CSAPP 1강 시스템과 비트의 세계

Chapter1 시스템을 이해해야 하는 이유

❓ 추상화만으로 충분한가?

- 추상화는 코드를 깔끔하게 만든다.
  - 추상 자료형: `ICharacter`, `IMateria` ...
  - 점근적 분석: Big-O 표기법
- 하지만 현실을 무시하면...
  - 버그 디버깅이 어렵고
  - 성능 문제를 분석할 수 없다

👉 **시스템에 대한 깊은 이해는 프로그래머의 무기!**

현실1: 컴퓨터의 산술은 수학이 아니다

- ## 🧮 수학과 컴퓨터 산술의 차이

- `x^2 >= 0` 은 항상 참일까?
  - **float**: 항상 참
  - **int**: 오버플로우 발생 시 거짓될 수 있음
- `(x + y) + z == x + (y + z)` ?
  - **int**: 항상 참
  - **float**: 예) `1e20 + (-1e20 + 3.14)` → 결과는 `3.14`가 아님!

🧠 컴퓨터의 산술은 **정확하지만**, **수학적 성질은 보장하지 않음**

현실2: 어셈블리를 이해하자

🛠 꼭 어셈블리로 짜라는 건 아님!

- 어셈블리를 직접 짤 일은 드물다.
- 하지만 **읽고 해석할 수 있어야 한다**.
  - 버그가 발생했을 때 프로그램의 동작 파악
  - 성능 최적화 분석에 필요
    - 컴파일러가 해주는 최적화 / 컴파일러가 해주지 않는 최적화
    - 프로그램이 느린 원인 이해

현실3: 메모리는 균일하지 않다

- ## 💾 메모리는 계층 구조를 가진다

- L1, L2 캐시 → RAM → 디스크
- 접근 시간 차이 큼 (ns → ms)
- 프로그램을 메모리 시스템의 특성에 맞게 하는것

- ## ⚠ 메모리 오류는 치명적

- 논리적으로 관련 없는 메모리까지 손상
- 디버깅이 어렵고 오류가 지연되기도 함
- 예시:

```
int arr[3] = {1, 2, 3}; arr[5] = 10;  // 정의되지 않은 동작
```

- 가비지 컬렉터가 있는 언어 쓸거 아니면 공부 해야함^^

현실4: Big-O만으로 성능을 판단할 수 없다

🧠 상수 계수와 코드 구조도 중요!

- Big-O가 같아도 10배 성능 차이 날 수 있음
- 메모리 접근 패턴도 중요

예시:

```
void copyij(int src[2048][2048], int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}

void copyji(int src[2048][2048], int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

현실5: 시스템은 네트워크와 I/O도 포함한다

📡 입출력 시스템과 네트워크의 영향

- 느린 I/O는 프로그램 성능 병목
- 네트워크 환경에서는 동시성과 신뢰성 문제 발생

예시:

- 네트워크 패킷 손실 → 재전송 필요
- 멀티스레딩 → race condition, deadlock 등

여기까지가 컴퓨터 시스템을 이해해야 하는 이유

앞서 본 현실을 마주하자

- 추상화된 컴퓨터 시스템의 내부를 살펴볼 예정

Chapter2 비트, 바이트, 정수

비트: 컴퓨터 언어의 최소 단위

비트 모르는 사람 없죠?

🔌 왜 비트를 사용할까?

- 전자적인 구현이 간단
- 잡음에도 강한 신뢰성
- 읽어보면 좋은 글

바이트와 불 대수

🧱 바이트 = 8비트

- 범위: 0 ~ 255 (10진수), 00 ~ FF (16진수)

💡 불 연산

- `&`: AND
- `|`: OR
- `^`: XOR
- `~`: NOT

비트 벡터로 집합 표현 가능:

- A = `0b10110` → A집합에 1, 2, 4 포함됨

C에서의 비트 & 논리 연산

🔧 비트 연산자: `& | ~ ^`

- 모든 정수형 타입 사용 가능
  - long, int, short, char, unsigned
- 각 비트 단위로 연산 수행

🤔 논리 연산자: `&& || !`

- 0: 거짓, 0이 아닌 값: 참
- `&&`와 `||`는 **short-circuit** 동작 (Early termination)
- 0 또는 1을 반환

예시:

```
p && *p // null pointer 참조 방지

!0x41 == ~0x41 // 같을까?
!!0x41 == // 값은?
```

시프트 연산

▶ 좌/우 시프트

- `x << n`: 왼쪽으로 n비트 이동, 오른쪽은 0으로 채움
- `x >> n`: 오른쪽 이동

⚠ 오른쪽 시프트의 종류

- **논리 시프트**: 왼쪽을 0으로 채움 (unsigned)
- **산술 시프트**: MSB로 채움 (signed)

정수 표현: Unsigned & Signed

📐 비트로 숫자 표현하기

- Unsigned: $B2U(X) = \sum x_i * 2^i$
- Signed: $B2T(X) = -x_{w-1} * 2^{w-1} + \sum x_i * 2^i$

표현	값	이진수 (16비트)
UMax	65535	`11111111 11111111`
TMax	32767	`01111111 11111111`
TMin	-32768	`10000000 00000000`

Signed ↔ Unsigned 해석

🔄 비트는 같아도 해석이 다르다

Equivalence

- 0과 양수의 인코딩(표현 방법)이 동일하다.

Uniqueness

- 모든 비트 패턴이 각각 다른 정수 값을 표현한다.
- 모든 표현 가능한 정수가 고유한 비트 표현 방식을 가진다.

-> Can Invert Mappings

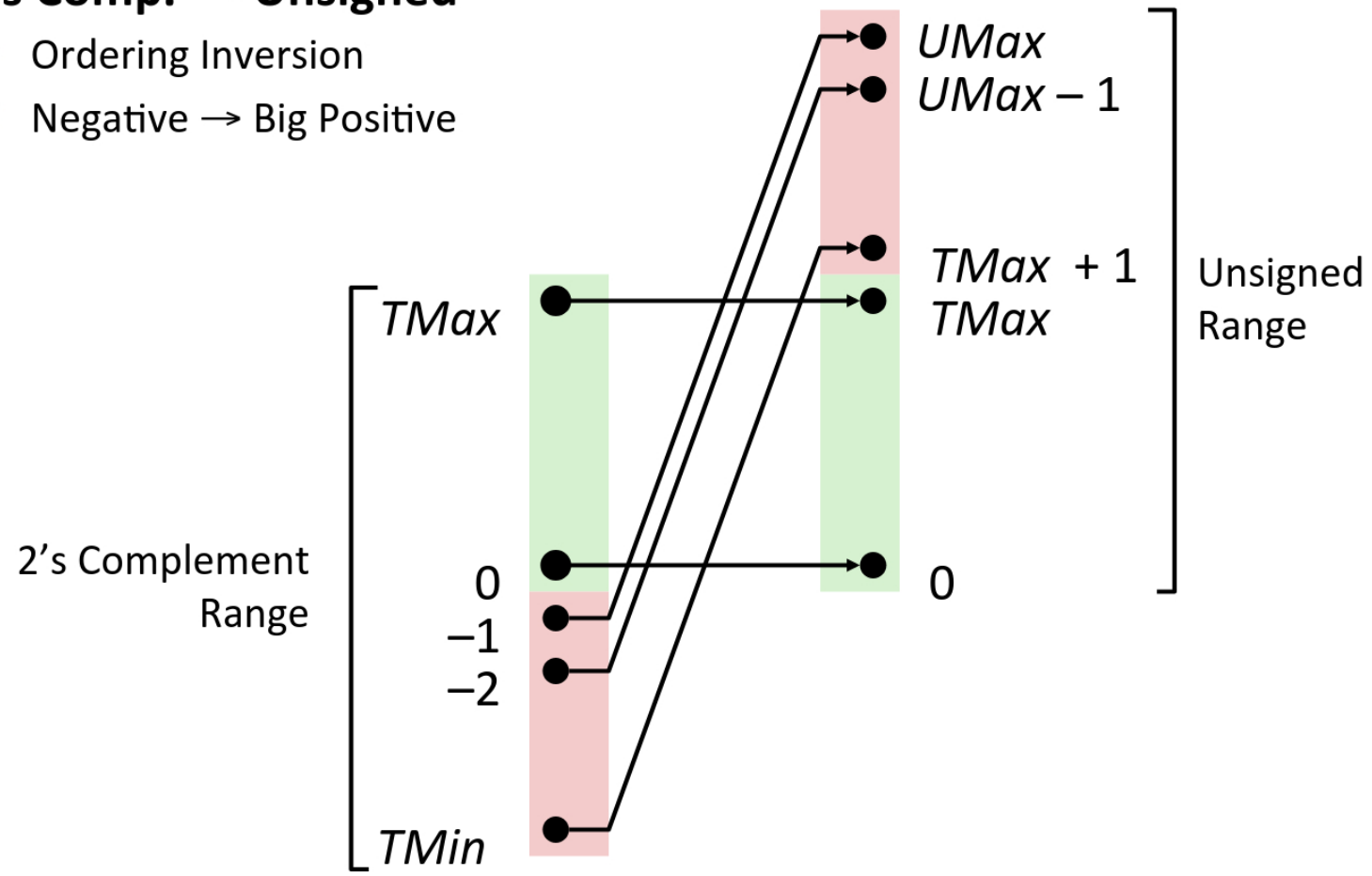
- $U2B(x) = B2U^{-}1(x)$ unsigned값을 비트패턴으로 변환할 수 있다.
- $T2B(x) = B2T^{-}1(x)$ signed값(2의 보수)을 비트패턴으로 변환할 수 있다.

예시:

```
비트: 10110001 -> Unsigned: 177 Signed (2's complement): -79
2의 보수표현 -4는 unsigned로 해석할 때 어떤 값을 갖는가?
```

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



C에서의 형 변환 주의

⚠ 암시적 변환의 함정

- Signed + Unsigned → 모두 Unsigned로 변환
- 결과는 의외일 수 있음

예시:

```
int x = -1;
unsigned int y = 1;
printf("%d\n", x > y); // true가 출력됨!

-1 < 0U // 참일까 거짓일까
2147483647 > (int)2147483648U
```

정수 확장과 절삭

📈 확장 (Expanding)

- Signed: 부호 비트로 채움
- Unsigned: 0으로 채움
- 확장 후에도 같은 값을 가짐

✂ 절삭 (Truncation)

- 상위 비트를 제거
- Unsigned: 나머지 연산과 동일
- Signed: 2의 보수 기준으로 해석
- 숫자가 충분히 작을 경우 기존 값을 그대로 가짐

C에서는

- 작은 타입을 큰 타입으로 확장
