Project Assignment#3

<Pipeline Architecture>

컴퓨터구조실험

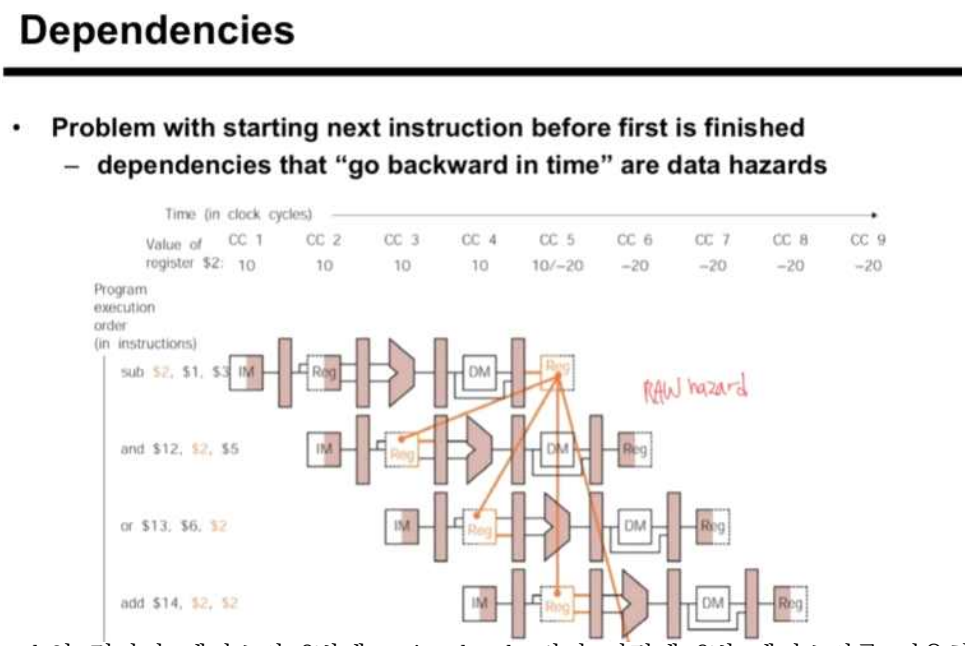
학번: 2021202045
이름: 김예은
담당 교수님: 이성원 교수님
분반: 컴구실 (수) 분반

Ⅰ.실험 내용

1. Hazard 및 Hazard 피하는 방법

1) Structural Hazard: 자원은 하나인데, 여러 명령이 동시에 수행되려고 할 때 발생한다. 예를 들어, memory가 하나인데, pipeline내에서 한 명령어는 instruction fetch를 위해 사용하고, 다른 명령어는 동시에, data fetch를 위해 memory에 접근하면 hazard가 생긴다. 해당 hazard를 피하기 위해 hardware resoucre를 추가하거나, 명령어 스케줄링을 통해 작업의 실행 순서를 조정하여 구조적 위험을 피할 수 있다.

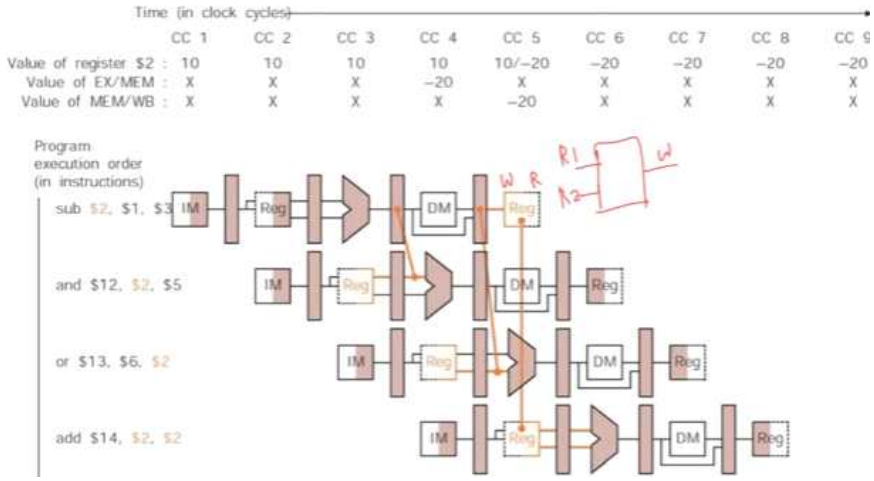
2) Data Hazard: 아직 pipeline 명령어가 끝나지 않은 register에 접근하여 명령어를 실행하고자 할 때 발생한다. 예를 들어 다음과 같은 경우가 있다.



sub의 결과가 레지스터 2번에 write back 되기 이전에 2번 레지스터를 이용하여 and, or, add 연산을 진행한다. 이때, 레지스터 2번에 data hazard가 발생한다. data hazard는 어떻게 피할 수 있을까? 앞의 명령어가 데이터를 write back 할 때까지, stall 하여 기다린다. stall을 할 시 해당 cycle에 명령어 수행을 할 수 없으므로 cycle 손해가 일어난다. 또는 forwarding 방식을 이용한다. forwarding 형식은 데이터값이 준비되는 대로 미리 값을 전달해주는 방식이다. 다음과 같다.

Forwarding

- **Use temporary results, don't wait for them to be written**
  - **register file forwarding to handle read/write to same register**
  - **ALU forwarding**

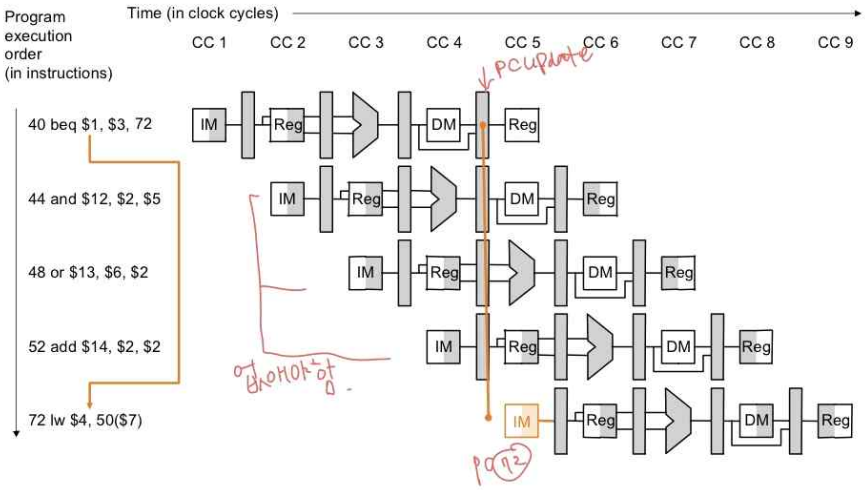


sub에서 execute 단계에서 ALU을 통해 레지스터 2번에 적을 값이 미리 나오기 때문에 해당 단계에서 레지스터 2번에 쓰일 ALU 결과 값을 미리 가져와 다음 명령어에 전달한다. software solution으로는, compiler에 nop를 적어 data hazard를 막는 방법이 있다. software solution의 경우 위의 sub과 and 사이에 nop를 3개 적어주어 고쳐준다.

3) Control Hazard: beq 또는 j 명령어 등 분기 명령어가 있을 때는 pc값이 변해버리기 때문에 바로 다음 cycle에 그 뒤의 명령어를 실행할 시 그 명령어는 쓸모가 없어진다. beq의 경우 보통 중간에 3개의 명령어가 쓸모 없어진다.(mem단계에서 pc를 update하기 때문이다.)

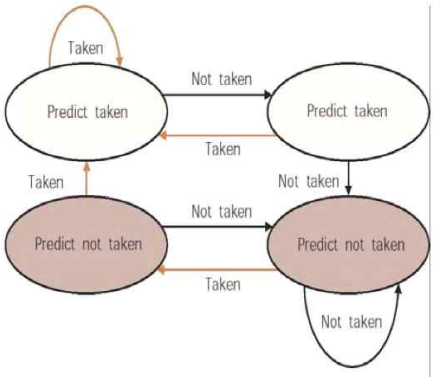
Branch Hazards

- **When we decide to branch, other instructions are in the pipeline!**



control hazard의 경우 stall을 통해 문제 해결까지 일시 중지를 한다. 이번 과제에서 분기 명령어는 decoding단계에서 결정되므로 stall 1개만으로 control hazard를 피할 수 있다. branch가 taken일지 not taken일지 예측하는 방법도 있다. 예측기는 여러 가지 종류가 있는데, 다음은 2-bit prediction schme이다.

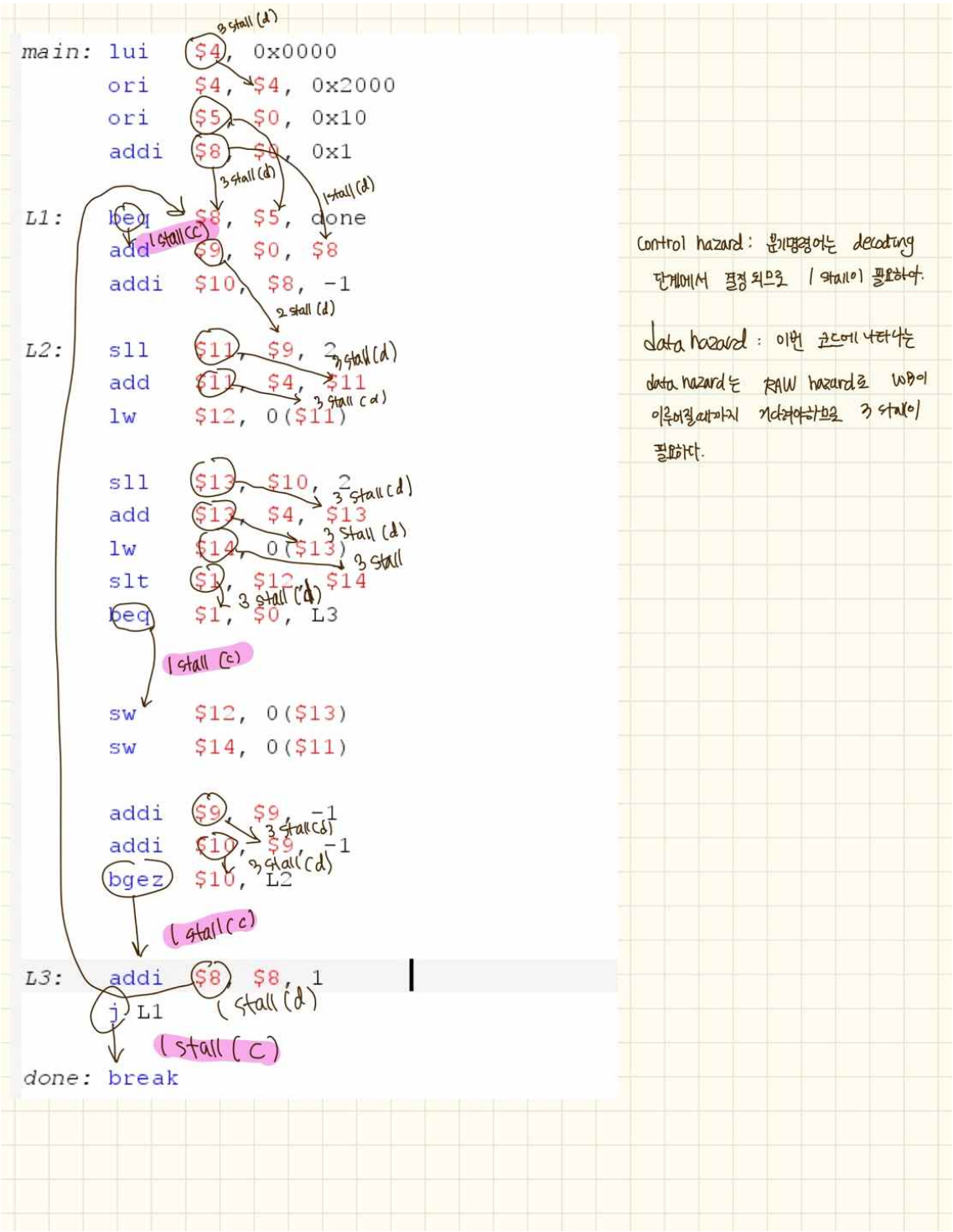
Solution: dynamic branch prediction



A 2-bit prediction scheme

Ⅱ. 검증 전략, 분석 및 결과

1) 초기 상태(no nop)의 assembly code의 dependency 분석



control hazard의 경우 분홍색으로 칠하여 구분하였고, 각각 data hazrd이면, 필요한 stall 수+(d), control hazard이면, 필요한 stall 수+(c)를 적었다.

위에 적었듯이 위의 코드에서 발생하는 data hazard는 RAW data hazard이다.

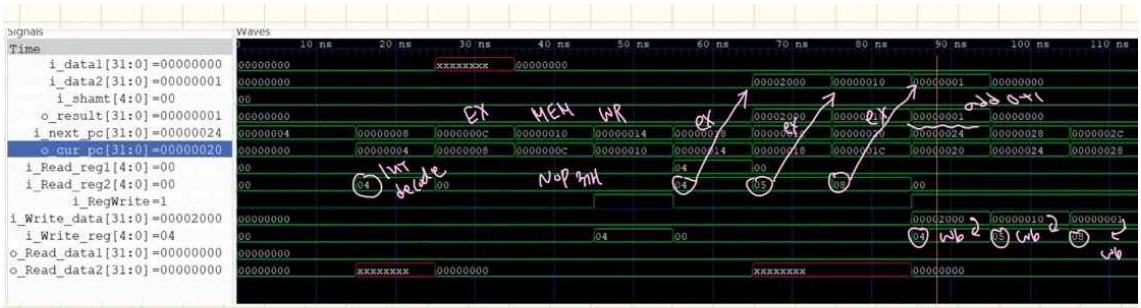
2) 기존 assembly code에서 최대한 nop를 제거(without rescheduling)

-구현한 assembly code

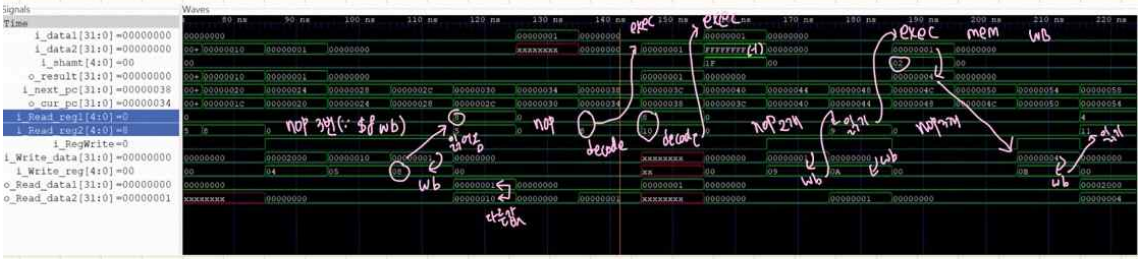
```
main: lui   $4, 0x0000
      nop
      nop
      nop
      ori   $4, $4, 0x2000
      ori   $5, $0, 0x10
      addi  $8, $0, 0x1
      nop
      nop
      nop

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

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

      sll   $13, $10, 2
      nop
      nop
      nop
      add   $13, $4, $13
      nop
      nop
      nop
      lw    $14, 0($13)
      nop
      nop
      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
      nop
      nop
      nop
      addi  $10, $9, -1
      nop
      nop
      nop
      bgez  $10, L2
      nop

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


done: break
```

위(1번 활동)에서 control hazard, data hazard를 파악 후, 필요한 stall 수를 파악한 대로 nop를 넣어줬다. 분기 명령어 뒤에는 분기 판단을 decode 단계에서 해주므로 바로 다음 명령어가 fetch단계에 오지 못하도록 무조건 1개의 nop를 넣어주고, write back 값이 다음 명령어에 사용될 때는 다음 명령어 시행 전 write back단계 까지 갈 수 있도록 기다려야하기 때문에, nop를 3개 넣어줬다. 위는 명령어의 재배치 없이 nop를 최대한 적게 써서 최적화한 코드이며, 시뮬레이션 돌려보면 다음과 같다.

-시뮬레이션 결과 분석



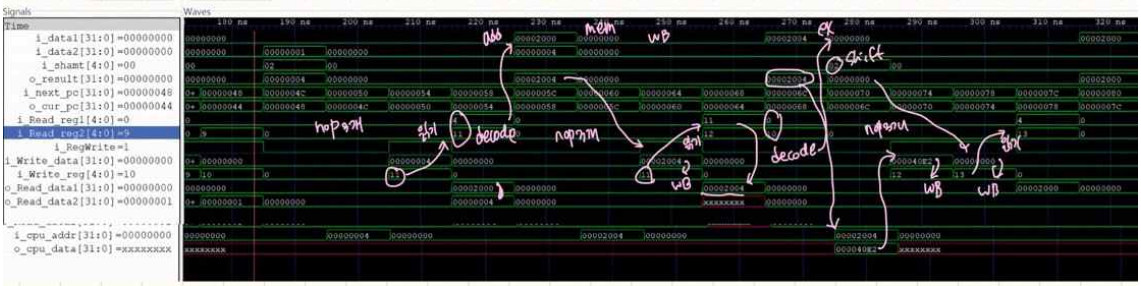
→ lwi $4 0x0000 → nop x3 → ori $4 $4 0x2000 할때 $4에 0x0000이 잘 들어가있는 상태에서 decoding 단계에 들어가서 data hazard를 피함



→ beq에 쓰이는 $8이 이전 명령어에서 WB까지 할수있도록 nop 3개를 넣어 지연함으로써 data hazard를 피함.

→ addi와 sll 사이에 명령어1개+nop 2개로 data hazard를 피함

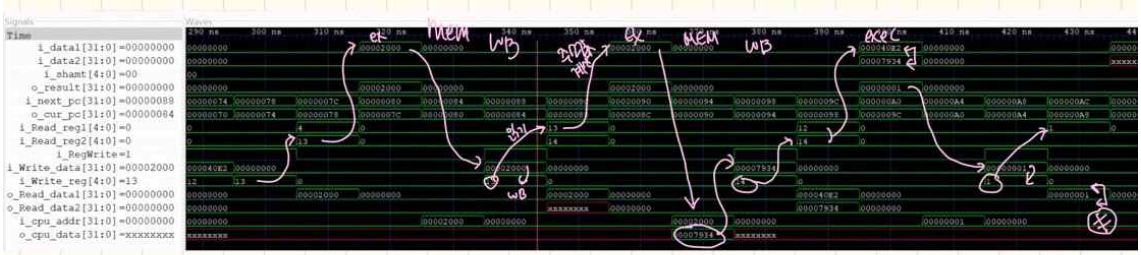
→ sll 명령어를 통해 $9 x 4 값이 $11에 적히고 (nop 3개) $11에서야 읽기가 실행됨.
(안전하게) ↳ 값이 sll결과값으로 덮힘



→ $11 와 $4를 더해 $11에 저장하고 nop 3번이후에 저장된 $11값을 읽어 lw을 위한 주소값 계산에 사용.

→ lw 초후의 명령어와 동시에 작동, lw 뒤는 독립적인 명령어라 nop없이 pipeline 해놈.

→ sll 결과값으로 0이 나온것은 $9에 저장하기까지 기다렸다가 주기를 읽음



→ lw의 결과값을 다음 명령어에 사용해야함으로 nop 2개 사용했더니 data hazard를 피할 수 있었음

→ $12과 $14(lw의 결과값) 비교했는데 $12 < $14이므로 참이어 1로 set

→ nop 3개를 통해 $1에 wb을 기다려서 $1과 $0을 비교 : 값이 다름

→ beq의 조건이 만족으로 PC = PC+4

→ sw, sw, addi의 명령어 연속적으로 들어옴으로 nop X / mem단계에서 sw가 잘 들어감

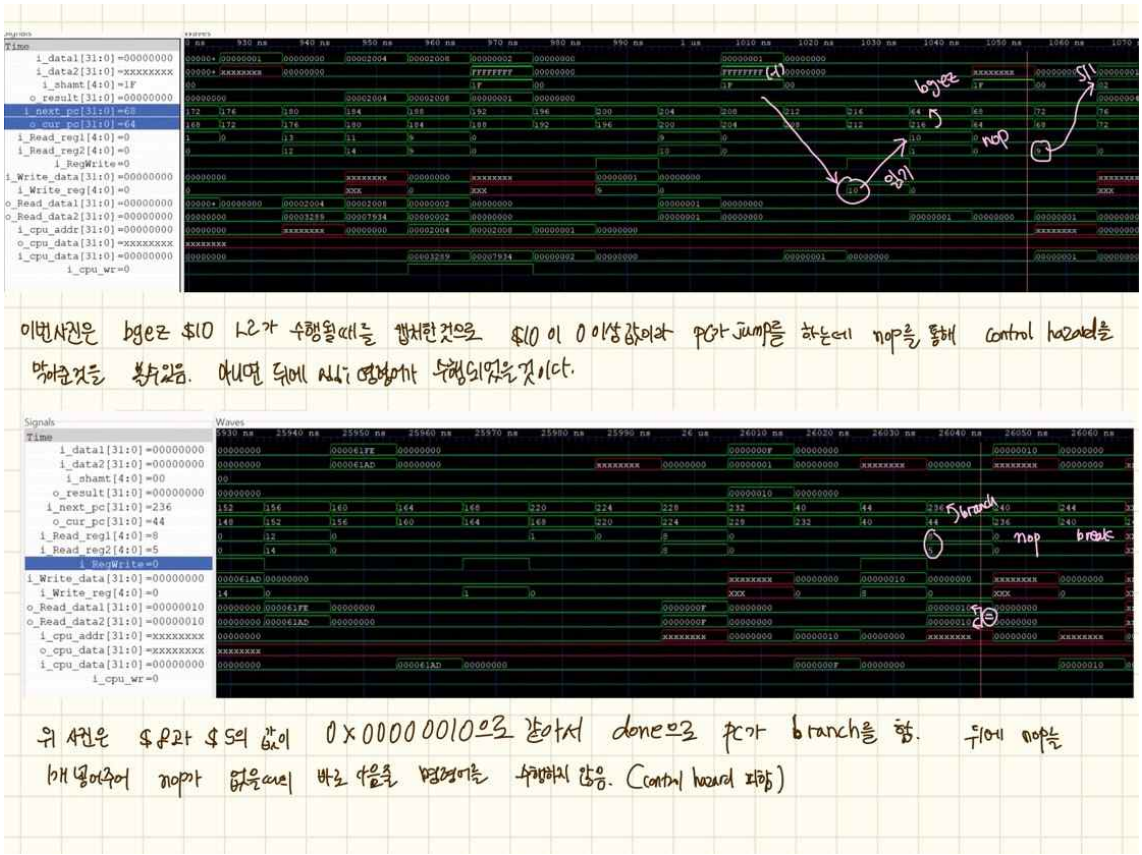
→ addi 뒤의 addi의 경우 data hazard가 일어남으로 nop 3개를 넣어 $9가 갱신이된 이후 그값을 읽게함.

→ bgez $10 안에 $10값이 -1로 0 이상이어서 PC+4 이다. control hazard를 위해 nop 2개를 넣음

→ addi $8, $8, 1을 해주고, 시뮬가면 바로 $8을 피연산으로 nop을 통해 WB까지 할 수 있게함

→ 분기명령어 뒤에 nop을 넣어줌으로써 break가 바로 수행 되는걸 볼 수 있음.

→ 내의 $8이 쓰이는데 J 니전에 $8이 WB된 후 add가 된 값을 참조할 수 있음.



맨 처음 처음부터 끝까지 data 및 control hazard 없이 data가 잘 쓰이고, 읽히는 것을 확인한 뒤로(그 뒤부터는 같은 flow를 단순 반복이므로 분석 생략), branch가 일어날 때와 마지막에 break를 만날 때를 캡처하여 분석하였다. 위에서 적었다시피, branch 명령어 뒤에는 nop를 하나 넣어, 다음 명령어가 분기 판단 전에 fetch가 되지 않도록 한다.

- 명령 수행에 걸린 총 cycle 수

```
C:₩Users₩USER₩Desktop₩컴구실자료₩컴구프젝3₩prj3_PCPU_2023₩prj3_PCPU_2023>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the file for the requested range [0:1023]
-------------------------------------------
| HO20-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:        2608
-------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 26195000 (1ps)

C:₩Users₩USER₩Desktop₩컴구실자료₩컴구프젝3₩prj3_PCPU_2023₩prj3_PCPU_2023>FC /L mem_dump_IS.txt mem_dump.txt
파일을 비교합니다: mem_dump_IS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩USER₩Desktop₩컴구실자료₩컴구프젝3₩prj3_PCPU_2023₩prj3_PCPU_2023>FC /L reg_dump_IS.txt reg_dump.txt
파일을 비교합니다: reg_dump_IS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

forwarding없이 nop만 제거하였을 때 2608 cycle이 나온다.

3) 기존 어셈블리코드에서 Forward 제어신호를 추가한 경우

-Forward 제어신호를 추가하여 재구성한 assembly code

```
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
```

이번 프로젝트에서 forwarding은 ALU결과값을 다음 명령어의 ALU input(A/B)로 forwarding해주거나, MEM의 access result를 다음 명령어의 ALU input(A/B)로 forwarding 해주는 선택지 2개 중 하나를 선택하여 적용한다. 이번 프로젝트 시뮬레이터는 register file forwarding이 없기 때문에, branch 명령어의 data hazard의 경우 forwarding으로 줄일 수 없다. RAW data hazard의 경우(단, LW,SW처럼 MEM에 접근하는 명령어 제외) write back을 해야하는 명령어와 그 값을 이용하여 연산해야 하는 명령어 사이에 nop가 1개도 없어도 된다. 왜냐하면,

IF ID EXEC MEM WB

IF ID EXEC MEM WB

위처럼 EXEC 단계에서 ALU 결과값이 forwarding 제어 신호를 통해 wb은 아직 하지 않았지만 WB 단계에 쓰일 값이 미리 준비되었기 때문에 다음 명령어의 EXEC 단계 즉, 다음 명령어의 연산 과정 전에 input으로 미리 넣어줄 수 있기 때문이다. 따라서 forwarding 없는 nop만 제거한 assembly code에서 RAW data hazard는 대부분 nop 3개였는데 이 부분을 삭제할 수 있다.

lw, sw처럼 memory에 접근하는 경우, EXEC 단계에서 결과값이 나오는 것이 아니라 MEM 단계에서 결과값이 나오므로, 바로 다음 명령어에 lw, sw의 결과값이 필요한 경우, 불가피하게 명령어 사이에 nop가 있어야 한다. (한 단계 쉬어야 함) 다음과 같다.

IF ID EXEC MEM WB

nop

IF ID EXEC MEM WB

branch 명령어에서 일어나는 data hazard의 경우 branch는 decoding 단계에서 결정되므로, forwarding이 적용될 수 없다. 따라서 branch 명령어에 쓰이는 값이 이전 명령어에 WB 단계까지 완전히 끝날 수 있도록 nop를 최대 3개까지 사용하여 data hazard를 피하는 방법밖에 없다. 다음과 같다.

IF ID EXEC MEM WB

nop

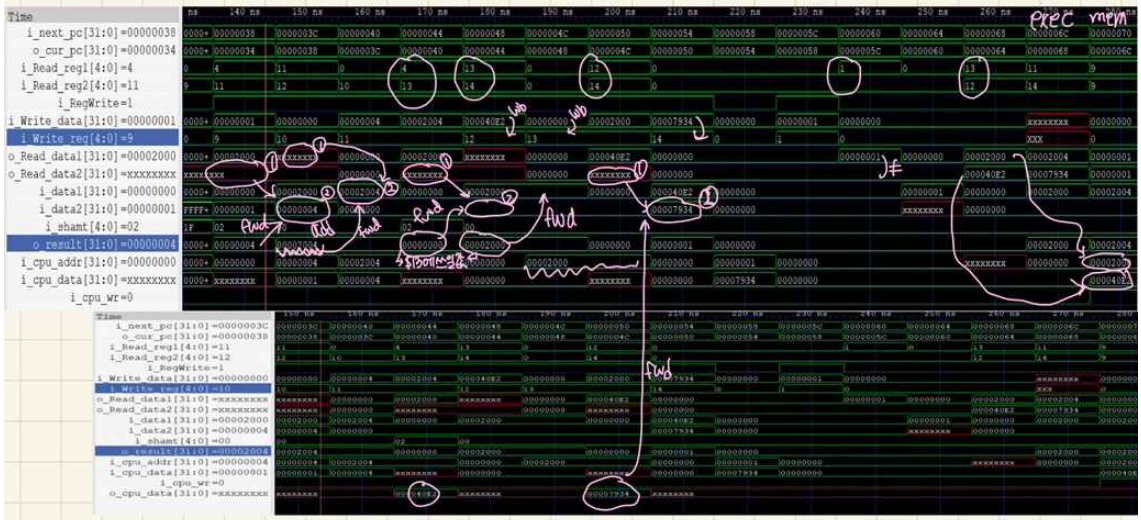
nop

nop

IF ID EXEC MEM WB

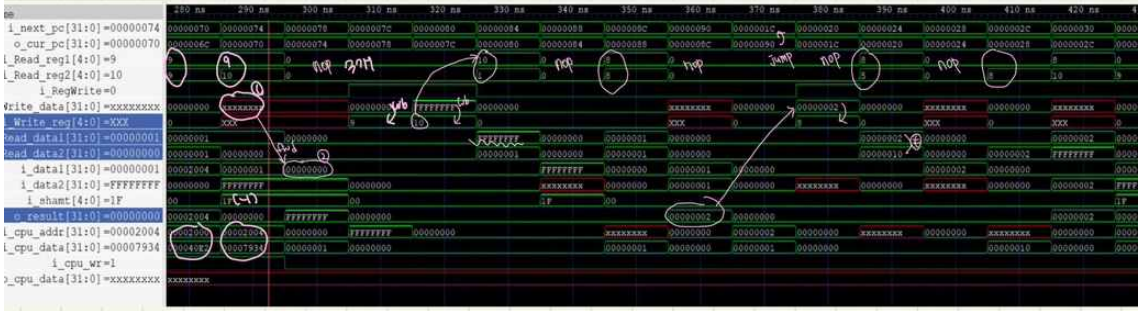
- 시뮬레이션 결과 분석

- i_write_reg 와 i_write_data을 통해 $4, $5 에 lui, ori가 잘 되었음을 확인가능
- addi $8, $0, 0x1 을 decoding 한 후 바로 다음 exec단계에서 0과 1을 더하여 결과값인 1을 $8에 적음.
- beq 이전에 $8에 WB까지 할 수 있도록 nop 3개 삽입
- beq 조건이 맞지 않고, fetch가 미리 되지 못하도록 nop 2개 삽입
- 그 뒤로 독립적인 명령어들이므로 fwd 없이 각각 add, addi 연산진행.
- sll $11, $9, 2를 decode할 때 $9에는 xxxxxxxx 값이 없는데 이전전 명령어의 add 결과값이 fwd되면서 EXEC 단계에서 $9의 값이 0x0000 0001 이 들어와 sll 연산을 무사히 수행
  shift 2칸 하여 x4를 해줌.



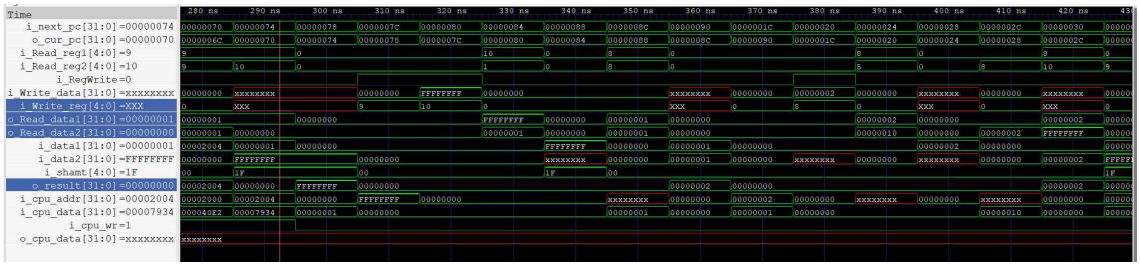
- add $11, $4, $11 의 decode 단계에서 $11은 쓰레기값이었으나 이전명령어 sll의 ALU 결과값이 fwd되면서 EXEC 단계에서 $11의 값이 0x4로 읽힘 / decode단계
- lw $12, 0($11)에서 $11의값이 아직 sll의 WB단계이전이라 쓰레기값이 나오고, double data hazard가 발생→ 가장 최근값을 fwd 하므로 add의 결과값인 0x00002004 값이 fwd 되어 lw의 exec 단계에서 주소값 계산이 수행됨
- 두 어셈블리 이상

- sll $13, $10, 2 는 이전 명령어와 상관이 없으므로 독립적으로 수행됨.
- add $13, $4, $13 의 decode 단계는 아직 $13이 WB 전이라 스테이지값이 원하지만 sll의 fwd 제어신호로 인해 sll의 연산결과인 0x0 값이 exec단계 ALU의 input 으로 들어갈 $13에 덮어가는 것은 확인가능
- lw $4, 0($13) 의 경우 double data hazard로 lw의 exec단계에서 add의 결과값인 0x00002000 값이 읽혀 해당 주소값으로 접근하여 값을 읽어와 $4에 WB해준다.
- lw는 mem 단계에서 결과값이 나오므로 nop을 1개 삽입하여 다음 명령어인 sll $1, $12, $4가 안전하게 진행될 수 있도록 함
- sll $1, $12, $4의 경우 decode 단계와 달리 exec단계에서 fwd를 통해 $4에 덮일 값인 0x00001934가 읽히는 것을 확인가능
- beq $1, $0, L3 에서 beq의 $1을 위해 nop 3개를 삽입하여 $1이 WB이 되기까지 기다림.
- beq 뒤에 바로 fetch 되지 않도록 nop 삽입 및 조건에 만족으로 다음 명령어인 sw수행.
- 0($13)의 주소 값으로가 $12 인 0x00004052값이 쓰임.



- sw $14, 0($11) 또한 앞의명령어들과 hazard가 발생하지않으므로 fwd없이 sw명령어를 수행.
- ① addi $9, $1, 1
  ② addi $10, $9, 1 — fwd — 해야하는데 실제로 2전에 $9값이 ①의 결과값이 (0x0)이 읽힘. (exec)
- beq 명령어를 위해 $10에 addi 명령이 WB 될때까지 기다리기위해 nop 3개 삽입
- beq 명령어 바로 직후 fetch되지 않도록 nop 1개 삽입
- beq 명령에서 조건에 맞지 않으므로 ($10<0) 뒤 명령어 진행하려 data hazard 가 없으므로 그냥진행.
- j L1 은 앞서 pc값이 0x00000090 → 0x0000001C로 덮어짐을 확인가능

-마지막에 $8, $5을 비교했을때 값이 0x000010으로 같으므로 done으로 branch (PC값이 9인) break을 안타 사물레이션종료



(위의 시뮬레이션 분석에 쓰인 testbench 사진만 첨부한 것)

- 명령 수행에 걸린 총 cycle 수

```
----------------------------------------------
| HO20-3-1647-01: Computer Architecture      |
|                 CE.KW.AC.KR                |
----------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
----------------------------------------------
 Break signal: 1,   # of Cycles:        1486
----------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 14975000 (1ps)

C:₩Users₩USER₩Desktop₩컴구실자료₩컴구프젝3₩prj3_PCPU_2023₩prj3_PCPU_2023>FC /L mem_dump_IS.txt mem_dump.txt
파일을 비교합니다: mem_dump_IS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩USER₩Desktop₩컴구실자료₩컴구프젝3₩prj3_PCPU_2023₩prj3_PCPU_2023>FC /L reg_dump_IS.txt reg_dump.txt
파일을 비교합니다: reg_dump_IS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩USER₩Desktop₩컴구실자료₩컴구프젝3₩prj3_PCPU_2023₩prj3_PCPU_2023>gtkwave tb_PC.vcd
```

forwarding 제어 신호를 추가한 후 추가로 nop를 더 제거하였을 때 1486 cycle이 나온다.

-M_TEXT_FWD.txt 분석

```
01_00  //  0x000  lui  $4, 0x0000    //ori의 $4(A port)에 ALU fwd
00_00  //  0x004   ori $4, $4, 0x2000
00_00  //  0x008   ori $5, $0, 0x10
00_00  //  0x00C   addi $8, $0, 0x1
00_00  //  0x010   nop //beq전에 $8에 값을 wb되기까지 대기
00_00  //  0x014   nop
00_00  //  0x018   nop

00_00  //  0x01C   L1 : beq $8, $5, done
00_00  //  0x020   nop //분기 판단 이전에 바로 fetch되는 것을 방지
00_00  //  0x024   add $9, $0, $8
00_10  //  0x028   addi  $10, $8, -1 //$9에 쓰일값을 다음 명령어의 B로 ALU fwd

00_01  //  0x02C  L2 : sll  $11, $9, 2 //ALU 결과값을 다음 명령어의 B로 fwd
01_00  //  0x030   add $11, $4, $11  //ALU 결과값을 다음 명령어의 A로 fwd
00_00  //  0x034   lw $12, 0($11)  //ALU 결과값을 다음 명령어의 A로 fwd

00_01  //  0x038  // sll  $13, $10, 2  //ALU 결과값을 다음 명령어의 B로 fwd
01_00  //  0x03C   add   $13, $4, $13  //ALU 결과값을 다음 명령어의 A로 fwd

00_00  //  0x040   lw  $14, 0($13)
00_10  //  0x044   nop  //lw의 mem 값을 다음 slt 명령어의 B로 fwd
00_00  //  0x048   slt $1, $12, $14
00_00  //  0x04C   nop //beq전에 $1에 wb되는 것을 대기
00_00  //  0x050   nop
00_00  //  0x054   nop
00_00  //  0x058   beq $1, $0, L3
00_00  //  0x05C   nop ////분기 판단 이전에 바로 fetch되는 것을 방지
00_00  //  0x060   sw  $12, 0($13)
00_00  //  0x064   sw  $14, 0($11)
01_00  //  0x068   addi  $9, $9, -1 //addi의 결과값을 다음 명령어의 A로 ALU fwd
00_00  //  0x06C  addi  $10, $9, -1
00_00  //  0x070   nop //beq전에 $10에 wb되는 것을 대기
00_00  //  0x074  nop
00_00  //  0x078   nop
00_00  //  0x07C   bgez  $10, L2
00_00  //  0x080   nop //분기 판단 이전에 바로 fetch되는 것을 방지

00_00  //  0x084   L3 : addi $8, $8, 1
00_00  //  0x088   nop //만약 L1으로 갈 시, $8에 data hazard방지
```

00_00 // 0x08C j L1
00_00 // 0x090 nop //jump는 decode단계에 하므로 바로 fetch되는 것을 방지
00_00 // 0x094
00_00 // 0x098

Ⅲ. 문제점 및 고찰

이번 프로젝트는 instruction fetch memory와 data memory를 따로 써서 structural hazard를 피할 수 있었지만, 처음에 프로젝트에 접근할 때 두 개 단계를 하나의 memory에서 하는 줄 알고 다음과 같이 structural hazard까지 고려하여 nop를 줬더니 cycle 수가 1607로 나왔다.

```
lw    $12, 0($11)

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

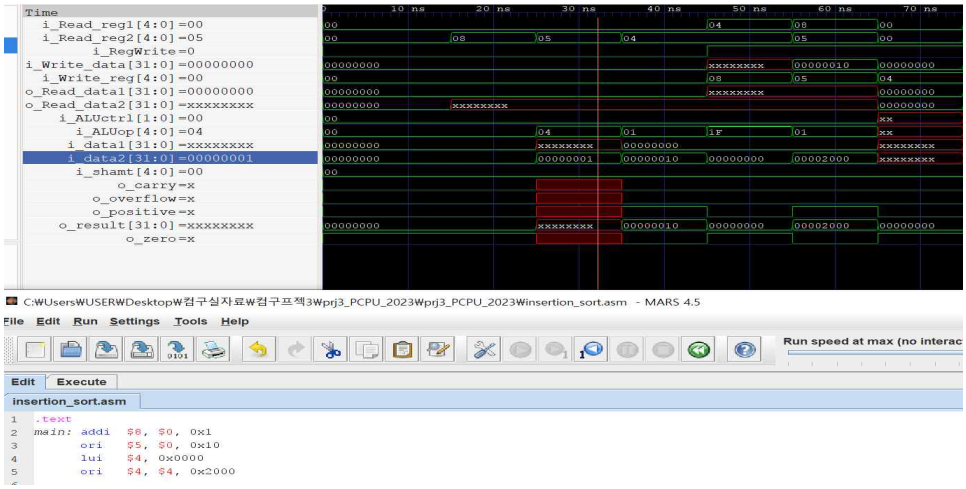
```
IF  ID  EXEC  MEM  WB  //lw
    IF   ID    EXEC  MEM  WB //sll
         IF    ID    EXEC  MEM  WB //add
               nop
                     IF    ID    EXEC  MEM WB //lw
```

위처럼 lw의 MEM 단계와 그 뒤의 lw의 fetch가 동시에 일어나지 않도록, nop를 넣어주었었다. 그러다가 정말 우연찮게 명령어 하나를 structural hazard가 발생하는 데 nop를 안 넣어주고 돌려줬었는데 결과가 잘 나오는 것을 확인하고, structural hazard를 고려하지 않아도 된다는 사실을 깨달아서 nop를 더 줄이고 cycle을 더 최소화시킬 수 있었다.

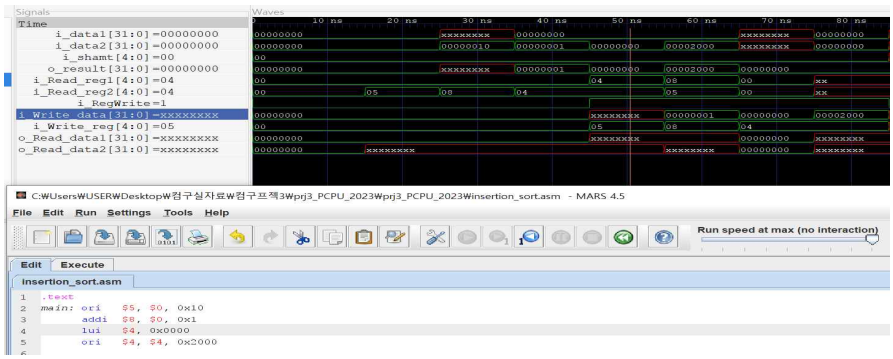
또한 명령어 rescheduling을 통해 더 nop를 줄일 수 있겠다고 생각하여 명령어 재배치를 해줬었다. 맨 처음의

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

L1:   beq   $8, $5, done
```

에서 ori $5, $0, 0x10와 addi $8, $0, 0x1의 경우 맨 처음에 진행하면 다음 beq를 위해 nop를 3번까지 안해줘도 된다고 생각하였다. 그래서 명령어 위치를 바꿔줬더니



addi $8, $0, 0x1의 exec단계에서 0번 레지스터와 imm값 0x1을 읽어 더하는 것이 아니라 8번 레지스터를 읽어 ALU input으로 넣어주어 쓰레기값이 연산되었다. 뒤의 ori명령어는 잘 해주었다. 그래서 순서 문제인가 싶어, ori와 addi의 위치를 바꿨더니 다음과 같이 나왔다.



방금까지만 해도 잘 연산되던 **ori $5, $0, 0x10** 명령어가 exec단계에서 5번 레지스터값을 읽어 쓰레기 값이 나오고, 또 두 번째로 addi를 넣어줬더니 그 연산은 잘 수행하여 0x1값을 $8에 잘 넣어주는 것이 확인된다. 첫 번째 연산을 할 때 시뮬레이션 부분이 좀 이상한 건지 내가 잘못한 건지 모르겠지만, 명령어 재배치를 한다면 더 cycle수를 줄일 수 있을텐데 라는 아쉬움이 있다.

FWD_TEXT를 작성할 때, 01은 바로 forwarding시켜주는 명령어에 써주는 반면, 10은 nop 부분에 작성해서 한 사이클 쉬어줘야한다는 점이 헷갈렸다. 또한, MARS는 처음 써보는 tool에 비해 설명이나 적응 기간이 부족해서 적응 및 dump를 이해하는 데에 시간이 많이 소요됐다.