

Project #3

Pipeline Architecture

|  |  |  |
| --- | --- | --- |
| 과목명 |  | 컴퓨터구조 |
| 담당교수 |  | 이성원 교수님 |
| 학과 |  | 컴퓨터정보공학부 |
| 학년 |  | 3학년 |
| 학번 |  | 2019202009 |
| 이름 |  | 서여지 |
| 제출일 |  | 21.05.20 (목) |



1. 실험 내용
   1. 프로젝트 이론 내용에 대한 설명
      1. hazard

Pipeline architecture가 주어진 명령어를 해당 clock cycle에 정상적으로 수행할 수 없는 경우를 hazard라고 부른다. hazard는 그 원인에 따라 다음과 같이 분류할 수 있다.

structural hazard는 어떤 clock cycle에서 수행해야하는 동작을 진행할 module이 부족한 경우 발생한다. 예를 들어 multi cycle architecture의 경우 다른 clock cycle동안 동일한 ALU를 이용하여 다음 pc의 값과 명령어에 해당하는 계산을 모두 계산하지만, pipeline architecture의 경우 instruction fetch단계에서 이용하는 ALU와 execution 단계에서 이용하는 또다른 ALU로 분리하여 이용한다. structural hazard가 발생한 경우 부족한 module을 추가하거나, 명령어에 stall을 삽입하는 방법으로 문제를 해결할 수 있다.

data hazard는 어떤 명령어에서 이용하는 data가 해당 명령어의 앞에서 값이 변화하지만, 수행하려는 명령어를 decoding하기 전까지 갱신되지 않은 상황에서 발생하고, 이러한 경우 data dependency가 존재한다고 말한다. 예를 들어 ALU연산의 결과를 이용하여 곧바로 이어지는 branch의 조건을 검사하는 경우, pipeline architecture에서 branch의 decoding 단계 이전에 alu연산의 결과가 register에 write back 되어야 정상적으로 작동하지만, branch가 decoding 단계일 때 alu연산 명령어는 execute 단계이므로 아직 register의 값이 갱신되지 못하였다. 이러한 data hazard의 해결방법으로는 명령어에 stall을 삽입하여 register의 값이 갱신된 이후에 그 값을 이용하게 하는 방법과, code scheduling을 이용하여 register에 값을 갱신하는 명령어를 data dependency가 없는 다른 명령어보다 먼저 수행하여 해당 data를 이용하는 명령어의 decoding stage이전에 write back stage를 완료하는 방법이 있다.

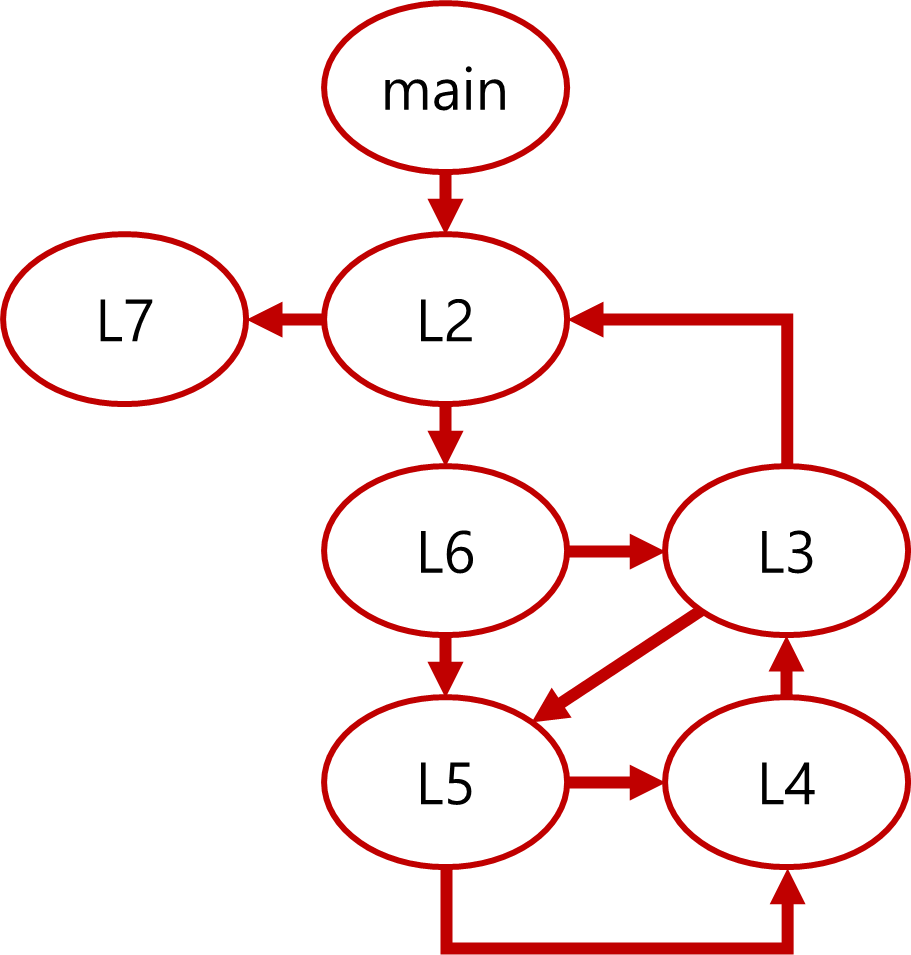
control hazard는 jump나 branch명령어의 사용으로 다음에 수행할 명령어가 pc+4에 위치한 명령어가 아니게 되는 경우 발생한다. 이 경우 다음 명령어의 주소를 구하기 전까지 fetch stage를 지난 명령어를 모두 삭제하고, 새로운 주소에 위치한 명령어를 fetch하여 이용한다.

* + 1. h/w

이번 프로젝트에서 제시된 pipeline architecture의 pipeline은 fetch, decoding, execute, memory, write back의 5개 stage로 구성되어 있다. hazard를 해결하기 위한 별도의 module이 없기 때문에 제시된 명령어에 stall을 삽입하는 방법으로 hazard를 회피해야 한다. 또한 실제 실행의 결과에서 jump address가 pc register에 전달되는 stage가, 제안서 구조도에서 제시된 memory state보다 두 단계 앞선 decoding stage인 것을 확인할 수 있었다.

* + 1. s/w

두 가지 프로그램이 제시되었다. instruction mix는 여러 명령어가 동작하며 register와 data memory에 값을 저장한다. 해당 프로그램은 이전 프로젝트 1, 2에서 제시된 프로그램과 유사하다. bubble sort는 data ROM의 정보를 오름차순으로 정렬하여 memory에 저장한다. bubble sort알고리즘은 가장 앞부터 순서대로 2개의 수를 비교하여 순서가 맞지 않는 경우 두 수의 위치를 교환하여 데이터를 마지막부터 정렬한다. 프로그램의 전체 구조는 다음과 같다.



1. 검증 전략, 분석 및 결과
   1. 자신이 구현한 assembly code 설명

주어진 assembly code에서 몇 가지 패턴에 따라 nop 명령어의 수를 조정하였다.

* + 1. data dependency 가 나타나는 경우

이 경우 먼저 fetch된 명령어의 write back stage가 종료된 이후에 다음 명령어가 decode 되어야 한다. 따라서 다음과 같이 stall을 이용하여 문제를 해결할 수 있다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ori | IF | ID | EX | MEM | WB |  |
| add |  | stall | stall | stall | IF | ID |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| add | IF | ID | EX | MEM | WB |  |  |
| add |  | IF | ID | EX | MEM | WB |  |
| add |  |  | stall | stall | IF | ID | EX |

* + 1. control hazard 가 나타나는 경우

주어진 pipeline architecture에서 jump address가 pc register에 새로 쓰이는 stage가 decoding stage이므로, decoding stage이후에 jump하여 다음 명령어를 fetch하게된다. 이렇게 jump address를 사용하는 경우 branch명령어 이후 여러 개의 nop명령어가 있더라도 하나만 fetch하고 jump address로 이동하게 된다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| beq | IF | ID | EX | MEM | WB |  |
| slt |  | stall | IF | ID | EX | MEM |

* + 1. control hazard와 data hazard가 동시에 발생하는 경우

branch를 이용하여 pc가 이동한 뒤, 새로 fetch한 명령어에서 data hazard가 발생한 경우 해당 문제를 해결하기 위하여 branch 명령어의 앞에 stall을 추가하였다.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| addi | IF | ID | EX | MEM | WB |  |  |  |  |
| addi |  | IF | ID | EX | MEM | WB |  |  |  |
| b |  |  | stall | IF | ID | EX | MEM | WB |  |
| slt |  |  |  |  | stall | IF | ID | EX | MEM |

* + 1. hazard가 나타나지 않은 경우

이 경우 stall 없이 pipeline을 채울 수 있다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| add | IF | ID | EX | MEM | WB |  |
| add |  | IF | ID | EX | MEM | WB |

* + 1. code scheduling

code scheduling을 통해 순서를 뒤바꾼 부분은 두 부분으로, 다음과 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| main | lui $6, 0x0 |  | main | lui $6, 0x0 |
|  | ori $6, $6, 0x20000 |  |  | addi $2, $0, 0x10 |
|  | addi $2, $0, 0x10 |  |  | addi $3, $0, 0 |
|  | addi $3, $0, 0 |  |  | ori $6, $6, 0x20000 |
|  | b L2 |  |  | b L2 |

(b -> bgez $0, L2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| L6 | sub $4, $2, $3 |  | L6 | sub $4, $2, $3 |
|  | addiu $4, $4, -1 |  |  | addi $5, $0, 0 |
|  | addi $5, $0, 0 |  |  | addiu $4, $4, -1 |
|  | beq $0, $0, L3 |  |  | beq $0, $0, L3 |

main lable이후의 내용을 code scheduling한 예를 들어보면 다음과 같다.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| lui | IF | ID | EX | MEM | WB |  |  |  |  |
| ori |  | stall | stall | stall | IF | ID | EX | MEM | WB |
| addi |  |  |  |  |  | IF | ID | EX | MEM |
| addi |  |  |  |  |  |  | IF | ID | EX |

<code scheduling 이전>

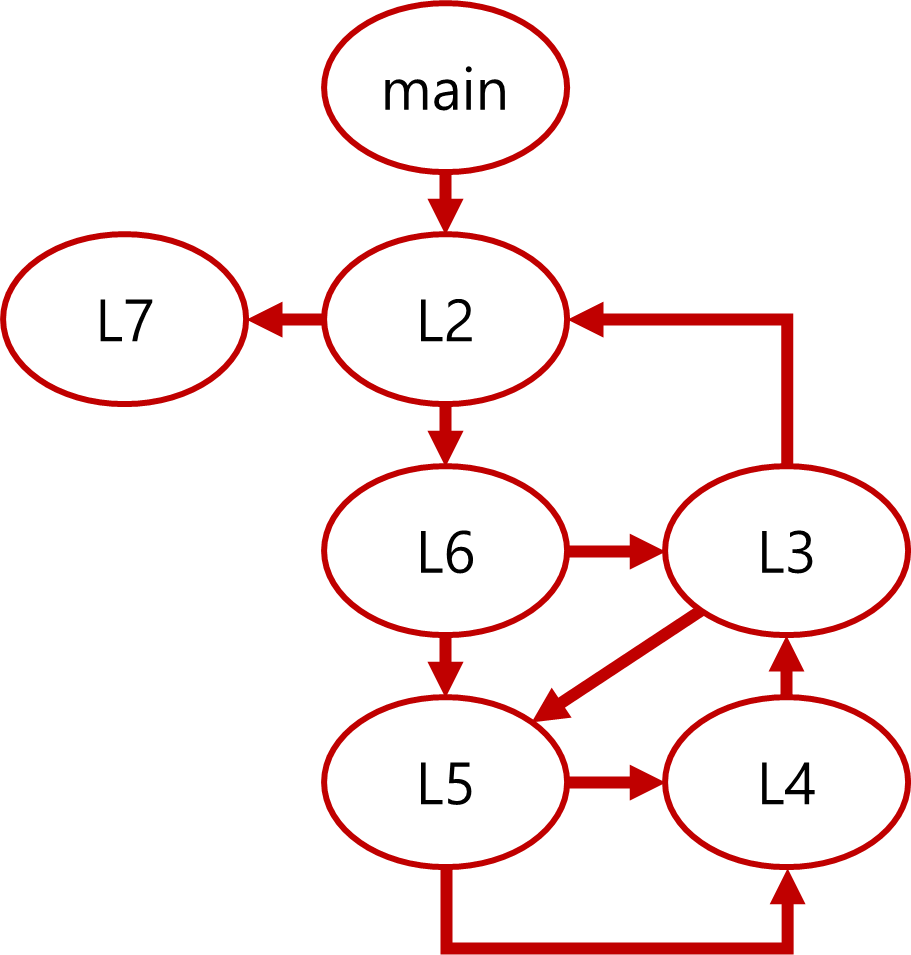
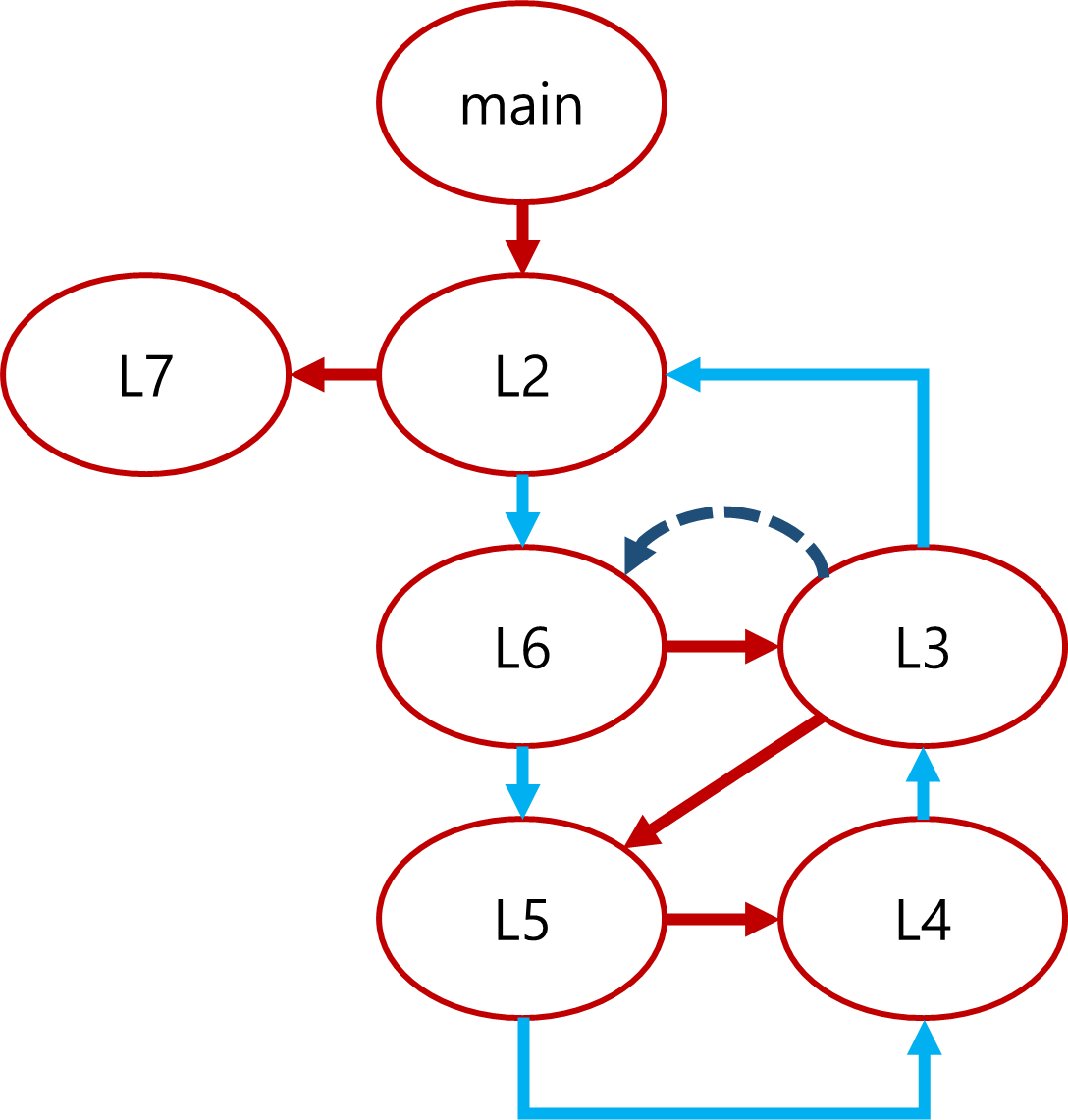
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| lui | IF | ID | EX | MEM | WB |  |  |  |  |
| addi |  | IF | ID | EX | MEM | WB |  |  |  |
| addi |  |  | IF | ID | EX | MEM | WB |  |  |
| ori |  |  |  | stall | IF | ID | EX | MEM | WB |

<code scheduling 이후>

code scheduling을 이용하여, lui와 ori사이 발생한 data hazard를 해결하기위해 삽입하는 stall의 수를 줄일 수 있었다.

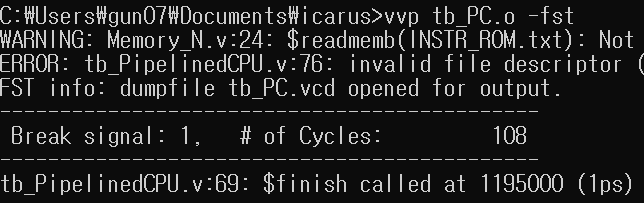
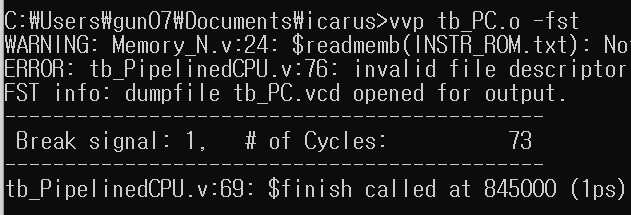
* + 1. loop unrolling

loop unrolling은 loop내부에서 반복된 것을 풀어 직접 나열하는 것을 통해 clock수를 줄여 프로그램 실행 속도에서 이점을 얻는 방법이다. loop의 조건을 확인하는 횟수가 줄어들기 때문에 프로그램의 실행속도가 개선된다. 그러나 반복되는 내용을 풀어 나열하기 때문에 프로그램 코드의 길이가 길어지는 단점이 있다. bubble sort 프로그램을 loop unrolling하여 두 개의 내용이 정렬될 때마다 loop의 조건을 확인하도록 작성하였다. 주어진 코드의 내용에서 전체 루프의 조건을 확인하는 L2 lable을 거치지 않고, L3에서 바로 L6으로 이동하도록 설계하였다. L3에서 이동할 lable을 확인하기 위하여 $11을 추가로 이용하였다.

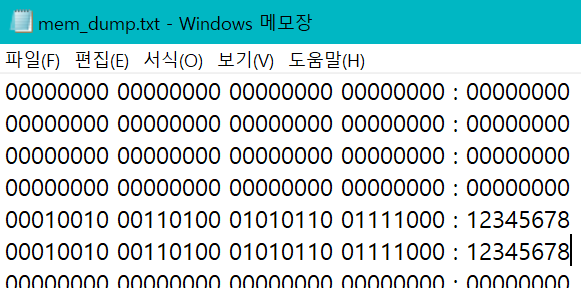
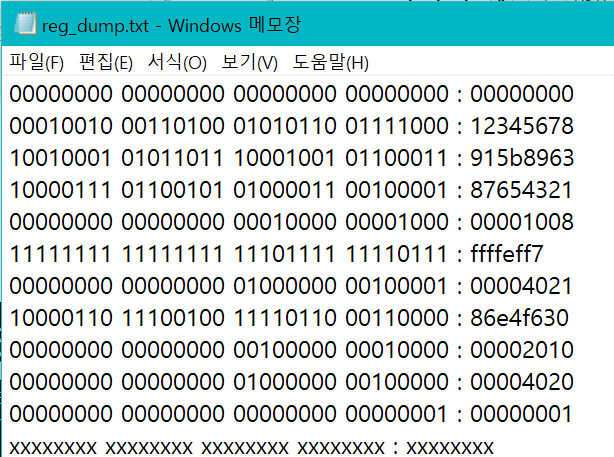
<loop unroll 이전> <loop unroll 이후>

* 1. 5개의 시뮬레이션 결과 분석
     1. instruction mix

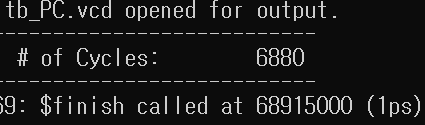
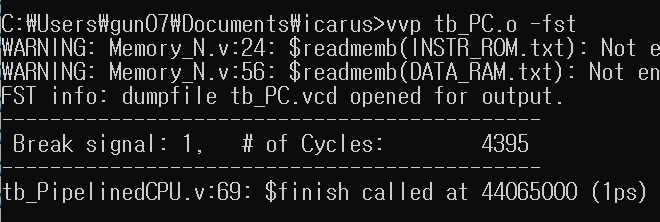
<nop 명령어 삭제 이전> <nop 명령어 삭제 이후>

실험 결과 pipeline을 이용하여 프로그램 실행시간이 단축된 것을 확인할 수 있다. 사용된 clock cycle 수는 108에서 73으로 줄어들어 약 67.6%로 감소하였다.



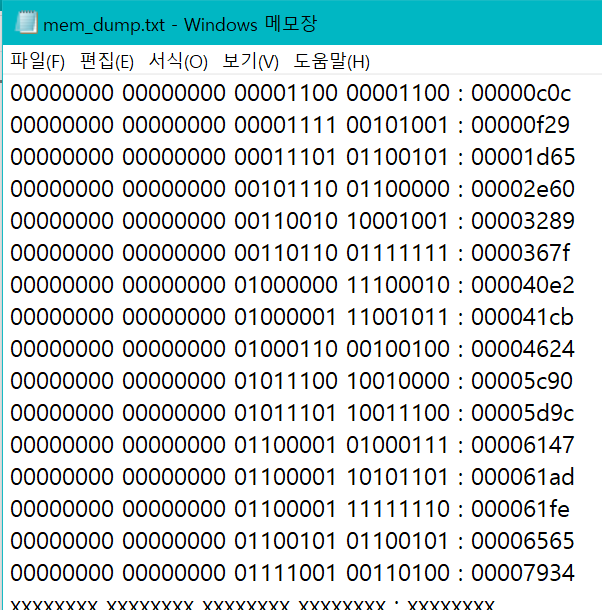
<실행 결과>

* + 1. bubble sort

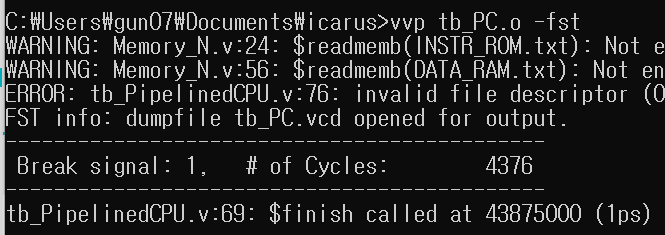
<nop 명령어 삭제 이전> <nop 명령어 삭제 이후>

16개의 수를 모두 정렬하는데 사용된 clock cycle은 6880에서 4395(63.9%)으로 감소하였다. 실험(1)의 instruction mix에서보다 약 3.7% 더 많이 감소한 것을 확인할 수 있다.



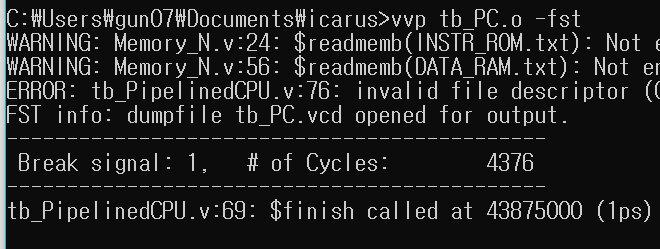
<정렬 결과>

* + 1. bubble sort code scheduling

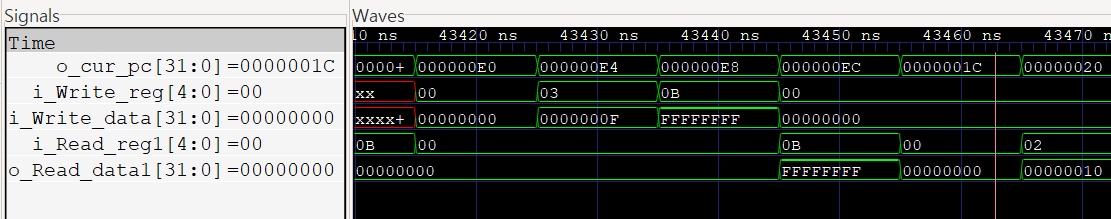
 <code scheduling 결과>

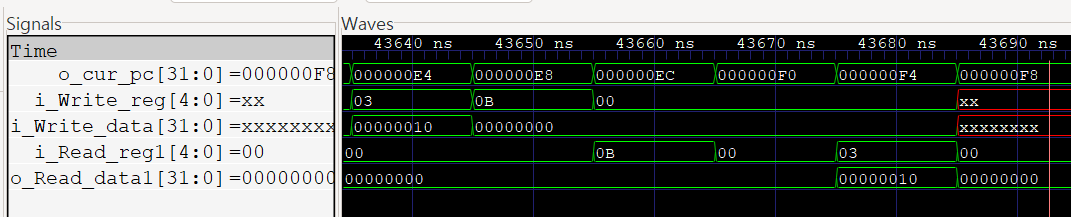
code scheduling을 통해 프로그램 실행시간이 4395 cycle에서 4376 cycle로 감소하였다. 0.4%의 적은 감소율을 보였다.

* + 1. bubble sort loop unrolling

 <loop unrolling 결과>

loop unrolling 결과 프로그램 실행시간이 줄어들지 않았다. 이는 L3에서 L2와 L6중 다음 주소를 확인하기 위해 새로 작성된 부분의 명령어가 L2를 실행하지 않아 줄어든 것과 동일한 양만큼 실행된 것으로, loop unrolling은 실행되었지만 실행시간은 동일했다. $11을 이용하여 다른 주소(각각 L2와 L6)로 이동하는 것을 통해 이를 확인할 수 있다.





* + 1. bubble sort loop unrolling & code scheduling

실험(4)에서 명령어의 순서를 바꿀 수 있는 부분을 발견하지 못하였다. 따라서 결과는 위와 동일하다.

1. 문제점 및 고찰

이번 프로젝트는 hazard를 해결하기 위한 별도의 장치가 없는 pipeline architecture에서 주어진 프로그램이 동작하도록 stall을 적절하게 삽입하는 과제였다. data hazard와 control hazard를 해결하기 위해 NOP operation의 수와 위치를 지정하였다. 또한 code scheduling과 loop unrolling을 이용하여 프로그램을 최적화하였다.

mars 실습 환경을 준비하는 것에 실패하여 과제를 진행하기 위해 jump와 branch 기계어 명령의 offset부분을 직접 수정해야 했는데, 이것을 잘못 계산하는 실수를 여러 번 반복하여 프로그램이 제대로 동작하지 않는 경우가 많았다. offset의 값은 branch 명령어를 fetch했을 때의 pc값이 아닌, 해당 pc 값에 4를 더한 값을 Label의 주소 값에서 뺀 것임에 주의해야했다.

loop unrolling 과정에서 조건을 비교하는 횟수를 줄이기 위하여 전체 loop의 조건을 확인하는 L2를 건너뛰도록 코드를 작성하였다. 그러나 이 방법은 몇 가지 부족한 점이 있다. 정렬하려는 값이 홀수 개인 경우 한 번의 loop에 2개의 수를 정렬하는 코드가 정상적으로 동작하지 않는다. 또한 결과에서 확인할 수 있듯이 실제 clock cycle이 감소하지 않았기 때문에, 기존 프로그램 코드보다 더 길어지기만 했을 뿐, 성능이 향상되지 않았고 사용하는 register의 수가 하나 증가하여 더욱 비효율적인 동작을 수행한다. 더 효율적인 프로그램을 작성하지 못한 점이 아쉽다.

1. 참고자료

이성원교수님, 컴퓨터구조 강의자료, 광운대학교 컴퓨터정보공학과 2021

Price, Charles. "MIPS IV instruction set." (1995).

Peter Rounce, mips examples,

http://www0.cs.ucl.ac.uk/staff/electran/gc03/pdf/07mips\_examples.pdf

mips opcodes, https://opencores.org/projects/plasma/opcodes