3. 명령어 집합 구조(ISA)

3.1 명령어의 형식 (형식, 오퍼랜드, 주소지정방식)

(형식, 오퍼랜드, 주소지정 방식 포함)

🧠 명령어란?

명령어(Instruction)는 CPU가 수행할 수 있도록 구성된 **이진 형태의 기계어 코드**로,
CPU는 이 명령어를 **해석(Decode)**하고, **실행(Execute)**하는 과정을 통해 프로그램을 수행한다.

🔧 명령어의 기본 구조

```
+------------------+-----------------+---------------------+
|   Opcode (연산)   |   오퍼랜드1      |      오퍼랜드2       |
+------------------+-----------------+---------------------+
```

- **Opcode (Operation Code)**: 명령 종류 (예: ADD, SUB, MOV 등)
- **Operands (피연산자)**: 대상 레지스터, 메모리 주소, 즉시 값 등
- **Addressing Mode**: 오퍼랜드가 어디서 오는지를 결정하는 방식

📦 명령어 형식의 주요 구성

필드	설명	예시
Opcode	어떤 연산을 수행할지 지정	`ADD`, `SUB`, `MOV`, `JMP` 등
Source Operand(s)	입력 값의 위치	`R1`, `#5`, `[MEM]`
Destination Operand	결과를 저장할 위치	`R0`, `ACC`
주소 지정 방식(Addressing Mode)	오퍼랜드의 해석 방법	직접, 간접, 레지스터 등

🧩 명령어 형식의 종류

ISA마다 명령어의 구조는 다르지만, 대표적인 형식은 다음과 같다.

✅ 1. 3-오퍼랜드 명령어 형식

```
ADD R1, R2, R3   ; R1 ← R2 + R3
```

- R1: 목적지(Destination)
- R2, R3: 소스(Source)

✅ 2. 2-오퍼랜드 형식

```
ADD R1, R2        ; R1 ← R1 + R2
```

- 목적지와 첫 번째 피연산자가 같음

✅ 3. 1-오퍼랜드 형식

```
INC R1            ; R1 ← R1 + 1
```

✅ 4. 0-오퍼랜드 형식 (스택 기반)

```
PUSH 5
PUSH 3
ADD               ; 스택의 두 값을 더해 결과 다시 PUSH
```

🧠 주소 지정 방식 (Addressing Modes)

명령어가 **피연산자에 접근하는 방식**을 정의하는 중요한 개념이다.

주소지정 방식	설명	예시
즉시 (Immediate)	상수를 명령어에 직접 포함	`MOV R1, #10`
레지스터 (Register)	피연산자가 레지스터에 있음	`ADD R1, R2`
직접 (Direct)	메모리 주소를 직접 명시	`MOV R1, [1000h]`
간접 (Indirect)	주소가 저장된 레지스터를 통해 접근	`MOV R1, [R2]`
베이스 + 오프셋	기준 주소 + 변위 값 사용	`MOV R1, [R2 + 4]`
상대 (Relative)	현재 위치 기준으로 점프	`JMP PC+5`
레지스터 간접 + 오토 증가/감소	주소 읽은 뒤 레지스터 값 자동 변경	`MOV R1, [R2++]`

🧪 주소지정 방식 예시 비교

어셈블리 코드	의미	설명
`MOV R1, #5`	즉시	R1에 5 저장
`MOV R1, R2`	레지스터	R2 값을 R1에 복사
`MOV R1, [1000]`	직접	주소 1000에 있는 값을 R1에 저장
`MOV R1, [R2]`	간접	R2가 가리키는 주소의 값 →
`MOV R1, [R2+8]`	베이스 + 오프셋	R2 + 8 주소의 값을 읽어옴

📊 명령어 형식의 ISA별 차이

ISA	특징
x86 (CISC)	다양한 오퍼랜드 수, 가변 길이 명령어, 복잡한 주소지정 방식
ARM (RISC)	고정 길이 명령어(32비트), 3-오퍼랜드, 레지스터 중심
RISC-V	단순화된 고정 포맷, Load/Store 전용 명령 구분
MIPS	명확한 3-오퍼랜드, 일관된 형식 (R형, I형, J형)

📐 RISC 명령어 포맷 예시 (R-type: 3-오퍼랜드)

```
31        25 24    20 19    15 14    12 11     7 6      0
+-----------+--------+--------+--------+--------+--------+
| funct7    | rs2    | rs1    | funct3 | rd     | opcode |
+-----------+--------+--------+--------+--------+--------+

ADD rd, rs1, rs2
```

📌 요약 정리

- 명령어는 **Opcode + 피연산자(Operands)** + **주소지정 방식**으로 구성되며, 구조는 ISA에 따라 다를 수 있음
- 오퍼랜드는 레지스터, 메모리, 즉시 값 등이 될 수 있고, 이를 어떻게 해석할지는 주소지정 방식이 정의함
- RISC는 명확하고 간결한 형식(고정 길이), CISC는 유연하지만 복잡한 형식을 갖는다

3.2 데이터 이동 명령어

🧠 정의

데이터 이동 명령어는 CPU 내부 또는 외부의 **레지스터, 메모리, 즉시 값(상수)** 사이에서 데이터를 **복사, 저장, 로드**하는 명령어를 말한다.

> 연산은 하지 않고, **데이터의 위치를 바꾸는 것**이 목적이다.

🔁 주요 기능

기능	예시	설명
레지스터 ← 상수	`MOV R1, #10`	즉시 값 10을 R1에 저장
레지스터 ← 레지스터	`MOV R1, R2`	R2의 값을 R1으로 복사
메모리 ← 레지스터	`STR R1, [1000]`	R1 값을 주소 1000에 저장
레지스터 ← 메모리	`LDR R1, [1000]`	주소 1000의 값을 R1에 로드
레지스터 ← I/O	`IN R1, PORT1`	PORT1의 입력을 R1에 저장
I/O ← 레지스터	`OUT PORT2, R1`	R1 값을 PORT2에 출력

🧩 기본 명령어 종류별 설명

✅ 1. MOV (Move)

> 데이터 복사. 가장 기본적인 전송 명령어.

```
MOV R1, R2      ; R1 ← R2의 값
MOV R1, #15     ; R1 ← 15
```

> ❗ 메모리 간 직접 이동은 대부분의 ISA에서 **불가능**하고, 레지스터를 거쳐야 한다.
> (예: `MOV [A], [B]` → 안 됨 → `LDR R0, [B]`, `STR R0, [A]`)

✅ 2. LDR / STR (Load / Store)

> 메모리와 레지스터 간 데이터 이동

```
LDR R1, [R2]        ; R2가 가리키는 주소에서 값을 R1에 로드
STR R1, [R3]        ; R1의 값을 R3가 가리키는 주소에 저장
```

- ARM, RISC-V, MIPS 등 Load/Store 구조에서 매우 중요
- 연산은 반드시 레지스터 간에서만 수행됨

✅ 3. PUSH / POP

> 스택에 데이터를 저장하거나 꺼내는 명령어

```
PUSH R1         ; R1 값을 스택에 저장
POP R1          ; 스택에서 데이터를 꺼내 R1에 저장
```

- 함수 호출, 복귀, 지역 변수 저장 등에 사용됨
- 스택 포인터(SP)가 자동 조정됨

✅ 4. LEA (Load Effective Address)

> 메모리 주소 자체를 로드

```
LEA R1, [R2 + 4]    ; 주소값(R2 + 4)을 R1에 저장
```

- 포인터 계산, 주소 오프셋 계산에 유용

✅ 5. XCHG / SWAP (교환)

> 레지스터나 메모리 간의 데이터 교환

```
XCHG R1, R2     ; R1과 R2 값 교환
```

📐 주소 지정 방식과 데이터 이동

명령어	주소지정 방식	동작 설명
`MOV R1, #5`	즉시	상수 5를 R1에 저장
`MOV R1, R2`	레지스터	R2의 값을 R1에 복사
`LDR R1, [1000]`	직접	주소 1000의 값을 R1에 저장
`STR R1, [R2]`	간접	R1의 값을 R2가 가리키는 주소에 저장

💡 데이터 이동의 ISA별 특징

ISA	이동 명령 특징
x86	`MOV`, `PUSH`, `POP`, `LEA`, `XCHG` 등 매우 다양
ARM	`MOV`, `LDR`, `STR`, `PUSH`, `POP`, `ADR`, `LDRB` 등
RISC-V	`LW`, `SW`, `ADDI`로 간접 이동 구조
MIPS	`LW`, `SW`, `LUI`, `ORI` 등 2단계 주소 구성

⚙️ 예시 프로그램 (스택 기반 변수 저장)

```
PUSH R1        ; 현재 값 저장
MOV R1, #100   ; 변수 대입
STR R1, [R2]   ; 주소 R2에 값 저장
POP R1         ; 이전 값 복원
```

🧠 컴파일러 최적화 관점

- **레지스터 할당이 제한될 경우**, Spill → 메모리로 이동 발생
- 레지스터 이동 명령어는 대부분 한 클럭 내에서 수행 가능
- **메모리 접근**은 느리기 때문에 **이동보다는 재사용**이 중요함

📌 요약 정리

- 데이터 이동 명령어는 CPU의 **연산 이외 동작** 중 가장 기본적인 역할을 수행
- **레지스터 ↔ 레지스터**, **레지스터 ↔ 메모리**, **레지스터 ↔ 상수** 간 이동이 가능함
- 현대 CPU는 대부분 **Load/Store 구조**를 사용하여 **메모리 직접 연산이 불가능**
- 주소지정 방식과 결합해 매우 다양한 형태로 구현되며, **성능 최적화에서 가장 민감한 부분** 중 하나임

3.3 산술 및 논리 명령어

🧠 정의

산술 명령어(Arithmetic Instructions)는 정수의 덧셈, 뺄셈, 곱셈, 나눗셈 같은 수학적 연산을 수행하는 명령어이다.
논리 명령어(Logical Instructions)는 비트 단위로 AND, OR, NOT, XOR 같은 논리 연산을 수행한다.

> 이 명령어들은 **ALU(Arithmetic Logic Unit)**에서 실행된다.

🔧 산술 명령어 종류 및 설명

명령어	설명	예시
`ADD`	두 수를 더함	`ADD R1, R2, R3` → R1 ← R2 + R3
`SUB`	두 수를 뺌	`SUB R1, R2, R3` → R1 ← R2 - R3
`MUL`	곱셈	`MUL R1, R2, R3` → R1 ← R2 × R3
`DIV`	나눗셈	`DIV R1, R2, R3` → R1 ← R2 ÷ R3
`INC`	증가 (+1)	`INC R1` → R1 ← R1 + 1
`DEC`	감소 (-1)	`DEC R1` → R1 ← R1 - 1
`NEG`	부호 반전	`NEG R1` → R1 ← -R1

🔩 논리 명령어 종류 및 설명

명령어	설명	예시
`AND`	비트 AND	`AND R1, R2, R3` → R1 ← R2 & R3
`OR`	비트 OR	`OR R1, R2, R3` → R1 ← R2
`XOR`	비트 XOR	`XOR R1, R2, R3` → R1 ← R2 ^ R3
`NOT`	비트 반전	`NOT R1, R2` → R1 ← ~R2
`SHL`, `SAL`	비트 왼쪽 시프트	`SHL R1, 1` → R1의 모든 비트를 왼쪽으로 1비트 이동
`SHR`, `SAR`	비트 오른쪽 시프트	`SHR R1, 2` → 오른쪽으로 2비트 이동

> `SHL`, `SHR`: 부호 없는 정수
> `SAL`, `SAR`: 부호 있는 정수에 적절한 시프트 수행

🧠 조건 플래그에 영향

산술/논리 명령어는 CPU 내부의 **상태 플래그(Flags)**를 변경할 수 있어. 이건 조건 분기 등에 사용된다.

플래그	설명
ZF (Zero Flag)	결과가 0이면 설정
CF (Carry Flag)	자리 올림/내림이 발생하면 설정
OF (Overflow Flag)	부호 있는 오버플로우 발생 시 설정
SF (Sign Flag)	결과의 부호가 음수면 설정
PF (Parity Flag)	결과의 1 비트 수가 짝수면 설정

📐 예시: 산술 연산

```
MOV R1, #10
MOV R2, #3
ADD R3, R1, R2    ; R3 ← 10 + 3 = 13
SUB R4, R3, #5    ; R4 ← 13 - 5 = 8
INC R4            ; R4 ← 9
```

📐 예시: 논리 연산

```
MOV R1, #0b1100
MOV R2, #0b1010
AND R3, R1, R2    ; R3 ← 0b1000
OR  R4, R1, R2    ; R4 ← 0b1110
XOR R5, R1, R2    ; R5 ← 0b0110
NOT R6, R1        ; R6 ← 0b0011 (반전)
```

🧩 시프트/로테이트 연산 예시

```
MOV R1, #0b0001_0100  ; 0x14 = 20
SHL R1, #1            ; R1 ← 0b0010_1000 (40)
SHR R1, #2            ; R1 ← 0b0000_1010 (10)
```

📊 산술/논리 명령의 ISA별 명명 방식

ISA	산술 명령	논리 명령
x86	`ADD`, `SUB`, `MUL`, `DIV`	`AND`, `OR`, `XOR`, `NOT`, `SHL`, `SHR`
ARM	`ADD`, `SUB`, `MUL`	`AND`, `ORR`, `EOR`, `MVN`, `LSL`, `LSR`
MIPS	`add`, `sub`, `mult`, `div`	`and`, `or`, `xor`, `nor`, `sll`, `srl`
RISC-V	`add`, `sub`, `mul`, `div`	`and`, `or`, `xor`, `sll`, `srl`, `sra`

🧠 연산 최적화에서의 중요성

- 산술/논리 명령은 대부분 **한 클럭 내 수행** 가능
- **곱셈/나눗셈은 느릴 수 있음** → Shift로 최적화
  - `x * 2` → `x << 1`
  - `x / 2` → `x >> 1`

📌 요약 정리

- **산술 명령어**는 정수 계산 (ADD, SUB, MUL, DIV 등)을 수행
- **논리 명령어**는 비트 기반 연산 (AND, OR, XOR, NOT 등)을 수행
- 모든 연산은 **레지스터를 통해 처리**되고, **결과는 플래그에 반영**되어 분기 등에 사용됨
- Shift/Rotate 연산은 **곱셈, 나눗셈 대체 또는 마스킹 최적화**에도 자주 활용됨

3.4 분기 및 제어 명령어

🧠 정의

분기 명령어(Branch Instructions)는 **현재 실행 중인 명령의 순서를 바꿔서** 프로그램의 **흐름을 제어**하는 명령어이다.

제어 명령어(Control Instructions)는 CPU의 동작에 영향을 주거나 시스템 수준에서 제어를 수행하는 명령어를 포함한다.

> 조건문(`if`), 반복문(`for`, `while`), 함수 호출, 예외 처리 등이 모두 이 명령어들로 구현된다.

🧩 분기 명령어의 종류

종류	설명	예시
무조건 분기 (Unconditional Branch)	조건 없이 항상 점프	`JMP 0x1000`
조건 분기 (Conditional Branch)	조건(플래그)에 따라 분기	`JZ`, `JNZ`, `JE`, `JG`, `JL`
비교 후 분기	`CMP`와 함께 사용	`CMP R1, R2` → `JE label`
상대 분기 (Relative)	현재 위치 기준으로 이동	`BEQ +4`
간접 분기 (Indirect)	레지스터나 메모리 주소로 점프	`JMP [R1]`
루프 제어	반복 횟수 제어 전용	`LOOP`, `DJNZ`

✅ 무조건 분기

```
JMP target      ; 항상 target 주소로 이동
```

- 함수 호출 전후, 루프 구성에서 자주 사용

✅ 조건 분기

조건 분기는 보통 이전 연산 결과에 설정된 **플래그**(Zero, Carry, Sign 등)를 검사하여 분기를 수행함.

대표 명령어 (x86 기준)

조건	의미	명령어
== 0	결과가 0이면	`JE` or `JZ`
≠ 0	결과가 0이 아니면	`JNE` or `JNZ`
< 0	음수이면	`JS`
≥ 0	양수 또는 0이면	`JNS`
> 0 (signed)	양수이면	`JG`
< 0 (signed)	음수이면	`JL`
carry 발생	자리올림 발생	`JC`
no carry	자리올림 없음	`JNC`

> 조건 분기 전에는 반드시 `CMP` 또는 연산 명령어로 플래그가 설정되어 있어야 함

```
CMP R1, #0
JZ  label1   ; R1 == 0이면 label1으로 점프
```

✅ 루프 제어

```
MOV CX, #5
loop_start:
  ... 실행 ...
  LOOP loop_start   ; CX -= 1, 0이 아니면 점프
```

> 고정 횟수 반복이 필요한 경우 유용

✅ 간접 분기 (Indirect Branch)

```
MOV R1, #0x3000
JMP R1          ; R1이 가리키는 주소로 이동
```

- 함수 테이블, 인터럽트 벡터, 가상 머신 구현 등에 사용

📦 제어 명령어 종류

명령어	설명
`NOP`	아무 작업도 하지 않음 (No Operation)

명령어	설명
`HLT`	CPU 작동 중단
`WAIT`	외부 이벤트 대기
`RET`	서브루틴 복귀
`CALL`	서브루틴 호출
`INT`	소프트웨어 인터럽트 발생
`IRET`	인터럽트 서비스 루틴 종료 후 복귀

🧠 함수 호출 관련 명령어 흐름

✅ CALL / RET 구조

```
CALL func
; 스택에 현재 주소 저장
; func 실행
; RET → 복귀 주소 POP → 원래 코드로 복귀
```

- 서브루틴(함수) 호출을 위한 전형적인 방식
- 재귀함수, 라이브러리 호출 시 필수적

📐 조건 분기 예시

```
CMP R1, #10
JL  LESS_THAN_10   ; R1 < 10일 때 분기
JMP END

LESS_THAN_10:
MOV R2, #0

END:
```

🔁 플래그 기반 제어 구조

```
1. ALU 명령 → 플래그 설정 (Zero, Sign, Carry 등)
2. 조건 분기 명령어가 플래그 상태를 확인
3. 조건이 맞으면 지정된 주소로 PC 변경
4. 아니면 다음 명령어로 진행
```

🧠 고급 분기 처리: 현대 CPU의 분기 예측

- 분기 명령이 많아지면 **파이프라인 낭비** 발생
  → 이를 줄이기 위해 **Branch Predictor(분기 예측기)** 사용

기술	설명
Static Prediction	항상 분기한다고 가정
Dynamic Prediction	실행 이력에 따라 예측 (BTB, BHT)
Speculative Execution	예측 기반으로 먼저 실행하고 틀리면 롤백

📌 요약 정리

- **분기 명령어**는 프로그램의 흐름을 **조건/비조건적으로 변경**하며, 제어문/루프/함수 호출 등에 사용된다.
- **CMP + 조건 분기 조합**은 조건 판단의 기본 구조이며, 분기 전 플래그 설정이 필요하다.
- **제어 명령어**는 CPU 상태를 직접 제어하는 명령으로, 서브루틴 호출, 인터럽트 처리, 대기, 종료 등 시스템 수준에서 중요한 역할을 수행한다.
- 현대 CPU는 분기 명령의 예측과 파이프라인 관리를 통해 성능을 유지하고 있다.

3.5 스택 및 함수 호출 명령어

🧠 스택(Stack)이란?

스택은 **후입선출(LIFO: Last-In, First-Out)** 구조의 메모리 영역으로,
함수 호출 시 복귀 주소, 지역 변수, 레지스터 값 등을 저장하는 데 사용된다.

> 스택은 함수 호출/복귀, 인터럽트, 루프 내 변수 저장 등에서 필수적인 저장 공간이다.

🔩 스택 동작 원리

- 스택 포인터(예: `SP`, `ESP`, `R13`)가 스택의 **상단 주소를 가리킴**
- `PUSH`: 값을 스택에 저장 → SP 감소
- `POP`: 값을 스택에서 꺼냄 → SP 증가

> 대부분 **스택은 메모리의 높은 주소 → 낮은 주소 방향**으로 쌓임 (top-down)

✅ 주요 스택 관련 명령어

명령어	설명	예시
`PUSH reg`	스택에 레지스터 값 저장	`PUSH R1`
`POP reg`	스택에서 값 꺼내 레지스터에 저장	`POP R1`
`PUSH imm`	즉시 값도 스택에 저장 가능	`PUSH #10` (지원 ISA에 따라 다름)
`PUSHF` / `POPF`	플래그 레지스터 저장 및 복구	인터럽트 처리 등에서 사용

🧠 함수 호출(Subroutine Call)

📎 호출(Call)

명령어	설명
`CALL label`	현재 주소를 스택에 저장하고, label로 분기
`BL func` (ARM)	Branch with Link: `PC+4`를 LR에 저장하고 분기
`JAL` (RISC-V, MIPS)	Jump and Link: 복귀 주소를 레지스터에 저장

📎 복귀(Return)

명령어	설명
`RET`	스택에서 복귀 주소를 꺼내 원래 위치로 복귀
`BX LR` (ARM)	LR 레지스터 값으로 분기
`JR RA` (MIPS)	RA 레지스터가 가리키는 주소로 점프

📐 함수 호출 시 전형적인 흐름

```
MAIN:
  CALL FUNC       ; 복귀 주소 PUSH → FUNC 실행

FUNC:
  PUSH R1         ; R1 백업 (보존)
  MOV R1, #100    ; 연산
  POP R1          ; R1 복원
  RET             ; 스택에서 복귀 주소 POP → MAIN으로 복귀
```

> 위 흐름에서 스택은 **복귀 주소 + 레지스터 백업 + 지역 변수 저장**을 동시에 관리함

🧩 함수 호출 시 스택 프레임 구성 (Frame Pointer 기반)

```
[ HIGH 주소 ]
+---------------------+ ← SP (Push 시 ↓)
| 이전 함수의 FP       |
+---------------------+
| 복귀 주소 (RET)      |
+---------------------+
| 지역 변수 / 백업     |
+---------------------+
| 매개변수 (arg n)     |
+---------------------+ ← FP (기준점)
[ LOW 주소 ]
```

- `FP`(Frame Pointer) or `BP`(Base Pointer): 함수 내 스택의 기준점

- `SP`: 현재 스택 위치
- 함수 시작 시: `PUSH FP`, `MOV FP, SP`
- 함수 끝날 때: `MOV SP, FP`, `POP FP`

📊 ISA별 함수 호출 명령

ISA	호출 명령	복귀 명령	스택 관련
x86	`CALL`, `RET`	`RET`	`PUSH`, `POP`, `ENTER`, `LEAVE`
ARM	`BL`, `BX LR`	`BX LR`	`PUSH`, `POP`, `STMFD`, `LDMFD`
RISC-V	`JAL`, `JALR`	`JR x1`	`SW`, `LW`로 스택 직접 조작
MIPS	`JAL`, `JR RA`	`JR`	`SW`, `LW`

💡 재귀 호출 예시

```
FACTORIAL:
  CMP R0, #1
  BEQ BASE_CASE
  PUSH R0
  SUB R0, R0, #1
  CALL FACTORIAL
  POP R1
  MUL R0, R0, R1
  RET

BASE_CASE:
  MOV R0, #1
  RET
```

→ 스택이 재귀 상태를 안전하게 유지함

🔐 스택 보호와 보안 이슈

문제	설명
스택 오버플로우	너무 많은 `PUSH`, 재귀 호출 → 메모리 침범
버퍼 오버플로우	스택에 의도치 않은 값이 들어가면서 RET 주소 덮음
스택 카나리	RET 주소 위에 무결성 검증용 값 삽입 (보안 기법)
ASLR	스택/힙 등의 메모리 위치를 무작위화하여 공격 방지

📌 요약 정리

- 스택은 함수 호출 중 복귀 주소, 지역 변수, 레지스터 상태를 저장하는 메모리 구조이며,
  `PUSH`, `POP`, `CALL`, `RET` 등을 통해 관리됨
- 함수 호출 시 복귀 주소는 **스택에 저장**되며, 복귀 시 해당 주소로 되돌아간다
- 현대 CPU는 스택 기반 구조를 통해 **재귀, 인터럽트, 스레드 전환**을 안정적으로 처리함
- 스택 기반 함수 호출은 **프레임 포인터** 구조를 통해 확장성과 유지보수성을 높일 수 있다

3.6 주소지정 방식

📌 개념 정의

Treap은 이진 탐색 트리(BST)의 구조를 유지하면서,
노드마다 추가적으로 **우선순위(priority)**를 가지며 **최소/최대 힙의 조건을 만족**하는 자료구조이다.

즉,

- **키(key)**는 BST 속성 유지
  → 왼쪽 < 루트 < 오른쪽
- **우선순위(priority)**는 Heap 속성 유지
  → 부모 > 자식 (Max Heap) 또는 부모 < 자식 (Min Heap)

🧠 왜 사용하는가?

트리 성질	유지 방식
BST의 탐색 속도	키 기준 이진 정렬
균형 유지	우선순위 기반 회전
균형 트리처럼 log n 보장	랜덤 우선순위가 트리를 평형하게 만듦

🔀 Treap의 핵심: 삽입 시 회전(Rotation)

- 새 노드를 BST 규칙에 따라 삽입
- 이후 **부모보다 우선순위가 높다면 회전** (Heap 조건 유지)

예시:

```
Insert (key=42, priority=99)
→ BST에 맞는 위치에 넣고,
→ 부모보다 priority가 높으면 회전
```

📊 연산 성능 요약

연산	평균 시간복잡도	최악 시간복잡도 (낮음 확률)
삽입	O(log n)	O(n)

연산	평균 시간복잡도	최악 시간복잡도 (낮음 확률)
삭제	O(log n)	O(n)
탐색	O(log n)	O(n)

> 💡 평균적으로는 매우 균형 잡힌 트리 형성 → **랜덤 우선순위**가 핵심!

🔧 노드 구조 (C/C++ 스타일)

```
struct TreapNode {
    int key;
    int priority;
    TreapNode* left;
    TreapNode* right;
};
```

🧩 삽입 알고리즘

1. **BST 규칙**에 따라 key 위치 찾아 삽입
2. 랜덤 우선순위 부여
3. **Heap 조건 불만족 시 회전**

🔁 회전 함수

Right Rotation

```
      A                  B
     / \                / \
    B   C     →        D   A
   / \                     / \
  D   E                   E   C
```

Left Rotation

```
    A                     C
   / \                   / \
  B   C       →         A   E
     / \               / \
    D   E             B   D
```

⚙️ 삽입 예시 (C++ 스타일 Pseudocode)

```
TreapNode* insert(TreapNode* root, int key) {
    if (!root) return new_node(key);

    if (key < root->key) {
        root->left = insert(root->left, key);
        if (root->left->priority > root->priority)
```

```
            root = rotateRight(root);
    } else {
        root->right = insert(root->right, key);
        if (root->right->priority > root->priority)
            root = rotateLeft(root);
    }

    return root;
}
```

🗑️ 삭제 알고리즘

1. **BST처럼 key 찾아감**
2. 해당 노드를 루트로 끌어올린 뒤
3. **우선순위 기반 회전으로 리프 노드로 내려보내고 제거**

> 삭제는 삽입보다 조금 더 복잡하지만, 원리는 동일: **회전 + BST 규칙 + 우선순위 유지**

✨ Treap의 장점

항목	설명
구현이 간단하다	AVL, Red-Black보다 코드 간결
무작위성 기반으로 평균 성능 우수	랜덤 priority 덕분에 균형 유지
로컬성(Locality) 유지	삽입/삭제 위치 인접
회전 연산 수 제한됨	O(log n)개 이하

⚠️ 단점

문제	설명
최악 케이스 가능성 존재	랜덤성이 망가지면 균형 붕괴
우선순위 중복 방지 필요	고유하게 만들거나 tie-breaking 필요
디버깅이 어려움	key와 priority가 동시에 작용하여 구조가 비직관적일 수 있음

🧠 Treap vs AVL vs Red-Black

항목	Treap	AVL Tree	Red-Black Tree
균형 유지 방법	랜덤 우선순위	높이 균형	색상 + 속성
코드 복잡도	낮음	높음	중간
최악 시간복잡도	O(n) (확률적)	O(log n)	O(log n)
평균 시간복잡도	O(log n)	O(log n)	O(log n)

항목	Treap	AVL Tree	Red-Black Tree
삽입/삭제 회전수	평균 2 이하	최대 log n	최대 2
사용 사례	알고리즘/시뮬레이션	실시간 시스템	자바, STL 등 기본 라이브러리

🧭 응용 사례

- **균형 잡힌 BST가 필요하지만 구현이 간단해야 할 때**
- **순서 통계 트리 (Order Statistic Tree)** 구현 시
- **이중 우선순위 구조** (key + 우선순위)
- **임베디드 시스템 / 실시간 처리** (빠른 구현, 예측 가능한 속도)
- **경쟁 프로그래밍, 알고리즘 대회** 단골 주제

💬 인터뷰 질문 예시

- Treap은 왜 랜덤 우선순위를 사용하는가?
- Treap과 AVL의 시간복잡도 차이점은?
- Treap은 worst-case O(n)이 되지 않게 하려면 어떻게 할까?
- Splay Tree와 Treap의 차이점은?

📌 정리 요약

속성	내용
키	BST 조건 (왼 < 루트 < 오)
우선순위	Heap 조건 (부모 > 자식)
삽입/삭제	회전 사용, 평균 O(log n)
랜덤성	균형을 유지하게 해주는 핵심 요소
공간복잡도	O(n)
응용	균형 이진 탐색, 순서 통계 트리, 실시간 삽입/삭제

3.7 명령어 인코딩 방식

📌 1. 개요

명령어 인코딩(Instruction Encoding)은 CPU가 이해할 수 있는 형태로 **명령어를 2진수 비트열로 표현하는 방식**이다.

즉, 사람이 읽는 `ADD R1, R2, R3` 같은 어셈블리 코드를
→ **0과 1로 된 바이너리 코드**로 바꾸는 방법.

> 이는 CPU가 명령어를 **디코딩하고 제어 신호로 변환하기 위한 핵심 설계**이다.

🔢 2. 기본 구성 요소

하나의 명령어는 보통 다음과 같은 필드로 구성됨:

필드 이름	설명
Opcode	수행할 연산 종류 (예: ADD, SUB)
Operand	연산 대상 (레지스터, 메모리 등)
주소 모드	오퍼랜드가 어디 있는지 (직접, 간접 등)
조건 비트	분기/조건 수행 여부
확장 비트	긴 명령어에서 확장 용도

> 각 필드는 고정된 비트 수를 가지며, 명령어의 구조는 **명령어 세트 아키텍처(ISA)**마다 다름 (예: RISC-V, x86, ARM 등)

🔧 3. 명령어 형식 (Instruction Format)

🟧 (1) 고정 길이 포맷 (Fixed-Length)

- 명령어의 길이가 모두 같음 (예: 32비트)
- **RISC 아키텍처** (ARM, MIPS, RISC-V 등)에서 많이 사용
- 디코딩이 단순하고 빠름

```
| Opcode(6) | rs(5) | rt(5) | rd(5) | shamt(5) | funct(6) | ← MIPS R-type
```

🟥 (2) 가변 길이 포맷 (Variable-Length)

- 명령어마다 길이가 다름 (1~15바이트 등)
- **CISC 아키텍처** (x86)에서 주로 사용
- 코드 크기가 작고 유연하나, 디코딩 복잡

```
ADD EAX, EBX  → 0x01 D8
MOV AX, 1234  → B8 34 12
```

🧩 4. 주요 명령어 유형별 인코딩 방식

✅ R형 (Register Type)

- 연산 대상이 모두 레지스터
- 빠르고 간단

```
MIPS 예시: ADD R1, R2, R3

| Opcode | rs | rt | rd | shamt | funct |
```

✅ I형 (Immediate Type)

- 상수 값을 오퍼랜드로 포함

```
MIPS 예시: ADDI R1, R2, 100

| Opcode | rs | rt | Immediate |
```

✅ J형 (Jump Type)

- 분기/점프 관련 명령어

```
MIPS 예시: J target

| Opcode | Target address |
```

🧮 5. 실제 예시 (MIPS 기준)

```
add $t0, $t1, $t2
```

필드	값 (2진수)
opcode	000000
rs = $t1	01001
rt = $t2	01010
rd = $t0	01000
shamt	00000
funct	100000

전체 명령어:

```
000000 01001 01010 01000 00000 100000
```

32비트 = `0x012A4020`

🧠 6. 명령어 인코딩 설계 시 고려사항

고려 요소	설명
비트 효율성	가능한 작은 비트 수로 많은 명령 표현
디코딩 속도	고정형이 유리 (RISC)
유연성	CISC 구조에 적합
확장성	ISA 확장 대비

고려 요소	설명
정렬 규칙	명령어 정렬, 패딩 삽입 등
파이프라인 설계 용이성	고정 포맷이 유리

🔐 7. ISA별 예시

ISA	인코딩 방식 특징
x86	가변 길이, 복잡, CISC
MIPS	고정 32비트, 단순, RISC
ARM	16/32비트 (Thumb 포함)
RISC-V	고정 길이, 모듈형 설계

💬 인터뷰/시험 문제 예시

- MIPS에서 ADD 명령어의 32비트 바이너리 표현은?
- CISC 구조가 인코딩에 유리한 이유는?
- Immediate 값을 가지는 명령어는 어떤 포맷을 쓰나?
- 인코딩 방식이 CPU 성능에 미치는 영향은?

✅ 정리 요약

항목	요약
목적	명령어를 CPU가 이해할 수 있는 이진수로 인코딩
구성 요소	opcode, operand, 주소 모드, immediate 등
형식 종류	고정 길이(RISC), 가변 길이(CISC)
대표 ISA	x86, MIPS, ARM, RISC-V
성능 고려사항	속도 vs 유연성, 디코딩 용이성