컴퓨터조직론 lab04 보고서

2019-12172 이재원

1. Introduction

이 실습에서는 single cylcle cpu를 design하는 것을 목표로 한다. 모든 명령어를 구현하지 않고 일부만 구현하며 반드시 datapath와 control signal을 다른 module에서 구현해야 한다.

2. Design

Cpu module은 memory에서 instruction을 읽는 부분과 control unit, datapath module을 연결하여 구현할 예정이다.

Control unit은 combinational circuit으로 output만 구현하면 되기 때문에 각 output마다 trinary operator로 상황에 따라 적절한 output을 출력할 예정이다.

Datapath는 안에 RF, ALU, PC module을 사용하여 구현할 예정이다. Single cycle 이기 때문에 IR은 구현하지 않고 instruction을 decode한다. RF, ALU는 이전의 lab에서 구현한 것을 그대로 사용할 것이나 ALU에서 LHI operation을 추가할 것이다. 또한, 매 cycle마다 num_inst을 1씩 증가한다.

Datapath는 wire를 이용하여 여러 module을 연결하고 single cycle cpu 설명 그대로 mux를 이용할 것이다. Wire를 이용하므로 trinary operator로 control_unit의 output인 control signal을 조건문으로 사용할 예정이다.

3. Implementation

cpu.v

```
23      // Datapath - control Unit
24      wire [3:0] opcode;
25      wire [5:0] func_code;
26      wire RegDst, Jump, ALUSrc, RegWrite, isWWD;
27      wire [3 : 0] ALUOp;
28
29      wire [15 : 0] instruction_wire;
30      reg [15 : 0] instruction;
```

Figure 1. cpu.v port declaration

opcode, func_code, RegDst, … , ALUOp는 skeleton code의 input을 그대로 사용했으며, control_unit, datapath module input에 사용된다. 추가한 것은 input_ready일 때마다 instruction을 저장하는

register와 그 register에 연결되어 datapath에 전달되는 instruction_wire이다. 이는 Figure 2에 있으며 간단하게 구현했다.

```
69      // read instruction from data
70      always @ (posedge inputReady) begin
71          if (clk) instruction <= data;
72      end
73
74
75      // decode instructions for control_unit
76      assign opcode = instruction[15 : 12];
77      assign func_code = instruction[5 : 0];
78
79      // assigned reg to wire for module(datapath) input
80      assign instruction_wire = instruction;
```

Figure 2. cpu.v code implementaion

control_unit.v

원래 trinary operator로만 구현할 예정이었으나 조건이 복잡해지는 경우가 있어 일부는 trinay operator, 일부는 case statement를 사용했다.

```
17      assign RegDst = (opcode == `typeR) ? 1 : 0;
18      assign Jump = (opcode == `OPCODE_JMP || opcode == `OPCODE_JAL) ? 1 : 0;
19      assign ALUSrc = (opcode == `typeR) ? 0 : 1;
20      assign isWWD = (opcode == `typeR && func_code == `FUNC_WWD) ? 1 : 0;
21
```

Figure 3. control_unit.v trinary operator

R type, WWD, JUMP에 해당겨우에만 control signal 출력하는 것은 figure 3처럼 trinary operator로 구현했다.

```
23      always @ (*) begin
24      case (opcode)
25          `typeR: begin
26              case (func_code)
27                  `FUNC_ADD : begin ALUOperation = `OP_ADD; RegWrite =1; end
28                  `FUNC_WWD : begin ALUOperation = `OP_ID; RegWrite = 0; end
29              endcase
30          end
31
32          `OPCODE_ADI: begin
33              ALUOperation = `OP_ADD;
34              RegWrite = 1;
35          end
36
37          `OPCODE_LHI: begin
38              ALUOperation = `OP_LHI;
39              RegWrite = 1;
40          end
41
42          `OPCODE_JMP: begin
43              RegWrite = 0;
44          end
45      endcase
46      end
```

Figure 4. control_unit.v case statement

ALUOperation, RegWrite는 trinary operator로 구현하기에 복잡해서 case statement를 사용했다.

Datapath.v

```
45    // ID stage
46    wire [1:0]           rs, rt, rd;
47    wire [7:0]           imm;
48    wire [11:0]          target_imm;
49    wire [`WORD_SIZE-1:0] imm_signed;
50    wire [15 : 0] read_data1;
51    wire [15 : 0] read_data2;
52    wire [1 : 0] register_addr3;
53
54    //EX
55    wire [15 : 0] ALU_i2;
56    wire [15 : 0] ALU_out;
57    wire ALU_Overflow;
58
```

Figure 5. datapath.v port declaration

rs, rt, rd는 register의 address이며 RF module의 address input에 연결된다.

imm은 I type에 상용되며 이것을 sign-extend하여 16bit로 만든 것은 imm-signed이다. 이는 concatenation으로 쉽게 구현했다. imm_signed는 ALU SRCB에 해당하는 mux에 연결된다.

```
98    assign imm = data[7:0];
99    assign imm_signed = {{8{imm[7]}}, imm}; //sign-extended
```

Figure 6. datapath.v imm, sign-extend

target_imm은 J type에 사용되며 PC에 input으로 들어간다.

read_data1은 RF에서 읽은 1번째 rs data이며, read_data2는 RF에서 읽은 2번째 rt data이다.

register_addr3는 RegDst를 조건으로 하는 MUX의 output이며 이는 RF의 write address에 연결되어 있다.

```
95    assign register_addr3 = RegDst ? rd : rt; // MUX
```

Figure 7. datapath.v MUX1

ALU_i2는 ALUSRCB에 해당하며 r type일 때 rt의 data를 선택하고, I type일 때 imm의 값을 선택하는 MUX의 output이며, 동시에 ALU의 B input이다.

```
102   //ALU input
103   assign ALU_i2 = ALUSrc ? imm_signed : read_data2; // MUX
```

Figure 8. datapath.v MUX2

ALU_out은 말 그대로 ALU의 output이며, 이 LAB에서는 load, store가 없으므로 MEM에 접근하는 경우는 없으며 WB일 때만 사용된다. 따라서 ALU_out은 RF의 write data에 다른 module 없이 바로 연결했다.

```
105   // output_port only when isWWD
106   assign output_port = isWWD ? read_data1 : 16'bz;
```

Figure 9. datapath.v output_port

output_port는 WWD인 경우에만 rs의 data인 read_data1의 값을 내보낸다.

```
108   //num_inst
109   always @ (posedge clk) begin
110       if (!reset_n) num_inst <= 1;
111       else num_inst <= num_inst + 1; readM = 1;
112   end
```

Figure 10. datapath.v num_inst

매 cycle마다 num_inst를 1 증가하면 된다. Reset_n일 때 num_inst을 0으로 초기화하지 않고, 1로 초기화했다. 그 이유는 testbench의 num_inst와 맞추기 위해서이다. testbench에서는 input_ready가 들어오자마자 num_inst가 1이라고 맞춰야 된다고 판단했다.

```
114   // readM
115   always @ (posedge inputReady) begin
116       readM <= 0;
117    end
```

Figure 11. datapath.v readM

inputReady 신호가 들어오면 datapath의 output address와 input data가 달라야 하므로 readM을 0으로 하여 instruction을 읽을 준비를 한다.

PC.v

평상시에는 PC를 1 증가하나 jump 명령이 들어온 경우에만 pc를 target_address와 concatenation을 한다. Always statement로 매 cycle마다 updatate했다.

```
8   always @ (posedge clk) begin
9       if (!reset_n) address <= 0;
10      else if (jump) address <= {address[15 : 12], target_address};
11      else address <= address + 16'd1;
12  end
```

Figure 12. pc.v

ALU.v, RF.v는 이전 lab의 code를 재사용했으며, 따라서 별도의 설명은 없습니다.

4. Discussion

Figure 10, 11을 보면 각각 다른 always statement안에 readM의 신호를 내보내는 것을 알 수 있다. 이는 Verilog 문법 상에서는 문제 없으나 실제 회로를 구현할 경우 문제가 생길 수 있다고 생각한다. 왜냐하면 register는 매 clock 등 하나의 signal에 대하여 업데이트해야 하지만 이 code에서는 clk와 inputReady 2개의 signal에 대하여 update하기 때문에 회로로 어떻게 그릴 수 없다고 생각된다. 만약 이를 제대로 해결하려면 하나의 module을 새로 만들고, 그 안에서 readM을 처리해야 할 것 같으나 구체적인 방법이 떠오르지 않아 실행하지 못했다.

5. Conclusion

모든 연산을 구현하지 않고 일부만 구현했기 때문에 간단한 방법으로 구현할 수 있었다. 다만, discussion에서 언급한 서로 다른 always statement에서 하나의 reg를 바꾸는 것이 문제가 될 수 있고, 추후에 이를 해결할 것이다.