5. 메모리 시스템

5.1 RAM, ROM, Flash, EEPROM 차이

🧠 기본 분류 기준

구분 기준	설명
휘발성 여부	전원이 꺼지면 데이터가 지워지는가
읽기/쓰기 가능성	데이터를 읽기만 가능한가, 쓰기도 가능한가
속도	CPU가 접근했을 때 속도가 얼마나 빠른가
사용 목적	임시 저장용인지, 영구 저장용인지

📦 1. RAM (Random Access Memory)

✅ 개념

- 데이터를 **임시로 저장하는 고속 메모리**
- **휘발성**: 전원이 꺼지면 데이터 사라짐

✅ 특징

항목	설명
읽기/쓰기	✅ 자유롭게 가능
속도	매우 빠름 (CPU 직접 접근)
전력 필요	항상 전원 필요
용도	프로그램 실행 중 임시 데이터 저장 (변수, 스택, 힙 등)
종류	SRAM, DRAM (SDRAM, DDR 등)

> 💡 **DRAM**이 일반적인 PC/MCU의 주기억장치
> 💡 **SRAM**은 캐시(Cache) 메모리에 사용됨

📦 2. ROM (Read-Only Memory)

✅ 개념

- 제조 시 **내용이 고정된 비휘발성 메모리**
- 전원이 꺼져도 데이터 유지됨

✅ 특징

항목	설명
읽기	✅ 가능
쓰기	❌ 불가능 (일반적 ROM은 변경 불가)
속도	읽기 속도는 빠름
용도	초기 부팅 코드, 고정 펌웨어 저장

> 💡 전통적 ROM은 한 번만 기록 가능 (OTP: One-Time Programmable)

📦 3. Flash Memory

✅ 개념

- **전기적으로 삭제·재기록 가능한 비휘발성 메모리**
- EEPROM을 **블록 단위로 확장한 것**

✅ 특징

항목	설명
읽기/쓰기	✅ 가능 (쓰기 시 블록 삭제 후 재기록 필요)
속도	읽기는 빠름, 쓰기·삭제는 느림
수명	**한 셀당 수천~수십만 번 쓰기 가능**
용도	SSD, USB, SD 카드, MCU의 코드 저장 공간
구조	NOR, NAND 구조 구분 있음

> 💡 **MCU 내장 Flash**는 대부분 프로그램 코드 저장용
> 💡 **NAND Flash**는 대용량 저장 장치용 (SSD, USB 등)

📦 4. EEPROM (Electrically Erasable PROM)

✅ 개념

- **전기적으로 1바이트 단위 삭제/쓰기 가능한 비휘발성 메모리**
- Flash보다 **세밀한 제어** 가능

✅ 특징

항목	설명
읽기/쓰기	✅ 가능 (바이트 단위 쓰기)
속도	느림 (Flash보다 느림)

항목	설명
수명	Flash보다 낮음 (약 10^4~10^5번 쓰기)
용도	**구성 정보, 설정값 저장** (예: 시리얼번호, 캘리브레이션)
형태	외장 칩 (I^2C, SPI 등), MCU 내장 가능

📊 네 가지 메모리 비교 요약표

항목	RAM	ROM	Flash	EEPROM
휘발성	✅ 예	❌ 아니오	❌ 아니오	❌ 아니오
읽기 가능	✅	✅	✅	✅
쓰기 가능	✅	❌	✅ (블록 단위)	✅ (바이트 단위)
속도	⏱️ 매우 빠름	⏱️ 빠름	⚠️ 중간	🐢 느림
용도	임시 데이터	부트 코드	프로그램, 파일 저장	설정, 구성값
사용 예	변수, 스택	BIOS, 펌웨어	SSD, MCU 코드	MCU 설정값 저장

🎯 정리 한 줄 요약

메모리	한 줄 설명
RAM	빠르지만 전원 꺼지면 사라짐. 실행 중인 데이터 저장용.
ROM	제조 시 고정된 읽기 전용 메모리. 변경 불가.
Flash	프로그램·파일 저장에 쓰는 재기록 가능한 비휘발성 메모리.
EEPROM	설정값 저장용, 바이트 단위로 쓰고 지울 수 있는 느린 비휘발성 메모리.

5.2 SRAM vs DRAM 비교

🧠 기본 정의

종류	정의
SRAM (Static RAM)	데이터를 **전기적으로 유지**하는 **정적 RAM**, 리프레시 불필요
DRAM (Dynamic RAM)	데이터를 **캐패시터에 저장**하는 **동적 RAM**, 주기적 리프레시 필요

🔬 내부 구조 비교

✅ SRAM: 트랜지스터 기반 (보통 6T Cell)

```
- 플립플롭 구조 (6개의 트랜지스터)
- 전압을 유지함으로써 상태 지속
```

✅ DRAM: 캐패시터 + 트랜지스터 (1T1C 구조)

```
- 1개의 트랜지스터 + 1개의 캐패시터
- 캐패시터에 저장된 전하가 점점 새어나가서 → 주기적으로 리프레시 필요
```

🔩 주요 비교표

항목	SRAM	DRAM
저장 방식	플립플롭(트랜지스터)	캐패시터 + 트랜지스터
리프레시 필요 여부	❌ 불필요	✅ 주기적으로 필요
접근 속도	⚙️ 매우 빠름	⚠️ 느림 (수 ns~수십 ns)
집적도	낮음 (셀 크기 큼)	높음 (셀 크기 작음)
용량	작음	큼
전력 소비	높음 (항상 유지)	낮음 (리프레시 때만 사용)
가격	비쌈	쌈
안정성	높음	리프레시 실패 시 불안정
사용 용도	캐시 메모리 (L1, L2, L3), MCU 내부 RAM	주 메모리(RAM), 그래픽 메모리

⚙️ 동작 예

상황	SRAM	DRAM
CPU가 변수에 접근	바로 읽고 씀 (항상 유지됨)	DRAM 컨트롤러가 리프레시하면서 읽고 씀
메모리 셀 상태	전기 상태 지속	전기 충전 → 시간 지나면 방전됨

🧪 실무 사용 예시

장치	SRAM 사용 위치	DRAM 사용 위치
CPU	L1/L2/L3 캐시	메인 메모리 (DDR4/DDR5)
MCU	내장 고속 RAM (stack 등)	외장 메모리 (확장 RAM)
GPU	레지스터 캐시, 제어버퍼	GDDR 메모리

장치	SRAM 사용 위치	DRAM 사용 위치
FPGA	FPGA 내부 RAM block	외부 DDR 인터페이스 연결

🧠 전력 측면 차이

- **SRAM**: 지속적으로 전압을 유지해야 하므로 소비 전력 ↑
- **DRAM**: 대기 시 소비 전력 낮고, 리프레시 시만 전력 ↑
  → 전력 효율이 중요한 모바일 기기나 노트북에서는 DRAM이 주력

📌 요약 정리

항목	SRAM	DRAM
속도	✅ 빠름 (수 ns)	❌ 느림 (10~100 ns)
가격	❌ 비쌈	✅ 저렴
용량	❌ 작음 (MB급)	✅ 큼 (GB급)
리프레시	❌ 불필요	✅ 필수
구조	6T	1T1C
사용	CPU 캐시, MCU 고속 RAM	PC 메인메모리, 그래픽 RAM

> ✅ **SRAM은 적고 빠른 캐시 용도**,
> ✅ **DRAM은 많고 느리지만 저렴한 메인 메모리용**

5.3 캐시 메모리 계층 구조

🧠 기본 개념

캐시(Cache)는 **CPU와 메인 메모리 사이에 위치**하며,
자주 사용하는 데이터를 임시로 저장해 **메모리 접근 시간을 줄이기 위한 고속 메모리**이다.

> 현대 CPU는 너무 빠르고, 메인 메모리는 상대적으로 너무 느리기 때문에
> 중간에서 데이터를 "미리 저장해두고 재사용"할 수 있는 캐시가 필수적이다.

🔺 왜 계층(Hierarchy)을 구성할까?

캐시는 다음의 **상충되는 요구**를 해결해야 한다:

요구	해결 방법
속도는 빠를수록 좋다	→ 작은 고속 캐시 사용 (L1)
용량은 클수록 좋다	→ 큰 저속 캐시 사용 (L2, L3)
비용은 적어야 한다	→ 고속 메모리의 양을 제한

그래서 캐시는 다음과 같은 **계층 구조**로 배치된다:

📐 계층 구조 요약

계층	위치	특징	용도
L1 캐시	CPU Core 내부	🏎️ 가장 빠름, ⏳ 용량 작음 (32~64KB)	명령어/데이터를 즉시 제공
L2 캐시	Core 내부 or 근접	⚡ 빠름, 💾 용량 중간 (128KB~1MB)	L1 미스 처리
L3 캐시	CPU 다이 내부 (공유)	🚗 느림, 📀 용량 큼 (2~64MB)	코어 간 데이터 공유
L4 캐시	일부 고급 CPU, 외장	매우 느림, 매우 큼	메모리처럼 동작 (거의 없음)
RAM (Main Memory)	외부 DRAM	📉 매우 느림, 🧠 대용량	모든 데이터의 원본 저장소

🔁 캐시 접근 흐름 (일반적인 메모리 접근 시퀀스)

1. CPU → **L1 캐시** 검색
2. 없다면 → **L2 캐시** 검색
3. 없다면 → **L3 캐시** 검색
4. 없다면 → **메인 메모리(RAM)** 접근

→ 빠를수록 먼저 접근, **미스(Miss)** 시 다음 단계로 이동
→ **히트(Hit)** 시 즉시 처리

🧩 캐시 계층별 비교표

항목	L1	L2	L3
위치	Core 내부	Core 내부/근처	CPU 전체 공유
속도	🏎️ 가장 빠름	⚡ 중간	🚗 가장 느림
용량	작음 (32~64KB)	중간 (256KB~1MB)	큼 (2~64MB)
히트율	낮음	중간	높음
접근 시간	~1ns	~~4~~12ns	~~20~~50ns
공유 여부	비공유	보통 비공유	공유

🔄 캐시 일관성 (Cache Coherency)

> 멀티코어 시스템에서는 각 코어의 캐시가 **서로 다른 값을 가지고 있으면 안 됨**
> 이를 위한 메커니즘이 **캐시 일관성 프로토콜**이다.

대표 프로토콜: MESI (Modified, Exclusive, Shared, Invalid)

상태	설명
Modified	캐시만 변경, 메모리와 불일치
Exclusive	캐시만 가지고 있음, 메모리와 동일
Shared	다른 캐시에도 있음
Invalid	유효하지 않은 상태

📦 캐시 설계 요소

항목	설명
블록(Block)	캐시에서 한 번에 가져오는 데이터 단위
라인(Line)	캐시 내 데이터 저장 단위
어소시에이티비티 (Associativity)	데이터를 어느 위치에 저장할 수 있느냐
교체 정책 (Replacement Policy)	`LRU`, `FIFO`, `Random` 등

🧪 예시

> i7 12700K 기준 (단일 코어당)

캐시 종류	크기	공유 여부
L1 I-Cache	32KB	❌
L1 D-Cache	32KB	❌
L2 Cache	1MB	❌
L3 Cache	30MB (전체 공유)	✅

📌 요약 정리

구분	설명
캐시 계층	L1 → L2 → L3 → RAM 순으로 구성됨
속도	L1이 가장 빠르고, RAM이 가장 느림
용량	반대로 RAM이 가장 크고, L1이 가장 작음
일관성 유지	MESI 같은 프로토콜로 코어 간 캐시 상태 동기화
목적	**CPU와 RAM 간 속도 차이를 완충**하여 전체 시스템 성능 향상

5.4 MMU(Memory Management Unit)

🧠 MMU란?

MMU(Memory Management Unit)는 CPU 내부 또는 외부에 존재하는 **하드웨어 장치**로,
가상 주소(Virtual Address)를 **물리 주소(Physical Address)**로 **변환(Mapping)**하고,
메모리 접근을 **관리, 보호, 제어**하는 역할을 한다.

> 쉽게 말하면:
> CPU는 메모리 주소 `0x00000000`이라고 하지만, 실제 RAM의 주소는 전혀 다를 수 있다.
> **그 주소를 번역해주는 번역기 + 보안 관리자 역할**을 하는 게 MMU이다.

🔁 MMU의 주요 역할

역할	설명
주소 변환	가상 주소 → 물리 주소 매핑
메모리 보호	프로세스 간 메모리 영역 침범 방지
페이지 관리	가상 메모리를 페이지 단위로 관리
캐시 제어	캐시 사용 여부를 제어할 수 있음
권한 체크	읽기/쓰기/실행 여부 확인

🧩 주소 변환 기본 개념

✅ 흐름 요약

```
[CPU] → 가상 주소(VA) → [MMU] → 물리 주소(PA) → [메모리 접근]
```

> 가상 주소를 사용하면, 여러 프로세스가 서로 **독립된 주소 공간**을 가진 것처럼 실행할 수 있다.
> 실제 메모리는 MMU가 알아서 **재배치하고 보호**해줌.

📐 MMU 내부 구성

구성 요소	설명
페이지 테이블(Page Table)	VA ↔ PA 변환 정보를 가진 테이블 (보통 메모리에 있음)
TLB (Translation Lookaside Buffer)	페이지 테이블 캐시 (자주 쓰는 주소 변환 정보 저장)
Access Control Logic	접근 권한 체크, 보호 도메인 관리
Cache Control Unit	캐시 사용 여부 지정 (ex. I/O 영역은 캐시 X)

📦 예: 32비트 주소 변환 (페이징 방식)

- 가상 주소(VA): `0x1234ABCD`
- 페이지 크기: 4KB (12비트 오프셋)
- 페이지 번호(Page Number): 상위 20비트 `0x1234A`
- 오프셋(Page Offset): 하위 12비트 `0xBCD`

MMU는 페이지 번호 `0x1234A`에 대응하는 물리 프레임을 찾아서,
해당 프레임의 시작 주소 + 오프셋을 더해 최종 **물리 주소(PA)**를 생성함.

🔎 MMU가 없는 경우의 문제점

문제	설명
보호 불가	한 프로세스가 다른 프로세스의 메모리 침범 가능
주소 충돌	모든 프로세스가 동일한 주소를 사용함 (재배치 불가)
멀티태스킹 불가능	하나의 주소 공간만 존재 → 프로세스 독립성 X
페이징, 스와핑 불가	가상 메모리 기능 불가능 → RAM 크기 제약

💡 MMU의 장점

기능	효과
가상 메모리 제공	프로세스마다 독립된 주소 공간 할당 가능
보호 메커니즘	메모리 경계 넘어가면 예외 발생 (`Page Fault`)
효율적 메모리 사용	물리 메모리를 단편화 없이 관리 가능
스와핑 및 페이징	디스크와 연동한 가상 메모리 구현 가능
TLB 캐시	빠른 주소 변환으로 성능 향상

🧠 MMU가 지원하는 메모리 관리 기법

기법	설명
페이징(Paging)	고정 크기 블록(페이지) 단위로 주소 변환
세그먼트(Segmentation)	논리적 단위(코드, 데이터 등)로 주소 구분
혼합 구조	x86: 세그먼트 + 페이징, ARM: 페이지 기반

📊 실무 예: ARM Cortex-A MMU 구조

구성 요소	특징
TTBR (Translation Table Base Register)	페이지 테이블의 시작 주소 등록
TLB	주소 변환 캐시
Domain Access Control	권한 레벨별 보호 설정
Page Table Entry	각 페이지별 접근 권한, 캐시 속성, 물리 주소 포함

📌 요약 정리

항목	설명
MMU	가상 주소를 물리 주소로 변환하는 하드웨어
기능	주소 변환 + 메모리 보호 + 접근 권한 제어
필수성	현대 멀티태스킹 운영체제에 반드시 필요
구성 요소	TLB, 페이지 테이블, 접근 제어, 캐시 제어
없는 경우	보호 불가, 충돌 다발, 가상 메모리 미지원

5.5 가상 메모리와 페이징

🧠 가상 메모리(Virtual Memory)란?

가상 메모리는 CPU가 접근하는 메모리 주소를 실제 물리 메모리와 다르게 **추상화된 주소 공간**으로 제공하는 시스템이다.

> 즉, CPU는 실제 RAM이 몇 GB인지 몰라도 `0x00000000`부터 마음껏 사용하는 것처럼 보이게 해주는 장치.

✅ 목적

항목	설명
프로세스 **독립적인 주소 공간** 제공	충돌 방지, 보안성 강화
메모리 **재배치 가능**	실제 주소에 관계없이 실행 가능
RAM 부족 시 디스크 일부를 메모리처럼 사용	스와핑(Swapping) 구현
메모리 보호	각 프로세스는 자기 메모리만 접근 가능

📦 주소 체계 요약

```
[가상 주소] → [MMU] → [물리 주소]
        ↑
    프로세스가 사용하는 주소
```

🧩 페이징(Paging)이란?

페이징은 가상 메모리 공간과 물리 메모리를 **고정 크기의 블록 단위(페이지)**로 나눠
페이지 테이블(Page Table)을 통해 주소를 매핑하는 방식이다.

✅ 기본 구조

용어	설명
페이지(Page)	가상 메모리 상의 고정 크기 블록 (ex. 4KB)
프레임(Frame)	물리 메모리 상의 블록 (페이지와 크기 동일)
페이지 테이블	각 페이지가 어느 프레임에 매핑되었는지 저장하는 테이블
TLB	자주 쓰는 페이지 매핑 캐시 (성능 향상)

📐 주소 변환 과정 예시

> 32비트 가상 주소, 페이지 크기 4KB (2^{12} = 4096)

- 가상 주소: `0x1234ABCD`
- 상위 20비트: **페이지 번호** (`0x1234A`)
- 하위 12비트: **페이지 오프셋** (`0xBCD`)

➡️ MMU는 페이지 번호에 해당하는 **프레임 번호**를 찾아
`프레임 시작 주소 + 오프셋`으로 **물리 주소** 계산

📊 페이지 테이블 항목 구성 (Page Table Entry, PTE)

필드	설명
Valid Bit	해당 페이지가 유효한지 여부
Frame Number	매핑된 물리 프레임 번호
R/W/X 권한	읽기, 쓰기, 실행 가능 여부
Dirty Bit	수정된 상태인지 여부 (쓰기 후 저장 필요 여부)
Accessed Bit	최근 접근 여부 (교체 알고리즘용)
Cache 속성	캐시 가능 여부 등

🚨 페이지 폴트(Page Fault)

> 가상 주소에 대해 페이지 테이블에 **유효한 매핑이 없을 때 발생하는 예외**

처리 흐름:

1. CPU → 가상 주소 접근
2. MMU → 유효한 매핑 없음
3. 페이지 폴트 예외 → OS 개입
4. 필요한 페이지를 디스크에서 RAM으로 로드
5. 페이지 테이블 갱신 후 재시도

🧠 가상 메모리 주요 기법

기법	설명
Demand Paging	실제 접근할 때만 페이지 로드
Copy-on-Write (COW)	fork 시 복사 대신 공유, 수정 시 복사
Swapping	페이지를 디스크에 저장하고 RAM에서 제거
Page Replacement	프레임 부족 시 어떤 페이지를 교체할지 결정 (LRU, FIFO 등)

💡 페이지 교체 알고리즘 예시

알고리즘	설명
FIFO	가장 오래된 페이지 제거
LRU	가장 오래 안 쓰인 페이지 제거
Clock (Second Chance)	LRU 근사 방식으로 효율적

📊 가상 메모리와 실제 주소 공간

구분	설명
가상 주소 공간	프로세스당 `4GB`, `2³²`, `2⁶⁴` 등 (논리적)
물리 메모리	실제 장착된 RAM (ex. 8GB)
매핑 방식	MMU가 페이지 단위로 가상 ↔ 물리 연결

📎 64비트 시스템의 경우

- 가상 주소: 64비트지만 실제 사용은 일부만 사용 (예: 48비트만 사용)
- 페이지 테이블은 다단계 구조 (multi-level page table)로 구성됨

📌 요약 정리

항목	설명
가상 메모리	CPU가 직접 물리 주소를 몰라도 되게 만드는 주소 추상화 기술
페이징	고정 크기 페이지 단위로 메모리를 분할하고 매핑
MMU	주소 변환과 메모리 접근 제어를 담당
페이지 테이블	가상 → 물리 주소 변환 정보 저장
페이지 폴트	매핑 없음 → OS 개입 → 페이지 로딩
스와핑	RAM 공간 부족 시 디스크와 교체

5.6 DMA(Direct Memory Access)

🧠 DMA란?

DMA(Direct Memory Access)는 **CPU를 거치지 않고**,
I/O 장치 ↔ 메모리 간 데이터를 직접 전송할 수 있도록 해주는 **하드웨어 컨트롤러 또는 기능**이다.

> 즉, CPU가 직접 데이터를 하나씩 옮기는 수고를 덜고,
> **DMA가 대신 데이터 이송을 맡는 구조**이다.

📦 왜 DMA가 필요한가?

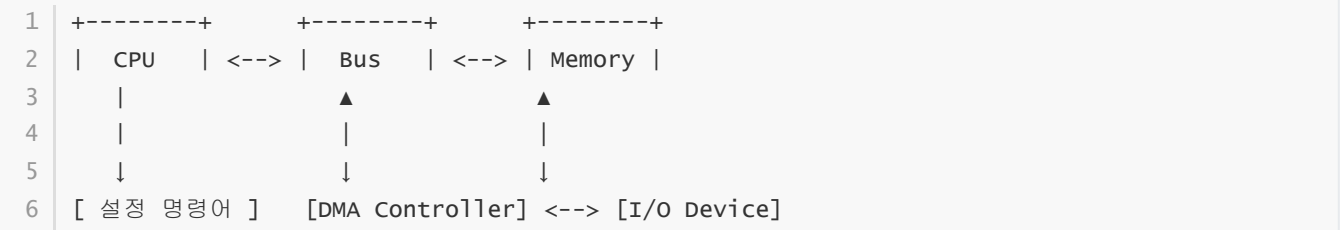
✅ 기존 방식 (Programmed I/O 방식)

- CPU가 직접 `LOAD`, `STORE` 명령으로 I/O 장치 ↔ 메모리 간 데이터를 이동
- 단점:
  - CPU가 모든 데이터를 처리해야 함 → **낮은 효율성**
  - 고속 I/O 장치 등장 이후 병목 발생

✅ DMA 방식

- CPU는 **초기 설정만 하고**,
- DMA 컨트롤러가 **메모리와 I/O 장치 간 전송을 직접 수행**
- CPU는 **그동안 다른 연산 수행 가능**

🔩 DMA 시스템 구성도



⚙️ DMA 동작 단계 (기본 시퀀스)

1. **CPU가 DMA에 다음을 설정**
    - 메모리 시작 주소
    - I/O 포트 주소
    - 데이터 크기
    - 전송 방향 (읽기/쓰기)
2. **DMA 시작 명령**
3. **DMA가 버스를 점유(Bus Arbitration)**
   → DMA가 CPU 대신 버스를 사용함
   → 이를 "버스 사이클 훔치기(Bus Stealing)"라고도 한다.
4. **전송 완료 시 인터럽트 발생**
5. **CPU는 인터럽트를 받아 후처리 수행**

📊 DMA 전송 방식 유형

방식	설명
Burst Mode	한 번 버스를 점유하여 블록 단위 데이터 전송 (빠름)
Cycle Stealing	CPU와 버스를 번갈아 사용 (중간 속도, CPU 병행 가능)
Transparent Mode	CPU가 버스를 사용하지 않을 때만 DMA가 사용 (가장 느림)

🧪 DMA 모드 유형

모드	설명
Single Transfer Mode	한 번에 1바이트씩 처리
Block Transfer Mode	지정된 블록 크기만큼 반복 처리
Demand Mode	I/O 장치가 요청할 때만 처리
Cascade Mode	여러 DMA를 연결하여 우선순위로 처리

📂 DMA의 사용 예시

시스템	DMA 사용 용도
MCU (STM32, AVR 등)	ADC → RAM 전송, UART 수신 버퍼 저장
SoC (ARM Cortex-A)	SPI, SDIO, Ethernet 고속 전송
PC 시스템	디스크 I/O, 그래픽 메모리 전송
GPU	VRAM ↔ CPU 메모리 간 데이터 공유

🔌 MCU용 DMA 예시 코드 (STM32 HAL)

```
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, 128);
```

> 이 코드는 `adc_buffer`에 ADC 데이터를 128개 만큼 DMA로 자동 저장하는 예시이다.

📈 DMA vs PIO 비교

항목	PIO (CPU 직접 제어)	DMA
CPU 개입	필수	거의 없음
효율	낮음 (Busy wait)	높음
전송 속도	느림	빠름
CPU 부하	높음	낮음
구현 복잡도	단순	약간 복잡 (초기 설정 필요)

📌 요약 정리

항목	설명
DMA	CPU 대신 I/O ↔ 메모리 전송을 수행하는 하드웨어 기능
역할	CPU 오버헤드 최소화, 병렬 처리 가능
구성	DMA 컨트롤러 + 설정 레지스터 + 인터럽트
모드	Burst, Cycle Stealing, Transparent
사용 예	ADC, UART, SPI, 파일 I/O, GPU 메모리 전송 등

5.7 메모리 인터페이싱 회로 설계

🧠 메모리 인터페이싱이란?

메모리 인터페이싱이란, 마이크로프로세서(CPU)와 메모리 칩 간에
주소, 데이터, 제어 신호들을 정확하게 주고받을 수 있도록
물리적/논리적 회로를 구성하는 과정이다.

> 단순히 선만 연결한다고 되는 게 아니라,
> 메모리 칩의 **동작 조건, 타이밍, 버스 규격**을 맞춰줘야 함.

🧩 기본 회로 구성 요소

```
+------------+     +--------------------+
|  CPU / MPU |<--->| Memory (SRAM/ROM)  |
+------------+     +--------------------+
      |                      ▲
      ├─ 주소 버스 (A0~An)─────┘
      ├─ 데이터 버스 (D0~Dn) ◀▶
      ├─ 제어선
      |    ├─ /RD (Read)
      |    ├─ /WR (Write)
      |    └─ /CS (Chip Select)
      └─ 타이밍 제어 회로 (필요 시 Wait 등)
```

🔧 설계의 핵심 신호

신호	설명
주소 버스 (A0~An)	CPU가 접근하려는 메모리 주소를 지정
데이터 버스 (D0~Dn)	메모리와 CPU 간 데이터 송수신
/RD (Read)	읽기 제어 신호, 메모리 → CPU
/WR (Write)	쓰기 제어 신호, CPU → 메모리
/CS (Chip Select)	해당 메모리 칩을 선택하는 신호
OE/WE	Output Enable / Write Enable (SRAM/Flash 등에서 사용)

🧠 메모리 맵 구성

예: CPU가 64KB 주소 공간을 갖고, 32KB SRAM + 32KB ROM을 사용하는 경우

주소 범위	메모리 종류	/CS 생성 방식
`0x0000~7FFF`	SRAM	A15 = 0 → /CS_SRAM 활성화
`0x8000~FFFF`	ROM	A15 = 1 → /CS_ROM 활성화

```
Chip Select 논리 회로 예:

/CS_SRAM = NOT(A15)
 /CS_ROM = A15
```

🔩 실제 회로 예: 32KB SRAM (e.g. 62256) 연결

신호	연결 대상
A0~A14	CPU 주소선 A0~A14
D0~D7	CPU 데이터선 D0~D7
/CS	주소 디코더 출력 (예: A15 = 0)
/OE	/RD 신호 연결
/WE	/WR 신호 연결

> SRAM은 비동기이므로 /OE와 /WE 신호만 잘 제어하면 동작 가능함

🧬 주소 디코딩 회로 설계

방식	설명
풀 디코딩(Full Decoding)	모든 주소선을 활용한 정확한 디코딩 → 중복 없음
부분 디코딩(Partial Decoding)	일부 상위 주소선만 사용 → 설계 간단, 중복 가능성 있음
디코더 칩 사용	74LS138 (3-to-8), 74LS139 등으로 디코딩 로직 구성

```
예: 74LS138 (3-to-8) 디코더로 최대 8개 메모리 영역 선택 가능
```

🕒 타이밍 다이어그램 이해

메모리와의 인터페이스에서 가장 중요한 건 **타이밍 일치**한다.
읽기 사이클(Read Cycle), 쓰기 사이클(Write Cycle)을 정확히 맞춰줘야 함.

SRAM 읽기 타이밍 예시:

```
CPU 주소 출력 → /CS = LOW → /RD = LOW
→ 데이터 버스에 메모리 값 출력 → CPU 읽음
→ /RD = HIGH → /CS = HIGH
```

→ 위 타이밍 사이클은 메모리 칩 데이터시트 참고하여 **SETUP, HOLD, ACCESS TIME** 등을 만족해야 함

🧪 고급 설계 고려사항

항목	내용
버스 폭 차이	16비트 CPU ↔ 8비트 SRAM → 데이터 버스 분할
Wait 상태 삽입	느린 메모리를 위해 /WAIT 삽입 필요
BIU (Bus Interface Unit)	고급 CPU는 외부 접근 시 BIU 통해 주소, 데이터 다중화

항목	내용
메모리 뱅킹	고용량 메모리 → 뱅크 전환 (A16~A20 등 활용)
주소 라인 반전	특정 칩은 A0~An 연결 순서에 따라 출력이 달라짐 주의

📌 요약 정리

항목	설명
메모리 인터페이스	CPU ↔ 외부 메모리를 정확히 연결하는 회로 설계
핵심 신호	주소선, 데이터선, 제어선(/RD, /WR, /CS)
주소 디코딩	어느 주소 범위가 어느 칩을 선택할지 결정
타이밍 설계	데이터 시점과 제어 타이밍을 정밀하게 설계
응용 예	SRAM, ROM, Flash, DRAM 인터페이스 회로, MCU 확장 메모리 설계