논리설계 및 실험 term project

16-bit single-cycle CPU on FPGA

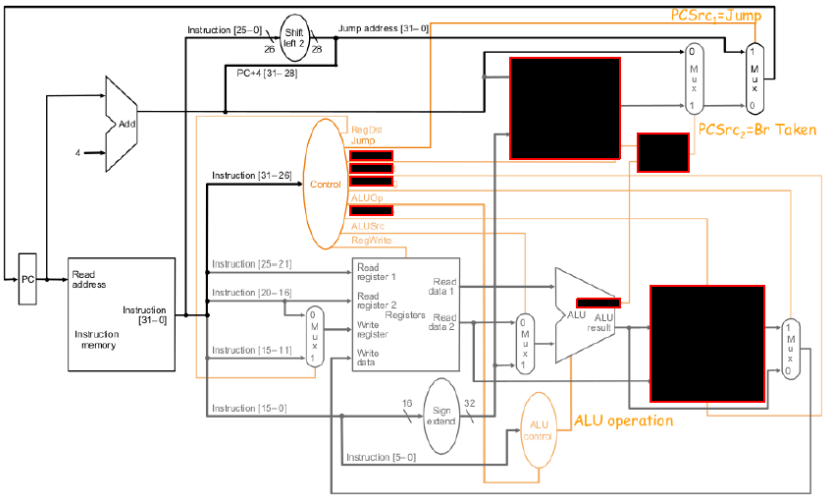
26 조(박선민, 송원재, 최석현)

1. Introduction

(1) 프로젝트 목표

1 cycle 동안 하나의 instruction 을 수행할 수 있는 single-cycle CPU 를 Verilog 를 이용하여 설계하는 것이다. 5 가지 instruction(ADD, ADI, LHI, JMP, WWD)을 수행할 수 있도록 설계하고 FPGA 보드 상에 구현하여 제대로 동작하는지 확인해본다.

(2) TSC 16-bit Instruction set architecture



TSC 16-bit Instruction set architecture 의 구조는 위와 같다. Instruction 을 저장하고 1 clock cycle 마다 하나의 Instruction 을 수행한다. Instruction 을 수행한 후에 PC register 에 다음 주소를 저장하고 다음 clock 에 해당 주소의 Instruction 을 수행한다. 위와 같은 module 을 Verilog 를 통해 구현하고 FPGA board 를 Programming 하여 목표한 동작을 순서에 맞게 수행하는지 확인한다. 이는 FSM 에서 하나의 state 와 input 을 구성하고, current state 에서 instruction 수행 후, 이들 장치에 있는 value 가 갱신될 때마다 current state 는 next state 로 이동한다. CPU 에서는 Instruction execution 과 state 갱신이 이뤄지며, 이를 Verilog 를 이용해 구현하고자 한다.

2. Design

(1) 각 Module 의 역할

1) Cpu_top module: input 과 Clock_generation_module 의 clock 을 받아들이고 cpu 가 역할을 할 수 있게 전달하고 cpu 에서 나온 output 을 output_logic_module 에 전달하여 원하는 output 을 도출할 수 있도록 한다.

2) Clock_generation module: MCLK signal 을 이용하여 새로운 주기의 clock 을 만든다.

3) Cpu module: cpu_top_module 에서 신호를 받아 위의 architecture 대로 Instruction 을 수행한 후에 결과에 따른 output 을 내보낸다.

4) Output_logic module: Instruction 수행 결과를 input 으로 하여 LED 와 7-segment 에 output 을 보여주기 위해 저장한다.

(2) CPU module

Instruction 은 5 가지 종류이다. ADD 는 $rs register 에 저장된 값과 $rt register 에 저장된 값을 더하고 이를 $rd register 에 저장한다. ADI 는 $rs register 에 저장된 값과 instruction 의 imm[7:0]에 저장된 값을 16-bit 로 sign extend 한 뒤에 이를 더하고 $rt register 에 저장한다. LHI 는 instruction 의 imm[7:0]에 저장된 값 뒤에 8-bit 0 을 concatenate 하고 $rt register 에 저장한다. JMP 는 PC[15:12]와 target[11:0]를 concatenate 하고 $PC 에 저장한다. WWD 는 $rs register 에 저장된 값을 7-segment 로 보여주기 위해 output_port 에 $rs register 에 저장된 값을 넘긴다.

3. Implementation

Source code 는 아래와 같이 크게 두 부분으로 나눌 수 있다.

1 동작 수행을 위한 Register 선언 및 Combinational Circuit 구현

2 Clock, Reset_cpu 신호에 따른 Sequential Circuit 구현

위 두가지 과정을 통해 16-bit Instruction set architecture 의 5 개 동작을 수행할 수 있다.

- Register 선언 및 Combinational Circuit

PC control 및 instruction 선언

```
reg [`WORD_SIZE-1:0] PC;
reg [`WORD_SIZE-1:0] Jump_target;
wire [`WORD_SIZE-1:0] Instruction;
assign Instruction = memory[PC];
```

: PC register 에 수행해야할 Instruction 의 주소를 저장하고 매 clock 마다 다음 target Instruction 주소를 저장한다. Jump 명령어를 수행할 경우 Instruction 의 target address 를 'Jump_target' register 에 저장한다. memory[PC]에 저장되어 있는 명령 data 를 'Instruction' port 를 통해 받는다.

Register, wwd_out, PC_below8bit 선언

```
reg [`WORD_SIZE-1:0] register[0:3];
reg [15:0] wwd_out;
assign PC_below8bit = PC[7:0];
```

: 연산 혹은 저장을 위한 register 를 선언한다. Project 목표에 맞게 16bit_size register 4 개를 선언하여 Instruction 수행에 이용하였다. WWD Instruction 을 수행하기 위해 FPGA 의 Display 에 나타낼 output 을

전달하기 위한 'wwd_out' register 를 선언한다. 그리고 output_logic_module 에 넘겨주기 위한 PC_below8bit 를 선언하고 저장한다.

- Clock, Reset_cpu 에 따른 Sequential Circuit

Clock, Reset Control

```
always@(posedge clk or posedge reset_cpu) begin
        ~
end
```

: clock 에 맞추어 Instruction 을 수행하기 위해 clock signal 이 posedge 일 때에 맞추어 Instruction 하나씩 수행한다. 또한 reset_cpu 신호를 asynchronous 하게 받아들이기 위해 clock signal 과 별개로 감지하고 reset_cpu 가 들어왔을 때 cpu 의 모든 값을 초기화 할 수 있도록 한다.

Reset_cpu 의 동작

```
if(reset_cpu == 1'b1) begin
        PC = 16'd0;
        register[0] = 16'd0;
        register[1] = 16'd0;
        register[2] = 16'd0;
        register[3] = 16'd0;
        wwd_out = 16'd0;
end
```

: 'reset_cpu' 신호가 들어왔을 때에 PC, register, wwd_out 의 모든 값을 0 으로 초기화한다.

CPU enable

```
If (cpu_enable == 1) begin
        ~
end
```

: 'cpu_enable' 신호가 active 일 때만 cpu 가 동작해야 한다.

PC 할당

```
if (Instruction[15:12] == 4'd9)
        PC = {PC[15:12],Instruction[11:0]}; //if jump instruction -> jump
else
        PC = PC + 1;
```

: 'jump' 명령을 수행했을 때에는 PC register 에 target address 를 저장하고 그 외의 경우에 다음 Instruction 을 읽어올 수 있도록 'PC + 1'을 저장한다.

연산 수행 : ADI, LHI, ADD, WWD

```
case(Instruction[15:12])
        4'd4:           //ADI
                register[Instruction[9:8]]
                 = register[Instruction[11:10]] + {{8{Instruction[7]}},Instruction[7:0]};
        4'd6:           //LHI
                register[Instruction[9:8]] = {Instruction[7:0],8'b0};
        4'd15:          //ADD & WWD
                begin
                if(Instruction[5:0] == 6'd0)
                        register[Instruction[7:6]]
                         = register[Instruction[11:10]] + register[Instruction[9:8]];
                else if(Instruction[5:0] == 6'd28)
                        wwd_out = register[Instruction[11:10]];
                end
endcase
```

: Instruction 의 앞 4bit, opcode 를 ADI, LHI, ADD&WWD 세가지로 분류하여 해당 동작을 수행
ADI - $rt <- $rs + sign_extended(imm[7:0])
LHI - $rt <- {Imm[7:0], 8'b0}
ADD - $rd <- $rs + $rt
WWD – wwd_out <- $rs

WWD output 출력

```
assign output_port = (wwd_enable == 0)? register[register_selection[1:0]]: wwd_out;
```

: 'wwd_enable' 값이 1 일 경우 최근에 수행한 WWD Instruction 의 output 을 wwd_out 에 저장하고 0 일 경우 register selection bit 가 가리키는 register 의 값을 wwd_dout 에 저장한다.

4. Discussion

Hardware 의 Circuit 을 코드로 구현하는 것이다 보니 0 과 1 로 표시되는 비트 하나하나를 고려해 주는 점이 낯설어 접근이 쉽지 않았다. 가령 '1111 1111' 비트에 1 을 더해준 결과 '0000 0000'이 출력되는 과정이 Code 로 직관적으로 이해하기에는 어려움이 있었다. CPU 의 동작 과정을 따라가는 데에도 이러한 이유로 더딤이 있었고 High level Coding 보다 조금 더 컴퓨터, 기계 친화적인 코드가 어떠한 것인지 경험하였다.

clock 을 이용하여 synchronous 동작들과 asynchronous 동작을 구분하는 데에 있어서 혼란스러웠다. Clock 에 맞춰 값을 저장 해야할지, 값을 내보내야 할지 port 를 어떻게 연결해야 할 지 많은 고민이 요구되었다. 특히 Output_port 의 경우 WWD Instruction 을 통해서 output 이 나가고 wwd_enable 신호로 asynchronous 하게 selection bit 의 register 값을 내보내기도 해야 했기에 output 을 7-segment 로 표출하는 것에 잦은 문제를 겪었다. 표출된 값이 logic 이 의도한 값이 맞는지, 맞는 flow 로 동작하고 있는 지 확인하는 데에도 쉽지 않았다.

Verilog 구현에서 가장 힘든 점은 바로 Debugging 과정이었다. 문법적으로 문제가 없는 코드지만 원하는 동작을 실행하고 있는지 확인하는 과정이 쉽지 않았고 예상한 결과와 다른 동작을 하는 경우가 많았다. 예로 Sign_extend 과정에서 가장 앞의 1bit 를 복사하여 붙여주는 과정에서의 문제점을 찾지 못해 Debugging 과정이 길어지기도 하였다. 이를 포함한 낯설음에서 유발되는 여러가지 문제들을 해결하기가 번거롭고 쉽지 않았다.

Single Cycle Cpu 의 동작 과정을 살펴보며 High version 인 CPU 의 구현 에 대해 생각해 볼 수 있었다. Single Cycle 에 한 개씩 들어가는 Logic 들을 여러 개를 만들어 동시에 다수의 Instruction 을 실행하거나 Clock 을 더 작은 단위로 나누어 다른 Logic 들이 작동할 시간에 다음에 올 Instruction 을 미리 실행하는 방법이 있다. 이를 구현하기 위해서는 회로가 더 커짐에 따라 Timing Issue 에 대해서 더 신경을 써야 할 것이고 Combinational Logic 들과 Sequential Logic 사이의 Timing 에 문제가 되지 않도록 설계해야 할 것이다.

5. Conclusion

주어진 Instruction set 의 동작 결과가 예상과 맞아 떨어졌고 FPGA Board 위에 원하는 결과값을 성공적으로 표출하였다. 논리 회로가 어떤 방식으로 동작하고 어떤 식으로 설계가 되는지 느껴볼 수 있는 프로젝트였고 간단한 CPU 의 동작을 확인할 수 있었다.