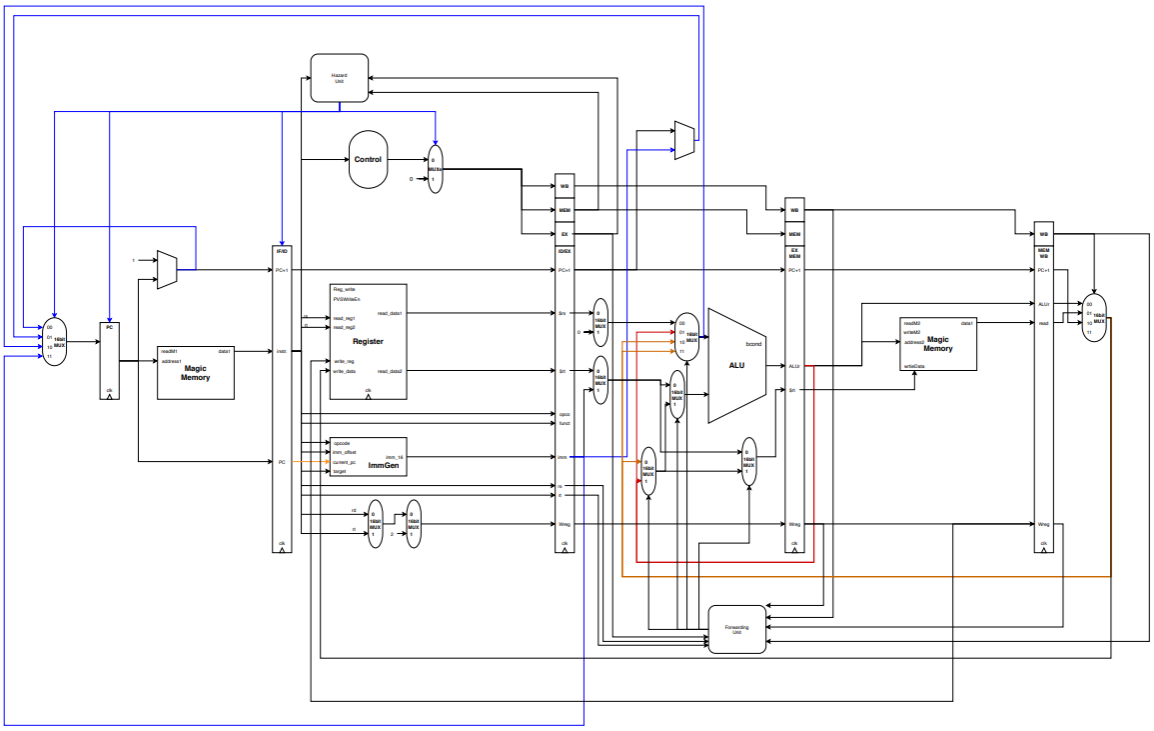
LAB 5. Pipelined CPU

201800038 박형규, 20180480 성창환

1. Introduction

이번 Lab에서는 pipelined TSC CPU를 구현하는 것을 목적으로 하였다. TSC_manual.pdf에 나와있는 내용에 따라 각각 R type, I type, J type에서 올바른 action이 수행되도록 하였다. Pipelined CPU를 구현하면서 수업에서 배운 Pipelined CPU의 구조와 작동하는 원리를 익힌다. 또한 Hazard가 일어나는 것을 해결해주기 위한 Hazard Detection Unit과 Forwarding을 해주는 Forwarding Unit을 만들어 CPU의 성능을 향상시켰다.

2. Design



이전에 구현했던 Single Cycle CPU를 바탕으로 Pipeline들을 추가하고, Hazard Detection Unit과 Forwarding Unit을 추가하여 주었다.

3. Implementation

먼저 pipeline.v에서는 pipeline들을 구현해 주었다. 각 Stage를 넘어갈 때 필요한 여러 값들

을 pipeline에 저장하고 이를 다음 Stage에서 정상적으로 사용할 수 있도록 하였다. Control Signal 또한 pipeline들을 통하여 각 필요한 Stage에 정상적으로 전달될 수 있도록 하였다.

Hazard_forward.v는 Hazard Detection unit과 Forwarding Unit을 구현하였다. Hazard Detection Unit은 Load Instruction이 실행된 뒤에 동일한 레지스터를 사용하는 instruction이 오게 되면 한번 Stall이 필요한 것과 Branch문과 Jump문에서 Miss prediction이 일어났을 때 Stall이 필요한 것을 처리해주기 위한 unit이다. 또한 Hazard Detection Unit에서는 next PC값을 결정하도록 해주는 Signal도 발생시켜주었다.

Forwarding Unit에서는 ALU앞에 input으로 들어가는 MUX에 해당되는 signal들인 FORWARD A와 FORWARD B signal을 발생시켜 주는 unit이다. Forwarding의 의미에 맞게 레지스터가 겹치게 된다면 정상적으로 Forwarding이 이루어 질 수 있도록 하였다.

Datapath.v에는 이전에 구현하였던 Multi Cycle CPU에서의 ALU, imm_gen, GPR, program_counter, MUX들을 거의 동일하게 구현하였고, pc값 계산에 필요한 adder만 추가해주었다.

Control.v에서는 alu control unit과 main control unit을 구현하였다. Alu control unit은 이전 Multi Cycle CPU와 동일하게 구현하였고, main control unit에서는 Pipelined CPU를 구현하는데 필요한 여러 Signal들을 발생시켜주었다.

마지막으로 CPU.v에서는 여러 wire들을 연결해주고, num_inst를 증가시켜주었다. 또한 Halt와 WWD도 이 모듈에서 구현되었다.

4. Discussion

먼저 이전 Lab에서 구현하였던 Multi Cycle CPU를 TB Code를 돌렸을 때 몇 사이클이 나오는지를 확인해 보았다.

```
VSIM 2> run -all
# Clock # 8474
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Modeltech_pe_edu_10.4a/examples/cpu_TB.v(151)
#    Time: 847550 ns  Iteration: 2  Instance: /cpu_TB
# 1
# Break in Module cpu_TB at C:/Modeltech_pe_edu_10.4a/examples/cpu_TB.v line 151
```

위의 그림처럼 총 8474cycle이 소요되었다. 이후에는 이번 Lab에서 구현한 Pipelined CPU를 TB Code를 돌렸을 때 몇 사이클이 나오는지 확인해 보았다.

```
VSIM 2> run -all
# Clock # 1435
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Users/hyeongkyu/Desktop/pipeline_final/cpu_TB.v(153)
#    Time: 143650 ns  Iteration: 2  Instance: /cpu_TB
```

위 그림처럼 총 1435cycle이 소요되었다. 이는 기존 Multi Cycle CPU는 한 instruction이 완전히 종료되어야 다음 instruction을 가져오는데, Pipelined CPU는 현재 instruction이 완전히 종료되지 않아도 다음 instruction을 받아와 진행시키기 때문에 기존에 비해 더 나은 throughput을 가지게 된 것이다. 이 같은 성능의 증가는 약 83%정도 증가한 성능을 보이게 되었다.

Pipelined CPU를 구현할 때 Num instruction이 증가하는 타이밍이 원하는 타이밍과 맞지 않아서 어려움을 겪었다. 하지만 여러 control signal들을 조합하여 결국 성공적으로 Pipelined CPU를 구현할 수 있었다.

5. Conclusion

이번 랩을 통해서 pipelined cpu를 성공적으로 구현했다. 이전에 구현하였던 Multi Cycle CPU는 한 instruction이 완전히 종료되어야 다음 instruction을 Fetch할 수 있지만, Pipelined CPU는 현재 instruction이 완전히 종료되지 않아도 다음 instruction을 받아와서 진행하기 때문에 기존에 비해 더 나은 throughput을 가지게 된다. 또한 RAW dependency로 인해 Pipelined CPU에서 다음 instruction을 올바르게 수행하기 위해서는 데이터가 온전히 Write Back 될 때까지 대기해야 하거나, 필요한 만큼 대기해주지 않아 잘못된 value를 이용하게 될 수 있다. 따라서 이를 방지하기 위하여 기존 WB Stage에서 레지스터에 값을 쓰는 대신, 레지스터에 쓰이는 결과값은 Ex Stage에서 도출되고, instruction상에서 레지스터의 데이터를 가져오는 시기도 Instruction Decode 대신 Ex Stage에서 가져옴으로써 dependency를 최대한 줄이는 Forwarding 방법을 사용하였다.