

# 4. 버스 및 인터페이스

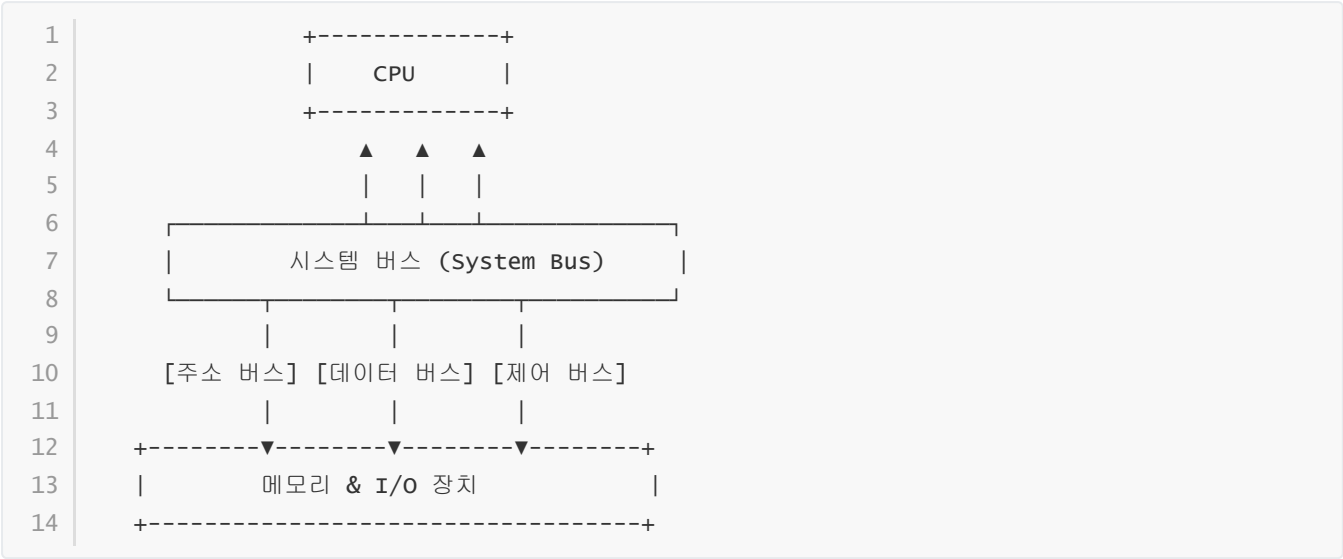
## 4.1 주소 버스, 데이터 버스, 제어 버스

### 🧠 개요: 버스란 무엇인가?

**버스(Bus)**란 CPU, 메모리, I/O 장치 사이에서 **데이터, 주소, 제어 신호를 전달하기 위한 공용 전송선로**이다. 일종의 전기적 "도로망"이라고 생각하면 된다.

버스는 데이터 흐름을 정리하고, 여러 하드웨어 요소들이 **동기화된 방식**으로 정보를 주고받게 한다.

### 🔗 전체 구조 요약



### 1. 주소 버스 (Address Bus)

#### ✓ 정의

CPU가 어떤 메모리 주소 또는 I/O 주소를 접근할지를 지정하기 위한 일방향 신호선.

#### ✓ 특징

- 일방향 (CPU → 메모리/I/O)
- 주소 버스의 비트 수에 따라 CPU가 접근할 수 있는 최대 메모리 용량이 결정됨

#### ✓ 예시

- 16비트 주소 버스 →  $2^{16} = 65,536$  개의 주소 공간 (64KB)
- 32비트 주소 버스 →  $2^{32} = 4GB$  주소 공간
- 64비트 주소 버스 → 이론상  $2^{64} = 18EB$  (실제 구현은 제한됨)

## 2. 데이터 버스 (Data Bus)

### 정의

CPU와 메모리/장치 간에 실제 데이터를 주고받는 이중방향 통신선.

### 특징

- 양방향 (CPU ↔ 메모리/I/O)
- 데이터 버스의 폭(bit 수)에 따라 한 번에 주고받을 수 있는 데이터 양이 결정됨

### 예시

- 8비트 데이터 버스 → 한 번에 1바이트 전송
- 32비트 데이터 버스 → 한 번에 4바이트 전송
- 64비트 → 최신 CPU 수준

## 3. 제어 버스 (Control Bus)

### 정의

전송되는 데이터가 무엇인지, 언제 전송할지를 정의하는 명령 신호들을 전달하는 통로.

### 주요 제어 신호

신호	역할
RD (Read)	메모리 또는 장치로부터 데이터 읽기
WR (Write)	메모리 또는 장치에 데이터 쓰기
CLK (Clock)	동기화용 클럭 신호
INT (Interrupt)	인터럽트 요청 수신
RESET	시스템 초기화
MEM/IO 선택 신호	접근 대상이 메모리인지 I/O인지 구분

제어 버스는 시스템이 “언제, 누구에게, 무엇을 할지”를 조율하는 전자 신호의 리더이다.

## 비교 요약표

항목	주소 버스	데이터 버스	제어 버스
방향	CPU → 외부	양방향	주로 CPU → 외부 (일부는 반대도 가능)
역할	주소 지정	데이터 전달	동작 제어
결정 요소	최대 주소 공간	단일 전송량	통신 타이밍 및 종류
비트 수	16, 32, 64 등	8, 16, 32, 64 등	다양한 신호선 (보통 10~20개 이상)

## 예시: 메모리 읽기 사이클

- 1 CPU가 주소 A에서 데이터를 읽으려 할 때:
- 2
- 3 1. 주소 A를 주소 버스에 출력
- 4 2. RD(Read) 신호를 제어 버스에 출력
- 5 3. 메모리에서 데이터 D를 데이터 버스에 출력
- 6 4. CPU가 데이터 D를 수신

## 요약 정리

- 주소 버스(Address Bus): 접근할 메모리나 장치의 위치 지정용 (일방향)
- 데이터 버스(Data Bus): 실제 데이터 전송용 (양방향)
- 제어 버스(Control Bus): 읽기/쓰기/인터럽트 등 제어 신호 전송용

이 세 가지는 CPU가 외부와 소통할 때 반드시 필요한 하드웨어 계층의 기본 통신 구조이다.

## 4.2 버스 아비트레이션(Bus Arbitration)

### 기본 개념

버스 아비트레이션이란, 둘 이상의 장치가 동시에 버스를 요청했을 때, 누가 먼저 버스를 사용할 수 있을지를 결정하는 과정이다.

마치 여러 사람이 한 개의 마이크를 쓰려고 손을 들었을 때, 사회자가 누구를 먼저 지명할지 정하는 절차와 비슷하다.

### 왜 필요한가?

- CPU, DMA, I/O 디바이스, 메모리 등 여러 장치가 동일한 시스템 버스(주소, 데이터, 제어)를 공유함
- 동시에 버스를 요구하면 충돌(Contend)이 발생함
- 이를 방지하고 시스템을 질서 있고 효율적으로 동작시키기 위해 아비트레이션이 필요하다.

### 기본 동작 흐름

- 1 [1] 여러 장치가 버스를 요청 (Bus Request)
- 2 [2] 버스 아비트레이터(Arbiter)가 우선순위를 판단
- 3 [3] 한 장치에만 Bus Grant 신호를 줌
- 4 [4] 해당 장치만 버스를 사용
- 5 [5] 작업 완료 후 Bus Release

### 구성 요소

구성 요소	설명
Bus Arbiter	중앙 통제 장치 또는 분산 제어 회로
Bus Request (BR)	장치가 버스를 요청하는 신호
Bus Grant (BG)	버스 사용 허가 신호

구성 요소	설명
Bus Busy (BB)	버스가 현재 사용 중임을 알리는 상태 신호

## 📦 버스 아비트레이션 방식 유형

### ✅ 1. 중앙집중형(Centralized Arbitration)

하나의 중앙 제어 장치(Arbiter)가 전체 버스 접근을 관리

방식	설명
Daisy Chain	우선순위가 선형으로 고정됨
Parallel Priority	모든 장치에 개별 신호 → Arbiter가 병렬 판별
Rotating Priority (Round Robin)	매번 우선순위를 순환함

#### 🔗 Daisy Chain 예

1	Request → [D1] → [D2] → [D3] → Arbiter
2	Grant ← ← ←

- D1이 항상 최우선 (공정성 ↓)
- 구현은 간단함 (선 연결 방식)

### ✅ 2. 분산형(Distributed Arbitration)

각 장치가 자율적으로 판단하여 서로 조정

특징	설명
없음	중앙 제어기 없이 하드웨어끼리 충돌 감지 및 회피
복잡함	하지만 분산 시스템에서는 필수
예시	CAN 버스, PCI 버스, Ethernet CSMA/CD 등

## 📊 아비트레이션 방식 비교

항목	중앙집중형	분산형
제어 구조	단순	복잡
구현	쉬움	어려움
확장성	낮음	높음
신뢰성	중앙장치 고장 시 전체 정지	분산 처리 가능
예	전통적 버스 시스템	현대 네트워크 버스, IoT, CAN, PCIe 등

## 💡 고급 기능: DMA와 아비트레이션

DMA(Direct Memory Access)는 CPU 없이 메모리와 장치 간 데이터 전송을 하게 해주는 컨트롤러이다.  
이때 DMA도 버스를 CPU와 경쟁하게 되고, **Arbiter가 중재**해야 한다.

우선순위 예시
1 인터럽트 컨트롤러
2 DMA 컨트롤러
3 CPU
4 일반 I/O 장치

→ 시스템의 실시간성, 데이터량, 긴급성에 따라 우선순위를 다르게 설정할 수 있음

## 📌 요약 정리

키워드	설명
Bus Arbitration	다수 장치의 버스 사용 요청 중 우선권을 중재
Centralized	Arbiter가 하나의 중심 제어기
Distributed	각 장치가 충돌 감지 및 회피
Daisy Chain	선형 연결 방식, 우선순위 고정
DMA 연계	CPU보다 높은 우선순위를 갖고 버스를 요청함
목적	충돌 없이 공정하고 효율적으로 자원 공유

## 4.3 메모리 맵 I/O vs I/O 맵 구조

### 🧠 기본 개념

CPU는 **외부 장치(I/O 디바이스)**에 접근해야 한다.

이 접근 방식은 두 가지로 나뉜다:

1. Memory-Mapped I/O (MMIO)
2. Isolated I/O (I/O-Mapped I/O)

### ◆ 1. Memory-Mapped I/O (MMIO, 메모리 맵 I/O)

장치(레지스터)를 메모리 주소 공간 안에 포함시킴

→ 메모리처럼 접근 가능

## ✓ 특징

- 장치 = 메모리 주소의 일부
- CPU는 `LOAD`, `STORE`, `MOV`, `LDR`, `STR` 같은 일반 명령어로 I/O 레지스터에 접근
- 별도 I/O 명령어 필요 없음
- 메모리와 장치 주소 공간이 통합되어 있음

## ✓ 예시 (ARM, RISC-V, 대부분의 RISC CPU)

```
1 | LDR R0, [0x4000_0000] ; 장치의 데이터 읽기
2 | STR R1, [0x4000_0010] ; 장치에 데이터 쓰기
```

0x4000\_0000 같은 주소가 UART, GPIO 등 장치 레지스터에 매핑되어 있음

## ◆ 2. I/O-Mapped I/O (Isolated I/O, 포트맵 I/O)

장치를 메모리 주소 공간과 완전히 분리된 별도의 공간으로 관리

→ 장치에 접근할 땐 전용 명령어(`IN`, `OUT`)를 사용

## ✓ 특징

- 장치와 메모리 주소 공간이 완전히 분리됨
- CPU는 `IN`, `OUT` 등의 전용 I/O 명령어로만 장치를 접근
- 주소 공간이 분리되어 있어 메모리 공간을 절약 가능
- 다만 CPU 명령어 집합이 복잡해짐

## ✓ 예시 (x86, 일부 CISC CPU)

```
1 | IN AL, 0x60 ; 포트 0x60의 값을 AL로 읽음 (예: 키보드 입력)
2 | OUT 0x64, AL ; AL 값을 포트 0x64로 보냄
```

## 📊 구조 비교표

항목	Memory-Mapped I/O	I/O-Mapped I/O
주소 공간	메모리와 공유	메모리와 분리
명령어	일반 LOAD/STORE	전용 I/O 명령어 ( <code>IN</code> , <code>OUT</code> )
CPU 설계	간단 (명령어 일원화)	복잡 (명령어 종류 증가)
하드웨어 인터페이스	메모리 버스 = I/O 버스	I/O 전용 버스 필요
접근 편의성	포인터 연산 등 사용 용이	전용 접근 필요
사용 예	ARM, MIPS, RISC-V 등	x86 계열
메모리 공간 소비	일부 주소 공간 사용됨	주소 공간 절약 가능
장치 수 제한	메모리 주소 수 제한 내	보통 256~64K 포트로 한정

## 실제 적용 예시

### ✓ Memory-Mapped I/O (RISC-V, ARM 등)

```
1 #define UART_BASE 0x40000000
2
3 void write_uart(char c) {
4     *(volatile char *) (UART_BASE) = c; // STORE 명령 사용
5 }
```

→ 장치 제어 레지스터를 메모리처럼 취급

### ✓ I/O-Mapped I/O (x86)

```
1 mov dx, 0x3F8      ; UART 포트 주소
2 mov al, 'A'
3 out dx, al         ; DX가 가리키는 포트로 A 출력
```

→ I/O 포트 주소에 접근하기 위해 `OUT` 명령 사용

## 어떤 방식이 더 좋을까?

관점	권장 방식
단순성, 일관성	Memory-Mapped I/O (CPU 설계가 간단해짐)
메모리 주소 공간 절약	I/O-Mapped I/O (x86의 옛 방식)
고성능, 파이프라인 최적화	Memory-Mapped I/O (RISC에 적합)
임베디드, ARM, MCU	대부분 MMIO
레거시 PC BIOS, ISA	대부분 I/O-Mapped I/O

## 요약 정리

분류	설명
Memory-Mapped I/O	장치를 메모리처럼 접근, 일반 명령어 사용 가능, 주소 공간 일부 소모
I/O-Mapped I/O	장치를 전용 명령어로 접근, 메모리 주소 절약, 명령어 및 버스 구조 복잡

→ 최신 CPU, 임베디드, RISC 아키텍처는 대부분 **Memory-Mapped I/O**를 사용하고 있음.



## 4.4 직렬(Serial) vs 병렬(Parallel) 통신

### 기본 개념

구분	설명
직렬 통신	데이터를 1비트씩 한 줄(line)로 순차 전송하는 방식
병렬 통신	데이터를 여러 비트(예: 8비트, 16비트)를 동시에 여러 선으로 전송하는 방식

즉, 데이터를 줄지어 보내냐, 일렬로 늘어놓고 한꺼번에 보내냐의 차이이다.

### 시각적 구조 비교

1	 병렬 통신 예 (8비트):
2	[비트1] [비트2] [비트3] ... [비트8] → 동시에 8선으로 전송
3	
4	 직렬 통신 예:
5	[비트1] → [비트2] → [비트3] → ... → 한 줄로 순차 전송

### 주요 비교 항목

항목	병렬 통신 (Parallel)	직렬 통신 (Serial)
데이터 전송 방식	여러 비트를 동시에 전송	한 비트를 순차적으로 전송
선(Line) 수	데이터 비트 수만큼 필요 (8~64선)	일반적으로 1~2개
속도	단거리 고속에 유리	장거리 고속에 유리 (신호 보정 쉬움)
간섭 및 노이즈	신호 간 간섭 많음 (skew 발생)	간섭 적고 안정적
하드웨어 비용	전송선, 커넥터 많아 고비용	적은 선로, 저비용
거리	수 미터 이하 (USB 1.1, 내부 버스 등)	수 m ~ 수 km (RS-232, I <sup>2</sup> C, CAN, SPI)
동기화	모든 선의 클럭 동기 필수	동기 or 비동기 가능 (스타트/스톱 비트)
적용 예	RAM ↔ CPU, 내부 데이터 버스	UART, SPI, I2C, USB, Ethernet 등

### 대표 프로토콜 예시

구분	대표 기술 / 인터페이스
병렬	DDR, PCI, IDE, 내부 BUS
직렬	UART, SPI, I2C, USB, SATA, Ethernet, RS-232



## 🧠 병렬 통신: 특징과 용도

### ✅ 장점

- 한 클럭에 많은 데이터 전송 가능 (즉시성 ↑)
- 간단한 디코딩

### ❌ 단점

- 클럭 스큐(Skew): 선마다 신호 도달 시간이 달라짐
- 거리 증가 시 노이즈 ↑
- 하드웨어 선로 수 ↑

### ✅ 주로 사용되는 곳

- CPU ↔ RAM, 내부 시스템 버스, FPGA 내부 통신

## 🔦 직렬 통신: 특징과 용도

### ✅ 장점

- 선 수가 적어 회로 간단
- 장거리 통신에 유리
- 동기/비동기 모두 가능

### ❌ 단점

- 한 번에 한 비트 → 느릴 수 있음 (하지만 고속 직렬화 기술로 극복됨)

### ✅ 주로 사용되는 곳

- 외부 통신: UART, USB, SPI, I2C, CAN
- 고속 연결: SATA, PCIe, Thunderbolt, LVDS

## ⚡ 현대의 경향

### ✅ "병렬 → 직렬" 전환이 주류 흐름

- PCI → PCIe
- PATA(IDE) → SATA
- 병렬 버스 → LVDS/SerDes 기반 고속 직렬 통신

➡ 이유: 고속 + 장거리 + 신뢰성을 직렬이 더 잘 만족시킴

## 📌 요약 정리

항목	병렬 통신	직렬 통신
데이터 전송	다중 선, 동시에	단일 선, 순차적으로
선 수	많음 (비용 ↑)	적음 (비용 ↓)

항목	병렬 통신	직렬 통신
속도	짧은 거리에서 빠름	긴 거리에서도 빠르고 안정
간섭/노이즈	큼	작음
적용	내부 시스템, CPU ↔ RAM	외부 연결, 장거리 버스, 센서
대표	DDR, PCI	SPI, I2C, UART, USB, SATA, CAN

## 4.5 공통 버스 구조 설계

### 기본 개념

공통 버스 구조(Common Bus Architecture)는 CPU, 메모리, I/O 장치 등이 하나의 공통된 통신 경로(Bus)를 통해 데이터를 주고받는 구조이다.

"하나의 길(버스)을 다 같이 사용해서 번갈아가며 말을 주고받는 방식"이라고 생각하면 이해하기 좋다.

### 구성 요소

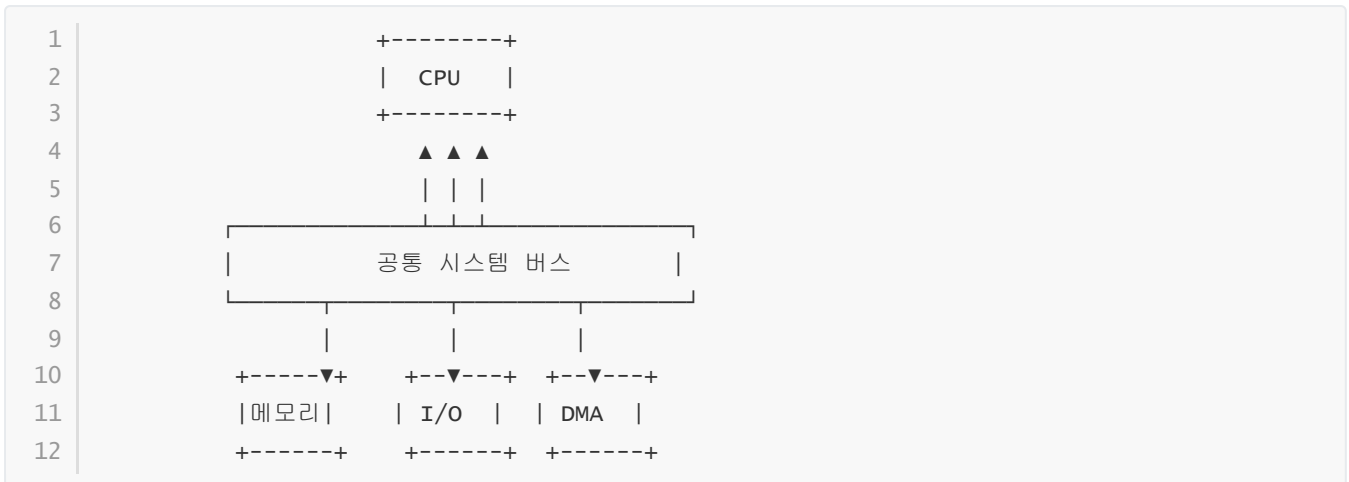
구성 요소	설명
버스 (Bus)	주소, 데이터, 제어 신호가 오가는 전송선 집합
CPU	데이터 처리 및 연산 수행
메모리	프로그램 및 데이터 저장
I/O 장치	키보드, 디스크, 네트워크 등 외부와의 인터페이스
버스 인터페이스	각 장치를 버스에 연결하는 회로
제어 회로 (Arbiter)	버스 사용 권한 조정 (Bus Arbitration 포함)

### 공통 버스의 종류

공통 버스는 보통 세 가지로 구성돼:

버스 종류	방향	용도
주소 버스	CPU → 메모리/I/O	접근 대상 주소 전송
데이터 버스	양방향	실제 데이터 전송
제어 버스	CPU → 모든 장치	읽기/쓰기/동기화 신호 제어

## ✖ 기본 설계 구조도



## 🔄 동작 흐름 예시: 메모리 읽기

1. CPU가 주소 버스에 원하는 주소 출력
2. 제어 버스에 **READ** 신호 출력
3. 메모리는 해당 주소의 데이터를 데이터 버스로 출력
4. CPU가 데이터 수신

## 🧠 장점과 단점

### ✅ 장점

항목	설명
구조 단순	모든 장치가 하나의 버스에 연결
확장 용이	장치 추가 시 동일 버스 사용 가능
비용 절감	배선 수, 회로 수 감소

### ❌ 단점

항목	설명
병목현상	버스가 공유되므로 동시 접근 불가
속도 제한	모든 장치가 같은 버스 속도에 의존
충돌 방지 회로 필요	Arbiter 필요 (Bus Arbitration)

## 공통 버스 구조 vs 전용 버스 구조

항목	공통 버스 구조	전용 버스 구조
설계 복잡도	낮음	높음
성능	병목 있음	높음
확장성	우수	제한적
비용	저렴	고가
사용 예	초창기 CPU 시스템, MCU	현대 고성능 SoC, 멀티코어 CPU 내부

## 실제 시스템 예

시스템	구조 형태
8051, AVR	단일 공통 버스
ARM Cortex-M	고성능 SoC 내부는 AMBA 버스 계층화
초기 x86	CPU, RAM, I/O가 공유 버스 사용
현대 PC	버스는 계층 구조 (FSB → PCIe, Memory Bus 등으로 분리)

## 고급 확장: 다중 버스 구조로의 진화

공통 버스 구조의 병목 문제를 해결하기 위해 발전된 구조들:

구조	특징
다중 버스 구조	CPU ↔ 메모리, CPU ↔ I/O 각각 전용 버스 분리
하버드 구조	명령어/데이터 버스 완전 분리
파이프라인 버스	내부적으로 명령어 단계마다 독립된 버스 사용
SoC 내부 버스	AMBA (AHB, APB), AXI, Wishbone 등 고속 직렬 구조

## 요약 정리

항목	설명
공통 버스 구조	하나의 공유된 버스를 통해 CPU, 메모리, I/O가 통신
3종 버스	주소(Address), 데이터(Data), 제어(Control)
장점	단순 설계, 저비용, 확장성
단점	동시 접근 불가 → 병목 발생, Arbiter 필수
현대 시스템	고속 처리 위해 다중 버스 구조로 진화 중