컴퓨터구조 Project #3

(Pipeline Architecture)

수업명: 컴퓨터구조

교수님: 이성원 교수

수업 시간: 월 3교시, 수 4교시

소속: 컴퓨터정보공학부

학번: 2018202074

이름: 김상우

Introduction

위 프로젝트에서는 bubble sort를 진행하는 어셈블리 코드를 받게 된다. 단, 해당 코드는 제공된 실행환경에서 명령어당 5개의 stage로 분리된 pipeline을 베이스로 움직이게 된다. 기존에 제공된 M_TEXT_SEG에는 이러한 점을 고려하여 NOP을 각 명령어당 4개씩 부여, 명령어가 끝날 때까지 다음 명령어가 실행환경내로 들어올 수 없게 만들었다.

그러나 이는 pipeline의 장점을 살리지 못하는 코드이다. 우리는 이러한 점을 바탕으로 Nop을 최대한 제거하여 효율을 올리고 forward를 이용해 더욱 효율을 상승시켜야 한다.

제공되는 bubble sort를 진행하는 코드의 기본적인 모습은 다음과 같다.

```
.text
main:
        lui     $6,0x0000
        ori     $6,$6,0x2000
        addi    $2,$0,0x10
        addi    $3,$0,0

L1:
        sub     $4,$2,$3
        addiu   $4,$4,-1
        addi    $5,$0,0

L2:
        sll     $7,$5,2
        addu    $7,$6,$7
        lw      $8,0($7)
        lw      $9,4($7)
        slt     $1,$8,$9
        bne     $1,$0,L3
        add     $10,$8,$0
        add     $8,$9,$0
        add     $9,$10,$0
        sw      $8,0($7)
        sw      $9,4($7)

L3:
        addiu   $5,$5,1
        slt     $1,$5,$4
        bne     $1,$0,L2

        addiu   $3,$3,1
        slt     $1,$3,$2
        bne     $1,$0,L1

        break

.data
        .word   31028
        .word   16610
        .word   12937
        .word   7525
        .word   25005
        .word   17956
        .word   23964
        .word   13951
        .word   3084
        .word   23696
        .word   3881
        .word   11872
        .word   24903
        .word   16843
        .word   25957
        .word   25086

```

위 코드를 요약하면 다음과 같다.

Lui&ori를 통해 레지스터 6에 값 0x00002000 저장.

Addi를 통해 register 2,3에 0x10(input되는 값의 수), 0x00저장

L1:레지스터 4에 (레지스터2의 값)-(레지스터3의 값)의 값을 넣는다.(sub사용)

레지스터 4의 값을 1 뺀다.(addiu사용), 레지스터 5에 0을 넣는다.(addi 사용)

L2:register5의 값을 shift left 2하여 register 7에 넣는다.(sll이용)

Addu를 이용해 register7에 register6값을 더해 memory 접근 주소를 만든다.

Lw를 이용해 레지스터 8,9에 0($7) , 4($7) 주소의 메모리 값을 넣는다.

SLT를 이용해 register 8과 9의 값을 비교하여 결과를 레지스터 1에 저장한다.

BNE를 이용해 register 1과 0을 비교하여 둘이 다를 경우(register 8>9) L3로 이동한다.

Add를 이용 register 8값을 register 10에 저장한다.

ADD를 이용 register 9값을 register 8에, register 10값을 register 9에 저장한다.

Sw를 통해 register 8, 9의 값을 메모리의 0($7) , 4($7) 주소에 넣는다.

L3:register5에 값을 1 더하고 register 5와 4값을 비교하여 register1에 결과를 저장한다.

Register1과 0을 비교하여 다를 경우(register 5<register 4(내부 반복문)) L2로 이동한다.

Register3에 1을 더하고 register 3과 2값을 비교하여 register1에 결과를 저장한다.

Register1과 0을 비교하여 다를 경우(register 3<register 2(외부 반복문)) L1로 이동한다.

Break를 통해 해당 코드를 종료한다.

위의 정리를 바탕으로 우리는 변수 $3을 이용해 $2값만큼 반복하는 L1과 변수 $5를 이용해 $4($2-$3)만큼 반복하는 L2를 기준으로 하는 2중 반복문을 실행한다.

L2내부에서는 $6에 해당하는 주소에서부터 &5번째의 해당하는 값과 그 다음 번째의 값을 비교하여 뒷 값이 더 클 경우 둘의 값을 바꾸는 코드가 존재한다.

L1은 L2를 ($2-$3)만큼 반복시킨다.

L1은 전체 코드와 L3에 의해 $2만큼 반복된다.

-이론

위 프로젝트는 pipeline을 사용한다. Pipeline은 다수의 명령어를 사용하는 부분을 기준으로 stage로 나눠 동시에 처리하게 하여 효율을 높여주는 효율적인 수단이다. 이때 같은 부분을 2 이상의 명령어가 사용하게 될 경우 같은 부분에서 다수의 동작이 일어나는 Structural hazards가 발생할 수 있다. 이 때문에 하드웨어는 각 stage에 대해서 확실하게 분리하여 이를 해결해야 한다.(한개의 stage에는 1개 이하의 명령어) 이 외에도 dependency에 의해 hazard가 발생할 수 있다.

우선 Data dependency에 의해 Data hazard가 발생할 수 있다. 여기서 Data dependency란, 컴퓨터의 특성상 이전에 작업한 결과를 다시 사용하는 경우가 발생한다.(Register에 결과 저장 후 불러와 명령어에 활용하는 경우) 이때 명령어의 순서 등에 의해 원하는 Data를 불러오지 못하는 경우 Data hazard가 발생한다. Data hazard의 경우 다음과 같은 것들이 존재한다.

WAW(Write After Write): 병렬처리로 인해 다수의 값이 한 곳에 적히는 경우 마지막에 입력된 값이 이전 값들을 덮어 이전 결과가 없어지는 상황이 생기게 된다.

WAR(Write After Read): 병렬처리로 인해 한 값이 Read와 Write의 목표로서 동시에 설정될 경우, Read에서 읽을 값이 이전 값인지 새로 Write된 값인지 알 수 없다.(thread 종료 시간에 따라 다름)

RAW(Read after Write): 값이 들어와도 저장(Write back)에는 시간이 걸린다. 이 작업이 끝나기 전에 Read를 실행할 경우 원하는 값이 아닌 이전값을 불러오게 되어 Hazard가 발생한다.

WAW와 WAR는 Write되는 대상을 동시에 Read나 Write의 대상으로 하지 않도록 하여, RAW의 경우 Stall, bublle삽입, Forward를 통한 저장전 미리 값을 가져오기 등을 이용해 해결이 가능하다.

또한 Control Dependency에 의해 Control hazard가 발생할 수 있다. Control dependency란 조건에 의해 다음 명령어 등의 수행여부가 결정되는 것으로 pipeline의 경우 Branch등의 명령어의 이동이 필요한 경우, 이가 ID에서 작업이 일어나게 되므로 1 stage를 진행할 시간이 필요하게 된다. 이때 이 곳에 명령어가 있을 경우 이를 실행하게 되며, 이 때문에 Control dependency와 관계없이 무조건 일어나게 되고 이 값이 예상치 못하거나 조건부 실행일 때 Contorl hazard가 발생하게 된다. 이는 stall로 ID까지의 시간을 주거나 공통으로 실행되는 명령어가 있다면 이를 삽입하여 동작 시키거나(Delayed Branch)혹은 Predict를 통해 해당 명령어를 제거해 주는 등의 방법으로 방지할 수 있다.

우리가 사용하게 될 Forwarding은 Stage 경계에 각각 EX/MEM, MEM/WB라는 부품을 추가하고 이들로 부터 값을 받아 EX stage의 ALU의 input으로 가는 값으로 받을 수 있게 설계하여 진행한다. 이들의 값을 쓸지 쓰지 않을지를 판단하기 위한 Mux와 signal도 준비되어야 한다. 해당 MUX를 선택하는 Signal의 경우 forwarding unit이 현재 명령어의 RF,RD의 값과 받아오는 stage에서의 예상 register를 받아 이를 바탕으로 EX stage의 ALU input을 선택할 수 있게 해야 할 것이다.

(S/W-coding에 대해선 설계 파트에서 설명)

-설계

Nop을 제거하기 이전 각 명령어들이 사용하게 되는 5stage는 IF, ID, EXEC, MEM, WB라고 정의된다. 해당 코드에서 사용되는 명령어들은 다음과 같다.

Lui, ori, addi, sub, addiu, sll, lw, slt, bne, add, sw, break

Rtype과 itype에 해당하는 lui, ori, addi, sub, addiu, sll, slt, add의 경우 각 stage에서 다음과 같은 부분들을 주로 사용하게 된다.,

IF-Instruction memory&PC ID-Registers, EXEC-ALU, MEM-(사용하지 않는다), WB-Registers

Sw의 경우 다음과 같다.

IF-Instruction memory&PC, ID-Registers, EXEC-ALU, MEM-Memory, WB-no used

Lw의 경우 다음과 같다.

IF-Instruction memory&PC, ID-Registers, EXEC-ALU, MEM-Memory, WB-Registers

Bne의 경우 다음과 같다.

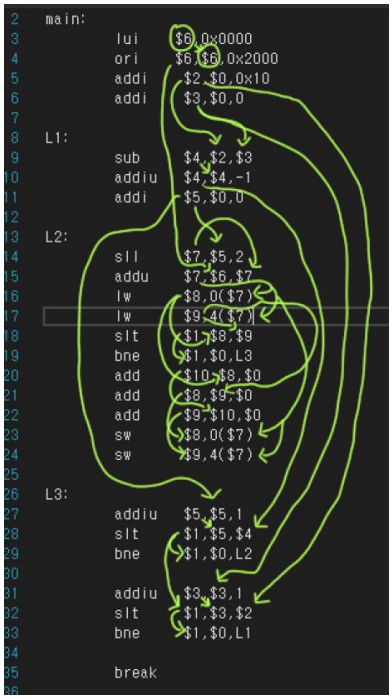
IF-Instruction memory&PC, ID-Registers, EXEC-ALU&PC MEM-no used, WB-no used

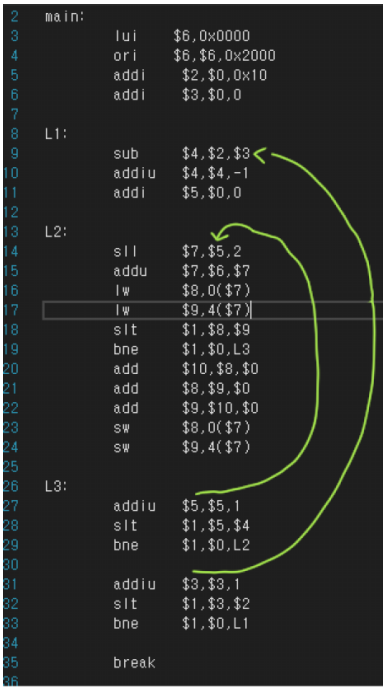
위 정보들을 바탕으로 코드에서 structural hazard가 발생하지 않게 조율할 수 있을 것이다.

또한 Data hazards의 경우 값이 변경되는 register가 변경을 위한 명령어를 받고 Write Back이 이뤄질 때까지 바뀌지 않는다는 점을 이용해 해결해야 할 것이다. 만약 Write Back이 이뤄지기 전에 해당 Register를 호출할 경우 Data hazards가 발생, 변경 이전의 값을 호출하게 될 것이다.

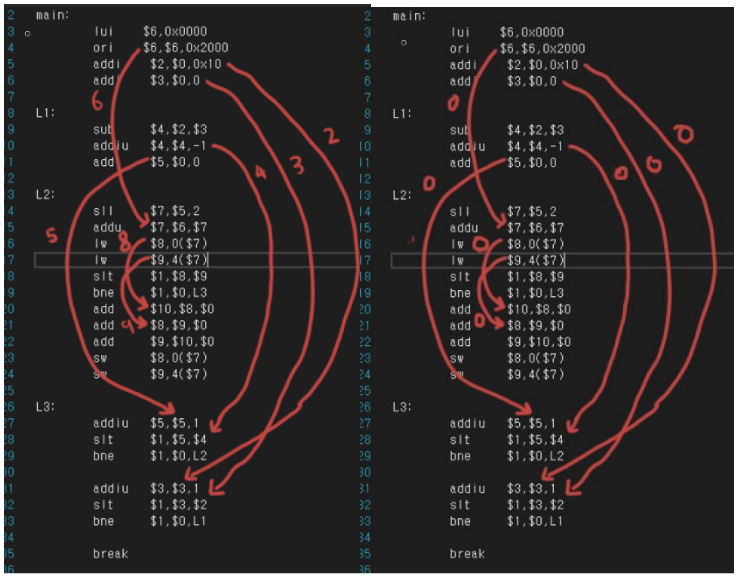
또한 Control hazards의 경우 해당 코드에서 Branch관련 명령어(Bne)를 수행하는 데 있어서 발생할 우려가 있다. 이는 BNE를 수행할 때 수행여부를 ID에서 결정하기 때문에 이에 대한 수행이 끝나고 현재 위치를 변경하기 전에 명령어가 들어오면 이를 수행하여야 하기 때문에 발생한다. 이는 BNE이후 ID를 위한 clock을 마련할 경우 해결할 수 있을 것이다.

2.1-1-1(Data dependency)



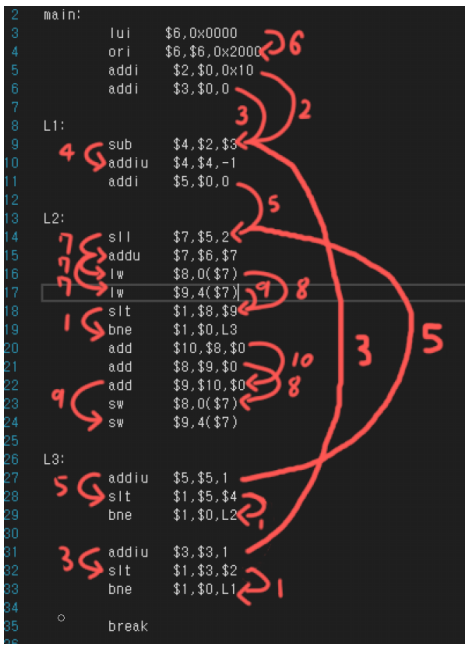


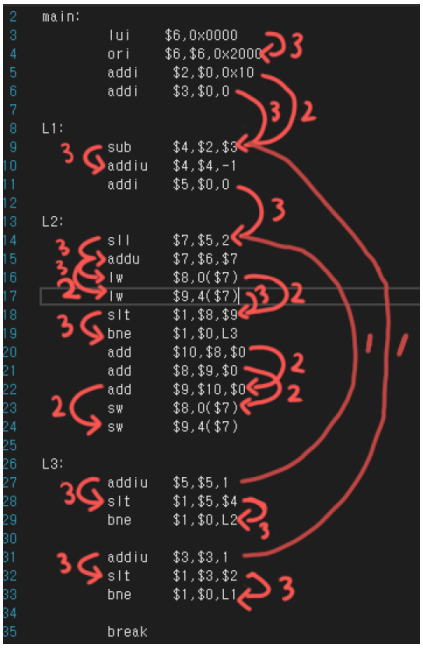
위는 Data를 받아오게 되는 모든 경우를 화살표로 나타낸 것이다. 왼쪽은 반복문에 걸쳐 진행되는 경우와 반복없이 진행될 경우, 우측은 첫 반복문을 제외한 경우에 추가적으로 발생하는 경우에 대해 기록하고 있다. 여기서 모든 명령어는 Data hazard를 막기 위해 Write Back할 때 까지의 시간이 필요한 데, Write Back은 IF-ID-EXEC-MEM-WB의 과정을 거쳐 일어나므로 Forwarding을 하지 않았다면 저장을 위해 Register값의 결과가 나오는 시점과 해당 Register가 사용되는 구간 사이에는 **3개의 stage만큼의 시간**이 필요하다. 이때 stall은 내부에 명령어를 넣는 걸로 대체 할 수 있는 것을 바탕으로 다음과 같이 분류할 수 있다.



(사이에 명령어가 3개 이상인 경우, stall이 더 필요없다.)

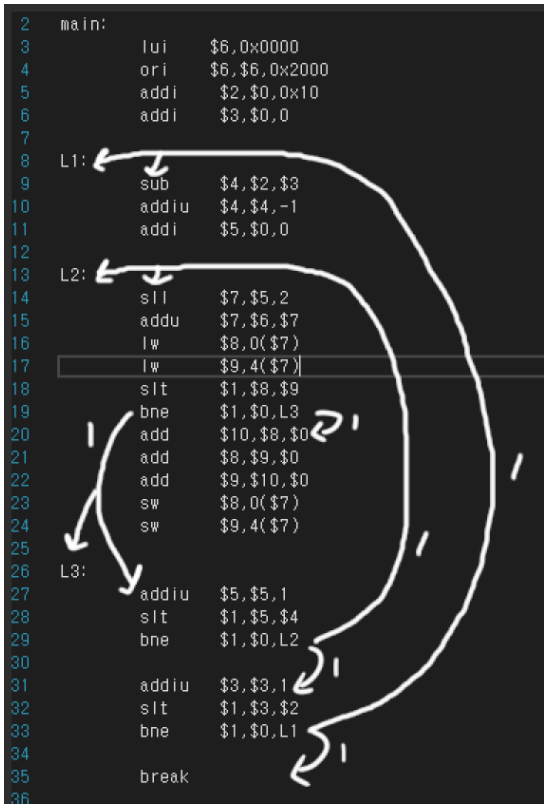
화살표에 적힌 숫자는 data가 오가는 register(좌)와 필요한 stall수(우)이다.





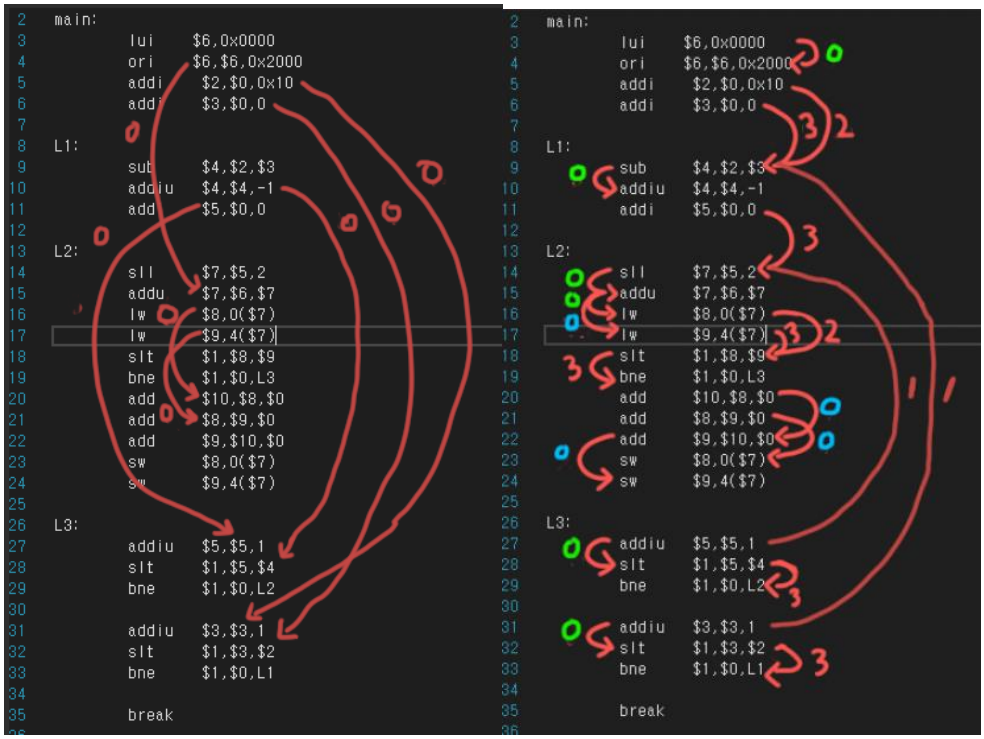
위 그림들은 사이에 명령어가 3개 미만이며 값들을 받아오는 경우를 화살표로 나타낸 것이다. 각각 Data dependency가 발생하는 register(좌)와 각 명령어에 대해 필요한 stall의 수이다.(우) stall의 수의 경우 중간에 명령어가 있는 경우에는 stall을 하는 대신 해당 명령어로 대체할 수 있으므로 이를 바탕으로 필요한 stall 수를 정리하면 오른쪽 그림과 같은 값이 나온다.

2.1-1-2(Control dependency)



위는 control dependency에 대해 필요한 stall의 수를 나타낸 것이다. Branch의 경우 Branch의 유무를 결정하기 위해 명령어가 호출된 후 1clock의 stall이 필요하다.(ID(2번째 clock)에서 유무를 결정한다.) 이는 조건 확인의 결과에 의한 Branch의 실행 유무와 관계 없다. (다만 1 stage가 지난 후 가리키는 명령어는 다르게 될 것이다.) 그 때문에 각 bne에서 1번의 NOP(stall)이 필요할 것이다.

2.1-1-3(resolved by data forwarding to ALU)



위는 기존에 확인되었던 Data dependency들을 forward를 통해 제거할 수 있는 만큼 제거한 것이다. 기존에 이미 Data hazard가 없어 stall이 필요없는 경우는 forward를 통해 제거할 필요가 없다.(좌) 기존에 Data hazard를 막기위해 stall이 필요한 경우들은 다음과 같이 forward를 적용할 수 있었다.

Lui $6, 0x0000와 ori $6,$6,0x2000의 register 6에 대하여 EX/MEM을 통해 값을 받아 0stall 필요.

Addi $2,$0,0x10과 Addi $3,$0,0과 sub $4,$2,$3에 대해 forward를 적용하려 하나 해당 명령어는 addiu $3,$3,1의 영향을 branch를 통해 받게 된다. 이들은 필요한 stall수가 달라 forward을 하나로 좁힐 수 없다. 그 때문에 stall수가 유지된다.

Sub $4,$2,$3과 addiu $4,$4,-1에 대하여 register 4에 대해 EX/MEM을 통해 값을 받아 0 stall 필요

Addi $5,$0,0과 sll $7,$5,2에 대해 forward를 적용하려 하나 해당 명령어는addiu $5,$5,1의 영향을 branch를 통해 받게 된다. 이들은 필요한 stall수가 달라 forward을 하나로 좁힐 수 없다. 그 때문에 stall수가 유지된다.

Sll $7,$5,2과 addu $7,$6,$7에 대하여 register 7에 대해 EX/MEM을 통해 값을 받아 0 stall 필요

addu $7,$6,$7와 lw $8,0($7)에 대하여 register 7에 대해 EX/MEM을 통해 값을 받아 0 stall 필요

addu $7,$6,$7와 lw $9,4($7)에 대하여 register 7에 대해 MEM/WB을 통해 값을 받아 0 stall 필요

lw $8,0($7)과 slt $1,$8,$9에 대해 forward를 적용하려 하나 해당 명령어는 lw $9,4($7)의 영향을 또한 받고 있다. Lw에 대해 forward를 적용하려면 MEM/WB에서 값을 받아와야 하지만, LW 2개로부터 값을 받아오므로 2개를 동시에 Forward를 통해 처리할 수 없다. 또한 LW는 WB stage에서만 받을 수 있어 2개의 LW가 같을 곳을 향할 경우 forward를 적용할 수 없다. 그 때문에 stall수가 유지된다.

Bne $1,$0,L3에 대해서는 register값이 ID에서 필요하기에 forward로 stall수를 줄일 수 없다.

add $10,$8,$0와 add $9,$10,$0에 대해 register10에 대해 MEM/WB을 통해 값을 받아 0 stall 필요

add $8,$9,$0와 sw $8,0($7)에 대해 register 8에 대해 MEM/WB을 통해 값을 받아 0 stall 필요

add $9,$10,$0와 sw $9,4($7)에 대해 register9에 대해 MEM/WB을 통해 값을 받아 0 stall 필요

addiu $5,$5,1와 slt $1,$5,$4에 대하여 register 5에 대해 EX/MEM을 통해 값을 받아 0 stall 필요

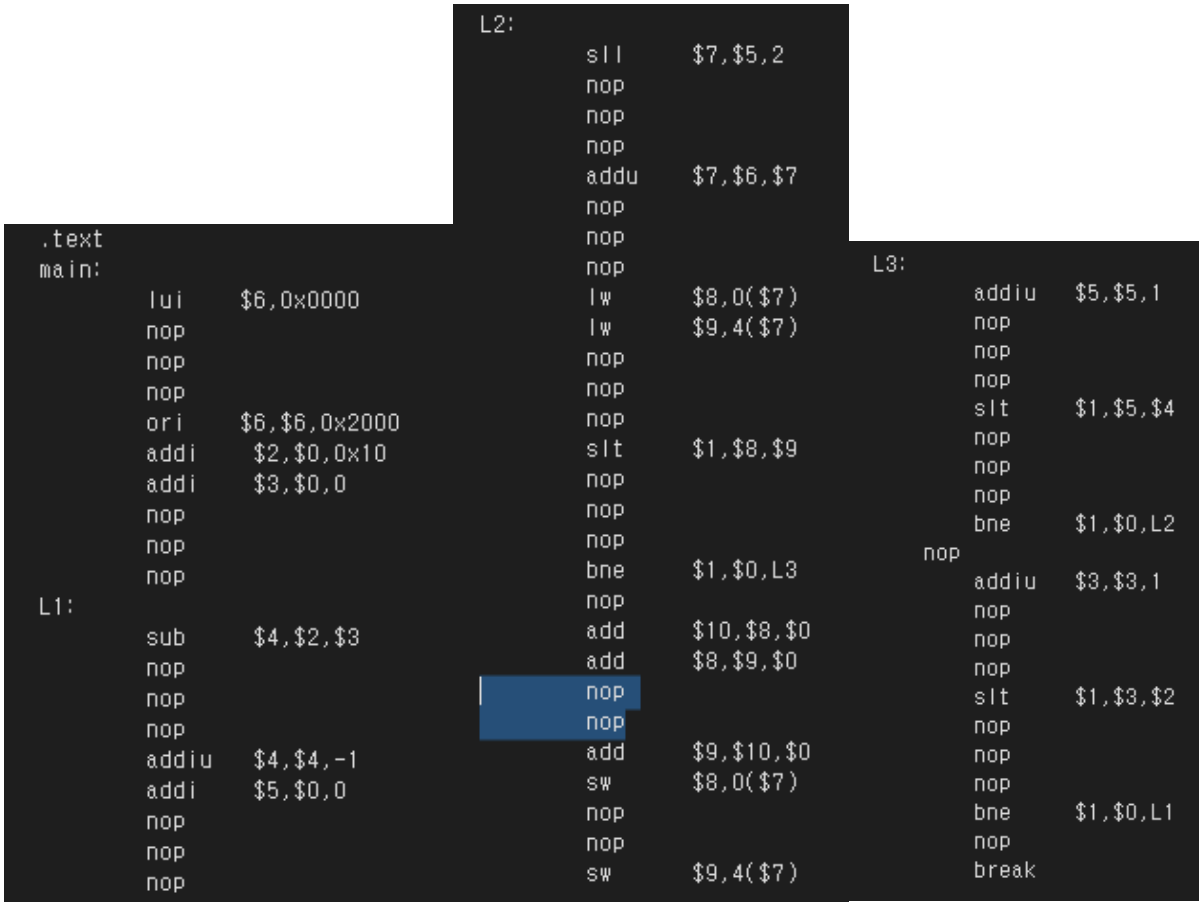
Bne$1,$0,L2에 대해서는 register값이 ID에서 필요하기에 forward로 stall수를 줄일 수 없다.

addiu $3,$3,1와 slt $1,$3,$2에 대하여 register 3에 대해 EX/MEM을 통해 값을 받아 0 stall 필요

Bne $1,$0,L1에 대해서는 register값이 ID에서 필요하기에 forward로 stall수를 줄일 수 없다.

각 명령어에 대해 다음과 같은 이유로 forward로 줄일 수 있는 경우는 우측 그림과 같이 나오게 된다.

2.1-2-1(Remove NOP with out code rescheduling)



```
.text
main:
        lui     $6,0x0000
        nop
        nop
        nop
        ori     $6,$6,0x2000
        addi    $2,$0,0x10
        addi    $3,$0,0
        nop
        nop
        nop
L1:
        sub     $4,$2,$3
        nop
        nop
        nop
        addiu   $4,$4,-1
        addi    $5,$0,0
        nop
        nop
        nop
L2:
        sll     $7,$5,2
        nop
        nop
        nop
        addu    $7,$6,$7
        nop
        nop
        nop
        lw      $8,0($7)
        lw      $9,4($7)
        nop
        nop
        nop
        slt     $1,$8,$9
        nop
        nop
        nop
        bne     $1,$0,L3
        nop
        add     $10,$8,$0
        add     $8,$9,$0
        nop
        nop
        add     $9,$10,$0
        sw      $8,0($7)
        nop
        nop
        sw      $9,4($7)
L3:
        addiu   $5,$5,1
        nop
        nop
        nop
        slt     $1,$5,$4
        nop
        nop
        nop
        bne     $1,$0,L2
    nop
        addiu   $3,$3,1
        nop
        nop
        nop
        slt     $1,$3,$2
        nop
        nop
        nop
        bne     $1,$0,L1
        nop
        break
```

위는 2.1-1-1, 2.1-1-2의 결과를 바탕으로 NOP을 넣은 값이다.

Data hazards를 막기 위해 레지스터의 값을 변경하는 명령어(lui, ori, addi, sub, addiu, sll, slt, add, lw)의 경우 변경되는 레지스터가 불리기 이전에 3clock이 진행되도록 설계된 것을 위 코드에서 확인할 수 있다.

Lui $6, 0x0000 – No hazard

Ori $6,$6,0x2000 Lui $6, 0x0000과의 사이에 nop이 3개 추가되어 r6의 Data hazard해결.

Addi $2, $0, 0x10-No hazard

Addi $3, $0, 0-No hazard

Sub $4, $2, $3- Addi $2, $0, 0x10, Addi $3,$0,0사이에 3개의 nop추가로 $2,$3의 Data hazard 해결

Bne $1, $0,L1이후 nop을 추가해 register 3의 Data hazard, control hazard 해결

Addiu $4, $4, -1 - sub $4,$2,$3와 중간에 3개의 nop추가로 $4에 대한 Data hazard 해결

Addi $5,$0,0-No hazard

Sll $7, $5, $2- Addi $5,$0,0사이에 3개의 nop추가로 $5의 Data hazard 해결

Bne $1, $0,L2이후 nop을 추가해 $5의 Data hazard, control hazard 해결

Addu $7,$6,$7- Sll $7, $5, $2사이에 3개의 nop 추가로 $7의 Data hazard 해결

Lw $8, 0($7)- Addu $7,$6,$7사이에 3개의 nop 추가로 $7의 Data hazard해결

Lw $9, 4($7)- Addu $7,$6,$7사이에 3개의 nop 추가로 $7의 Data hazard해결

Slt $1,$8,$9- Lw $8, 0($7),Lw $9, 4($7)와의 사이에 nop3개를 추가해 $8,$9의 Data hazard 해결

Bne $1,$0,L3-Slt $1,$8,$9와의 사이에 nop 3개를 추가해 $1의 Data hazard 해결

Add $10, $8, $0-Bne $1,$0,L3와의 사이에 nop1개를 추가해 control hazard 해결

Add $8,$9,$0-No hazard

Add $9,$10,$0-add $10,$8, $0와의 사이에 nop을 2개 추가해 $10의 data hazard 해결

Sw $8,0($7)- Add $8,$9,$0와의 사이에 nop을 2개 추가해 $8의 data hazard 해결

Sw $9,4($7)- Add $9,$10,$0와의 사이에 nop을 2개 추가해 $9의 data hazard 해결

Addiu $5,$5,1-Bne $1, $0,L3이후 nop을 추가해 control hazard 해결

slt $1,$5,$4- Addiu $5,$5,1와의 사이에 nop을 3개 추가해 $5의 data hazard 해결

bne $1,$0,L2-slt $1,$5,$4와의 사이에 nop을 3개 추가해 $1의 data hazard 해결

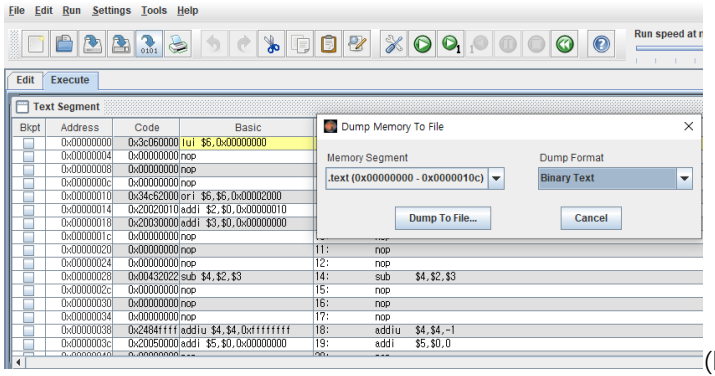
Addiu $3,$3,1-Bne $1, $0,L2이후 nop을 추가해 control hazard 해결

slt $1,$3,$2- Addiu $3,$3,1와의 사이에 nop을 3개 추가해 $3의 data hazard 해결

bne $1,$0,L1-slt $1,$3,$2와의 사이에 nop을 3개 추가해 $1의 data hazard 해결

break- bne $1,$0,L1이후에 nop을 1개 추가해 control hazard 해결

해당 값들을 MARS를 이용, 다음과 같이 Binary의 형태로 바꾸어 해당 코드를 실행하여 보았다.



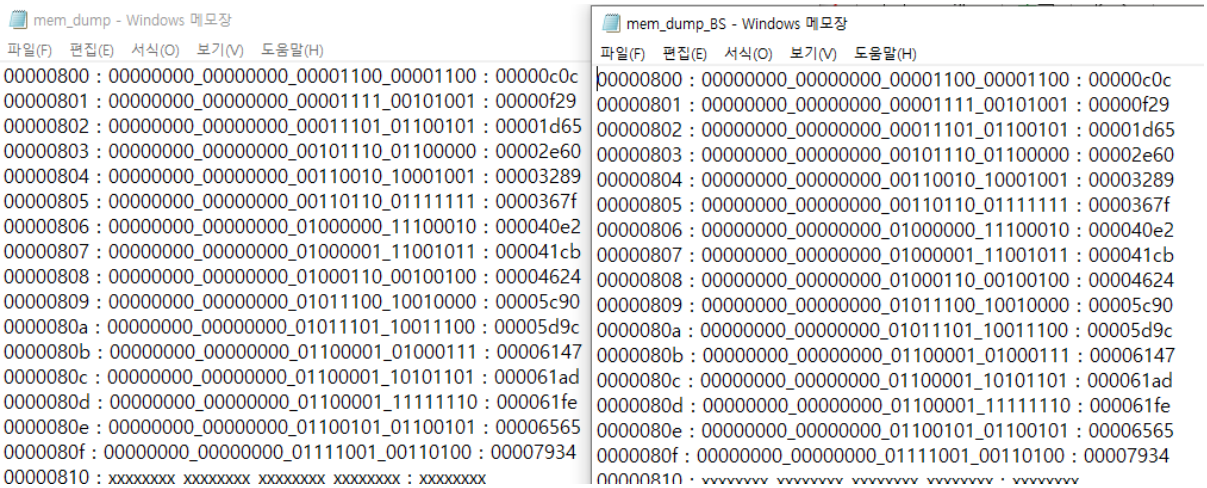
(Mars를 통해 Binary로 변경)

```
00111100000001100000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00110100110001100010000000000000
00100000000000100000000000010000
00100000000000110000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000010000110010000000100010
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00100100100001001111111111111111
00100000000001010000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000001010011100010000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000110001110011100000100001
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
10001100111010000000000000000000
10001100111010010000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
```

```
00000000000000000000000000000000
00000001000010010000100000101010
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00010100001000000000000000001010
00000000000000000000000000000000
00000001000000000101000000100000
00000001001000000100000000100000
00000000000000000000000000000000
00000000000000000000000000000000
00000001010000000100100000100000
10101100111010000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
10101100111010010000000000000100
00100100101001010000000000000001
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000010100100000100000101010
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00010100001000001111111111011011
00000000000000000000000000000000
00100100011000110000000000000001
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000011000100000100000101010
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00010100001000001111111111001000
00000000000000000000000000000000
00000000000000000000000000001101
```

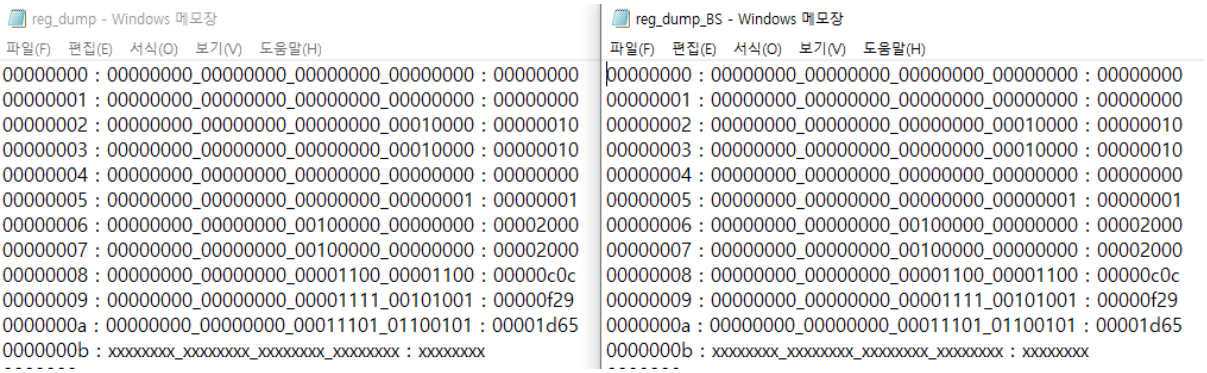
(M_TEXT_SEG에 들어가게 되는 Binary code)

위 값이 적절한 해답이라면 Introduction의 내용을 기반으로 한 Bubble sort가 발생, M_DATA_SEG의 값들이 Mem_dump에 작은 수 ~ 큰 수 순서대로 정렬된 결과가 출력될 것이다. 올바른 정답은 mem_dump_BS, reg_dump_BS과 같이 나오게 될 것이다.(각각 bubble sort결과, 마지막 register끼리의 비교와 마지막으로 사용된 memory의 주소 값 등이 저장되어 있을 것이다.)

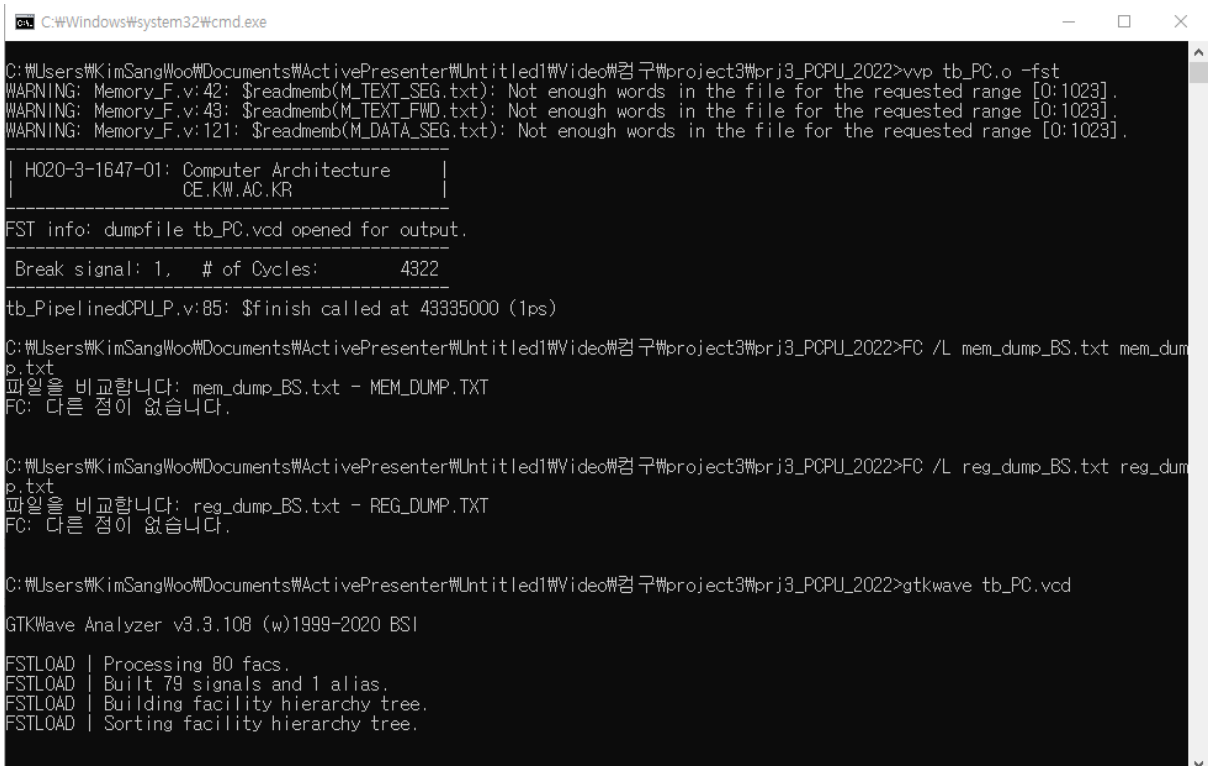


위는 실행후의 결과이다. mem_dump와 mem_dump_BS의 결과가 같은 걸 확인할 수 있다.

예측대로 작은 수(0x00000c0c)부터 큰 수(0x00007934)까지 나열된 것을 확인할 수 있다.

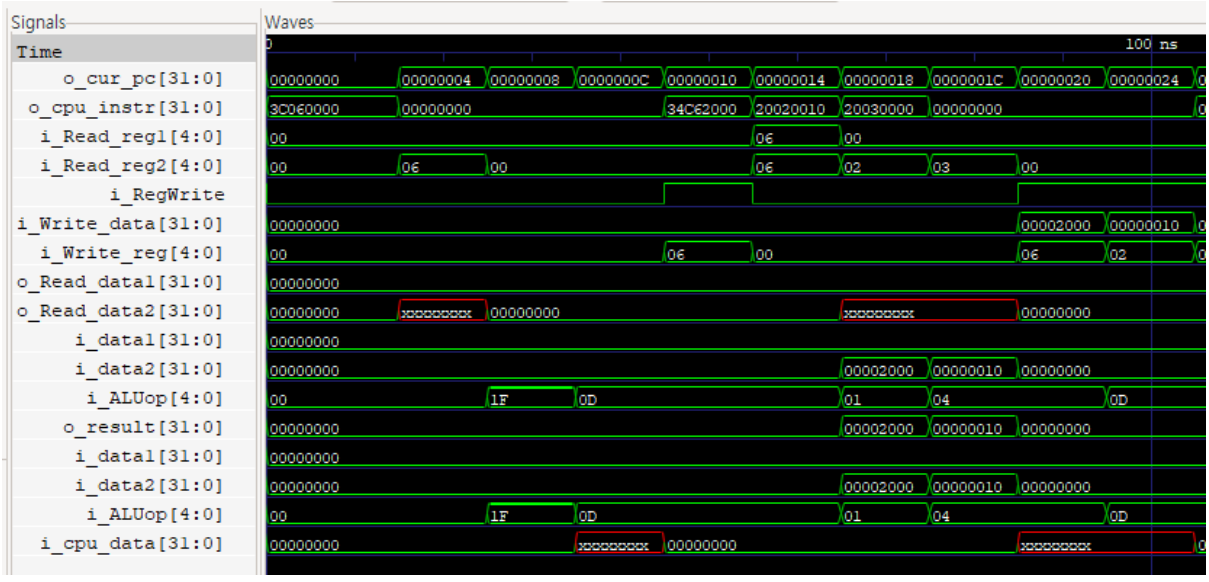


reg_dump와 reg_dump_BS의 결과 또한 같은 걸 확인할 수 있다.

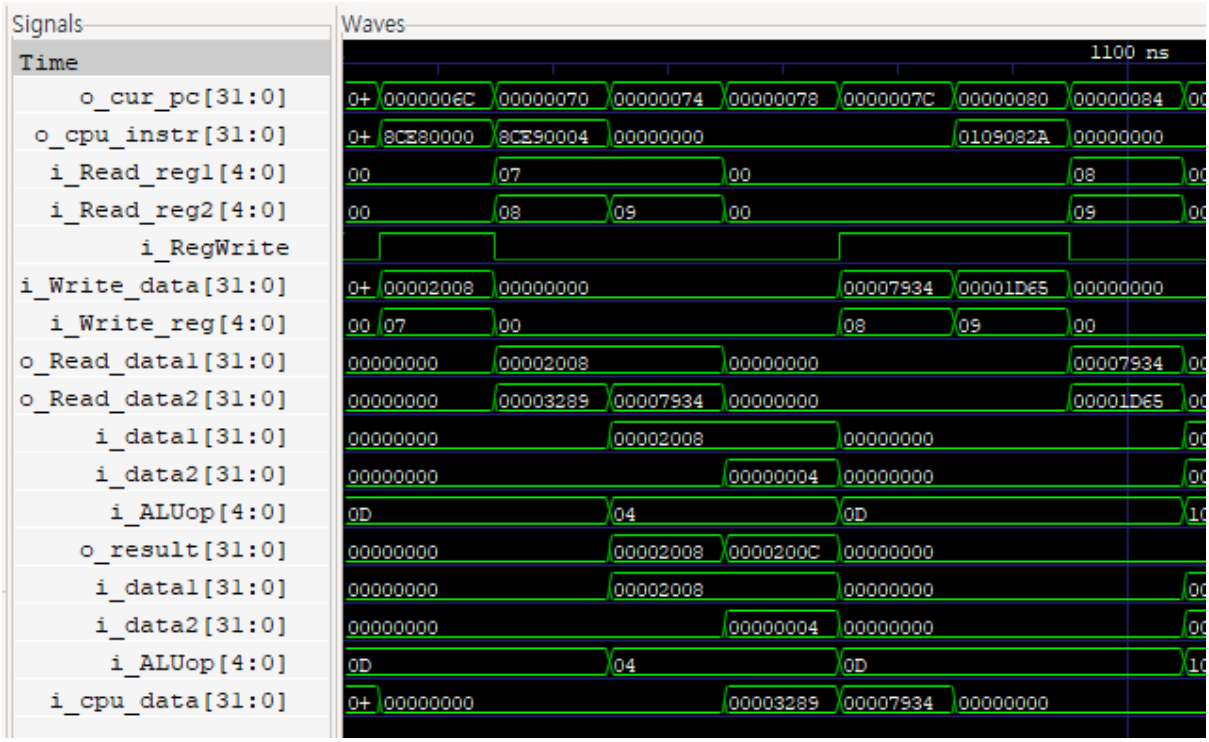


Cycle은 다음과 같이 4322가 나온다. 프로그램 결과로도 결과와 예측 결과가 같게 나왔음을 확인할 수 있었다.

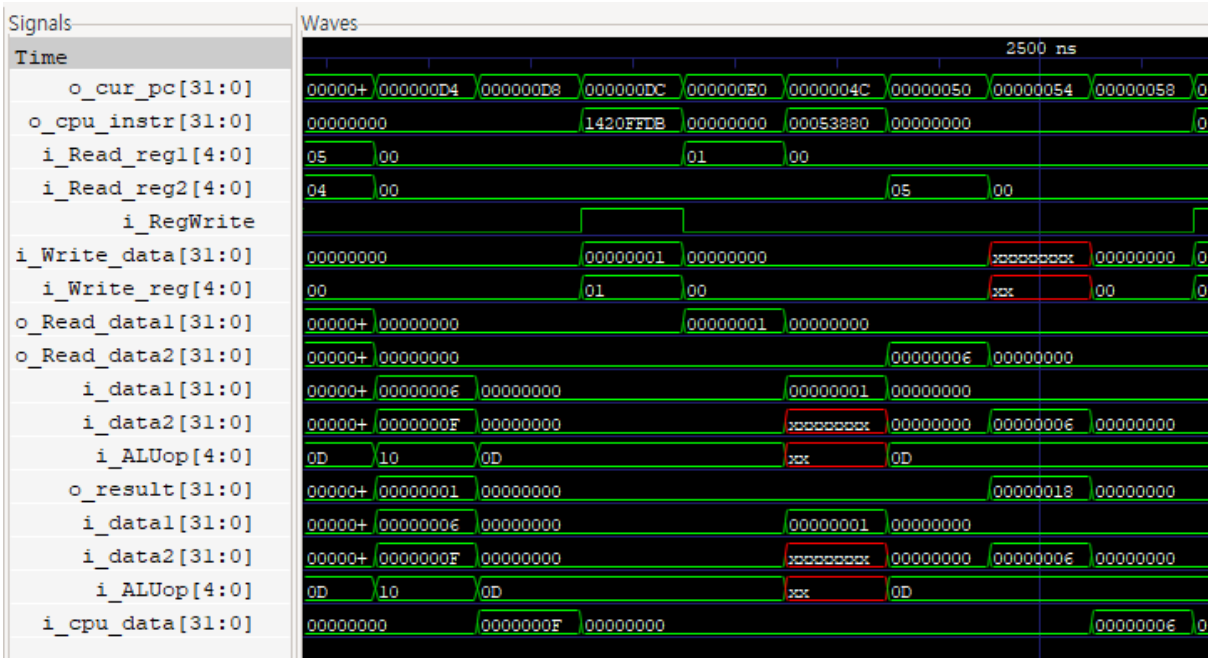
단, 내부적으로 파이프라인이 잘 작동하는지 확인할 필요가 있다. 해당 프로그램은 GTKWave를 지원하므로 이를 바탕으로 확인이 가능할 것이다.



위는 GTKWwave의 초반 부분을 캡쳐한 것이다. 위에서 알 수 있듯, lui $6, 0x0000이 instruction으로 받아지고, 이후 i_read_reg에 값이 담기며, WB단계에서 해당 register에 0의 값을 저장하는 걸 확인할 수 있다. 다수의 명령어가 겹칠 시에도 문제 없이 작동하는 것을 0x00000010부터 확인할 수 있다.(ori $6,$6,0x2000, addi $2,$0,0x10, addi $3,$0,0 병행처리)



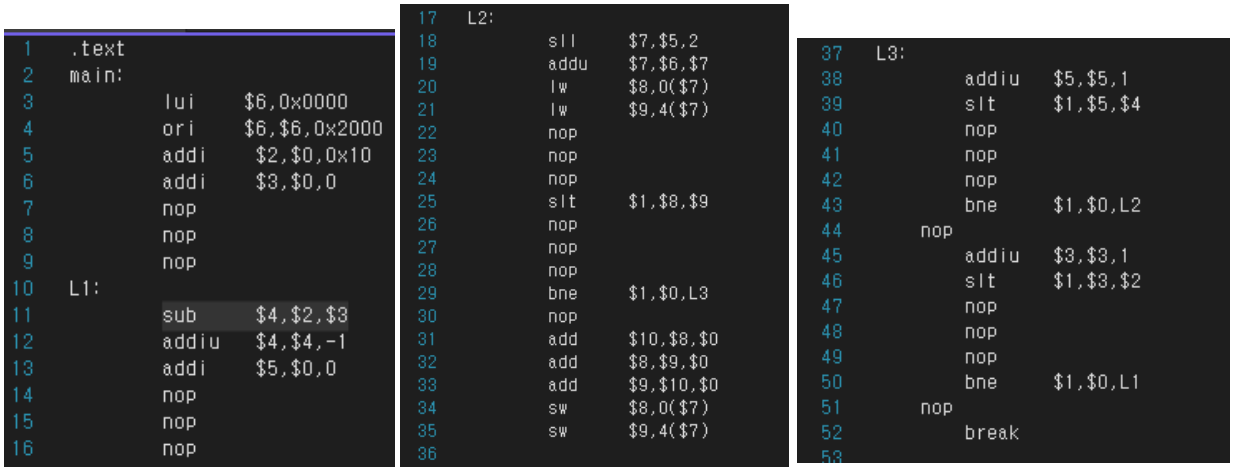
위는 lw $8,0($7)과 lw $9,4($7) 명령어에 해당하는 부분이다. lw또한 instruction이 선언된 후 ID단계에서 i_read_reg로 값이 들어가고, 값이 출력되어 EX단계에서 ALU를 통해 주소를 계산후(i_data1,2 o_result)MEM을 거쳐 WB에서 꺼내온 값을 register에 저장하는 것을 확인할 수 있다.



위는 bne $1,$0,L2에 대한 부분이다.(o_cur_pc=0x000000DC부터) 이 부분에서 명령어를 받고 다름 pc에서 register값을 받고 Ex부분에서 이를 확인해 pc가 변경됨을 확인할 수 있다.

이 과정을 통해 각 타입의 명령어들도 pipeline에서 잘 돌아갔음을 확인했다.

2.1-2-2(Remove NOP by inserting forward control signal)



```
1   .text
2   main:
3           lui     $6,0x0000
4           ori     $6,$6,0x2000
5           addi    $2,$0,0x10
6           addi    $3,$0,0
7           nop
8           nop
9           nop
10  L1:
11          sub     $4,$2,$3
12          addiu   $4,$4,-1
13          addi    $5,$0,0
14          nop
15          nop
16          nop
17  L2:
18          sll     $7,$5,2
19          addu    $7,$6,$7
20          lw      $8,0($7)
21          lw      $9,4($7)
22          nop
23          nop
24          nop
25          slt     $1,$8,$9
26          nop
27          nop
28          nop
29          bne     $1,$0,L3
30          nop
31          add     $10,$8,$0
32          add     $8,$9,$0
33          add     $9,$10,$0
34          sw      $8,0($7)
35          sw      $9,4($7)
36
37  L3:
38          addiu   $5,$5,1
39          slt     $1,$5,$4
40          nop
41          nop
42          nop
43          bne     $1,$0,L2
44      nop
45          addiu   $3,$3,1
46          slt     $1,$3,$2
47          nop
48          nop
49          nop
50          bne     $1,$0,L1
51      nop
52          break
53
```

위는 2.1-2-1의 코드에 2.1-1-3에서 확인한 forward 적용시의 stall수 결과를 바탕으로 위와 같은 코드를 작성하였다. 각 명령어들은 아래 설명과 같은 방식으로 처리되었다.

Lui $6, 0x0000 – No hazard

Ori $6,$6,0x2000-Lui $6, 0x0000의 ALU결과를 EX/MEM으로부터 (r6)값을 받아 Data hazard해결.

Addi $2, $0, 0x10-No hazard

Addi $3, $0, 0-No hazard

Sub $4, $2, $3- Addi $2, $0, 0x10, Addi $3,$0,0사이에 3개의 nop추가로 $2,$3의 Data hazard 해결

Bne $1, $0,L1이후 nop을 추가해 register 3의 Data hazard, control hazard 해결

Addiu $4, $4, -1 - sub $4,$2,$3의 ALU결과를 EX/MEM으로부터 ($4)값을 받아 Data hazard 해결

Addi $5,$0,0-No hazard

Sll $7, $5, $2- Addi $5,$0,0사이에 3개의 nop추가로 $5의 Data hazard 해결

Bne $1, $0,L2이후 nop을 추가해 $5의 Data hazard, control hazard 해결

Addu $7,$6,$7- Sll $7, $5, $2의 ALU결과를 EX/MEM으로 부터 ($7)값을 받아 Data hazard 해결

Lw $8, 0($7)- Addu $7,$6,$7의 ALU결과를 EX/MEM으로 부터 ($7)값을 받아 Data hazard해결

Lw $9, 4($7)- Addu $7,$6,$7의 ALU결과를 MEM/WB으로 부터 ($7)값을 받아 Data hazard해결

Slt $1,$8,$9- Lw $8, 0($7),Lw $9, 4($7)와의 사이에 nop3개를 추가해 $8,$9의 Data hazard 해결

Bne $1,$0,L3-Slt $1,$8,$9와의 사이에 nop 3개를 추가해 $1의 Data hazard 해결

Add $10, $8, $0-Bne $1,$0,L3와의 사이에 nop1개를 추가해 control hazard 해결

Add $8,$9,$0-No hazard

Add $9,$10,$0-add $10,$8, $0의 ALU결과를 MEM/WB로부터 ($10)값을 받아 data hazard 해결

Sw $8,0($7)- Add $8,$9,$0의 ALU결과를 MEM/WB로부터 ($8)값을 받아 data hazard 해결

Sw $9,4($7)- Add $9,$10,$0의 ALU결과를 MEM/WB로부터 ($9)값을 받아 data hazard 해결

Addiu $5,$5,1-Bne $1, $0,L3이후 nop을 추가해 control hazard 해결

slt $1,$5,$4- Addiu $5,$5,1의 ALU결과를 EX/MEM으로부터 ($5)값을 받아 data hazard 해결

bne $1,$0,L2-slt $1,$5,$4와의 사이에 nop을 3개 추가해 $1의 data hazard 해결

Addiu $3,$3,1-Bne $1, $0,L2이후 nop을 추가해 control hazard 해결

slt $1,$3,$2- Addiu $3,$3,1의 ALU결과를 EX/MEM으로부터 ($3)값을 받아 data hazard 해결

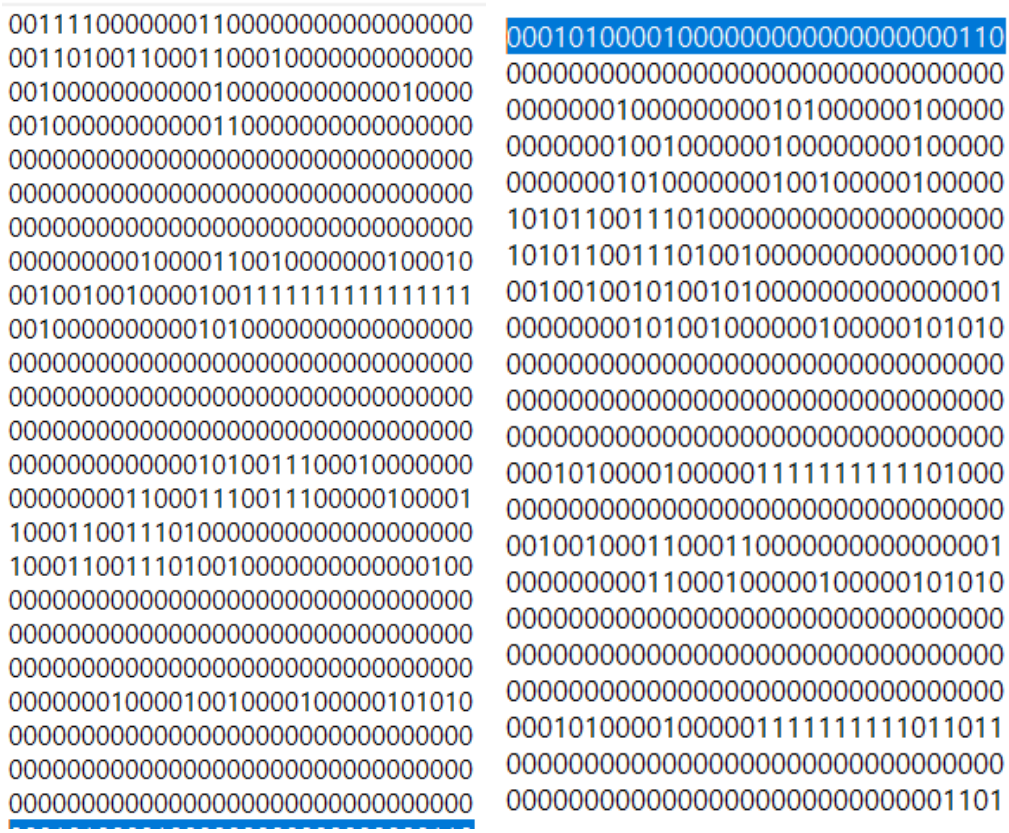
bne $1,$0,L1-slt $1,$3,$2와의 사이에 nop을 3개 추가해 $1의 data hazard 해결

break- bne $1,$0,L1이후에 nop을 1개 추가해 control hazard 해결

위에서 Forwarding이 된 것들을 바탕으로 M_TEXT_FWD.txt를 작성하면 다음과 같게 된다.

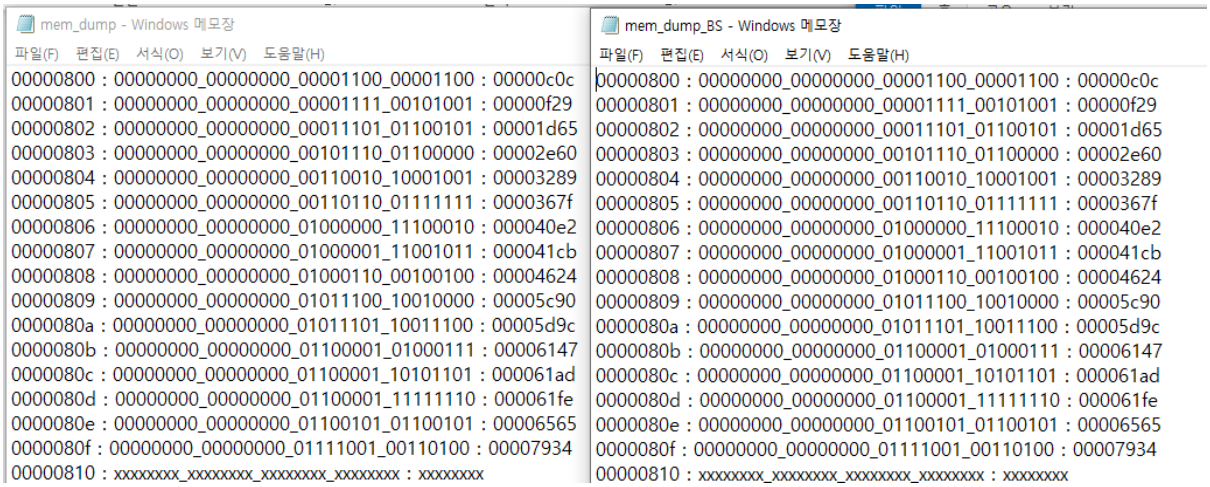
```
00_00  //  0x000
01_00  //  0x004 Ori $6,$6,0x2000
00_00  //  0x008
00_00  //  0x00C
00_00  //  0x010
00_00  //  0x014
00_00  //  0x018
00_00  //  0x01C
01_00  //  0x020 Addiu $4, $4, -1
00_00  //  0x024
00_00  //  0x028
00_00  //  0x02C
00_00  //  0x030
00_00  //  0x034
00_01  //  0x038 Addu $7,$6,$7
01_00  //  0x03C Lw $8, 0($7)
10_00  //  0x040 Lw $9, 4($7)
00_00  //  0x044
00_00  //  0x048
00_00  //  0x04C
00_00  //  0x050
00_00  //  0x054
00_00  //  0x058
00_00  //  0x05C
00_00  //  0x060
00_00  //  0x064
00_00  //  0x068
00_00  //  0x06C
10_00  //  0x070 add $9,$10,$0
00_10  //  0x074 sw $8, 0($7)
00_10  //  0x078 sw $9, 4($7)
00_00  //  0x07C
01_00  //  0x080 slt $1,$5,$4
00_00  //  0x084
00_00  //  0x088
00_00  //  0x08C
00_00  //  0x090
00_00  //  0x094
00_00  //  0x098
01_00  //  0x09C slt $1,$3,$2
00_00  //  0x0A0
00_00  //  0x0A4
00_00  //  0x0A8
00_00  //  0x0AC
00_00  //  0x0B0
00_00  //  0x0B4
00_00  //  0x0B8
00_00  //  0x0BC
00_00  //  0x0C0
```

위에서 확인된 asem코드를 Mars를 통해 Binary형태로 변경하면 다음과 같게 된다.

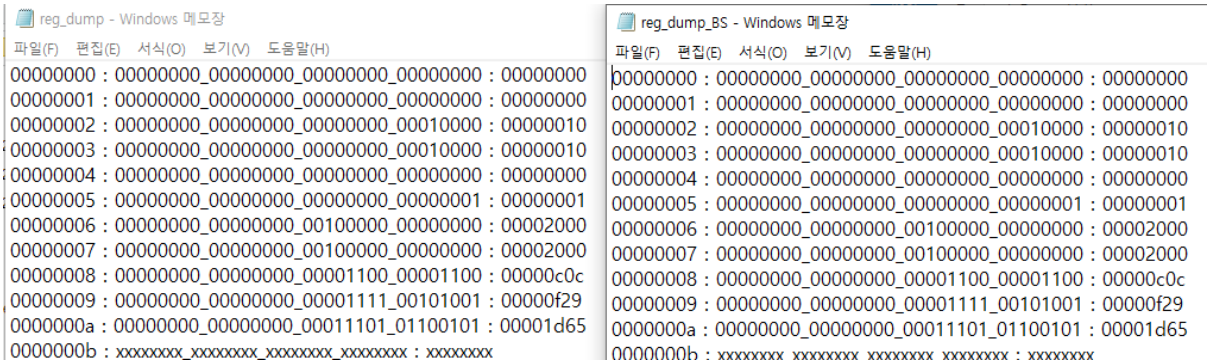


다음값을 M_TEXT_SEG.txt파일에 넣어 결과를 확인하였다.

위 값이 적절한 해답이라면 Introduction의 내용을 기반으로 한 Bubble sort가 발생, M_DATA_SEG의 값들이 Mem_dump에 작은 수 ~ 큰 수 순서대로 정렬된 결과가 출력될 것이다. 올바른 정답은 mem_dump_BS, reg_dump_BS과 같이 나오게 될 것이다.(각각 bubble sort결과, 마지막 register끼리의 비교와 마지막으로 사용된 memory의 주소 값 등이 저장되어 있을 것이다.)



Mem_dump.txt와 mem_dump_BS.txt가 같음을 확인할 수 있다.



reg_dump.txt와 reg_dump_BS.txt가 같음을 확인할 수 있다.

```
C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-2>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the file for the requested range [0:1023].
-----------------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-----------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------------
 Break signal: 1,   # of Cycles:        2914
-----------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 29255000 (1ps)

C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-2>FC /L mem_dump_BS.txt mem_dump.txt
파일을 비교합니다: mem_dump_BS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-2>FC /L reg_dump_BS.txt reg_dump.txt
파일을 비교합니다: reg_dump_BS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-2>gtkwave tb_PC.vcd

GTKWave Analyzer v3.3.108 (w)1999-2020 BSI

FSTLOAD | Processing 80 facs.
FSTLOAD | Built 79 signals and 1 alias.
FSTLOAD | Building facility hierarchy tree.
FSTLOAD | Sorting facility hierarchy tree.
```

프로그램 상에서도 이 txt파일들이 같다고 나오는 것을 확인할 수 있다.

Cycle은 2914만큼 나왔다.

이는 forwarding을 사용하지 않았을 때 보다 Cycle이 4432-2970= 1462만큼 더 감소하였음을 알 수 있다. 이 이유는 forwarding을 통해 더 많은 NOP을 제거하였고,(Forwarding이전에는 43개 였지만 Forwarding이후에는 21개로 감소) NOP또한 Cycle을 소모하므로 다음과 같이 줄어들게 된 것이다. 줄인 NOP수 보다 더 많은 Cycle이 줄어든 것은 반복문 내부의 NOP를 줄여 더욱 많은 Cycle이 확보된 것이다.

단, 우리는 명령어를 바꾸어 이 cycle을 줄일 수 있다.

(해당 과제에서는 명령어 순서 변경(re scheduling)은 허용되나 명령어 추가 및 삭제가 허용되지 않는다.)

위의 결과들을 바탕으로 수정할 경우 다음과 같이 수정할 수 있다

```
1   .text
2   main:
3           addi    $2,$0,0x10
4           addi    $3,$0,0
5           lui     $6,0x0000
6           ori     $6,$6,0x2000
7   L1:
8       addi    $5,$0,0
9           sub     $4,$2,$3
10          addiu   $4,$4,-1
11          nop
12  L2:
```

```
12  L2:
13          sll     $7,$5,2
14          addu    $7,$6,$7
15          lw      $8,0($7)
16          lw      $9,4($7)
17          nop
18          nop
19          nop
20          slt     $1,$8,$9
21          nop
22          nop
23          nop
24          bne     $1,$0,L3
25          nop
26          add     $10,$8,$0
27          add     $8,$9,$0
28          add     $9,$10,$0
29          sw      $8,0($7)
30          sw      $9,4($7)
```

```
32  L3:
33          addiu   $5,$5,1
34          slt     $1,$5,$4
35          nop
36          nop
37          nop
38          bne     $1,$0,L2
39      nop
40          addiu   $3,$3,1
41          slt     $1,$3,$2
42          nop
43          nop
44          nop
45          bne     $1,$0,L1
46      nop
47          break
```

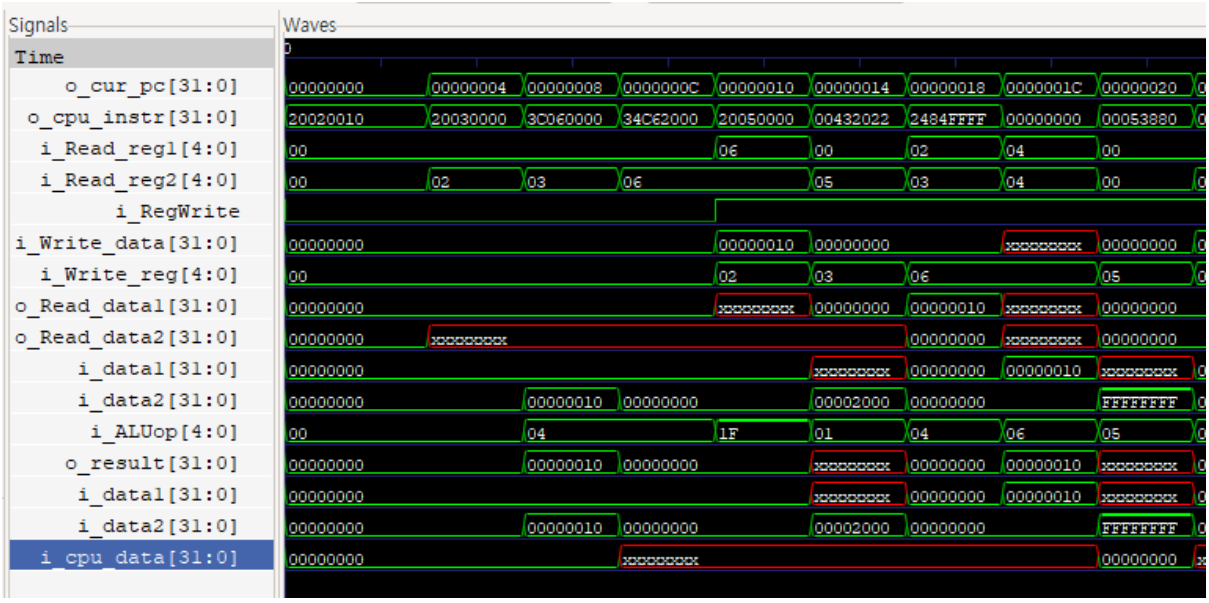
위에서 수정된 부분은 다음과 같다.

L1의 sub $4, $2,$3에서 register2,3에 대해 addi $2,$0,0x10와 addi $3, $0,0을 가장 위로 올리고 addi $5,$0,0을 L1의 처음으로 올림으로서 중간에 명령어가 3개 들어가게 되었고 그 때문에 nop을 3개 제거할 수 있었다.

L2의 sll $7,$5,2에 대해 addi $5,$0,0을 L1의 처음으로 올림으로서 중간에 명령어가 3개 들어가게 되었고 그 때문에 nop을 2개 제거할 수 있었다.

```
C:₩Windows₩system32₩cmd.exe

C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-3>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the file for the requested range [0:1023].
-----------------------------------------------
| HO2O-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-----------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------------
 Break signal: 1,   # of Cycles:        2879
-----------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 28905000 (1ps)

C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-3>FC /L mem_dump_BS.txt mem_dump.txt
파일을 비교합니다: mem_dump_BS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-3>FC /L reg_dump_BS.txt reg_dump.txt
파일을 비교합니다: reg_dump_BS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.


C:₩Users₩KimSangWoo₩Music₩prj3_PCPU_2022-3>gtkwave tb_PC.vcd

GTKWave Analyzer v3.3.108 (w)1999-2020 BSI

FSTLOAD | Processing 80 facs.
FSTLOAD | Built 79 signals and 1 alias.
FSTLOAD | Building facility hierarchy tree.
FSTLOAD | Sorting facility hierarchy tree.
```

결과적으로 cycle이 2914보다 35만큼 더 줄게 되었다. txt파일의 비교가 정상인 것으로 프로그램도 정상 작동 했음을 알 수 있다.



위는 해당 코드의 초반을 GTK WAVE로 나타낸 것이다. 기존에 예상한대로 명령어가 적힌 곳에 해당하는 지점에서 instruction을 받고, 이후 register를 받은 후 EX stage에서 ALU계산을, WB stage에서 맞는 값을 writeback해주는 모습을 확인할 수 있다. 이들은 stage에 따라 사용하는 부분을 같은 시간에 공유하지 않고 있음 또한 확인할 수 있었다.

문제점 및 고찰

위 프로젝트는 Bubble sort를 진행하는 어셈블리 코드를 pipeline으로 변경하고, 이 과정에서 data dependency와 control dependency을 확인하고 이들을 바탕으로 hazards를 방지하며 stall을 넣어보는 등의 활동을 통해 구조를 이해하고 이를 응용할 수 있도록 하는 것에 목적이 있다.

무엇보다 Cycle을 줄이는 것에 대해 Forwarding을 적용하였을 때, 적용하지 않았을 때, Forwarding을 적용하고 동시에 순서 또한 바꾸었을 때를 구분하여 실행하고 그 결과 Cycle이 크게 바뀌는 것을 확인하며 Cycle을 줄이는 설계의 중요성과 방법을 확인할 수 있었다.

아쉬운 점이 있다면 위 프로젝트에서는 명령어의 추가나 삭제가 불가능하다. 이들이 허용될 경우 더 많은 경우의 수가 생겼을 것이며 그 결과 더 적은 Cycle로 돌리는 것도 가능하였을 것이다. 대표적으로 Bubble sort에서 swap을 위해 register 3개를 사용하고 ADD를 3번하게 되는데, 이 과정에서 Nop이 발생하거나 forward사용이 필요해 졌다. 그러나 이는 r8, r9에 저장후 바로 sw를 해주어도 되는 맹점이 있다. 이는 반복문 내부에 있는 만큼 삭제할 경우 많은 cycle이 확보될 것이다.

앞으로 컴퓨터 구조에 대해 공부하게 되며 pipeline에서 본 병행처리 기술은 주요하게 사용될 것이며 이를 위해 완전히 숙지할 필요가 있다고 생각하였다.