10. 마이크로프로세서 설계 및 시뮬레이션

10.1 기본 CPU 설계 (8비트/16비트)

🧠 기본 CPU란?

기본 CPU 설계는 **최소한의 하드웨어 자원으로도 명령어 실행이 가능하도록 구성된 중앙처리장치**이다.
대개 다음 요소들을 포함하고:

구성 요소	기능
레지스터(Register)	임시 데이터 저장
ALU	산술/논리 연산
Program Counter(PC)	명령어 위치 추적
메모리 인터페이스	RAM/ROM 접근
제어장치(Control Unit)	전체 동작 제어
버스 구조	데이터/주소 전달 경로

📏 8비트 vs 16비트 CPU란?

항목	8비트 CPU	16비트 CPU
데이터 폭	한 번에 8비트 처리	16비트 처리
레지스터 크기	8비트	16비트
주소 버스 폭	보통 16비트 (64KB 주소공간)	최대 64KB 이상
연산 능력	단순, 저속	더 빠르고 정밀
대표	6502, Z80, 8051	8086, MSP430, HCS12

🧱 기본 CPU 구성 블록 다이어그램

```
    +--------------+
    | Program      | ← ROM
    | Counter (PC) |
    +------+-------+
           |
           v
    +--------------+
    | Instruction  | ← 명령어 메모리
    | Register (IR)|
    +------+-------+
           |
           v
```

```
    +--------------+       +---------------+
    | Control Unit | ----> |   Control     |
    +--------------+       |   Signals     |
           |                      |
           v                      v
    +------+--------+      +------------------+
    | ALU           | <--> | Register File    |
    +---------------+      +------------------+
           |
           v
    +--------------+
    | Data Memory  | ← RAM
    +--------------+
```

🔁 명령어 실행 흐름 (Fetch → Decode → Execute)

1. **Fetch**: PC가 가리키는 메모리에서 명령어 읽음
2. **Decode**: 명령어를 해석하고 제어신호 생성
3. **Execute**: ALU가 연산 or 메모리 접근
4. **Write-back**: 결과를 레지스터나 메모리에 저장
5. **PC 갱신**: 다음 명령어로 이동

🔢 명령어 집합 예 (8비트 기준)

명령	설명	포맷
`LD Rn, imm`	즉시값 로드	`0001 rrrr dddd dddd`
`ADD R1, R2`	덧셈	`0010 rrrr ssss xxxx`
`SUB R1, R2`	뺄셈	`0011 rrrr ssss xxxx`
`JMP addr`	무조건 점프	`0100 aaaa aaaa aaaa`
`BZ addr`	0이면 점프	`0101 aaaa aaaa aaaa`
`OUT R1`	포트로 출력	`0110 rrrr xxxx xxxx`

🧮 ALU 기본 구성

- 입력: A, B (레지스터 or 메모리 데이터)
- 출력: 결과 + 상태 플래그(Z, N, C, V)
- 연산: `ADD`, `SUB`, `AND`, `OR`, `XOR`, `NOT`, `SHL`, `SHR`

📦 레지스터 구성 예

이름	설명
R0~R7	범용 레지스터 (8개, 8비트 또는 16비트)
ACC	누산기 (ALU 기본 연산에 사용)
PC	프로그램 카운터
IR	명령어 레지스터
SP	스택 포인터
SR/FLAGS	상태 레지스터 (Z, C, N, V 플래그)

🔌 데이터 버스와 제어 신호

신호	설명
Data Bus	ALU ↔ 레지스터 ↔ 메모리 간 데이터 전달
Address Bus	PC 또는 마이크로제어기에서 RAM/ROM 주소 지정
Read/Write	메모리 접근 시 방향 제어
ALU_OP	연산 종류 선택
LOAD/STORE	레지스터 ↔ 메모리 전송 시 사용

🧪 설계 예시: 8비트 RISC CPU

- **레지스터 파일**: R0~R7 (8개, 8비트)
- **ALU**: 1 사이클 동작
- **ROM**: 명령어 저장 (256바이트)
- **RAM**: 데이터 저장용 (128바이트)
- **PC**: 8비트 (256개 명령어 공간)
- **제어기**: FSM 기반 하드와이어드 방식 or 마이크로코드 기반

🛠️ 설계 시 고려사항

항목	설명
명령어 폭 결정	고정형(예: 16비트) or 가변형
파이프라인 여부	단순 구조는 일반적으로 비파이프라인
제어 방식	FSM 하드와이어드 / 마이크로코드 선택 가능
확장성	인터럽트, I/O 포트, 스택 구조 필요 여부

항목	설명
디버깅용 인터페이스	PC, IR 출력 가능하게 설계

📌 요약 정리

항목	8비트 CPU	16비트 CPU
데이터 폭	1바이트	2바이트
명령어 크기	보통 8~16비트	보통 16~32비트
레지스터	8비트 폭, 소형	16비트 폭, 확장
ALU 구성	기본 연산	정밀 연산까지 포함
설계 난이도	낮음	중간 수준
응용	교육용, 임베디드	고급 제어기, RTOS 대상

10.2 Verilog/VHDL을 이용한 프로세서 구현

🧠 왜 HDL(Hardware Description Language)로 CPU를 구현하는가?

HDL(Verilog 또는 VHDL)은 **하드웨어 동작을 논리적으로 표현**하는 언어로,
디지털 회로(특히 CPU)를 **설계, 시뮬레이션, 합성(Synthesis)**할 수 있게 해준다.

> 명령어 해석, ALU 연산, 메모리 접근, 제어 신호 발생 등을
> **RTL(Register Transfer Level)**로 기술하여 실제 FPGA나 ASIC 구현이 가능해짐.

📦 기본 CPU 설계 구성 (모듈 단위)

```
Top-Level CPU
 ├── Program Counter (PC)
 ├── Instruction Memory (ROM)
 ├── Instruction Decoder
 ├── Register File
 ├── ALU
 ├── Data Memory (RAM)
 └── Control Unit
```

🔧 Verilog 모듈 예제 구조

```
module cpu (
    input wire clk,
    input wire reset
);
    // 내부 신호 선언
    wire [7:0] instruction;
    wire [7:0] reg_a, reg_b, alu_out;
    wire [2:0] alu_op;
```

```
    // 인스턴스 연결
    program_counter pc(...);
    instruction_memory imem(...);
    decoder decode(...);
    register_file rf(...);
    alu my_alu(...);
    control_unit cu(...);
endmodule
```

🧮 주요 모듈별 Verilog 구현 예시

✅ 1. ALU 예제 (8비트)

```
module alu (
    input [7:0] a, b,
    input [2:0] op,
    output reg [7:0] result,
    output zero
);
    always @(*) begin
        case (op)
            3'b000: result = a + b;
            3'b001: result = a - b;
            3'b010: result = a & b;
            3'b011: result = a | b;
            3'b100: result = ~a;
            default: result = 8'b0;
        endcase
    end

    assign zero = (result == 8'b0);
endmodule
```

✅ 2. Register File (8개 레지스터, 8비트)

```
module register_file (
    input clk,
    input [2:0] read_addr1, read_addr2, write_addr,
    input write_en,
    input [7:0] write_data,
    output [7:0] read_data1, read_data2
);
    reg [7:0] regs [0:7];

    always @(posedge clk)
        if (write_en)
            regs[write_addr] <= write_data;

    assign read_data1 = regs[read_addr1];
    assign read_data2 = regs[read_addr2];
```

```
endmodule
```

✅ 3. Program Counter

```
module program_counter (
    input clk,
    input reset,
    input [7:0] next_pc,
    output reg [7:0] pc
);
    always @(posedge clk or posedge reset)
        if (reset) pc <= 8'b0;
        else pc <= next_pc;
endmodule
```

✅ 4. 간단한 Control Unit (FSM 기반)

```
module control_unit (
    input [7:0] opcode,
    output reg alu_src,
    output reg reg_write,
    output reg mem_read,
    output reg mem_write,
    output reg [2:0] alu_op
);
    always @(*) begin
        case (opcode[7:4])
            4'b0001: begin // ADD
                alu_op = 3'b000;
                alu_src = 0;
                reg_write = 1;
            end
            4'b0010: begin // SUB
                alu_op = 3'b001;
                alu_src = 0;
                reg_write = 1;
            end
            // ...
            default: begin
                alu_op = 3'b000;
                reg_write = 0;
            end
        endcase
    end
endmodule
```

🧪 테스트벤치 예시

```
module cpu_tb;
    reg clk = 0;
    reg reset = 1;

    cpu uut (
        .clk(clk),
        .reset(reset)
    );

    always #5 clk = ~clk;

    initial begin
        #10 reset = 0;
        #100 $stop;
    end
endmodule
```

🧰 시뮬레이션 & 합성 흐름

단계	도구 예시
작성	Verilog/VHDL 텍스트 작성
시뮬레이션	ModelSim, Icarus Verilog, Vivado Simulation
검증	Testbench 통해 기능 확인
합성(Synthesis)	Vivado, Quartus, Design Compiler 등
타이밍 분석	STA 도구를 통해 동작 속도 검증
배치 배선	FPGA/ASIC 물리 구현 단계
다운로드	FPGA에 Bitstream 다운로드

📌 요약 정리

항목	설명
HDL 사용 목적	CPU 하드웨어 논리를 디지털 논리로 기술
모듈 구성	PC, ALU, RegFile, RAM, 제어기 등으로 분리
RTL 설계	Clock, 상태, 제어 신호 흐름 중심
시뮬레이션 필수	검증 없이는 회로 동작 보장 어려움
합성 준비	타이밍, 자원 사용량, 최적화 고려 필요

10.3 FSM 기반 제어기 설계

> 명령어 실행 흐름을 상태(state)로 정의하여 제어한다.

🧠 FSM이란?

FSM(Finite State Machine)은 시스템의 동작을
유한한 상태들의 집합과 상태 전이 규칙으로 모델링한 제어 방식이다.

> 하나의 CPU 명령어 실행을 **여러 상태(State)**로 나누고
> 각각의 상태에서 **어떤 제어 신호를 낼지 결정**하는 방식으로 제어기를 구성함.

🔁 FSM 제어기의 특징

항목	설명
정형적	상태 전이표 또는 상태도에 따라 제어
단계별 실행	명령어 하나를 여러 클럭에 걸쳐 처리
명확한 타이밍 제어	각 상태마다 고유한 동작 수행 가능
하드와이어드 제어기와 대비	마이크로코드 대신 상태 머신으로 직접 제어 구현

🧩 Moore vs Mealy FSM

항목	Moore FSM	Mealy FSM
출력	상태에만 의존	상태 + 입력에 의존
반응 속도	1클럭 느림	반응 빠름
회로 복잡도	간단함	복잡함
제어기 활용	CPU 제어기, 순차 로직	UART 등 실시간 반응 요구 장치

✅ CPU 제어기에서는 일반적으로 **Moore FSM** 구조가 널리 사용됨.

📦 FSM 기반 CPU 제어기 예시

🎯 목표: 명령어 실행 흐름을 상태로 나눈다.

예: `ADD R1, R2` 명령어

상태	동작
S0	명령어 fetch (PC → MAR, read from memory)
S1	명령어 decode (IR ← M[PC])
S2	피연산자 읽기 (레지스터 R1, R2 → ALU 입력)
S3	연산 수행 (ALU: R1 + R2)

상태	동작
S4	결과 저장 (결과 → R1), PC+1
S5	다음 명령어 준비 (상태 초기화)

📈 상태 전이 다이어그램 (간단 FSM)

```
[S0] -- fetch --> [S1] -- decode --> [S2] -- read operands -->
[S3] -- execute --> [S4] -- writeback --> [S5] --→ [S0]
```

🔧 FSM 제어기 Verilog 구현 예시 (Moore 타입)

```
module control_unit (
    input clk, reset,
    input [7:0] opcode,
    output reg [2:0] alu_op,
    output reg pc_write, ir_write, reg_write,
    output reg [2:0] state
);

    // 상태 정의
    localparam S0 = 3'd0,
               S1 = 3'd1,
               S2 = 3'd2,
               S3 = 3'd3,
               S4 = 3'd4;

    // 상태 변수
    reg [2:0] next_state;

    // 상태 전이
    always @(posedge clk or posedge reset) begin
        if (reset) state <= S0;
        else state <= next_state;
    end

    // 다음 상태 결정
    always @(*) begin
        case (state)
            S0: next_state = S1;
            S1: next_state = S2;
            S2: next_state = S3;
            S3: next_state = S4;
            S4: next_state = S0;
            default: next_state = S0;
        endcase
    end

    // 상태별 제어 신호 출력
    always @(*) begin
```

```
        // 기본값
        pc_write = 0; ir_write = 0;
        reg_write = 0; alu_op = 3'b000;

        case (state)
            S0: begin // 명령어 fetch
                pc_write = 1;
                ir_write = 1;
            end
            S2: begin // ALU 입력 준비
                alu_op = 3'b000; // ADD
            end
            S4: begin // 결과 write-back
                reg_write = 1;
            end
        endcase
    end
endmodule
```

🧠 FSM 제어기의 장점

장점	설명
구현이 직관적	명령어 흐름 = 상태 흐름
모듈화 용이	명령어 유형마다 상태 분기 가능
디버깅이 쉬움	어느 상태에서 멈췄는지 쉽게 확인 가능
소규모 CPU에 적합	마이크로코드 없이도 제어 가능

🧪 실전: FSM 기반 8비트 CPU 설계 흐름

1. **명령어 집합 정의**
    - ex) `LD, ADD, SUB, JMP, BZ`
2. **명령어별 동작 분해**
    - 각 명령어를 몇 단계로 나눌 것인지 결정
3. **상태 정의 및 다이어그램 구성**
4. **제어 신호 결정**
    - 각 상태에서 어떤 제어선이 켜지는지
5. **Verilog/VHDL 코드 작성**
6. **시뮬레이션 및 타이밍 검증**

📌 요약 정리

항목	설명
FSM 제어기	상태 기반으로 명령어 실행 흐름을 구성
Moore FSM	상태에만 의존한 출력 (CPU에 적합)
상태 다이어그램	명령어 처리 단계를 시각화
Verilog 구현	상태 변수, 전이 로직, 출력 로직 분리
활용 범위	단순/중형 CPU, 컨트롤러, 시리얼 장치 등

10.4 타이밍 분석 및 클럭 스케줄링

> CPU 동작의 안전성과 속도를 결정하는 "시간"의 문제

🧠 타이밍 분석이란?

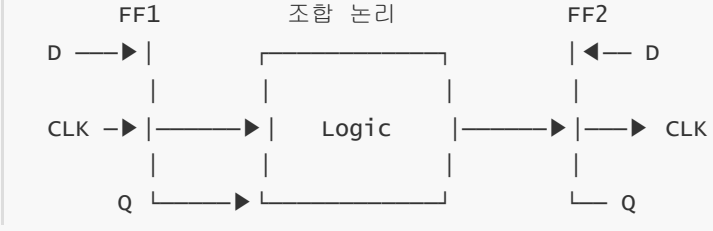
타이밍 분석(Timing Analysis)은
디지털 회로가 **클럭 사이클 내에 올바르게 동작하는지를 검증**하는 과정이다.

> 각 조합논리 경로에서 신호가 도달하는 **최대 지연 시간**을 계산해서
> 시스템이 안정적으로 작동하는 **최소 클럭 주기(clock period)**를 결정함.

📦 기본 타이밍 요소

항목	설명
클럭 주기 (Tclock)	한 클럭 사이클의 시간
프로파게이션 지연 (tpd)	게이트/회로를 통과하는 데 걸리는 시간
셋업 타임 (Tsetup)	D플립플롭이 데이터 샘플링 전에 안정되어야 하는 시간
홀드 타임 (Thold)	클럭 이후에도 데이터를 유지해야 하는 최소 시간
클럭 스큐 (Clock Skew)	동일 클럭이 도착하는 시점의 차이
크리티컬 패스 (Critical Path)	회로 내 가장 긴 지연 시간 경로

🔁 동기식 회로 타이밍 모델



조건:

데이터가 FF1에서 Q로 나가서 Logic을 통과하여
FF2의 셋업 타임 전까지 도달해야 함

📏 최소 클럭 주기 계산 공식

```
Tclock ≥ Tsetup + Tlogic_max + Tskew
```

용어	설명
Tsetup	플립플롭의 셋업 타임
Tlogic_max	조합 논리에서의 최장 지연 경로
Tskew	클럭 도달 시점 차이

> 이 값을 줄이면 고속 동작 가능 → **파이프라인, 최적화, 리타이밍, 멀티사이클 경로 처리 등**이 필요함

📉 크리티컬 패스 찾기

크리티컬 패스(Critical Path)는
회로 내 가장 오래 걸리는 데이터 경로이며, 전체 시스템의 **최대 속도를 제한**함.

> 타이밍 분석 도구(예: Vivado, Quartus STA Report)를 통해 자동 분석 가능

🔁 멀티사이클 경로 처리

어떤 연산은 1클럭에 끝나지 않고, 2~3클럭이 걸릴 수 있다.
→ 해당 경로는 타이밍 제약에서 **느슨하게 처리 가능**

```
set_multicycle_path -from ALU_STAGE -to REG_OUT -setup 2
```

> 클럭당 1개씩만 동작한다고 가정하면 비효율 → 멀티사이클 경로로 선언하여 속도 향상 가능

🔧 클럭 스큐 (Clock Skew) 문제

유형	설명
Positive Skew	클럭이 목적지 FF에 더 늦게 도착함
Negative Skew	클럭이 목적지 FF에 더 빨리 도착함
심각한 경우	타이밍 위반, 셋업/홀드 조건 미달

해결 방법:

- 균형 잡힌 클럭 트리 (H-tree 구조)
- Clock Buffer 삽입
- **Clock Domain Crossing (CDC)** 시엔 동기화 회로 삽입

🕐 클럭 도메인과 CDC (Clock Domain Crossing)

현대 CPU는 **여러 클럭 도메인**을 갖기도 한다:

예시	설명
CPU Core	고속 클럭 (1GHz 이상)
Memory	중속 (~400MHz)
Peripherals	저속 (50MHz 등)

> 다른 클럭 도메인 간 신호 전달은 **동기화 회로(2-stage FF)**나 **FIFO + 핸드셰이크**로 처리해야 함

📈 클럭 스케줄링이란?

클럭 스케줄링은 다음을 포함한다:

1. **클럭 도메인 간 타이밍 계획**
2. **연산 블록별 클럭 정렬**
3. **슬로우 패스에는 멀티사이클 지정**
4. **스케줄에 따라 최적 클럭 생성기 구성 (PLL, MMCM 등)**

🛠️ 타이밍 분석 도구 예

도구	설명
Vivado STA (Xilinx)	Synthesis → Timing Report
Quartus TimeQuest (Intel FPGA)	정확한 타이밍 그래프 제공
Synopsys PrimeTime	ASIC용 대표 STA 도구
STA Constraints File (SDC)	`create_clock`, `set_input_delay`, `set_output_delay` 등으로 설정

📌 요약 정리

항목	설명
타이밍 분석	경로 지연 시간 기반으로 클럭 안정성 검증
셋업 조건	데이터가 클럭 전에 도달해야 함
홀드 조건	데이터가 클럭 이후에도 유지되어야 함
크리티컬 패스	시스템 전체의 속도 병목 요소
멀티사이클 경로	느린 연산의 타이밍 제약 완화
클럭 스큐	클럭 도달 시점 불균형 → 스킵/오류 가능성
CDC 처리	동기화 회로, FIFO 등 필요

10.5 명령어 시뮬레이터 제작

> 실제 CPU처럼 동작하는 "소프트웨어 CPU"

🧠 ISS란?

명령어 시뮬레이터(Instruction Set Simulator)는
실제 하드웨어 없이도 **CPU의 명령어 집합(ISA)을 해석하고 실행해보는 소프트웨어 모델**이다.

> 개발 목적:

- CPU 설계 전 시뮬레이션
- 컴파일러 검증
- 운영체제 이식 실험
- 소프트웨어 디버깅

📦 주요 구성 요소

구성 요소	역할
레지스터 파일	범용 레지스터와 특수 레지스터 저장
메모리 모델	가상 RAM/ROM 공간
명령어 디코더	명령어의 Opcode 분석
실행기(Executor)	명령어 의미에 따라 실제 효과 반영
PC(Program Counter)	다음 실행 명령어 주소 추적
플래그 레지스터	Zero, Carry, Negative 등 상태 추적
I/O 모델 (선택)	입출력 포트 시뮬레이션

🔁 실행 흐름

```
while (!halted) {
    instr = memory[PC];
    opcode, operands = decode(instr);
    execute(opcode, operands);
    PC = update_PC(opcode, operands);
}
```

🔢 예시 ISA 정의 (8비트 간단 ISA)

명령	포맷	의미
`LD Rn, imm`	`0001 nnnn dddddddd`	Rn ← imm
`ADD Rn, Rm`	`0010 nnnn mmmm0000`	Rn ← Rn + Rm

명령	포맷	의미
`SUB Rn, Rm`	`0011 nnnn mmmm0000`	Rn ← Rn - Rm
`JMP addr`	`0100 aaaaaaaa`	PC ← addr
`BZ addr`	`0101 aaaaaaaa`	if Zero == 1, PC ← addr

🛠️ C 기반 시뮬레이터 구현 예시

✅ 레지스터/메모리 모델

```c
#define REG_COUNT 8
#define MEM_SIZE  256

uint8_t reg[REG_COUNT];
uint8_t memory[MEM_SIZE];
uint8_t pc = 0;
uint8_t zero_flag = 0;
```

✅ 디코딩 및 실행

```c
void execute_instruction() {
    uint8_t instr = memory[pc++];

    uint8_t opcode = (instr & 0xF0) >> 4;
    uint8_t operand = instr & 0x0F;

    switch(opcode) {
        case 0x1: // LD Rx, imm
            reg[operand] = memory[pc++];
            break;

        case 0x2: // ADD Rx, Ry
            reg[operand] += reg[memory[pc++]];
            zero_flag = (reg[operand] == 0);
            break;

        case 0x3: // SUB Rx, Ry
            reg[operand] -= reg[memory[pc++]];
            zero_flag = (reg[operand] == 0);
            break;

        case 0x4: // JMP addr
            pc = memory[pc];
            break;

        case 0x5: // BZ addr
            if (zero_flag)
                pc = memory[pc];
            else
                pc++;
```

```
            break;
        }
}
```

✅ 메인 루프

```
void run() {
    while (1) {
        execute_instruction();
    }
}
```

🧪 테스트 프로그램 예

```
// memory[0] = LD R0, 0x05
// memory[2] = LD R1, 0x03
// memory[4] = ADD R0, R1
memory[0] = 0x10; memory[1] = 0x05;
memory[2] = 0x11; memory[3] = 0x03;
memory[4] = 0x21; memory[5] = 0x00; // ADD R1, R0
```

→ 실행 결과: `R1 = 8`

📈 기능 확장 아이디어

확장 요소	설명
스택 구조 구현	PUSH/POP, SP 레지스터
I/O 포트 모델링	`IN`, `OUT` 명령
메모리 맵 I/O	주소에 따라 I/O 동작 연결
인터럽트 처리	인터럽트 벡터, ISR 진입
파이프라인 시뮬레이션	단계별 레지스터 유지 구조
GUI 디버거	레지스터/메모리 뷰 제공

🛠️ 디버깅 및 검증 방법

도구	활용 방법
`printf`	PC, 명령어, 레지스터 상태 출력
Trace 파일 생성	실행 로그 기록 및 비교
Unit Test 작성	명령어 단위 검증
Instruction Trace	명령어별 사이클, 플래그 확인용

📌 요약 정리

항목	설명
ISS	소프트웨어로 구현한 명령어 해석기
주요 구성	PC, 메모리, 레지스터, 디코더, 실행기
기능	명령어 해석, 실행, 상태 유지
응용	CPU 테스트, 툴체인 개발, 교육용
확장	파이프라인 모델, 인터럽트, 캐시 모델링 가능