

컴퓨터구조 3차 프로젝트

컴퓨터정보공학부

2023202083 최우진

수요일 실습 수강

1. Introduction

파이프라인 구조를 갖는 processor를 이용해 주어진 Radix Sort 코드의 성능을 개선하는 시뮬레이션을 수행하였다. 파이프라인 구조에서는 명령어 간 데이터 또는 제어 의존성으로 인해 hazard가 발생하고, 이를 회피하기 위해 NOP 명령어가 삽입된다. 그러나 이러한 NOP은 실행 주기를 낭비하게 되므로, 성능 개선을 위해 제거가 필요하다. 첫 번째 시뮬레이션에서는 명령어 순서를 유지한 채 불필요한 NOP만 제거하였다. Radix Sort 코드 전반에 걸쳐 명령어 간의 데이터 및 제어 의존성을 분석하고, hazard가 발생하지 않는 구간에서 NOP을 제거하여 전체 cycle 수를 줄일 수 있었다. 두 번째 시뮬레이션에서는 ALU 데이터 forwarding 제어 신호를 수동으로 삽입하여 더 많은 NOP을 제거하였다. Forwarding을 통해 연산 결과를 register에 기록하지 않고도 바로 다음 명령어에서 사용할 수 있게 하여, 데이터 대기를 최소화하였다. 이로 인해 실행 속도 역시 더욱 향상되었다. 이번 과제를 통해 pipeline 구조에서의 hazard 처리 방식과 forwarding의 활용 효과를 직접 실험하고 분석해 볼 수 있었으며, 소프트웨어적 코드 최적화가 하드웨어 성능에 어떤 영향을 미치는지를 구체적으로 체감할 수 있었다.

2. 실험 내용

1. Hazard

Pipeline은 기존 Multi-cycle processor에서 하나의 명령어를 여러 단계로 나누어 처리하던 구조를 확장하여, 각 stage마다 서로 다른 명령어를 동시에 실행함으로써 전체 시스템의 throughput을 높이는 방식이다. 하지만 이러한 구조는 기존의 Multi-cycle 방식에서는 존재하지 않던 새로운 문제들을 유발하며, 주로 세 가지 hazard로 분류된다. 즉, Structural Hazard, Data Hazard, 그리고 Control Hazard이다. Structural Hazard는 하드웨어 자원의 공유에서 발생하는 문제로, 프로세서 내부의 기능 유닛을 동시에 여러 명령어가 사용하려고 할 때 생긴다. 예를 들어 Multi-cycle processor에서는 명령어에 따라 수행 cycle 수가 달라 R-type 명령어는 4 cycle, branch는 3 cycle, read 명령어는 5 cycle 등으로 다양하다. 이러한 상태에서 pipeline 구조로 그대로 옮기면 특정 기능 유닛이 동시에 여러 명령어에서 요구되는 경우 충돌이 발생하게 된다. 이를 방지하기 위해 pipeline processor에서는 대부분의 명령어를 고정된 5단계로 분리하여, 각 stage가 동일한 시점에 하나의 유닛만 접근하도록 설계한다. 단, branch 명령어는 예외적으로 3 cycle만을 사용하기도 한다. Data Hazard는 명령어 간의 데이터 흐름에 따라 발생하는 의존성 문제로, 주로 register의 읽기

및 쓰기 순서가 꼬이면서 발생한다. 대표적인 유형으로는 Read After Write, Write After Write, Write After Read 등이 있으며, 이 중 Read After Write가 가장 일반적인 형태이다. 예를 들어 첫 번째 명령어가 register에 값을 쓰기도 전에 다음 명령어가 같은 register의 값을 읽으려 할 경우, 원하는 최신 값이 아닌 이전 값이 읽히는 문제가 발생한다. 이를 해결하기 위한 기본적인 방법으로는 stall을 통해 지연시키는 방법이 있으며, 보다 고급 방식으로는 forwarding unit을 통해 이전 단계의 연산 결과를 직접 가져오는 방식이 활용된다. Control Hazard는 프로그램 흐름이 분기되는 경우, 즉 jump나 branch 명령어가 등장할 때 발생하는 문제이다. 분기 조건의 결과에 따라 이후 명령어를 계속 실행할지 버릴지를 결정해야 하므로, 이 판단이 끝나기 전까지는 다음 명령어를 확정적으로 실행할 수 없다. 기본적인 해결책은 stall을 사용하는 것이지만, 이 경우 일반적으로 2 cycle의 성능 손실이 발생한다. 이를 개선하기 위해 EX 단계가 아닌 ID 단계에서 분기 여부를 판단하는 구조를 취하면 손실을 1 cycle로 줄일 수 있다. 더 나아가 현대의 프로세서에서는 branch prediction을 도입하여, 분기 여부를 미리 예측함으로써 stall 자체를 줄이는 방향으로 발전하고 있다.

2. S/W-forwarding

S/W-forwarding은 파이프라인 상에서 발생하는 hazard를 해결하기 위한 가장 기본적인 방법으로, 명령어 사이에 stall 명령어, 즉 nop을 삽입하여 데이터가 준비될 때까지 다음 명령어의 실행을 지연시키는 방식이다. 구조는 단순하지만, stall이 많아질수록 전체적인 throughput이 감소하게 되는 단점이 있다. 따라서 이 방식의 핵심은 hazard를 회피하면서도 가능한 한 stall의 개수를 최소화하는 데 있다. 일반적으로 processor가 어떠한 상황에서도 문제 없이 작동하도록 하기 위해 고정적으로 4개의 stall을 삽입하는 방법이 사용되기도 한다. 하지만 이러한 방식은 매우 비효율적이며, 불필요한 지연을 유발해 성능 저하로 이어진다. 보다 효율적인 방법은 pipeline 상에 필요한 데이터가 어느 시점에 존재하는지를 분석하여, 해당 데이터가 실제로 연산에 사용될 수 있을 때까지 필요한 만큼만 stall을 삽입하는 것이다. 또한, 명령어의 순서를 재배치하는 방식도 존재한다. 연관성이 없는 명령어들을 hazard가 발생하는 명령어 사이에 끼워 넣음으로써 nop의 삽입 자체를 피할 수 있으며, 이 역시 throughput 향상에 효과적인 전략이다. 이러한 방식들은 모두 소프트웨어 차원에서 코드의 실행 흐름을 조정함으로써 하드웨어적인 forwarding 없이도 hazard를 해결할 수 있도록 돕는다.

3. H/W-forwarding

H/W-forwarding은 pipeline 내부에 이미 존재하는 데이터를 활용하여 hazard를 해결하는 방식으로, 접근하려는 register의 값이 아직 메모리에 기록되지 않았더라도 해당 데이터를 forwarding unit과 multiplexer를 통해 직접 가져와 사용할 수 있도록 한다. 이 방식은 연산 결과를 기다릴 필요 없이 다음 명령어가 곧바로 실행될 수 있게 해주기 때문에, stall을 크게 줄일 수 있다는 장점이 있다. 물론 하드웨어적으로 별도의 제어 회로와 선택기를 추가해야 하므로 구조가 복잡해지고, 전체 회로의 크기와 설계 비용이 증가한다는 단점도 존재한다. 그럼에도 불구하고, pipeline의 효율성과 throughput을 높이는 데 있어 매우 효과적이기 때문에 실제 프로세서 설계에서는 널리 사용되는 기법이다. 특히 고성능 시스템에서는 H/W-forwarding을 통해 거의 모든 데이터 hazard를 해결하고, 성능 저하 없이 명령어 흐름을 유지할 수 있다.

3. 검증 전략, 분석 및 결과

1. 구현한 assembly code 설명

1-1 코드 설명

```
.text
```

```
.globl main
```

```
main:
```

```
    addi $2, $0, 0x2000    # $2 = 배열 시작 주소
```

```
    addi $3, $0, 16       # $3 = 배열 크기
```

```
    add  $4, $0, $3       # $4 = i ← size ($3 → $4) → 이후 L1  
에서 사용됨
```

```
L1:
```

```
    srl  $4, $4, 1        # i = i / 2 (RAW: $4 from add)
```

```
    blez $4, done        # 제어 hazard: $4 → ID에서 분기 판  
단
```

```
    add  $5, $4, $0       # j ← i (RAW: $4 → $5)
```

```
L2:
```

```
    slt  $1, $5, $3       # if j < size (RAW: $5, $3 → $1)
```

```
    beq  $1, $0, L1       # 제어 hazard: $1 → ID에서 분기 판  
단
```

```

        sll  $6, $5, 2          # offset = j * 4 (RAW: $5 → $6)
        add  $6, $6, $2          # addr = base + offset (RAW: $6, $2
→ $6)
        lw   $7, 0($6)          # $7 = arr[j] (RAW: $6 → $7)
        add  $8, $5, $0          # k ← j (RAW: $5 → $8)

L3:
        sub  $9, $8, $4          # $9 = k - i (RAW: $8, $4 → $9)
        bltz $9, L4              # 제어 hazard: $9 → ID에서 분기 판단
        sll  $10, $9, 2          # offset = (k - i) * 4 (RAW: $9 → $10)
        add  $10, $2, $10        # addr = base + offset (RAW: $10, $2
→ $10)
        lw   $11, 0($10)         # $11 = arr[k] (RAW: $10 → $11)
        slt  $1, $7, $11         # if arr[j] < arr[k] (RAW: $7, $11 → $1)
        beq  $1, $0, L4          # 제어 hazard: $1 → ID에서 분기 판
단
        sw   $11, 0($6)          # arr[j] = arr[k] (RAW: $11, $6)
        add  $6, $10, $0          # $6 ← addr of k (RAW: $10)
        add  $8, $9, $0          # k ← k - 1 (RAW: $9 → $8)
        j    L3                  # jump to loop

L4:
        sw   $7, 0($6)          # arr[k] ← original arr[j] (RAW: $7, $6)
        addi $5, $5, 1          # j++
        j    L2                  # next outer loop

```

done:

break

Assembly 코드는 주어진 16개의 정수 배열에 대해 내림차순으로 정렬하는 Shell Sort 알고리즘을 구현한 것이다. 정렬 대상 데이터는 .data 세그먼트에 정의되어 있으며, .text 영역의 main:부터 시작되는 명령어 흐름은 총 세 개의 중첩 루프 구조로 이루어진다. 외부 루프는 정렬 범위를 점차 줄여가며 반복하고, 내부 루프는 인접한 데이터를 비교하여 더 큰 값이 앞에 오도록 위치를 바꾸는 방식으로 동작한다. 이 과정에서 사용하는 레지스터는 다음과 같다. \$2는 배열의 시작 주소를 저장하며, \$3는 배

열 크기, \$4는 정렬 범위를 나타내는 인덱스 i, \$5는 현재 비교 위치 j, \$6는 주소 계산용 임시 저장소, \$7, \$11은 비교되는 값, \$8, \$9, \$10은 루프 변수 및 인덱스 계산에 사용된다. 조건 분기 명령어로는 blez, beq, bltz 등이 사용되어 루프 제어 및 조건 판단을 수행한다.

1-2 Dependency

```
.text
.globl main

main:
    addi $2, $0, 0x2000    # $2 = 배열 시작 주소
    addi $3, $0, 16       # $3 = 배열 크기
    add $4, $0, $3        # $4 = i ← size ($3 → $4) → 이후 L1에서 사용됨

L1:
    srl $4, $4, 1         # i = i / 2 (RAW: $4 from add)
    blez $4, done         # 제어 hazard: $4 → ID에서 분기 판단
    add $5, $4, $0        # j ← i (RAW: $4 → $5)

L2:
    slt $1, $5, $3        # if j < size (RAW: $5, $3 → $1)
    beq $1, $0, L1        # 제어 hazard: $1 → ID에서 분기 판단
    sll $6, $5, 2         # offset = j * 4 (RAW: $5 → $6)
    add $6, $6, $2        # addr = base + offset (RAW: $6, $2 → $6)
    lw $7, 0($6)         # $7 = arr[j] (RAW: $6 → $7)
    add $8, $5, $0        # k ← j (RAW: $5 → $8)

L3:
    sub $9, $8, $4        # $9 = k - i (RAW: $8, $4 → $9)
    bltz $9, L4           # 제어 hazard: $9 → ID에서 분기 판단
    sll $10, $9, 2        # offset = (k - i) * 4 (RAW: $9 → $10)
    add $10, $2, $10      # addr = base + offset (RAW: $10, $2 → $10)
    lw $11, 0($10)       # $11 = arr[k] (RAW: $10 → $11)
    slt $1, $7, $11       # if arr[j] < arr[k] (RAW: $7, $11 → $1)
    beq $1, $0, L4        # 제어 hazard: $1 → ID에서 분기 판단
    sw $11, 0($6)         # arr[j] = arr[k] (RAW: $11, $6)
    add $6, $10, $0       # $6 ← addr of k (RAW: $10)
    add $8, $9, $0        # k ← k - 1 (RAW: $9 → $8)
    j L3                 # jump to loop

L4:
    sw $7, 0($6)         # arr[k] ← original arr[j] (RAW: $7, $6)
    addi $5, $5, 1       # j++
    j L2                 # next outer loop

done:
    break
```

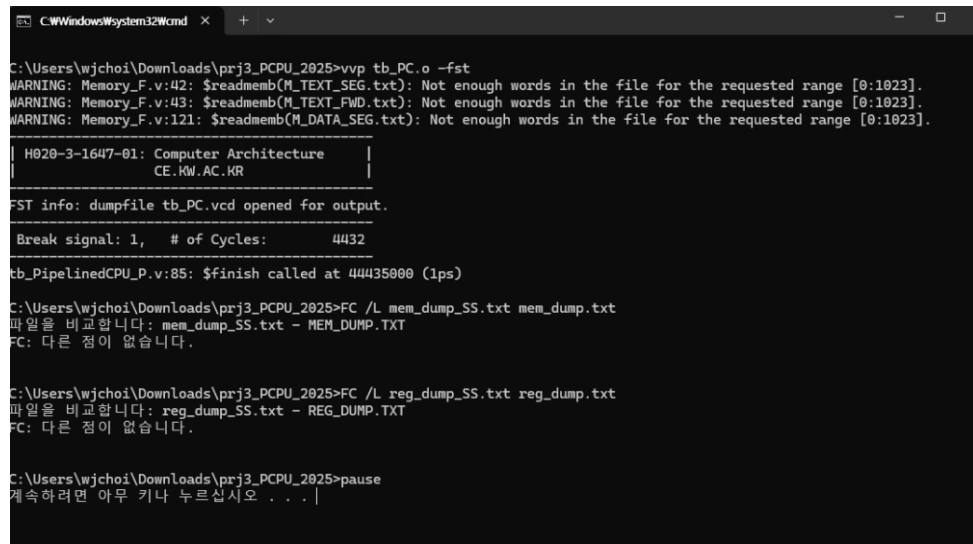
파이프라인 구조에서 주의해야 하는 것은 명령어 간 데이터 의존성이다. 예를 들어 lw \$11, 0(\$10) 명령 직후 slt \$1, \$7, \$11 명령이 사용될 경우, \$11의 값이 아직 WB(Write Back) 단계에 도달하지 않았기 때문에 올바른 값을 읽지 못할 수 있다. 이를 Read-after-Write (RAW) hazard라고 하며, 이를 해결하지 않으면 잘못된 결과가 발생하거나 프로그램이 중단된다. 또한 slt \$1, \$7, \$11 후 beq \$1, \$0, L4 명령이 바로 이어질 경우에도 문제가 발생한다. 이는 slt의 연산 결과가 아직 WB 단계에 도달하지 않았는데, beq가 ID 단계에서 비교를 시도하기 때문이다. 이러한 데이터 hazard를 방지하기 위해 forwarding이 없을 경우 적절한 수의 nop을 삽

입해야 하며, forwarding이 있을 경우에도 분기 명령은 ID 단계에서 판단되기 때문에 일부 stall은 여전히 필요하다.

2. 2개의 시뮬레이션 결과 분석 + 명령 수행에 걸린 총 사이클 수

1-1 forwarding 있을 때

Forwarding이 활성화된 환경에서는 데이터 hazard 중 대부분이 ALU의 결과가 WB 단계에 도달하지 않아도 다음 명령어가 EX 단계에서 직접 가져올 수 있기 때문에, nop 삽입이 거의 필요 없다. 이번 실험에서 forwarding을 명시적으로 M_TEXT_FWD.txt에 반영하고 시뮬레이션을 수행한 결과, 전체 명령어의 흐름이 정상적으로 실행되었으며, 메모리 정렬 결과 및 레지스터 덤프도 기준 출력과 완전히 일치하였다. 총 수행 사이클 수는 약 4300 사이클 대로, nop 삽입을 최소화한 덕분에 성능이 대폭 향상되었음을 확인할 수 있었다.



```
C:\Users\wjchoi\Downloads\prj3_PCPU_2025>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: 4432

tb_PipelinedCPU_P.v:85: $finish called at 44435000 (1ps)

C:\Users\wjchoi\Downloads\prj3_PCPU_2025>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.

C:\Users\wjchoi\Downloads\prj3_PCPU_2025>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.

C:\Users\wjchoi\Downloads\prj3_PCPU_2025>pause
계속하려면 아무 키나 누르십시오 . . .
```

1-2 forwarding 없을 때

반면, 하드웨어 forwarding을 비활성화한 상태에서는 명령어 간 데이터 의존성을 완화하기 위해 다수의 nop 명령어를 삽입해야 했다. lw 직후에 데이터를 사용하는 명령어나 slt 후 beq와 같은 제어 흐름 분기 이전에는 최소 2~3개의 nop을 사용하였다. 이로 인해 전체적인 코드 길이와 명령어 수가 증가하였고, 시뮬레이션 수행 시간 역시 증가하였다. 실험 결과 forwarding 없이 정렬이 정상적으로 수행되었으며, reg/mem 덤프도 기준과 일치하였다. 다만 총 수행 사이클 수는 약 4260 사이클로, forwarding 버전보다 600 사이클 이상 더 소요되어 성능 차이가 명확히 드러났다.

```
C:\Windows\System32\cmd x + v
C:\Users\wjchoi\Downloads\prj3_PCPU_2025>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: 4236
-----
tb_PipelinedCPU_P.v:85: $finish called at 42475000 (1ps)

C:\Users\wjchoi\Downloads\prj3_PCPU_2025>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.

C:\Users\wjchoi\Downloads\prj3_PCPU_2025>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.

C:\Users\wjchoi\Downloads\prj3_PCPU_2025>pause
계속하려면 아무 키나 누르십시오 . . . |
```

4. 문제점 및 고찰

처음에는 명령어 간 데이터 의존성을 정확히 파악하지 못해 불필요한 nop을 많이 삽입하였고, 이로 인해 성능이 저하되었다. 특히 분기 명령어의 제어 hazard와 lw 이후의 data hazard 처리에 어려움을 겪었다. 이후 pipeline 단계별 동작을 분석하고 forwarding 개념을 적용하면서 nop을 최소화할 수 있었고, 사이클 수 또한 줄일 수 있었다. 이 과정을 통해 pipeline 구조에서의 hazard 처리 중요성과 하드웨어 자원 제어 방식의 효과를 명확히 이해하게 되었다.