논리설계 및 실험 보고서

Final Project - 16-bit single cycle CPU

2017-10194 전기정보공학부 신지원

2017-19460 전기정보공학부 박준영

논리설계 및 실험 보고서

Final Project - 16-bit single cycle CPU

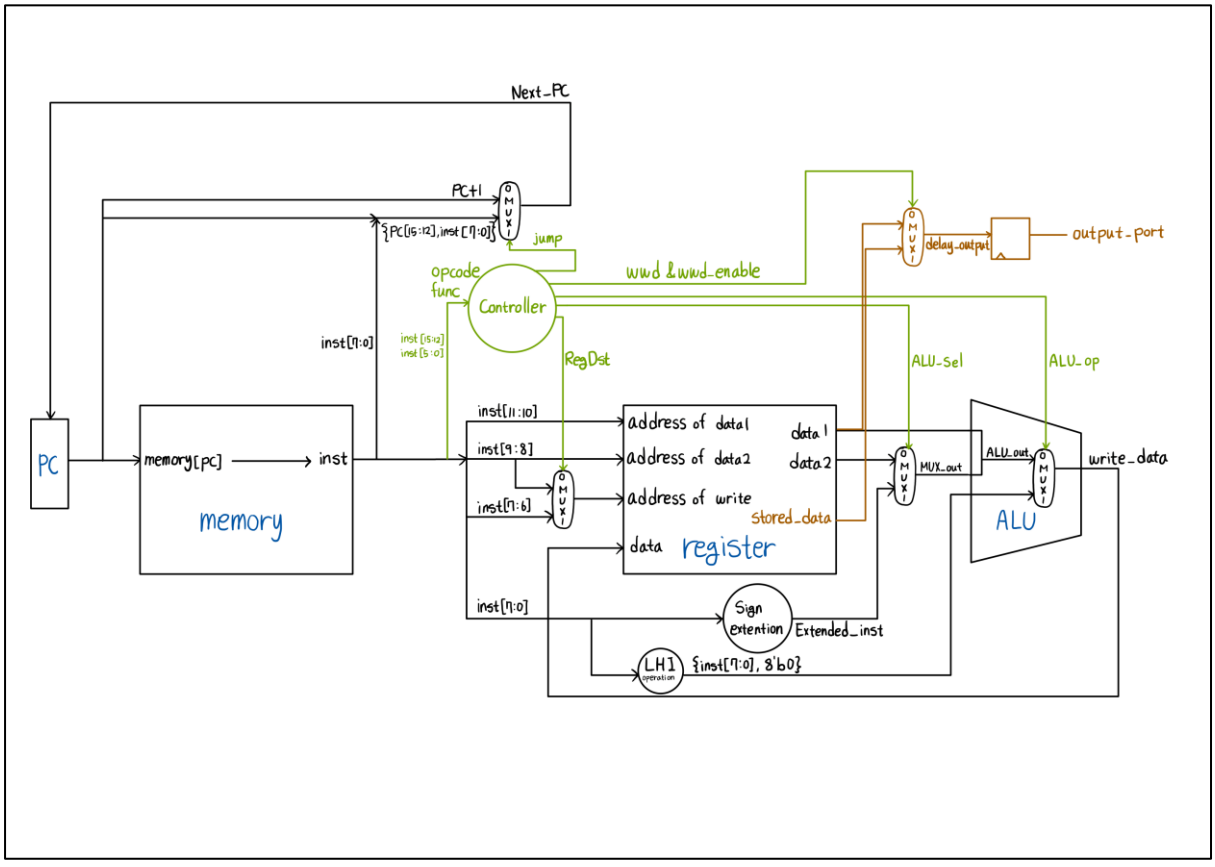
1. Introduction

논리설계 및 실험 기말 프로젝트는 ADD, ADI, LHI, JMP, WWD 의 5 개 연산만을 수행하는 16-bit single cycle CPU 를 verilog 언어를 이용해 FPGA 위에 구현하는 것을 목적으로 한다. 이 프로젝트를 통해 CPU 의 기본적인 구조와 verilog 의 문법을 이해하고 FPGA 의 사용법을 익힐 수 있었다.

2. Design

CPU 에서 구현해야 할 연산들은 다음과 같다.

- **ADD(R):** Add **rs** and **rt** register and save the result to **rd** register.
- **ADI(I):** Add **rs** register and **sign-extended imm[7:0]**, and save the result to **rd** register.
- **LHI(I):** Concatenate **imm[7:0]** with **eight 0's**, and save the result to **rt** register.
- **JMP(J):** Concatenate **pc[15:12]** with **target[11:0]**, save the result to pc register.
- **WWD(I):** Export the **rs** register value to the **output port**.



[그림 1] The brief schematic of the CPU

위 5 개의 기능을 구현하기 위한 회로도를 [그림 1]과 같이 간략하게 설계하였다. [그림 1]에 대해 조금 더 자세히 설명하자면, CPU 라는 하나의 큰 module 을 구현하기 위해 PC, Register, Controller, Instruction memory, ALU 로 나누어 구현한 다음 이들을 연결하였다.

- **PC:** 현재 instruction 의 주소를 저장하고, 만약 CPU 가 enable 된 상태라면 다음 clock 에서 PC 를 다음 instruction 이 저장된 memory 의 주소로 변경하는 module 이다.
- **Register:** Register 의 주소를 받는 port 2 개, destination register 의 주소를 받는 port 1 개, data 를 받는 port 1 개를 input 으로 받아 각각의 register 에 있는 data 2 개와 stored data 1 개를 output 으로 내보내는 module 이다.
- **Controller:** opcode(=inst[15:12])와 func(=inst[5:0])을 받아 instruction 을 구분하고, 이에 따라 multi-plexer 를 control 하는 module 이다.
- **Instruction memory:** Instruction 이 저장되어 있는 공간이다. PC 를 input 으로 받아 inst 를 output 으로 내보낸다.
- **ALU:** ALU 는 arithmetic logic unit 의 준말로 수학적인 연산을 담당하는 module 이다. 그러나 본 프로젝트에서는 단순히 두 input 을 더해 하나의 output 으로 내보내는 기능만을 수행한다.

3. Implementation

```
78      assign PC_below8bit = PC[7:0];
79      // initialize
80      always @(negedge clk) begin
81          if(reset_cpu == 1) begin
82              PC <= -1;
83              register[0] <= 16'b0;
84              register[1] <= 16'b0;
85              register[2] <= 16'b0;
86              register[3] <= 16'b0;
87          end
88          else if (cpu_enable == 1) begin
89              if(PC > `MEMORY_NUM-1 && &PC == 0) begin
90                  // over memory number
91              end
92              else begin
93                  PC <= Next_PC;
94                  if(Reg_write_en == 1) begin
95                     register[Write_reg] <= Write_data;
96                  end
97              end
98          end
99      end
100
101     always @(PC) begin
102         inst <= memory[PC];
103     end
```

```
142     // register path connect
143     assign data1 = register[inst[11:10]];                    // read data1
144     assign data2 = register[inst[9:8]];                   // read data2
145     assign stored_data = register[register_selection];// stored register
146     ////////////////////////////////////////////////////////////////////////
```

Register 와 PC 부분의 code 이다. 먼저 크기가 4 이고 각 원소가 16-bit 의 저장 공간을 가지는 reg 변수 'register'를 선언하고 그 초기값을 모두 0 으로 한다. 이후 'cpu_enable'이 들어오면 PC 를 Next_PC 로 업데이트 하고, register 에 Write_data 값을 쓰게 하였다. 'inst'는 PC 가 바뀔 때마다 그에 해당하는 memory 를 저장한다. always @(negedge clk)에서 알 수 있듯 register 는 clock-sensitive 하다.

```
105     // set controller output
106     assign opcode = inst[15:12];
107     assign func = inst[5:0];
108 
109     always @(opcode or func) begin
110         RegDst = 0;
111         jump = 0;
112         ALU_op = 0;
113         ALU_sel = 0;
114         Reg_write_en = 0;
115         wwd = 0;
116 
117         if (cpu_enable == 0) begin
118             Reg_write_en = 0;
119         end
120         if (opcode == 15 && func == `FUNC_ADD) begin
121             RegDst = 1;
122             Reg_write_en = 1;
123         end
124         if (opcode == 15 && func == `FUNC_WWD) begin
125             RegDst = 1;
126             wwd = 1;
127         end
128         if (opcode == `OPCODE_ADI) begin
129             ALU_sel = 1;
130             Reg_write_en = 1;
131         end
132           if (opcode == `OPCODE_LHI) begin
133             ALU_op = 1;
134             Reg_write_en = 1;
135         end
136         if (opcode == `OPCODE_JMP) begin
137             jump = 1;
138         end
139     end
```

Controller 부분의 code 이다. input 으로 'opcode(=inst[15:12])'와 'func(=inst[5:0])'을 받아 어떤 instruction 을 원하는지 판단하고, 이에 따라 MUX 를 조절하는 output 을 내보내는 과정을 구현하였다. output 은 0 으로 initialize 되어 있다.

```
151     assign Extended_inst = (inst[7] == 0) ? {8'b0, inst[7:0]} : {8'b11111111, inst[7:0]};
```

```
153     assign ALU_out = data1 + MUX_out;
```

Sign extension 과 ALU 의 code 이다. Sign extension 은 inst[7]의 bit 가 1 이면 앞의 8 개 bit 에 1 을, 0 이면 0 을 붙여서 output 으로 내보낸다. 이는 ADI 를 구현하기 위한 module 이다. ALU 에서는 data1 과 MUX_out 을 더해 output 으로 내보내고 있다.

4. Discussion

- **구현 과정에서의 어려움**

  주어진 PC 가 하나씩 증가하다가 27 에 도달한 후에 멈추어야 하는데, 이 때 PC 가 -1 인 것을 27 보다 크다고 인식하는 문제가 있었다. 이 때문에 &PC == 0 을 이용해 이 문제를 제거해야 했다. 이외에 verilog 의 문법에 익숙하지 않아 몇 가지 문제가 발생하였다.

- **디자인의 변화**

  초기에는 register 와 controller 를 별도의 module 로 구현하고자 하였으나 수정하였다.

5. Conclusion

본 프로젝트의 목표인 주어진 5 개 연산만을 수행하는 16-bit single cycle CPU 를 verilog 언어를 이용해 FPGA 위에 구현하는 데 성공하였다.