4장. 프로세서 구조

4.4 파이프라이닝의 일반 원리

실생활에서 파이프라인 – 자동차 세차 예

Sequential

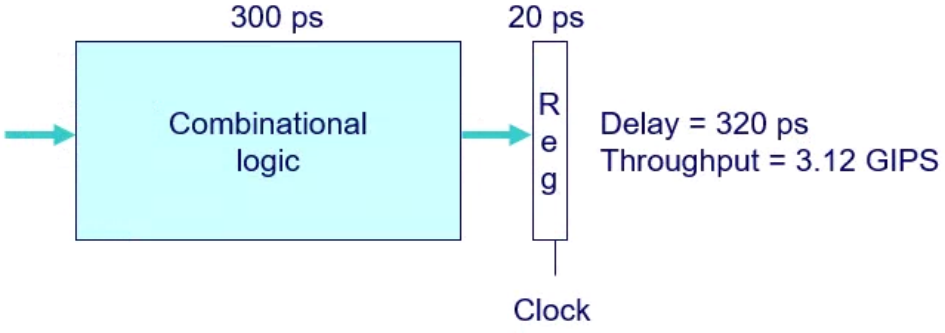
Parallel

Pipelined

- 핵심 개념
  - 처리과정을 독립적인 단계(stage)로 분할
  - 객체(자동차)가 순서대로 단계들을 이동
  - 특정 순간에 다 수의 객체들이 처리 (세차)

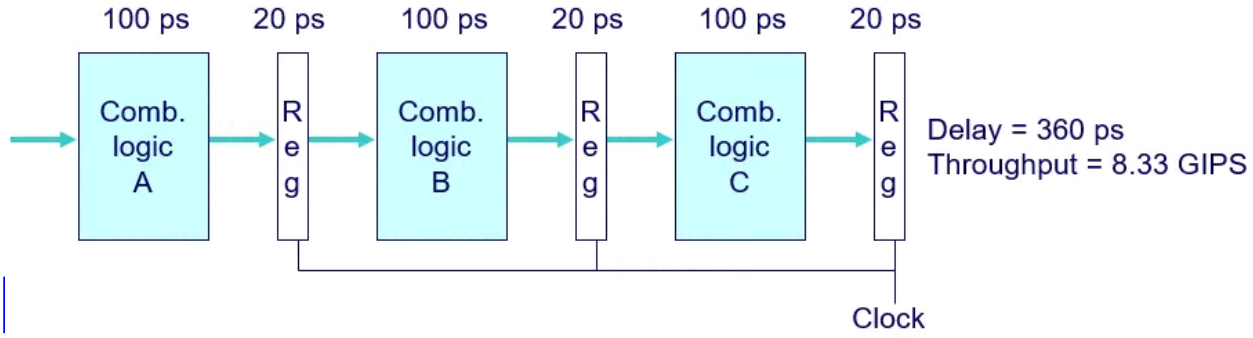
계산 하드웨어 예 – 비 파이프라인

- 지연시간 (delay): 320 ps
  - 조합논리 회로 (계산, computation): 300 ps
  - 레지스터 (적재, loading): 20ps
- 처리량 (throughput): 단위 시간 당 작업량
  - 1/320 ps = 1/(320x10^{-12}) = 3.12x10^{9} = 3.12 GIPS
    (Giga Instructions Per Second)



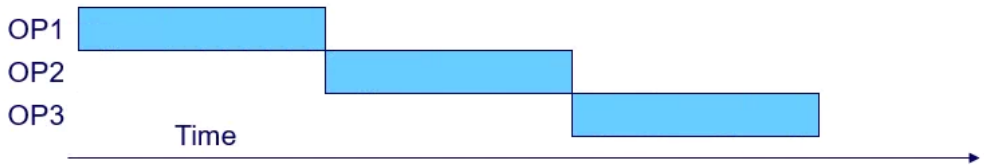
계산 하드웨어 예 – 3단계 파이프라인

- 조합논리 회로를 100 ps의 3개의 블록으로 분할
  - 이 전 연산이 단계 A를 마치자마자 새로운 연산 시작
    - 새 연산은 매 120 ps 마다 시작
- 한 연산의 전체 지연시간은 증가
  - 시작에서 종료까지 360 ps
- 처리량은 증가
  - 1/120 ps = 8.33 GIPS



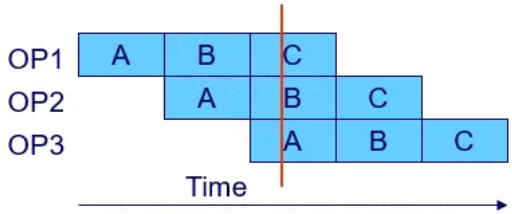
파이프라인 다이어그램

Unpipelined



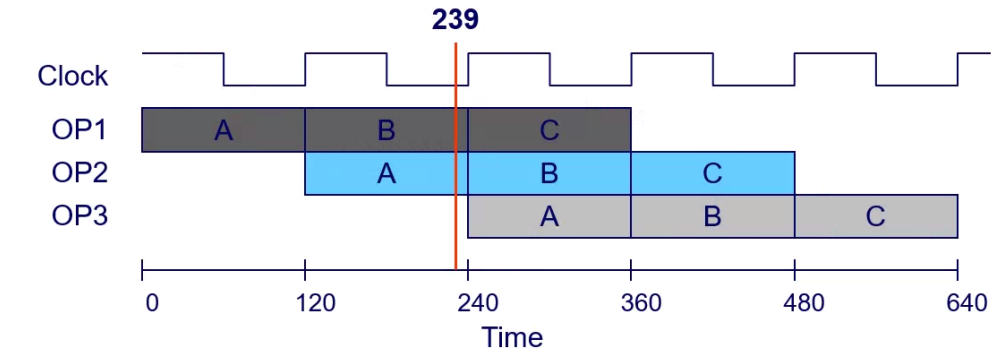
- **Cannot start new operation until previous one completes**

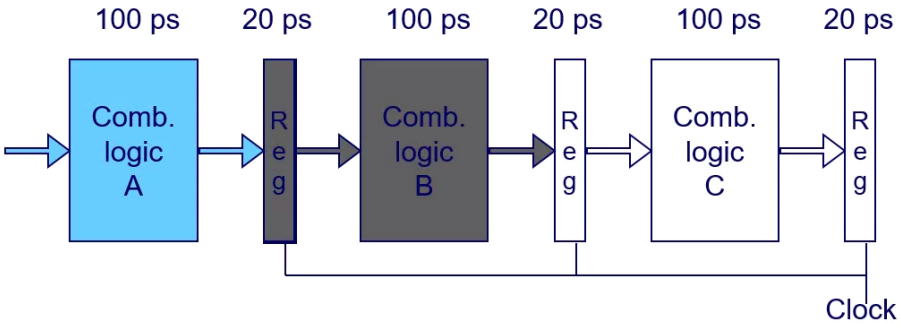
3-Way Pipelined



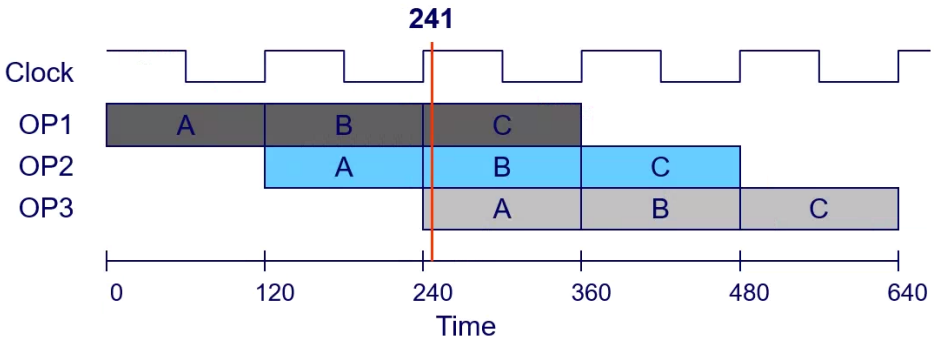
- **Up to 3 operations in process simultaneously**

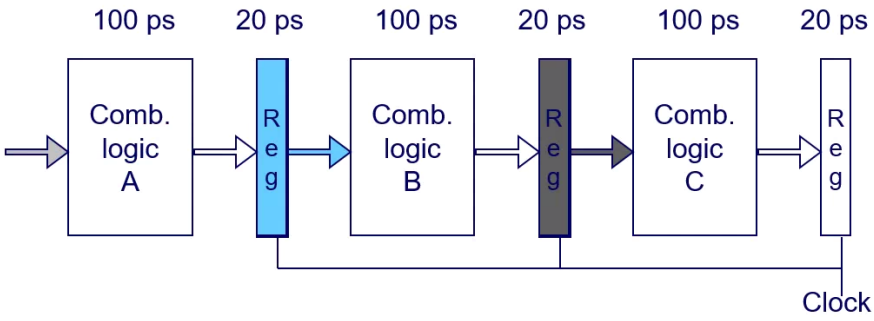
파이프라인 동작 – 시간 239



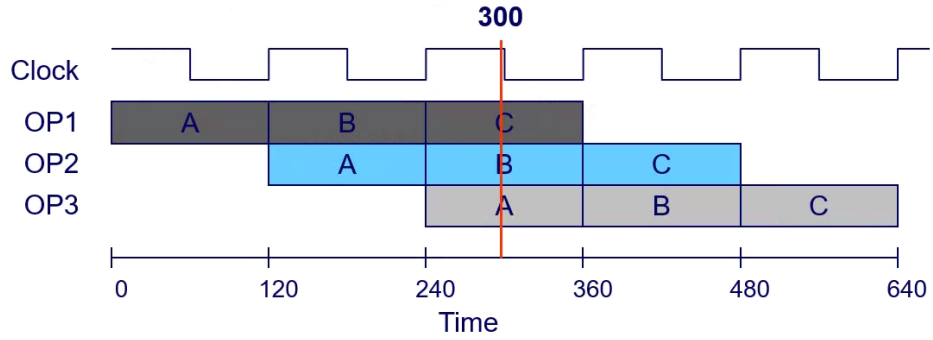


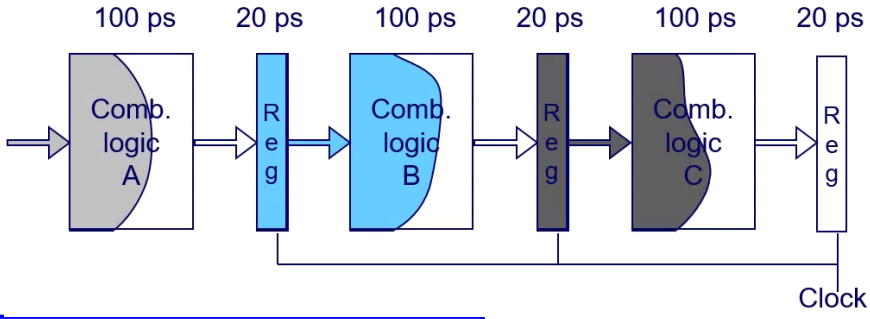
파이프라인 동작 – 시간 241



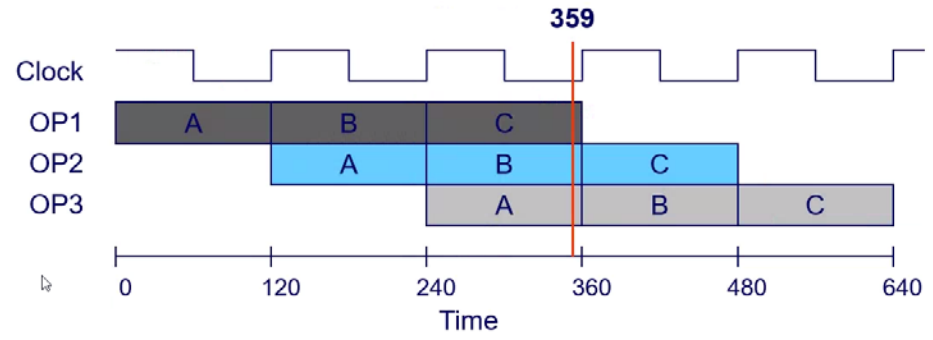


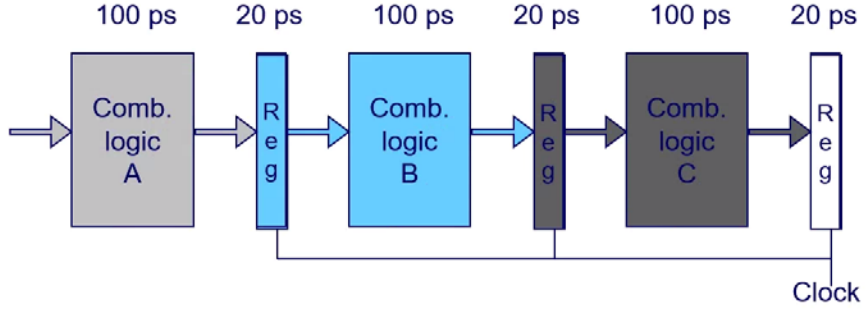
파이프라인 동작 – 시간 300





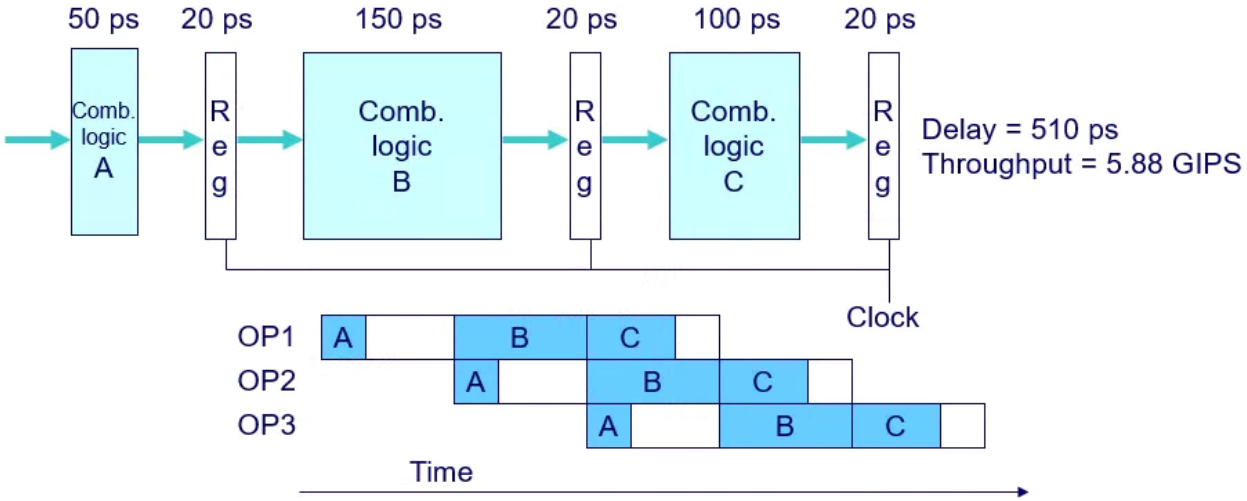
파이프라인 동작 – 시간 359





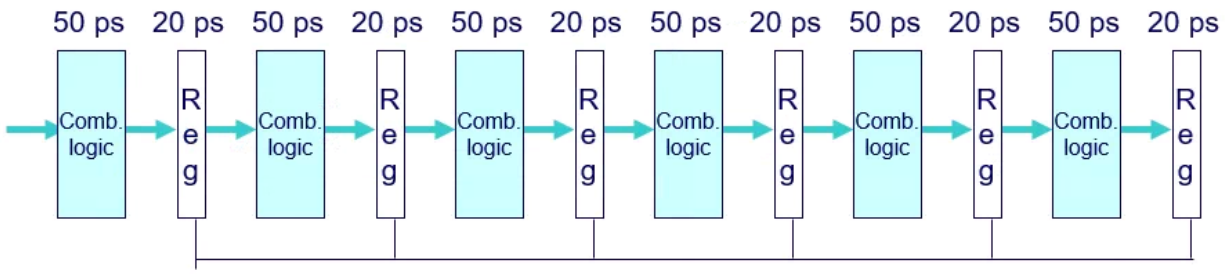
파이프라인 한계 – 비균일 지연 (Nonuniform Delay)

- 처리량을 가장 늦은 단계로 제한
  - 다른 빠른 단계들은 대기
- 시스템을 균형된 단계들로 분할하는 것이 중요



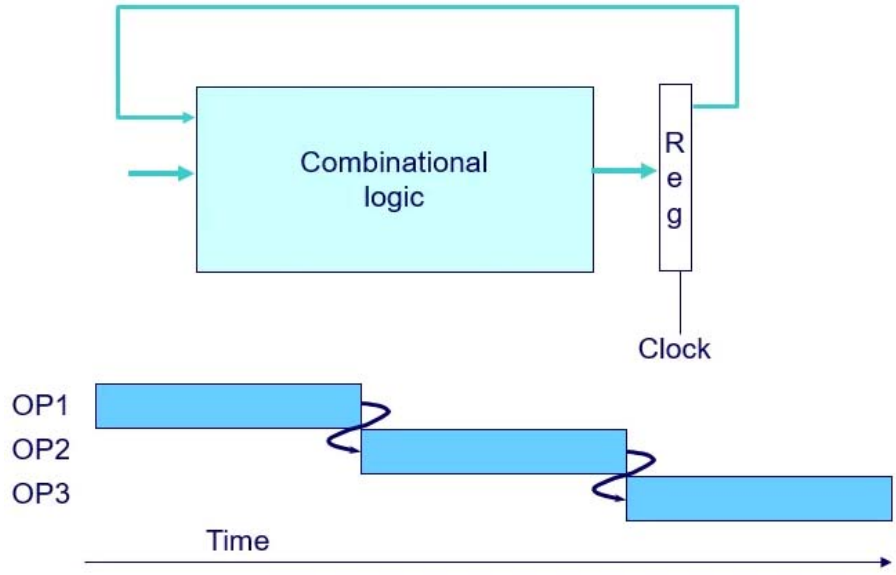
파이프라인 한계 – 레지스터 오버헤드 (Register Overhead)

- 파이프라인이 깊어지면(단계가 증가하면) 레지스터에 적재하는 오버헤드가 커짐
  - 전체 클럭 사이클에서 레지스터 적재 비율
    - 1-단계 파이프라인: 6.25%
    - 3-단계 파이프라인: 16.67%
    - 6-단계 파이프라인: 28.57%
  - 최신 프로세서들은 클럭 속도를 최대로 올리기 위해 매우 깊은 파이프라인을 사용 (15단계 이상)



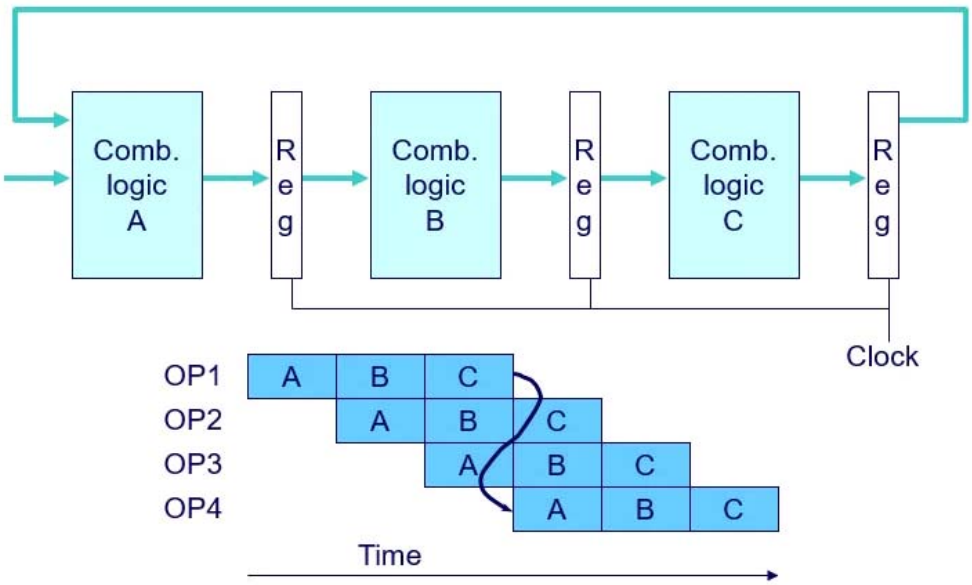
데이터 의존성 (Data Dependencies)

- 명령어의 연산은 이전 명령어의 결과에 의존

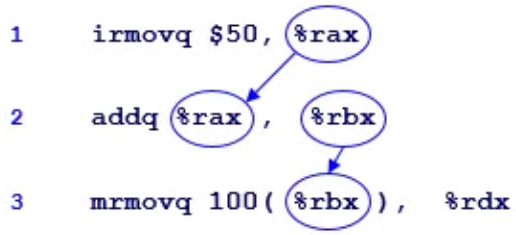


데이터 해저드 (Data Hazards)

- 명령어의 결과가 다음 명령어에 제 때 공급되지 못함
- 파이프라인이 시스템의 동작을 변경



프로세서의 데이터 의존성

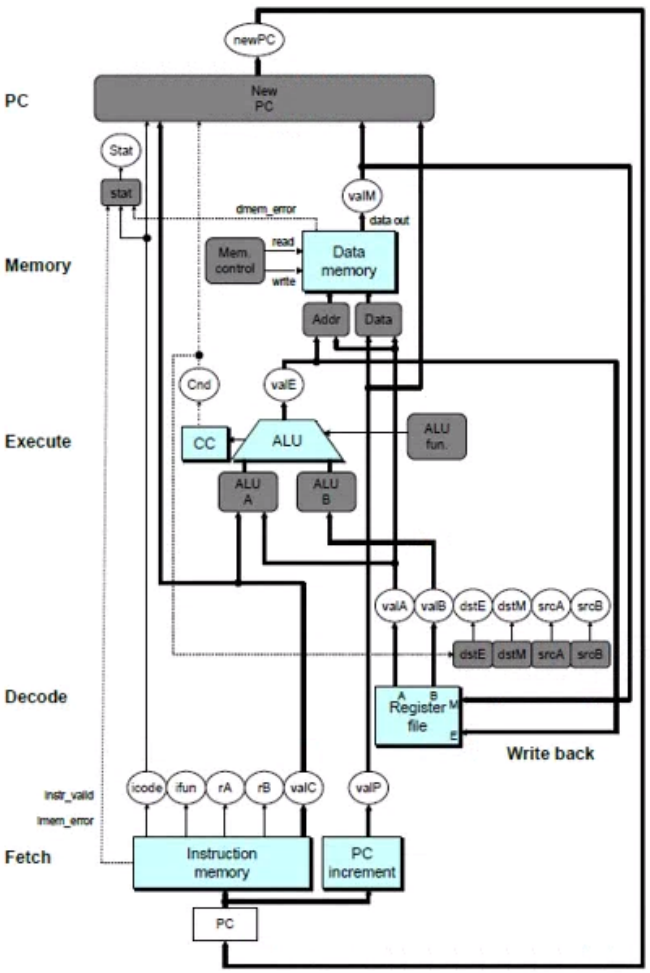


- 한 명령어의 결과가 다른 명령어의 오퍼랜드로 사용
  - RAW (Read-After-Write) 의존성
- 프로그램에서는 빈번하게 나옴
- 파이프라인이 RAW 의존성을 대처해야 함
  - 프로그램이 올바르게 동작
  - 성능에의 영향을 최소화

4.5 파이프라인형 Y86-64의 구현

SEQ 하드웨어

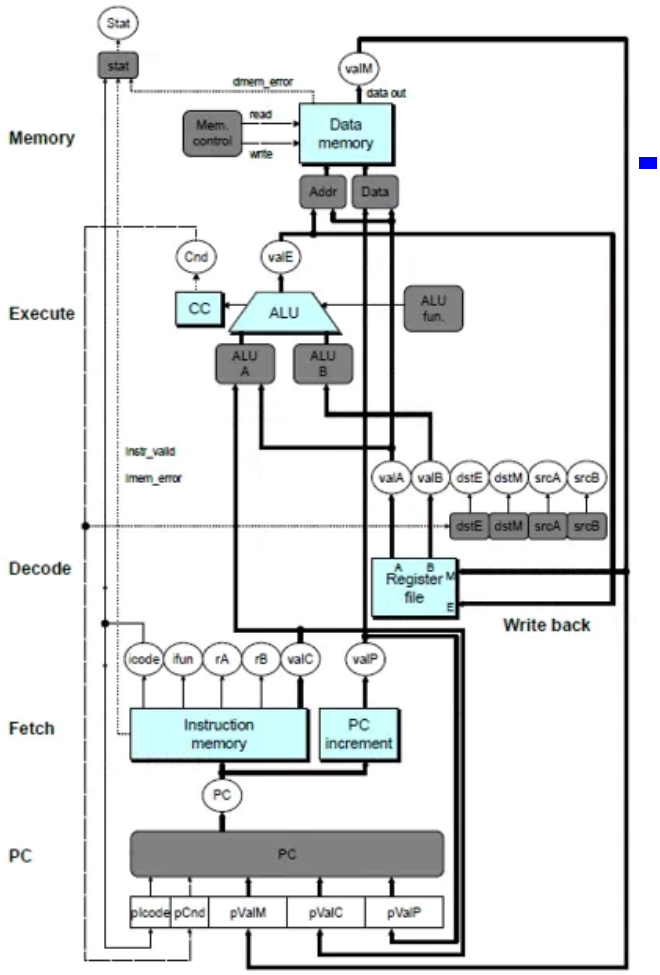
- 각 단계들이 순서대로 발생
- 한 클럭에 하나의 명령어 실행



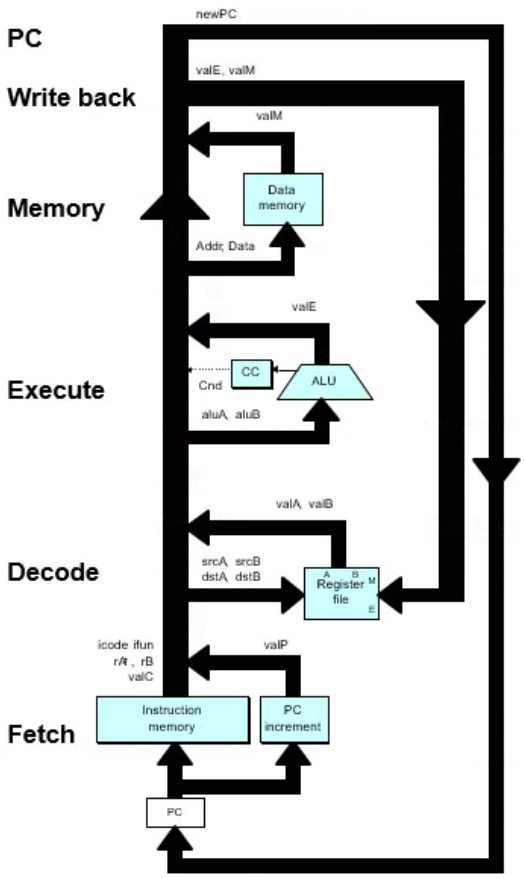
SEQ+ 하드웨어

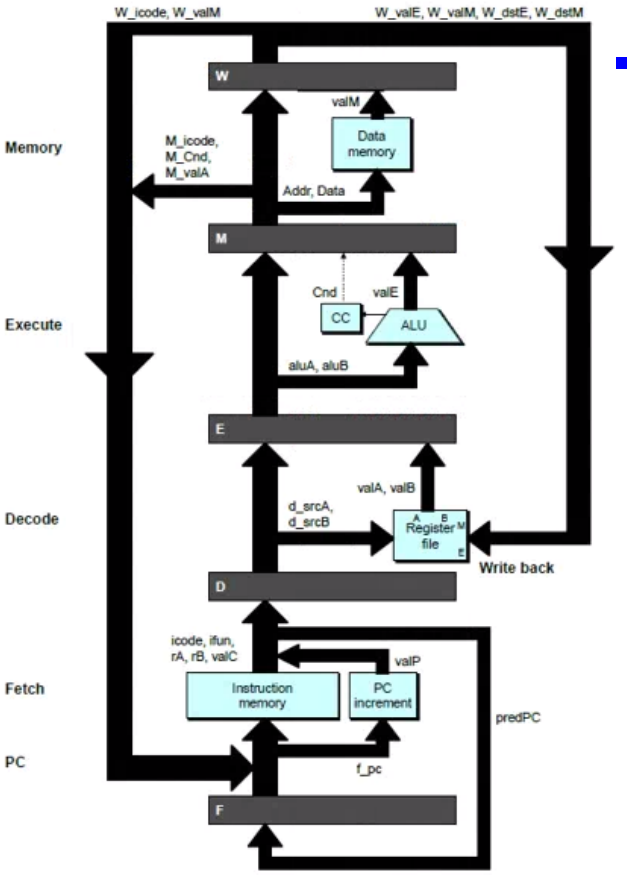
- 순차적 구현
- PC 단계를 처음으로 이동

- PC 단계
  - 현재 명령어의 PC 선택
  - 이 전 명령어의 계산 결과에 따라 선택
- 프로세서 상태
  - PC가 레지스터에 저장되지 않음
  - 그러나, 다른 저장 값에 기반하여 PC 결정



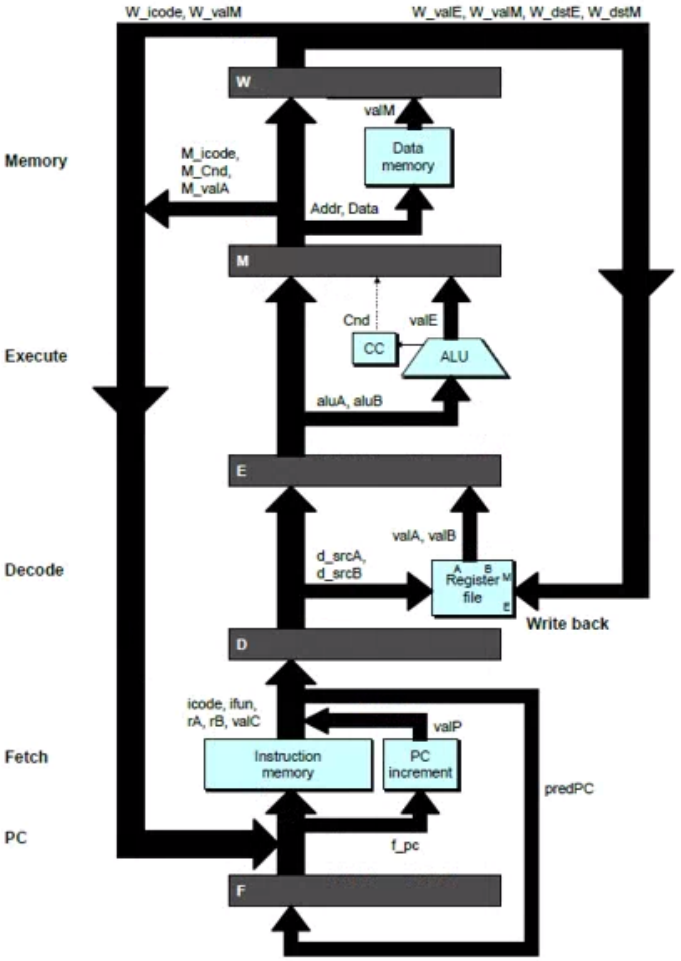
파이프라인 레지스터 삽입





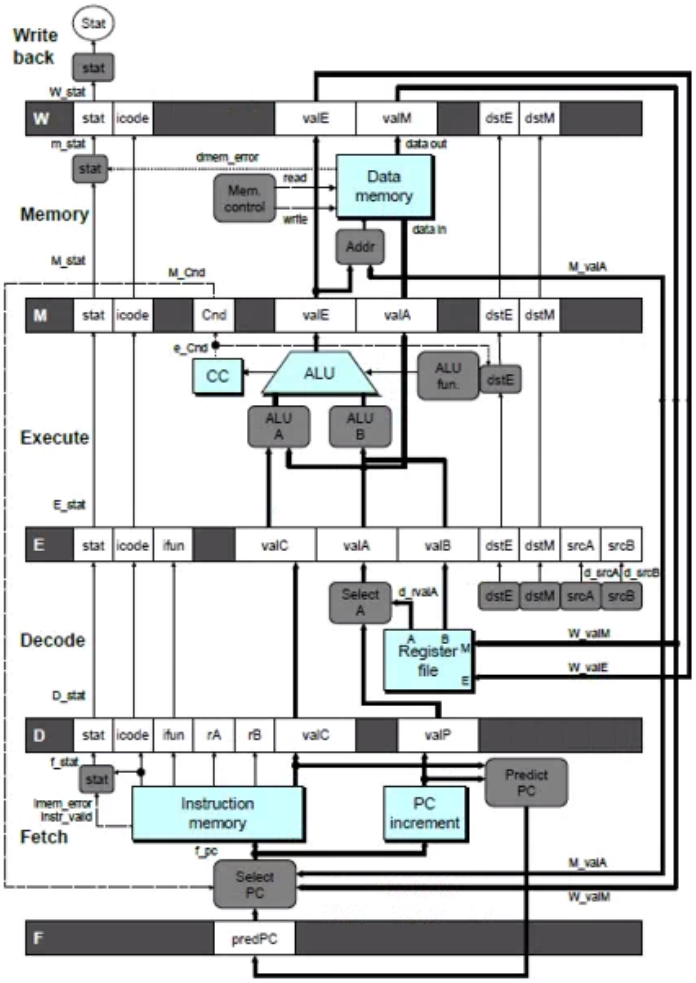
파이프라인 단계

- 페치 (Fetch)
  - 현재 PC 선택
  - 명령어 읽기
  - 증가된 PC 계산
- 해독 (Decode)
  - 프로그램 레지스터 읽기
- 실행 (Execute)
  - ALU 연산
- 메모리 (Memory)
  - 데이터 메모리 읽기 또는 쓰기
- 재기록 (Write Back)
  - 레지스터 파일 갱신



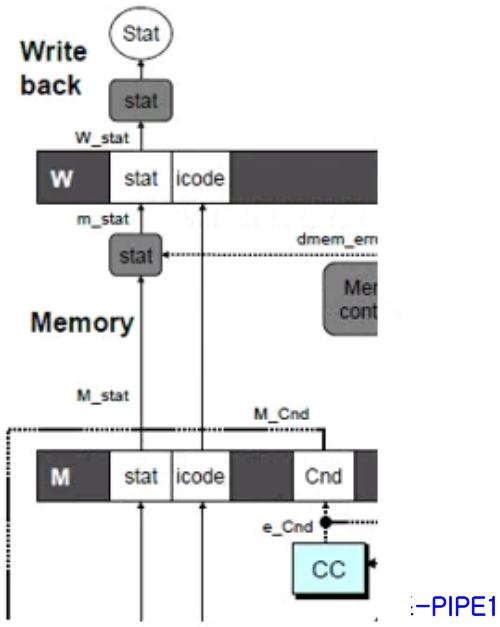
PIPE 하드웨어

- 파이프라인 레지스터들은 수행되는 명령어의 중간 값들을 저장
- 순방향(윗방향) 경로
  - 한 단계에서 다음 단계로 값들이 전달
  - 과거의 단계(역방향)로 전달은 안됨
    - 예) valC는 해독 단계를 거쳐 전달



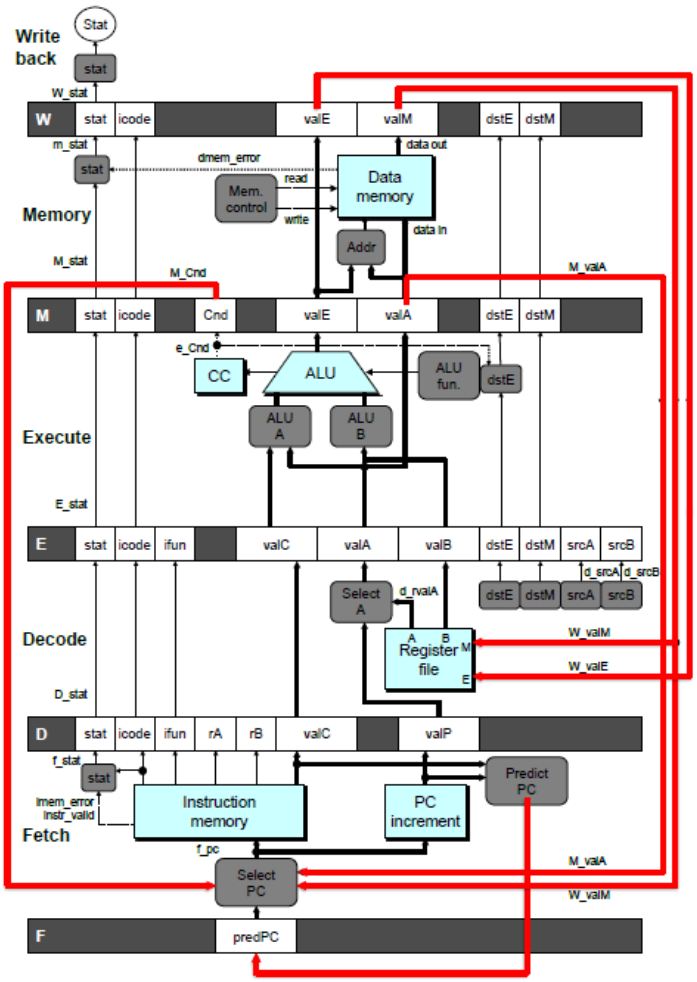
신호들의 명명 규칙

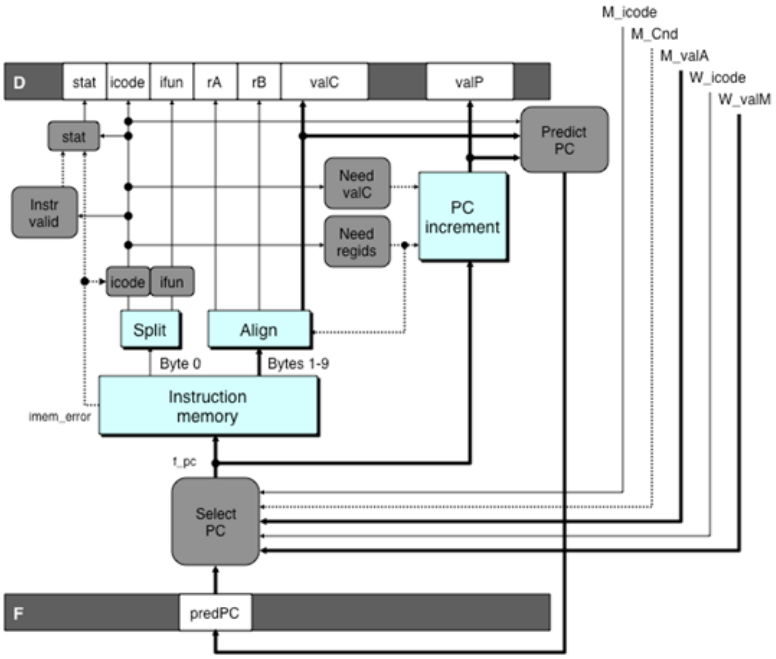
- S_Field
  - 단계 S의 파이프라인 레지스터에 저장된 필드의 값
- s_Field
  - 단계 S에서 계산된 필드의 값



피드백 경로

- 예측된 PC (Predicted PC)
  - 다음 PC 값을 추정
- 분기 정보
  - taken / not-taken
  - 다음 명령어 주소 또는 타겟 주소
- 리턴 지점
  - 메모리에서 읽음
- 레지스터 갱신
  - 레지스터 파일 쓰기 포트에서 갱신



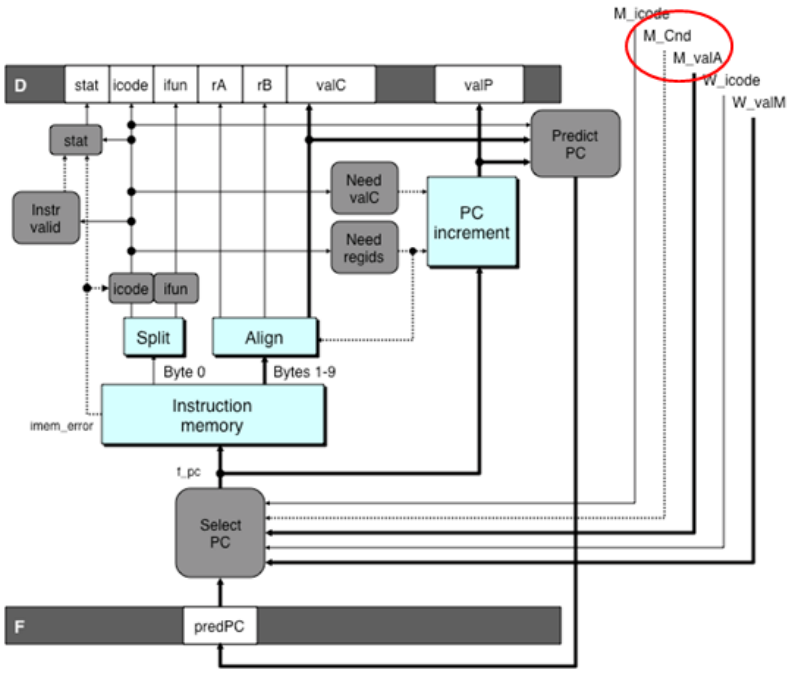


PC 예측

- 현재 실행 중인 명령의 페치 단계 끝나자마자 새로운 명령어 페치 시작
  - 안정적으로 다음 명령어를 결정할 시간이 없음
- 다음에 실행될 명령어를 추정
  - 예측이 잘못된 경우에는 복구 (revover)

교재의 예측 전략

- 제어 흐름의 이동이 없는 명령어
  - valP를 다음에 수행될 명령어 PC로 예측
  - 항상 올바름
- 호출(call) 및 무조건 분기 명령어
  - valC (목적지)를 다음 PC로 예측
  - 항상 올바름
- 조건 분기 명령어
  - valC (목적지)를 다음 PC로 예측
  - 분기가 taken인 경우만 올바름
    - 일반적으로 60%의 정확도
- 리턴 명령어
  - 예측을 하지 않음

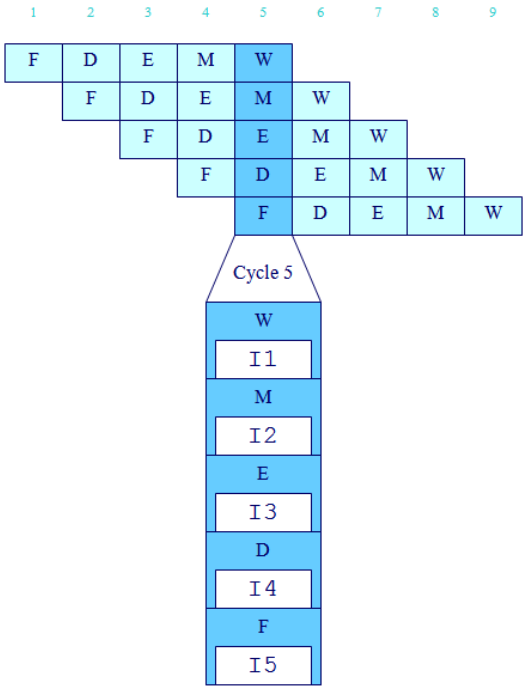


잘못된 예측으로 부터 복구

- 잘못 예측된 분기
  - 조건 분기 명령어가 메모리 단계 도달했을 때 분기 조건 플래그 참조
  - valA (M_valA 값)로 부터 다음 (fall-through) 명령어 PC 획득
- 리턴 명령어
  - ret 명령어가 재기록 단계에 도달했을 때 복귀 주소 (W_valM) 획득

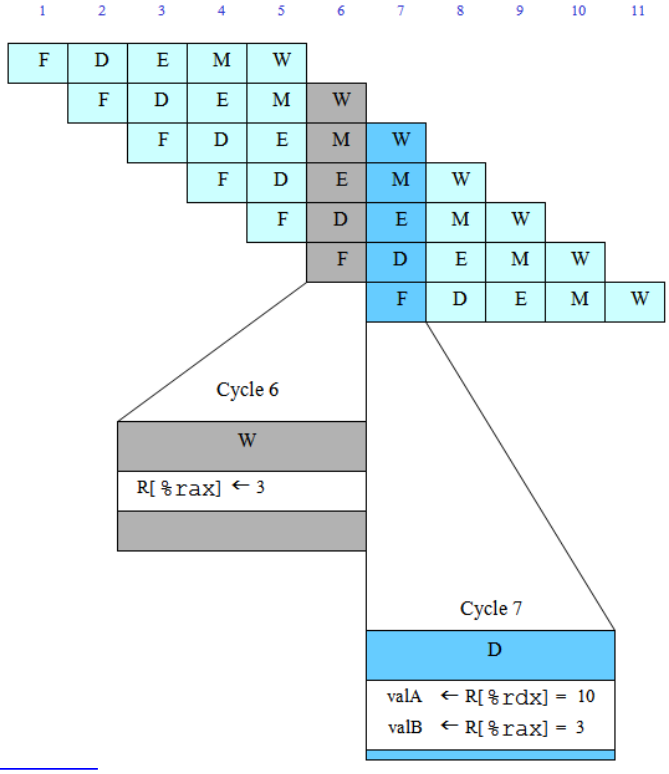
파이프라인 예

```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



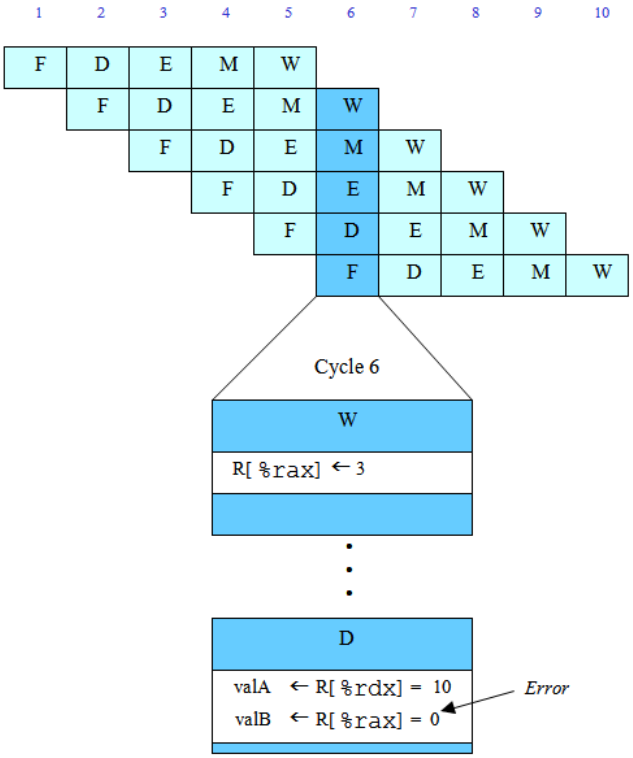
데이터 의존성 (Data Dependencies) – 3개의 Nop

```
# demo-h3.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```



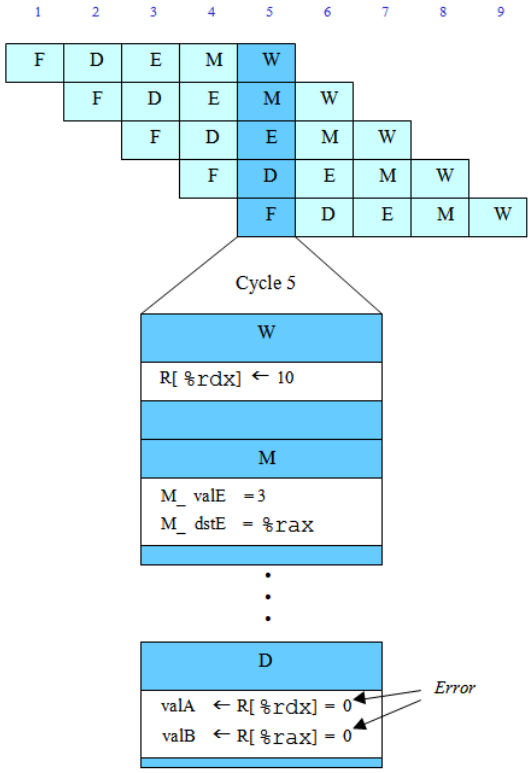
데이터 의존성 – 2개의 Nop

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



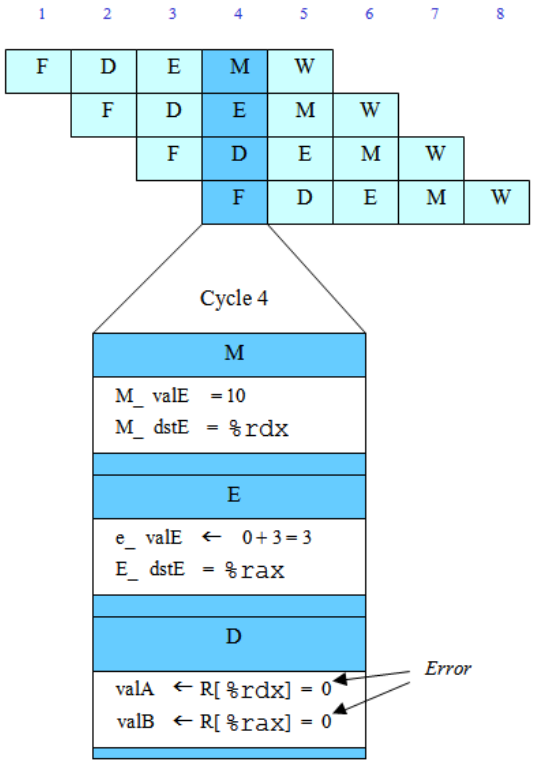
데이터 의존성 – 1개의 Nop

```
# demo-h1.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



데이터 의존성 – Nop 없음

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



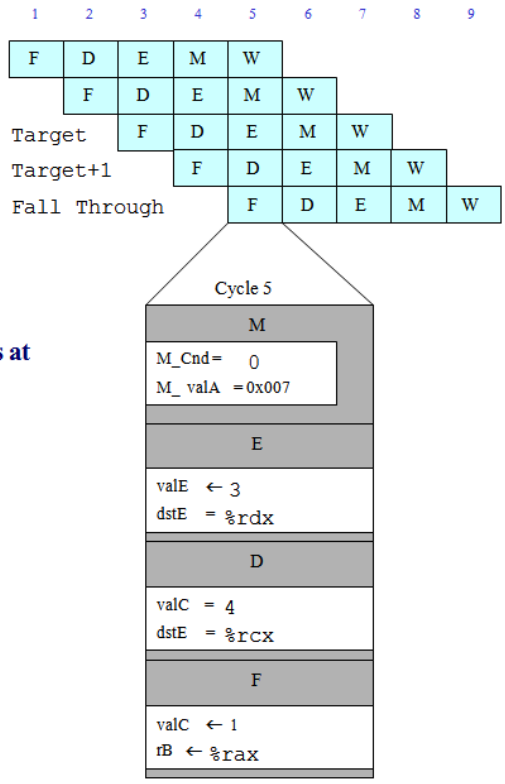
잘못된 분기 예측 (Branch Misprediction) 예

```
0x000:     xorq %rax,%rax
0x002:     jne  t               # Not taken
0x00b:     irmovq $1, %rax      # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx      # Target (Should not execute)
0x023:     irmovq $4, %rcx      # Should not execute
0x02d:     irmovq $5, %rdx      # Should not execute
```

- **Should only execute first 8 instructions**

잘못된 분기 예측 추적

```
# demo-j
0x000:    xorq %rax,%rax
0x002:    jne t # Not taken
0x019: t: irmovq $3, %rdx # Target
0x023:    irmovq $4, %rcx # Target+1
0x00b:    irmovq $1, %rax # Fall Through
```



- **Incorrectly execute two instructions at branch target**

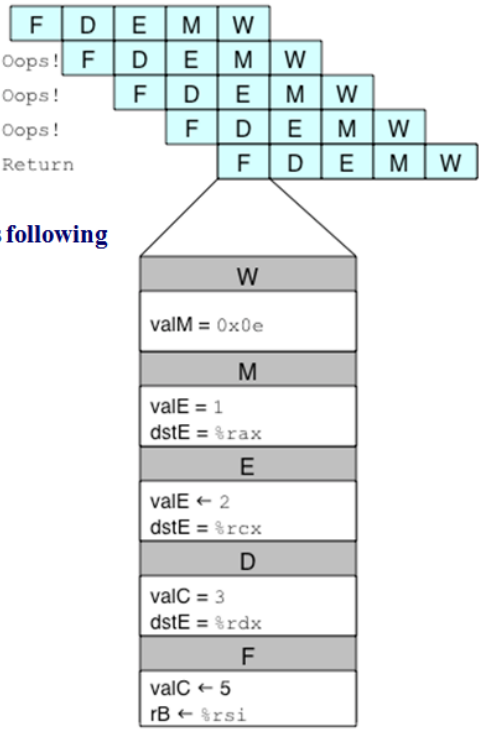
리턴 예

```
  0x000:    irmovq Stack,%rsp  # Intialize stack pointer
0x00a:    nop                # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p             # Procedure call
0x016:    irmovq $5,%rsi     # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p: nop                 # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax      # Should not be executed
0x02e:    irmovq $2,%rcx      # Should not be executed
0x038:    irmovq $3,%rdx      # Should not be executed
0x042:    irmovq $4,%rbx      # Should not be executed
0x100: .pos 0x100
0x100: Stack:                 # Initial stack pointer
```

- **Require lots of nops to avoid data hazards**

잘못된 리턴 예

```
# demo-ret
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



- **Incorrectly execute 3 instructions following `ret`**

요 약

요 약

- 파이프라인 개념
  - 명령어 실행을 5 단계로 분할
  - 파이프라인 모드로 명령어들이 연속적으로 실행
- 제약점
  - 다음 명령어와의 간격이 좁을 때 명령어들 간에 의존성을 처리할 수 없음
  - 데이터 종속성 (data dependencies)
    - 한 명령어가 레지스터 쓰기를 하고, 이 후 명령어가 이의 읽기를 할 때 발생
  - 제어 종속성 (control dependencies)
    - 파이프라인이 올바르게 예측하지 못한 명령어가 PC를 세트
    - 잘못된 분기 예측과 리턴
- 파이프라인의 개선이 필요