컴퓨터 구조

Pipeline Architecture

학　　과: 컴퓨터정보공학부

담당교수: 이성원교수님

학　　번: 2017202087

성　　명: 홍 세 정

1. 실험 내용

Instruction : Pipeline Architecture에서 selection sort하는 코드를 harzrd가 생기지 않게 구현하고, 이 코드를 scheduling하여 코드 사이즈를 줄여 효율적으로 구현한다. Schedulling한 코드를 unrolling하여 코드 사이즈는 늘어나지만, cycle 수를 줄여 구현하고, 이 코드 또한 scheduling하여 코드 사이즈를 더욱더 줄일 수 있다.

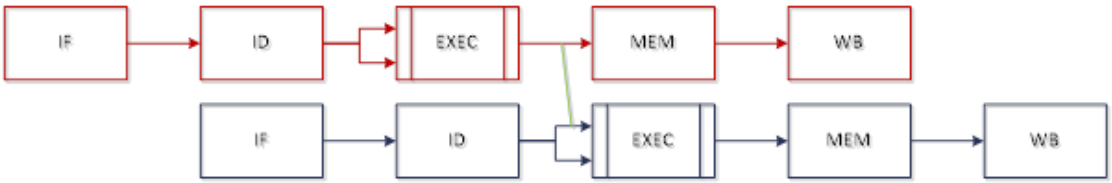
Harzard : speed를 줄이기 위해 pipeline한다. 이 때 instruction을 stage에 맞게 순차적으로 적용하여 처리되는 시간을 줄였을 때 몇 가지 문제가 발생하는 것이다. Instruction이 다음 clock cycle에서 정상적으로 수행되지 못하는 것을 나타낸다. Harzard의 종류에는 structural hazarde, data hazard, control harzard 3가지로 나눌 수 있다.

- Structural harzard

  하드웨어가 여러 명령들의 수행을 지원하지 않기 때문에 발생한다.(자원 충돌) Instruction memory와 data memory가 합쳐져서 하나의 메모리만 존재할 때 같은 시간에 memory access와 fetch가 동시에 접근하면서 일어나는 문제이다. 해결 방법은 memory를 나누는 방법이 있다. 또한, lw instruction 이후에 stall을 추가하여 명령어의 실행을 지연시킨다.

- Data harzard

  명령의 값이 현재 파이프라인에서 수행 중인 이전 명령의 값에 종속하여 발생한다. 예를 들면 add r1, r2, r3 add r4, r1, r2 명령어를 구현한다고 하였을 때 두번째 명령어에서 첫번째 줄의 결과 값을 이용한다. Single, multi에서는 한 명령어가 다 끝난 후에 다음 명령어를 실행하기 때문에 문제가 생기지 않는다. 하지만 이처럼 pipeline에서는



  결과 값이 저장되기 전에(EXEC state 실행 전)에 다음 명령어를 실행하기 때문에 원하는 결과값이 나오지 않게 된다. 그래서 이런 문제를 해결하기 위해서 위 그림처럼 연결을 해주어 값을 미리 전달할 수 있게 한다. 또, 첫번째 명령어와 두번째 명령어 사이에 NOP처럼 아무런 영향이 없는 명령을 넣어 값이 완전히 바뀐 후 두번째 명령어를 수행하는 방법을 사용할 수 있다.

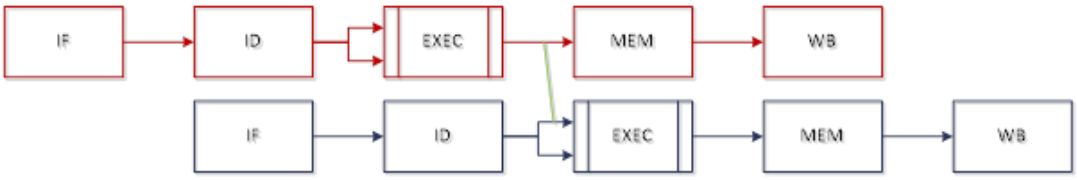
- Contraol harzard

  Jump, branch 명령어에 의해서 발생한다. 이동이 결정된 시점에 잘못된 명령이 파이프라인에 있기 때문에 발생한다. Jump나 barnch가 발생하고 그 결과가 판단되기 전까지 진행하 잘못된 실행을 하게 되어 harzard가 발생하게 된다. Jump나 branch 뒤에 stall을 추가하여 판단을 완료한 후 다음 명령어를 실행하도록 지연시키는 해결 방법이 있고, branch를 예측하는 방법이 있다. 예측하는 방법은 다소 오류가 있을 수 있다.

Harzard를 피하기 위한 방법에는 h/w forwarding, s/w code scheduling 방법이 있다.

- H/W-Forwarding

  lw 혹은 sw instruction으로 memory에서 나온 값을 가져오게 할 수 있다. 중간 값을 가져오는 방법이다. 구조를 바꾸어서 해결하는 방법이다. 이전 명령이 끝나기 전에 연산이 끝난 중간 값을 다음 명령어에 가져와 계산하는 방법이라고 할 수 있다.



- S/W-code scheduling

  명령어의 순서를 바꾸면서 문제를 해결한다. Scheduling을 해주지 않으면 NOP인 명령어를 사이에 집어넣어 순서를 뒤로 미뤄준다. 다만 이 방법은 cycle이 너무 많이 늘어나 효율적인 코드는 아니다 그래서, 명령어의 순서를 바꿔주어 NOP을 없애 주는 방식으로 scheduling을 하여 코드 사이즈를 줄여줄 수 있다. 여기서 명령어의 순서를 바꿔줄 때는 아무거나 바꿔주면 안되고, 아무런 영향을 받지 않는 코드만 순서를 바꿔서 NOP을 최소화하는 방법으로 scheduling할 수 있다.

1. 검증 전략, 분석 및 결과

이번 코드 구현에 사용된 명령어, register

t0 : 01000 t1 : 01001 t2 : 01010 t3 : 01011

t4 : 01100 t5 : 01101 t6 : 01110 t7 : 01111

s0 : 10000 s1 : 10001 s2 : 10010 s3 : 10011

s4 : 10100 s5 : 10101 s6 : 10110 s7 : 10111 r0 : 00000

addi = 001000 lw = 100011 beq = 000100 sll = 000000(function)

add = 100000 slt = 101010 j = 000010 sw = 101011 add = 100000

- Simulate the provided original assembly code.

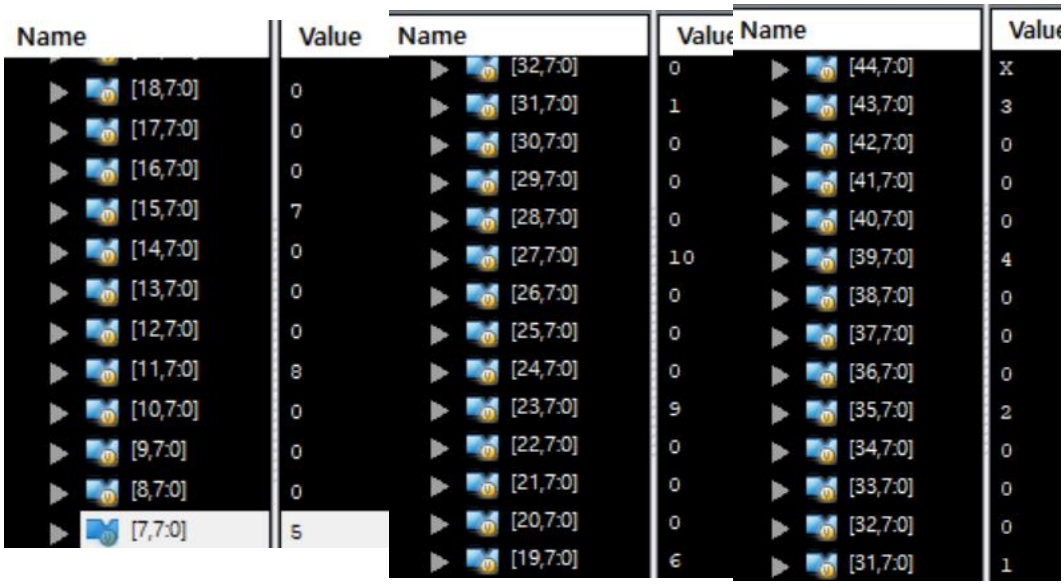
No(pc)	Machine code	assemdly	function
0(0)	00000000 00000000 00000000 00000000	Nop	Sort
1(4)	10001100 00010000 00000000 00000000	lw $s0, n	
2(8)	00000000 00000000 00000000 00000000	NOP	
3(12)	00000000 00000000 00000000 00000000	NOP	
4(16)	00100010 00001000 11111111 11111111	addi $t0, $s0, -1	
5(20)	00100001 00101001 00000000 00000000	addi $t1, $t1, 0	
6(24)	00100010 01110011 00000000 00000000	addi $s3, $s3, 0	
7(28)	00100000 00010011 00000000 00000100	addi $s3, $r0, baseaddress	
8(32)	00010001 00101000 00000000 00110110	beq $t0, $t1, END (+54)	LOOP1
9(36)	00000000 00000000 00000000 00000000	NOP	
10(40)	00000000 00000000 00000000 00000000	NOP	
11(44)	00100001 00101010 00000000 00000000	addi $t2, $t1, 0	
12(48)	00100000 00010001 00000000 00000100	addi $s1, $r0, baseaddress	
13(52)	00100000 00010010 00000000 00000100	addi $s2, $r0, baseaddress	
14(56)	00000000 00001010 01011000 10000000	sll $t3, $t2, 2	
15(60)	00000000 00000000 00000000 00000000	NOP	
16(64)	00000000 00000000 00000000 00000000	NOP	
17(68)	00000010 01001011 10010000 00100000	add $s2, $s2, $t3	
18(72)	00000000 00001001 01011000 10000000	sll $t3, $t1, 2	
19(76)	00000000 00000000 00000000 00000000	NOP	
20(80)	00000000 00000000 00000000 00000000	NOP	
21(84)	00000010 00101011 10001000 00100000	add $s1, $s1, $t3	

22(92)	00100001 00101100 00000000 00000000	addi $t4, $t1, 0	
23(96)	00000000 00000000 00000000 00000000	NOP	
24(100)	00000000 00000000 00000000 00000000	NOP	
25(104)	00100001 10001100 00000000 00000001	addi $t4, $t4, 1	
26(108)	00000000 00000000 00000000 00000000	NOP	
27(112)	00000000 00000000 00000000 00000000	NOP	
28(116)	00010001 10010000 00000000 00010110	beq $s0, $t4, Swap (+22)	LOOP2
29(120)	00000000 00000000 00000000 00000000	NOP	
30(124)	00000000 00000000 00000000 00000000	NOP	
31(128)	00100000 00010011 00000000 00000100	addi $s3, $r0, baseaddress	
32(132)	10001110 01010101 00000000 00000000	lw $s5, ($s2)	
33(136)	00000000 00001100 01101000 10000000	sll $t5, $t4, 2	
34(140)	00000000 00000000 00000000 00000000	NOP	
35(144)	00000000 00000000 00000000 00000000	NOP	
36(148)	00000010 01101101 10011000 00100000	add $s3, $s3, $t5	
37(152)	10001110 01110110 00000000 00000000	lw $s6, ($s3)	
38(156)	00000000 00000000 00000000 00000000	NOP	
39(160)	00000000 00000000 00000000 00000000	NOP	
40(164)	00000010 10110110 10111000 00101010	slt $s7, $s5, $s6	
41(172)	00000000 00000000 00000000 00000000	NOP	
42(176)	00000000 00000000 00000000 00000000	NOP	
43(180)	00010000 00010111 00000000 00000100	beq $s7, $r0, Continue(+4)	
44(184)	00000000 00000000 00000000 00000000	NOP	
45(188)	00000000 00000000 00000000 00000000	NOP	
46(192)	00100001 10001010 00000000 00000000	addi $t2, $t4, 0	
47(196)	00000010 01100000 10010000 00100000	add $s2, $s3, $0	
48(200)	00100001 10001100 00000000 00000001	addi $t4, $t4, 1	Continue
49(204)	00001000 00000000 00000000 00011100	J Loop2 (28)	
50(208)	00000000 00000000 00000000 00000000	NOP	
51(212)	00010001 00101010 00000000 00001000	beq $t2, $t1, Continue2(+8)	Swap
52(216)	00000000 00000000 00000000 00000000	NOP	
53(220)	00000000 00000000 00000000 00000000	NOP	
54(224)	10001110 01010101 00000000 00000000	lw $s5, ($s2)	
55(228)	10001110 00101111 00000000 00000000	lw $s7, ($s1)	

56(232)	00000000 00000000 00000000 00000000	NOP	
57(236)	00000000 00000000 00000000 00000000	NOP	
58(240)	10101110 01001111 00000000 00000000	sw $t7, ($s2)	
59(244)	10101110 00110101 00000000 00000000	sw $t5, ($s1)	
60(248)	00100001 00101001 00000000 00000001	addi $t1, $t1, 1	Continue2
61(252)	00001000 00000000 00000000 00001000	J Loop1 (8)	
62(256)	00000000 00000000 00000000 00000000	NOP	
63(260)			END
64(264)			

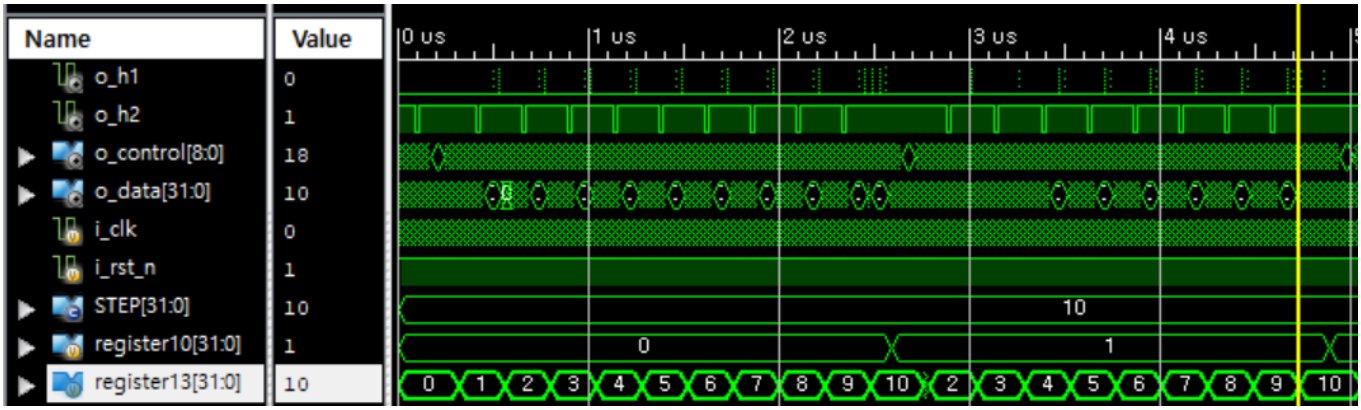
다음과 같이 machine code를 나타낼 수 있다.

기본적으로 제공된 assembly code에서 구현되도록 NOP만 추가하여 machine 코드로 바꾼 것이다.

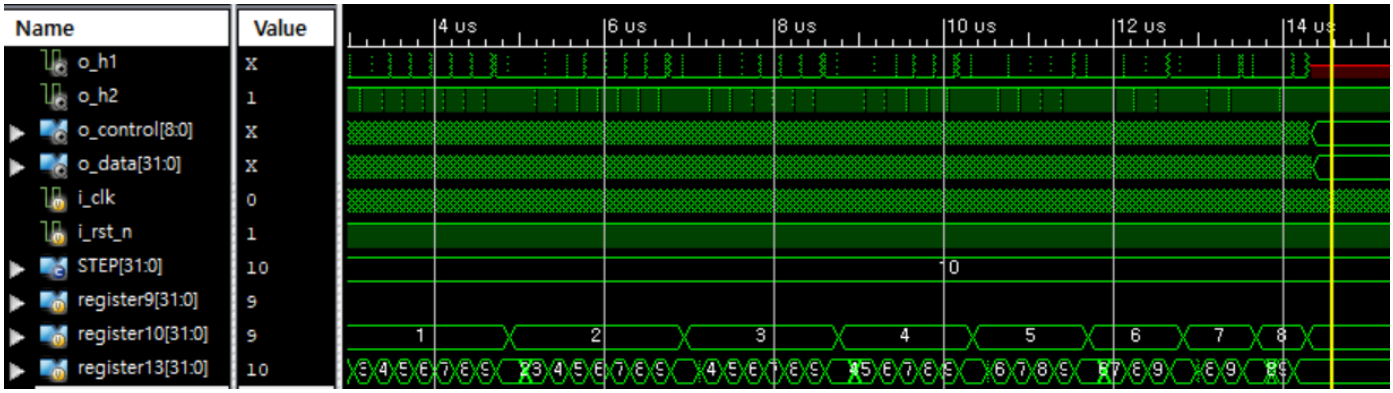


다음과 같이 값이 변경되서 저장되는 것을 확인할 수 있다.

Array[10] = {5, 8, 7, 6, 9, 10, 1, 2, 4, 3}; 이다.



j 값인 register($t4)의 값이 1씩 커지는 것을 확인하고 10일 때는 반복문을 빠져나가는 것을 확인할 수 있다.



I 값이 register($t1)의 값이 1씩 증가하고 9일 때 sorting을 끝내는 것을 확인할 수 있다. T1이 t0값이 같을 때 Loop1을 종료한다. T0의 값은 9, 값을 비교하여 9값을 가질 때 end로 이동하는 것을 확인한다.

총 code는 62줄이다. 총 cycle 1430 cycle이다

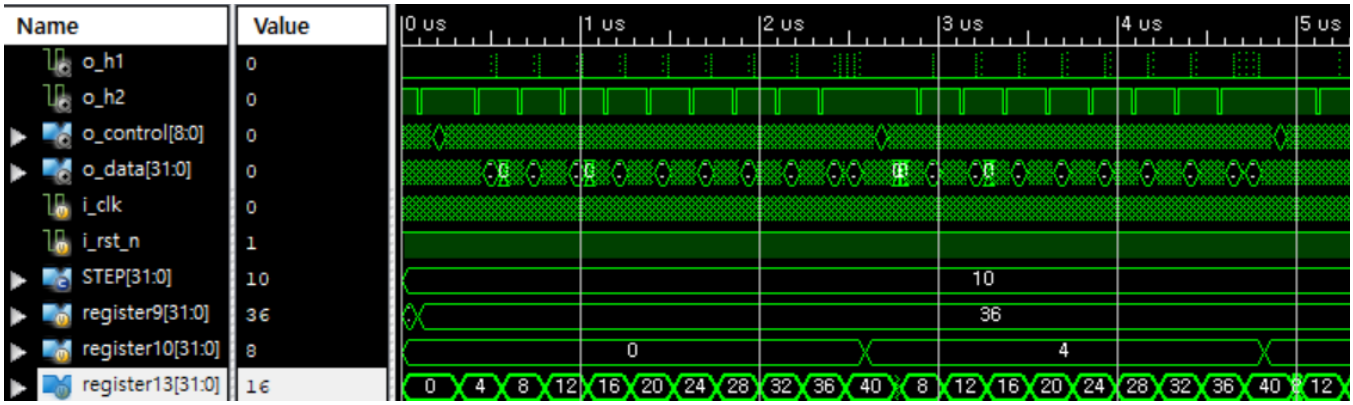
- Do code-scheduling

No(pc)	Machine code	assemdly	function
0(0)	00000000 00000000 00000000 00000000	Nop	Sort
1(4)	10001100 00010000 00000000 00000000	lw $s0, n	
2(8)	00100010 01110011 00000000 00000000	addi $s3, $s3, 0	
3(12)	00100001 00101001 00000000 00000000	addi $t1, $t1, 0	
4(16)	00100010 00001000 11111111 11111100	addi $t0, $s0, -4	
5(20)	00100000 00010011 00000000 00000100	addi $s3, $r0, baseaddress	
6(24)	00000000 00000000 00000000 00000000	NOP	
7(28)	00010001 00101000 00000000 00101010	beq $t0, $t1, END (+42)	LOOP1
8(32)	00000000 00000000 00000000 00000000	NOP	
9(36)	00000000 00000000 00000000 00000000	NOP	
10(40)	00100001 00101100 00000000 00000000	addi $t2, $t1, 0	
11(44)	00100001 00101010 00000000 00000000	addi $s1, $r0, baseaddress	
12(48)	00100000 00010001 00000000 00000100	addi $s2, $r0, baseaddress	
13(52)	00100001 00101100 00000000 00000100	addi $t4, $t1, 4	
14(56)	00000010 01001010 10010000 00100000	add $s2, $s2, $t2	
15(60)	00000010 00101001 10001000 00100000	add $s1, $s1, $t1	
16(64)	00010001 10010000 00000000 00010110	beq $s0, $t4, Swap (+22)	LOOP2
17(68)	00000000 00000000 00000000 00000000	NOP	
18(72)	00000000 00000000 00000000 00000000	NOP	
19(76)	00100000 00010011 00000000 00000100	addi $s3, $r0, baseaddress	
20(80)	00000000 00000000 00000000 00000000	NOP	
21(84)	10001110 01010101 00000000 00000000	lw $s5, ($s2)	
22(92)	00000010 01101100 10011000 00100000	add $s3, $s3, $t5	
23(96)	00000000 00000000 00000000 00000000	NOP	
24(100)	00000000 00000000 00000000 00000000	NOP	
25(104)	10001110 01110110 00000000 00000000	lw $s6, ($s3)	
26(108)	00000000 00000000 00000000 00000000	NOP	
27(112)	00000000 00000000 00000000 00000000	NOP	
28(116)	00000010 10110110 10111000 00101010	slt $s7, $s5, $s6	
29(120)	00000000 00000000 00000000 00000000	NOP	
30(124)	00000000 00000000 00000000 00000000	NOP	

31(128)	00010000 00010111 00000000 00000100	beq $s7, $r0, Continue(+4)	
32(132)	00000000 00000000 00000000 00000000	NOP	
33(136)	00000000 00000000 00000000 00000000	NOP	
34(140)	00100001 10001010 00000000 00000000	addi $t2, $t4, 0	
35(144)	00000010 01100000 10010000 00100000	add $s2, $s3, $0	
36(148)	00100001 10001100 00000000 00000100	addi $t4, $t4, 1	Continue
37(152)	00001000 00000000 00000000 00010000	J Loop2 (16)	
38(156)	00000000 00000000 00000000 00000000	NOP	
39(160)	00010001 00101010 00000000 00000111	beq $t2, $t1, Continue2(+7)	Swap
40(164)	00000000 00000000 00000000 00000000	NOP	
41(172)	00000000 00000000 00000000 00000000	NOP	
42(176)	10001110 00101111 00000000 00000000	lw $s7, ($s1)	
43(180)	10001110 01010101 00000000 00000000	lw $s5, ($s2)	
44(184)	00000000 00000000 00000000 00000000	NOP	
45(188)	10101110 01001111 00000000 00000000	sw $t7, ($s2)	
46(192)	10101110 00110101 00000000 00000000	sw $t5, ($s1)	
47(196)	00100001 00101001 00000000 00000100	addi $t1, $t1, 4	Continue2
48(200)	00001000 00000000 00000000 00000111	J Loop1 (7)	
49(204)	00000000 00000000 00000000 00000000	NOP	
50(208)			END
51(212)			

위에서 했던 machine 코드를 schduling을 하여 코드사이즈를 줄이고 효율적으로 구현하였다. 명령어의 순서를 바꿔 NOP을 최소화하는 방법으로 scheduling할 수 있었다. 또한, 중복되는 코드를 줄여서 코드 사이즈를 바꾸고, i와 j를 +4씩 증가시켜 주소 값을 더하기 위해 +4를 했던 과정을 없애고 i와 j를 통해 주소 값을 받아와 바로 대입하고, 비교할 수 있었다. 그래서 sll의 과정이 필요없어 없애 주는 방법으로 scheduling한 코드를 구현할 수 있었다.

전체적인 코드 사이즈는 많이 줄지 않았지만, LOOP문이 더 짧아져 반복하는 부분을 줄여주니 전체 clock수는 많이 주는 것을 확인할 수 있다.



다음과 같이 i와 j 모두 4씩 증가하고, j = 40일 때 반복문을 빠져나오고 I는 36일 때 반복문을 빠져나오는 것을 볼 수 있다.

총 코드는 52 줄로 scheduling 하기 전보다 12줄 정도 코드 사이즈가 줄어든 것을 확인할 수 있다. 그만큼 cycle도 줄어들게 되었다.

- Do loop-unrolling

  다음과 같은 방법은 최대한 branch와 jump를 사용하지 않도록, 즉 Loop문의 반복을 적게 하는 방법이다. 다음과 같이 언롤링하면 Loop문안에 코드는 길어지겠지만 그 Loop문을 반복하는 횟수가 줄어들고, branch와 jump하는 횟수를 줄여 cycle을 줄일 수 있게 되고 효율이 더 좋은 코드를 구현할 수 있다.

- Do code-scheduling

  Unrolling한 코드를 더 효율이 좋게 만들기 위해 scheduling 한다. 그래서 구현한 4개의 코드 중에서는 제일 효율이 좋고 cycle이 적은 것을 알 수 있다.

2. 문제점 및 고찰

이번 3차 프로젝트에서는 hazard를 공부하고 hazard를 해결하고 코드가 정상적으로 돌아가도록 machine 코드를 작성하였다. 이번 프로젝트를 하면서 아쉬운 점도 많았고, 새로 알게 된 내용도 많았다.

처음 구현하였을 때 Jump나 branch를 할 때 주소 값이 이동하고 싶은 줄의 주소 값, pc값을 입력해주었지만, 원하는 주소로 이동하지 않고 엉뚱한 곳으로 이동하여 오류가 나는 것을 보았다. 그 이유가 branch가 이동할 때 immediate에는 상대적인 값을 적어줘야 한다는 것을 알 수 있었다. 상대적인 값이라고 하면 이동하고 싶은 주소와 명령어의 주소의 주소 값 차이를 확인하고 그 차이만큼 이동해야 한다는 것을 알 수 있었다. 따라서 jump는 절대적인 주소 값, branch는 상대적인 주소 값이라 생각하고 구현할 수 있었다. 여기서 코드를 수정할 때 마다 값을 변경해줘야 하는 번거로움이 있었지만, 표에 정리하여 알아볼 수 있게 정리를 해, 값을 쉽게 변경할 수 있었다.

또한, 이번 과제를 하면서 하드웨어 동작의 검토하는 방법을 제대로 알 수 있었다. 전체 모듈의 output만으로도 어느 부분에서 문제가 발생하였는지, 값이 제대로 입력되었는지, 출력되었는지 알 수 없다. 그래서 알고 싶은 값을 모듈에서 드래그하여 원하는 값이 나왔는지 왜 이러한 값이 나왔는지 확인하여 어떤 부분에서 잘못 구현하였는지 확인할 수 있었다.