20170699 최진성 20170080 김동규 Lab Report

Lab 5. Pipelined CPU Lab

1. Introduction

Pipelined CPU를 Verilog를 사용하여 실제로 구현한다. 지난 Lab3와 Lab4에서 각각 구현하였던 Single-cycle, Multi-cycle CPU를 기반으로, 각각의 클럭 사이클마다 분리된 5단계의 동작 (IF, ID, EX, MEM, WD)이 이루어지는 동시에 through-put을 증가시키는 디자인을 구현하였다. 이를 위해 각 동작에 필요한 clk만 사용할 뿐만 아니라 각 단계에서의 결과값, 각 동작에 필요한 control signal 들을 stage 사이에 있는 register에 저장하여 clk에 따라 다음 단계로 이동하도록 한다. 이를 위하여 각 stage마다 어떤 결과값들과 control signal이 필요한지, 이를 어떻게 저장할 것인지에 대한 이해가 필요하다. Pipeline을 설계함에 따라 발생하는 여러가지 문제들 (Control Hazard, Data Hazard)를 해결하기 위한 forwarding과 필요에 따른 Stall, 그리고 2-bit Branch Prediction까지 모두 구현하였다.

2. Design

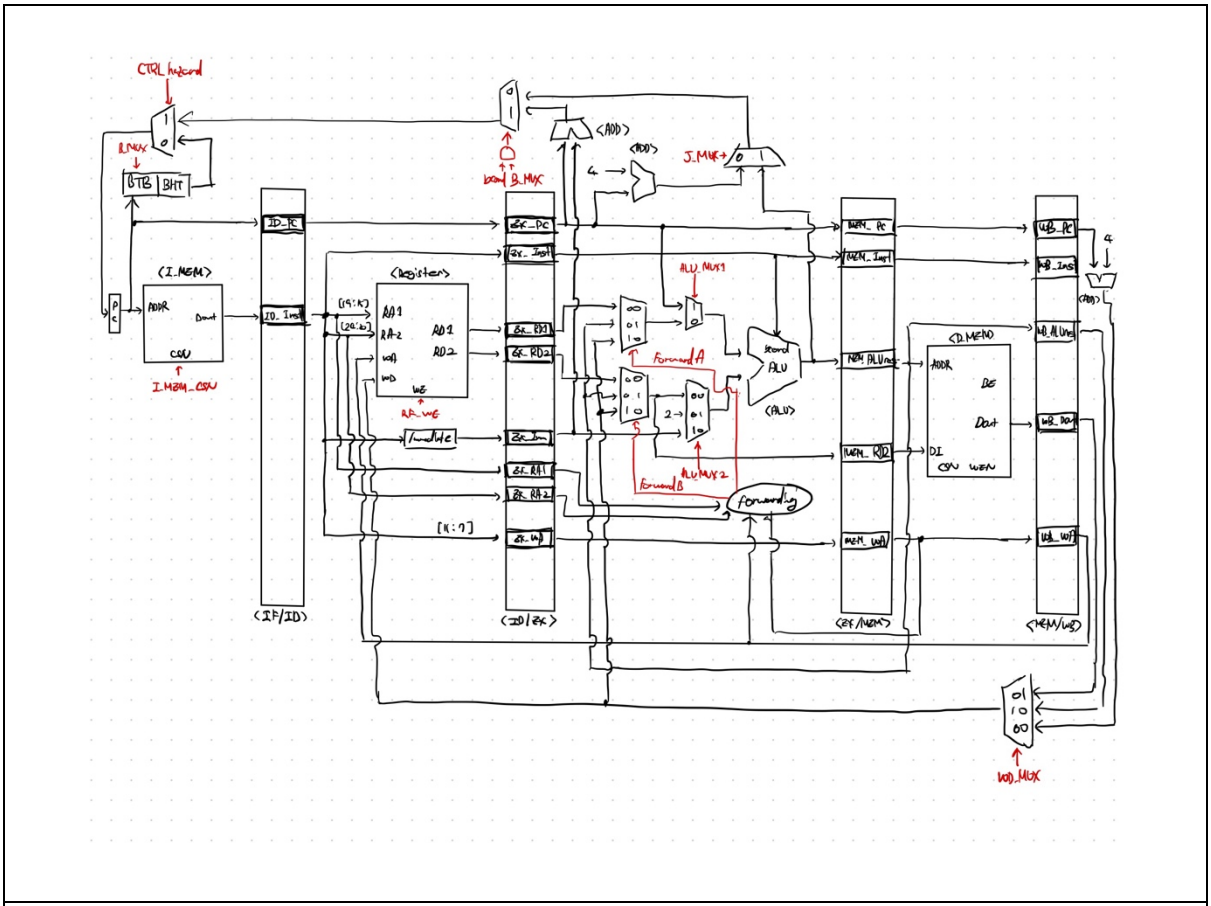


Figure 1. Lab 5에서 구현하고자 하는 Pipelined CPU의 Design

구현에 앞서 만들고자 하는 CPU의 구조를 위와 같이 설계하였다. 해당 Diagram은 data path를 나타낸 그림이다. 각 Stage 사이에 다음 stage에 필요한 정보를 저장할 수 있도록 register를 배치하였다. 이때 발생하는 여러 hazard에 의한 cost를 최소화 하기 위하여 여러가지 기법을 구현하였다. 첫째로 data dependency distance를 줄이기 위한 **forwarding unit을 구현**하였다. 특정 조건을 만족하면 MEM stage 혹은 WB stage에 있는 값을 바로 현재 Execution Stage에서 쓸 수 있도록 한다. 그러나 여전히 Load Instruction에서는 Data hazard가 발생하게 된다. 이럴 경우 1번 동안 Stall을 할 수 있도록 해당 Load에 의한 Data Hazard를 detect하는 hazard detector를 구현하였다. Control Logic에 의한 Hazard 또한 존재한다. 따라서 이러한 control hazard를 detect하기 위한 detector 또한 구현하였다. Branch Prediction을 활용하여 control로 인한 stall을 줄이도록 하였고, Branch와 Jump Instruction일 때의 Target PC를 저장하는 BTB와, 더 나은 non-taken, taken 예측을 위한 **2-bit saturation counter를 BTB와 함께 구현하였다**. Load에 의한 data hazard, Misprediction으로 인한 control hazard의 경우 bubble을 각각의 경우 1개와 2개씩을 삽입하여 정상적인 작동이 이루어질 수 있도록 하였다.

3. Implementation

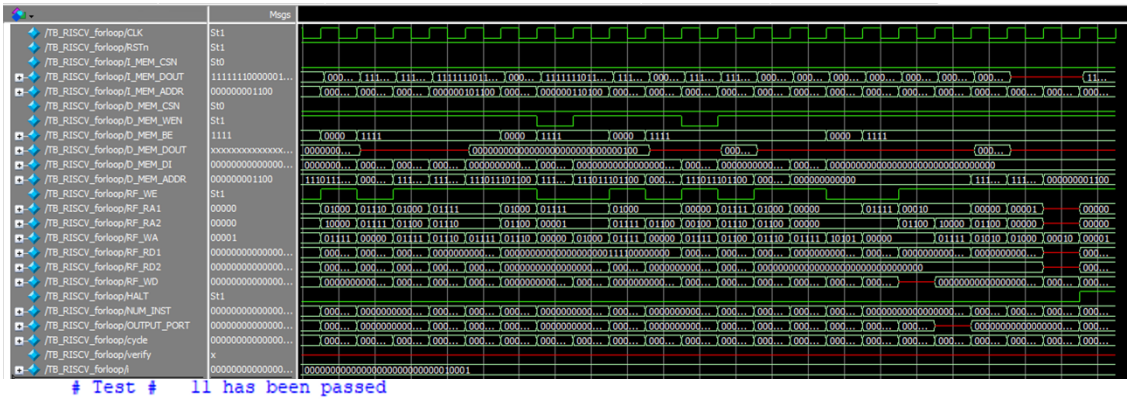
위에서 설계한 Register들을 기존의 Multi-Cycle CPU Lab에서 구현하였던 기본 구조 위에 얹혀져서 각 clock cycle마다 각 단계들이 서로 다른 instruction의 서로 다른 stage를 동작할 수 있도록 RISCV_TOP을 구현하였다. 새롭게 추가된 Unit인 Forwarding Unit과 Data Hazard Detection Unit, Control Hazard Detection Unit, BTB module를 별로의 모듈로 구현하였고 이를 RISCV_TOP에서 모두 연결하였다. ALU, CTRL 파일은 기존의 틀을 그대로 활용하였고, CTRL의 경우 각 control signal이 register를 타고 이동하는 것은 data path와 마찬가지로 RISCV_TOP에서 동작하도록 구현하였다. 각 stage별로 추가된 module들의 output이 올바른 MUX input으로 동작하여 hazard를 최소화하며 정상 동작할 수 있도록 구현하였다.

4. Evaluation

완성된 코드를 컴파일한 이후 주어진 3 개의 testbench 파일을 사용하여 시뮬레이션 결과를 확인하였다. 그 결과 모든 테스트에 대해서 다음과 같이 정상적인 결과값이 나오는 것을 확인하였다. 특히 Always predict to PC+4 case 와 2-bit saturation counter with BTB case 를 비교하여 2-bit saturation counter 와 BTB 구현이 Cycle 수를 줄였다는 것을 forloop 와 sort 에서 확인할 수 있었다. 시뮬레이션 결과는 다음과 같다.

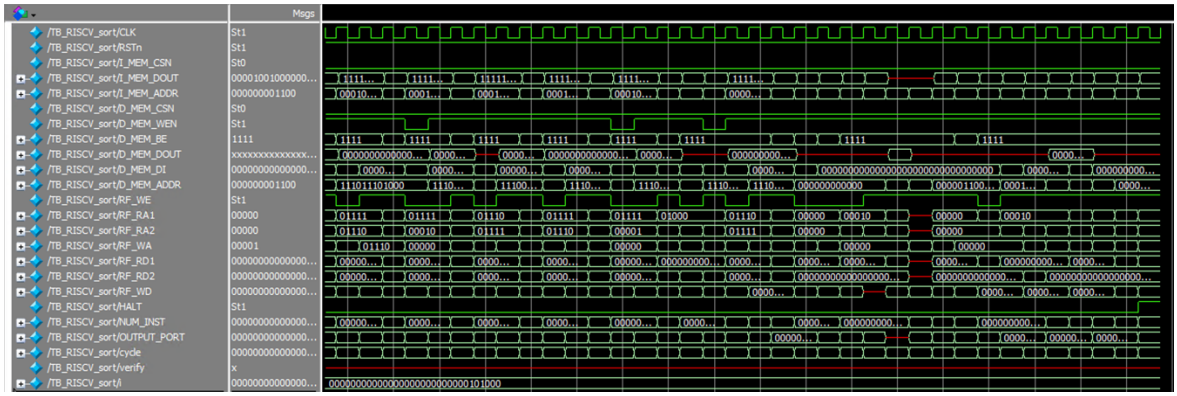
```
# Test #    21 has been passed
# Test #    22 has been passed
# Test #    23 has been passed
# Test #    24 has been passed
# Test #    25 has been passed
# Test #    26 has been passed
# Finish:         31 cycle
# Success.
# ** Note: $finish    : C:/modelsim_project/lab5/TB_RISCV_inst.v(179)
#    Time: 425 ns  Iteration: 1  Instance: /TB_RISCV_inst
```

Figure 2. 주어진 Testbench 파일로 실행한 시뮬레이션 결과 – inst



```
# Test #    11 has been passed
# Test #    12 has been passed
# Test #    13 has been passed
# Test #    14 has been passed
# Test #    15 has been passed
# Test #    16 has been passed
# Test #    17 has been passed
# Finish:         97 cycle
# Success.
# ** Note: $finish    : C:/modelsim_project/lab5/TB_RISCV_forloop.v(167)
#    Time: 1085 ns  Iteration: 1  Instance: /TB_RISCV_forloop
```

Figure 3. 주어진 Testbench 파일로 실행한 시뮬레이션 결과 (with BTB) – forloop



```
# Test #    35 has been passed
# Test #    36 has been passed
# Test #    37 has been passed
# Test #    38 has been passed
# Test #    39 has been passed
# Test #    40 has been passed
# Finish:       13462 cycle
# Success.
# ** Note: $finish    : C:/modelsim_project/lab5/TB_RISCV_sort.v(193)
#    Time: 134735 ns  Iteration: 1  Instance: /TB_RISCV_sort
```

Multi-Cycle Case:

```
# Test #    35 has been passed
# Test #    36 has been passed
# Test #    37 has been passed
# Test #    38 has been passed
# Test #    39 has been passed
# Test #    40 has been passed
# Finish:       41477 cycle
# Success.
# ** Note: $finish    : C:/modelsim_project/lab4/TB_RISCV_sort.v(193)
#    Time: 414875 ns  Iteration: 1  Instance: /TB_RISCV
```

Figure 4. 주어진 Testbench 파일로 실행한 시뮬레이션 결과 (with BTB) - sort

```
# Test #    12 has been passed
# Test #    13 has been passed
# Test #    14 has been passed
# Test #    15 has been passed
# Test #    16 has been passed
# Test #    17 has been passed
# Finish:        103 cycle
# Success.
# ** Note: $finish    : C:/modelsim_project/lab5/TB_RISCV_forloop.v(167)
#    Time: 1145 ns  Iteration: 1  Instance: /TB_RISCV_forloop
# 1
# Test #    35 has been passed
# Test #    36 has been passed
# Test #    37 has been passed
# Test #    38 has been passed
# Test #    39 has been passed
# Test #    40 has been passed
# Finish:       14730 cycle
# Success.
# ** Note: $finish    : C:/modelsim_project/lab5/TB_RISCV_sort.v(193)
#    Time: 147415 ns  Iteration: 1  Instance: /TB_RISCV_sort
# 1
```

Figure 5. BTB가 없는 경우 (always PC+4)의 forloop와 sort 결과

Waveform 관찰을 통해 각 clock마다 instruction들이 정상적으로 stage를 이동하며 동시에 여러 instruction들의 각기 다른 stage를 실행하는 것을 확인할 수 있다. 또한 testbench를 통과하기 위한 total cycle 수가 multi-cycle CPU에서 41477 cycles 이었던 것에 반해 해당 Lab에서 구현한 코드는 13642 cycles 로 필요한 cycle 수가 현저히 줄었다는 것을 확인할 수 있다. (Figure 4 참조)

5. Discussion

구현해야 하는 Pipeline CPU가 동작해야 하는 Instruction마다 어떤 Stage들을 거치고, 이를 위해 각 cycle마다 stage별 결과값들을 저장하는 register 설계하여 동작하게 할 것인지에 대한 고민이 필요하였다. 또한 이전에는 없었던 Control hazard와 Data hazard를 어떻게 detect하고 어디에 얼마만큼의 bubble을 넣을 것인지, 더 나은 동작을 위하여 BTB와 forwarding, 2-bit saturation counter를 구현할 것인지에 대한 고민이 추가적으로 필요하였다. 이전에 수행한 과제로부터의 경험이 도움이 되었지만 굉장히 많은 수의 변수들과 이를 연결하는 것들이 보다 복잡해지면서 어려움을 겪었다.

6. Conclusion

Single-Cycle CPU와 Multi-Cycle의 단점을 모두 보완한 보다 복잡한 Pipeline-CPU를 구현하며 실제로 많이 사용되는 CPU의 기초적인 구현 방식을 더 명확하게 이해하게 되었다. Lab3~5를 거치며 가장 간단한 형태의 CPU부터 출발하여 현대 사용되는 CPU에 가장 가까운 모습까지 구현을 거듭하며 각각의 차이와 장단점, Stage를 세분화해서 동시에 실행하며 생기는 여러 hazard를 해결하는 방법까지 더 명확하게 이해하게 되었다.