**Computer Organization Report**

- Single Cycle CPU -

2012-11442 허문혁

* Introduction

이번 Lab에서 구현할 것은 Single Cycle CPU이다. 기존에 작성했던 ALU와 RF 등을 이용해서 CPU를 구현하는 것이다. 크게는 Datapath와 Control Unit으로 나누어 구현한다. 이번 Lab을 통해서 실제로 CPU가 어떤 방식으로 Instruction을 처리하는 지 알 수 있을 것이다.

* Design

CPU는 크게 Datapath와 Control Unit으로 나누어서 구현할 것이다. Control Unit은 CPU가 Fetch한 Instruction의 Opcode에 따라서 이에 맞는 Control Signal을 Datapath에 주는 역할을 한다. Datapath는 CPU 내에서 Data들이 흘러가는 곳이다. Register, ALU 모두 이곳에 속한다. 이번 Lab에서 구현을 할 때 최대한 Module을 세세하게 짜고자 한다. 이전까지의 Lab에서 구현한 것과는 다르게 많은 Module들이 필요하고, wire로 연결해주어야 하는 것들도 많기 때문에 최대한 Module을 세세하게 나누어서 부분별로 구현한 후에 Datapath에 올리고자 한다. 그리고 이번 Lab에서는 몇 가지 안되는 Opcode만 구현하기 때문에 굳이 Branch Mux나 AND Gate 등을 구현할 필요는 없었지만 이후에 Multi Cycle CPU나 Pipelined CPU를 구현할 때 어차피 필요한 것이므로 최대한 이에 맞게 구현하고자 했다.

* Implementation

처음으로는 Instruction Memory에서 Instruction을 Fetch해 오는 것을 구현했다. Clk의 Positive Edge에 readM Signal을 1로 올려주고 Memory에 Instruction을 읽겠다는 신호를 보낸다. Instruction Memory에서 readM을 확인하고, Instruction을 보내고 나면, CPU에게 Data를 보냈다는 신호인 inputReady Signal을 주게 되는데, 이를 확인하고 빠르게 Data를 받은 다음 다시 readM을 내려주어야 한다. 즉 inputReady Signal이 1로 바뀌었을 때의 Data를 받아야 하므로, inputReady의 Positive Edge에 Data를 받아오도록 구현했다. 아래와 같이 이 과정을 구현했다.

always @(posedge clk) begin

if(!reset\_n)

readM <= 0;

else

readM <= 1;

end

always @(posedge inputReady) begin

fetch = data;

num\_inst = num\_inst + 1;

readM = 0;

end

다음으로는 Control Unit의 구현 과정이다. Control Unit은 Input으로 Opcode를 받아서 이에 맞는 Control Signal을 DataPath로 보내주는 역할을 한다. 그래서 받아온 OPCode별로 Case문을 이용해서 구현하였다. 그리고 Signal 중에서 0이든 1이든 Instruction을 수행하는 데 문제가 없는 Don’t care라면, 0으로 구현하였다. 아래가 Opcode에 따라 바뀌는 Control Signal을 구현한 부분이다.

case(OP)

4'd1:

begin

end

4'd2:

begin

end

4'd3:

begin

end

4'd4: // ADI

begin

RegDst <= 0;

Jump <= 0;

Branch <= 0;

MemRead <= 0;

MemtoReg <= 0;

MemWrite <= 0;

ALUSrc <= 1;

RegWrite <= 1;

end

4'd5:

begin

end

4'd6: // LHI

begin

RegDst <= 0;

Jump <= 0;

Branch <= 0;

MemRead <= 0;

MemtoReg <= 0;

MemWrite <= 0;

ALUSrc <= 1;

RegWrite <= 1;

end

4'd7:

begin

end

4'd8:

begin

end

4'd9: // JMP

begin

RegDst <= 0; // Don't Care

Jump <= 1;

Branch <= 0; // Don't Care

MemRead <= 0;

MemtoReg <= 0; // Don't Care

MemWrite <= 0;

ALUSrc <= 0; // Don't Care

RegWrite <= 0;

end

4'd10:

begin

end

4'd11:

begin

end

4'd12:

begin

end

4'd13:

begin

end

4'd14:

begin

end

4'd15: // ADD, SUB, AND, ORR, NOT, TCP, SHL, SHR, RWD, WWD Operation.

begin

RegDst <= 1;

Jump <= 0;

Branch <= 0;

MemRead <= 0;

MemtoReg <= 0;

MemWrite <= 0;

ALUSrc <= 0;

if(func == 6'd28)

begin

RegWrite = 0;

end

else

begin

RegWrite = 1;

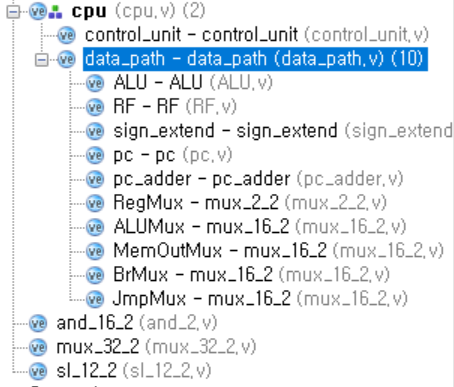
end

end

endcase

end

마지막으로 DataPath인데, 새롭게 많이 구현한 것은 없고 최대한 모듈 단위로 잘게 쪼개서 구현하였다. Branch Mux나 And Gate 등도 차후에 사용하기 때문에 구현만 해 놓은 상태이다.



* Discussion

CPU를 직접 구현해 보았는데, 생각보다 어려운 것은 Wire나 Reg 값들의 값을 알기 쉽게 표현하는 것이었다. 아무래도 Module도 많아지고, 코드도 길어지다보니 알아보기 쉽게 코드를 짜는 것이 어려웠다. 그리고 Clk에 따라서 바꿔주어야할 값과 그냥 Assign을 해야하는 값을 구분하기가 좀 어려웠다. 그리고 Debugging을 하는 데 특히 시간이 많이 걸렸는데 Register FIle이라든지 ALU라든지 Mux의 Input Output이라든지 봐야하는 값이 너무 많아서 Debugging이 전 과제들보다 훨씬 어려웠다. 또 가장 어려웠던 것 중 하나는 readM과 inputReady 타이밍에 맞춰서 Instruction을 Fetch해 오는 것이었다. ReadM과 InputReady가 모두 1 일때만 Data가 주어지기 때문에 이 타이밍에 맞추어서 Reg 값에 저장하고, 빠르게 ReadM을 내려주도록 구현하는 게 가장 어려웠던 것 같다.

* Conclusion

이번 Lab을 통해서 Single Cycle CPU를 구현해 보았다. 물론 수업시간에 배운 내용이었지만 실제적으로 구현해보면서 CPU에 대해서 더 자세히 이해하게 되었다. Instruction 을 Memory로부터 읽어오고, Instruction을 Decode해서 각 Instruction에 맞는 Control Signal을 생성하고, 이를 전달해주어 이에 맞는 Operation을 하는 과정을 하나하나 구현하면서 CPU의 Instruction 처리 과정을 이해하였다. 그리고 이번에는 Module을 구현한 후, 그 Module을 상위 Module에 올리는 방법을 사용해서 구현하였는데, 이를 통해 Module화 하여 구현하는 방법에 대해서 알게 되었다.