컴퓨터구조실험

과제 이름: Assignment #3

담당교수: 이성원

학 과: 컴퓨터정보공학부

학 번: 2019202021

이 름: 정 성 엽

제출일: 2023/5/24

1. 실험 내용

A. 프로젝트 이론

- **Hazard란**

  pipeline architecture CPU는 INST들을 동시에 수행한므로 한 cycle 당 instruction의 수행속도를 보다 늘린다. 하지만 동시에 수행하는 instruction은 서로 의존성을 가지거나, 구조상의 문제, 분기점에서의 처리 문제로 Hazard가 발생할 수도 있다. hazard 종류는 아래와 같다

  1. **Structural hazard**

     pipeline architecture의 구조적 한계로 인해 발생하는 hazard이다. 이 hazard 같은 경우 같은 cycle 실행되는 동안 instruction이 모듈에 동시에 접근하여 사용할 때 발생한다.

  2. **Data hazard**

     한번에 여러 instruction를 수행하는 pipeline의 특성상 instruction이 의존성이 존재할 수 있기 때문에 발생하는 hazard이다.

     A. RAW – Read After Write

        한 instruction이 Register file에 값을 write 해야 하고 이 동작이 마치지 못했을 때, 다음 instruction이 Read를 시도한다면 발생한다.

     B. WAW – Write After Write

        한 instruction이 Register file에 값을 write 해야 하고 이 동작이 마치지 못했을 때, 다음 instruction이 write를 시도한다면 발생한다. 만약 memory access하는 instruction에서 위와 같은 경우가 발생하면 잘못된 값을 write하게 된다.

     C. WAR – Write After Read

        한 instruction이 register file의 값을 read 해야 하고 이 동작을 마치지 못했을 때, 다음 instruction이 write하여 사용하려고 할 때 발생한다.

  3. **Control hazard**

     Branch 또는 Jump와 같이 PC값이 바뀌는 명령어에서, pipeline에 들어있는 다음 명령어는 이전 pc를 기준으로 fetch한다. 따라서 바뀐 pc값에 저장되어 있는 명령어를 다시 수행하기 위해 stall한다.

- **Harard 회피 방법**

  Hazard를 회피하는 방법은 Hazard를 감지하는 HW unit을 추가하여 방지하거나, 코드 스케쥴링 같은 SW 방법을 사용하기도 한다. 예를 들어 stall(bubble = nop inst)를 추가하여 모든 hazard를 회피할 수 있지만, 그 만큼 성능에 저하가 발생할 수 있기 때문에 최후의 수단으로 사용한다. 아래는 각 hazard 별 회피 방법이다. 이는 H/W로와 S/W로 회피 방법이다.

  1. **Structural Hazard**

     H/W: memory에 접근하는 instruction mem과 data mem으로 분리하여 동

일한 cycle 내에서 같은 모듈에 접근하는 것을 막는다. 만약 모던 프로세서라면 메모리 컨트롤러에서 해당 hazard를 처리한다.

S/W: 같은 모듈에 접근하는 것을 확인하여 stall한다. 다만 stall을 위해 bubble(NOP) 연산을 삽입하므로 CPU 성능이 저하될 수 있다.

2. **Data Hazard**

H/W: 같은 register에 접근하는 것을 감지하는 H/W를 추가하는 방법이 있다. 이는 다음 instruction에 forwarding 해주는 unit 또한 추가한다.

S/W : WAW 또는 WAR의 경우 다른 Reg를 임시저장소로 사용하거나 코드 컴파일러 차원에서 S/W적으로 해결 가능하지만 RAW는 nop하거나 forwarding unit을 통해서만 해결 가능하여 성능이 저하될 수 있다.

3. **Control Hazard**

H/W: Control Hazard를 피하는 방법은 branch와 jump의 여부를 완벽히 예측해야 한다. 그렇기 때문에 완벽하게 회피하는 방법은 존재하지 않는다. 다만 branch는 어느 특정 규칙을 파악하여 예측하는 unit을 사용하면 모던 프로세서 어느 정도 예측이 가능하다.

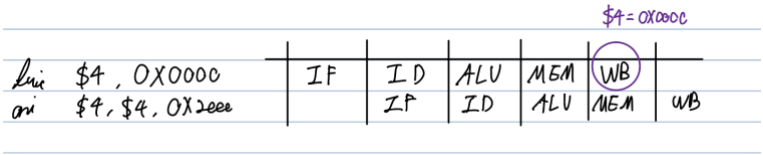
S/W: nop을 통해 방지하여 CPU 성능이 저하될 수 있다.

2. 검증 전략, 분석 및 결과

A. 어셈블리 코드 설명

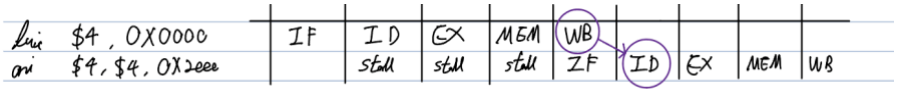
- main

처음 main의 lui 명령어와 ori 명령어에서 lui는 $4에 data를 write하려고 하며 ori는 $4의 값을 read한 후 write 하려고 한다. 이 때 data hazard가 발생한다.



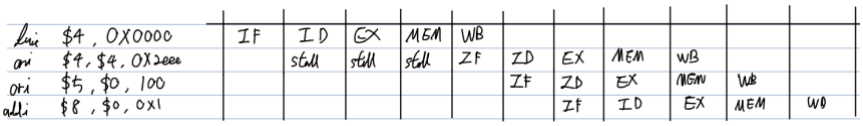
nop가 없다면 $4에 0x0000이 쓰이기 전에 아래 ori에서 ALU 연산을 진행하게 되어 data hazard가 발생한다. 이를 해결하기 위해 nop로 미루거나 forwarding 기법을 사용한다.

1. nop만 사용

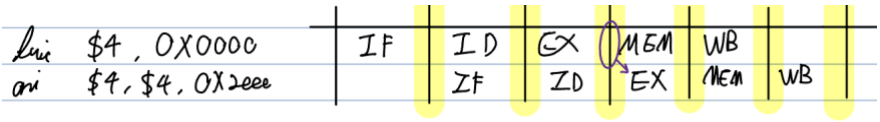


한 번만 nop한 경우 WB과 EX가 겹쳐서 불가하고 두 번만 nop 한경우 WB과 ID가 겹처서 불가하다. 그래서 nop를 3번하여 진행한다.

main 나머지를 구현하면 아래와 같다.

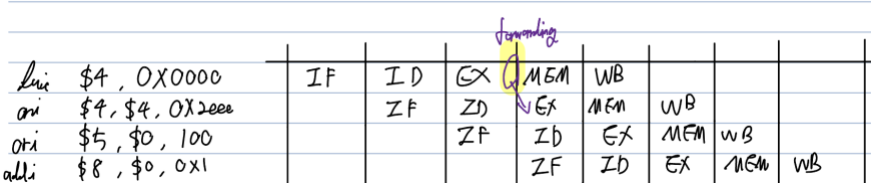


2. forwarding 사용



forwarding을 사용하여 lui의 EX stage의 연산 값을 ori EX stage에서 사용한다. M_TEXT_FWD에서 0번째에 01_00을 추가한다.

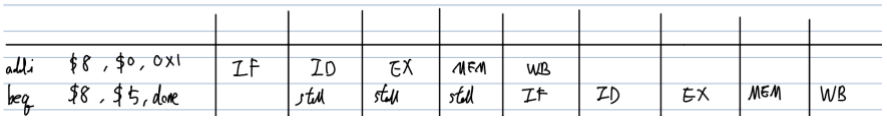
main 나머지를 구현하면 아래와 같다.



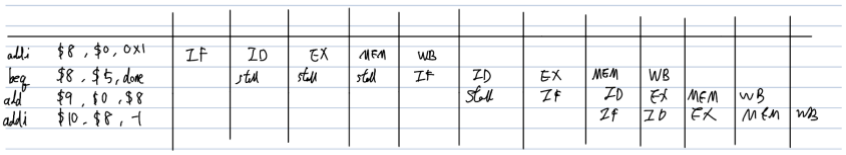
- L1

main의 addi와 L1의 beq가 $8에 대해 의존성을 가진다. 그리고 forwarding에서는 $5에 대해 의존성을 가질 수도 있다.

1. nop만 사용



이전과 비슷하게 nop을 3번 사용하였다. 나머지 L1은 아래와 같다.



beq 다음에 nop를 한번 실행한다.

2. forwarding 사용

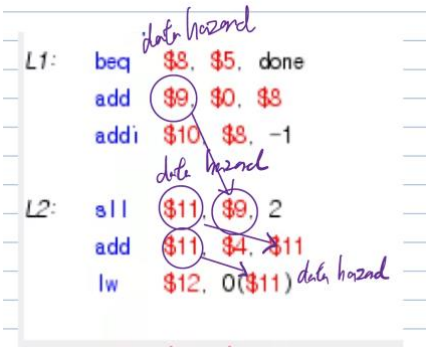
$5또한 의존하기에 nop을 한번 진행하여 $5는 WB된 이후에, $8은 forwarding을 통해 가져와 연산한다. 하지만 이때도 beq의 ID 단계와 ori의 WB 단계가 겹치므로 한 번 더 nop를 하며 addi와도 WB 단계와 겹치므로 stall은 총 3번 쓰인다. 하지만 원칙을 다시 살펴보면 branch 명령에 대해서는 forwarding이 불가하도록 되어있기에 원리만 이해하도록 한다.

나머지 L1은 아래와 같다.

- L2

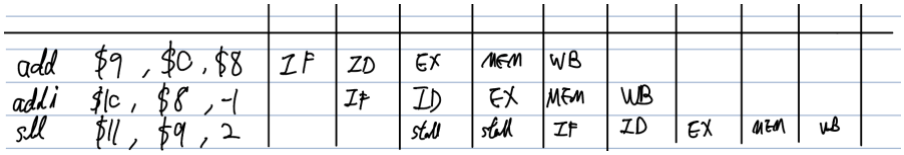
```
L2:   sll   $11, $9, 2
      add   $11, $4, $11
      lw    $12, 0($11)

      sll   $13, $10, 2
      add   $13, $4, $13
      lw    $14, 0($13)

      slt   $1, $12, $14
      beq   $1, $0, L3

      sw    $12, 0($13)
      sw    $14, 0($11)

      addi  $9, $9, -1
      addi  $10, $9, -1
      bgez  $10, L2
```

sll부터 $9의 의존성을 확인해야 한다.

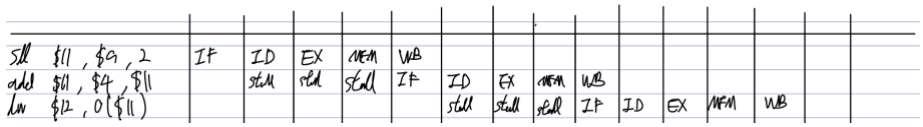


위에서 sll의 $9에 대해 data hazard가 발생할 가능성이 있다고 판단하였다.

1. nop 만 사용했을 때

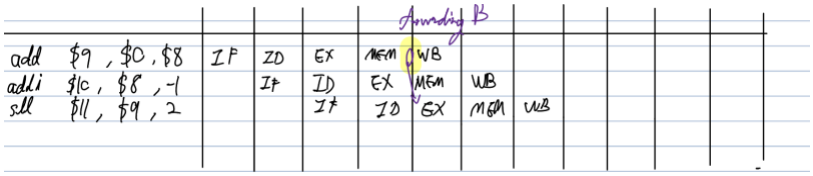


sll은 nop를 2번을 걸어서 data hazard를 회피한다.

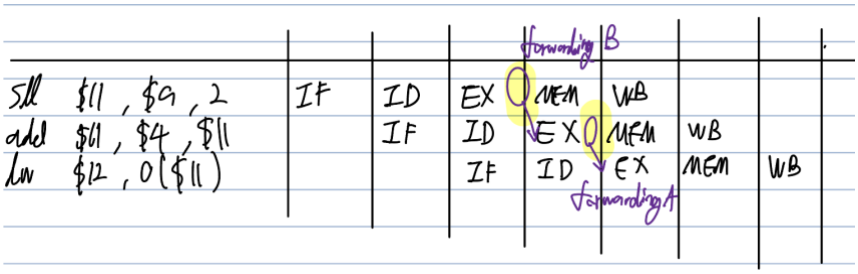


각 명령어는 $11에 data dependency가 있으므로 nop 3번 후 명령어를 실행한다.

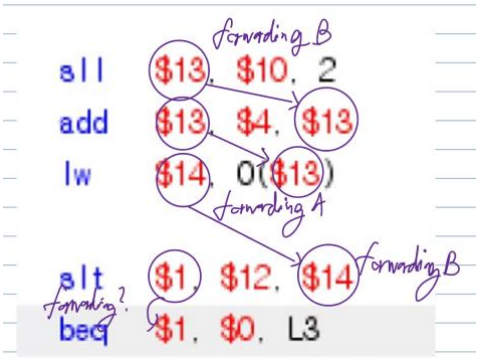
2. forwarding 사용했을 때



nop만 사용했을 때와 다르게 WB으로 가는 데이터를 받아서 ALU 연산을 한다.

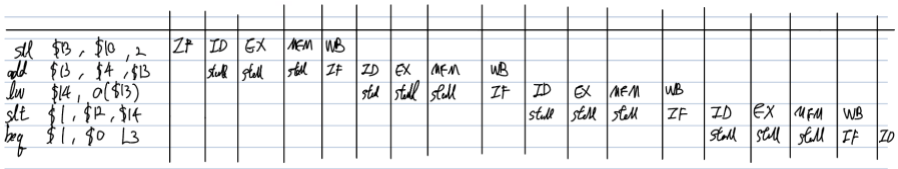


sll의 ALU 연산 결과를 add ALU 연산의 input_B로 가져와 연산하고 add ALU 연산 결과를 lw ALU 연산의 input_B로 가져와 연산한다.



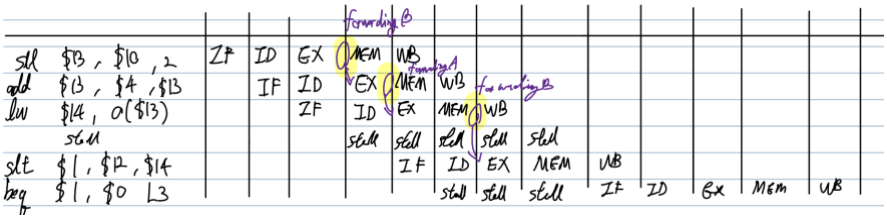
해당 어셈블리 코드에서는 여러 register가 data hazard인 것으로 보이나 특이점으로 $1은 data hazard가 발생하지 않도록 하되 forwarding은 사용 불가하다.

3. nop만 사용하는 경우



각 명령어가 nop 3개 이후 실행되도록 해야 data hazard가 발생하지 않고 control hazard 또한 발생하지 않는다.

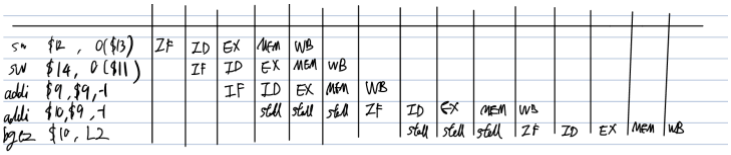
4. forwarding 사용하는 경우



sll과 add에서 forwarding을 진행하며 lw가 $14를 다루지만 forwarding을 하지 않는 이유는 Mem 단계를 지나야 $14에 유의미한 값이 저장되기 때문이다. 그래서 nop을 한번 진행시켜 해당 자리에 forwarding을 WB값으로 두어 전달한다.

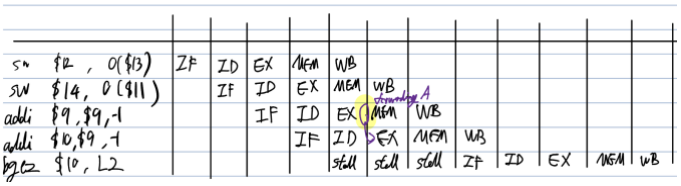
addi 명령어에서 $9에 대해 data hazard가 발생할 수 있음을 확인하였다.

5. nop만 사용하는 경우



nop만 사용할 때 addi의 $9가 data hazard가 있어서 nop 3번 호출하였고, branch에 대한 호출에도 nop를 3번 사용하였다.

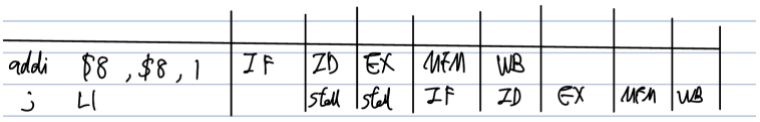
6. forwarding 사용하는 경우



forwarding의 경우 addi의 $9와 -1의 ALU연산을 forwarding 하여 다음 addi의 ALU연산에 가져온다.

- L3

1. nop만 사용할 때와 forwarding 사용할 때



jump에서 따로 EX로 비교하거나 연산할 것이 없으므로 nop를 2번 사용한다.

이를 바탕으로 nop만 사용하는 경우 구현한 assembly code는 아래와 같다.

insertion_sort.asm

```
1   .text
2   main: lui   $4, 0x0000
3         nop
4         nop
5         nop
6         ori   $4, $4, 0x2000
7         ori   $5, $0, 0x10
8         addi  $8, $0, 0x1
9         nop
10        nop
11        nop
12
13  L1:   beq   $8, $5, done
14        nop
15        add   $9, $0, $8
16        addi  $10, $8, -1
17        nop
18        nop
19
20  L2:   sll   $11, $9, 2
21        nop
22        nop
23        nop
24        add   $11, $4, $11
25        nop
26        nop
27        nop
28        lw    $12, 0($11)
29        sll   $13, $10, 2
30        nop
31        nop
```

```
32      nop
33      add   $13, $4, $13
34      nop
35      nop
36      nop
37      lw    $14, 0($13)
38      nop
39      nop
40      nop
41      slt   $1, $12, $14
42      nop
43      nop
44      nop
45      beq   $1, $0, L3
46      nop
47      sw    $12, 0($13)
48      sw    $14, 0($11)
49      addi  $9, $9, -1
50      nop
51      nop
52      nop
53      addi  $10, $9, -1
54      nop
55      nop
56      nop
57      bgez  $10, L2
58      nop
59
60  L3: addi  $8, $8, 1
61      nop
```

```
61      nop
62      j L1
63      nop
64
65  done: break
66
67  .data
68  LC0:
69          .word   31028
70          .word   16610
71          .word   12937
72          .word   7525
73          .word   25005
74          .word   17956
75          .word   23964
76          .word   13951
77          .word   3084
78          .word   23696
79          .word   3881
80          .word   11872
81          .word   24903
82          .word   16843
83          .word   25957
84          .word   25086
```

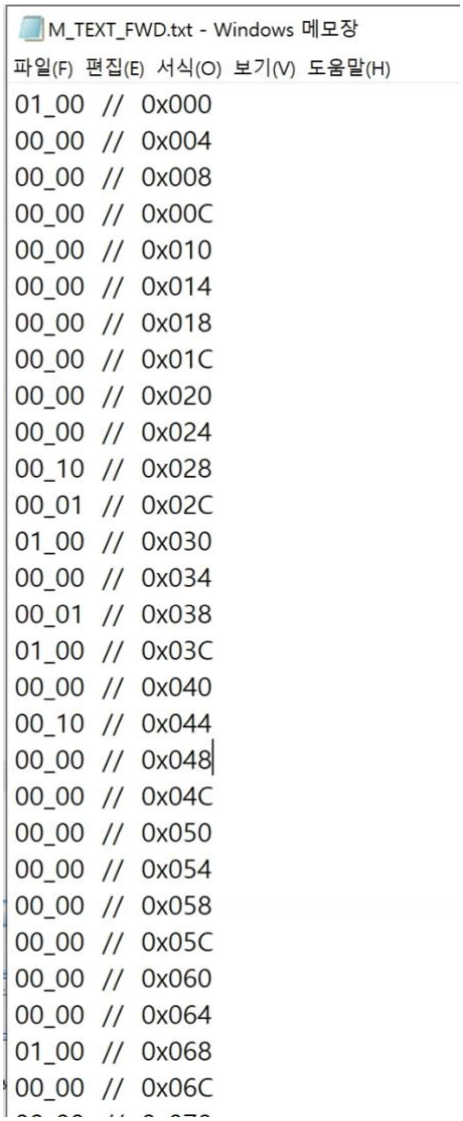
nop는 총 37개 사용되었다.

또한 위를 바탕으로 forwarding을 사용하여 구현한 코드는 다음과 같다.

```
.text
main: lui   $4, 0x0000
      ori   $4, $4, 0x2000
      ori   $5, $0, 0x10
      addi  $8, $0, 0x1
      nop
      nop
      nop

L1:   beq   $8, $5, done
      nop
      add   $9, $0, $8
      addi  $10, $8, -1

L2:   sll   $11, $9, 2
      add   $11, $4, $11
      lw    $12, 0($11)
      sll   $13, $10, 2
      add   $13, $4, $13
      lw    $14, 0($13)
      nop
      slt   $1, $12, $14
      nop
      nop
      nop
      beq   $1, $0, L3
      nop
      sw    $12, 0($13)
      sw    $14, 0($11)
      addi  $9, $9, -1
      addi  $10, $9, -1
      nop
      nop
      nop
      bgez  $10, L2
      nop

L3:   addi  $8, $8, 1
      nop
      j L1
      nop

done: break

.data
L00:
      .word   31028
      .word   16610
      .word   12937
      .word   7525
      .word   25005
      .word   17956
      .word   23964
      .word   13951
      .word   3084
      .word   23696
      .word   3881
      .word   11872
      .word   24903
      .word   16843
      .word   25957
      .word   25086
```

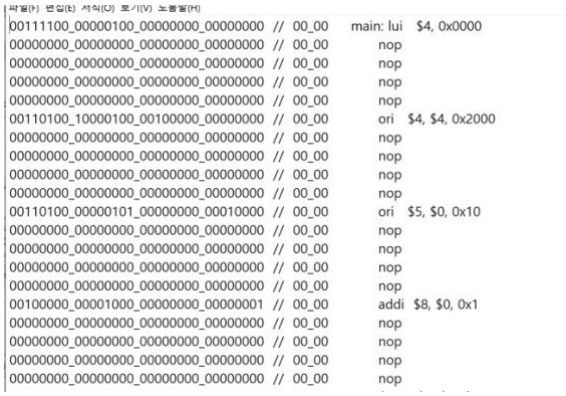
nop는 총 15개 사용하였다.

또한 이때 M_TEXT_FWD.txt의 세팅값은 아래 사진과 같다.



B. 명령 수행에 걸린 총 cycle 수

- 기존 명령어 당 nop 4개인 경우



사진과 같이 명령어 당 4개의 nop을 실행하는 되는 경우 cycle 수는 아래와 같다.

```
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------------
 Break signal: 1,    # of Cycles:        4185
-----------------------------------------------
```

총 4185번의 cycle이 실행되었다.

- 변경 후
  1. nop만 사용할 때

```
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------------
 Break signal: 1,    # of Cycles:        2608
-----------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 26195(
```

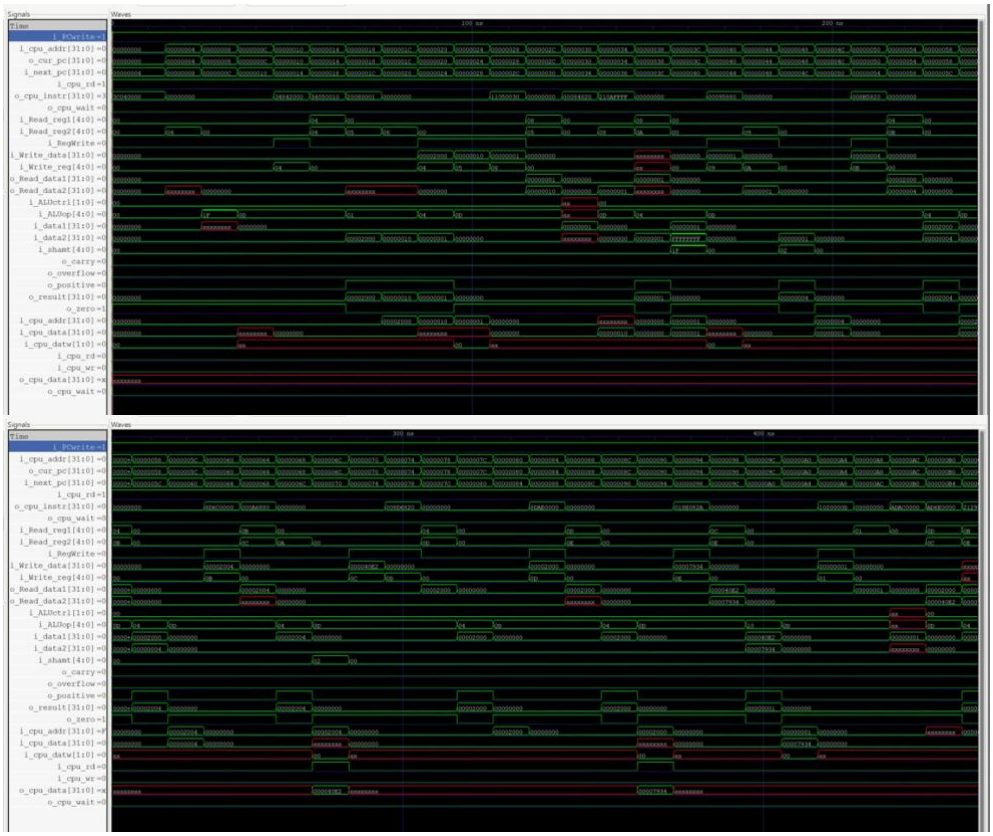
총 2608 번의 cycle이 실행되었다.

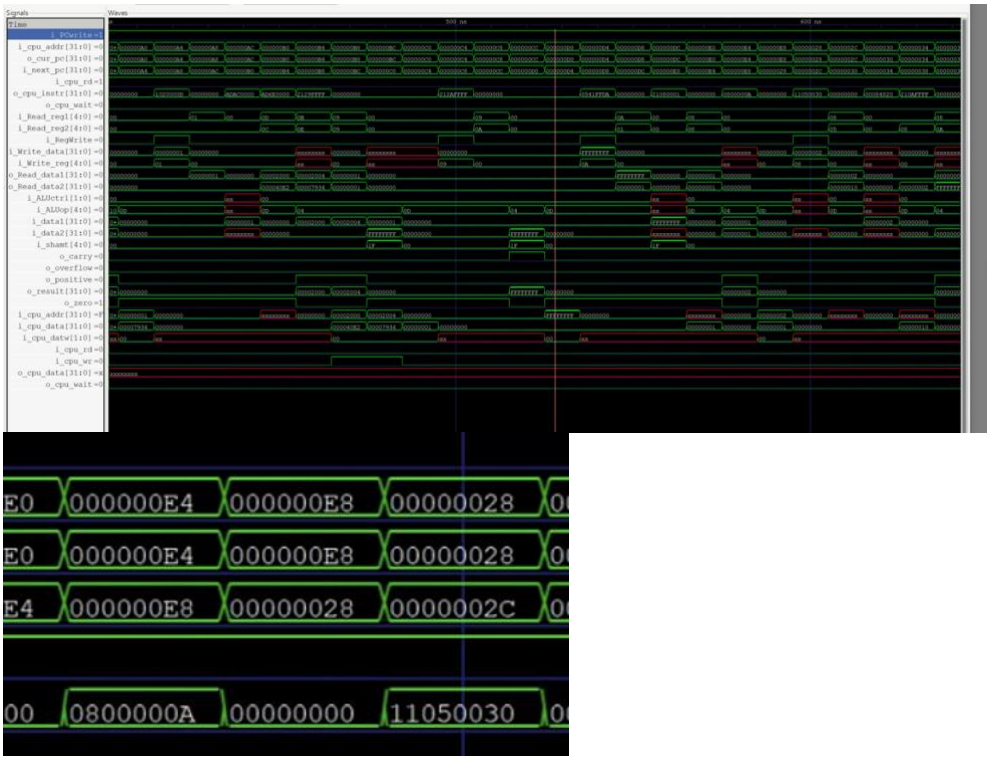
  2. forwarding 사용할 때

```
-----------------------------------------------
| H020-3-1647-01: Computer Architecture      |
|                 CE.KW.AC.KR                |
-----------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------------
 Break signal: 1,    # of Cycles:        1486
-----------------------------------------------
```

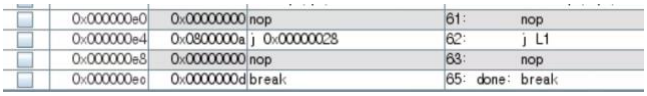
총 1486 번의 cycle이 실행되었고 앞서 nop을 각 명령어 당 4번 했을 때와 nop만을 사용하여 줄였을 때보다 훨씬 많은 cycle이 줄어들었음을 확인할 수 있었다.

C. 기존 어셈블리코드에서 nop를 제거한 시뮬레이션

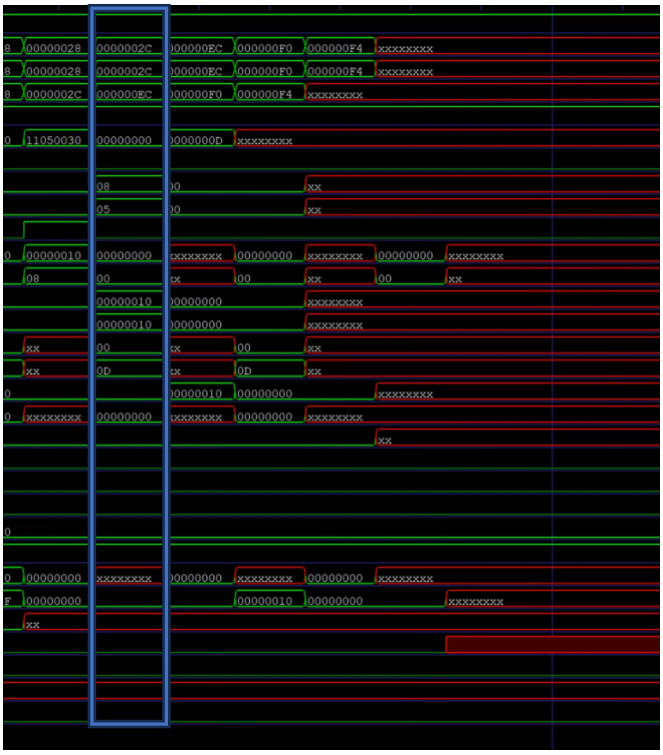




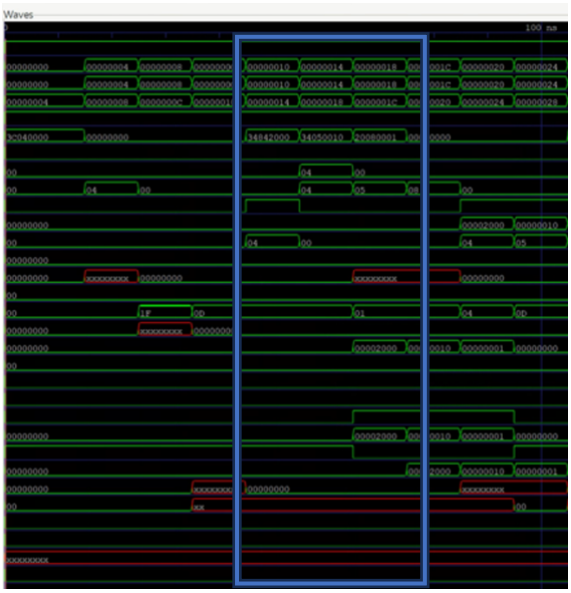
마지막 사진에서 해당 부분을 보면 0x0800000A의 명령어의 EX 단계에서 PC값이 바뀐 것을 알 수 있다.



해당 명령어는 j L1으로 PC가 0x28인 L1으로 jump 하였음을 확인할 수 있다. 또한 이것들이 반복되고 L1에서 $8과 $5가 같을 때 Done으로 이동하여 break 됨을 아래 파형을 통해 확인할 수 있다.

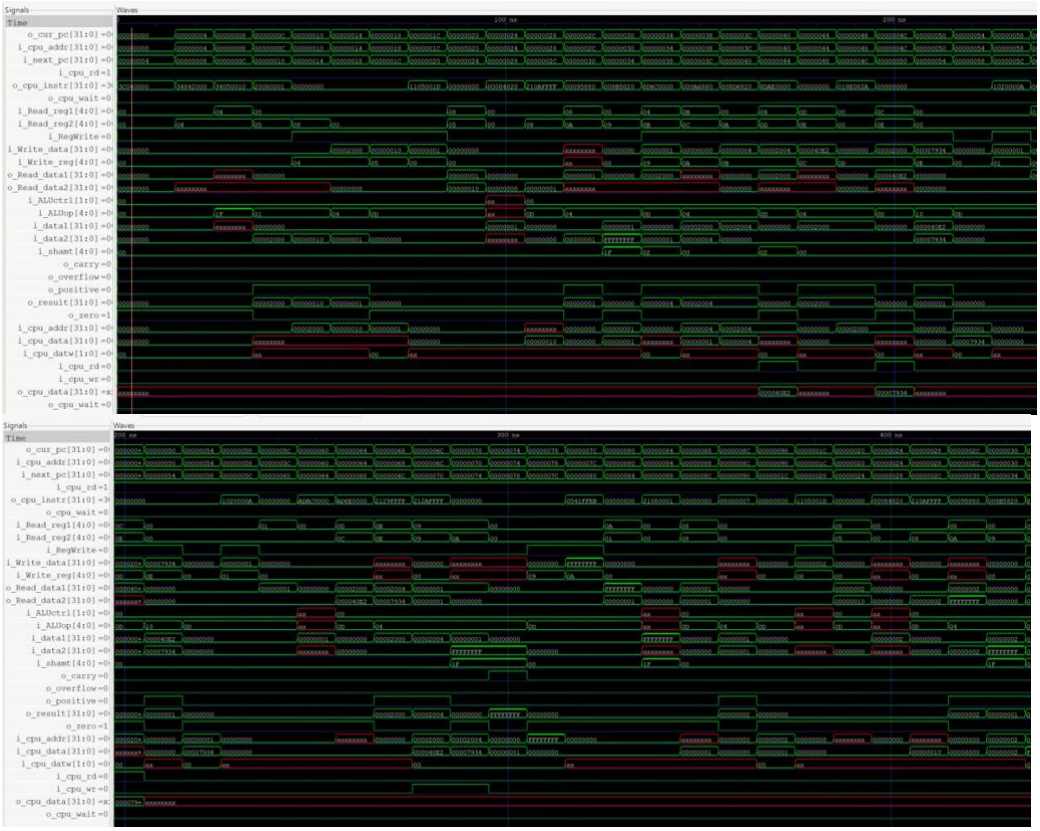


$5와 $8이 0x10으로 같아졌고 이 때 done으로 이동하여 break 되었다.



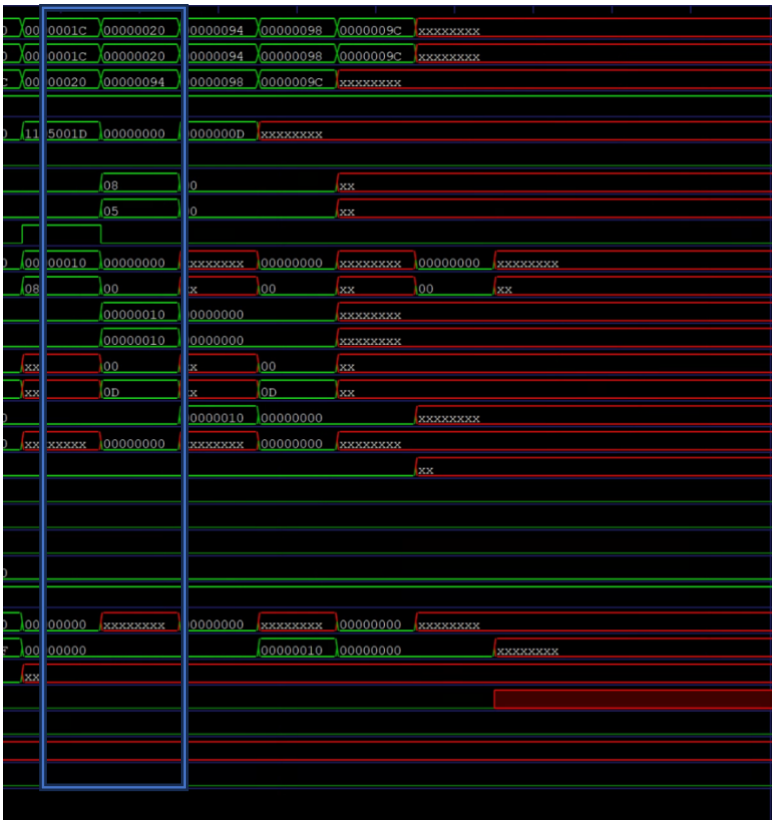
또한 nop의 개수를 줄이는 과정에서 본 과제에서 원했던 pipeline 형태를 만들 수 있었다. 한 사이클 당 하나의 명령어가 시작되어 각각의 stage를 거쳐 명령이 완료되는 형태를 보이고 있다.

D. 기존 어셈블리코드에서 forward 제어신호를 추가하여 더 많은 nop를 제거하여 재구성된 어셈블리 코드 시뮬레이션



이전의 nop만 사용하여 줄였을 때 보다 촘촘하게 명령어들이 cycle 마다 fetch되어 실행됨을 확인 할 수 있다. 이 때 시뮬레이션 또한 j L1으로 PC가 0x1C인 L1으로 jump 하였음을 확인할 수 있다. 또한 이것들이 반복되고 L1에서 $8과 $5가 같을

때 Done으로 이동하여 break 됨을 아래 파형을 통해 확인할 수 있다.



3. 문제점 및 고찰

이번 Project3를 통해 pipeline structure에서의 명령어 사용을 익히고 동작에 대한 이해를 더욱 할 수 있었다. 처음에는 기존 멀티사이클처럼 nop을 4번 두어 5사이클마다 명령어 하나를 실행하도록 했으며, 다음에는 각 stage가 간섭하지 않는 만큼의 nop을 제거하여 총 cycle 수를 줄였다. 또한 forwarding을 사용하여 ALU의 결과 또는 WB에 들어갈 값들을 미리 다음 명령어에 호출하여 각 stage 간섭을 줄이고, nop의 사용을 줄여 총 Cycle수를 더욱 줄일 수 있었다.

처음 쓰는 프로그램 Mars에서 어째서인지 처음에 setting 했던 memory configuration의 Compact, Text at Address 0의 체크가 풀려 매번 j의 주소가 달라지면서 코드가 달라져 계속 다르다고 나왔다. 이에 작성한 코드가 잘 못된 줄 알고 다시 시도했을 때 실패하자, 처음으로 돌아가 nop을 각 4번씩 사용했을 때도 문제가 생겨, 이것은 프로그램의 오류라 생각해 다시 설치하면서 setting을 고쳤다. 다행히 처음 구상한 코드와 예상하는 정답이 거의 일치하여 빠르게 해결할 수 있었다.