2. 마이크로프로세서 내부 구조

2.1 ALU (산술 논리 연산장치)

🧠 정의

ALU란 마이크로프로세서 내부에서 수행되는 **산술 연산(Arithmetic)**과 **논리 연산(Logic)**을 처리하는 **디지털 회로**다.
모든 수치 연산, 조건 판단, 비트 처리 명령은 결국 ALU를 통해 실행된다.

> ALU는 CPU가 실행하는 연산 명령의 "실행 엔진"이다.

🔧 주요 기능

연산 종류	세부 기능	예
산술 연산	덧셈, 뺄셈, 곱셈, 나눗셈	`ADD R1, R2` / `SUB A, B`
비트 연산	AND, OR, XOR, NOT	`AND R1, R2` / `XOR A, B`
시프트 연산	좌우 시프트, 로테이트	`SHL R1`, `ROR A`
비교 연산	두 값의 크기 비교	`CMP R1, R2` → 플래그 설정
증감 연산	증가, 감소	`INC A`, `DEC B`

🧩 ALU 내부 구조 예시

```
          +----------------------+
Input A → |                      |
Input B → |     Arithmetic       | → ADD, SUB, MUL, DIV
          |      Block           |
          +----------------------+
                         ↓
          +----------------------+
          |      Logic Block     | → AND, OR, XOR, NOT
          +----------------------+
                         ↓
          +----------------------+
          |    Comparator Block  | → CMP, 조건 판단
          +----------------------+
                         ↓
        Output → 연산 결과
```

ALU는 위 세 블록(산술, 논리, 비교)을 **선택 신호(Opcode)에 따라 활성화**해 결과를 출력함.

⚙️ 동작 흐름 요약

1. **명령어 해석**: `ADD A, B` 명령어가 CPU 내 Control Unit에 의해 해석됨
2. **오퍼랜드 입력**: 레지스터 A와 B의 값이 ALU로 전달됨
3. **연산 선택**: Control Unit이 ALU에 "덧셈" 신호를 보냄
4. **연산 수행**: ALU가 `A + B`를 수행하고 결과를 출력
5. **결과 저장**: 결과는 다시 레지스터나 메모리에 저장됨
6. **플래그 설정**: 연산 결과에 따라 Zero, Carry, Overflow 플래그 등을 설정

🧪 ALU 연산 결과와 상태 플래그

ALU는 연산 결과뿐 아니라 **조건 판단을 위한 플래그(Flags)**도 설정한다:

플래그 이름	설명
Zero (Z)	결과가 0이면 1
Carry (C)	자리올림/내림 발생 여부
Sign (S)	결과의 부호
Overflow (V)	표현 범위 초과 여부
Parity (P)	결과의 1의 개수 짝수 여부

이 플래그들은 조건 분기 명령(예: `JZ`, `JC`, `JNE`) 등에 사용됨.

🧠 ALU와 FPU의 차이

항목	ALU	FPU
이름	Arithmetic Logic Unit	Floating Point Unit
처리 데이터	정수 중심	부동소수점 중심
연산 범위	기본 산술/논리 연산	실수 계산 (소수점, 지수 등)
포함 여부	거의 모든 CPU에 내장	일부 고성능 CPU에만 포함되거나 외장

📱 실제 예시 (어셈블리)

```
MOV R1, #5        ; R1 ← 5
MOV R2, #3        ; R2 ← 3
ADD R3, R1, R2    ; R3 ← R1 + R2 → 8
CMP R3, #8        ; 비교 → Zero 플래그 = 1
```

→ 이 모든 과정은 ALU에서 수행됨

🧬 현대 ALU의 확장

현대 CPU에서는 ALU가 단순 회로가 아니라 다음과 같은 **확장된 유닛**으로 구성됨:

- **멀티 ALU 파이프라인** (슈퍼스칼라 구조)
- **벡터 ALU (SIMD)**: SSE, AVX 명령어 처리
- **병렬 ALU 연산**: 멀티코어, 다중 스레드 지원

📌 요약 정리

- **ALU는 산술 연산, 논리 연산, 조건 판단을 담당하는 프로세서의 핵심 회로**다.
- ALU는 연산 결과뿐 아니라 플래그를 통해 **조건 분기와 상태 판단의 기준**도 제공한다.
- 현대 CPU에서는 다수의 ALU를 병렬로 배치하거나 벡터 ALU로 확장해 연산 성능을 극대화한다.

2.2 레지스터 파일

🧠 기본 개념

레지스터 파일(Register File)은 CPU 내부에서 데이터를 임시로 저장하고 고속으로 접근하기 위한 **레지스터들의 집합**이자, 이를 **읽고 쓰는 회로 인터페이스**까지 포함한 구조를 말한다.

> 간단히 말하면, “CPU의 내부 메모리 뭉치”이자 “ALU의 가장 가까운 데이터 창고”다.

🔧 구성 요소

구성 요소	설명
N개의 레지스터	8, 16, 32, 64개 등. 보통 32비트 또는 64비트 폭
읽기 포트 (Read Ports)	동시에 2개 이상의 레지스터 값을 읽는 회로
쓰기 포트 (Write Port)	연산 결과를 특정 레지스터에 기록하는 회로
디코더 (Decoder)	레지스터 주소를 해석해 특정 레지스터 선택
제어 신호 (Control Signals)	쓰기 활성화, 읽기 선택 등을 제어

📐 구조도 예시

```
          +-----------------------------------+
          |          Register File            |
          |  +---------+   +---------+        |
          |  |   R0    |   |  Read A | ← 읽기 포트 1
          |  +---------+   +---------+        |
          |  |   R1    |   |  Read B | ← 읽기 포트 2
          |  +---------+   +---------+        |
          |  |   R2    |   |  Write  | ← 쓰기 포트
          |  +---------+   +---------+        |
          |  |  ...    |                       |
          |  +---------+                       |
          +-----------------------------------+
```

🔢 일반적인 레지스터 구성 (RISC vs CISC)

구조	일반적 레지스터 수	특징
RISC (예: ARM, RISC-V)	32~128개	대부분 범용(GPR), 이름도 R0, R1, ...
CISC (예: x86)	8~16개	용도에 따라 이름 분화 (EAX, EBX, ESP 등)

🧠 레지스터의 종류

종류	설명	예시
GPR (General Purpose Register)	범용 연산 및 데이터 저장용	R0~R31, EAX
SPR (Special Purpose Register)	프로그램 제어용	PC, SP, LR
FPR (Floating Point Register)	부동소수점 연산용	F0~F31
Vector Register	SIMD 연산용	XMM0, YMM0
Status Register (Flags)	연산 상태 저장	ZF, CF, OF 등

🧪 사용 예시 (어셈블리)

```
MOV R1, #5        ; R1 ← 5
MOV R2, #10       ; R2 ← 10
ADD R3, R1, R2    ; R3 ← R1 + R2 → 15
```

→ 위의 모든 연산은 **레지스터 파일 내부 값**만을 조작하며, 메모리는 건드리지 않음
→ 그래서 연산이 **메모리보다 훨씬 빠르게 수행**됨

⏱️ 시간과 성능 측면

저장소	접근 속도	용량	접근 방식
레지스터	**가장 빠름 (한 클럭 이내)**	작음 (KB 이하)	CPU 내부 직접
캐시	빠름	수십~수백 KB	계층적 접근
메모리 (RAM)	느림	수 GB	외부 버스
디스크	매우 느림	TB	파일 시스템

> 그래서 **컴파일러는 최대한 레지스터만 사용**하려고 하고, 부족하면 그때 메모리를 씀

💬 레지스터 할당 및 컴파일러 역할

컴파일러는 코드 실행 시:

1. **자주 쓰이는 변수**를 레지스터에 유지
2. 레지스터가 부족하면 **spill**이라고 해서 메모리에 저장함
3. 고급 최적화 기법인 **레지스터 할당 알고리즘 (Graph Coloring 등)** 사용

🧠 현대적 확장

구조	설명
Scoreboarding	연산 중 레지스터 사용 충돌 관리
Register Renaming	가상 레지스터 이름으로 충돌 방지
Out-of-order Execution	연산 순서 최적화 시 레지스터 추적 필요
SIMD 레지스터	병렬 데이터 연산 전용 (128~512비트 단위)

📌 요약 정리

- **레지스터 파일**은 CPU 내부의 초고속 데이터 저장 공간으로, GPR, FPR, Flags 등 다양한 레지스터가 포함됨
- 모든 연산은 메모리 대신 레지스터를 통해 빠르게 이루어지며, **명령어당 수 ns 내 처리 가능**
- 현대 CPU에서는 레지스터의 수와 구조가 **성능을 결정짓는 중요한 요소**이며, 컴파일러는 이를 최적화하기 위해 복잡한 알고리즘을 사용함

2.3 제어 유닛 (Control Unit)

🧠 정의

제어 유닛(Control Unit)은 CPU 내부에서 **명령어를 해석(Decode)**하고, 해당 명령에 따라 **CPU 내 각 부품(ALU, 레지스터, 메모리 등)**이 무엇을 해야 할지를 **제어 신호(Control Signals)**로 알려주는 디지털 회로다.

> 쉽게 말해, 제어 유닛은 CPU 안에서 "누가 언제 무엇을 할지"를 지시하는 **작전 지휘관**이다.

📦 주요 역할

기능	설명
명령어 인출(Fetch)	메모리로부터 명령어를 읽어옴
명령어 해석(Decode)	명령어의 의미를 분석하고 필요한 동작 판단
제어 신호 생성	ALU, 레지스터, 버스 등에 신호를 보내 동작을 지시
클럭 동기화	모든 동작을 클럭 사이클에 맞춰 순서대로 실행
상태 플래그 처리	조건 분기 등에서 플래그에 따라 흐름 결정

🔄 명령어 실행 흐름

명령어는 보통 **4단계 (또는 5단계)** 흐름으로 실행돼. 제어 유닛은 이 전체를 조율한다:

1. **Fetch**: PC에 저장된 주소에서 명령어를 메모리에서 읽어옴
2. **Decode**: 명령어 형식을 분석해 어떤 연산인지 파악
3. **Execute**: ALU나 로직 유닛에게 연산을 지시
4. **Memory Access (선택적)**: 필요한 경우 메모리 접근
5. **Write Back**: 결과를 레지스터에 저장

→ 이 전체 과정을 **제어 유닛이 클럭 단위로 단계별로 관리**한다.

🔄 명령어 실행 흐름

명령어는 보통 **4단계 (또는 5단계)** 흐름으로 실행돼. 제어 유닛은 이 전체를 조율한다:

1. **Fetch**: PC에 저장된 주소에서 명령어를 메모리에서 읽어옴
2. **Decode**: 명령어 형식을 분석해 어떤 연산인지 파악
3. **Execute**: ALU나 로직 유닛에게 연산을 지시
4. **Memory Access (선택적)**: 필요한 경우 메모리 접근
5. **Write Back**: 결과를 레지스터에 저장

→ 이 전체 과정을 **제어 유닛이 클럭 단위로 단계별로 관리**한다.

🧩 제어 유닛 구현 방식

✅ 하드와이어드 제어 (Hardwired Control)

- 논리 회로(AND, OR, NOT 등)로 제어 신호를 **물리적으로 구성**
- 빠르고 간단하지만 **유연성 없음**
- RISC 구조에서 자주 사용됨

✅ 마이크로프로그래밍 제어 (Microprogrammed Control)

- ROM에 제어 시퀀스를 저장한 형태
- 각 명령어마다 일련의 제어 신호들이 **마이크로명령어(microinstruction)**로 구성됨
- 느리지만 **유연하고 확장 가능**
- CISC 구조에서 많이 사용됨 (예: Intel x86)

📊 제어 신호 예시

명령어	제어 신호
`ADD R1, R2`	`Reg[R1] → ALU A, Reg[R2] → ALU B, ALU = ADD, 결과 → Reg[R1]`
`LOAD R1, [1000]`	`주소 1000 → MAR, Mem[MAR] → MDR, MDR → Reg[R1]`
`JZ 2000`	`ZF=1이면 PC ← 2000`

📐 간단 구조도

```
         +-------------------------+
         |     Control Unit        |
         | +---------------------+ |
         | |  Instruction Reg    | |
         | +---------------------+ |
         | |    Decoder          | |
         | +---------------------+ |
         | | Control Signal Gen  | |
         | +---------------------+ |
         +-----------↓-------------+
                     ↓
        +-----------------------------+
        |      ALU / Reg / BUS        | ← 이들에게 신호 전달
        +-----------------------------+
```

🧠 현대적 기능 확장

기능	설명
분기 예측기 (Branch Predictor)	제어 흐름 예측 (파이프라인 효율 ↑)
파이프라인 컨트롤러	명령어 간 충돌 관리
Out-of-Order 스케줄러	실행 순서 동적 조정
동적 명령어 발행기	동시에 여러 유닛에 명령 전달

📌 요약 정리

- **제어 유닛은 명령어를 해석하고**, CPU의 모든 부품이 **언제, 무엇을, 어떻게 할지를 제어 신호로 지시**한다.
- 구현 방식은 하드와이어드(속도)와 마이크로프로그래밍(유연성)으로 나뉜다.
- 현대 CPU는 제어 유닛의 역할을 넘어서 **분기 예측, 명령어 재정렬, 동시 실행 제어까지 확장**하고 있다.

2.4 클럭과 동기화 회로

🧠 클럭(Clock)이란?

클럭(Clock)은 마이크로프로세서 내부 회로들이 **동기화된 방식으로 동작하도록 주기적인 타이밍 신호(펄스)를 생성**하는 신호다.

> 쉽게 말하면, 클럭은 CPU가 **'지금 연산하라!'**, **'지금 메모리 읽어라!'** 라고 외치는 메트로놈이자 시간 관리자야.

⏱️ 클럭의 핵심 역할

역할	설명
명령어 실행 속도 제어	클럭 주파수(Hz)가 높을수록 연산 속도가 빨라짐
회로 동기화	ALU, 레지스터, 메모리 등 모든 모듈이 **동일한 타이밍 기준**으로 작동
파이프라인 분할 기준	한 명령어를 여러 클럭 사이클로 분리해 처리
타이밍 기반 제어 신호 생성	제어 유닛과 동기적으로 동작하도록 구성

🔁 클럭의 구성 요소

구성 요소	역할
클럭 소스(발진기)	기본 진동 생성 (크리스털, PLL 등)
클럭 발생기 (Clock Generator)	진동을 디지털 신호(펄스)로 변환
클럭 분배기 (Clock Distributor)	CPU 내부 서브 모듈에 클럭 신호 전달
PLL (Phase Locked Loop)	클럭 주파수 조정 및 안정화
게이팅 회로 (Clock Gating)	전력 절약을 위한 클럭 차단 제어

📊 클럭 주파수와 성능

항목	설명
클럭 주파수 (GHz)	1초에 몇 번 신호가 발생하는가
1 클럭 사이클	한 번의 명령 단계가 수행되는 시간
CPI (Clock per Instruction)	한 명령을 처리하는 데 걸리는 클럭 수
IPC (Instruction per Cycle)	1클럭에 처리 가능한 명령 수 (슈퍼스칼라 구조 등)

예시:

- 1GHz CPU = 1초에 10억 번의 펄스
- 명령어당 1클럭 걸린다면 → 초당 10억 개 명령어 실행 가능

📐 동기 회로(Synchronization Circuit)

> 클럭에 맞춰 서로 다른 구성 요소가 데이터 전송, 제어를 수행할 수 있게 **시점을 일치시키는 회로**

주요 동기화 기법:

기법	설명
플립플롭(D Flip-Flop)	동기식 회로의 기본, 클럭 엣지에 따라 데이터 저장
동기 버스(Synchronous Bus)	모든 장치가 **공통 클럭**에 따라 통신
동기 카운터	시간 기반 상태 전이 관리
FIFO + 핸드셰이킹	서로 다른 클럭 도메인 간 인터페이스 구현

🧩 클럭 관련 회로 예시

```
        +--------------------+
        | Oscillator (크리스털) |
        +--------------------+
                  ↓
        +--------------------+
        |  PLL (주파수 안정화) |
        +--------------------+
                  ↓
        +--------------------+
        | Clock Generator    |
        +--------------------+
                  ↓
        +----------------------------+
        |  ALU / Register / Control  |
        |   ← 클럭 신호 분배           |
        +----------------------------+
```

⚡ 고급 클럭 제어 기술

기술	설명
Dynamic Frequency Scaling (DFS)	필요에 따라 클럭 주파수 조절
Dynamic Voltage and Frequency Scaling (DVFS)	전압과 클럭을 동시에 제어해 전력 최적화
Clock Gating	특정 회로 블록에 클럭 공급 중단 (전력 절감)
Asynchronous Design	클럭 없이 동작하는 회로 설계 (최근 연구 영역)

🔐 클럭 관련 보안 고려

- 클럭 주파수를 외부에서 조작하면 타이밍 기반 공격 가능성 있음 (예: **클럭 글리치 공격**)
- 보안 SoC에서는 **클럭 무결성 보호 회로**를 내장하기도 함

📌 요약 정리

- **클럭은 CPU의 모든 동작을 시간 기준에 따라 동기화시키는 신호**다.
- 클럭 회로는 **발진기 → PLL → 분배기 → 모듈**의 흐름으로 구성된다.
- 클럭 주파수는 성능, 전력, 발열, 타이밍의 핵심이며, 클럭 동기화 회로는 각 모듈이 동시에 정확히 작동하게 해준다.
- 현대 시스템은 **전력 효율과 멀티코어 타이밍 문제**를 해결하기 위해 매우 정교한 클럭 제어 기술을 사용한다.

2.5 파이프라이닝 구조

🧠 기본 개념

파이프라이닝(Pipelining)은 마치 공장에서 **제품을 여러 작업 단계로 나눠서 동시에 처리**하듯,
CPU도 **하나의 명령어를 여러 단계로 나눠 각 단계가 병렬로 작동하도록** 만든 구조이다.

> 즉, 명령어를 "한 줄씩 완전히 처리하는 것"이 아니라, "여러 명령을 동시에 겹쳐서 처리"함으로써 처리량을 높이는 방식이다.

🧩 비유로 이해하기

- 📦 공장 조립라인:
  - 1단계: 부품 자르기
  - 2단계: 부품 조립
  - 3단계: 품질 검사
  - 4단계: 포장

→ 각 작업자가 동시에 다른 제품을 작업 중이라면 **1개 제품 처리 속도는 같아도 전체 생산량은 증가**함.

📐 기본 파이프라인 단계 (5단계 RISC 구조 기준)

단계	약어	설명
1️⃣ 명령어 인출	IF (Instruction Fetch)	메모리에서 명령어를 읽어옴
2️⃣ 명령어 해석	ID (Instruction Decode)	명령어를 해석하고 오퍼랜드 결정
3️⃣ 연산 수행	EX (Execute)	ALU 연산 or 주소 계산 수행
4️⃣ 메모리 접근	MEM (Memory Access)	메모리에서 데이터 읽기/쓰기
5️⃣ 결과 저장	WB (Write Back)	연산 결과를 레지스터에 저장

📜 명령어 병렬 실행 예시

사이클	IF	ID	EX	MEM	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6		I5	I4	I3	I2
7			I5	I4	I3
8				I5	I4
9					I5

→ 명령어 하나당 실행 시간이 줄어든 건 아니지만, **전체 처리량(Throughput)이 5배 향상**됨.

📊 성능 공식

```
파이프라인 성능 향상 ≈ 파이프라인 단계 수만큼 향상 가능
```

단, 이론상 성능이고 현실에선 **해저드(Hazard)** 때문에 제한됨.

❗ 파이프라이닝의 문제점: 해저드(Hazards)

1. ⛔ 구조적 해저드 (Structural Hazard)

- 동일한 하드웨어 자원을 여러 명령어가 동시에 사용하려고 할 때 발생
  예: 한 번에 하나의 메모리만 접근 가능한데, IF와 MEM이 동시에 요구

2. ⛔ 데이터 해저드 (Data Hazard)

- 명령어 간의 **데이터 의존성**이 있을 때 발생
  예: `ADD R1, R2, R3` → `SUB R4, R1, R5`
  → 두 번째 명령이 첫 번째 결과를 기다려야 함

해결:

- **Forwarding (데이터 전달)**
- **Stall (파이프라인 정지)**
- **Reorder Buffer / Renaming**

3. ⛔ 제어 해저드 (Control Hazard)

- **분기 명령(Branch)**이 발생할 때 다음 명령어가 무엇인지 확정되지 않아서 생김

해결:

- **분기 예측기(Branch Predictor)**
- **Delay Slot 기법**
- **명령어 무효화(Flush)**

🧠 현대 CPU의 파이프라인 구조

구조	설명
슈퍼스칼라(Superscalar)	한 사이클에 여러 명령어를 병렬 실행
하이퍼파이프라인(Hyper-pipelining)	단계 수를 세분화하여 고클럭 가능
Out-of-Order Execution	명령어 순서를 재배치하여 병목 제거
Speculative Execution	예측 기반으로 명령어 미리 실행 (추후 무효화 가능)

🔋 파이프라인의 한계와 설계 트레이드오프

장점	단점
명령어 처리량 증가	회로 복잡도 증가
높은 클럭 속도 가능	해저드 발생 확률 증가
병렬 처리 구조 확장 가능	전력 소비 및 발열 ↑

📌 요약 정리

- **파이프라이닝**은 명령어를 단계별로 나누고, 각 단계가 병렬로 실행되게 하여 **처리량을 극대화**하는 기법이다.
- 명령어 하나의 실행 시간은 그대로지만, 동시에 여러 명령어를 겹쳐 실행함으로써 **전체 처리 성능(Throughput)**이 비약적으로 증가한다.
- 해저드(자원 충돌, 데이터 의존성, 분기 불확정성)로 인해 실제 성능은 이론적 최대보다 낮지만, 이를 해결하기 위한 다양한 기술이 발전해 왔다.

2.6 인터럽트 컨트롤러

🧠 기본 개념

인터럽트 컨트롤러는 외부 또는 내부에서 발생한 **인터럽트(Interrupt)** 요청을 수신하고,
CPU에 적절한 타이밍과 우선순위에 따라 **처리를 요청하는 중재기 역할**을 하는 회로 또는 장치다.

> 간단히 말하면, "CPU에게 **지금 긴급한 일이 생겼으니 일 좀 멈추고 처리해줘**!"라고 말해주는 신호 관리자이다.

⛳ 인터럽트란?

인터럽트(Interrupt)는 **프로세서가 실행 중인 명령어 흐름을 중단하고,**
지정된 인터럽트 처리 루틴(ISR, Interrupt Service Routine)으로 분기해 우선적인 일을 처리한 후
원래 프로그램 흐름으로 복귀하는 **비동기 이벤트 처리 메커니즘**이다.

🔧 인터럽트 컨트롤러의 주요 역할

역할	설명
인터럽트 요청 수신 (IRQ)	하드웨어/소프트웨어 인터럽트를 감지
우선순위 결정	다수 인터럽트 중 어떤 것을 먼저 처리할지 판단
마스킹(Masking)	특정 인터럽트의 일시 차단 여부 제어
벡터 주소 제공	ISR(인터럽트 서비스 루틴)의 시작 주소 제공
신호 동기화 및 중복 방지	여러 신호가 겹칠 경우 중복 방지 및 디버깅 지원

🧩 인터럽트 발생 구조

```
+------------+     +---------------------+
|  Timer     |     |     Keyboard        |
+------------+     +---------------------+
      ↓                   ↓
  [ 인터럽트 요청 (IRQ) ]
      ↓                   ↓
+---------------------------------------+
|         Interrupt Controller          |
|  - 우선순위 판별                        |
|  - 마스킹 필터                          |
|  - 벡터 주소 전달                       |
+---------------------------------------+
               ↓
         +-------------+
         |     CPU     | ← 현재 작업 중단, ISR 수행
         +-------------+
```

📌 인터럽트의 종류

1. 하드웨어 인터럽트

- 외부 장치 (예: 키보드, 타이머, 센서)가 CPU에 직접 신호를 보냄
- 예: `INT0`, `IRQ1`, `NMI(Non-maskable Interrupt)`

📎 2. 소프트웨어 인터럽트

- 명령어를 통해 명시적으로 인터럽트 발생
- 예: `INT 0x80` (리눅스 시스템 콜), `SVC` (ARM)

📎 3. 내부 인터럽트 (예외, 트랩)

- CPU 내부에서 오류나 특정 조건 발생 시
- 예: 0으로 나누기, 페이지 폴트, 오버플로우

📊 인터럽트 컨트롤러의 종류

종류	설명
PIC (Programmable Interrupt Controller)	고전적인 x86 구조에서 사용, 예: 8259A
APIC (Advanced PIC)	멀티코어 및 고급 시스템용
NVIC (Nested Vectored Interrupt Controller)	ARM Cortex-M에서 사용, 중첩 인터럽트 지원
GIC (Generic Interrupt Controller)	ARM Cortex-A에서 사용되는 고성능 컨트롤러
외장 컨트롤러	GPIO, 센서 입력 등을 위한 MCU용 인터럽트 입력 모듈

🧠 인터럽트 우선순위와 마스킹

개념	설명
우선순위(Priority)	높은 번호일수록 긴급하게 먼저 처리
마스킹(Masking)	특정 인터럽트는 무시하고, 필수 인터럽트만 받도록 설정 가능
NMI (Non-Maskable Interrupt)	절대 마스킹되지 않는 긴급 신호 (예: 하드웨어 오류)

🔁 인터럽트 처리 흐름 요약

1. 주변 장치에서 인터럽트 신호 발생 (IRQ)
2. 인터럽트 컨트롤러가 요청을 감지
3. CPU에게 인터럽트 처리 요청
4. CPU는 현재 명령을 마무리 후 **ISR(Interrupt Service Routine)**으로 분기
5. ISR 실행 후 복귀 명령(예: `IRET`)을 통해 원래 프로그램으로 복귀

🧬 고급 기능 (현대 인터럽트 컨트롤러)

기능	설명
중첩 인터럽트(Nested Interrupt)	처리 중 다른 더 높은 우선순위 인터럽트 수용
벡터 인터럽트(Vector Interrupt)	ISR 주소를 직접 전달받아 분기

기능	설명
레벨 vs 엣지 트리거	인터럽트를 **레벨(High/Low 유지)** 혹은 **엣지(변화 순간)**로 감지
멀티프로세서 분배 (Interrupt Affinity)	인터럽트를 특정 코어로 라우팅
DMA 연동	CPU 개입 없이 인터럽트 + 메모리 접근 동기화

📌 요약 정리

- **인터럽트 컨트롤러는 외부나 내부에서 발생한 이벤트를 감지하고, CPU가 적절히 이를 처리할 수 있게 중재하는 장치**다.
- 인터럽트는 프로그램 흐름을 일시 중단하고 비동기 이벤트를 처리하는 효율적인 메커니즘이다.
- 현대 시스템에서는 **다중 인터럽트, 우선순위, 마스킹, 벡터화, 멀티코어 분배**까지 다양한 고급 기능을 제공하며 성능과 안정성을 동시에 보장한다.

2.7 캐시 구조 (L1, L2, L3)

🧠 기본 개념

캐시(Cache)란, **CPU와 메인 메모리(RAM) 사이의 속도 차이를 줄이기 위한 고속 임시 메모리**이다.

> 캐시는 **CPU가 자주 참조하는 데이터나 명령어를 미리 저장해두고**,
> 다시 접근할 때 빠르게 불러올 수 있도록 해준다.
> 이를 통해 **메모리 병목(bottleneck)**을 완화하고, CPU가 **풀속력으로 연산**할 수 있게 해줌.

🔺 왜 캐시가 필요한가?

구성 요소	접근 속도	예
레지스터	수 ns 이하	CPU 내부
L1 캐시	수 ns	CPU에 가장 가까운 캐시
L2 캐시	수십 ns	약간 멀리 있지만 빠름
L3 캐시	수백 ns	여러 코어가 공유
RAM (DRAM)	수 백 ns ~ 1 μs	메인 메모리
SSD/HDD	수 μs ~ ms	영구 저장장치

→ **CPU는 너무 빠르기 때문에**, RAM만 사용하면 **항상 기다려야 하는 상황**이 생김
→ 그래서 **작지만 빠른 캐시 메모리를 계층적으로 구성해서** 속도를 극복함

🧩 캐시 계층 구조

```
                [ CPU ]
                   |
            +------------+
            |  L1 Cache  | ← 매우 작고 매우 빠름 (명령어/데이터 분리 가능)
            +------------+
```

```
                |
          +------------+
          |  L2 Cache  | ← 중간 속도, 더 큰 용량
          +------------+
                |
          +------------+
          |  L3 Cache  | ← 비교적 느림, 다수 코어 간 공유
          +------------+
                |
            [ RAM ]
```

📦 L1, L2, L3 캐시 비교

항목	L1 캐시	L2 캐시	L3 캐시
위치	CPU Core 내부	CPU Core 내부 or 외부	CPU Die 내부, 다수 코어와 공유
속도	가장 빠름	빠름	느린 편 (RAM보다는 빠름)
크기	16~128KB	128KB~1MB	2MB~64MB
용도	명령어/데이터 캐시	L1보다 큰 데이터 저장	코어 간 데이터 공유
구조	분리형 (I-Cache, D-Cache)	통합형이 많음	통합형
접근 대상	해당 코어	해당 코어	모든 코어
교체 정책	자주 갱신	중간	가장 늦게 제거됨

🧠 작동 방식 (Cache Miss와 Hit)

용어	설명
Cache Hit	CPU가 원하는 데이터가 캐시에 **이미 있음** → 빠르게 처리 가능
Cache Miss	CPU가 원하는 데이터가 캐시에 없음 → RAM에서 불러와야 함
Write-back	캐시에만 쓰고, 나중에 메모리에 반영
Write-through	캐시와 메모리에 동시에 쓰기

→ 성능 좋은 시스템일수록 **Hit 비율이 높아야 함**

🧬 캐시 메모리의 기술 요소

기술	설명
Set-Associative Mapping	특정 메모리 주소가 여러 캐시 라인 중 하나에 매핑 가능
Least Recently Used (LRU)	가장 오래 안 쓰인 데이터를 먼저 제거
Prefetching	자주 사용되는 패턴을 미리 읽어와서 캐시에 저장

기술	설명
Inclusive vs Exclusive Cache	L3에 있는 데이터가 L1에도 있는지 여부에 따라 구조 설계가 달라짐

🧠 멀티코어에서의 캐시

캐시 수준	접근 방식
L1/L2	보통 각 CPU 코어 전용
L3	여러 코어가 공유 (공유 캐시)
문제점	캐시 일관성(Coherency) 유지 필요 → MESI, MOESI 프로토콜 사용