Computer Architecture

Pipeline Architecture

날 짜 : 2019. 05.25

교수님 : 이성원 교수님

학 과 : 컴퓨터정보공학부

학 번 : 2015722025

이 름 : 정 용 훈

1. 실험 내용

A. 실험 내용에 대한 설명

해당 프로젝트는 Pipeline CPU의 이해를 돋기 위해 assembly code로 구현된 bubble sorting 프로그램을 hazard가 발생하지 않도록 구현하고, 최종적으로 cycle수를 줄이는데 목적을 두고 있다. 여기서 hazards의 종류는 3가지가있는데, 각 각 다음과 같다.

1. **Structural hazards**

   Structural hazards는 두 명령이 동시에 한 hardware resource에 접근하는 경우 발생한다.

2. **Data hazards**

   Data hazards는 앞선 명령에서 사용중인 (즉, 값이 쓰이기 전인) data에 뒤에 있는 명령에서 접근하는 경우 발생한다.

3. **Control hazards**

   명령에 의해 PC값이 바뀌는 경우 (Instruction이 Fetch되는 장소가 바뀌는 경우)이다.

제공된 코드를 기반으로 위와 같은 hazards가 발생하지 않도록 code를 제시된 실험 순서에 맞게 구현하여 이를 비교하는데 목적을 둔다. 추가적으로 unrolling을 통한 Cycle의 변화도 확인할 수 있다.

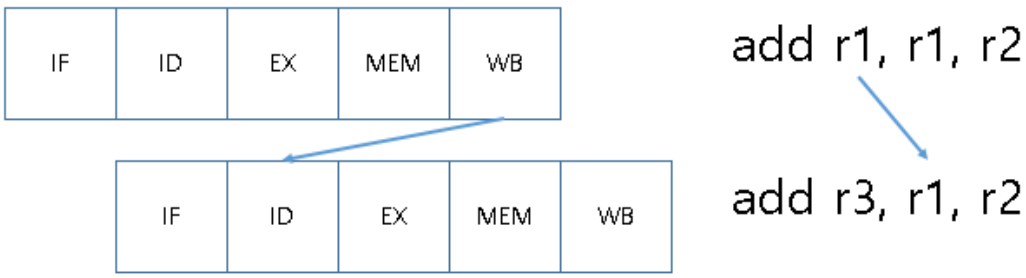
B. 문제해결 방안

각각의 hazards를 해결하는 방법은 다음과 같이 나타낼 수 있다.

1. Structural hazards

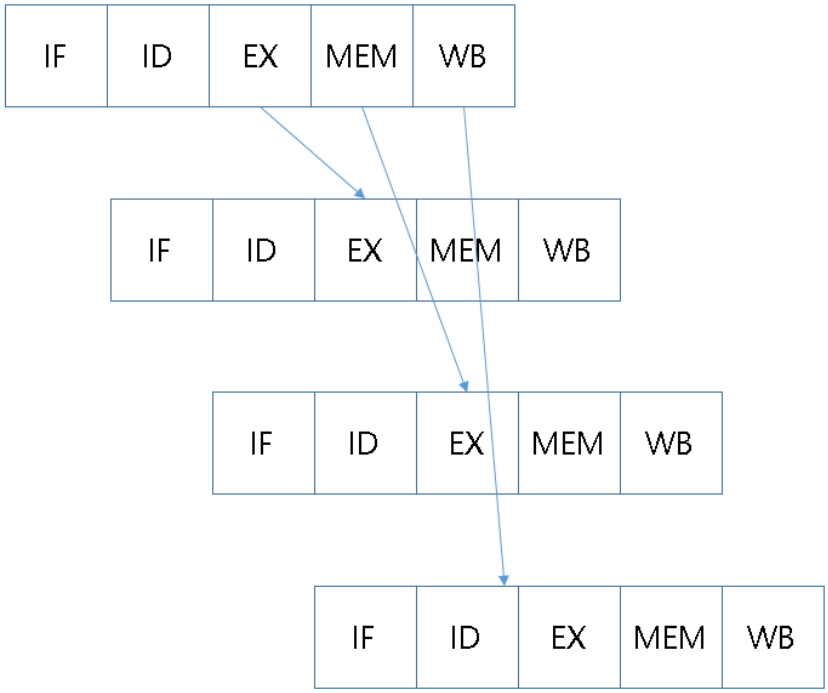
처음 그림은 instruction이 같은 hardware에 동시 접근하며, 생기는 구조적 문제인 structural hazards를 나타낸 그림이다. Structural hazards는 Stall을 넣어 간단하게 해결 할 수 있다. 이러한 이유로 IM와 DM도 나누어져 있다.

2. Data hazards



다음은 이전 명령에서 아직 계산이 완료되지 않았지만 다음 명령에서 해당 값을 원하게 되면 생기는 hazards이다. 다음과 같은 경우 이번 과제에서 수행 해야 하는 scheduling이나 stall, 아니면 아예 레지스터를 바꾸는 과정을 통하여 해당 문제를 해결할 수 있다. 그에 대한 예시는 아래와 같다.

HW-Forwarding을 통한 해결 방법



다음은 하드웨어를 변형하여 forwarding unit을 통한 방법이다. 원래 WB과정을 거쳐야 정확한 값이 레지스터에 쓰이지만 애초에 정확한 값은 ALU에서 계산이 된 후 완성이 되어있으므로 완성된 값을 그 이후 명령에서 쓰인다면 다음과 같이 완성된 값만 넘겨주는 방식으로 stall없이 명령을 수행할 수 있도록 할 수 있다.

SW-Scheduling을 통한 해결 방법

```
lw   t1, 0(t0)
lw   t2, 4(t0)
add t3, t1, t2
sw   t3, 12(t0)
lw   t4, 8(t0)
add t5, t1, t4
sw   t5, 16(t0)
```

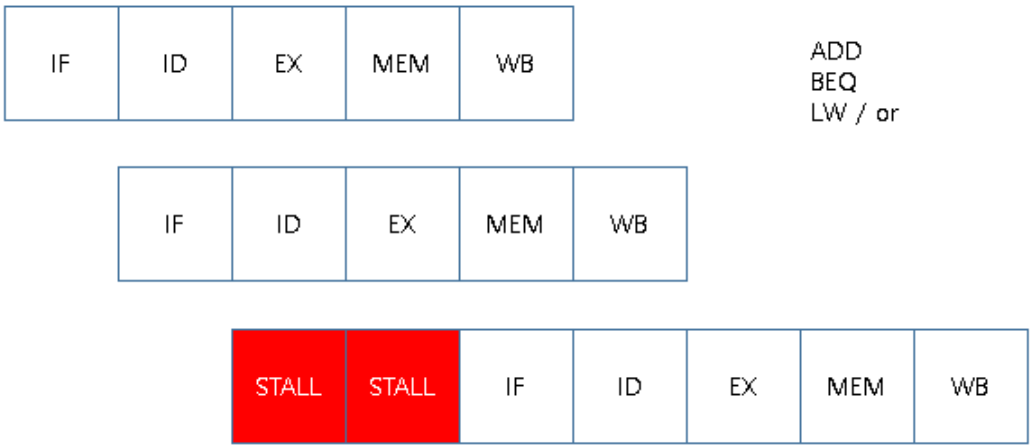
13 cycles

```
lw   t1, 0(t0)
lw   t2, 4(t0)
lw   t4, 8(t0)
add t3, t1, t2
sw   t3, 12(t0)
add t5, t1, t4
sw   t5, 16(t0)
```

11 cycles

다음은 code의 위치 등을 변형하여 stall을 대신하여 hazards가 발생하지 않도록 고치는 방법이다. 똑같은 동작을 하는 범위에서 다음과 같이 코드의 위치를 바꾸는 경우 cycle이 줄어드는 것을 확인할 수 있다.

3. Control hazards



다음은 branch에 의한 control hazards다. Delayed branch나 extra hardware를 추가하지 않는다면 stall을 2개 넣어 Jump가 결정되는 것을 확인하고 IF단계가 이루어지는 것이 정석이라 cycle의 손해가 굉장히 크다.

(1) 다음 문제를 해결하여 cycle을 줄일 수 있는 방법은 HW적으로 EX단계가 아닌 ID 단계 끝에서 미리 값을 계산하여 Jump를 할지 하지 않을지 결정하여 stall자체를 2개가 아닌 한 개만 필요할 수 있도록 설계하는 방법이다.

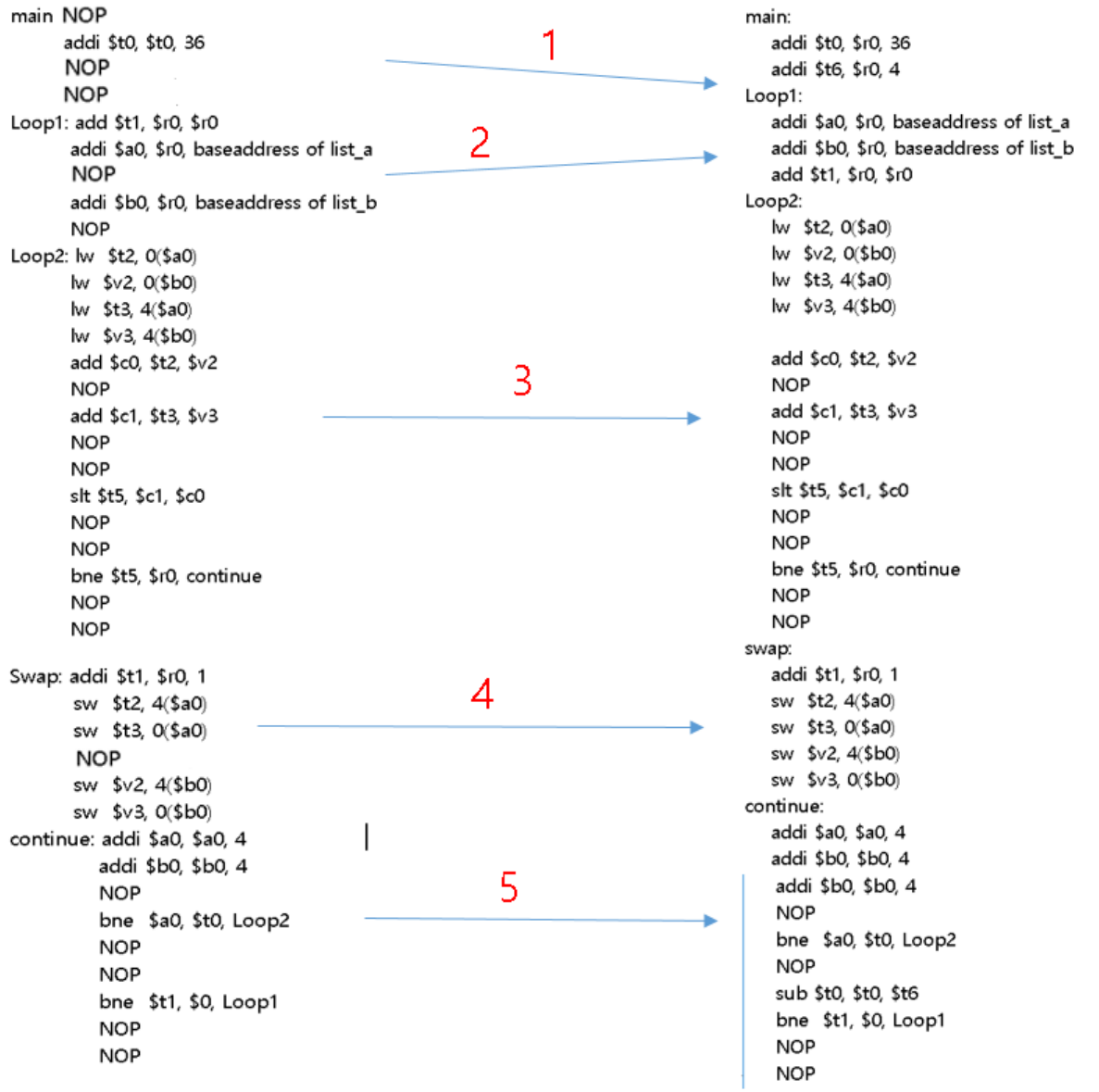
(2) 두 번째는 위와 같은 방법을 이용하면서 branch slot을 이용하는 것이다. 우선 branch명령이 들어오게 되면 branch아래있는 instruction은 무조건 수행하게 되며, (단, data dependency가 일어나면 안 된다)원래 stall이 들어가 있어야 하지만 instruction하나를 수행하면서 해당 stall을 없애는 방법이다.

2. 검증 전략, 분석 및 결과

검증 하는 방법으로는 1, 2번째 수행한 코드를 비교하고 분석하며, 2, 3번째 수행한 코드를 비교하고 분석하고 마지막으로 3, 4번째 수행한 코드를 비교하며, 분석하겠습니다. 분석하는 항목으로는 첫째 코드의 변화 둘째 cycle수, 셋째 cycle의 변화 이유로 설명하겠습니다.

(1) 1번째, 2번째

1번째코드는 제공된 코드로써 단순히 어셈블리 코드를 기반으로 머신 코드를 구현하여 실행시킨 결과이며, 2번째코드는 1번째 코드를 기반으로 cycle을 줄이기 위하여 scheduling한 코드입니다. 아래 사진은 1번째 코드와 2번째 코드를 비교한 것입니다.



1번째 코드의 변화는 우선 base address가 0부터 시작이므로 base address를 더하는 과정을 생략

하고 루프 자체가 끝났는지 안 끝나는지 확인하는 숫자인 36을 더해줍니다. 다음으로 2번째 과정에서는 t1에 r0+r0를 하는 과정을 아래로 내림으로써 NOP이 들어가있던 자리에 NOP을 대신하여 instruction을 수행하는 과정으로 대체하여 cycle을 줄이며 스케줄링 하게 되었습니다. 3번째는 루프2에 관련된 명령들인데 솔직히 스케줄링 하는 아이디어가 떠오르지 않아 그대로 두게 되었으며 원래 beq였던 과정을 SLT를 반전시켜 bne로 계산하였고 2번째 코드 에서도 마찬가지로 bne만 사용함으로 동일한 동작이 가능하게 했습니다. 4번째 과정에서는 sorting을 진행하는데 필요하지 않은 과정이 있어 addi 명령을 하나 제거하였고 continue또한 리스트 하나에 관련된 조건만 생각하면 되므로 b에 관련된 명령을 지우게 되었습니다. 또한 전체 도는 loop의 횟수를 줄이기 위하여 sub명령을 통해 전체 주소에서 loop가 한번 돌 때 마다 빼주며 loop횟수를 줄였습니다. 이에 대한 결과 비교는 아래와 같습니다. (필요 없는 코드는 NOP으로 대체했습니다.)

1번 코드의 결과

List A

/tb_PipelineCPU_Ha...	3	3
/tb_PipelineCPU_Ha...	1	1
/tb_PipelineCPU_Ha...	7	7
/tb_PipelineCPU_Ha...	5	5
/tb_PipelineCPU_Ha...	2	2
/tb_PipelineCPU_Ha...	4	4
/tb_PipelineCPU_Ha...	9	9
/tb_PipelineCPU_Ha...	10	10
/tb_PipelineCPU_Ha...	6	6
/tb_PipelineCPU_Ha...	8	8

List B

/tb_PipelineCPU_Ha...	2	2
/tb_PipelineCPU_Ha...	6	6
/tb_PipelineCPU_Ha...	1	1
/tb_PipelineCPU_Ha...	4	4
/tb_PipelineCPU_Ha...	8	8
/tb_PipelineCPU_Ha...	7	7
/tb_PipelineCPU_Ha...	3	3
/tb_PipelineCPU_Ha...	5	5
/tb_PipelineCPU_Ha...	10	10
/tb_PipelineCPU_Ha...	9	9

2번 코드의 결과

List A

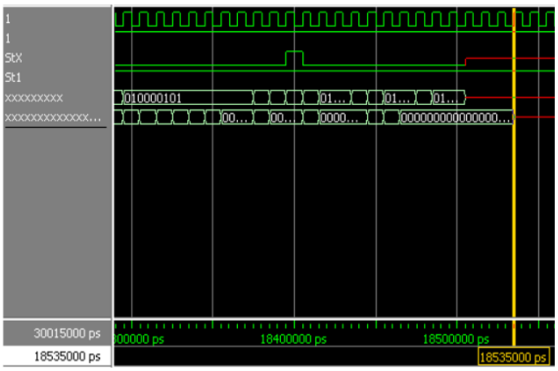
/tb_PipelineCPU_Ha...	3	3
/tb_PipelineCPU_Ha...	1	1
/tb_PipelineCPU_Ha...	7	7
/tb_PipelineCPU_Ha...	5	5
/tb_PipelineCPU_Ha...	2	2
/tb_PipelineCPU_Ha...	4	4
/tb_PipelineCPU_Ha...	9	9
/tb_PipelineCPU_Ha...	10	10
/tb_PipelineCPU_Ha...	6	6
/tb_PipelineCPU_Ha...	8	8

List B

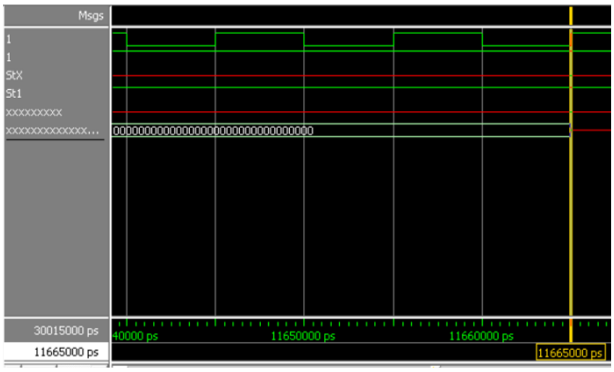
/tb_PipelineCPU_Ha...	2	2
/tb_PipelineCPU_Ha...	6	6
/tb_PipelineCPU_Ha...	1	1
/tb_PipelineCPU_Ha...	4	4
/tb_PipelineCPU_Ha...	8	8
/tb_PipelineCPU_Ha...	7	7
/tb_PipelineCPU_Ha...	3	3
/tb_PipelineCPU_Ha...	5	5
/tb_PipelineCPU_Ha...	10	10
/tb_PipelineCPU_Ha...	9	9

1번과 2번 모두 같은 결과를 확인할 수 있으며, 코드를 수정한 차이만 있습니다. 아래는 해당 코드에 관련한 cycle의 차이입니다.

1번 코드



2번 코드



1번 : 18535000ps

2번 : 11665000ps

1cycle에 10000ps 이므로 해당 값을 10000으로 나누면 1번은 **1853.5 cycle** 2번은 **1166.5 cycle**이 나옵니다. 확실히 cycle의 수가 많이 차이는데 해당이유는 쓸데 없는 instruction을 줄이는 동시에 stall을 줄일 수 있었고, sub를 통한 전체적인 loop를 줄였기 때문에 가능한 결과였습니다.

(2) 2번째, 3번째

```
main: addi $t0, $r0, 36
Loop1: addi $a0, $r0, baseaddress of list_a
       addi $b0, $r0, baseaddress of list_b
       add $t1, $r0, $r0
Loop2: lw  $t2, 0($a0)
       lw  $v2, 0($b0)
       lw  $t3, 4($a0)
       lw  $v3, 4($b0)
       add $c0, $t2, $v2
       NOP
       add $c1, $t3, $v3
       NOP
       NOP
       slt $t5, $c1, $c0
       NOP
       NOP
       bne $t5, $r0, Loop3
       NOP
       NOP
Swap1: addi $t1, $r0, 1
       sw  $t2, 4($a0)
       sw  $t3, 0($a0)
       sw  $v2, 4($b0)
       sw  $v3, 0($b0)
```

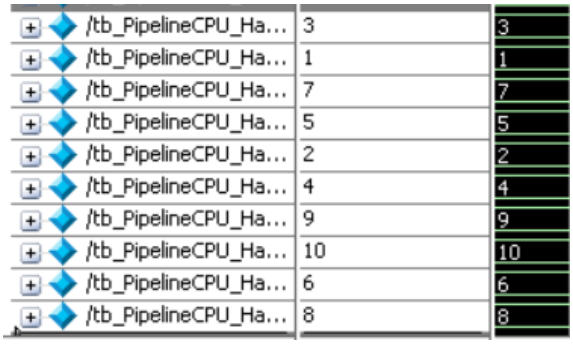
```
Loop3: lw  $t2, 4($a0)
       lw  $v2, 4($b0)
       lw  $t3, 8($a0)
       lw  $v3, 8($b0)
       add $c0, $t2, $v2
      NOP
       add $c1, $t3, $v3
       NOP
       NOP
       slt $t5, $c1, $c0
       NOP
       NOP
       bne $t5, $r0, Loop4
       NOP
       NOP
Swap2: addi $t1, $r0, 1
       sw  $t2, 8($a0)
       sw  $t3, 4($a0)
       sw  $v2, 8($b0)
       sw  $v3, 4($b0)
```

```
Loop4: lw  $t2, 0($a0)
       lw  $v2, 0($b0)
       lw  $t3, 4($a0)
       lw  $v3, 4($b0)
       add $c0, $t2, $v2
       NOP
       add $c1, $t3, $v3
       NOP
       NOP
       slt $t5, $c1, $c0
       NOP
       NOP
       bne $t5, $r0, continue
       NOP
       NOP
Swap3: addi $t1, $r0, 1
       sw  $t2, 12($a0)
       sw  $t3, 8($a0)
       sw  $v2, 12($b0)
       sw  $v3, 8($b0)
continue: addi $a0, $a0, 12
          addi $b0, $b0, 12
          NOP
          bne  $a0, $t0, Loop2
          NOP
          NOP
          bne  $t1, $r0, Loop1
          NOP
          NOP
End:
```

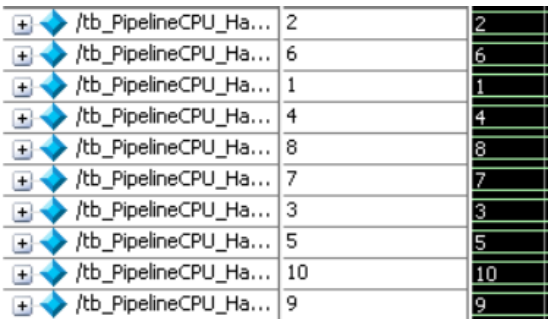
다음 코드는 2번째 코드를 기반으로 3번 unrolling하여 구현한 코드입니다. 계산 편의에 의해 3 loops unrolling을 구현하였고 보기와 마찬가지로 code size가 커진 것을 확인할 수 있습니다. 다음 코드의 동작 결과는 다음과 같습니다.

3번의 코드 결과

List_A

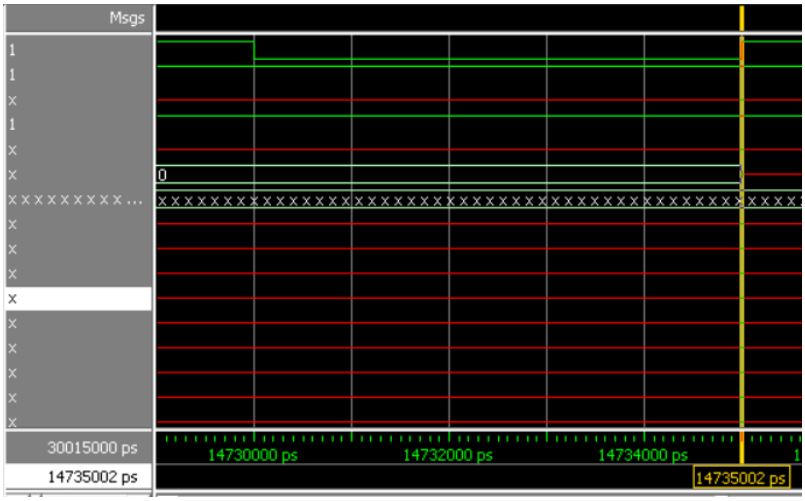


List_B



다음과 같이 결과가 똑 같은 모습을 확인할 수 있다. 알고리즘이 바뀌지 않고 단순히 같은 동작을 unrolling을 통하여 바꾼 것이다. Unrolling을 통하여 도출된 사이클은 아래와 같다.

Cycle



최종적인 Cycle은 **14735002/10000 = 1473.5** 가 나오게 된다. 2번째 코드보다 cycle이 긴 이유는 sub를 통해 전체 loop를 줄여준 2번째 코드와는 다르게 모든 루프를 돌고 unrolling만 시켰기 때문에 다음과 같은 문제가 발생한 것이다. 하지만 처음 코드와 비교했을 때를 생각해보면 확실히 많이 줄어든 것을 확인할 수 있다.

(3) 3번째, 4번째

3번에서 4번을 수행하면서 정말 많은 고민을 하였다 최적화를 해야 하지만 이미 2번의 코드가 최적화된 상태였으며, 최적화 한 코드를 단순히 unrolling했기 때문에 최적화를 하기 위해 코드를 바꾸기에는 무리가 있었다. 해당과제 자체가 스케쥴링 하는 것이 목적이기에 현재 코드에서 명령의 위치를 바꾸어 계산하고 싶었지만 능력에 무리가 있었다. 최종적으로 4번째 코드는 3번째 코드와 같은 코드로 제출하며 cycle은 같지만 바꿀 수 있는 instruction을 제시하고, 해당 머신 코드도 마찬가지로 동일하게 제출하겠다. 마지막 아이디어는 다음과 같다.

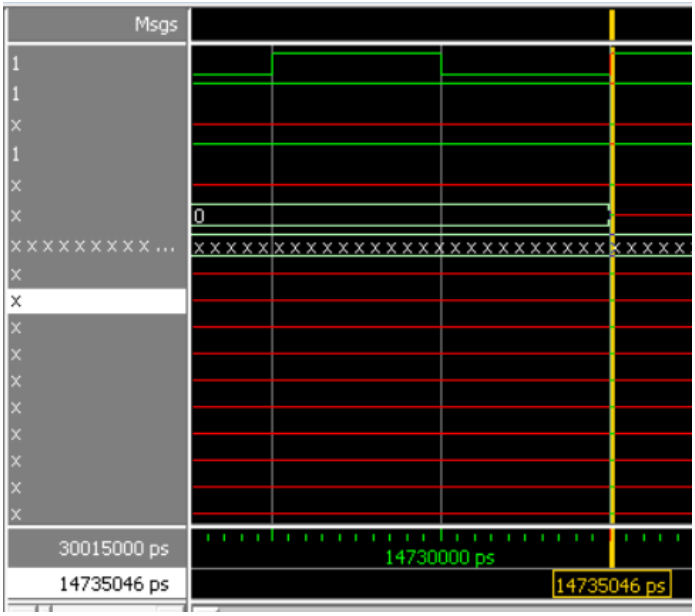
```
main: addi $t0, $r0, 36
Loop1: addi $a0, $r0, baseaddress of list_a
       addi $b0, $r0, baseaddress of list_b
       add $t1, $r0, $r0
Loop2: lw  $t2, 0($a0)
       lw  $v2, 0($b0)
       lw  $t3, 4($a0)
       lw  $v3, 4($b0)
       add $c0, $t2, $v2
       NOP
       add $c1, $t3, $v3
       NOP
       NOP
       slt $t5, $c1, $c0
       NOP
       NOP
       bne $t5, $r0, Loop3
       NOP
       NOP
Swap1: addi $t1, $r0, 1
       sw  $t2, 4($a0)
       sw  $t3, 0($a0)
       sw  $v2, 4($b0)
       sw  $v3, 0($b0)
```

```
Loop3: lw  $t2, 4($a0)
       lw  $v2, 4($b0)
       lw  $t3, 8($a0)
       lw  $v3, 8($b0)
       add $c0, $t2, $v2
       NOP
       add $c1, $t3, $v3
       NOP
       NOP
       slt $t5, $c1, $c0
       NOP
       NOP
       bne $t5, $r0, Loop4
       NOP
       NOP
Swap2: addi $t1, $r0, 1
       sw  $t2, 8($a0)
       sw  $t3, 4($a0)
       sw  $v2, 8($b0)
       sw  $v3, 4($b0)
```

```
Loop4: lw  $t2, 0($a0)
       lw  $v2, 0($b0)
       lw  $t3, 4($a0)
       lw  $v3, 4($b0)
       add $c0, $t2, $v2
       NOP
       add $c1, $t3, $v3
       NOP
       NOP
       slt $t5, $c1, $c0
       NOP
       NOP
       bne $t5, $r0, continue
       NOP
       NOP
Swap3: addi $t1, $r0, 1
       sw  $t2, 12($a0)
       sw  $t3, 8($a0)
       sw  $v2, 12($b0)
       sw  $v3, 8($b0)
continue: addi $a0, $a0, 12
          addi $b0, $b0, 12
          NOP
          bne  $a0, $t0, Loop2
          NOP
          NOP
          bne  $t1, $r0, Loop1
          NOP
          NOP
End:
```

다음은 3번 코드와 일치하며, 위치를 바꿀 수 있는 instruction에 대하여 제시한다. 화살표에 해당하는 add 명령은 2번째 화살표인 NOP에 바꾸어 넣을 수 있는 상태이다. 하지만 빠진 add 자리에는 다음 명령 때문에 NOP이 들어가기 때문에 자리를 바꾸어도 cycle은 동일한 문제가 생긴다. 그 후 나머지 loop2나 swap에서는 스케쥴링을 통한 cycle감소는 어렵다고 생각되며, 명령을 바꾸어 돌린 결과는 아래와 같다.

4번의 코드 결과



보는 것과 같이 코드의 위치를 바꾸어 보았지만 결론적으로 NOP이 하나 더 추가됨으로써 위 3번과 같은 결과를 가지고 있다.

최종적으로는 2번의 cycle이 가장 적게 나왔으며, 마지막 4번 같은 경우는 본인의 능력이 부족하여 다른 sub와 같은 방법이 아닌 스케쥴링만 해서 cycle을 줄이기에는 힘든 과제였다.

3. 문제점 및 고찰

이번 프로젝트는 베릴로그 code를 원하면 수정하고 어셈블리 코드를 스케쥴링하여 그에 대한 performance를 비교하는데 있었다. 1번) 원래 제공된 코드를 머신코드로 바꾸어 돌리고, 2번) 해당 코드를 스케쥴링하며, 3번) 스케쥴링된 코드를 기반으로 unrolling을 시행한다. 마지막으로 4번) unrolling된 코드를 또 한번 스케쥴링하여 각각의 cycle을 비교하여 실험하는 것이다. 이번 프로젝트의 문제점은 본인이 생각하였을 때 마땅히 스케쥴링이 가능한 코드가 없었다고 생각된다. 제공된 코드에서는 필요 없는 instruction이 많아 대부분 지우고 2번 과정을 수행하게 되었는데 중복적으로 사용된 레지스터도 생각보다 많지 않아 NOP의 개수도 예상한 것 보다 적게 나왔다. 결론적으로 실험을 하기 위해선 스케쥴링 필요한데 그러기 위한 본인의 실력이 부족하다고 생각되었다. 알고리즘을 바꾸어 구현하게 된다면, 다양한 방법으로 가능성이 있다고 생각되지만 이미 제공된 코드는 굉장히 깔끔하다고 생각되어 효율적인 실험을 하지 못했다고 생각된다.