컴퓨터구조

이성원교수님

Project #3

학과 : 컴퓨터 정보 공학부

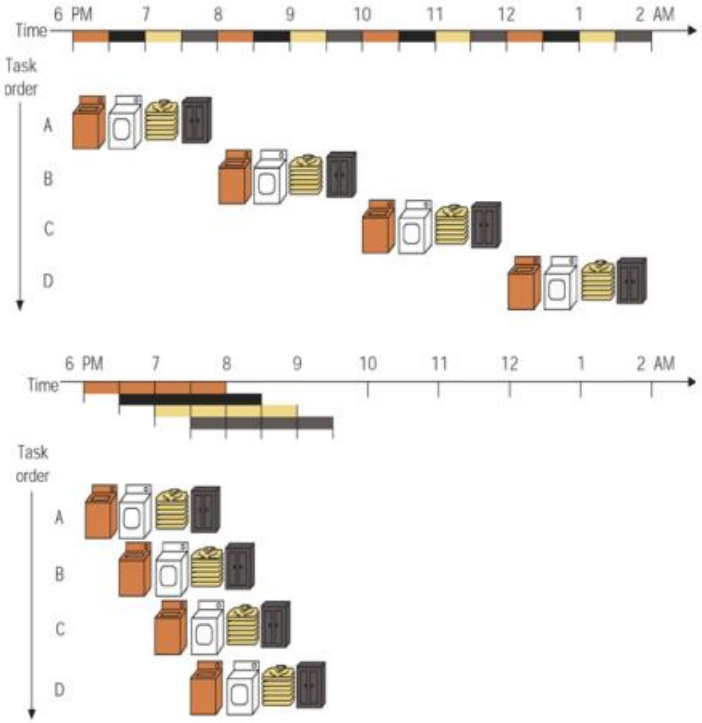
학번 : 2019202103

이름 : 이은비

제출 날짜 : 2023.05.24

[실험 내용]

수업 자료를 참고하면 알 수 있듯이 일반적이 single cycle에서 sequential executions of four jobs take 8 hours인 반면에 Pipelined executions of four jobs take 3.5 hours인 것을 알 수 있습니다.



그리고 이때 Pipelining은 throughput을 향상시킴에 따라서 성능을 획기적으로 상승시킵니다. 그러나 pipeline을 사용하면서 크게 3가지 hazard, Structure hazard, Data hazard, Control hazard가 발생하는데 Structural hazards 즉, 우리는 1개의 메모리만 사용한다는 전제에 있을 때 같은 cycle에 2개의 instruction이 write을 시도하기 때문에 발생하는 문제가 있으며 Data hazard는 instruction이 바로 전의 instruction에 따라 영향을 받을 때 즉 정 순서로 이어지는 전제에 있으므로 ready 상태가 되지 않은 즉 결과로 처리되지 않은 instruction에 대해서 item을 사용할 시도하여 2개의 insturctions이 같은 strorage에 접근하면서 발생하는 문제입니다. control hazards는 branch instructions에 있어서 beq r1, loop add r1, r2, r3 와 같이 상태를 평가하기 전에 결정을 내리려고 시도하여서 다른 insturction이 영향을 받아 발생하는 문제입니다.

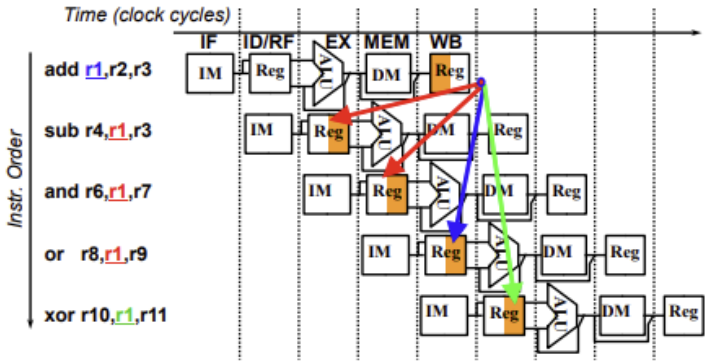
그리고 각각의 hazard를 해결하기 위한 방법을 보면 다음과 같습니다.

Structual hazard는 hardware resource(memory)를 더 두어 cost는 더 늘지만 hazard를 극복할 수 있습니다.

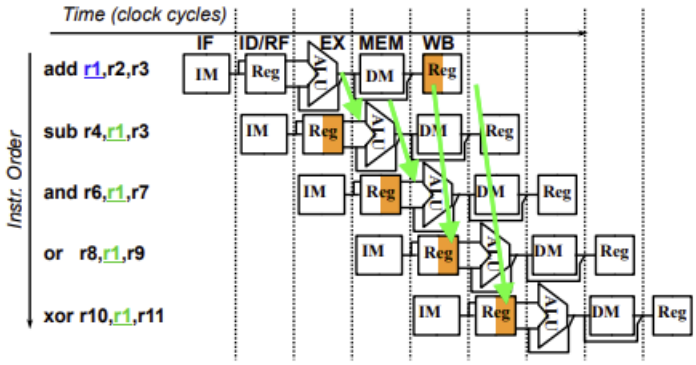
Data hazard는 1. Stall을 하는 방식, 2. hardware resource 사용 (multi-cycle resource를 사용)

3. Harvard Architecture 같이 resource 복제한다. (data와 instruction 분리해서 받음으로써 cost는 늘지만 상황에 따라 좋은 방법이 될 수도 있습니다.)

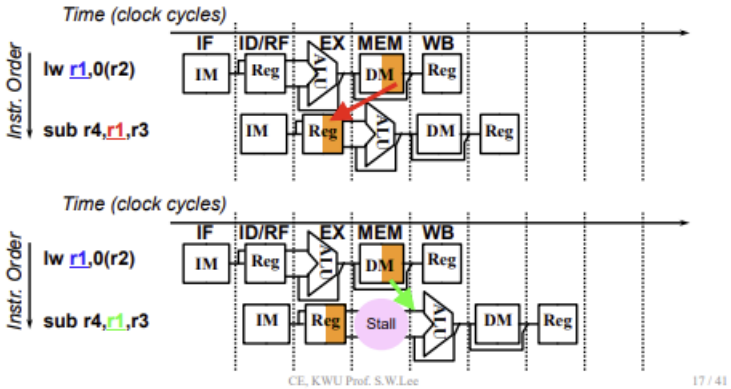
Data hazard는 그냥 stall즉, bubble을 사용하는 방식과 forwarding (or bypassing)을 사용하는 방식을 따릅니다. 우선적으로 forwarding을 사용하는데 forwarding으로 충분하게 hazard를 극복하지 못했을 때는 stall을 사용하는 방식입니다.



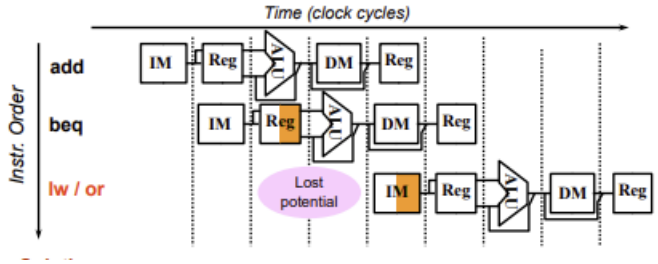
ALU의 계산 결과가 원래와 같으면 에서 WB으로 data가 적용되는 시점은 xor이 연산할 때이다. 따라서 xor 앞에 명령어는 다 data hazards 상태인 것을 알 수 있습니다.



따라서 이때 위의 그림과 같이 forwarding을 사용하여 계산이 끝난 것을 바로바로 다음 instruction에서의 ALU의 input으로 전달해줌으로써 stall을 방지하는 것입니다.

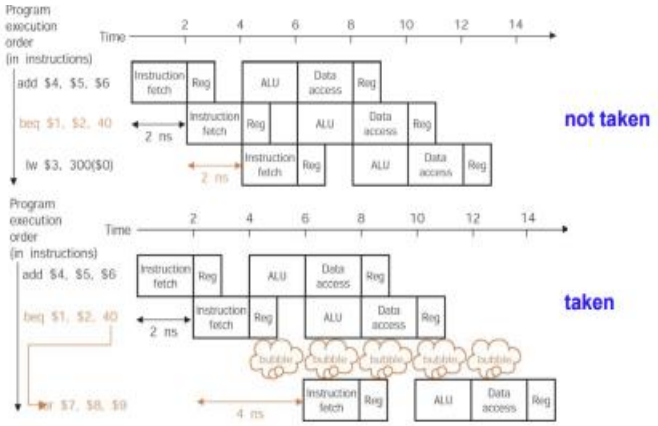


그런데 이때, 만약 add가 아니라 lw 명령어라면 ALU가 아닌 Dmem에서 r1이 결정되므로 어쩔 수 없이 stall 써야 하는 경우가 생깁니다.

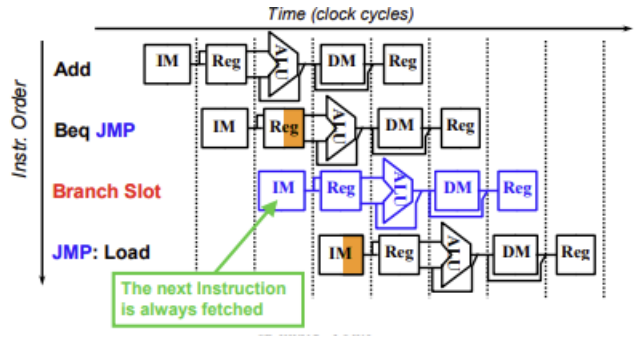


그리고 위와 같은 beq 다음 lw/or insturctions을 진행 하였을 때의 control hazard를 해결하기 위한 방법은 크게 다음과 같으며 1. Stall, 2. Prediction, 3. Delayed branch, 4. Add extra hardware 각각을 간단히 설명하면 다음과 같습니다.

1. stall 방법으로 간단하지만 성능이 별로인 편입니다.



2. Prediction은 Predict Branch Not Taken 즉, branch가 틀렸다고 가정하는 경우와 Predict Branch Taken 즉, branch가 맞다고 가정하는 경우로 볼 수 있습니다 주고 Taken방식이 확률이 높습니다. 그리고 Branch외의 Taken backwards Not taken Forwards은 taken backward 즉, for문이나 while문 같은 건 Taken이라 가정하는 것과 Not taken Forwards 즉, if문 같은 건 not Taken하는 것입니다. 보통 for문이나 while문이 반복할 확률이 높아서 성능 나쁘지 않은 편입니다.



3. Delayed branch는 순서를 바꾸어서 Branch 연산하는 동안 영향을 안 받는 다른 명령어 가져와서 먼저 수행하게 하는 것입니다.

4. Add extra hardware

Cpu를 두개로 두어 branch를 만나면 두개 다 쓰는데 cost대비 성능향상이 떨어지는 편이라고 볼 수 있습니다.

[검증 전략, 분석 및 결과]

<Nop operation을 적용한 결과의 cycle수 및 시간, simulation결과>

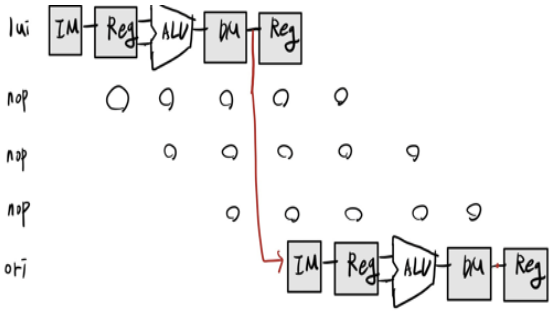
기존 코드에 nop을 적용한 결과와 run.bat을 하였을 때 결과입니다.

```
main: lui   $4, 0x0000
        nop
        nop
        nop
      ori   $4, $4, 0x2000
      ori   $5, $0, 100
        nop
        nop
      addi  $8, $0, 0x1

L1:   beq   $8, $5, done
      add   $9, $0, $8
        nop
        nop
        nop
      addi  $10, $8, -1

L2:   sll   $11, $9, 2
        nop
        nop
        nop
      add   $11, $4, $11
      nop
      nop
      nop
      lw    $12, 0($11)
      sll   $13, $10, 2
      nop
      nop
      nop
      add   $13, $4, $13
      nop
      nop
      nop
      lw    $14, 0($13)
      slt   $1, $12, $14
      nop
      beq   $1, $0, L3
      sw    $12, 0($13)
      sw    $14, 0($11)
      addi  $9, $9, -1
      nop
      nop
      nop
      addi  $10, $9, -1
      nop
      nop
      nop
      bgez  $10, L2

L3:   addi  $8, $8, 1
      j L1

done: break
```

```
C:\Users\leunb\Desktop\2023_ComputerArchitecture\prj3_PCPU
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enc
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enc
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not en
-----------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-----------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------
 Break signal: 0,   # of Cycles:        7000
-----------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 70115000 (1ps)
```

[구현한 code에 대한 설명]

Main 문에서 lui이후 ori가 진행 될 때



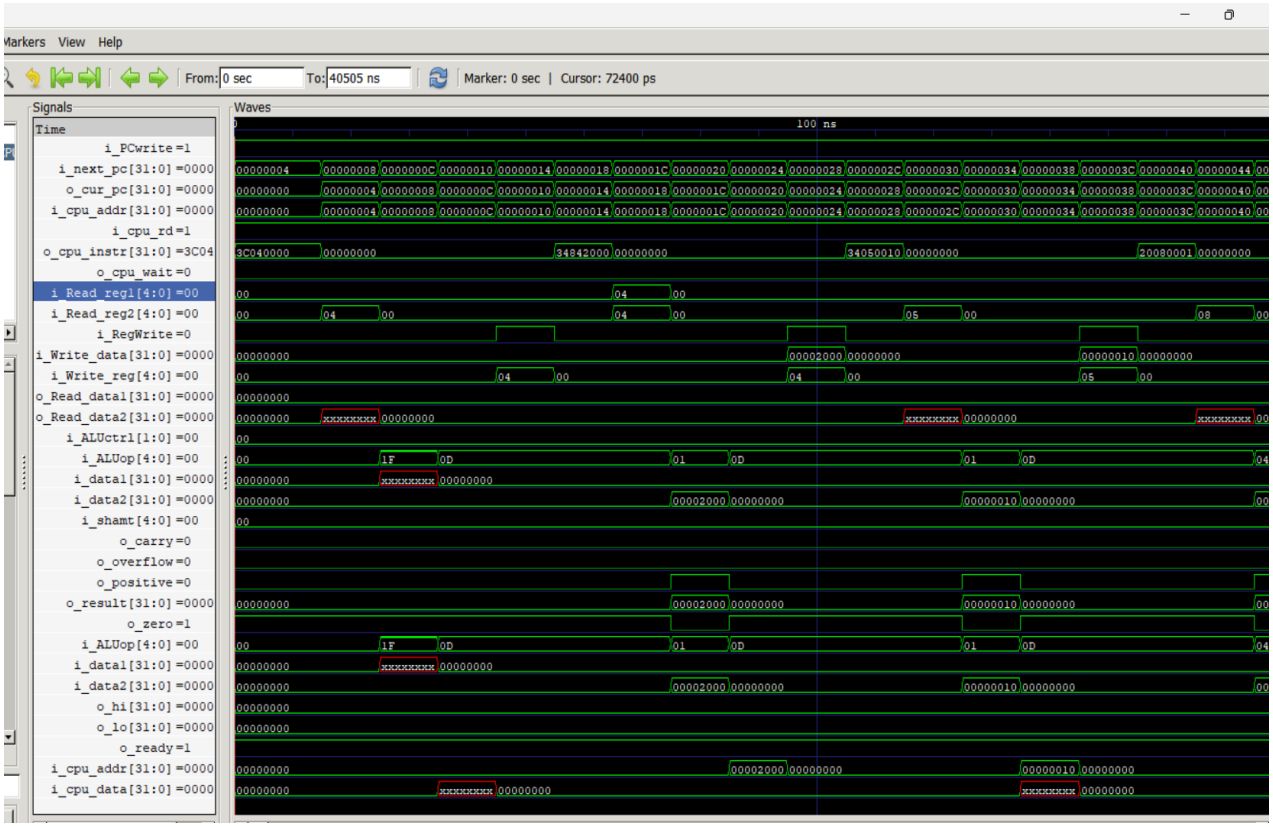
위의 그림과 같은 이유로 nop을 3개 추가합니다.

그 이후의 ori의 reg에서 쓰이는 register($0)를 그 다음 addi의 bport에서 사용하므로 nop 2개가 필요하고

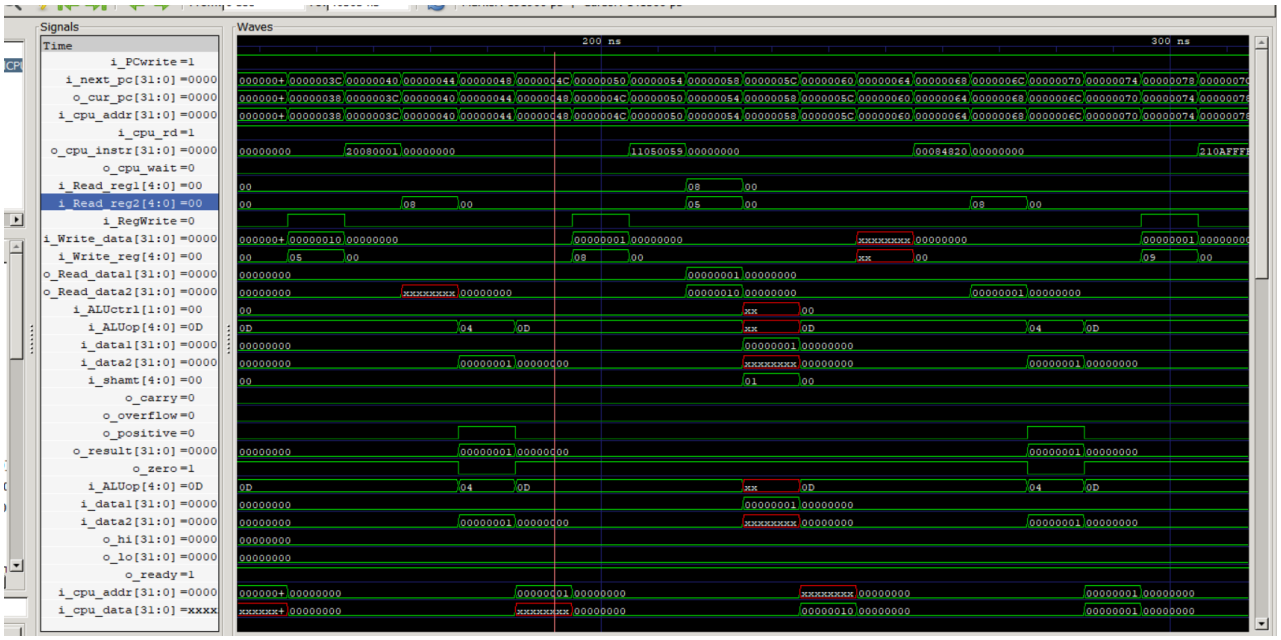
L1: 에서 add이후 addi의 경우에서도 register($8)이 이후 addi의 register에서 aport로 들어가므로 이때도 nop을 배치합니다. 또한 L2: 에서는 sll의 연산 결과 즉 writeback단계에서 의 결과가 다시 그 다음 operation인 add의 register 단자로 이어지므로 그 사이에도 nop 3개를 배치합니다 같은 이유로 이후의 sll와 add, add와 lw사이, addi 와 addi사이, addi 와 bgez 사이에도 nop 3개를 배치합니다.

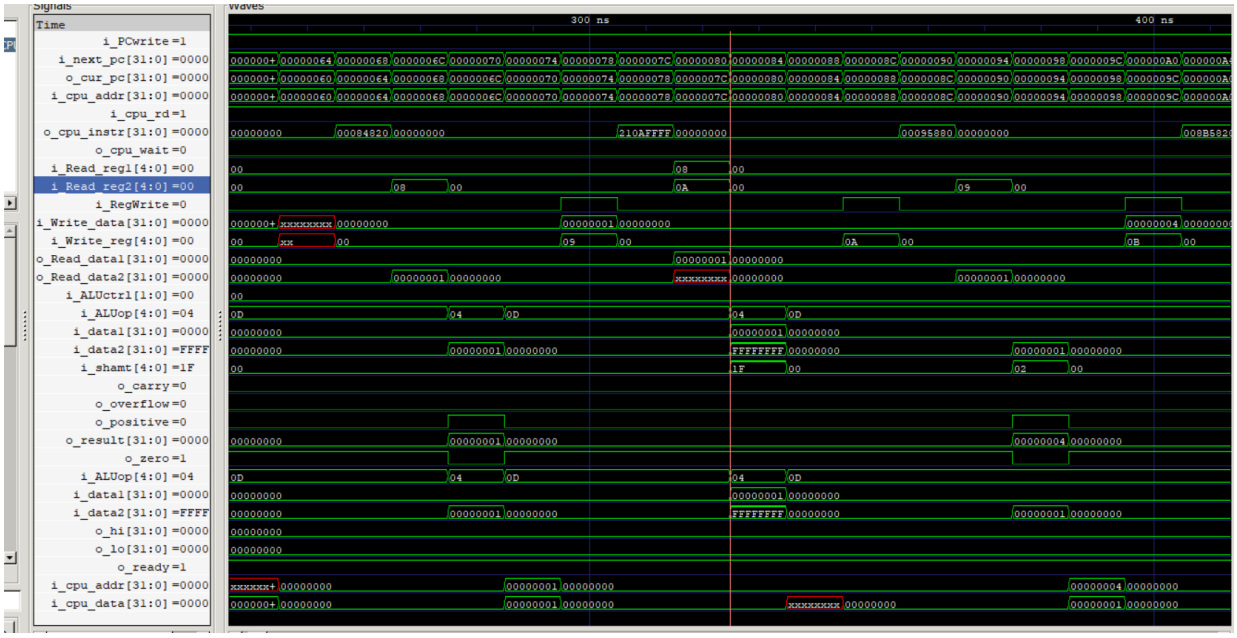
[시뮬레이션 결과 분석]

<nop을 추가한 결과>

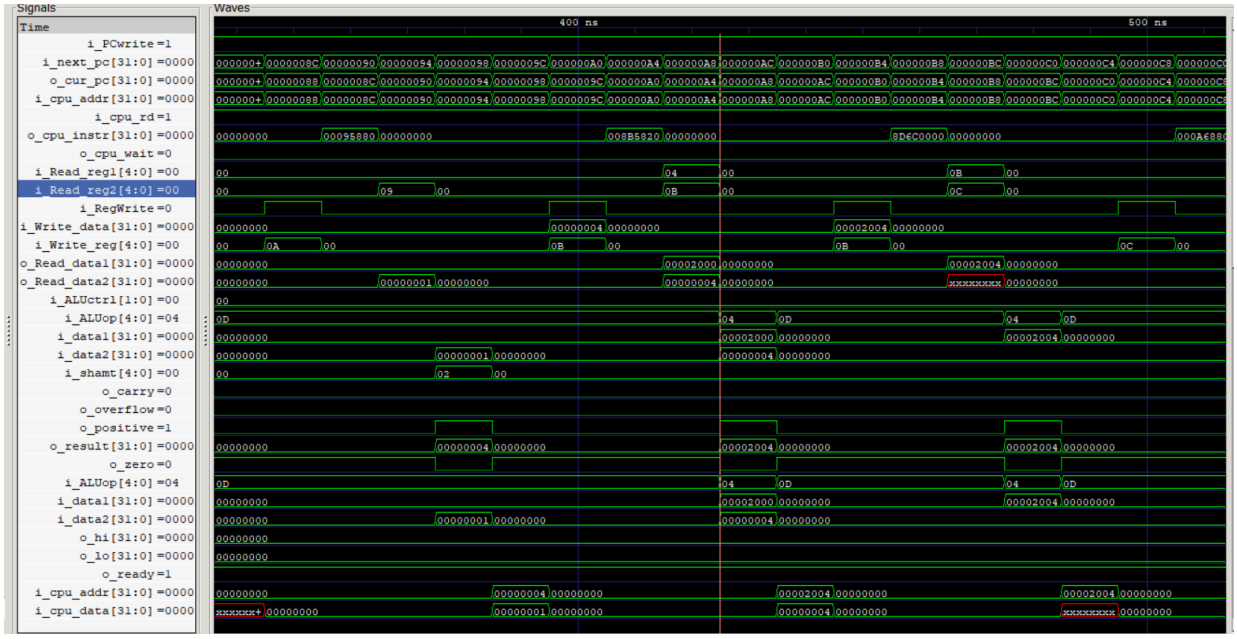


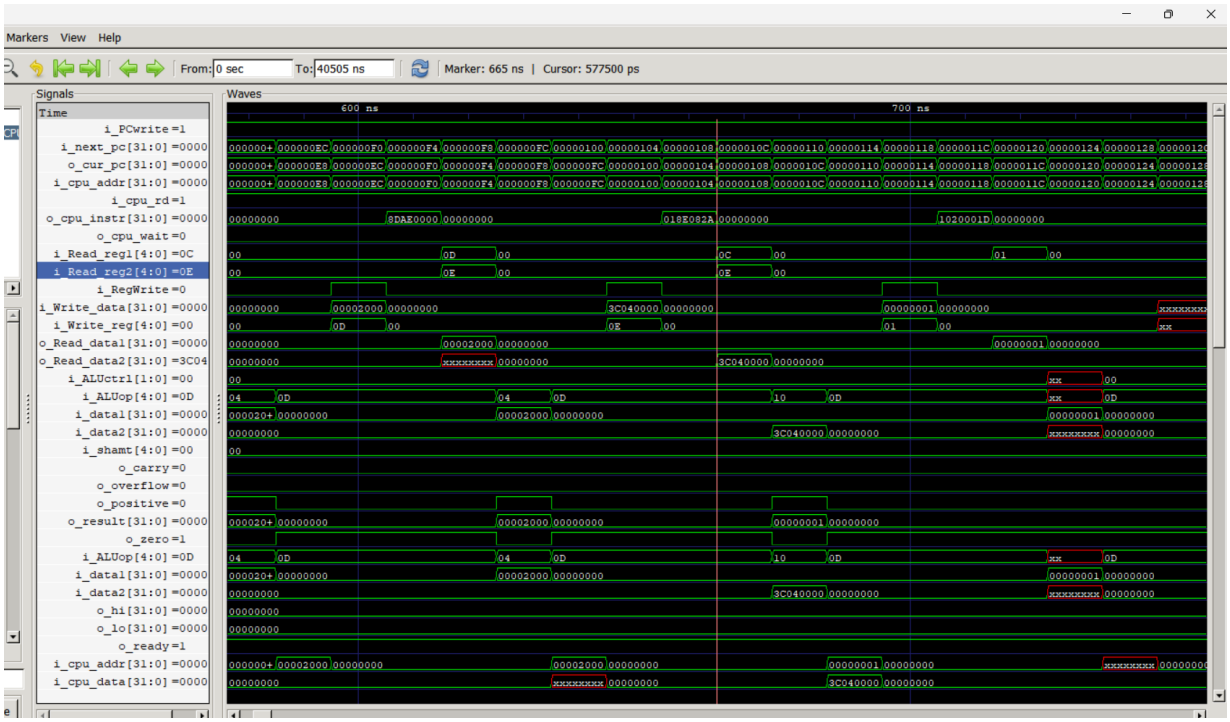
위의 그림은 main문의 내용이 수행된 시뮬레이션 결과 입니다.

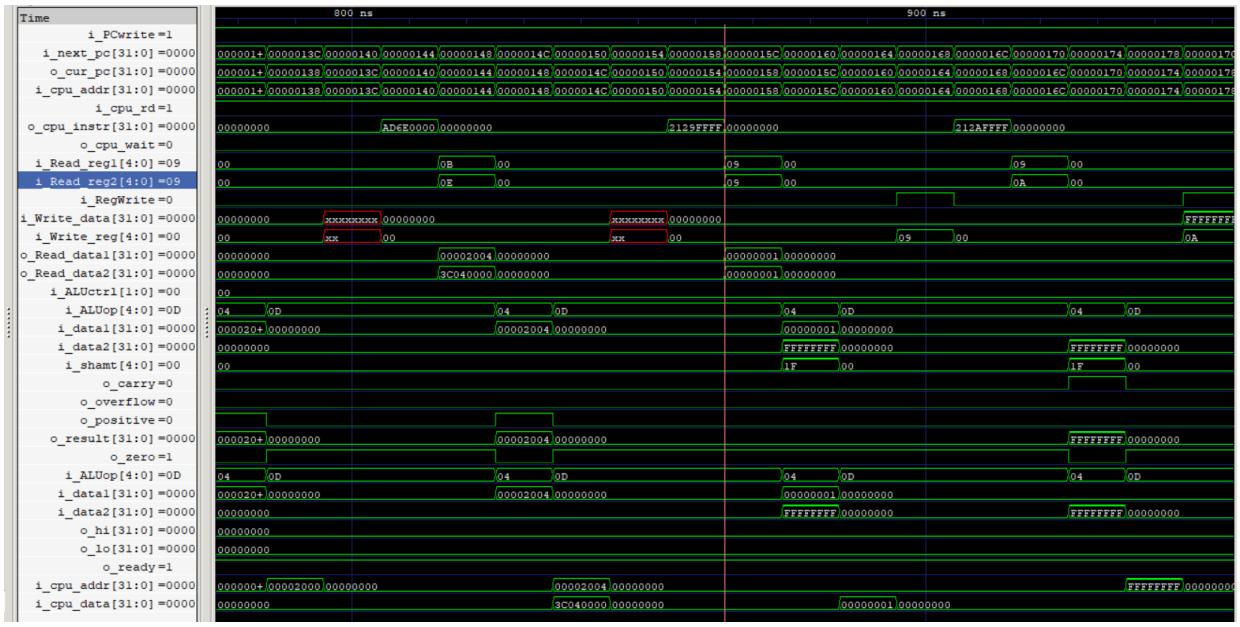


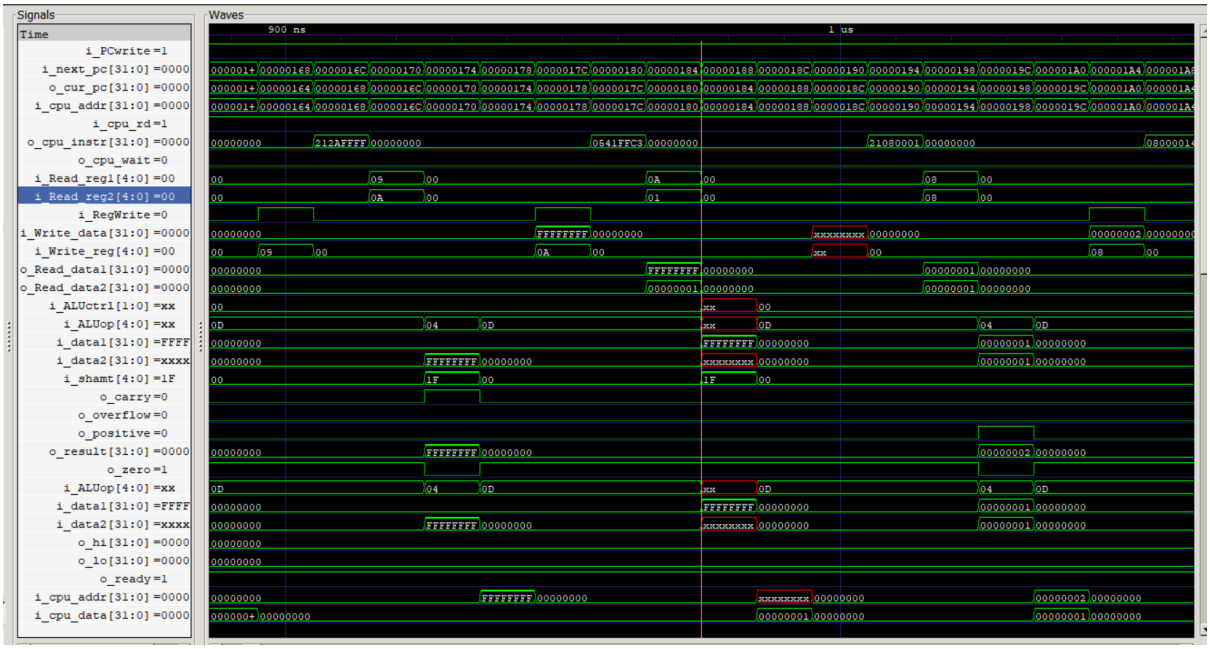


위의 시뮬레이션 결과들은 L1:의 내용이 수행된 결과 입니다.




Markers View Help
From: 0 sec
To: 40505 ns
Marker: 665 ns | Cursor: 577500 ps
Signals
Waves
600 ns
700 ns
Time
i_PCwrite =1
i_next_pc[31:0] =0000
o_cur_pc[31:0] =0000
i_cpu_addr[31:0] =0000
i_cpu_rd =1
o_cpu_instr[31:0] =0000
o_cpu_wait =0
i_Read_reg1[4:0] =0C
i_Read_reg2[4:0] =0E
i_RegWrite =0
i_Write_data[31:0] =0000
i_Write_reg[4:0] =00
o_Read_data1[31:0] =0000
o_Read_data2[31:0] =3C04
i_ALUctrl[1:0] =00
i_ALUop[4:0] =0D
i_data1[31:0] =0000
i_data2[31:0] =0000
i_shamt[4:0] =00
o_carry =0
o_overflow =0
o_positive =0
o_result[31:0] =0000
o_zero =1
i_ALUop[4:0] =0D
i_data1[31:0] =0000
i_data2[31:0] =0000
o_hi[31:0] =0000
o_lo[31:0] =0000
o_ready =1
i_cpu_addr[31:0] =0000
i_cpu_data[31:0] =0000


800 ns
900 ns
Time
i_PCwrite =1
i_next_pc[31:0] =0000
o_cur_pc[31:0] =0000
i_cpu_addr[31:0] =0000
i_cpu_rd =1
o_cpu_instr[31:0] =0000
o_cpu_wait =0
i_Read_reg1[4:0] =09
i_Read_reg2[4:0] =09
i_RegWrite =0
i_Write_data[31:0] =0000
i_Write_reg[4:0] =00
o_Read_data1[31:0] =0000
o_Read_data2[31:0] =0000
i_ALUctrl[1:0] =00
i_ALUop[4:0] =0D
i_data1[31:0] =0000
i_data2[31:0] =0000
i_shamt[4:0] =00
o_carry =0
o_overflow =0
o_positive =0
o_result[31:0] =0000
o_zero =1
i_ALUop[4:0] =0D
i_data1[31:0] =0000
i_data2[31:0] =0000
o_hi[31:0] =0000
o_lo[31:0] =0000
o_ready =1
i_cpu_addr[31:0] =0000
i_cpu_data[31:0] =0000

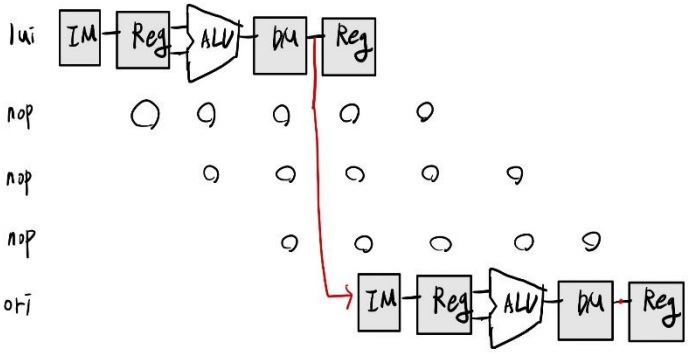


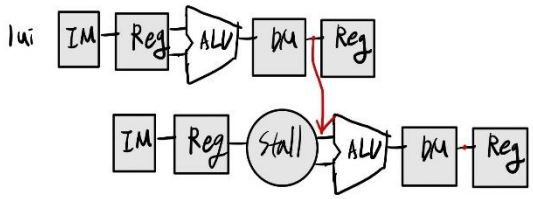
위의 시뮬레이션 결과들은 L2: 의 내용이 수행된 결과입니다.

<forwarding을 통한 nop 제거 계획 및 코드내용, 사이클 수, 시뮬레이션 결과>

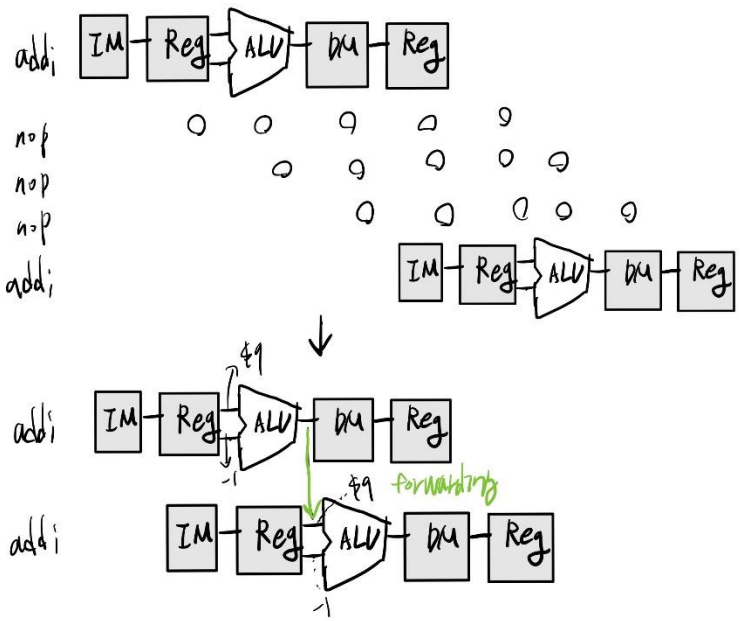
Main문 내부에서 nop대신 forwarding을 적용한다면 다음과 같이 진행 할 수 있다고 생각하였습니다.

기존에 nop을 사용한다면 아래 그림의 위의 경우이지만 forwarding을 적용하면 아래 그림의 아래 경우와 같다고 생각할 수 있습니다.

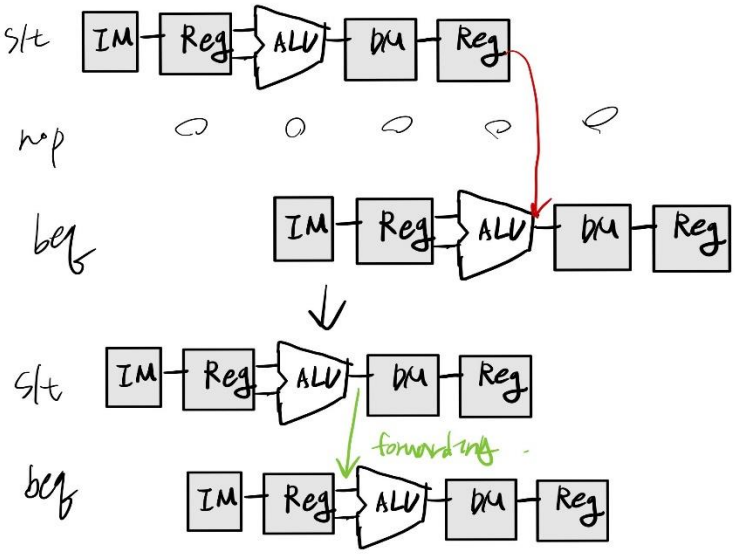




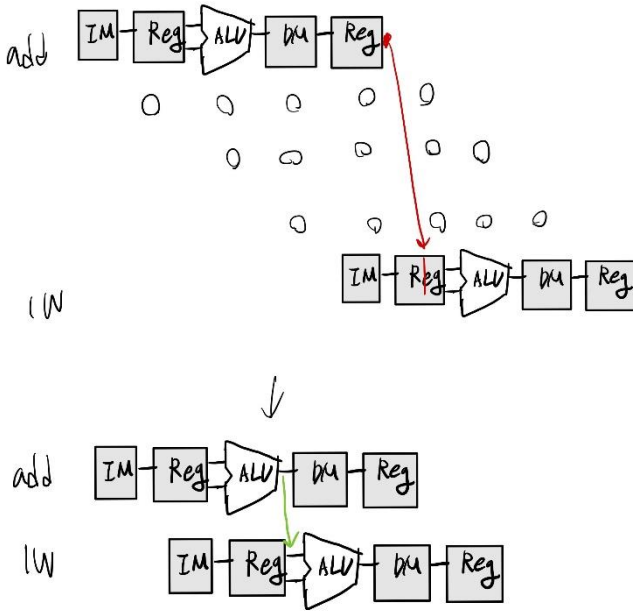
기존에 nop을 사용한다면 아래 그림의 위의 경우이지만 forwarding을 적용하면 아래 그림의 아래 경우와 같다고 생각할 수 있습니다.



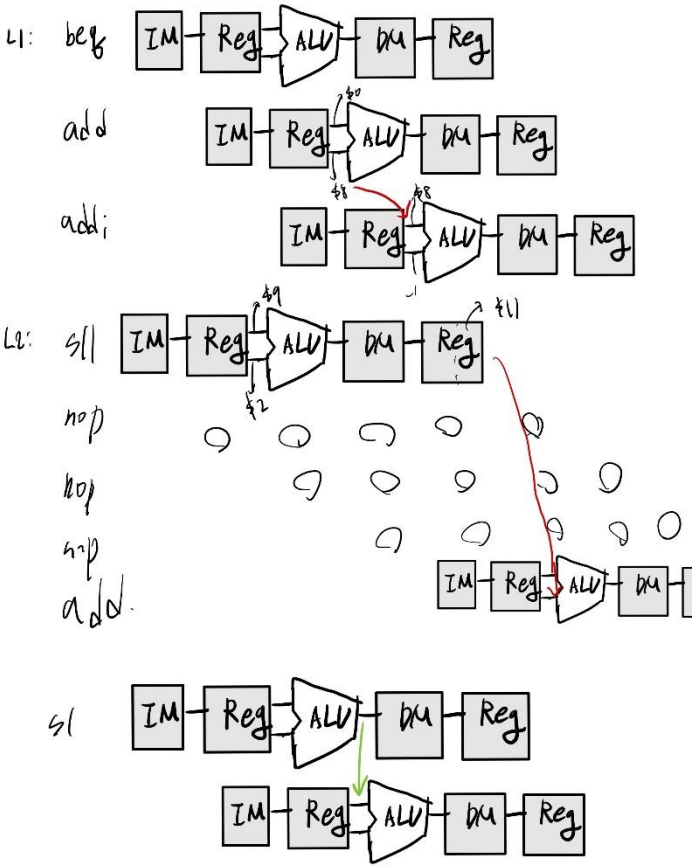
기존에 nop을 사용한다면 아래 그림의 위의 경우이지만 forwarding을 적용하면 아래 그림의 아래 경우와 같다고 생각할 수 있습니다.



기존에 nop을 사용한다면 아래 그림의 위의 경우이지만 forwarding을 적용하면 아래 그림의 아래 경우와 같다고 생각할 수 있습니다.



기존에 nop을 사용한다면 아래 그림의 위의 경우이지만 forwarding을 적용하면 아래 그림의 아래 경우와 같다고 생각할 수 있습니다.



위의 결과를 이용한 경우의 코드는 다음과 같습니다.

[코드내용/cycle수/시뮬레이션 화면]

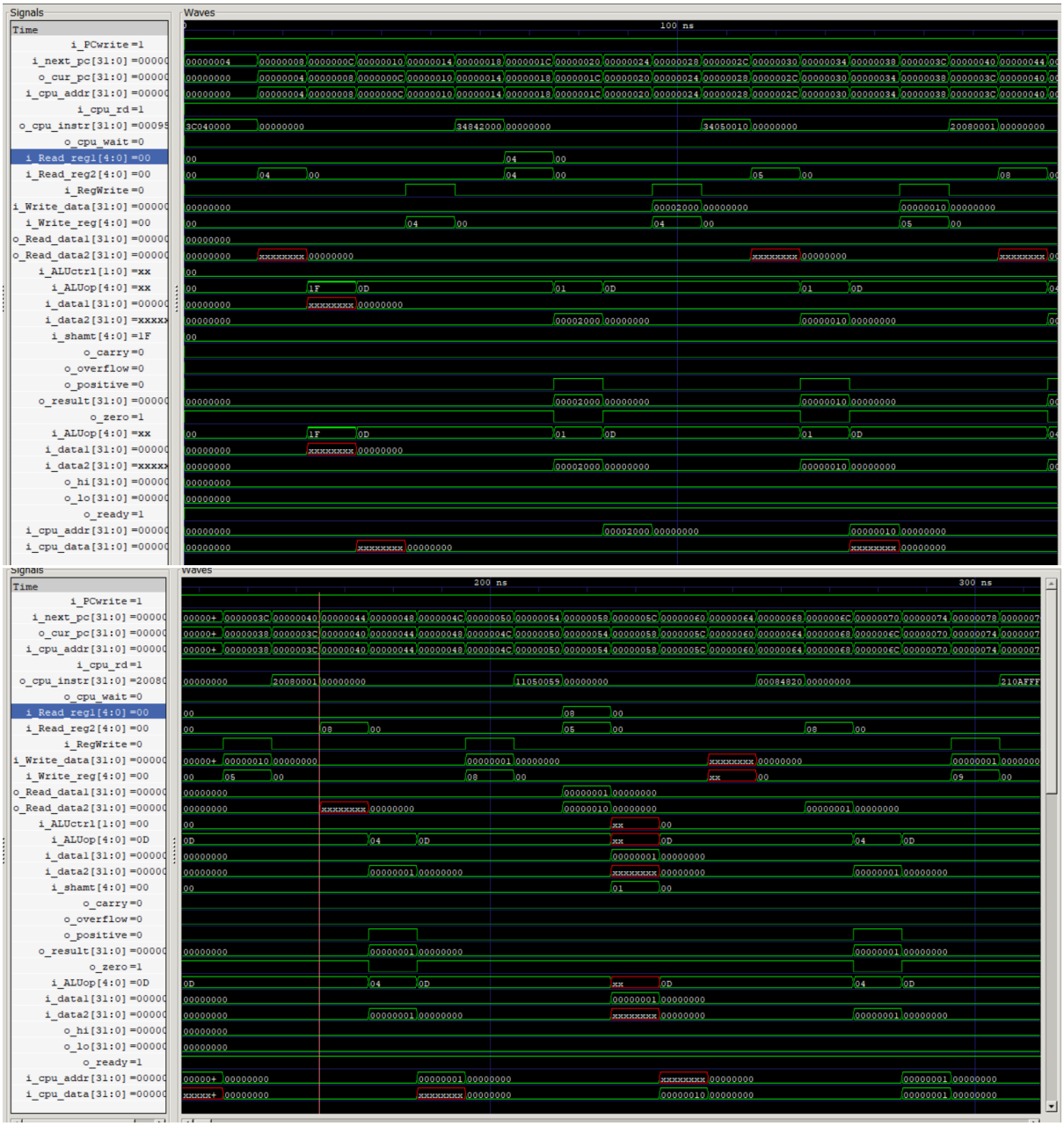
```
main: lui   $4, 0x0000
       nop
       #forwarding 10
      ori   $4, $4, 0x2000
      ori   $5, $0, 100
         #forwarding 10
      addi  $8, $0, 0x1

L1:   beq   $8, $5, done
      add   $9, $0, $8
     #forwarding 00
     nop
     nop
     nop
      addi  $10, $8, -1

L2:   sll   $11, $9, 2
       #forwarding 10 to B port
      add   $11, $4, $11
      #forwarding 10 to Aprot
      lw    $12, 0($11)

       #same
      sll   $13, $10, 2
      #forwarding 10 to Bport

     add   $13, $4, $13
      #forwarding 10
     lw    $14, 0($13)

      slt   $1, $12, $14
      nop
      nop
      beq   $1, $0, L3

      sw    $12, 0($13)
      sw    $14, 0($11)

      addi  $9, $9, -1
      #forwarding 10 to Aport
      addi  $10, $9, -1
      nop
      nop
      bgez  $10, L2

L3:   addi  $8, $8, 1
      j L1

done: break
```

```
-------------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:        5493
-------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 55045000 (1ps)
```

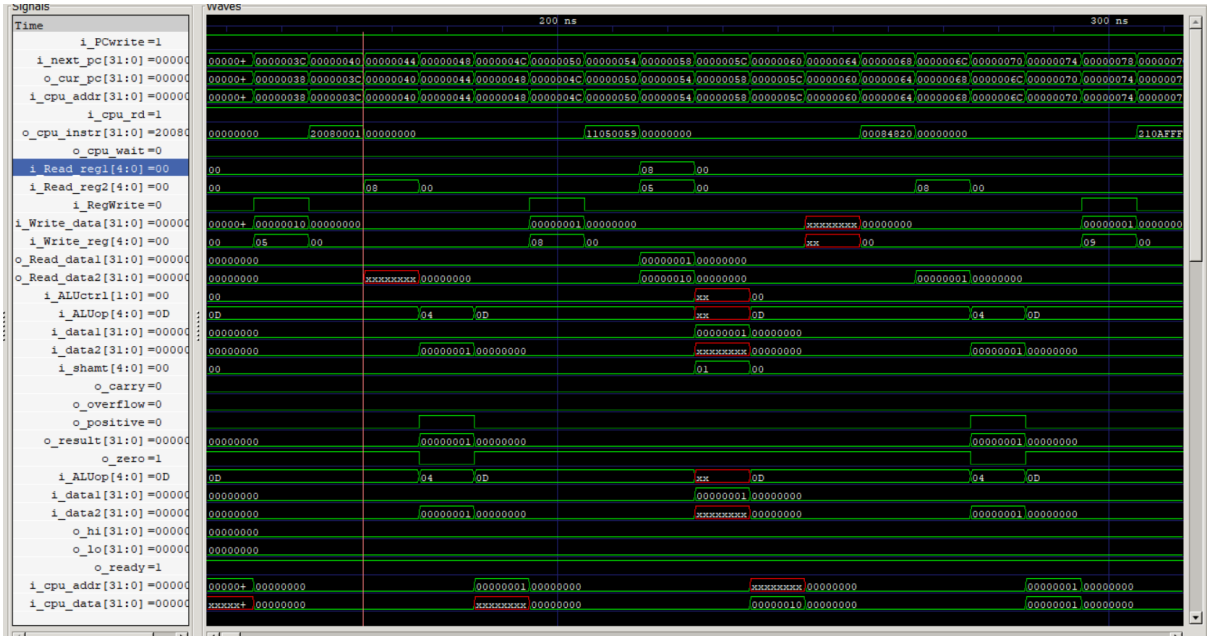
위에서 언급한 내용을 바탕으로 nop을 줄일 수 있도록 하였습니다.

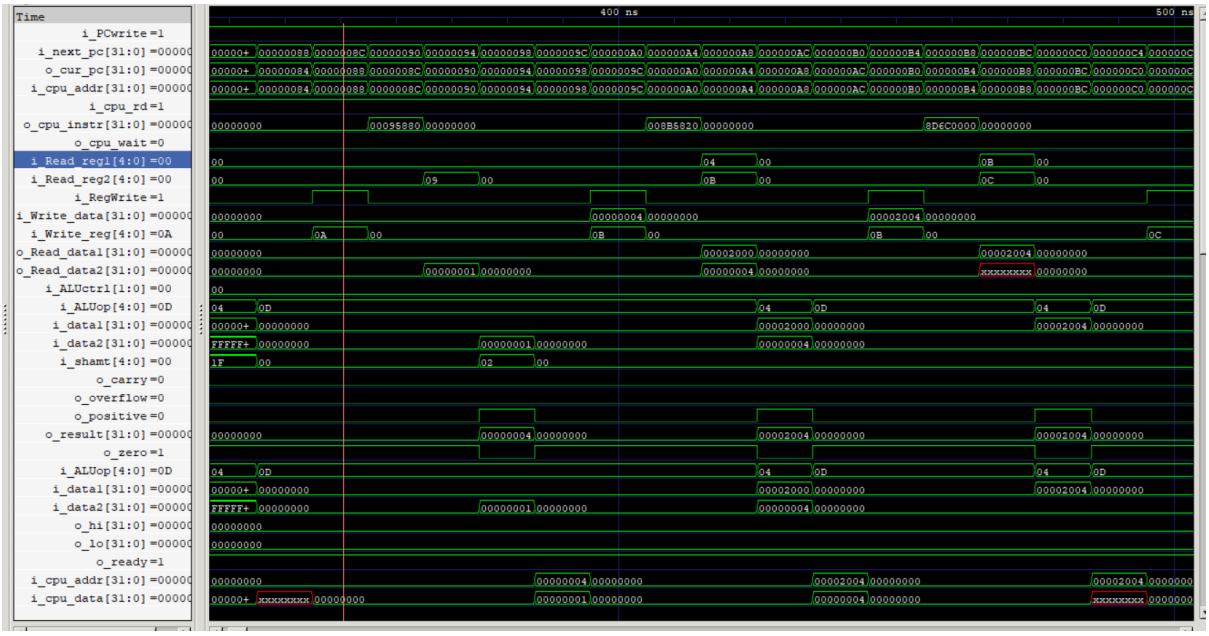
cycle수는 5493으로 줄었지만 L1에서인지 break signal이 존재하는 것을 확인하였습니다.

아래는 시뮬레이션 결과입니다.

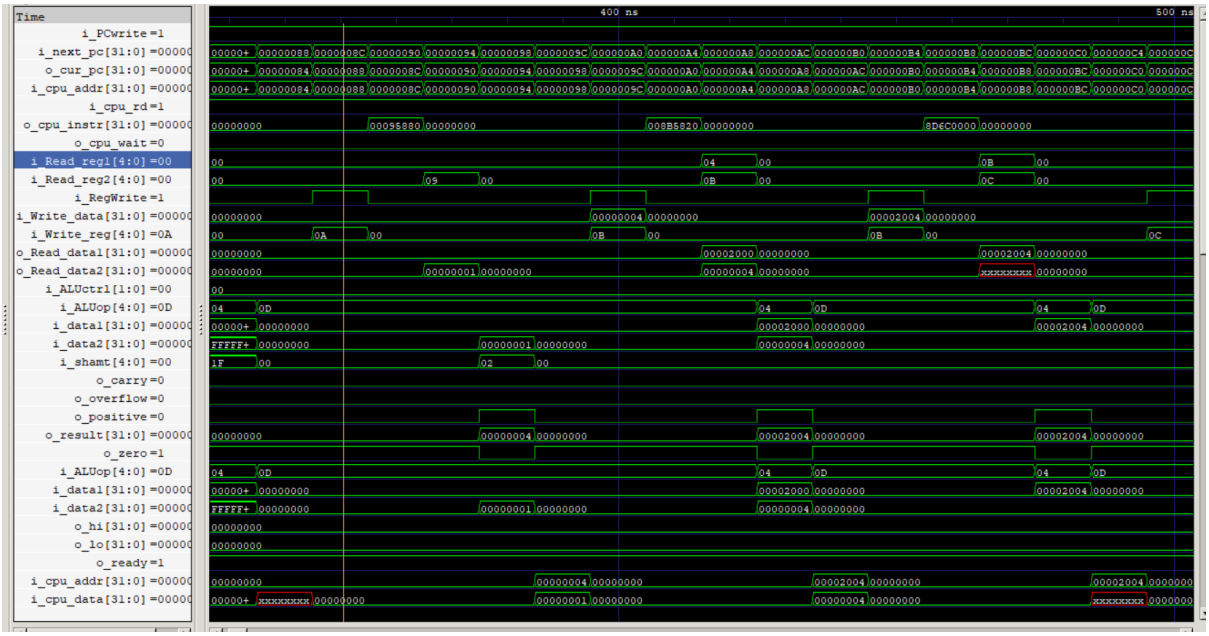


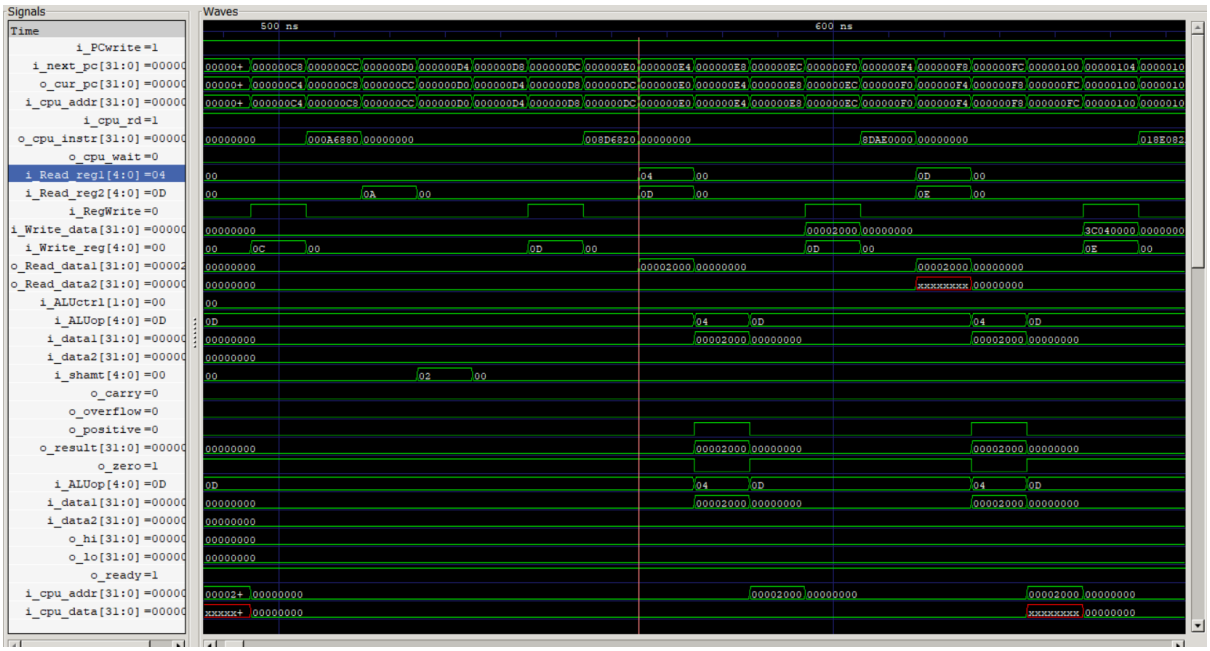
위의 내용은 main문에서의 진행된 내용입니다.

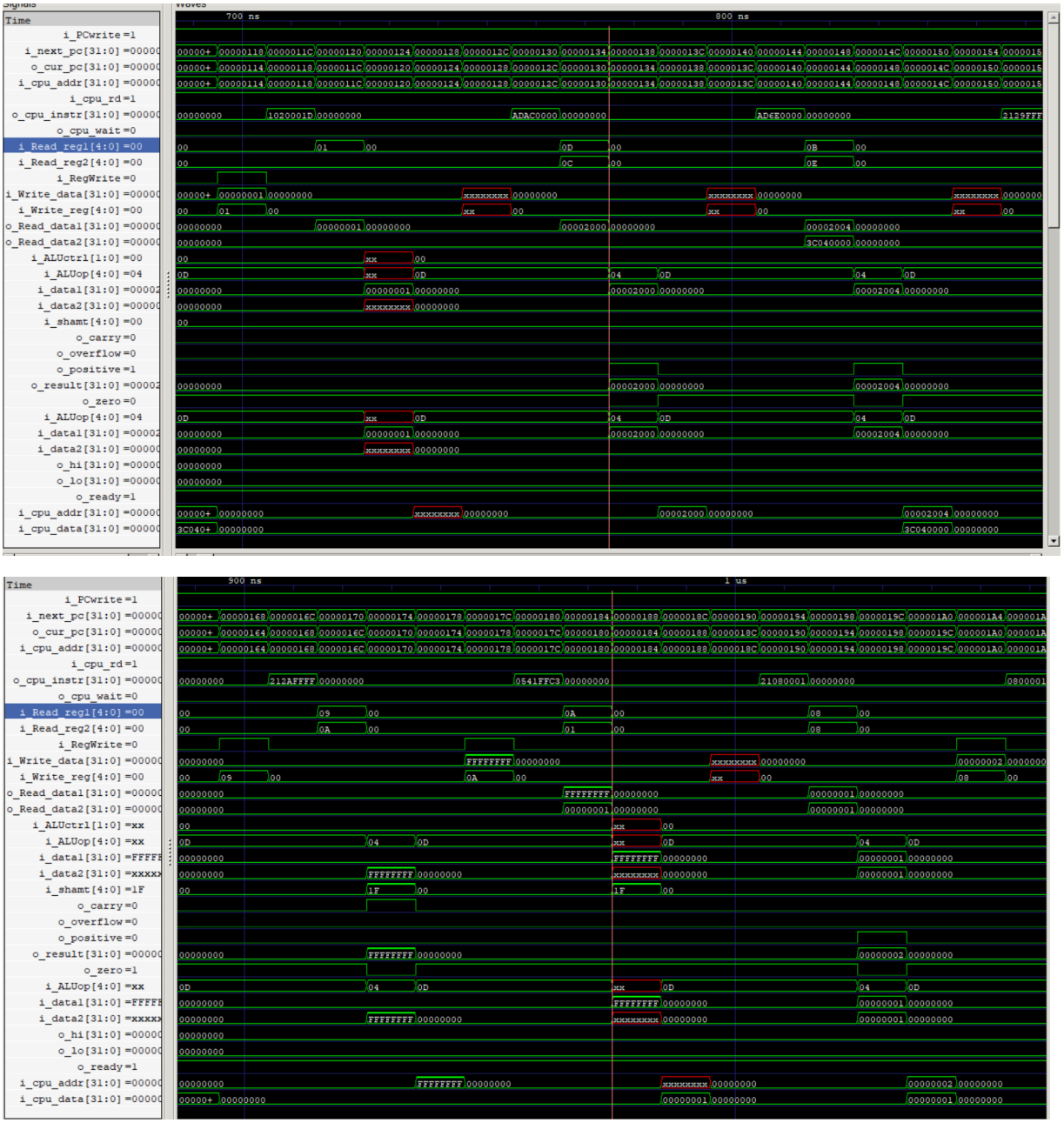




위의 내용은 L1:이 진행된 결과입니다.


Time
i_PCwrite =1
i_next_pc[31:0] =00000
o_cur_pc[31:0] =00000
i_cpu_addr[31:0] =00000
i_cpu_rd =1
o_cpu_instr[31:0] =00000
o_cpu_wait =0
i_Read_reg1[4:0] =00
i_Read_reg2[4:0] =00
i_RegWrite =1
i_Write_data[31:0] =00000
i_Write_reg[4:0] =0A
o_Read_data1[31:0] =00000
o_Read_data2[31:0] =00000
i_ALUctrl[1:0] =00
i_ALUop[4:0] =0D
i_data1[31:0] =00000
i_data2[31:0] =00000
i_shamt[4:0] =00
o_carry =0
o_overflow =0
o_positive =0
o_result[31:0] =00000
o_zero =1
i_ALUop[4:0] =0D
i_data1[31:0] =00000
i_data2[31:0] =00000
o_hi[31:0] =00000
o_lo[31:0] =00000
o_ready =1
i_cpu_addr[31:0] =00000
i_cpu_data[31:0] =00000
400 ns
500 ns


Signals
Waves
Time
i_PCwrite =1
i_next_pc[31:0] =00000
o_cur_pc[31:0] =00000
i_cpu_addr[31:0] =00000
i_cpu_rd =1
o_cpu_instr[31:0] =00000
o_cpu_wait =0
i_Read_reg1[4:0] =04
i_Read_reg2[4:0] =0D
i_RegWrite =0
i_Write_data[31:0] =00000
i_Write_reg[4:0] =00
o_Read_data1[31:0] =00002
o_Read_data2[31:0] =00000
i_ALUctrl[1:0] =00
i_ALUop[4:0] =0D
i_data1[31:0] =00000
i_data2[31:0] =00000
i_shamt[4:0] =00
o_carry =0
o_overflow =0
o_positive =0
o_result[31:0] =00000
o_zero =1
i_ALUop[4:0] =0D
i_data1[31:0] =00000
i_data2[31:0] =00000
o_hi[31:0] =00000
o_lo[31:0] =00000
o_ready =1
i_cpu_addr[31:0] =00000
i_cpu_data[31:0] =00000
500 ns
600 ns



위의 내용은 L2: 이 진행된 결과입니다.

■ 문제점 및 고찰

Nop을 최대한 제거하여 시뮬레이션을 한 결과, cycle수와 시간은 줄어 들었지만 break signal이 발생하였고, 반복해서 부분 수정하였으나 고쳐지지 않았습니다. operation사이의 forwarding에 대한 이해가 부족한 것일 수도, port와 forwarding되는 비트를 잘못 써넣은 것 중의 원인이 있다 생각되는데 충분한 시간이 있었다면 각 nop마다 확인할 수 있을 수도 있었지만 시간적인 여유가 많지 않아 그렇게 하지는 못했습니다. 또한 이번 과제를 통해 pipeline,hazard,그리고 그에 맞는 해결방법을 알아보고 공부할 수 있었던 것 같습니다. 그리고 확실히 branch에 대한 처리가 까다로워 그냥 nop으로 처리하게 된 부분이 아쉬운 것 같습니다.