으로는 write_reg가 rd가 되어야하지만, JAL과 JRL은 항상 register 2에 pc를 저장하기 때문에, jp control로 구분하여 write_reg값을 결정하였다.

c. ALU / Execution

```
//EXE
alu_input_1 = read_out1;
if(alu_src) alu_input_2 = immediate;                    //ADI, ORI, LHI, LWD, SWD
else begin
    if (alu_op == `FUNC_ID2) alu_input_2 = PC;          //JAL, JRL
    else                     alu_input_2 = read_out2;
end
```

alu_src control에 따라 alu input을 설정한다.

JAL, JRL instruction의 경우 $2 <-- $pc가 실행되는데, write data인 alu_output이 PC가 되어야 하기 때문에 alu_op가 `FUNC_ID2일 경우로 구분하고 alu_input2 값을 PC로 설정하여 alu_output을 잘 설정하였다.

d. Memory Access

```
//MEM
if(mem_read || mem_write) address = alu_output;         //LWD, SWD
```

memory에 접근해야 하는 경우 즉, Load와 Store instruction일 경우, address값을 alu_output의 결과로 설정해준다.

mem_read/mem_write control에 의해 readM/writeM이 1로 설정되고, 이에 따라 cpu module은 address에 해당하는 data를 읽어 오거나 쓰게 된다.

read의 경우 1번 data fetch에서 설명한 것과 마찬가지로 memoryDataFetch flag에 의해 memory register에 저장되어 이후, writa data로 사용된다.

e. Write Back

```
//WB
if(mem_to_reg) begin                                    //LWD
    if(memoryDataFetch) write_data = memory;
end else                write_data = alu_output;
```

memory의 값을 저장하는 것이 아니라면, write_data를 alu_output로 설정하여 이를 register에 쓰도록 한다.

Write되는 것은 register file에서 reg_write와 clk에 의해 그 여부가 결정된다.

f. PC(Program Counter) Update

pc_src에 따라 아래와 같이 PC_NXT를 설정해준다.

```
case(pc_src)
    `PC_DEF : PC_NXT = PC + 1;
    `PC_IMM : begin
        if ((branch == `BRANCH_NE && alu_output != 0) ||
            (branch == `BRANCH_EQ && alu_output == 0) ||
            (branch == `BRANCH_GZ && alu_output > 0)  ||
            (branch == `BRANCH_LZ && alu_output < 0))   PC_NXT = PC + immediate + 1;
        else                                            PC_NXT = PC + 1;
    end
    `PC_TAR : PC_NXT = {PC[15:12] , target};    //JMP, JAL
    `PC_REG : PC_NXT = read_out1;               //JPR, JRL
endcase
```

clk에 따라 설정해둔 PC_NXT로 PC를 Update하였다.

```
always @(posedge clk) begin
    instrDataFetch <= 0;

    if(!reset_n) begin
        PC <= `WORD_SIZE'd0;
        PC_NXT <= `WORD_SIZE'd0;
        address <= `WORD_SIZE'd0;
        instrDataFetch <= 0;
    end
    else begin
        PC <= PC_NXT;
        address <= PC_NXT;
    end
end
```

Discussion

각 submodule을 구현하고 이를 연결하는 과정은 single cycle cpu의 구조도를 보면서 하여, 크게 어렵진 않았다.

처음 헤매던 부분은 readM과 writeM을 그냥 mem_read, mem_write와 같은 값이라고 착각했던 것이다. instruction data를 받기 위해서는 memory에 접근해야하기 때문에 이에 대한 signal이 필요한데, 원래 control unit은 instruction이 fetch 된 이후에 mem_read와 mem_write의 signal을 보낼수 있기때문에 이 사이에서 모순이 생겼다. 이를 해결하기 위해, control unit에 instruction이 fetch 되었는지 확인하는 flag를 input으로 집어넣어 이에 따라 mem_read와 mem_write에 대한 추가적인 구현을 하였다. 그 후 data에 접근하는 것을 확인할 수 있었다.

가장 힘들었던 부분은 data memory를 다루는 것이었다. clk에 따라 Instruction Data Fetch와 Memory data fetch가 한 clk에서 둘 모두 이루어져야한다는 것을 알고 posedge와 negedge로 나눠야한다는 사실은 알았지만, 이를 코드로 표현하기가 어려웠다. 앞에서 말한 문제가 해결된 이후에도 data에서 계속 원하는 값을 받지 못하였다. 이를 해결하기 위해, inputReady가 되었을 때 write data를 memory로 가져와야하는데 이와 관계없이 받았던게 문제였다. 따라서 이에 대한 flag도 같이 구현하여 정확한 data를 읽는 것을 확인할 수 있었다. ackOutput signal을 통해 조금 더 괜찮게 코드를 작성할 수 있었을 것 같았는데 시간관계상 inputReady만을 이용하여 구현한것이 조금 아쉬웠다.

Conclusion

single cycle CPU의 동작 방식을 이해하고 구현하였다.

각 sub module들의 input과 output에 주의하여 각각을 연결하고 구현하고, Top level module에서 clk과 signal들에 따라 각 Stage의 선후관계에 주의하여 구현하였다.

```
#   28 :
# PASSED
#   38 :
# PASSED
#   39 :
# PASSED
#   40 :
# PASSED
#   41 :
# PASSED
# Passed = 15, Failed = 0
# ** Note: $finish    : C:/intelFPGA_pro/20.4/lab3/cpu_evaluation_tb.v(116)
#    Time: 4650 ns  Iteration: 1  Instance: /tb_cpu
```

Lab3에서 TestBench에 주어진 경우를 모두 통과한 모습이다.