컴퓨터 구조 (월, 수)

프로젝트 3

이성원 교수님

컴퓨터정보공학부

2018202076

이연걸

■ 실험 내용

- 프로젝트 이론 내용에 대한 설명

  Processor의 성능은 critical path에 의해 좌우된다. 이전까지 우리는 무어의 법칙을 활용한 scaling과 multicycle 등을 활용했지만 이번에는 **pipelining** 기법을 사용한다. 한 명령어의 실행이 종료될 때까지 다음 명령어가 기다리지 않고 단계별로 실행하는 방법으로 여러 명령어를 동시에 실행할 수 있다. 앞에서 단계별로 실행이라는 말을 했는데, MIPS 명령어 파이프 라인은 5단계로 이루어져 있다.

  IF(Instruction Fetch) 명령어 Fetch

  ID(Instruction Decode) 명령어 해독 + read register

  EX(Execution) 실행 및 주소 계산

  MEM(Memory) DM에 접근

  WB(Write Back) Write register

  이 5개의 단계는 Data path를 5부분으로 나눈 것이다. 명령어 하나당 5단계를 거치는데 5단계를 모두 거친 후 다음 명령어를 실행하는 것이 아닌, 단계별로 실행해 **Throughput**을 개선할 수 있다.

  이 기법을 사용해 우리는 버블 정렬을 더 빠르게 구현할 것이고 그 과정에서 발생할 수 있는 **harzard**를 피할 것이다.

  Harzard란 구조적, 명령어간 의존성에 의해 명령어가 제대로 실행되지 않는 것을 의미한다.

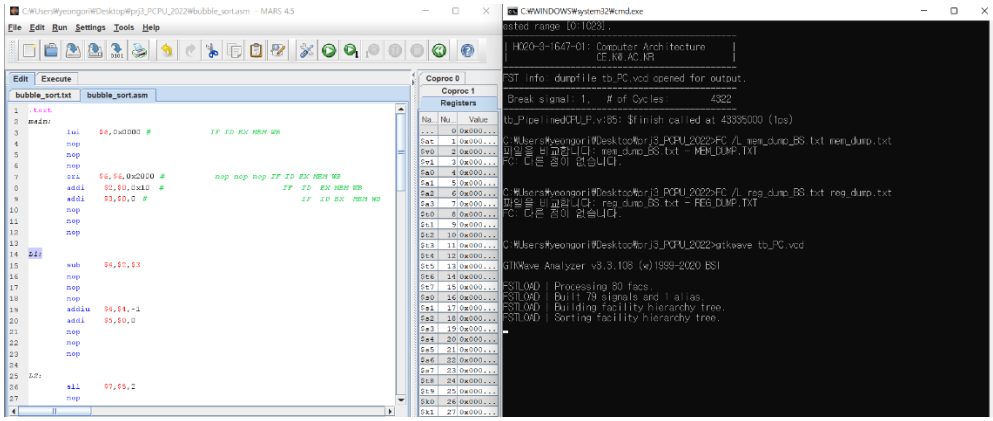
  lw명령어가 메모리에 접근할 때 다른 명령어를 메모리에서 가져오는 경우를 **Structure harzard**, 다음 명령어가 WB단계까지 거치지 않은 이전 명령어의 destination 레지스터를 참조하는 경우를 **Data harzard**, 브랜치 명령이 실행될 때 다음 명령어가 ID단계에 있는 경우를 **Control harzard** 라고 한다.

  이번 과제에서는 nop을 사용해 bubble을 추가하거나 forward을 사용해 bubble수를 줄여 Harzard를 해결할 것이다.

■ 검증 전략, 분석 및 결과

- Assembly code 설명

우선 nop(bubble)을 사용해 명령어 순서를 조절한 코드를 설명하겠다. 이를 **Stalling**이라 한다.

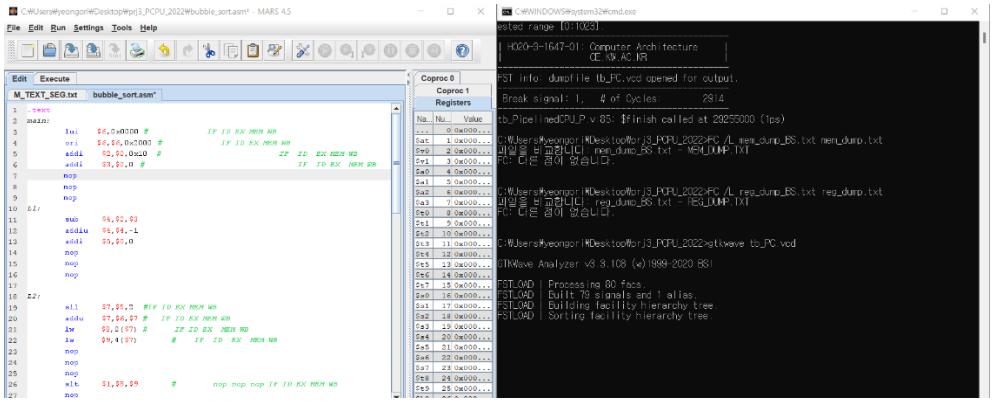


이 방법은 SW적인 해결방법으로 다음 명령어가 WB가 끝나지 않은 이전 명령어의 Destination 메모리를 참조해야 할 때, 즉 Dependency가 있을 때 bubble을 사용해 빈 cycle을 만들어 명령어의 실행을 뒤로 미룬다.

예시로 addi $3, $0, 0과 sub $4, $2, $3 사이에 3개의 nop이 들어간 경우를 살펴보겠다. Addi의 Destination register는 $3이다. 다음 명령어 sub의 Target register또한 $3이다. sub에서 $3은 addi명령어를 거친 $3을 가져와야 하는데 addi명령이가 WB단계에서 register에 값을 쓰지 않는 이상 해당 $3값을 가져올 수 없다. 이것을 Dependency가 존재한다 한다. Pipelining으로 바로 sub명령어가 실행된다면 Data harzard가 발생하게 되는데 이를 피하기 위해서 bubble(nop)이 필요하다.

따라서 addi의 ID단계에서 sub의 IF단계로 바로 들어가지 않게 nop을 3개 추가해 addi의 WB단계에서 sub의 IF단계가 실행될 수 있게 해준다. 이런 방식으로 명령어의 실행을 조절해주면 pipelining도중 발생할 수 있는 harzard를 피할 수 있다.

다음으로 **forwarding**을 사용해 불필요한 nop(bubble)을 줄이면서도 harzard를 피할 수 있는 방법 설명하겠다.



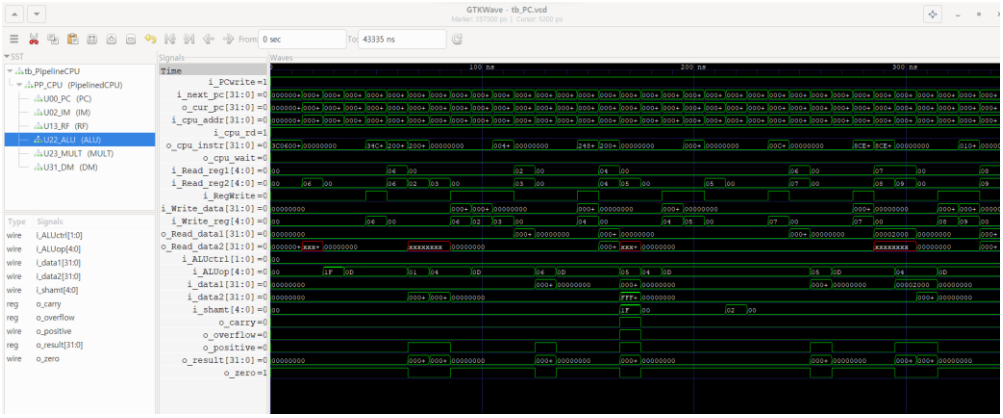
이번에는 위와 다르게 nop의 개수가 줄어든 것을 볼 수 있다. 이전에는 pipelining시 발생할 수 있는 Harzard를 피하기 위해서 nop(bubble)을 사용해 명령어의 실행단계를 조절했지만 이번에는 **forwarding**이란 기법을 사용해 bubble의 개수를 줄일 것이다.

Forwarding이란 이전 명령어의 EX나 WB의 결과를 바로 가져오는 것이다. 예시로 L2의 sll과 addu를 보자

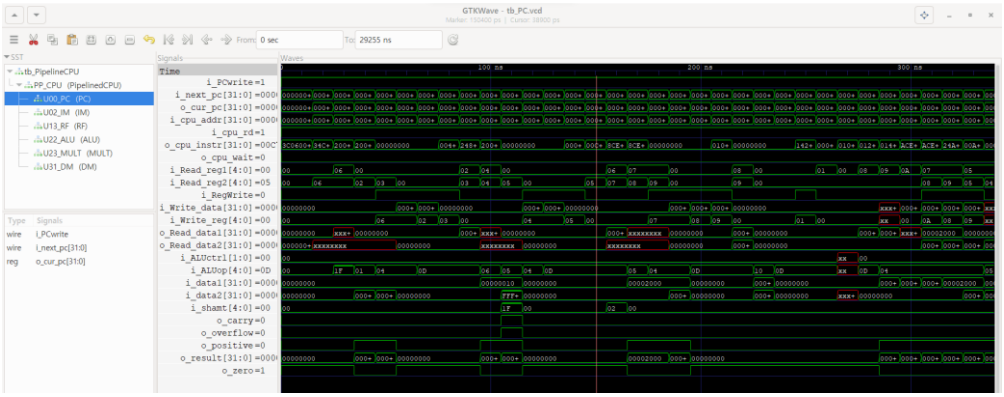
sll의 destination register는 $7이며 addu의 target, destination regieter도 $7이다. 여기서는 nop을 사용해 명령어의 실행을 조절해주지 않았는데 이렇게 되면 sll의 EX단계를 거친 값이 $7레지스터에 담기기 전에 $7레지스터를 참조하게 되고 이는 Data harzard를 발생시킨다. 이를 피하기 위해서 **forwarding**이 사용된다. sll의 EX값을 forwarding해 addu의 EX단계에서 사용하는 것이다. SII의 EX 값은 $5를 shift left 2한 값인데 그 값을 그대로 참조할 수 있다면 addu의 연산도 가능하다 따라서 Data harzard를 피할 수 있게 된다.

다른 예시로는 22번 행의 lw가 있다. lw는 WB에서 값을 가져와야 한다. 따라서 이전 명령어 21번이 아닌 20번 행의 addu의 WB값을 참조해 연산을 진행한다.

- 2개 시뮬레이션 결과 분석



^ **Stalling**: harzard없이 연산이 완료된 것을 확인할 수 있다.

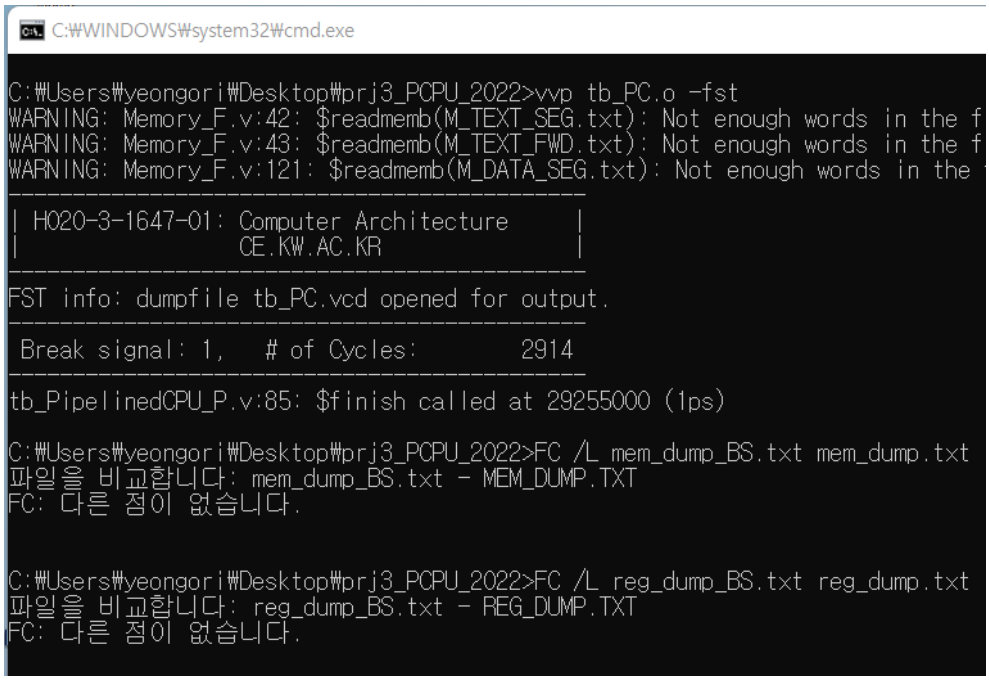


^ **Forwarding**: harzard없이 연산이 완료된 것을 확인할 수 있다.

- 명령 수행에 걸린 총 cycle 수

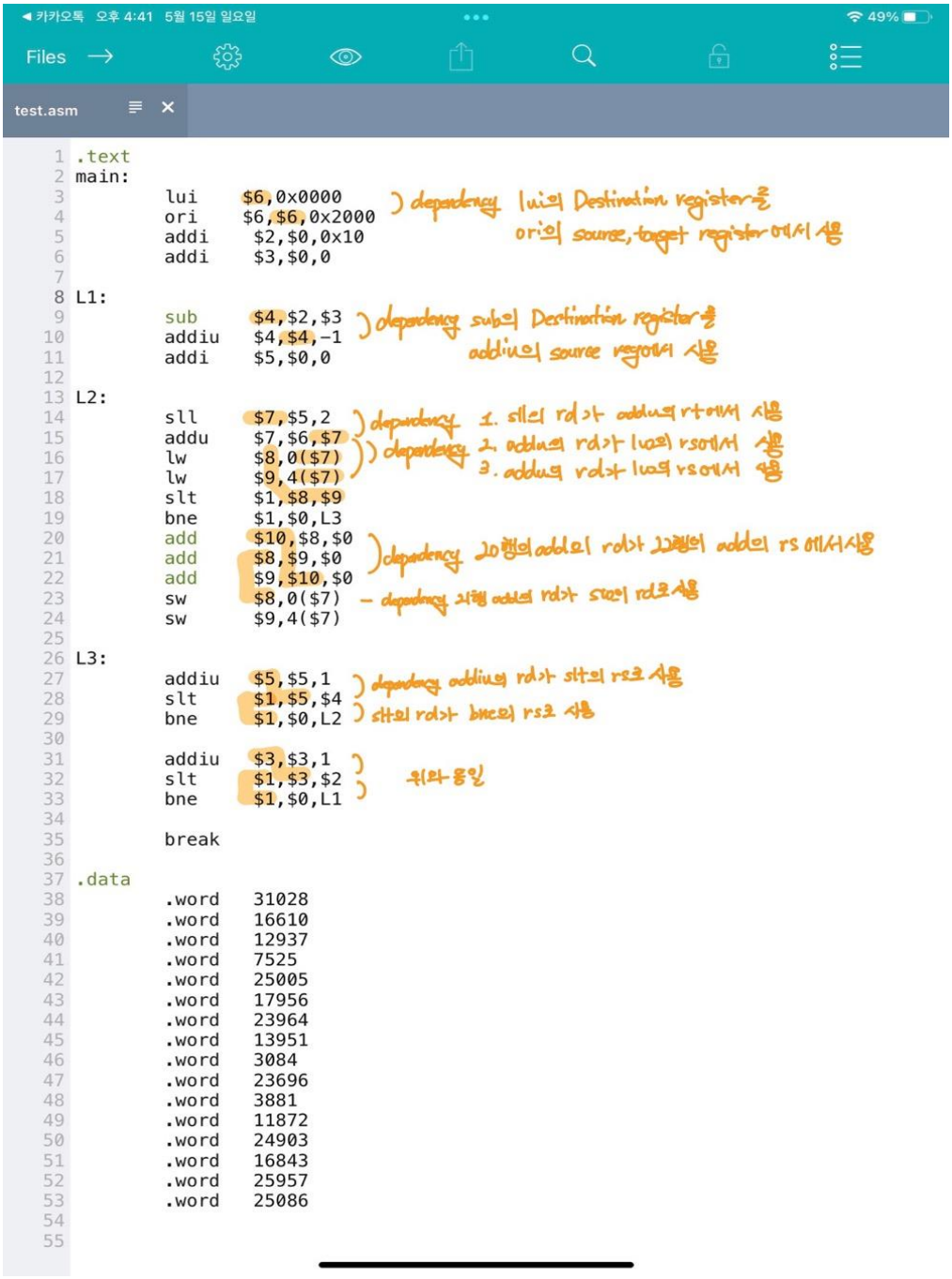
```
C:₩WINDOWS₩system32₩cmd.exe

C:₩Users₩yeongori₩Desktop₩prj3_PCPU_2022>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the fi
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the fi
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the fi
-----------------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-----------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------------
 Break signal: 1,   # of Cycles:        4322
-----------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 43335000 (1ps)

C:₩Users₩yeongori₩Desktop₩prj3_PCPU_2022>FC /L mem_dump_BS.txt mem_dump.txt
파일을 비교합니다: mem_dump_BS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩yeongori₩Desktop₩prj3_PCPU_2022>FC /L reg_dump_BS.txt reg_dump.txt
파일을 비교합니다: reg_dump_BS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

Stalling을 사용했을 때는 4322개의 Cycle이 필요했다. 하지만 여기서 Forwarding을 사용해 bubble의 개수를 줄이자



이처럼 2914개로 확연히 줄어든 모습을 볼 수 있다.

1. 제시된 Bubble sort의 dependency들을 코드가 적힌 종이에 표시 및 설명



2. Bubble Sort의 2가지 시뮬레이션 진행 및 설명

   위의 "Assembly code 설명" 부분과 동일하다.

■ 문제점 및 고찰

Stalling을 먼저 진행했는데 계속해서 결과값이 일치하지 않았다. 이유를 찾기 위해 branch부분을 중점적으로 보았는데 branch명령어가 진행되면서 이전 명령어의 WB가 끝나지 않고 branch되는 부분을 발견하여 bubble을 추가해 수정했다.

하지만 여전히 결과값이 일치하지 않았는데 바로 Control harzard를 고려하지 않았던 것이다. 다음 명령어가 ID될 때 branch 명령어가 EX되면 Control harzard가 발생한다.

Control harzard를 해결하기 위해서 모든 branch명령어 뒤에 bubble을 추가해 명령어 순서를 조절했다. 이 방법은 실행될 필요가 없는 cycle을 늘려 성능을 저하시키는 원인이 되는데, 브랜치를 예측할 수 있는 Branch History table등의 방법을 사용할 수 있었다면 개선할 수 있었을 것이다.

Forwarding기법을 사용할 때 bubble을 최대한 지양하는 방향으로 시작했다. 진행 도중 현재 명령어가 실행될 때 필요한 register가 이전과 그 이전 명령어들의 Destination register인 경우가 종종 있었다. 이때는 forwarding을 사용할 수 없고 bubble을 추가해야 했다. 그리고 lw가 중첩되어 나오는 경우에 한번은 MM stage에 있는 값을, 두번은 WB stage에 있는 값을 가져왔어야 했지만 둘다 MM 값을 가져와 일치하지 않았던 적이 있었는데, Forwarding이 바로 이전 명령어의 단계만 참조하지 않는 다는 점을 배울 수 있었다.