Computer Architecture

Assignment #3

담당 교수님 : 이성원 교수님

컴퓨터정보공학부

2018202013

정영민

■ 실험 내용

● 프로젝트 이론 내용

◆ Hazard

Hazard란 pipelining에서 다음 instruction이 다음 clock cycle에서 정상적으로 수행되지 못하는 것을 나타낸다. 보통 instruction과 cycle간 비율을 CPI라고 하고 이상적인 Pipeline은 CPI가 1인 구조를 취하는 데, Hazard가 발생할 시 CPI가 1보다 높아지는 경우가 발생할 수 있는 것이다.

Hazard의 경우 3가지가 있다. Structural hazard와 control hazard, data hazard이다.

Structural hazard의 경우 두 개의 모듈이 하나의 모듈로 동시에 signal을 보낼 때 서로 다른 모듈이 결과를 얻기 위해 충돌하는 것이다. 예시로 Load – Inst1 – inst2 – inst3의 명령어가 수행된다고 할 때, Load의 memory는 lw할 때 사용되고, inst3의 IF에서 fetch를 할 때 memory가 사용된다. 이렇게 되면 같은 모듈에 다른 두 instruction이 동시에 접근하는 것이기 때문에 hazard가 발생한다. 이러한 경우에는 memory를 두 개 두어 해결하면 된다고 교수님께서 말씀하셨다.

Control hazard의 경우는 branch or jump instruction의 경우, 해당 instruction이 jump를 할 때 이미 뒤에 이어져 fetch된 명령어들을 버려야 하는 문제점이 있다. 이로 인해 의도하지 않은 stall이 발생하여 결론적으로 제 성능을 나타내지 못하는 것이다. 이를 해결하는 대표적인 방법은 prediction이다. 보통 not-taken으로 predict한다.

Data hazard의 경우 RAW, WAW, WAR의 경우로 나뉜다. 이는 instruction간의 dependency때문에 발생한다. 교수님께서는 WAW나 WAR의 경우 stall로 해결할 수 있다고 하셨는데, RAW의 경우 다른 해결방법을 통하여 해결 가능하다고 말씀하셨다. 해결 방법은 Hardware적인 방법과 Software적인 방법이 있다.

◆ H/W-Forwarding

Forwarding은 Data hazard가 add r1, r2, r3 – sub r4, r1, r3 – and r6, r1, r7과 같은 instruction이 입력되었을 때, 뒤에 있는 instruction에서 사용할 r1의 값을 add instruction의 연산이 끝나기 전에 전달하는 것이다. 이가 가능한 이유는 다음과 같다. 보통 instruction이 수행되었다고 하는 것은 값이

register에 쓰고 난 이후가 되는데, 사실 우리가 원하는 결과는 ALU를 거치고 나온 직후의 결과이다. 즉 EX에서 ALU연산을 한 값을 바로 이후 instruction에 전달하면 해당 값을 이용하여 instruction을 수행할 수 있는 것이다. 결과가 지나갈 길을 만들자는 뜻에서 Forwarding이라고 하는데 이걸로 hazard를 해결할 수 있다. 이 방식은 IF ID EX MEM WB과 같은 단계의 사이에 있는 register에서 값을 저장하고 있다가 전달하게 된다.

- **S/W-code scheduling**

  소프트웨어적으로 data hazard를 피하는 방법은 code scheduling이 있다. 이는 nop라는 명령어로 처리하는 것이다. 그 외에도, instruction들의 순서를 바꿈으로써 stall을 줄일 수 있기 때문에 최대한 효율적으로 instruction의 실행 순서를 배치하는 것이 중요하다.

■ 검증 전략, 분석 및 결과

● 구현한 Assembly code

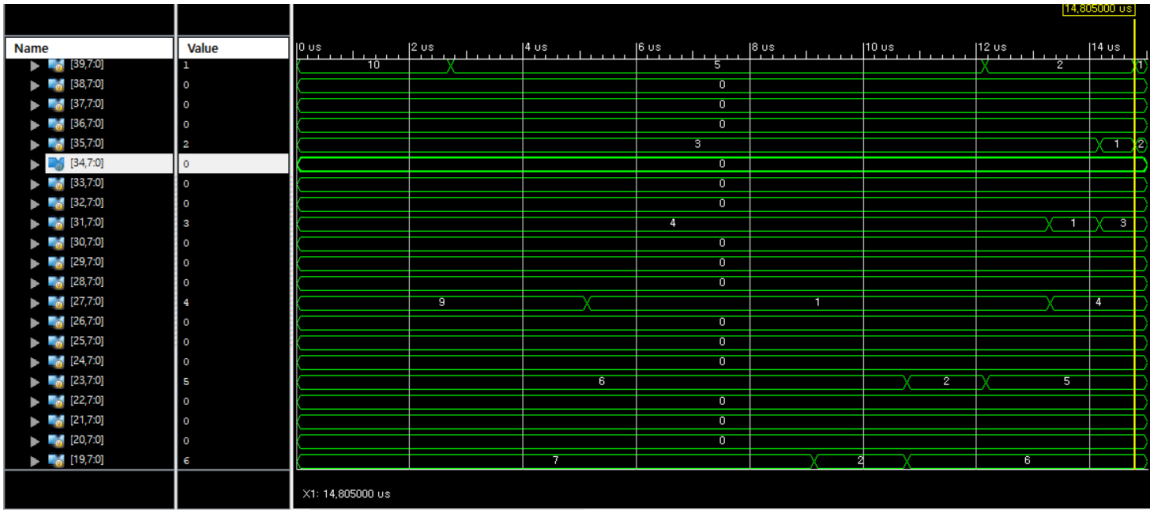
```
Sort:   NOP
        addi $s0, $s0. 40                       # n = 10, s0 = n * 4
NOP 1
        addi $t1, $t1, 0                        # i = 0
        addi $t0, $s0, -4                       # t0 = (n-1) * 4
NOP 2
LOOP1:  beq $t0, $t1, END                       # if i = n-1, goto END
NOP 2
        addi $t3, $t1, 0                        # t3 = min_idx = i
        addi $t2, $t1, 4                        # t2 = t1 + 4 (j = i + 1)
        addi $s1, $t1, 4                        # s1 = i + 1
        addi $t4, $t1, 4                        # t4 = min_idx1 = i + 1
        addi $t6, $t1, 8                        # t6 = t1 + 8 (j1 = i + 2)
        addi $s2, $t1, 8                        # s2 = i + 2
        addi $t5, $t1, 8                        # t5 = min_idx2 = i + 2
        addi $t7, $t1, 12                       # t7 = t1 + 12 (j2 = i + 3)

LOOP2:  beq $s0, $t2, Swap                      # if j == n => Exit
NOP 2
        lw $s5, ($t3)                           # s5 = arr[min]
        lw $s6, ($t2)                           # s6 = arr[j]
NOP 2
        slt $s7, $s5, $s6                       # if arr[min] < arr[j], s7 = 1
NOP 2
        beq $s7, $r0, Continue                  # if s7 == 0,  goto continue
NOP 2
        addi $t3, $t2, 0                        # min = j

Continue:addi $t2, $t2, 4                       # j++
NOP 1
        j LOOP2

Swap:   beq $t3, $t1, LOOP2_1           # if min == i goto Continue2
NOP 2
        lw $s5, ($t3)                   # s5 = arr[min]
        lw $s7, ($t1)                   # s7 = arr[i]
NOP 2
        sw $s7, ($t3)                   # arr[min] = s7
        sw $s5, ($t1)                   # arr[i] = s5
```

```
LOOP2_1:        beq $s0, $t6, Swap_1                    # if j1 == n => Exit
NOP 2
        lw $s5, ($t4)                           # s5 = arr[min1]
        lw $s6, ($t6)                           # s6 = arr[j1]
NOP 2
        slt $s7, $s5, $s6                       # if arr[min1] < arr[j1], s7 = 1
NOP 2
        beq $s7, $r0, Continue_1                        # if s7 == 0,  goto continue
NOP 2
        addi $t4, $t6, 0                        # min1 = j1

Continue_1:addi $t6, $t6, 4                     # j1++
NOP 1
        j LOOP2_1



Swap_1: beq $t4, $s1, LOOP2_2           # if min1 == i+1 goto Continue2
NOP 2
        lw $s5, ($t4)                   # s5 = arr[min1]
        lw $s7, ($s1)                   # s7 = arr[i+1]
NOP 2
        sw $s7, ($t4)                   # arr[min1] = s7
        sw $s5, ($s1)                   # arr[i+1] = s5


LOOP2_2:        beq $s0, $t7, Swap_2                    # if j2 == n => Exit
NOP 2
        lw $s5, ($t5)                           # s5 = arr[min2]
        lw $s6, ($t7)                           # s6 = arr[j2]
NOP 2
        slt $s7, $s5, $s6                       # if arr[min2] < arr[j2], s7 = 1
NOP 2
        beq $s7, $r0, Continue_2                        # if s7 == 0,  goto continue
NOP 2
        addi $t5, $t7, 0                        # min2 = j2

Continue_2:addi $t7, $t7, 4                     # j2++
NOP 1
        j LOOP2_2

Swap_2: beq $t5, $s2, Continue2         # if min2 == i+2 goto Continue2
NOP 2
         lw $s5, ($t5)                  # s5 = arr[min2]
         lw $s7, ($s2)                  # s7 = arr[i+2]
NOP 2
         sw $s7, ($t5)                  # arr[min2] = s7
         sw $s5, ($s2)                  # arr[i+2] = s5

Continue2: addi $t1, $t1, 12                    # i+= 3
NOP 1
         j LOOP1

END:
```

이는 Loop-unrolling한 코드를 다시 code scheduling한 Assembly code이다. NOP 옆의 숫자는 NOP의 개수를 나타낸다. Unroll 3 loops이기 때문에 Loop2, Continue, Swap이 각각 3번씩 나오는 것을 확인할 수 있다. Sort부터 Loop1까지는 변수 초기화이며 Loop1부터 Loop2전까지는 반복문에서 사용할 min_idx변수들과 j변수 초기화이다. Loop2부터 Continue까지는 Memory에 저장된 값을 Load한 후에 그 값을 비교하여 min_idx의 값을 변경할 지에 대한 코드이다. Continue는 j값을 증가시키고 다시 j변수에 해당하는 반복문으로 돌아가도록 한다. Swap에서는 min_idx가 i값과 비교하여 값이 바뀌었는지 확인하고 바뀌었을 경우 해당 index의 값을 swap한다. Continue2는 i값을 증가시키고 i변수에 해당하는 반복문으로 돌아가도록 한다.

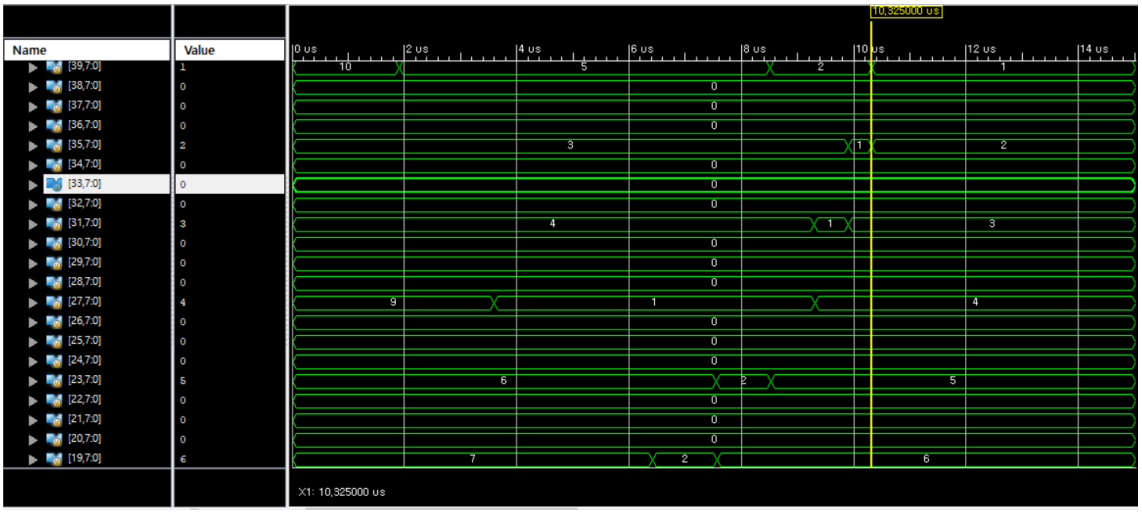
- **시뮬레이션 결과**

a. 기존 어셈블리 코드 시뮬레이션



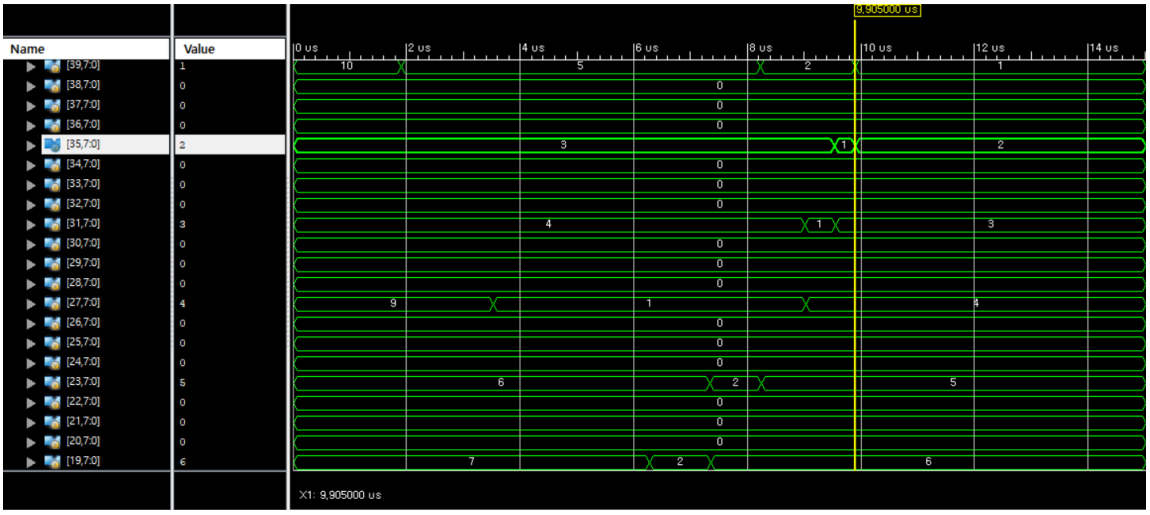
이는 맨 처음 코드를 시뮬레이션한 결과이다. 해당 값은 U3_MEM의 U0_DM의 메모리 값을 출력시킨 결과이다. 결과를 확인하면 10부터 1까지 순서대로 정렬이 된 것을 확인할 수 있으며, 정렬에 걸린 시간은 14.805us이다. 단위는 ns이므로 수정하면 14805ns이고, 15ns부터 동작을 수행하였으므로 실제 정렬에 걸린 시간은 14790ns이며, 이는 1479cycle이 소모된 것을 확인할 수 있다.

b. 기존 코드를 code scheduling하여 재배치된 코드 시뮬레이션



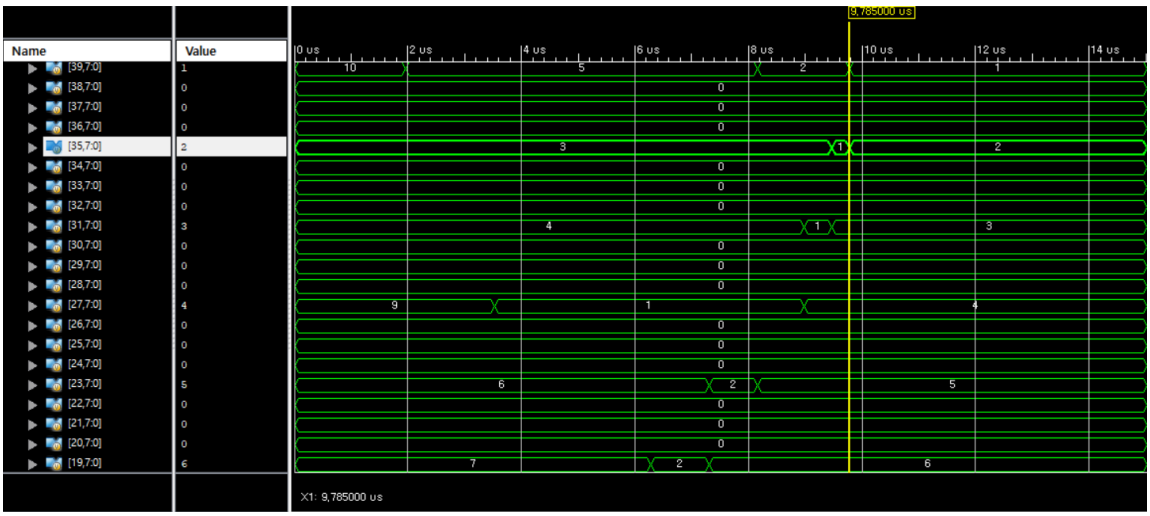
이는 code scheduling을 하여 재배치한 코드를 시뮬레이션한 결과이다. 이 또한 메모리를 확인하여 제대로 정렬이 된 것을 확인하였으며 정렬하는 데에 걸린 시간이 10.325us이다. 이는 10325ns이고 실제 정렬에 걸린 시간은 10310ns이므로 1031cycle이 소모된 것을 확인할 수 있다.

c. 재배치된 코드를 loop-unrolling하여 시뮬레이션



이는 재배치된 코드를 loop-unrolling하여 시뮬레이션한 결과이다. 제대로 정렬이 되는 것을 확인하였으며, i 변수 반복문에서 loop-unrolling을 하였다. Unroll 3 loops를 하였으며, 정렬하는 데에 걸리는 시간은 9.905us 즉 9905ns이며 15ns를 빼면 9890ns이므로 989cycle이 소모된 것을 확인할 수 있다.

d. Loop-unrolling된 코드를 다시 스케쥴링하여 시뮬레이션



Loop-unrolling된 코드의 code-scheduling을 한 후 시뮬레이션을 한 결과이다. 정렬하는 데에 걸리는 시간은 9785ns이고 977cycle이 소모된 것을 확인할 수 있다.

■ 문제점 및 고찰

● 프로젝트 내용 전체 정리 및 고찰

이번 프로젝트의 내용은 IF - ID – EX – MEM – WB의 동작 순서를 갖는 pipelining에서 여러 instruction을 입력 받았을 때 생기는 Data Hazard에 관련하여 처리를 하는 프로젝트였다. Data Hazard를 처리하는 방법 중 S/W로 처리하는 방식인 Code Scheduling 방식을 사용하였으며 Code Scheduling과 Loop-unrolling 방식까지 이용하여 입력 받은 여러 instruction을 수행하는 데에 걸리는 cycle 수를 최대한 줄이는 것이 목표이다. 이때 입력 받은 instruction들은 Memory에 무작위로 저장되어 있는 값을 정렬하도록 동작한다.

이번 프로젝트는 code scheduling 이전에 selection sort 알고리즘을 machine code로 바꾸어 실행하였을 때 제대로 실행되지 않아서 어려움을 많이 겪었다. 맨 처음에 주어진 assembly code가 아닌 알고리즘을 유지한 다른 방식으로 code를 구현하여 이를 machine code로 바꾸어 입력하였는데, 정렬이 되지 않아 이를 포기하고 주어진 예제코드를 사용하였다. 예제 코드의 i++부분과 j++부분이 오류가 있어 이를 고친 후 NOP를 적절하게 사용하여 testbench를 하였는 데, 이때도 정렬이 되지 않았다. 왜 정렬이 제대로 되지 않았는 지에 대해 확인하여 보니 우선 branch의 명령어에 주소 값을 제대로 전달하지 않았던 부분이 있었고, 또 공지사항에 바뀐 file을 확인하지 못하여 잘못된 파일로 프로젝트를 진행하고 있던 문제점도 있었다. 문제점을 확인한 후 code scheduling에서 이전에 구현한 code를 사용할 수 있게 되었다. Loop-unrolling을 할 때, 기존 instruction이 40개정도였다면 unrolling을 한 후 instruction이 103개가 나와서 오류가 생길 때 코드의 어떤 부분이 문제인 지 찾기가 힘들었다.