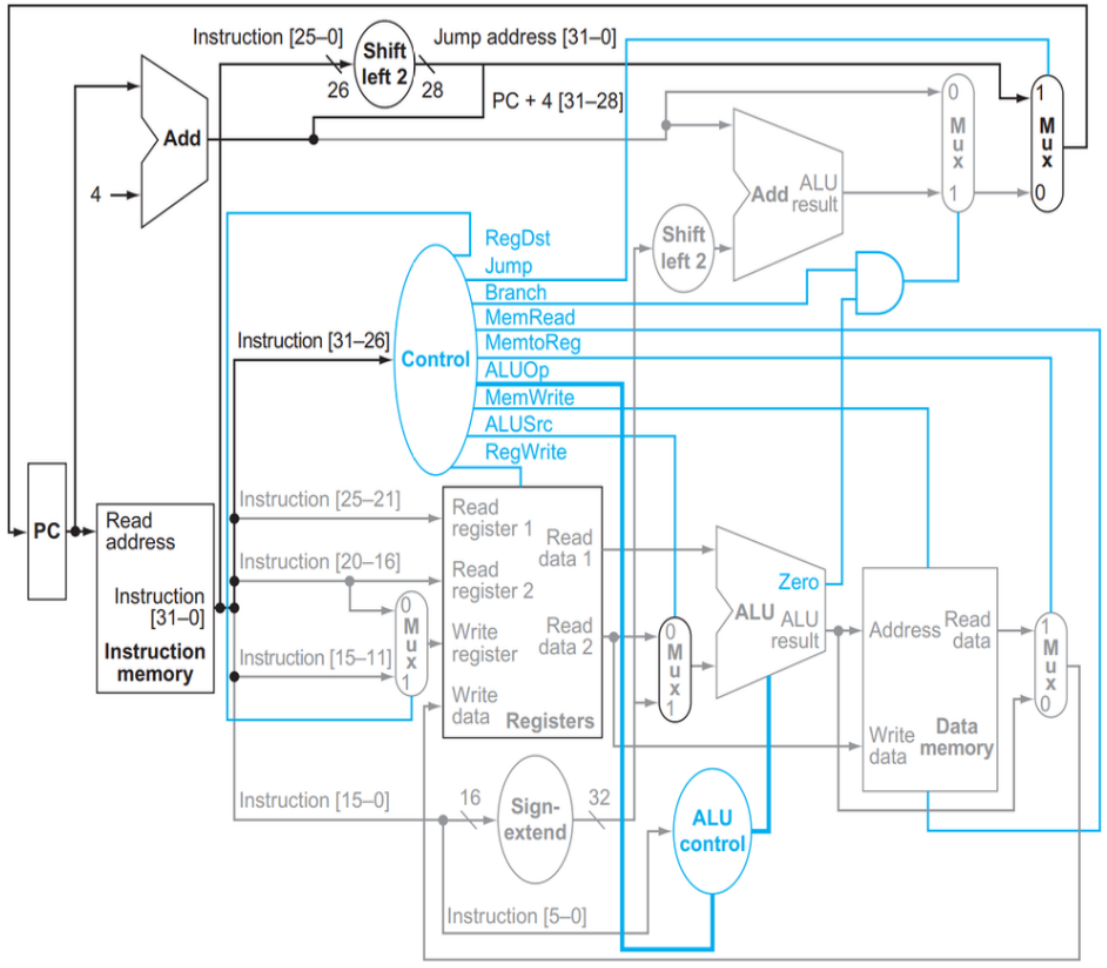
Computer Architecture (ENE1004)

▼ Lec 12

Lec 12: The Processor 5

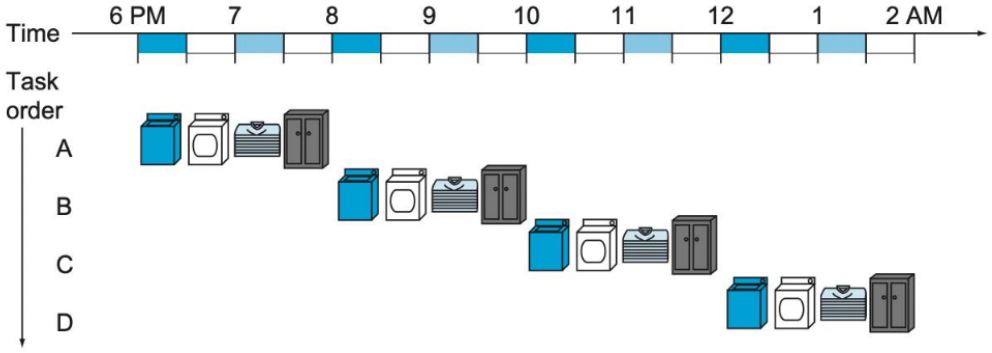
A Simple Implementation(구현) of DataPath



- This implementation can process any given instruction (이 구현은 주어진 명령어를 처리할 수 있다)
  - R-type, load/store, branch, jump
- However, this is not used in today's processors; why?

    - A new instruction cannot be executed till its previous instruction is completed; only one instruction at time! (이전 명령어가 완료될 때까지 새 명령어를 실행할 수 없으며, 한 번에 하나의 명령어만 실행할 수 있기 때문이다!)
    - Data flow of an instruction involves various units, which needs a long time (명령어의 데이터 흐름에는 다양한 단위가 포함되므로 시간이 오래 걸린다)
        - lw takes the longest time, as it needs (i) fetching instruction, (ii) reading registers, (iii) performing an ALU operation, (iv) reading memory, and
          (v) writing to a register
    - So, this implementation is not a good choice from a performance standpoint (따라서 이 구현은 성능 관점에서 볼 때 좋은 선택이 아니다)
- Let us see an advanced version(고급 버전)

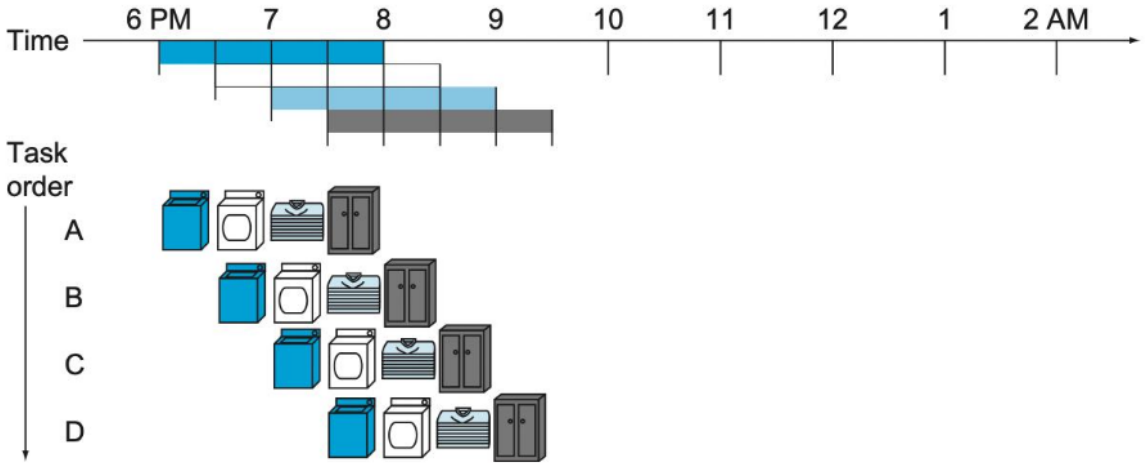
An Analogy of Laundary (세탁소의 비유)



- For a single load of clothes, (세탁물을 한 번 넣을 경우)
    - (1) Place the load in the washer (세탁기에 세탁물을 넣는다)
    - (2) When the washer is finished, place the wet load in the dryer (세탁기가 끝나면 젖은 세탁물을 건조기에 넣는다)
    - (3) When the dryer is finished, place the dry load on a table and fold (건조기가 끝나면 마른 세탁물을 테이블 위에 놓고 접는다)
    - (4) When folding is finished, put the clothes in the wardrobe (접기가 끝나면 옷을 옷장에 넣는다)
- Assume that

    - Each of the four steps takes 30 minutes - 30 minutes for a job (네 단계 각각에 30분이 소요된다 - 한 작업에 30분)
    - There are multiple loads of clothes (Task A, B, C, D, ...) (옷이 여러 개 있는 경우 (작업 A, B, C, D, ...))
- One way of doing the laundry is to process tasks one after another (빨래를 하는 한 가지 방법은 작업을 차례로 처리하는 것이다)
    - For Task A, do everything (washer-dryer-folding-wardrobe) (작업 A의 경우 모든 작업(세탁기-건조기-접기-옷장)을 수행한다)
    - When Task A is done, start over the entire process for Task B (작업 A가 완료되면 작업 B의 전체 프로세스를 다시 시작한다)
    - When Task B is done, start over the entire process for Task C (작업 B가 완료되면 작업 C의 전체 프로세스를 다시 시작한다)
    - ...
- This way takes a total of 8 hours for the four tasks
    - It starts at 6PM and ends at 2AM
- Is there any better way to reduce the total time?

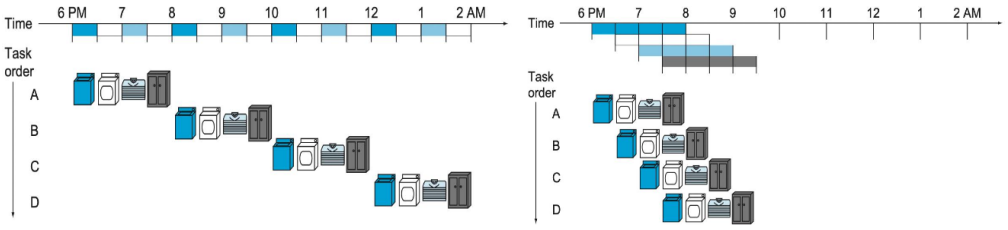
Pipelined Approach (파이프라인 접근 방식)



- Why don’t we start the next tasks as soon as possible?
    - When the washer is finished with Task A (at 6:30), place Task A in the dryer & Task B in the washer

    - When drying A is done (at 7:00), start folding it, move B to the dryer, & put C into the washer
    - When A is put to wardrobe (at 7:30), start folding B, drying C, & putting D into the washer
- In this way, the total time for A, B, C, & D is significantly reduced from 8 to 3.5 hours (이렇게 하면 A, B, C, D의 총 시간이 8시간에서 3.5시간으로 크게 단축된다)
- Here, all steps (washer, dryer, folding, wardrobe) are operating concurrently (여기서 모든 단계(세탁기, 건조기, 접기, 옷장)가 동시에 작동한다)
    - We call this way of processing tasks “pipelining” and each step of the pipelining “stage” (이러한 작업 처리 방식을 "pipelining"이라고 하고 pipelining 각 단계를 "stage"라고 한다)

Pipelined Approach : Response Time vs Throughput (파이프라인 접근 방식 : 응답 시간 vs 처리량)

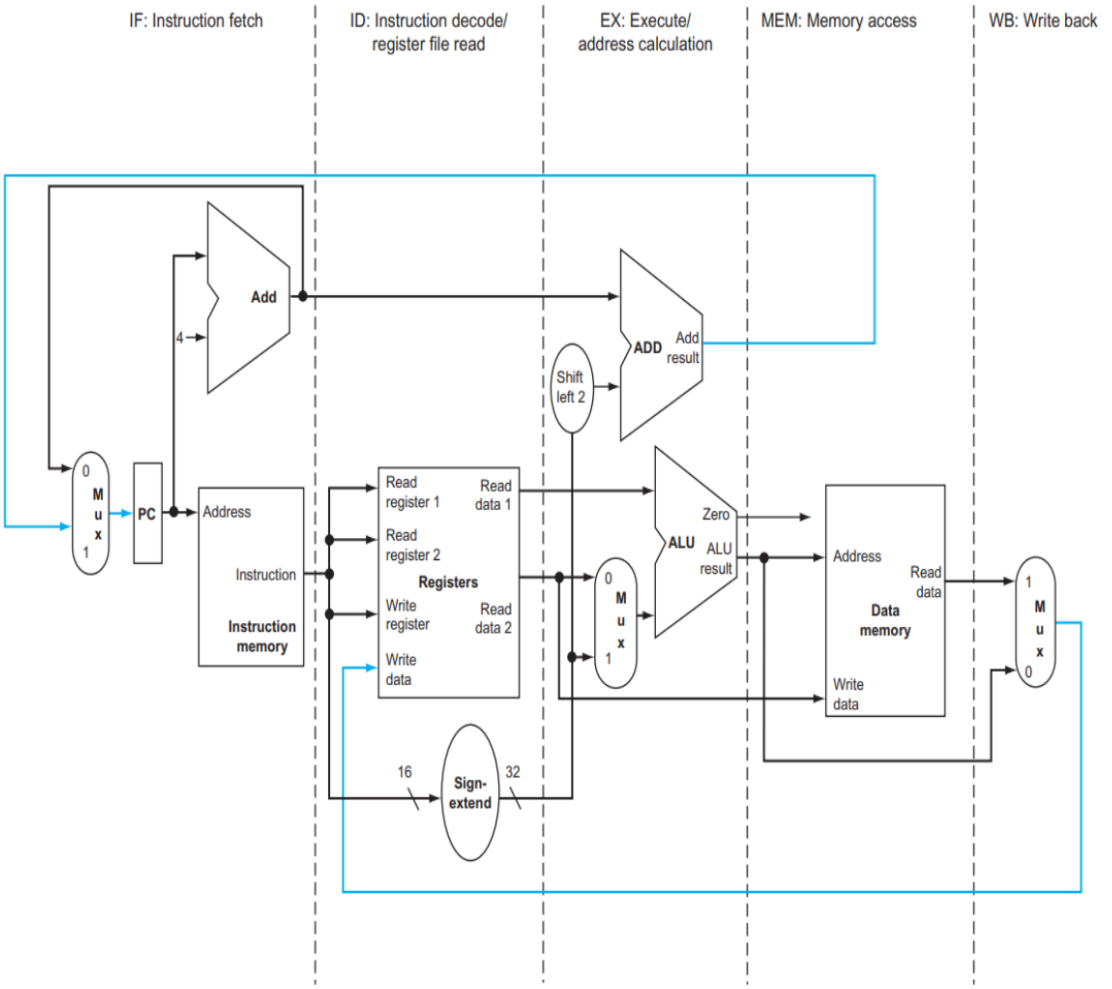


- Pipelined approach is better than non-pipelined approach in terms of performance (Pipelined 방식이 non-pipelined 방식보다 성능 측면에서 더 우수하다)
    - Is the time to do laundry for a single load (a single task : A) reduced? (세탁물 한 개 (단일 작업 : A)를 세탁하는 데 걸리는 시간이 단축되는가?)
    - No, it still takes 2 hours to go through the four stages — from washer, dryer, folding, to wardrobe (아니다. 세탁기, 건조기, 접기, 옷장까지 4단계를 거치는 데 여전히 2시간이 걸린다)
    - However, the time to do laundry for the four loads (four tasks) is reduced (그러나 4가지 세탁물 (4가지 작업)에 대한 세탁 시간이 줄어든다)
- Recall the two performance metric, response time vs throughput, discussed in Lec-1 (Lec-1에서 설명한 응답 시간 vs 처리량이라는 두 가지 성능

지표를 상기해라)

- Pipelining can improve the throughput (the number of tasks to be processed in a unit time) (Pipelining은 처리량 (단위 시간 당 처리할 작업 수)을 개선할 수 있다)
- Pipelining cannot improve the response time (the execution time of each single task) (Pipelining은 응답 시간 (각 단일 작업의 실행 시간)을 개선할 수 없다)

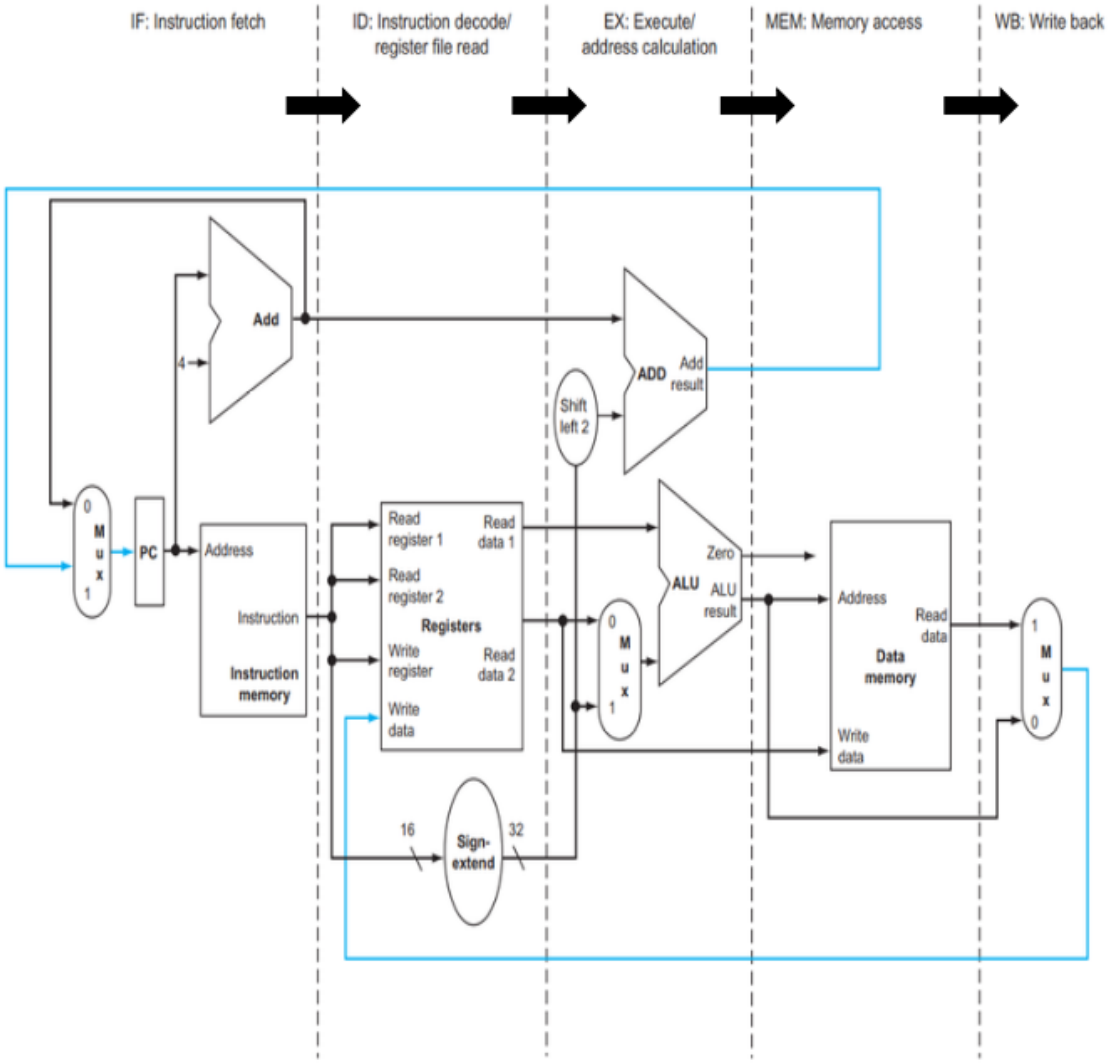
Five Stages of a Datapath



- Analogy (비유)
  - Task = Instruction (작업 = 지시 사항)
  - Stage = An activity with a hardware unit while processing the instruction (단계 = 명령어를 처리하는 동안 하드웨어 단위를 사용하는 활동)

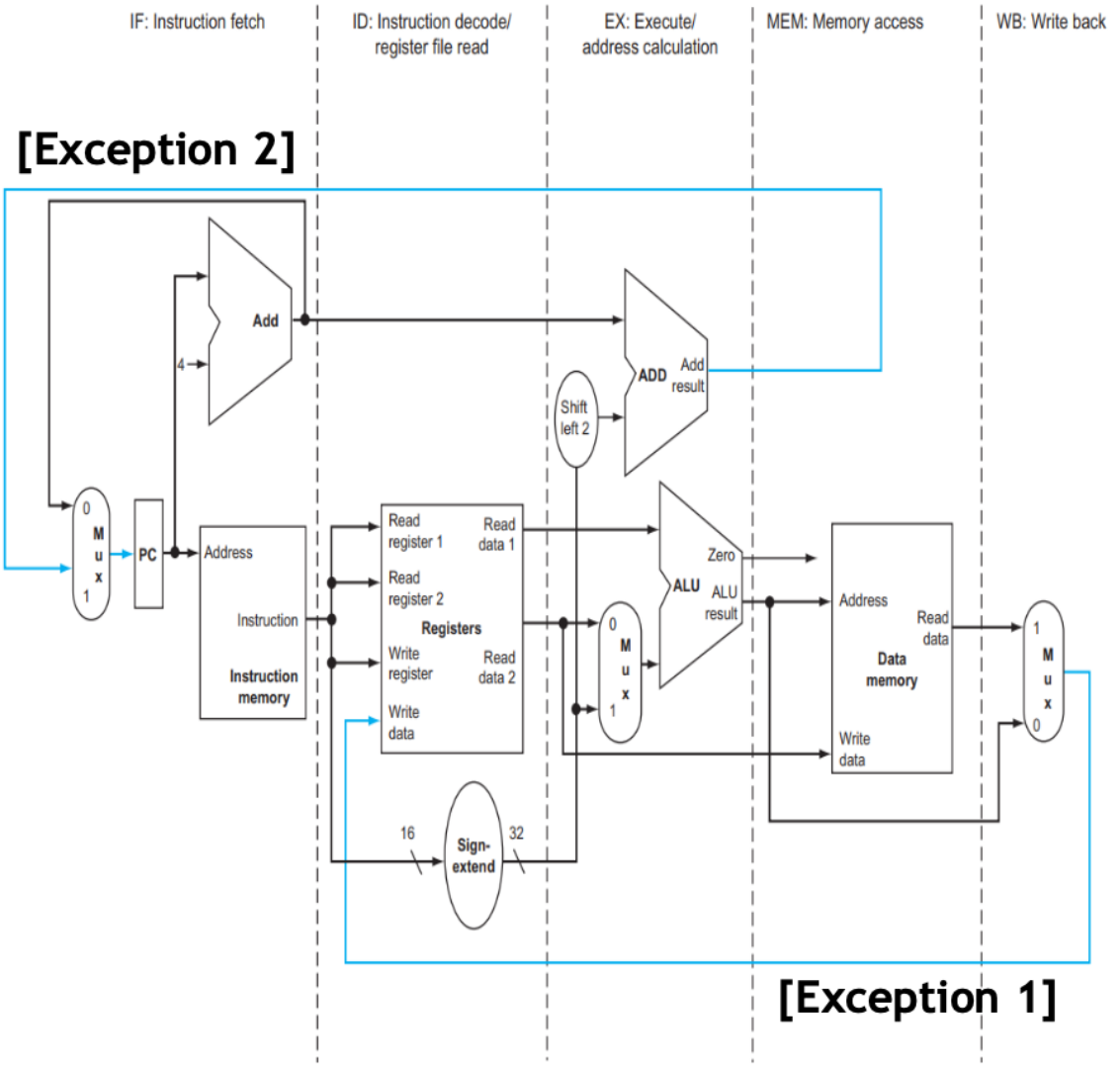
- Stages in the datapath (datapath의 단계)
  - (1) IF: Fetch instruction from memory (메모리에서 명령어 가져오기)
  - (2) ID: Read registers while decoding instruction (명령어를 decoding 하는 동안 레지스터 읽기)
  - (3) EX: Execute operation or compute address (연산 실행 또는 주소 계산)
  - (4) MEM: Access an operand in data memory (데이터 메모리에서 피연산자 접근)
  - (5) WB: Write the result into a register (결과를 레지스터에 쓰기)

Five Stages of a Datapath : Left-to-Right Flow (왼쪽에서 오른쪽 Flow)



- Let us employ pipelining in this datapath (datapath에서 pipelining을 사용하겠다)
    - Once 1st instruction moves from IF to ID, 2nd instruction can be started in IF (첫 번째 명령어가 IF에서 ID로 이동하면 두 번째 명령어가 IF에서 시작될 수 있다)
    - When 1st instruction moves from ID to EX, 2nd and 3rd instructions move to ID and IF, respectively (첫 번째 명령어가 ID에서 EX로 이동하면 두 번째 및 세 번째 명령어가 각각 ID와 IF로 이동한다)
- Pipelining is feasible in a datapath since instructions move generally from left to right through the five stages (명령어는 일반적으로 5단계를 통해 왼쪽에서 오른쪽으로 이동하기 때문에 데이터 경로에서 Pipelining이 가능하다)
    - In the laundry analogy, clothes also get cleaner, drier, and more organized as they move through the line from left to right (빨래에 비유하자면, 옷은 왼쪽에서 오른쪽으로 라인을 따라 이동하면서 더 깨끗해지고, 더 건조해지고, 더 정돈된다)
    - But, they never move backwar (하지만 절대 뒤로 이동하지 않는다)

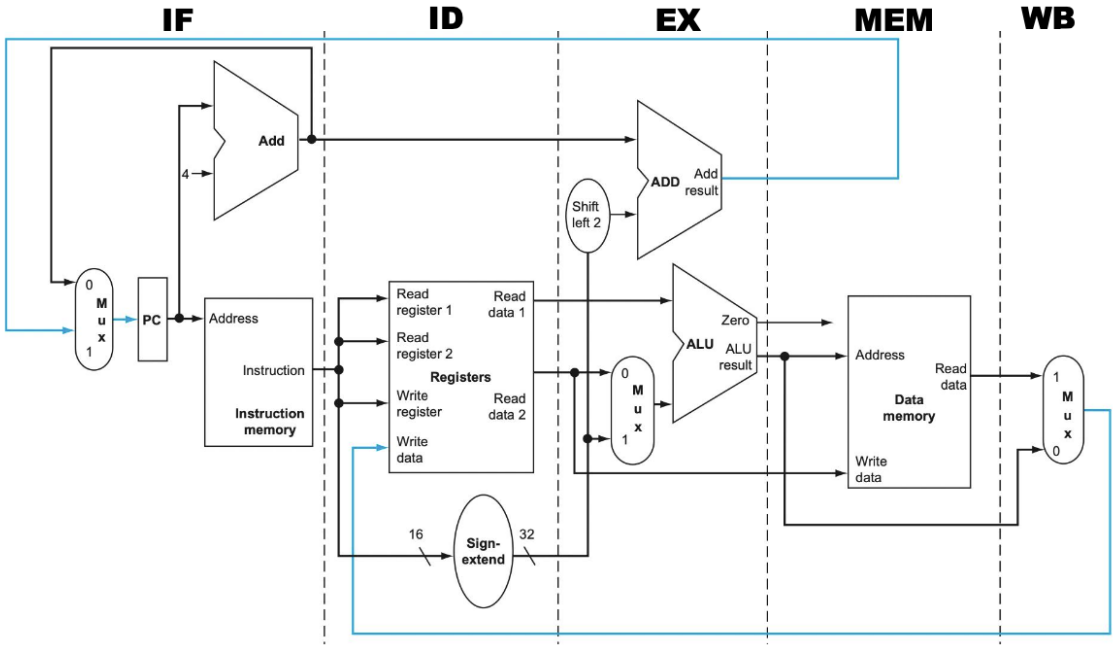
Five Stages of a Datapath : Two Exceptions (두 가지 예외 사항)



- There are two exceptions to this left-to- right flow of instructions (왼쪽에서 오른쪽으로의 명령 흐름에는 두 가지 예외가 있다)
  - Data can move from right to left (데이터가 오른쪽에서 왼쪽으로 이동할 수 있다)
  - These reverse data movements influence following instructions in the pipeline (이러한 역방향 데이터 이동은 pipeline의 다음 명령어에 영향을 미친다)
- Exception 1: WB stage (예외 1 : WB 단계)
  - This stage places the result back into the register file in the middle of the datapath (이 단계에서는 결과를 데이터 경로의 중간에 있는 레지스터 파일에 다시 배치한다)

    - This influences the instruction now in ID stage (이는 현재 ID 단계의 명령어에 영향을 미친다)
- Exception 2: Update of PC (예외 2 : PC 업데이트)
    - The next value of the PC should be selected from the incremented PC (PC+4) or the branch target address (PC의 다음 값은 증가된 PC (PC+4) 또는 분기 대상 주소에서 선택해야 한다)
    - The branch target address is obtained in EX stage, and it needs to move to the PC (branch 대상 주소는 EX 단계에서 얻어지며, PC로 이동해야 한다)
    - This influences the instruction now in IF stage (이것은 이제 IF단계의 명령어에 영향을 미친다)

Need of keeping Information of Instructions (명령어의 정보 보관의 필요성)




```
시간 t : beq $t1, $t2, 100
시간 t+1 : lw $s1, 10($s2) beq $t1, $t2, 100
시간 t+2 : 3rd instruction lw $s1, 10($s2) beq $t1, $t2,
100
```

- There is another serious problem in the pipelined datapath (pipelined datapath에는 또 다른 심각한 문제가 있다)

- The information of an instruction in a stage is lost when the next instruction enters to the stage (다음 명령어가 단계에 진입할 때 단계의 명령어 정보가 손실된다)
    - @t, beq is fetched in IF (t, beq는 IF에서 가져온다)
    - @t+1, beq moves to ID (t+1, beq는 ID로 이동한다)
    - @t+1, lw is fetched in IF (t+1, lw는 IF에서 가져온다)
        - Here, (PC+4)+4; but, beq needs PC+4 (여기서, (PC+4)+4; 하지만 beq는 PC+4가 필요하다)
    - @t+2, beq & lw move to EX & ID (t+2, beq & lw는 EX & ID로 이동한다)
        - Here, a sign-extended value of 10 can be given to the adder; but, beq needs a sign-extended value of 100 (여기서 부호가 확장된 값 10을 덧셈기에 지정할 수 있지만, beq는 부호가 확장된 값 100이 필요하다)
- Similar problematic situations can happen to every stage (모든 단계에서 비슷한 문제가 발생할 수 있다)

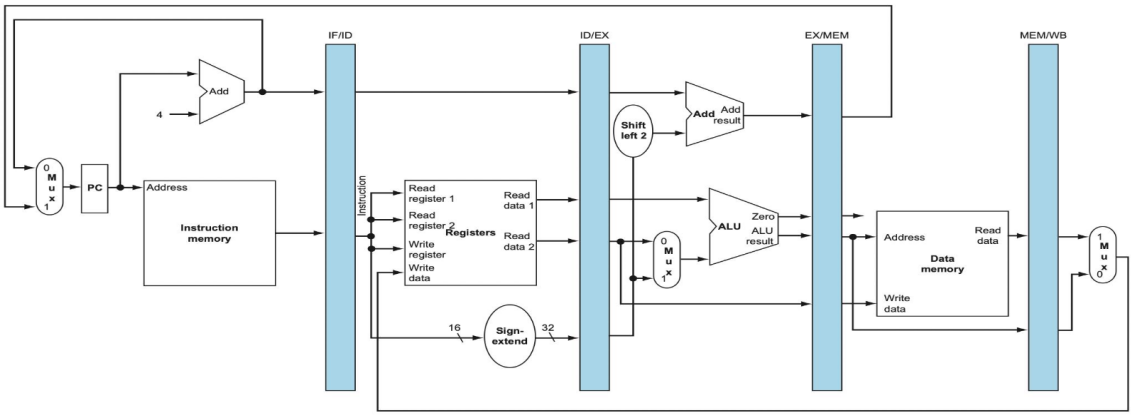
Chat GPT 4 요약

제공된 다이어그램과 텍스트는 파이프라인 처리되지 않은 데이터 경로에서 명령어 파이프라이닝과 관련된 컴퓨터 아키텍처에서 흔히 발생하는 문제를 설명하고 있습니다. 특히 강조된 문제는 다양한 명령어가 프로세서의 단계를 거치면서 명령어 정보가 손실되는 문제입니다. 문제의 상세 내용은 다음과 같습니다:

1. **명령어 컨텍스트의 손실**:
    - 파이프라인 처리되지 않은 프로세서에서는 하나의 명령어가 모든 단계(인출, 디코드, 실행, 메모리 접근, 쓰기)를 완료한 후 다음 명령어가 첫 번째 단계를 시작합니다. 그러나 파이프라인 프로세서에서는 여러 명령어가 동시에 다른 단계에 있을 수 있습니다. 문제는 특정 명령어가 필요로 하는 특정 조건이나 컨텍스트가 다른 명령어가 동일한 하드웨어 단계로 진입하면서 덮어쓰어지거나 사용할 수 없게 될 수 있다는 것입니다.
2. **구체적인 예시**:
    - 시간 `t`에, `beq` (분기 동일 비교) 명령어가 인출됩니다. 이 명령어는 프로그램 카운터(PC)가 특정 값에 있어야 분기 목표를 계산할 수 있습니다.
    - 시간 `t+1`에, `beq`는 디코드 단계로 이동하고 새로운 명령어 `lw` (워드 로드)가 인출됩니다. 여기서 문제가 발생하는데, `lw` 명령어는 주소를 `(PC+4) + 4`를 사용하여 계산하는 반면 `beq`는 `PC+4`만 필요로 합니다. 이로 인해 프로그램 카운터 업데이트가 잘못되고 결과적으로 메모리 접근 오류가 발생할 수 있습니다.
    - 시간 `t+2`에, 두 명령어가 다음 단계로 이동하면서 필요한 부호 확장 값이 적절히 처리되지 않아 추가적인 문제가 발생합니다. `beq`는 부호 확장된 값 100이 필요하지만, 파이프라인의 명령어 중첩으로 인해 처리 장치가 10의 값만 제공할 수 있습니다.
3. **일반적인 함의**:
    - 이러한 중첩은 잘못된 분기 결정, 잘못된 데이터 인출 또는 쓰기, 결국 프로그램 실패 또는 충돌과 같은 심각한 운영 문제를 초래할 수 있습니다. 파이프라인의 각 단계는 다른 명령어의 파이프라인에 방해받지 않고 각 명령어가 처리하는 의존성과 특정 요구사항을 관리해야 합니다.

이 상황은 컴퓨터 아키텍처에서 명령어 파이프라인의 신중한 설계와 관리가 중요한 이유를 명확하게 보여줍니다. 파이프라인 정지, 포워딩 및 기타 위험 제어 형태와 같은 기술이 파이프라인 아키텍처에서 이러한 유형의 오류를 방지하기 위해 필요합니다.

Pipelined Datapath with Pipeline Registers



- We place (pipeline) registers between any two stages (두 단계 사이에 (pipeline) 레지스터를 배치한다)
  - In the laundry analogy, we might have a basket between any two steps to hold the clothes (빨래를 비유하자면, 두 단계 사이에 빨래를 담는 바구니가 있을 수 있다)
  - The registers are named for the two stages separated by that register (e.g., ID/EX btw ID and EX) (레지스터는 해당 레지스터로 구분된 두 단계의 이름을 따서 이름이 지정된다 : e.g., ID와 EX 사이면 ID/EX)
- All instructions advance from one pipeline register to the next at a time (모든 명령은 한 번에 한 pipeline 레지스터에서 다음 레지스터로 진행된다)
  - As an instruction advances, its required information is also copied to the next pipeline registers (명령어가 진행됨에 따라 필요한 정보도 다음 pipeline 레지스터로 복사된다)