# 명령어 실행 하드웨어

# 1. Micro-architecture 개요

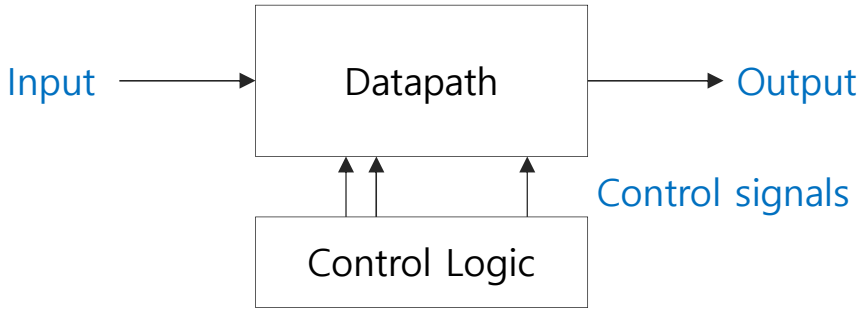
- **명령어 실행 하드웨어**
  - Micro-architecture
    - 유연한 디지털 시스템
    - 제어 로직이 가변적 (프로그래밍 가능)

- **Micro-architecture**
  - 주어진 명령어 세트 실행에 최적화된 하드웨어 구조
    - 데이터 경로부(Datapath)와 제어 로직(Control Logic)으로 구성.

# 2. Micro-architecture 구조

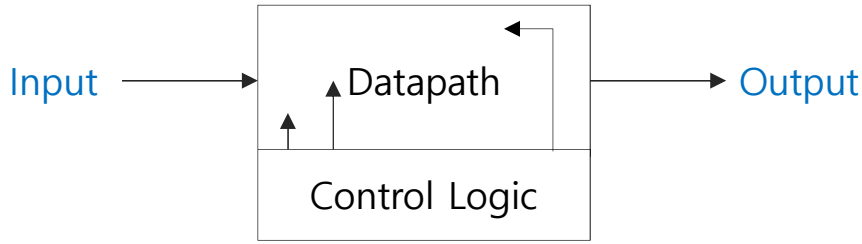
- **디지털 시스템 일반**
  - 디지털 시스템은 데이터 경로부(Datapath)와 제어부(Control Logic)로 구성.
  - Datapath
    - 데이터의 흐름 경로
    - 입력→처리→출력
  - Control Logic
    - Datapath의 데이터 흐름을 제어하는 회로.

- **디지털 시스템 구현방법**
  - Hardwired system
  - Programmable system



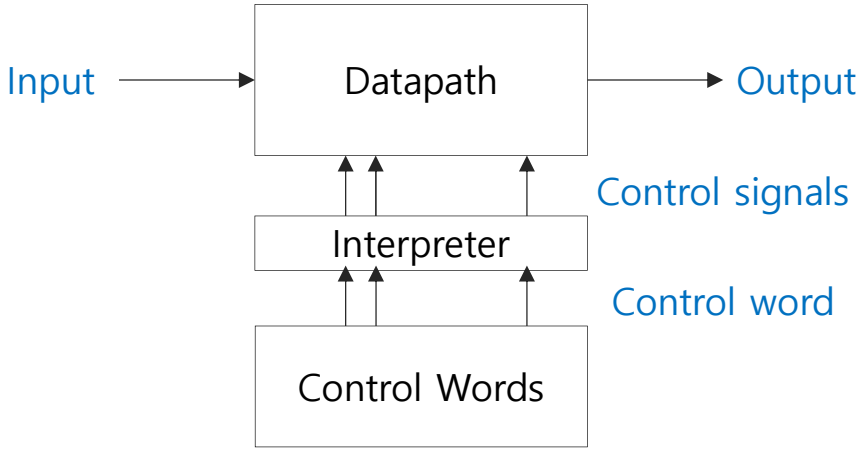
# 2.1 Hardwired System

- **Hardwired System**
  - Datapath와 Control Logic을 일체화
    - 유연성 부족 : 데이터 흐름 변경 → 시스템 재설계
    - 실행속도 향상 : 최적화된 데이터 흐름과 최적화된 제어회로 구현가능
  - Control Logic을 로직회로를 사용하여 직접 구현



# 2.2 Programmable System

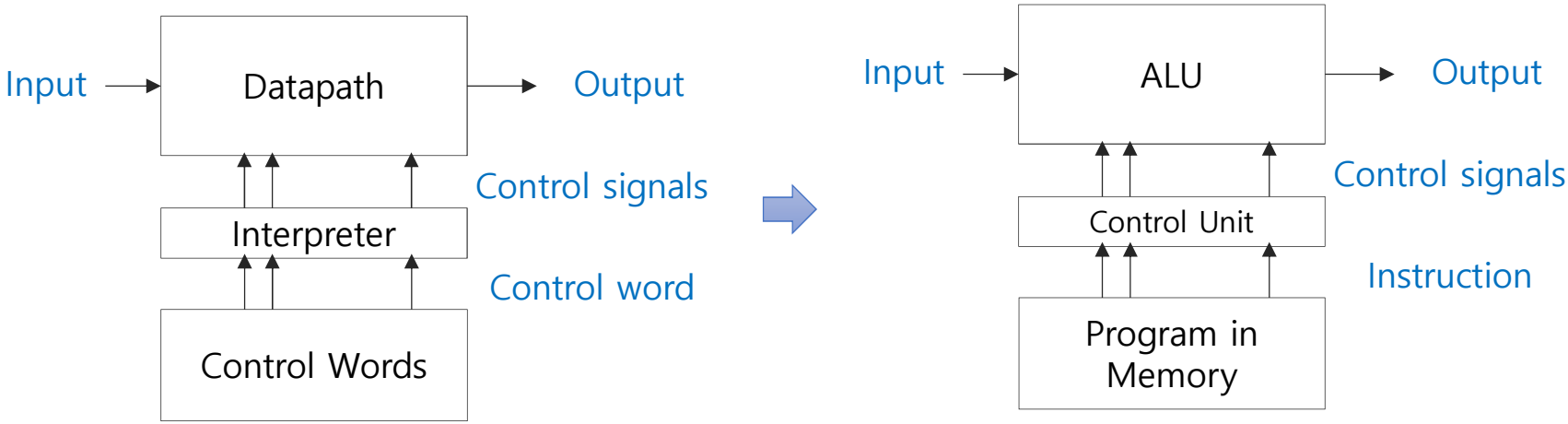
- **Programmable System**
  - Datapath와 Control Logic을 분리
  - 재구성 가능한 Datapath 적용
  - 유연한 Control Logic 구현
    - 시스템 유연성 극대화
    - 실행성능 하락




- **유연한 Control Logic**
  - Control Signal 들을 Control Word로 정의
  - Control Word들을 순서적으로 Datapath에 적용
    - 해석기(Interpreter)를 사용하여 Control Word를 Control Signal들로 변환.
  - Control Word의 Datapath 적용방법
    - 직접입력
    - 메모리에 저장후 순서적으로 읽어서 적용.
  - 제어방법의 변경 → Control Word를 재정의하거나 실행순서를 재정의.

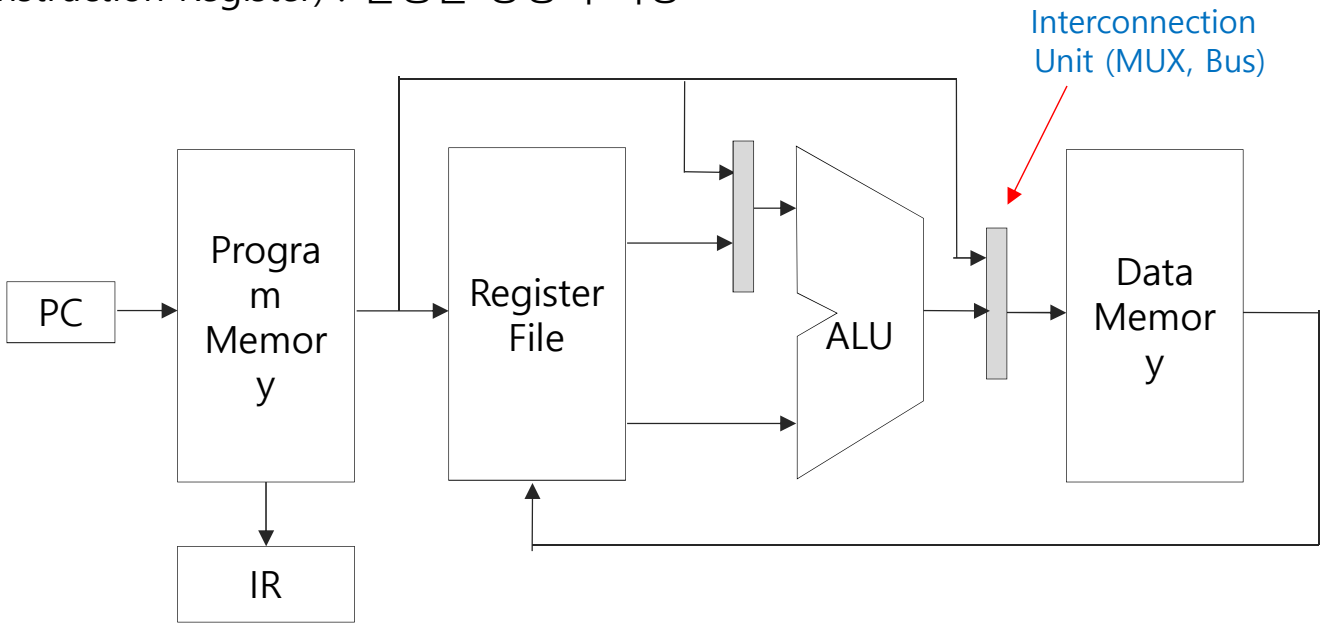
## 2.3 Programmable System과 Micro-architecture 관계

- **Micro-architecture는 대표적인 Programmable System.**
  - Datapath = ALU
  - Control Word = Instruction (인코딩된 Control word)
  - Control Word들의 순서적 배열 = Program
  - Control Word의 적용방법 = 메모리에 저장후 실행(Stored Program 방식)
  - Control Word 해석기 = Control unit의 instruction decoder



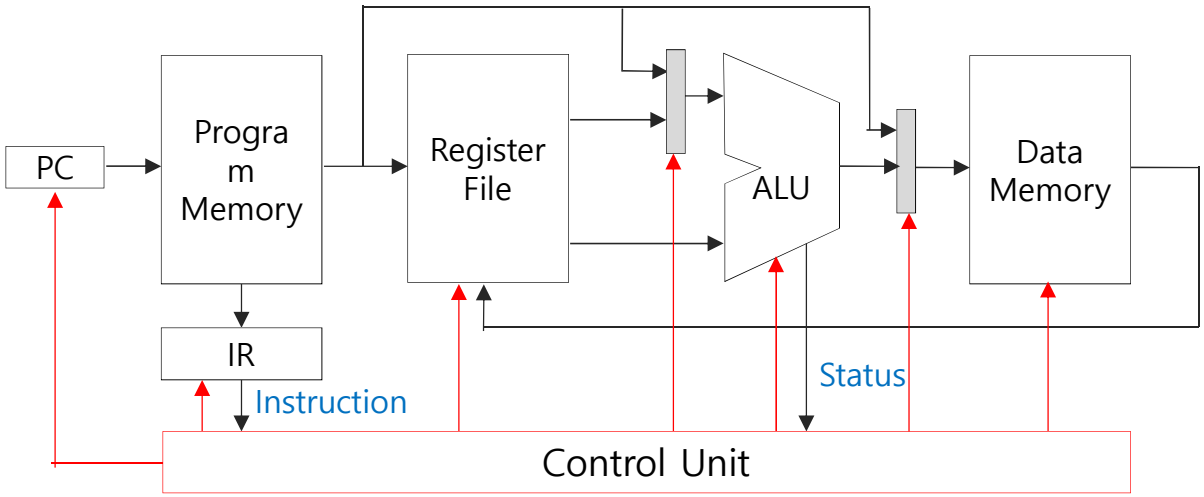
# 3. Micro-architecture의 Datapath

- **Micro-architecture의 Datapath는 ALU와 레지스터 파일(Register File)로 구성된다.**
  - 레지스터 파일 : 레지스터 집합.
  - PC(Program Counter) : 실행할 명령어의 저장위치
  - IR(Instruction Register) : 실행할 명령어 저장



# 4. Micro-architecture의 Control Unit

- **Control Unit**
  - 명령어를 해석하여 명령어 실행에 필요한 제어신호들만을 활성화 시킴.
  - Micro-architecture의 Datapath의 구성요소에 필요한 모든 제어신호 생성.



# 4. Micro-architecture 설계 절차

- **Micro-architecture 설계 절차**
  - 명령어 세트 분석
  - 각 명령어에 대한 Micro-operation 정의
  - 모든 Micro-operation 실행에 필요한 Datapath 구성요소 및 상호연결 구조 정의
  - Datapath 구성요소 및 연결 회로에 필요한 제어신호 정의
  - Control Unit 설계
    - 명령어 해석기(Instruction Decoder)설계
    - Control Sequencer 설계
    - Control Logic 회로 설계

# 명령어 싸이클의 RTL 동작

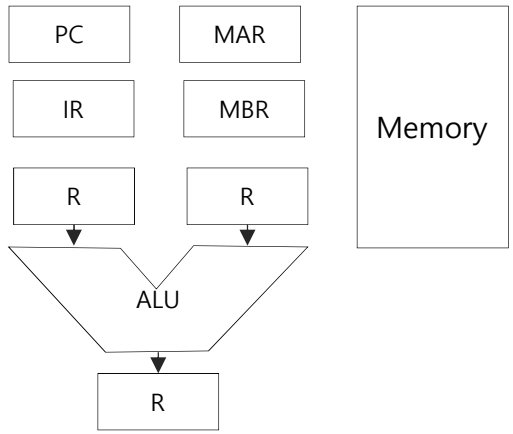
# 1. 명령어 사이클 예시

- **컴퓨터의 명령어 사이클(Instruction Cycle)을 RTL 수준에서 설명**
- **명령어 사이클 (예시)**
  - Instruction Fetch (IF)
    - 프로그램 메모리에서 명령어 가져오기
  - Instruction Decode (ID)
    - 명령어 해석하기
  - Data Fetch (DF)
    - 명령어 실행에 필요한 데이터 가져오기
  - Execute (EX)
    - 명령어 실행
  - Write Back (WB)
    - 실행 결과 저장

# 1.1 컴퓨터의 RTL 구성요소 예시

- **컴퓨터의 명령어 사이클 설명을 위한 컴퓨터의 RTL 구성 요소**
  - PC(Program Counter) : 실행할 명령어의 저장위치 정보(어드레스) 저장
  - IR(Instruction Register) : 실행할 명령어 저장
  - MAR(Memory Address Register) : 메모리 접근에 사용되는 어드레스 저장
  - MBR(Memory Buffer Register) : 메모리에 읽거나 쓸 데이터 값
  - R(Register) : 데이터 임시 저장용 레지스터
  - ALU(Arithmetic & Logic Unit) : 산술/논리 연산회로
  - 메모리 : 명령어와 명령어 실행에 필요한 데이터 저장

간단한 컴퓨터 RTL 구성 요소



# 1.2 IF 동작 사이클

- **Instruction Fetch(IF) 동작 사이클의 동작**
  - 프로그램 메모리에서 명령어를 읽어오는 동작 사이클
    - 명령어의 저장위치에 대한 정보 : PC(Program Counter) 에 저장
    - 읽어온 명령어의 저장위치 : IR(Instruction Register)

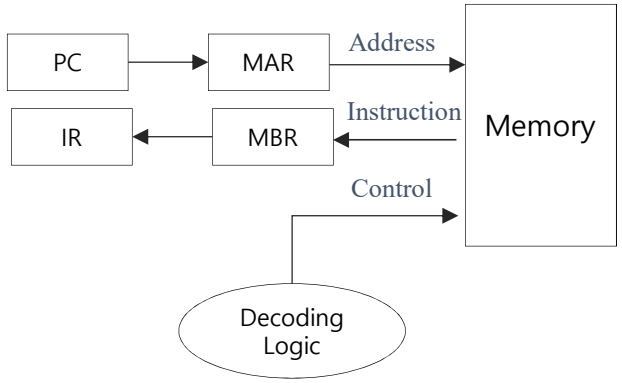
RTL Description

t1: MAR ← PC
t2 : MBR ← M[MAR]
t3 : IR ← MBR

  - 다음 명령어를 가져오기 위해 PC 값 업데이트
    - PC++ : 어드레스 증가량은 명령어 길이에 따라 다름
    - 실행위치는 t2 또는 t3

RTL Description

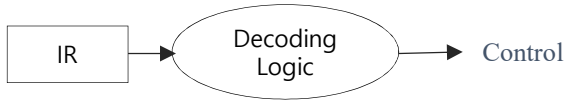
t1: MAR ← PC
t2 : MBR ← M[MAR]
t3 : IR ← MBR, **PC++**




Logic Diagram

# 1.3 ID 동작 사이클

- **Instruction Decode (ID) 동작 사이클의 동작**
  - IF 동작 사이클에서 읽어온 명령어를 해석해서 각 컴퓨터 구성요소에서 필요한 제어신호 발생.
  - 제어 유니트 구성방법에 따라 ID 동작 사이클에 소요되는 클럭 수가 달라짐
    - 제어신호 직접전달
      - ID 동작 사이클 생략가능.
      - ID 동작 사이클을 IF 동작 사이클에 포함




    - 제어신호 Latching
      - 생성된 제어신호를 래치에 저장

RTL Description t1: C ← IR Decoding

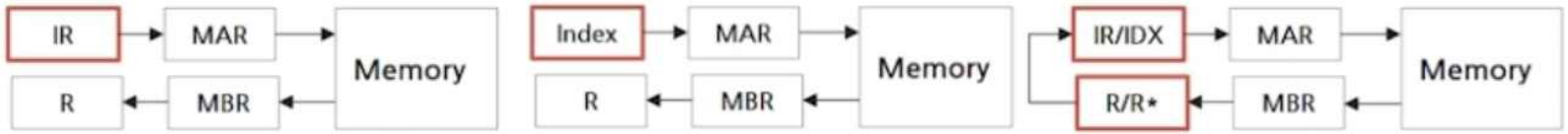

Logic Diagram

# 1.4 OF 동작 사이클

- **Operand Fetch (OF) 동작 사이클의 동작**
  - ID 동작 사이클에서 해석결과 연산에 필요한 데이터(오퍼랜드)가 필요한 경우, 데이터 메모리에서 데이터를 읽어온다.
  - 실행할 명령어 유형에 따라, 데이터 저장위치 정보(어드레스)의 저장장소와 어드레스 해석 방법이 다르다.
    - 어드레스 저장위치
      - 명령어 레지스터(IR)
      - 또 다른 레지스터 : 인덱스(Index) 레지스터 등
      - 메모리 : 어드레스가 메모리에 저장되어 있는 경우. 다양한 어드레싱 방법 적용.
    - 어드레스 해석방법
      - 어드레싱 모드에 따라 다름 : 직접 어드레싱, 간접 어드레싱.

# 1.4 OF 동작 사이클

- **Operand Fetch (OF) 동작 사이클의 동작 (계속)**
  - 오퍼랜드 접근을 위한 어드레스 계산이 필요한 경우, 어드레스 계산 사이클이 추가될 수 있다.




RTL Description

t1: MAR ← IR.addr
t2: MBR ← M[MAR]
t3: R ← MBR

t1: MAR ← Index
t2: MBR ← M[MAR]
t3: R ← MBR

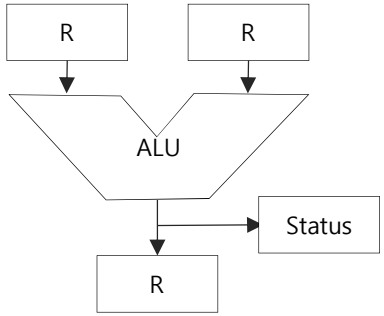
t1: MAR ← IR/IDX
t2: MBR ← M[MAR]
t3: R ← MBR
t4: MAR ← R
t5: MBR ← M[MAR]
t6: R* ← MBR

# 1.5 EX 동작 사이클

- **Execute (EX) 동작 사이클의 동작**
  - ID 동작 사이클에서 해석결과에 따라 필요한 연산을 수행한다.
    - 연산 유형 : 산술/논리 연산, 데이터 전송.
  - 실행할 연산 유형에 따라 실행시간 차이 발생.

t1: R3 ← R1 + R2

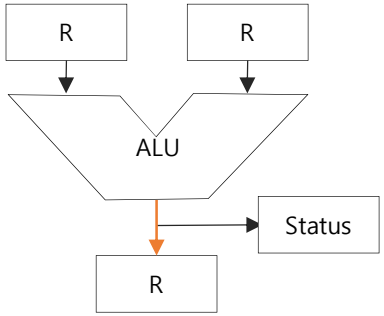
RTL Description




Logic Diagram

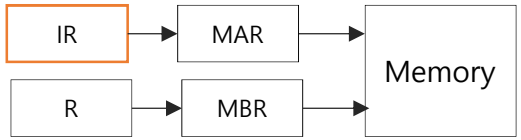
# 1.6 WB 동작 사이클

- **Write Back (EX) 동작 사이클의 동작**
  - EX 실행 결과를 레지스터 또는 메모리에 저장한다.
    - 레지스터에 저장하는 경우, 레지스터 구조에 따라 WB 사이클 불필요할 수 있다.
    - 메모리에 저장하는 경우, OF 동작사이클처럼 다양한 어드레싱 방법을 사용할 수 있다.




t1: R3←R1 + R2

레지스터에 저장하는 경우



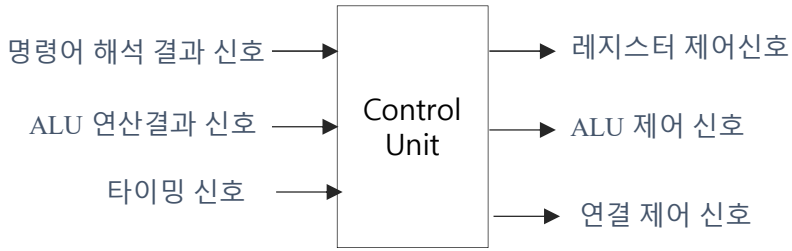

RTL Description

t1: MAR ← IR
t2 : MBR ← R
t3 : M[MAR] ← MBR

메모리에 저장하는 경우

# 2. 제어 유니트 설계

- **제어 유니트(Control Unit)**
  - 각 구성요소에서 필요한 제어신호 발생하는 회로
- **제어 유니트 출력신호**
  - RTL 동작제어에 필요한 제어신호.
  - 레지스터 제어신호 : load 신호
  - ALU 제어신호 : 연산 선택 신호 (function select) 신호
  - 레지스터 상호 연결에 필요한 제어신호
    - 연결유형에 따라 멀티플렉서 선택 제어신호 또는 3-state 버퍼의 제어신호.
- **제어 유니트 입력신호**
  - 명령어 해석 결과 신호 : ID 동작 사이클에서 실시한 명령어 해석 결과 신호
  - ALU 연산 결과 신호 : EX 동작 사이클에서 발생한 연산 결과에 따른 상태 신호
  - 타이밍 신호 : 제어 유니트에서 생성하는 타이밍 신호.

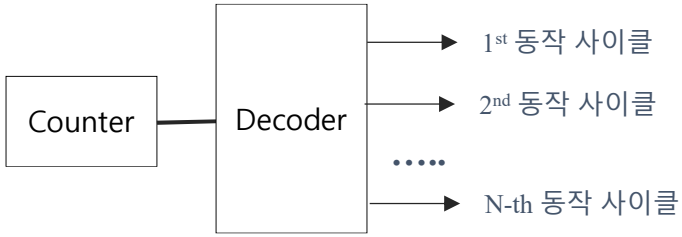


## 2.1 제어 유니트의 타이밍

- **제어 유니트에 발생하는 제어신호들은 제어 타이밍에 맞춰 활성화 된다.**
- **제어 타이밍 결정 요소**
  - 명령어 해석 결과
  - 명령어 사이클을 구성하는 동작 사이클 타이밍
    - 동작 사이클 타이밍 발생기에서 동작 사이클 타이밍 신호 발생
  - 동작 사이클을 구성하는 MOP 수준 타이밍
    - MOP 레벨 타이밍 발생기에서 MOP 실행 타이밍 신호 발생

# 2.2 동작 사이클 타이밍 생성

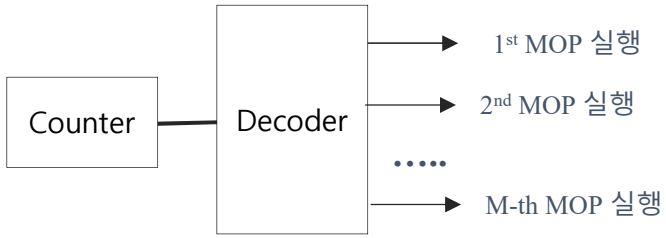
- **동작 사이클 타이밍 발생기(Operation Cycle Timing Generator)**
  - 명령어 사이클을 구성하는 최대 동작 사이클 수에 의해 타이밍 상태 수 결정
  - 타이밍 상태 천이는 카운터로 구현
    - 예) 명령어 사이클을 구성하는 최대 동작 사이클 수 = 5 인 경우, 3-비트 이진 카운터로 구성




동작 사이클 타이밍 발생기

# 2.3 MOP-레벨 타이밍 생성

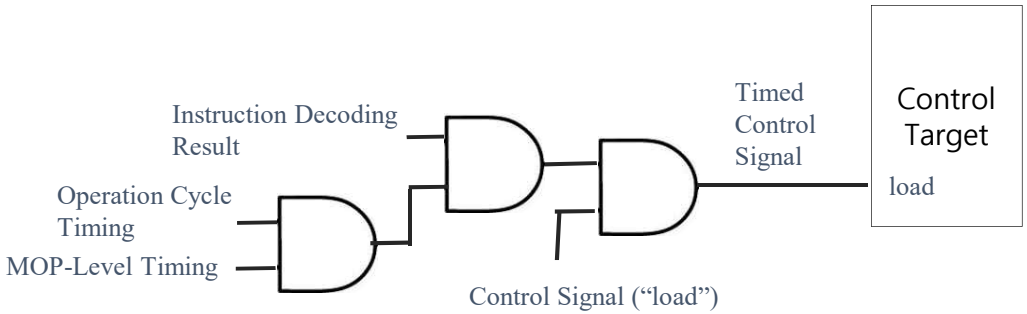
- **MOP-레벨 타이밍 발생기 (MOP-Level Timing Generator)**
  - 동작 사이클을 구성하는 최대 MOP-레벨 동작의 수에 의해 결정
  - 디코더를 사용하여 MOP-레벨 타이밍 신호를 활성화.
  - 예) 명령어 사이클을 구성하는 동작 사이클이  최대 3개의 MOP-레벨 동작으로 구성되었다면, 2-비트 카운터로 구현.



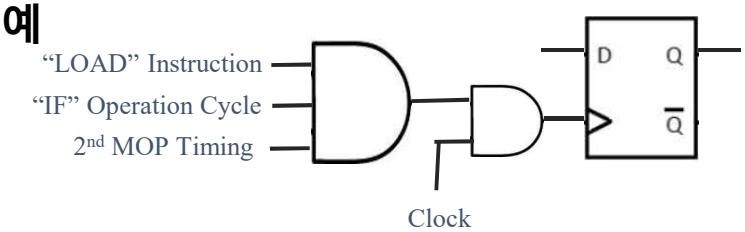

MOP-레벨 타이밍 발생기

# 2.4 제어 회로 구현

- 제어신호들은 제어 타이밍에 맞춰 제어 타겟에 적용된다.




- 실제 구현 예

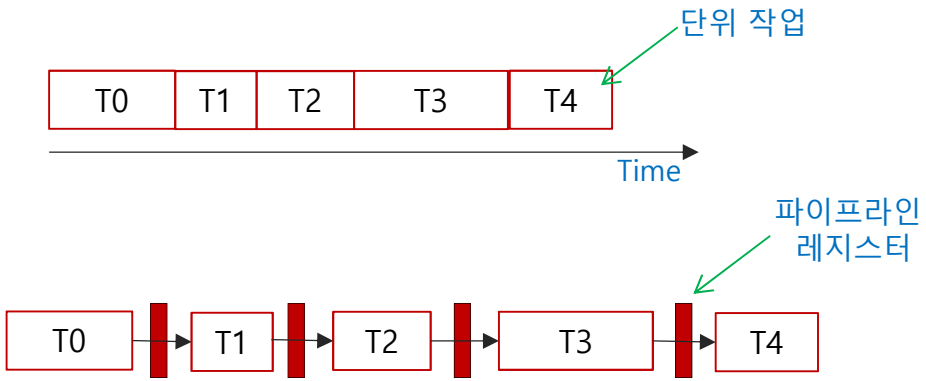



예) LOAD 명령어의 IF 동작 사이클의 2번째 MOP 에서 활성화

# 명령어 파이프라인

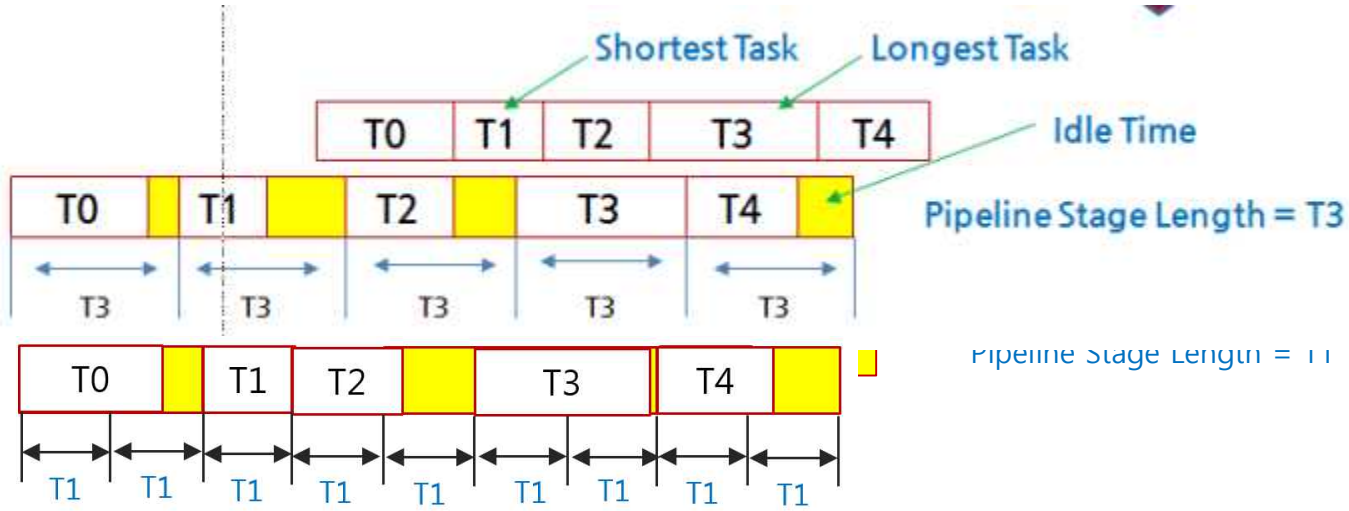
# 1. 파이프라인 개요

- **파이프라인**
  - 하나의 작업을 여러 개의 단위 작업으로 나누어 연속적으로 실행하는 것
    - 단위 작업 : Stage
    - 단위 작업사이의 중간결과 저장소 : 파이프라인 레지스터
  - 공장에서의 조립과정과 유사



# 1. 파이프라인 개요

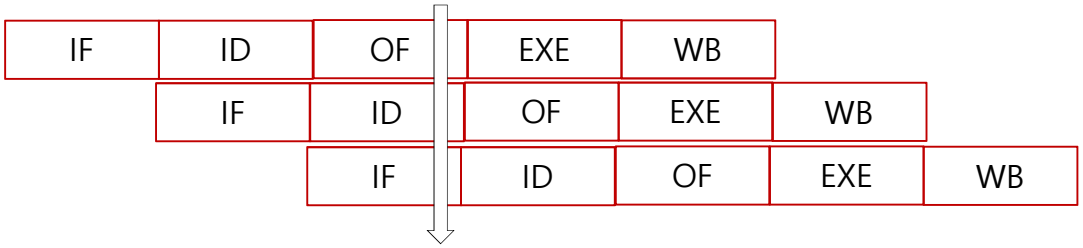
- **파이프라인 스테이지(Stage) 길이 결정 방법**
  - 실행시간이 가장 긴 단위 작업 시간을 파이프 스테이지 길이로 결정
    - 스테이지내의 휴지시간(Idle Time)이 증가
  - 실행시간이 가장 짧은 단위 작업 시간을 파이프 스테이지 길이로 결정
    - 파이프 라인 스테이지 수 증가



# 2. 명령어 파이프라인

- **명령어 실행과정에 파이프라인 적용**
  - 작업 : 명령어 싸이클
  - 단위 작업 : 동작 싸이클

- **명령어 파이프라인**
  - 하나의 명령어 싸이클을 구성하는 여러 개의 동작 싸이클들을 파이프 라인 형태로 실행
  - 명령어 선인출(Prefetch) 가능
  - 여러 개의 명령어들을 중첩 실행하는 것이 가능




명령어의 중첩실행

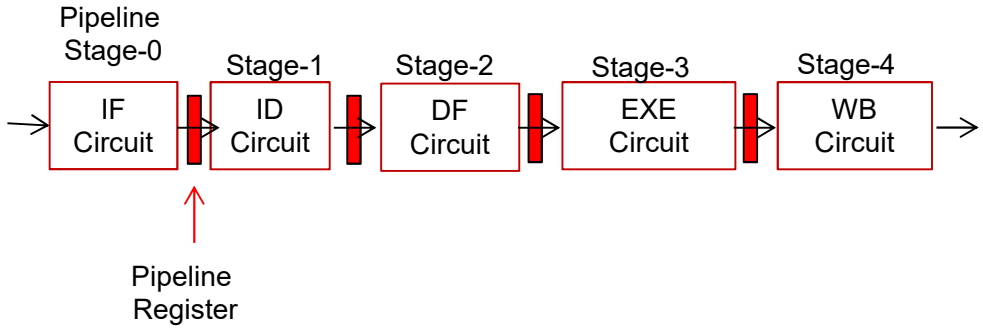
## 2.1 선인출 (Prefetch)

- **실행 싸이클은 메인 메모리를 액세스 하지 않음.**
  - 실행 싸이클 동안 다음 명령어를 미리 인출할 수 있음.
    - 명령어 선인출 (prefetch)

- **명령어 선인출시 고려사항**
  - 실행 싸이클이 인출 싸이클 보다 길다.
    - 여러 개의 명령어를 선인출 할 수 있을까 ?
  - 조건 분기 명령어 실행시 다음에 실행할 명령어의 주소를 알 수 없다.
    - 추정 (Guessing)

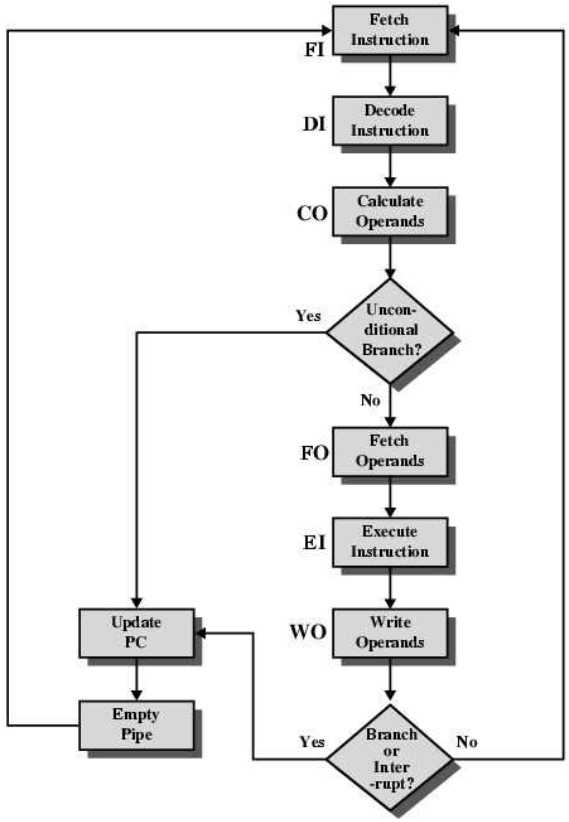
# 2.2 파이프라인 오버헤드

- **성능향상을 위해 파이프 라인 단계 (pipeline stage) 를 증가**
  - 파이프라인의 각 단계에서 버퍼들 간의 데이터 이동과 준비 및 전달 기능의 수행에 따른 오버헤드 (overhead) 발생.
  - 파이프라인의 단계가 많아질수록 기억장치와 레지스터 의존성 처리 및 파이프라인 사용의 최적화를 위한 제어 논리회로 증가.

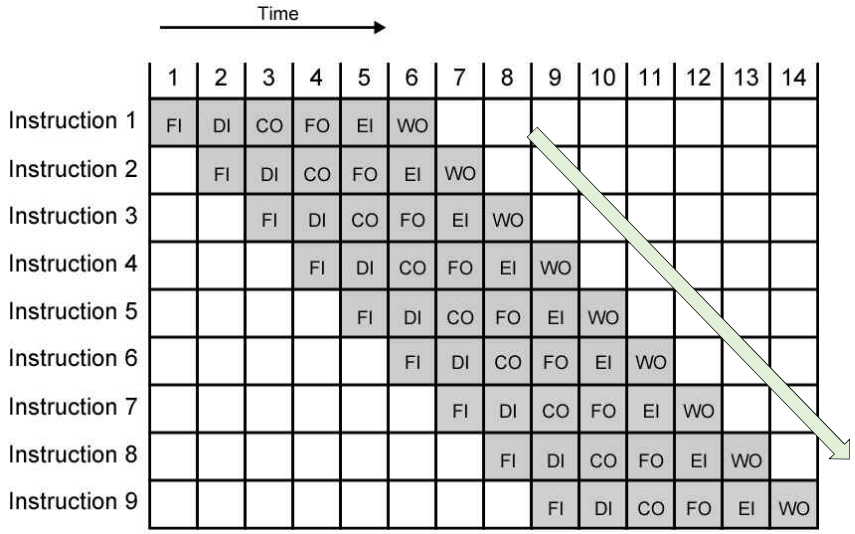


# 2.3 명령어 파이프라인 예시

- 6-Stage 파이프라인 경우



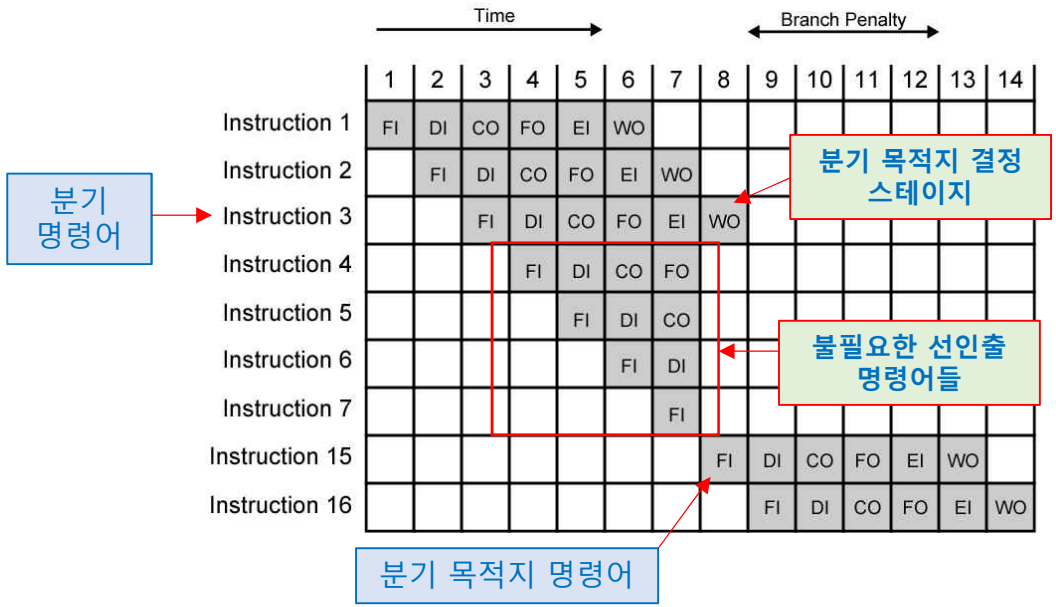
# 2.3.1 명령어 파이프라인 동작의 시간 흐름도




순차적 실행

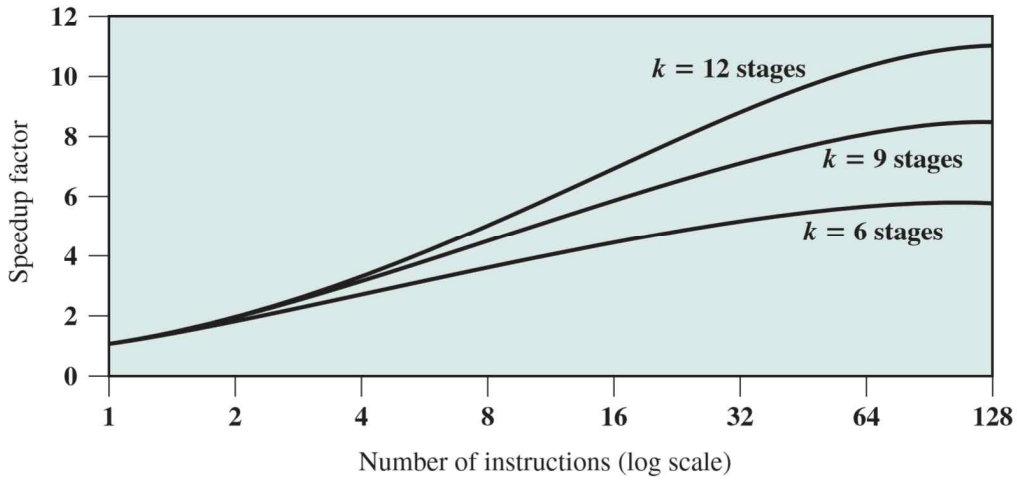
**FI : Fetch instruction**
**DI : Decode instruction**
**CO : Calculate operands (i.e. EAs)**
**FO : Fetch operands**
**EI : Execute instructions**
**WO : Write result**

# 2.3.2 조건분기가 명령어 파이프라인 동작에 미치는 영향



## 2.4 명령어 파이프라이닝에 따른 속도향상

- **파이프라인 Stage 증가에 따른 파이프라인 오버헤드 증가로 속도향상(Speedup)이 제한적임.**
  - 파이프 라인 오버헤드 : 파이프라인 레지스터 수 증가
  - 분기 명령어에서의  불필요한 선인출 명령어 제거 부담
  - 분기 명령어에서의 파이프라인 페널티(Penalty) 부담



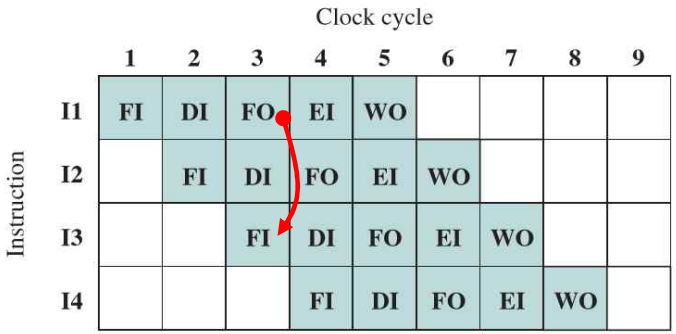
# 3. 파이프라인 해저드

- **파이프라인 실행조건이 만족되지 않을 경우, 파이프라인 실행을 계속 할 수 없어, 파이프라인 일부분이 멈추어야 (stall) 한다. 이러한 파이프 라인 멈춤을 파이프라인 해저드 (pipeline hazard) 라고 하며, 파이프 라인 버블 (pipeline bubble) 이라고도 한다.**

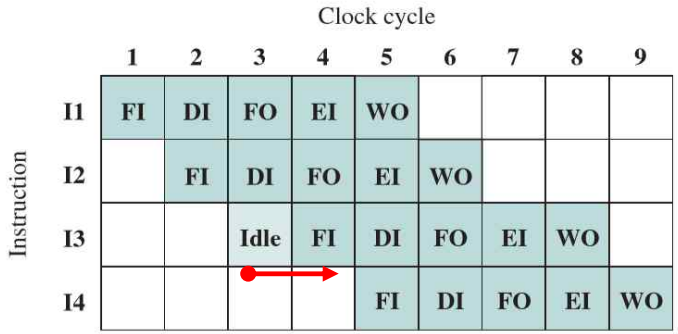
- **파이프 라인 해저드 유형**
  - 자원 해저드 (resource hazard)
  - 데이터 해저드 (data hazard)
  - 제어 해저드 (control hazard)

# 3.1 자원 해저드

- **자원 해저드 (resource hazard)**
  - 파이프라인에 들어와 있는 명령어들이 동일한 자원을 사용할 때 발생.
  - 구조적 해저드 (structural hazard).



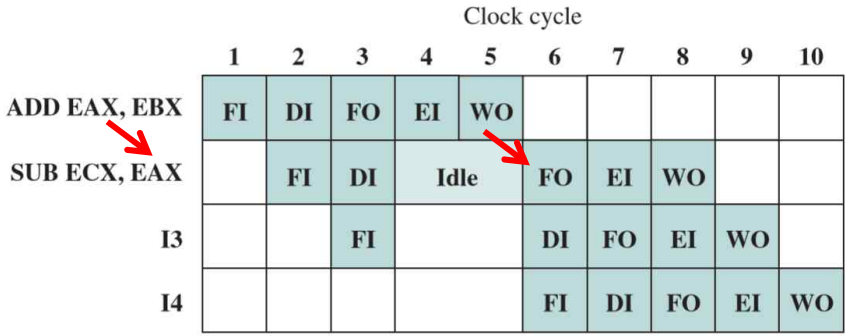

I1 명령어의 FO Stage 와 I3 명령어의 FI Stage 에서
동일한 메모리 접근




I3 명령어 시작 전에 Idle 스테이지
추가하여 메모리에 대한 동시 접근을 해소

# 3.2 데이터 해저드

- **데이터 해저드 (data hazard)**
  - 파이프 라인에 들어와 있는 명령어들 사이에 데이터 의존성이 존재할 때 발생.
    - 쓰기후 읽기 (RAW : read after write) : true dependency
    - 읽기후 쓰기 (WAR : write after read) : anti-dependency
    - 쓰기후 쓰기 (WAW : write after write) : output dependency



# 3.3 제어 해저드

- **제어 해저드 (control hazard)**
  - 파이프 라인에 들어와 있는 명령어들의 실행 흐름이 바뀔 때 발생.
    - 분기 목적지 예측이 잘못되어 이미 파이프 라인으로 들어온 불필요한 명령어들을 버려야 (flush) 할 때 발생.

- **제어 해저드를 최소화하는 방법**
  - 분기 목적지 예측의 정확도 향상
    - 다중 스트림
    - 분기 목적지의 선인출
    - 루프 버퍼
    - 분기 예측
    - 지연 분기

## 3.3.1 다중 스트림

- **다중 스트림(Multiple Stream)을 사용한 분기 목적지 예측 정확도 향상**
  - 2 개의 파이프라인을 사용.
  - 각 분기 코드를 2 개의 파이프 라인에서 각각 실행.
  - 조건에 따라 2 개중 하나를 선택해서 사용.

- **문제점**
  - 레지스터와 메모리 액세스 과정에서 경합 지연 (contention delay) 발생.
  - 분기 판단이 완료되기 전에 다른 분기 명령어가 파이프라인으로 들어올 수 있다.

- **IBM 370/168, IBM 3033 에서 사용.**

## 3.3.2 분기 목적지의 선인출

- **분기 목적지의 선인출(Prefetch Branch Target)를 사용한 분기 목적지 예측의 정확도 향상**
  - 조건 분기가 발생되었을 때, 분기 명령어의 다음 명령어와 함께, 분기 목적지의 명령어까지 함께 인출한다.
  - 분기 명령어가 실행될때까지 분기 목적지를 유지.

- **IBM 360/91 에서 사용**

## 3.3.3 루프 버퍼

- **루프 버퍼(Loop Buffer)를 사용한 분기 목적지 예측의 정확도 향상**
  - 루프 버퍼 : 파이프라인의 명령어 인출 단계에 포함되어 있는 소용량 고속 메모리.

- **분기 목적지 예측 절차**
  - 루프 버퍼에 가장 최근에 인출된 n 개의 명령어들이 순서대로 저장.
  - 분기가 발생하면, 하드웨어는 먼저 분기 목적지가 루프 버퍼에 있는지 검사.

- **장점**
  - 루프 버퍼는 현재의 명령어 보다 앞에 있는 명령어들을 저장하고 있어서 메모리 액세스시간이 필요 없다.
  - 분기명령어의 목적지가 버퍼에 포함되어 있는 경우, 그 목적지의 주소가 이미 버퍼에 저장됨.
    - 명령어 실행 루프나 반복 실행에 적합.

- **CRAY-1 에서 사용.**

## 3.3.4 분기 예측

- **분기 예측(Branch Prediction)을 사용한 분기 목적지 예측의 정확도 향상**

- **분기가 발생할 것인지를 미리 예측**
  - 정적 예측 (static prediction) : Predict never taken, predict always taken, predict by opcode.
  - 동적 예측 (dynamic prediction) : taken/not taken switch, branch history table

- **Predict never taken**
  - 분기가 발생하지 않을 것이라 예측
    - 항상 다음 명령어를 인출
  - 68020 & VAX 11/780 에서 사용.
  - 가장 많이 사용

- **Predict always taken**
  - 분기가 발생할 것이라 예측
    - 항상 분기 목적지 명령어를 인출

# 3.3.4 분기 예측

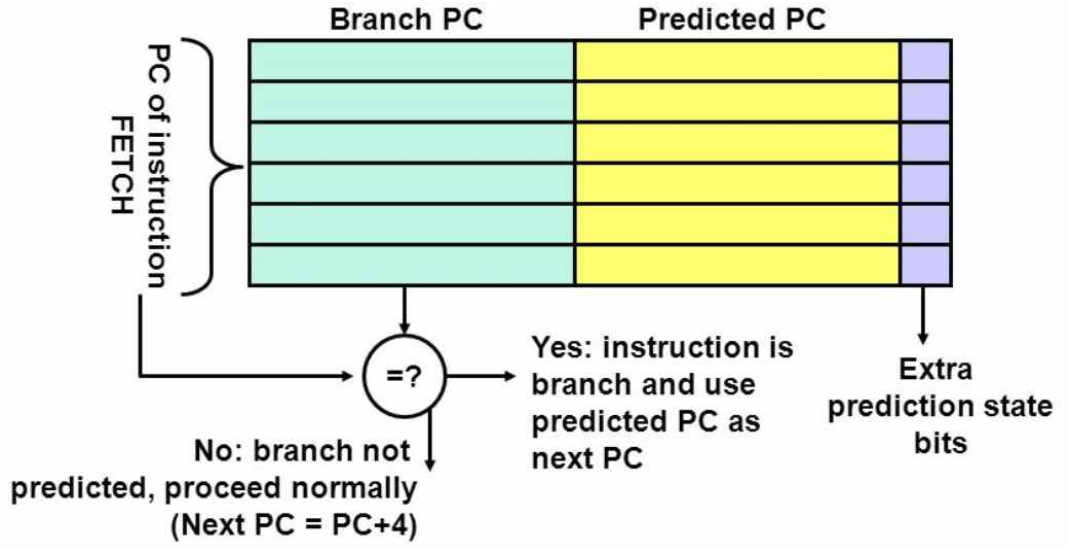
- **Predict by Opcode**
  - 분기 명령어의 연산 코드에 근거하여 결정.
  - 75% 성공률 발표

- **Taken/Not taken switch**
  - 이전 분기 이력 (history) 을 근거해서 예측.
    - History bits : 같은 분기 명령어를 만났을 때 어떤 판단을 내릴지를 알려주는 비트.
  - 루프에 적합.

- **Branch History Table (Branch Target Buffer) 사용한 분기 목적지 예측**
  - BTB : 소용량 캐시 메모리, 분기 목적지에 대한 정보를 저장하고 있는 Lookup Table
    - 분기 명령어 주소.
    - History bits
    - 목적지 명령어에 대한 정보 (주소 또는 명령어)
  - 명령어 인출단계에서 BTB를 참조하여 분기 목적지를 예측

## 3.3.4 분기 예측

- BTB 블록 다이어그램



## 3.3.5 지연 분기 (Delayed Branch)

- **지연 분기(Delayed Branch)를 사용한 분기 목적지 예측의 정확도 향상**
- **Delayed Branch**
  - 분기 발생 시점을 지연시킨다.
  - 명령어 재배치
    - 분기에 영향을 주지 않는 명령어들을 분기 명령어의 다음에 배치함으로써, 분기에 의한 손실을 최소화.

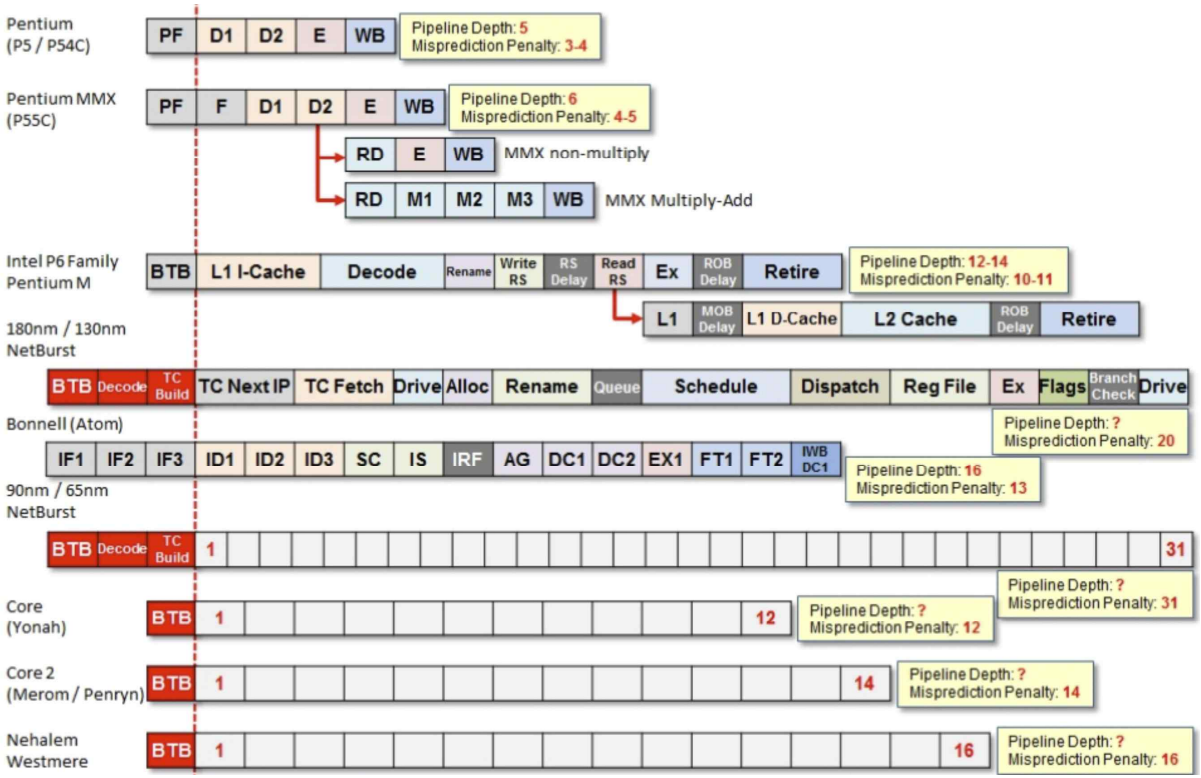
| Address | Normal Branch | Delayed Branch | Optimized Delayed Branch |
|---|---|---|---|
| 100 | LOAD X, rA | LOAD X, rA | LOAD X, rA |
| 101 | ADD 1, rA | ADD 1, rA | JUMP 105 |
| 102 | JUMP 105 | JUMP 106 | ADD 1, rA |
| 103 | ADD rA, rB | NOP | ADD rA, rB |
| 104 | SUB rC, rB | ADD rA, rB | SUB rC, rB |
| 105 | STORE rA, Z | SUB rC, rB | STORE rA, Z |
| 106 | | STORE rA, Z | |

## 3.3.6 분기예측에서의 절충

- **분기처리에 있어서 성능향상과 비용과의 절충(Trade-off)필요.**

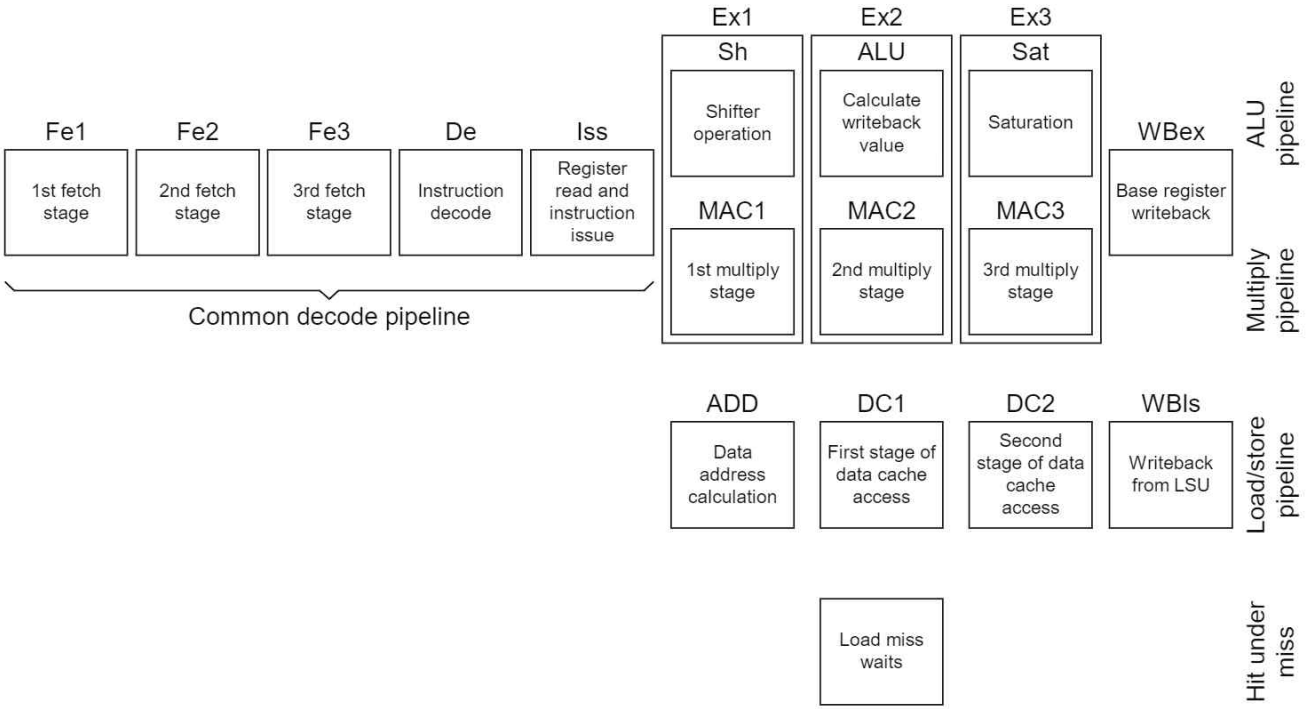
- **파이프라인 비용**
  - 하드웨어 비용
  - 파이프라인 스테이지 (stage) 사이의 시간지연
  - 파이프라인 제거 (flushing) 비용

# 4. x86 프로세서 파이프라인



# 5. ARM 프로세서 파이프라인

- **ARM1156T2-S 프로세서 파이프라인**



# 명령어 세트 특성

# 1. 명령어 실행 특성

- **명령어 실행 특성 분석**
  - HPL (high-level programming language)를 사용한 상용 프로그램들을 분석하여 프로그램에 사용된 명령어들의 실행 특성을 분석할 필요가 있다.
  - 새로운 프로세서, Micro-architecture 또는 명령어 세트 설계에 분석 결과 반영

- **명령어 실행 특성분석 요소**
  - 명령어 동작 분석
  - 명령어에서 사용하는 오퍼랜드 분석
  - 명령어 실행 순서 분석

# 1.1 명령어 동작분석

- **명령어 동작 분석**
  - 다양한 종류의 프로그램의 분석결과 나타난 HPL 기능별 발생빈도
  - 프로세서가 실행하는 기능과 메모리와의 연동을 결정

| | Dynamic Occurrence | | Machine-Instruction Weighted | | Memory-Reference Weighted | |
|---|---|---|---|---|---|---|
| | Pascal | C | Pascal | C | Pascal | C |
| ASSIGN | 45% | 38% | 13% | 13% | 14% | 15% |
| LOOP | 5% | 3% | 42% | 32% | 33% | 26% |
| CALL | 15% | 12% | 31% | 33% | 44% | 45% |
| IF | 29% | 43% | 11% | 21% | 7% | 13% |
| GOTO | – | 3% | – | – | – | – |
| OTHER | 6% | 1% | 3% | 1% | 2% | 1% |

# 1.2 오퍼랜드 분석

- **명령어에서 사용하는 오퍼랜드 분석**
  - 메모리 조직과 어드레싱 모드를 결정

| | Pascal | C | Average |
|---|---|---|---|
| Integer Constant | 16% | 23% | 20% |
| Scalar Variable | 58% | 53% | 55% |
| Array/Structure | 26% | 24% | 25% |

# 1.3 실행순서 분석

- **명령어 실행 순서 분석**
  - 프로그램 실행 흐름과 파이프라인 구조를 결정
  - Tanenbaum 연구결과
    - 98% 함수가 6개 이하의 파라미터를 사용, 92% 함수가 6개 이하의 지역변수를 사용한다.
  - Berkeley RISC team report

| Percentage of Executed Procedure Calls With | Compiler, Interpreter, and Typesetter | Small Nonnumeric Programs |
|---|---|---|
| >3 arguments | 0–7% | 0–5% |
| >5 arguments | 0–3% | 0% |
| >8 words of arguments and local scalars | 1–20% | 0–6% |
| >12 words of arguments and local scalars | 1–6% | 0–3% |

## 2. 프로그램 분석결과 적용

- **HPL과 유사한 명령어 세트 구조를 설계하기 보다, 가장 많이 사용하고, 가장 시간이 많이 소요되는 HPL 기능을 최적화하는 것이 더 효과적.**

- **프로그램 실행 성능 향상 대책**
  - 오퍼랜드 참조 최적화
    - 많은 수의 레지스터 사용
    - 레지스터 사용을 최적화하기 위해 컴파일러 사용
  - 명령어 파이프 라인 설계 개선
    - 분기 예측 강화
  - 명령어 실행 비용 예측이 가능해야 한다.
    - 실행시간, 코드 크기, 소비전력 비용 예측이 가능해야 한다.

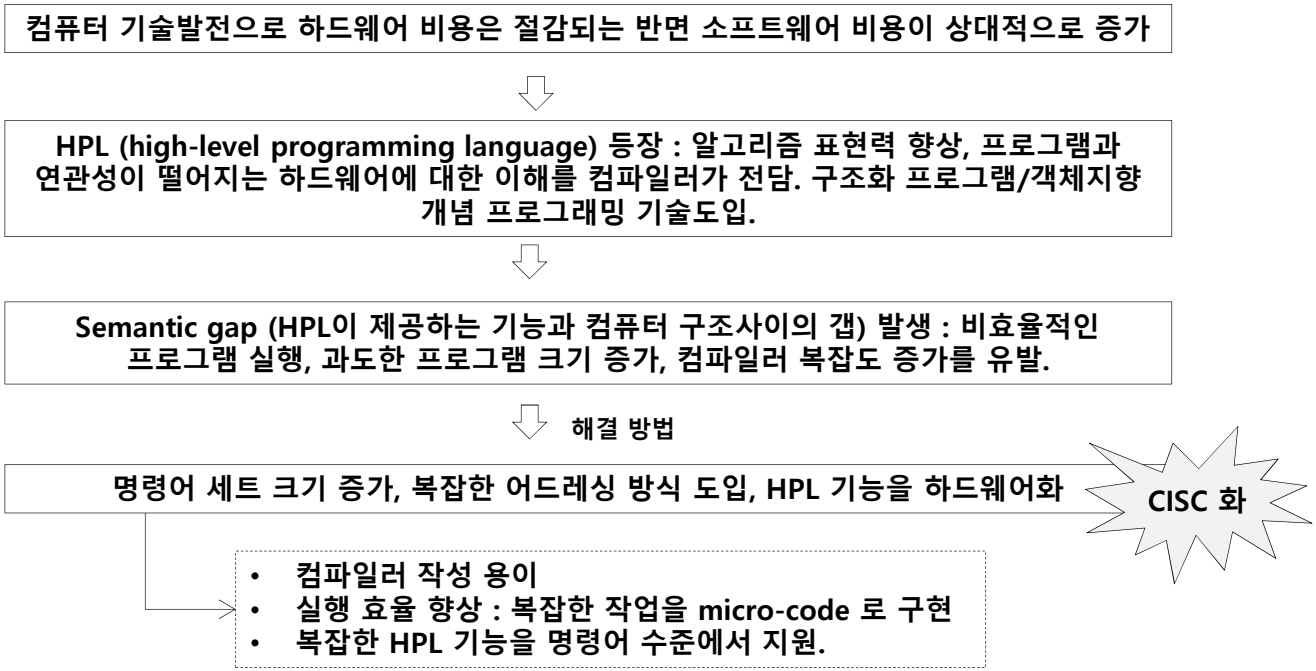
# 3. 명령어 세트 구조

- **프로그램 분석 결과를 반영하여 새로운 명령어 세트 설계**

- **명령어 세트 구조**
  - CISC
  - RISC
  - 명령어의 병렬 처리
    - VLIW
    - Superscalar
    - EPIC

# CISC

# 1. CISC 등장 배경


컴퓨터 기술발전으로 하드웨어 비용은 절감되는 반면 소프트웨어 비용이 상대적으로 증가
HPL (high-level programming language) 등장 : 알고리즘 표현력 향상, 프로그램과 연관성이 떨어지는 하드웨어에 대한 이해를 컴파일러가 전담. 구조화 프로그램/객체지향 개념 프로그래밍 기술도입.
Semantic gap (HPL이 제공하는 기능과 컴퓨터 구조사이의 갭) 발생 : 비효율적인 프로그램 실행, 과도한 프로그램 크기 증가, 컴파일러 복잡도 증가를 유발.
해결 방법
명령어 세트 크기 증가, 복잡한 어드레싱 방식 도입, HPL 기능을 하드웨어화
CISC 화
• 컴파일러 작성 용이
• 실행 효율 향상 : 복잡한 작업을 micro-code 로 구현
• 복잡한 HPL 기능을 명령어 수준에서 지원.

## 2. CISC 특성

- **CISC 프로세서 특성**
  - 하나의 명령어에서 수행하는 작업이 많다.
  - 다양한 어드레싱 모드를 사용.
  - 명령어의 길이가 가변적이다.
    - 명령어 해석이 복잡
  - 풍부한 명령어 세트
    - 많은 수의 명령어
    - 복잡한 명령어
  - 프로그램 길이가 짧다.

## 2. CISC 특성

- **CISC로 가는 이유**
  - 컴파일러를 간단하게 구현할 수 있다.
  - 프로그램 크기가 줄고, 실행속도가 빨라지게 된다.

- **CISC에 대한 반론**
  - 간단한 컴파일러
    - 복잡한 명령어는 사용하기 힘들다.
    - 최적화가 오히려 어렵다.
  - 프로그램 크기절약 + 실행속도 향상 ???
    - CISC 프로그램 크기가 RISC 프로그램보다 작다고 보장하기 힘들다.
    - 긴 명령어가 여러 개의 짧은 명령어 실행보다 빨리 실행된다고 보장하기 어렵다.

## 2. CISC 특성

- **대표적 CISC 프로세서**
  - x86 프로세서
    - 어드레싱 모드 : 12 가지
    - 명령어 길이 : 1~12 바이트
  - Motorla 680X0
    - 어드레싱 모드 : 18 가지
    - 명령어 길이 : 2~10 바이트

# RISC

# 1. RISC

- **Reduced Instruction Set Computer**
  - 간단한 명령어 세트 구조를 가진 컴퓨터

- **RISC 프로세서의 주요 특성**
  - 간단한 명령어 구조
    - 대부분 고정길이 명령어 사용.
    - 간단한 어드레싱 모드 및 명령어 포맷 사용.
    - 동작 클럭당 1개 명령어 실행가능.
  - 명령어 세트에 속한 명령어 수가 상대적으로 적음.
  - 비교적 많은 레지스터 (GPR) 사용
    - Load-Store 구조 : 모든 연산은 레지스터를 사용.
    - 효율적인 레지스터 사용을 위해 컴파일러 역할 강조.
  - 명령어 실행 파이프라인 최적화 강화
  - Harvard 메모리 구조 : 명령어와 데이터 전송경로 분리
  - 간단한 명령어 decoder 회로.
    - Micro-code 대신 hardwired 회로 사용.
  - 컴파일러 최적화 용이
    - Primitive 명령어에 대한 다양한 최적화 기술적용 가능.

# 1. RISC

- **대표적 RISC 프로세서**
  - ARM
    - ARM Holdings
    - ARM2(The first ARM)@1985, ARMv8(64-bit)@2011
  - MIPS
    - MIPS Technologies
    - R2000 (The first MIPS) @1985, R4000 (64-bit)@1991
  - SPARC
    - Oracle (Sun Microsystems)
    - SPARC V7(The first SPARC)@1986, SPARC V9(64-bit)@1993

# 2. 대표적 프로세서 비교

| Processor | Number of instruction sizes | Max instruction size in bytes | Number of addressing modes | Indirect addressing | Load/store combined with arithmetic | Max number of memory operands | Unaligned addressing allowed | Max number of MMU uses | Number of bits for integer register specifier | Number of bits for FP register specifier |
|---|---|---|---|---|---|---|---|---|---|---|
| AMD29000 | 1 | 4 | 1 | no | no | 1 | no | 1 | 8 | 3[a] |
| MIPS R2000 | 1 | 4 | 1 | no | no | 1 | no | 1 | 5 | 4 |
| SPARC | 1 | 4 | 2 | no | no | 1 | no | 1 | 5 | 4 |
| MC88000 | 1 | 4 | 3 | no | no | 1 | no | 1 | 5 | 4 |
| HP PA | 1 | 4 | 10[a] | no | no | 1 | no | 1 | 5 | 4 |
| IBM RT/PC | 2[a] | 4 | 1 | no | no | 1 | no | 1 | 4[a] | 3[a] |
| IBM RS/6000 | 1 | 4 | 4 | no | no | 1 | yes | 1 | 5 | 5 |
| Intel i860 | 1 | 4 | 4 | no | no | 1 | no | 1 | 5 | 4 |
| IBM 3090 | 4 | 8 | 2[b] | no[b] | yes | 2 | yes | 4 | 4 | 2 |
| Intel 80486 | 12 | 12 | 15 | no[b] | yes | 2 | yes | 4 | 3 | 3 |
| NSC 32016 | 21 | 21 | 23 | yes | yes | 2 | yes | 4 | 3 | 3 |
| MC68040 | 11 | 22 | 44 | yes | yes | 2 | yes | 8 | 4 | 3 |
| VAX | 56 | 56 | 22 | yes | yes | 6 | yes | 24 | 4 | 0 |
| Clipper | 4[a] | 8[a] | 9[a] | no | no | 1 | 0 | 2 | 4[a] | 3[a] |
| Intel 80960 | 2[a] | 8[a] | 9[a] | no | no | 1 | yes[a] | – | 5 | 3[a] |

# 3. RISC와 CISC 비교 방법

- **RISC와 CISC의 우열 판단 어려움.**
  - RISC와 CISC 특성을 절충한 프로세서 설계

- **비교 방법**
  - 정량적 비교
    - 프로그램 크기와 실행속도 비교
  - 정성적 비교
    - 고급 언어 지원 능력 및 칩 면적 비교

- **비교의 어려움**
  - 직접 비교 어려움
  - 성능 측정을 위한 테스트 프로그램 부재
  - 하드웨어와 컴파일러 영향을 분리해서 비교하기가 어려움.

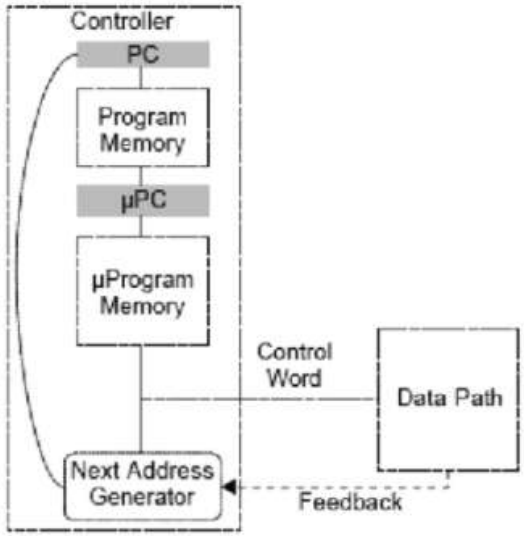
# 3. RISC와 CISC 비교

| CISC | RISC |
|---|---|
| 명령어 길이 가변 | 명령어 길이 고정 |
| 복잡한 어드레싱 모드 | 간단한 어드레싱 모드 |
| 적은 레지스터 사용 | 많은 레지스터 사용 |
| 메모리 오퍼랜드 허용 | 레지스터 오퍼랜드만 허용 |

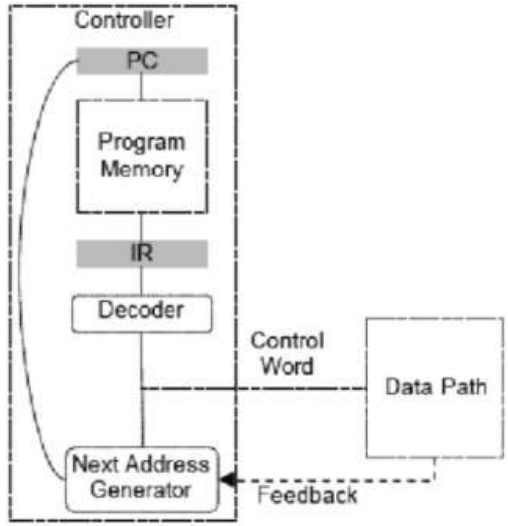
| CISC | RISC |
|---|---|
| 하드웨어 강조 | 소프트웨어 강조 |
| 다중 클럭을 사용하는 복잡한 명령어 | 단일 클럭을 사용하는 단순 명령어 |
| 대부분 명령어가 메모리 접근 가능 (Memory-to-Memory) | Load/Store 명령어만 메모리 접근 가능 (Register-to-Register) |
| 복잡한 명령어 디코더 필요 (Micro-programmed) | 간단한 명령어 디코더 필요(Hardwired) |
| 프로그램당 명령어 수가 적음. | 프로그램당 명령어 수가 많음. |

# 3. RISC와 CISC 비교

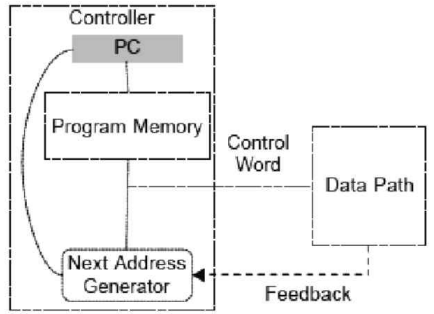
- **구조적 차이**
  - CISC : Micro-program 을 사용한 제어
  - RISC : Hardwired Logic을 사용한 제어




CISC



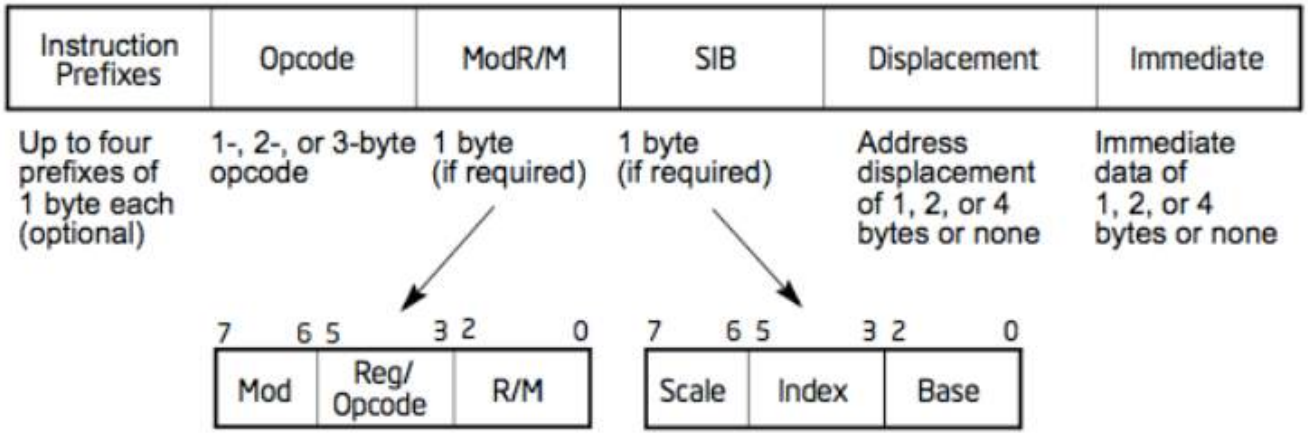

RISC




NISC ?
(No Instruction)

# 3. RISC와 CISC 비교

- **명령어 포맷 비교**
  - CISC : 복잡한 구조, 가변 길이
  - RISC : 간단한 구조, 고정 길이

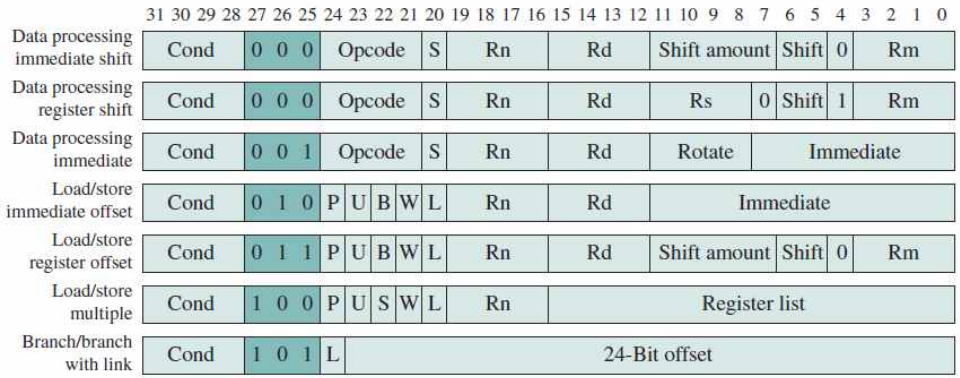



x86 Instruction Format

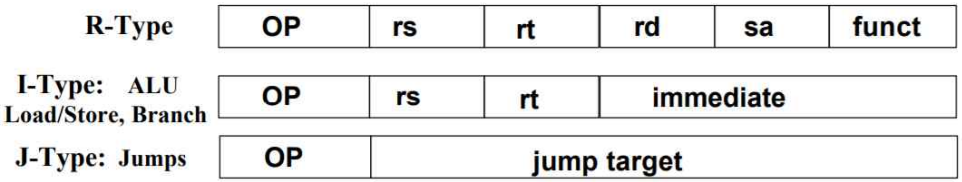
# 3. RISC와 CISC 비교

- 명령어 포맷 비교

ARM Instruction Format



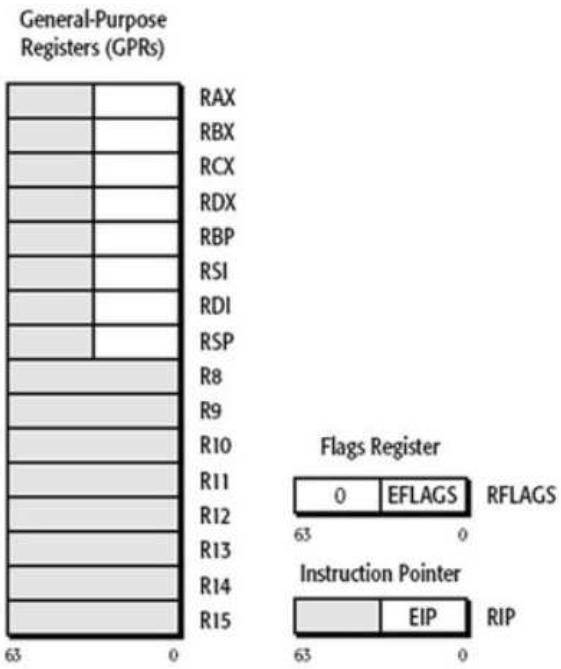

MIPS Instruction Format



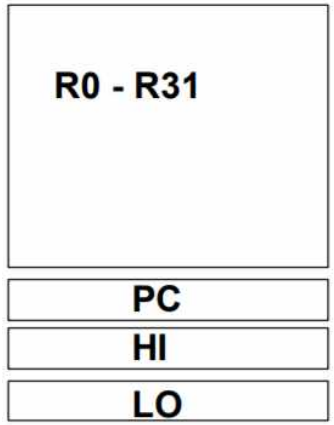
# 1. RISC

- 레지스터 비교

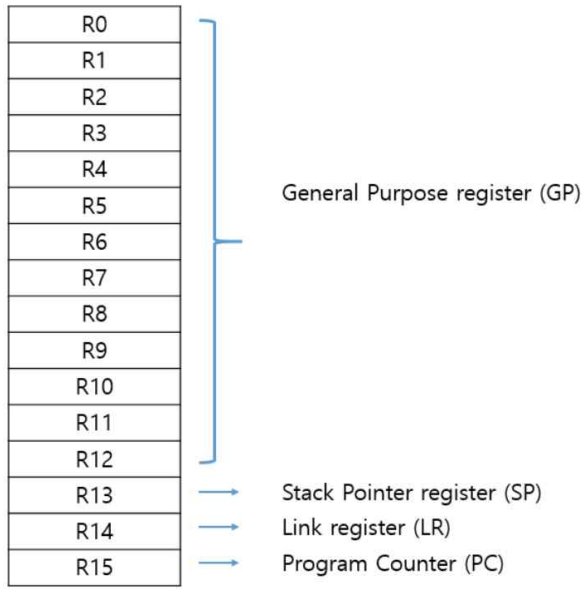
x86 Registers




MIPS Registers




ARM Registers



# 병렬처리 명령어

# 1. 명령어의 병렬처리

- **병렬처리를 위한 요구조건**
  - 명령어 사이의 상호 의존성(Dependency)를 검사하여, 서로 독립적인 명령어들을 동시에 처리.
  - 여러 개의  Functional Unit(ALU, FPU등)을 사용하여 동시에 연산 실행
  - 여러 개의 명령어의 동시 실행 시점 결정.

- **명령어 수준 병렬성(ILP : Instruction Level Parallelism)을 사용한 프로세서**
  - VLIW
  - Superscalar
  - EPIC

## 2. VLIW

- **VLIW(Very Long Instruction Word)**
  - 병렬로 실행할 수 있는 여러 개의 작업을 1개의 긴 명령어로 묶어서 사용.
    - VLIW 명령어의 길이는 CISC보다 길다.
  - 컴파일러를 통한 병렬처리 결정
    - 하드웨어 보다 더 넓은 범위에서 병렬성(Instruction-level Parallelism) 검사 가능.
    - 컴파일러는 프로그램 소스에 대한 정보를 병렬처리에 활용가능.
  - 컴파일러가 명령어들을 스케쥴링.
    - 컴파일러가 명령어들의 의존성을 검사하여 실행순서를 스케쥴링
    - 컴파일러가 Functional Unit 할당
  - 대용량 레지스터 파일 사용
    - Functional unit 들사이에서 공유.
  - 여러 개의 독립된 작업을 실행하는 RISC와 유사

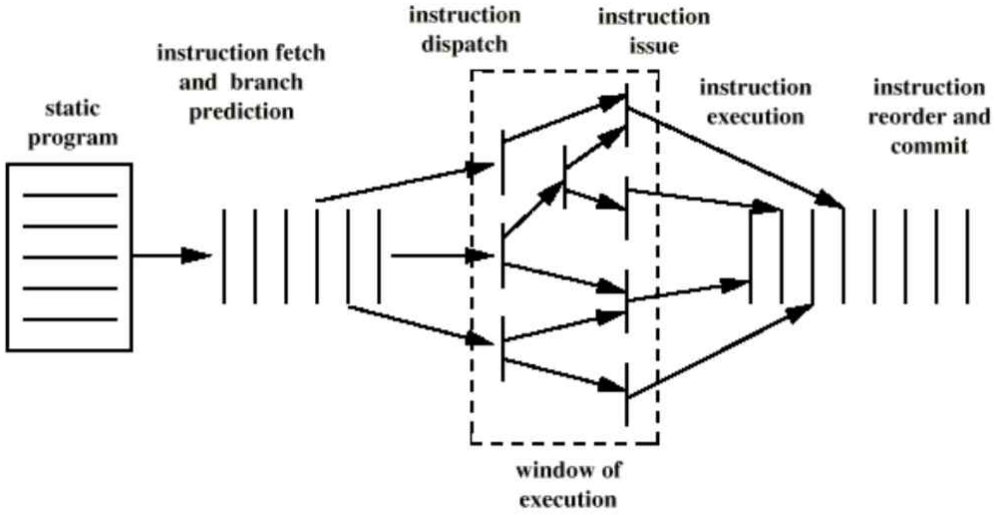
## 2. VLIW

- **VLIW 단점**
  - 호환성 유지가 어렵다.
  - Functional Unit 의 수와 latency 길이 차이에 따라 실행이 제한적임.
  - 스케쥴되지 않은 작업으로 인해 프로세서 전체가 멈출 수(Stall) 있음.

- **VLIW 프로세서**
  - DSP/멀티 미디어 응용 분야에 적합
  - IBM Yorktown, Intel iWarp, Chromatic, TI TMS320C6x

## 3. Superscalar

- 여러 개의 명령어를 동시에 인출하여 실행.
- 명령어들의 병렬 실행 여부는 하드웨어적으로 판단.
- CISC, RISC 명령어들을 병렬 처리할 수 있는 하드웨어 구조.
  - Superscalar CISC, Superscalar RISC



# 3. Superscalar 프로세서 사례

- **PowerPC**
  - 6개의 독립적 실행 유니트
    - Branch Execution Unit
    - Load/Store Unit
    - 3 개의 Integer Unit
    - Floating-point Unit
  - In-order issue
  - Register renaming
- **Pentium**
  - 3 개의 독립적 실행 유니트 사용
    - 2 개의 Integer Unit
    - Floating point Unit
  - In-order Issue

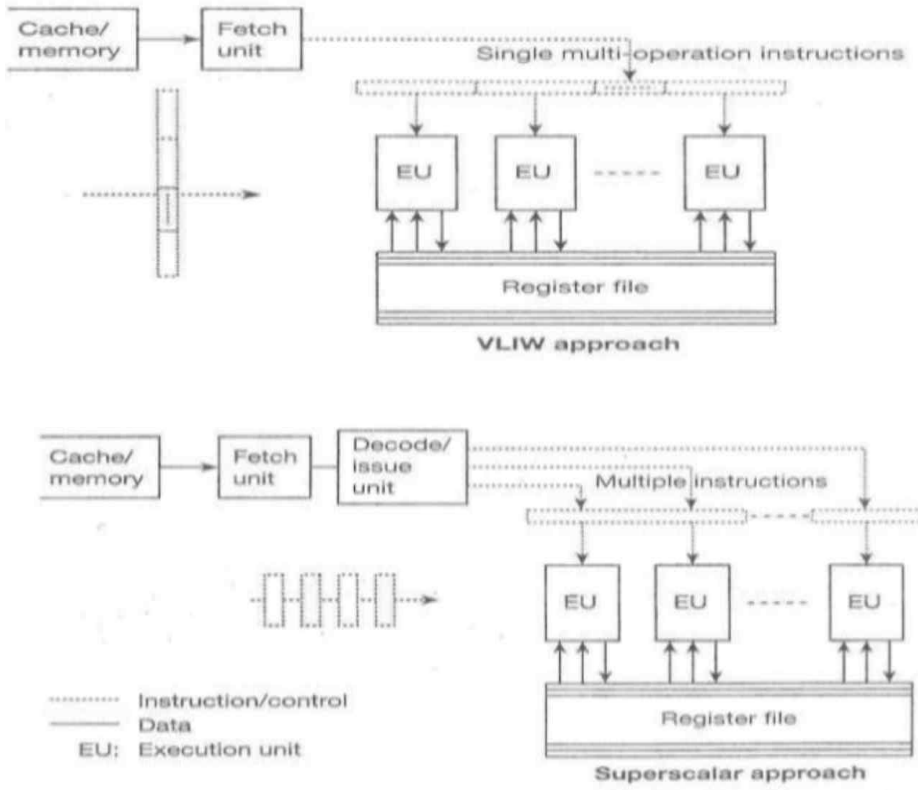
# 4. Superscalar와 VLIW 차이

- 특성 비교

| VLIW | Superscalar |
|---|---|
| 긴 명령어(보통 64~1024bit),<br>FU 수와 제어방식에 따라 명령어 길이 가변 | 일반 명령어(RISC, CISC) |
| 컴파일러에 의한 의존성 검사 및 명령어 실행 스케쥴링. | 하드웨어에 의한 의존성 검사 및 명령어 실행 스케쥴링. |
| 코드 밀도(Code Density) 저하. 비어있는 명령어 필드에 의한 메모리 공간 낭비 | 많은 레지스터 사용 |
| 하드웨어 복잡도 감소, 컴파일러 복잡도 증가 | |

# 4. Superscalar와 VLIW 차이

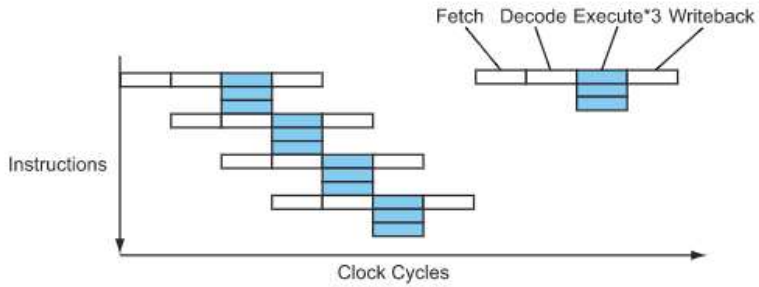
- 하드웨어 구성



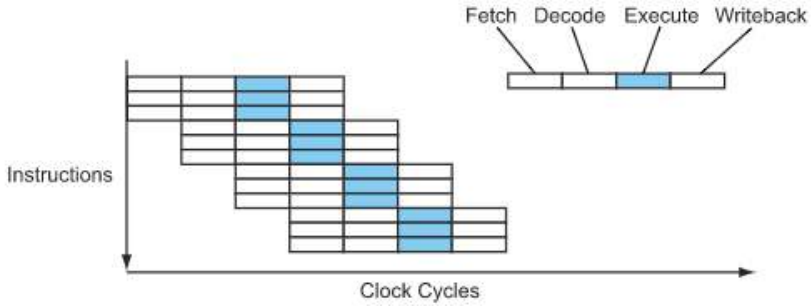
# 4. Superscalar와 VLIW 차이

- 명령어 실행 흐름

VLIW




Superscalar



# 5. EPIC

- **EPIC(Explicitly Parallel Instruction Computer)**
  - 64-bit 명령어 세트 : HP와 Intel 공동 설계
    - Intel Architecture-64 (IA-64) 에 사용 (Itanium 프로세서에 적용)

- **Superscalar 프로세서 장점을 채용한 VLIW**
  - 명령어들 사이의 병렬성은 컴파일러가 검사해서 하드웨어에 알려준다.

- **EPIC 특징**
  - 128 개의 레지스터 사용
  - Speculative loading, Branch Prediction 사용
  - 컴파일러에 의한 명령어의 병렬처리
  - 3개의 명령어를 1개의 128-bit Bundle로 인코딩.
    - 3 x 41-bit : 3개의 명령어를 41-bit로 인코딩
    - 5-bit Template Field : 명령어에 대한 부가 정보

# 5. EPIC

- **명령어의 병렬 실행에 필요한 작업**
  - 명령어 Grouping
    - 명령어 사이의 의존성을 검사하여, 병렬로 실행할 수 있는 명령어들을 Grouping.
  - FU 할당
    - 명령어들을  Functional Unit(ALU, FPU등)에 할당
  - 명령어 실행시점 결정
    - 명령어를 언제 실행할지(Initiation)를 결정

| | VLIW | Superscalar | EPIC |
|---|---|---|---|
| **Instruction Grouping** | Compiler | Hardware | Compiler |
| **FU Assignment** | Compiler | Hardware | Hardware |
| **Start Execution** | Compiler | Hardware | Hardware |

# end