SoC_Peri_Lecture04 Clock과 Reset & GPIO

Agenda

1. System Clock and Reset 이해 및 시뮬레이션
   - Clock 개념이해
   - Clock generator, Counter를 이용한 모듈 시뮬레이션
   - Reset의 개념
   - Reset 시뮬레이션
2. SoC 설계에서 GPIO 입출력 회로 구조 이해 및 시뮬레이션
   - Logic Level, CMOS Transistors, Push–Pull Output (Digital), Tri-state Output, Open-collector/Open-drain Output
3. GPIO에 대한 이해 및 시뮬레이션
   - SoC에서 사용되는 GPIO 구성요소
   - GPIO제어 구조
   - GPIO 레지스터 맵 구조

- Q&A

SoC 기본 System Clock 이해하기 - 클록(clock)의 기본개념

- 디지털 전자 시스템에서 클록은 시스템의 상태 업데이트를 위한 기준 신호, 일종의 "심장 박동" 역할
- 클록 신호는 일반적으로 오실레이터(예: 수정 발진기)에 의해 생성되며, 주기적으로 높은 상태와 낮은 상태 간을 빠르게 전환하여 메모리와 로직이 동시에 업데이트되도록 합니다. 이를 통해 경합 조건(race condition)을 방지하고 안정적인 회로 동작을 보장합니다

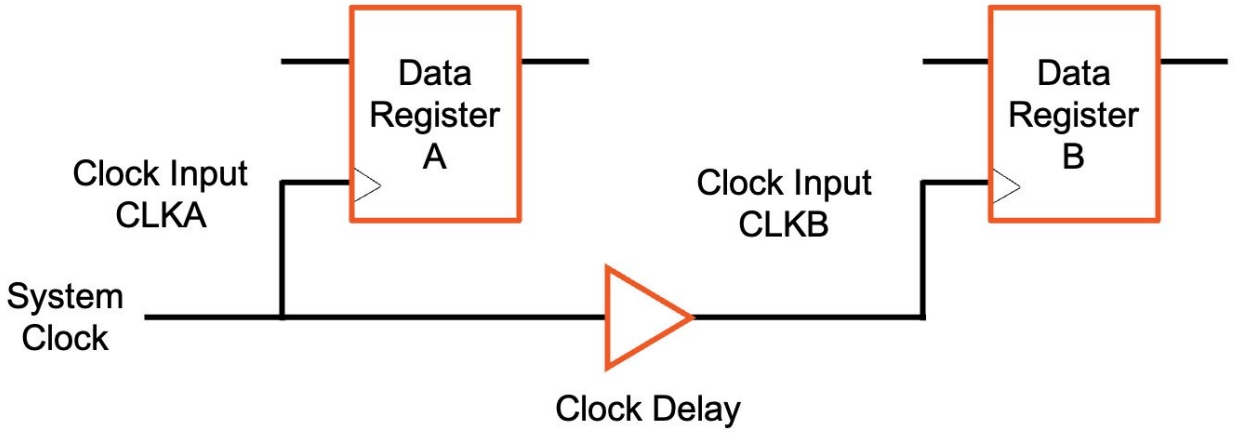
SoC 기본 System Clock 개념

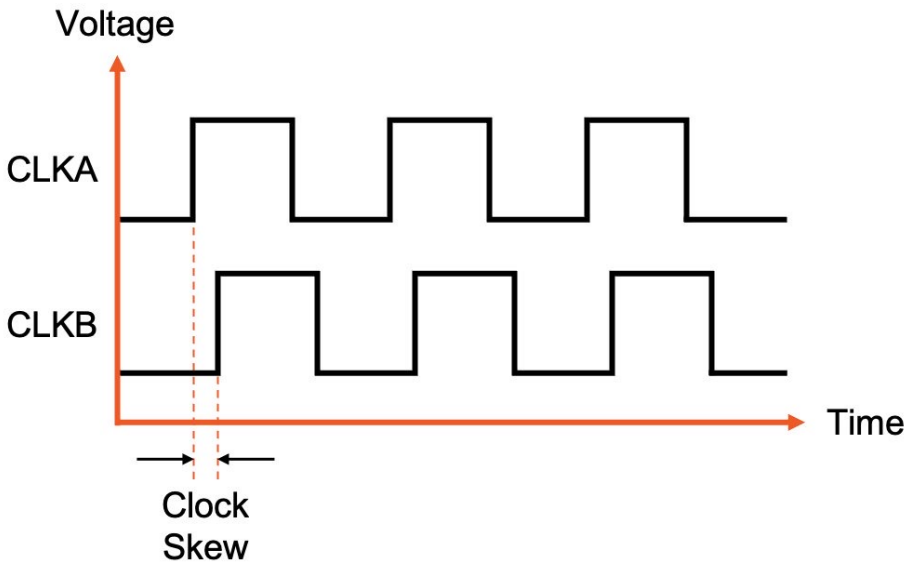
항목	설명
클록 신호의 역할	디지털 시스템에서 모든 구성 요소(CPU, 메모리 등)의 동작을 동기화하며 데이터 전송과 명령 실행의 기준 신호 제공.
클록 트리 설계(CTS)	클록 신호를 칩 내부의 모든 플립플롭과 레지스터에 균일하게 분배하도록 트리 구조를 설계. 지연 최소화와 신호 무결성 유지가 핵심.
클록 관리 유닛(CMU)	SoC에서 클록 소스와 각 모듈 간 클록 신호를 제어하며, 클록 온/오프 및 주파수 조정을 담당.
지터 관리	클록 신호의 시간적 변동을 최소화하여 정확한 타이밍 제공. PLL(Phase Locked Loop)과 같은 기술 활용.
저전력 설계	필요하지 않은 블록에 클록을 차단(Clock Gating)하여 전력 소비를 줄이고 시스템 효율성을 높임3.
자동화된 설계 도구	복잡한 SoC 설계를 위해 클록 트리 생성, 검증, 최적화를 자동화하는 EDA 도구 활용.

Clock 설계시 주의 사항

항목	설명	고려사항
클록 소스 선택	클록 신호를 생성하기 위해 사용되는 소스 (XTAL, PLL, RC 오실레이터 등)	주파수 정확도, 전력 소비, 비용, 안정성
클록 트리 설계	클록 신호를 SoC 내부의 다양한 블록으로 분배하는 구조	균일한 시간 지연, 낮은 지터(Jitter), 신호 무결성 유지
클록 도메인 분리	서로 다른 주파수로 동작하는 클록 도메인을 분리하여 설계	교차 클록 도메인(CCD) 문제 해결, 동기화 회로 삽입
PLL(Phase Locked Loop)	다양한 주파수의 고정밀 클록을 생성하기 위한 회로	주파수 변환 범위, 안정성, 지터 최소화
저전력 모드 지원	저전력 상태에서 클록 속도를 낮추거나 비활성화하여 전력 소비를 줄임	전환 속도, 복구 시간
자동화된 클록 생성	복잡한 SoC 설계를 위해 자동화된 알고리즘을 통해 클록 구조를 생성	RTL 코드 생성, 검증용 어설션(assertion), 제약 조건 관리
신호 무결성 유지	클록 신호의 품질을 유지하여 시스템의 안정적인 동작 보장	EMI(전자기 간섭) 최소화, PCB 레이아웃 최적화
지터(Jitter) 관리	클록 신호의 시간적 변동을 최소화하여 정확한 타이밍 제공	PLL 설계 최적화, 고품질 오실레이터 사용
클록 게이팅 (Clock Gating)	필요하지 않은 블록에 클록을 차단하여 전력 소비를 줄임	동적 전력 관리 (Dynamic Power Management), 타이밍 분석
테스트 및 디버깅	클록 시스템의 기능 검증 및 오류 수정	프로브 삽입, 시뮬레이션 및 하드웨어 디버깅 도구 활용

SoC 기본 System Clock 설계 시 주의할 점

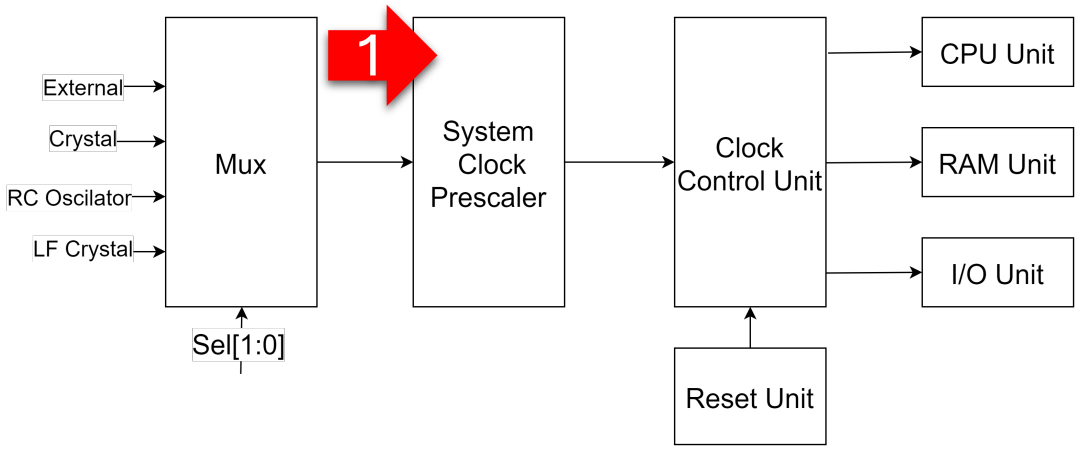




물리적 거리때문에 clock오차가 발생, Clock Skew, buffer를 추가해 맞춘다

SoC 기본 System Clock 실습1-1

- SoC에서의 클록 생성 및 분배
- 다음을 설계해 보시오



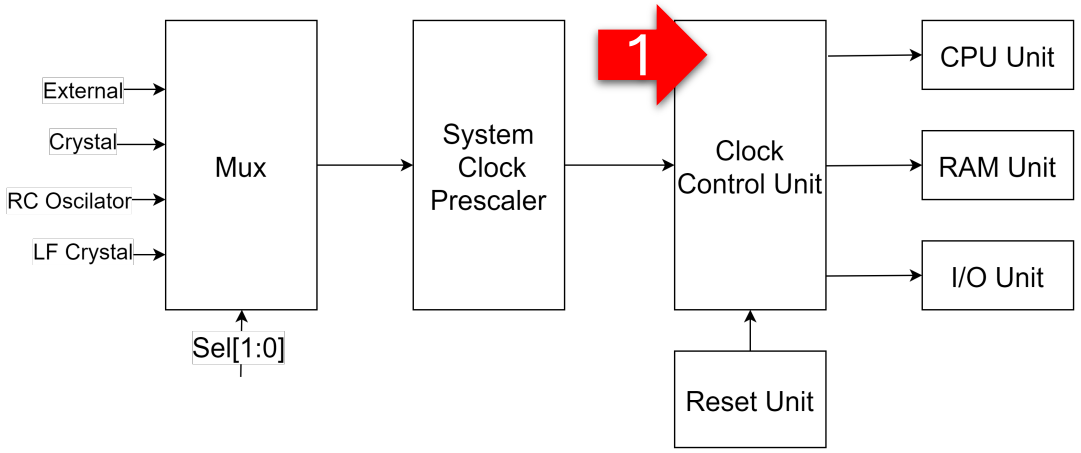

```
1   `timescale 1ns/1ns
2   module SystemClockPrescaler(
3       input wire clk_in,          // 입력 클럭
4       input wire reset,           // 리셋 신호
5       input wire [3:0] clkps,     // Prescaler 설정 값 (CLKPS[3:0])
6       output reg clk_out          // 출력 클럭
7   );
8
9       reg [7:0] prescale_counter; // Prescaler 카운터
10
11      always @(posedge clk_in or posedge reset) begin
12          if (reset) begin
13              prescale_counter <= 8'b0; // 리셋 시 카운터 초기화
14              clk_out <= 1'b0;          // 출력 클럭 초기화
15          end else begin
16              if (prescale_counter == (2**clkps - 1)) begin
17                  prescale_counter <= 8'b0; // 카운터 리셋
18                  clk_out <= ~clk_out;      // 출력 클럭 반전
19              end else begin
20                  prescale_counter <= prescale_counter + 1; // 카운터 증가
21              end
22          end
23      end
24  endmodule
```

2

3

SoC 기본 System Clock 실습1-2

- SoC에서의 클록 생성 및 분배
- 다음을 설계해 보시오




```
26  module ClockControlUnit(
27      input wire clk_in,           // 입력 클럭
28      input wire reset,            // 리셋 신호
29      input wire [3:0] prescaler, // Prescaler 설정 값
30      output wire cpu_clk,         // CPU 클럭 출력
31      output wire io_clk           // I/O 클럭 출력
32  );
33      wire divided_clk;
34
35      // Prescaler 모듈 인스턴스화
36      SystemClockPrescaler prescaler_unit (
37          .clk_in(clk_in),
38          .reset(reset),
39          .clkps(prescaler),
40          .clk_out(divided_clk)
41      );
42
43      assign cpu_clk = divided_clk;   // CPU 클럭에 분주된 클럭 연결
44      assign io_clk = divided_clk;   // I/O 클럭에 분주된 클럭 연결
45
46  endmodule
```

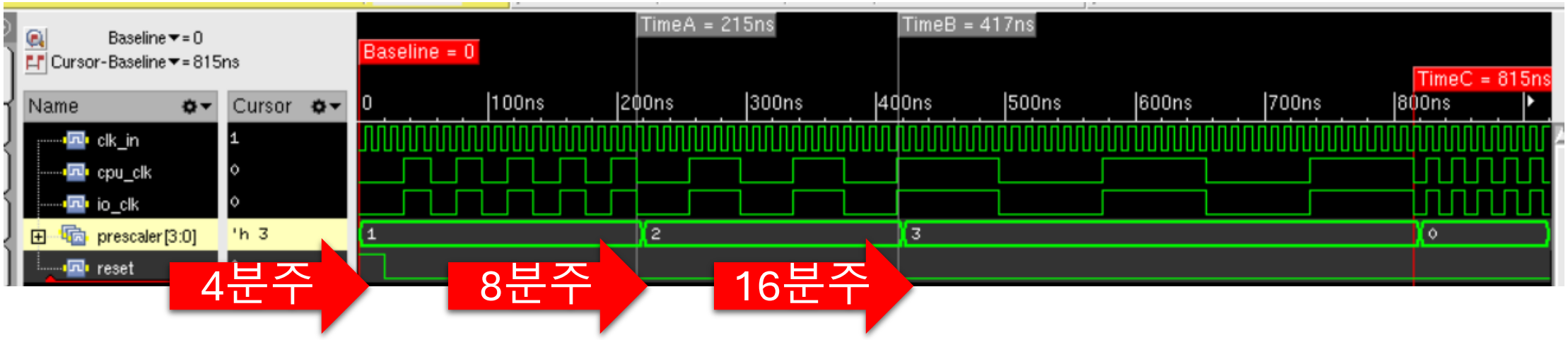
SoC 기본 System Clock 실습1 시뮬레이션

- 시뮬레이션 하십시오

```
1  `timescale 1ns/1ns
2
3  module tb_ClockControlUnit;
4
5      // 테스트벤치 신호 선언
6      reg clk_in;
7      reg reset;
8      reg [3:0] prescaler;
9
10     wire cpu_clk;
11     wire io_clk;
12
13     // ClockControlUnit 모듈 인스턴스화
14     ClockControlUnit uut (
15         .clk_in(clk_in),
16         .reset(reset),
17         .prescaler(prescaler),
18         .cpu_clk(cpu_clk),
19         .io_clk(io_clk)
20     );
21
22     // 입력 클럭 생성 (100MHz, 주기 10ns)
23     initial begin
24         clk_in = 0;
25         forever #5 clk_in = ~clk_in; // 5ns마다 반전 (100MHz)
26     end
```

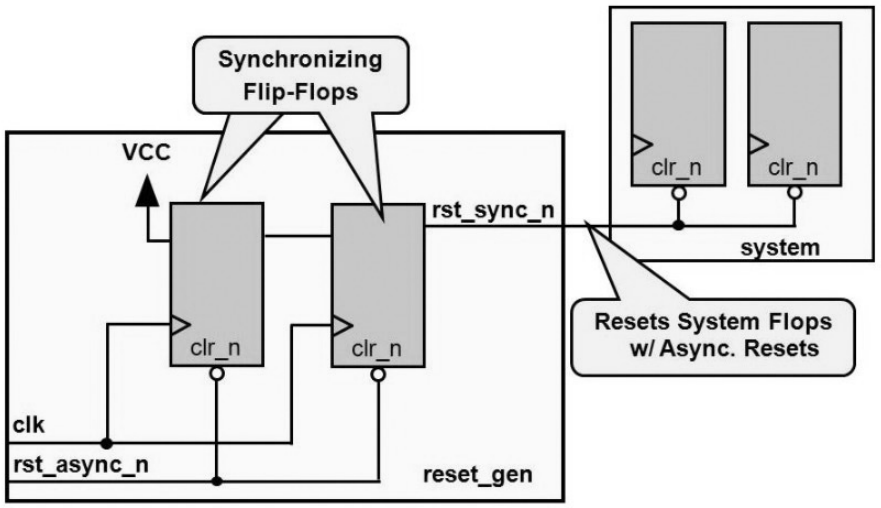
```
22     // 입력 클럭 생성 (100MHz, 주기 10ns)
23     initial begin
24         clk_in = 0;
25         forever #5 clk_in = ~clk_in; // 5ns마다 반전 (100MHz)
26     end
27
28     // 초기값 설정 및 테스트 시나리오
29     initial begin
30         // 초기값 설정
31         reset = 1;
32         prescaler = 4'd1; // prescaler = 1 (2^(1+1) = 4로 분주)
33
34         #20;              // 리셋 유지 (20ns)
35         reset = 0;        // 리셋 해제
36         #200;             // 200ns 동안 동작 관찰 (prescaler=1)
37         prescaler = 4'd2; // prescaler 변경: 2^(2+1)=8로 분주
38         #200;             // 추가로 200ns 동안 동작 관찰
39         prescaler = 4'd3; // prescaler 변경: 2^(3+1)=16로 분주
40         #400;             // 추가로 400ns 동안 동작 관찰
41         prescaler = 4'd0; // prescaler 변경: 분주 없음 (2^(0+1)=2로 분주)
42         #100;             // 추가로 100ns 동안 동작 관찰
43         $finish;          // 시뮬레이션 종료
44     end
45     // 파형 덤프 설정 (GTKWave 등에서 파형 확인용)
46     initial begin
47         $dumpfile("ClockControlUnit_tb.vcd");
48         $dumpvars(0, tb_ClockControlUnit);
49     end
50 endmodule
```

SoC 기본 System Clock 실습1 – 시뮬레이션 결과



SoC 기본 Reset 이해하기

항목	설명
리셋의 정의	디지털 회로를 초기 상태로 강제 설정하여 정상적인 동작을 보장하는 신호. 시스템을 "알려진 상태"로 만듦
리셋의 필요성	- 칩 초기화 시 정상 동작 보장 - 비정상 상태(예: 클럭 손실, 통신 오류)에서 복구
리셋의 유형	- 동기식 리셋(Synchronous Reset): 클럭 신호에 동기화되어 동작 - 비동기식 리셋(Asynchronous Reset): 클럭 신호와 무관하게 즉시 동작
동기식 리셋 특징	- 클럭 엣지에서만 리셋 신호 샘플링. - 메타스테이블 문제 없음. - 클럭 신호가 필요
비동기식 리셋 특징	- 클럭 없이 즉시 리셋 가능. - 빠른 초기화. - 메타스테이블 및 글리치 발생 가능성 있음
리셋 설계 시 고려사항	- 모든 플립플롭을 리셋할지 여부. - 리셋 트리 구조 및 타이밍 검증. - 멀티 클럭 도메인에서의 리셋 처리
리셋의 주요 사용 사례	- 전원 켜짐 초기화. - 통신 채널 재동기화. - 사용자 버튼 리셋. - 워치독 타이머 만료 후 시스템 복구
리셋 관련 문제점	- 비동기식 리셋 해제 시 메타스테이블 가능성. - 동기식 리셋은 클럭이 없으면 동작하지 않음



SoC 기본 Reset 이해하기

- 리셋(Reset)의 종류
    1. 비동기식 리셋(Asynchronous Reset): 클록 신호와 무관하게 즉시 작동하며, 빠른 리셋이 가능하지만 동기화 문제가 발생할 수 있습니다

```
// 비동기식 리셋 예제
always @(posedge clk or negedge creset_n)
begin
  if (!creset_n) begin
    // 리셋 로직
  end else begin
    // 정상 동작 로직
  end
end
```

SoC 기본 Reset 이해하기

- 동기식 리셋(Synchronous Reset): 클록 신호에 동기화되어 작동하며, 리셋 시 다음 클록 에지에서 시스템이 초기화됩니다. 이 방식은 클록 도메인 내에서 일관된 동작을 보장합니다

```
// 동기식 리셋 예제
always @(posedge clk)
begin
  if (!creset_n) begin
    // 리셋 로직
  end else begin
    // 정상 동작 로직
  end
end
```

SoC에서의 리셋 설계 고려사항

- ASIC 설계일반적 리셋의 활성화는 비동기적으로 하고 비활성화은 동기적으로 하는 것이 권장됩니다. 이는 메타안정성 문제를 방지하기 위함입니다.
- FPGA 설계에서는 활성화 비활성화 모두 동기적으로 처리하는 것이 일반적입니다
- 복잡한 시스템에서는 리셋 시퀀싱이 중요합니다.
  - 다양한 모듈, 프로세서, 인터커넥트/버스 구조, 주변장치등이 함께 리셋되어야 하지만 비활성화 시에는 적절한 순서로 진행되어야 할 수 있습니다. 일반적으로 시스템의 리셋은 PLL이 잠기고 클록이 안정될 때까지 유지되어야 합니다

Verilog 구현 모범 사례

- 클록과 리셋의 코딩 가이드라인
- Verilog에서 클록과 리셋을 구현할 때 다음과 같은 모범 사례를 따라야 합니다:
- always 블록에서의 이벤트 리스트: 순차 회로의 always 블록에서 이벤트 리스트(sensitivity list)에는 클록과 리셋만 포함되어야 합니다

```
// 올바른 방식
always @(posedge clock or negedge reset)
```

- 리셋 초기화: 모든 플립플롭은 반드시 리셋되어야 합니다. 리셋하지 않으면 초기 전원 인가 시 RS 플립플롭의 구조에 의해 초기값이 예측할 수 없게 됩니다.

Verilog Reset구현 모범 사례

- 리셋 에지 타입: 리셋은 일반적으로 negative edge(negedge)를 사용합니다. 이는 셀 설계자가 일반적으로 negative edge로 리셋을 설계하기 때문입니다
- 클록 에지 타입: 클록은 positive edge(posedge)를 사용합니다. 칩 전체에서 일관되게 이 규칙을 준수해야 합니다
- 논블로킹 할당 사용: 순차 회로에서는 논블로킹 할당(non-blocking assignment, <=)을 사용합니다. 이는 begin-end 블록 내의 여러 문장이 동시에 수행되도록 하기 위함입니다

```
// 올바른 방식
if (!reset)
  data_q <= 8'd0;
else
  data_q <= data_q + 1;
```

데이터 경로와 제어 신호 리셋

- SoC 설계에서 또 다른 중요한 고려사항은 데이터 경로(datapath)와 제어 신호에 대한 리셋 전략입니다. 일부 전문가들은 데이터 경로 레지스터를 리셋 불가능하게 만들면 리셋 팬아웃(fan-out)을 줄이고 설계의 논리 레벨을 줄일 수 있다고 주장합니다. 이는 혼잡하거나 큰 설계에서 도움이 될 수 있지만, 일반적으로 QoR(Quality of Results)에 미치는 영향은 적습니다.

- 데이터 신호보다는 제어 신호만 리셋하는 것이 더 나은 방법일 수 있습니다. 많은 제어 신호(예: valid 신호)는 명시적으로 필요할 때만 high 상태가 됩니다. 따라서 이러한 신호의 기본값을 비활성화 상태로 설정하고, 필요할 때만 덮어쓰는 방식이 효율적입니다

일반적인 문제와 해결 방법

- 타이밍 문제
  - 리셋과 관련된 타이밍 문제는 SoC 설계에서 흔히 발생합니다. 특히 비동기 리셋을 사용할 때 Recovery Time과 Removal Time이라는 중요한 타이밍 매개변수가 있습니다
  - Recovery Time: 리셋이 해제된 후 다음 활성 클록 에지까지 필요한 최소 시간
  - Removal Time: 클록 에지 이후에 리셋이 해제될 수 있는 최소 요구 시간
  - 이러한 타이밍 요구사항을 충족하지 못하면 metastability 문제가 발생할 수 있습니다.
- 도구 해석 문제
  - FPGA 설계 도구는 신호 이름에 따라 리셋의 극성을 해석할 수 있습니다.
  - 예를 들어, '_n' 접미사가 있는 이름은 active-low 리셋을 나타냅니다. 이로 인해 도구가 associate clock check를 켜거나 끄는 등의 동작이 달라질 수 있습니다

System Clock and Reset - 결론

- Verilog를 사용한 SoC 설계에서 클록과 리셋은 시스템의 안정성과 성능에 직접적인 영향을 미치는 핵심 요소입니다. 적절한 클록 생성 및 분배, 그리고 신중한 리셋 전략은 안정적인 SoC 구현의 기반이 됩니다.

- 클록은 일반적으로 positive edge를 사용하고, 리셋은 negative edge를 사용하는 일관된 방식을 전체 설계에서 유지해야 합니다. 또한 always 블록의 이벤트 리스트에는 클록과 리셋만 포함되어야 하며, 순차 회로에서는 논블로킹 할당을 사용해야 합니다.

- ASIC과 FPGA 설계에서는 리셋 전략이 다를 수 있으며, 복잡한 시스템에서는 리셋 시퀀싱이 중요합니다. 또한 클록 도메인 크로싱에서는 리셋이 각 도메인에 맞게 적절히 동기화되어야 합니다.

- 이러한 모범 사례를 따르면 안정적이고 예측 가능한 동작을 하는 SoC를 설계할 수 있으며, 시뮬레이션부터 실제 하드웨어 작동까지 문제를 줄일 수 있습니다

SoC 기본 Reset - 실습2

- SoC에서의 리셋
- 다음을 설계해 보시오

```
`timescale 1ns/1ns
module sync_async_reset (
    input clock,        // 클럭 입력
    input reset_n,      // 비동기 리셋 (Active Low)
    input data_a,       // 입력 데이터 A
    input data_b,       // 입력 데이터 B
    output reg out_a,   // 출력 데이터 A
    output reg out_b    // 출력 데이터 B
);

reg sync_reg1, sync_reg2; // 리셋 동기화용 레지스터

// 리셋 동기화 블록: 비동기적으로 활성화되고 동기적으로 해제됨
always @(posedge clock or negedge reset_n) begin
    if (!reset_n) begin
        sync_reg1 <= 1'b0; // 비동기적으로 리셋 활성화
        sync_reg2 <= 1'b0;
    end else begin
        sync_reg1 <= 1'b1; // 동기적으로 리셋 해제
        sync_reg2 <= sync_reg1;
    end
end

wire synchronized_reset = sync_reg2; // 동기화된 리셋 신호

// 데이터 처리 블록: 동기화된 리셋 신호를 사용하여 데이터 초기화 및 처리
always @(posedge clock or negedge synchronized_reset) begin
    if (!synchronized_reset) begin
        out_a <= 1'b0; // 리셋 시 초기화
        out_b <= 1'b0;
    end else begin
        out_a <= data_a; // 정상 작동 시 데이터 처리
        out_b <= data_b;
    end
end

endmodule
```

1

2

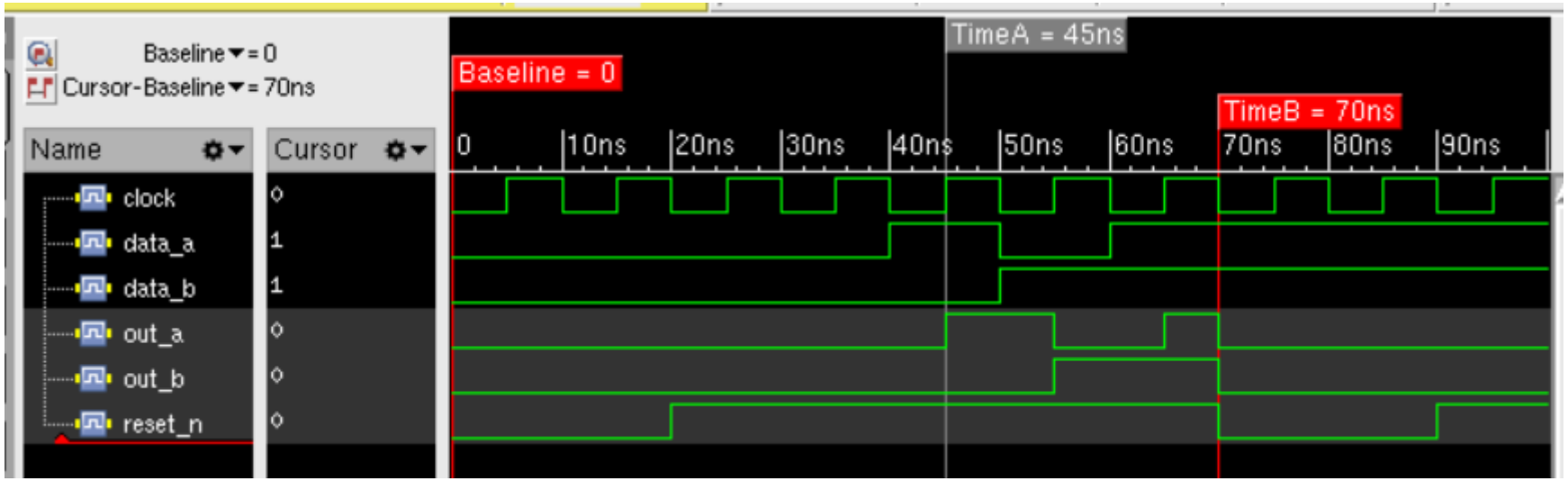
SoC 기본 Reset 실습2 시뮬레이션

- 시뮬레이션 하십시오

```
`timescale 1ns/1ns
module tb_sync_async_reset;

    // 테스트벤치용 신호 선언
    reg clock;          // 클럭 신호
    reg reset_n;        // 비동기 리셋 신호 (Active Low)
    reg data_a;         // 입력 데이터 A
    reg data_b;         // 입력 데이터 B
    wire out_a;         // 출력 데이터 A
    wire out_b;         // 출력 데이터 B

    // DUT(Design Under Test) 인스턴스화
    sync_async_reset uut (
        .clock(clock),
        .reset_n(reset_n),
        .data_a(data_a),
        .data_b(data_b),
        .out_a(out_a),
        .out_b(out_b)
    );

    // 클럭 생성: 10ns 주기 (100MHz)
    initial begin
        clock = 0;
        forever #5 clock = ~clock; // 5ns마다 클럭 토글
    end
    // 테스트 시나리오
    initial begin
        // 초기화
        reset_n = 1'b0;   // 리셋 활성화 (Active Low)
        data_a = 1'b0;
        data_b = 1'b0;
        #20;              // 20ns 동안 리셋 유지
        reset_n = 1'b1;   // 리셋 해제
        #20;
        // 정상 동작 테스트: 입력 데이터를 변경하며 출력 확인
        data_a = 1'b1;
        data_b = 1'b0;
        #10;
        data_a = 1'b0;
        data_b = 1'b1;
        #10;
        data_a = 1'b1;
        data_b = 1'b1;
        #10;
        // 리셋 재활성화 테스트
        reset_n = 1'b0;   // 리셋 활성화
        #20;
        reset_n = 1'b1;   // 리셋 해제
        #10;
        $stop;            // 시뮬레이션 종료
    end

endmodule
```

SoC 기본 Reset 실습2 – 시뮬레이션 결과

- 시뮬레이션 결과를 확인 하십시오



SoC GPIO

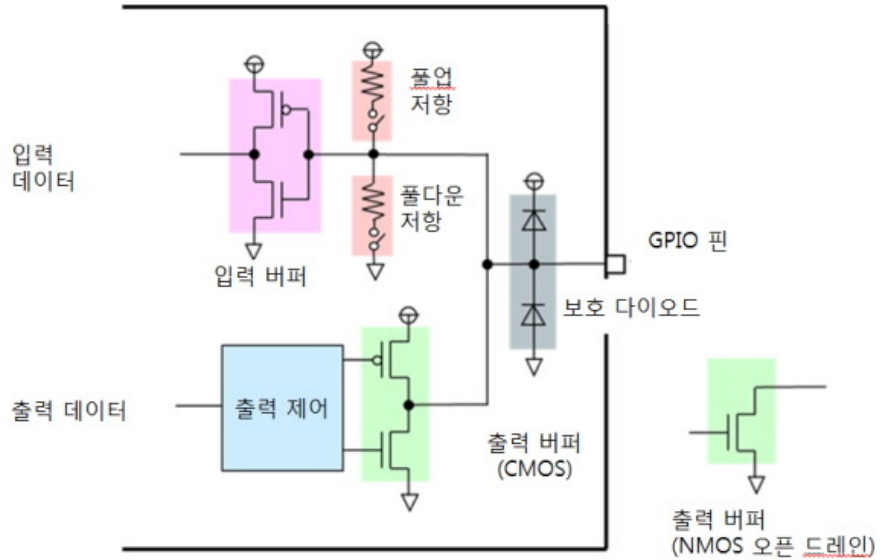
Agenda

1. SoC 설계에서 GPIO 입출력 회로 구조 이해 및 시뮬레이션
   - Logic Level
   - CMOS Transistors
   - Push–Pull Output (Digital)
   - Open-collector/Open-drain Output
   - Tri-state Output
2. GPIO에 대한 이해 및 시뮬레이션
   - SoC에서 사용되는 GPIO 구성요소
   - GPIO제어 구조
   - GPIO 레지스터 맵 구조

- Q&A

SoC에서 사용되는 GPIO 구성요소 이해하기

- Logic Level, CMOS Transistors,
- Push–Pull Output (Digital),
- Open-collector/Open-drain Output
- Tri-state Output,



출력 유형	주요 특징	용도
Push-Pull	강력한 소스/싱크 능력, 외부 저항 불필요	일반 디지털 신호, LED 드라이브
Open-Drain	플로팅 상태에서 풀업 저항 필요, 다중 장치 연결 가능	I2C, 인터럽트 라인
Tri-State	Hi-Z 상태 지원, 여러 장치가 동일 버스를 공유 가능	데이터 버스, 메모리 인터페이스
CMOS Transistors	낮은 전력 소비, 높은 입력 임피던스	모든 현대적 GPIO

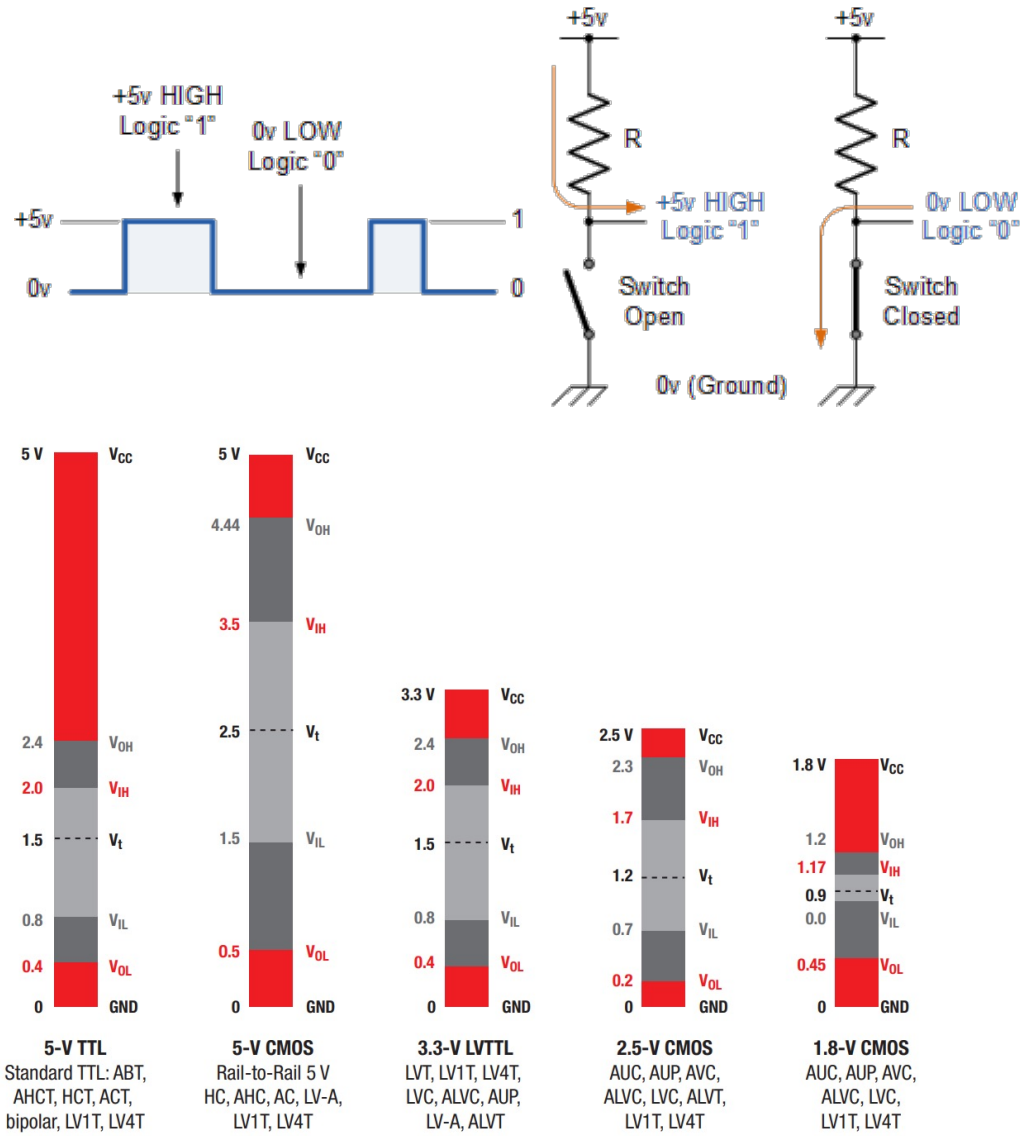
SoC에서 사용되는 디지털 Logic level 이해

Logic Level
Logic Level은 디지털 회로에서 신호의 전압 상태를 나타내며, 일반적으로 0 또는 1로 표현

Positive Logic: 낮은 전압(0V)은 논리 0, 높은 전압(예: 5V)은 논리 1을 나타냅니다.

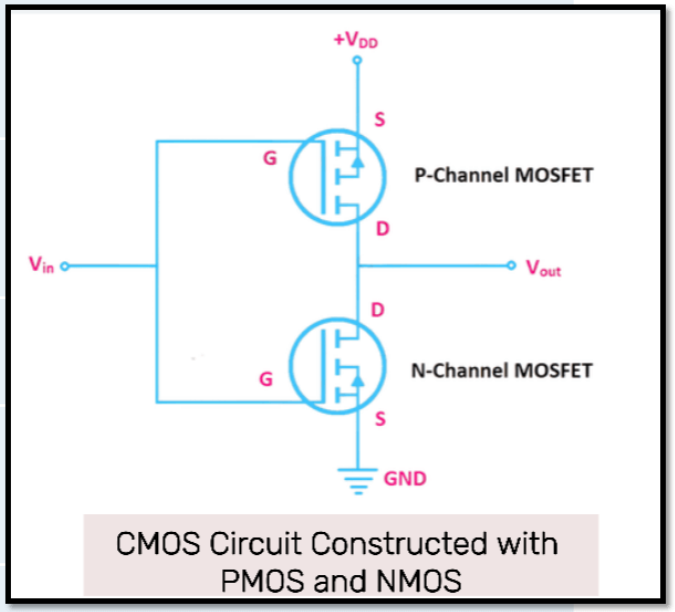
TTL (Transistor-Transistor Logic): 일반적으로 0~0.8V는 논리 0, 2~5V는 논리 1을 나타냅니다.

CMOS Logic: 공급 전압(VDD)의 비율로 논리 수준을 정의하며, 낮은 전압은 논리 0, 높은 전압은 논리 1을 나타냅니다



SoC에서 사용되는 디지털 CMOS Transistors 이해

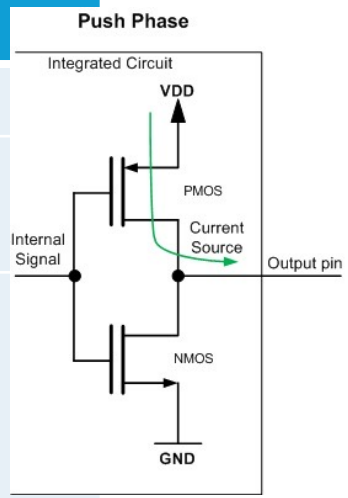
항목	설명
CMOS 기술 정의	CMOS는 NMOS와 PMOS 트랜지스터를 상보적으로 결합하여 전력 소비를 줄이고 높은 성능을 제공.
구조	- NMOS: Vdd에서 GND로 전류를 흐르게 함. - PMOS: GND에서 Vdd로 전류를 흐르게 함.
특징	- 낮은 정적 전력 소비. - 높은 노이즈 면역성. - 스위칭 시에만 전력 소비 발생.
장점	- 전력 효율성: 정적 상태에서 거의 전력이 소모되지 않음. - 열 발생 감소. - 고밀도 집적 가능.
단점	- 고주파에서 스위칭 시 순간적인 전력 소비 증가. - 누설 전류 문제(소형화에 따라 증가).
응용 분야	- 마이크로프로세서, 마이크로컨트롤러. - 메모리 칩, CMOS 이미지 센서. - RF 회로 및 데이터 컨버터.
스케일링 한계	- FinFET 및 나노시트 기술 도입으로 단위 면적당 트랜지스터 성능 향상. - 전력 공급 전압 감소 가능.
CMOS 2.0 혁신	- 3D 적층 기술을 활용한 기능 분리. - 고밀도 논리층과 저전압 트랜지스터를 조합하여 성능 최적화.
누설 전류 관리	- 고-κ 유전체 사용으로 게이트 누설 감소. - 다중 임계값 CMOS(MTCMOS)로 속도와 누설 전류 균형 유지.

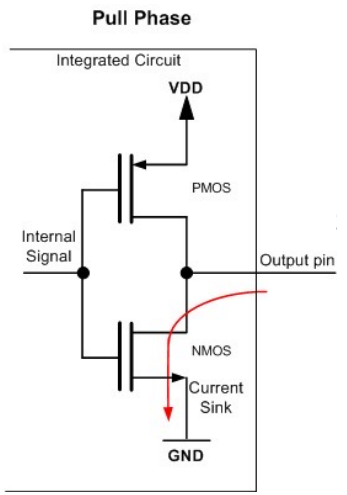


CMOS Circuit Constructed with PMOS and NMOS

SoC에서 사용되는 Push–Pull Output

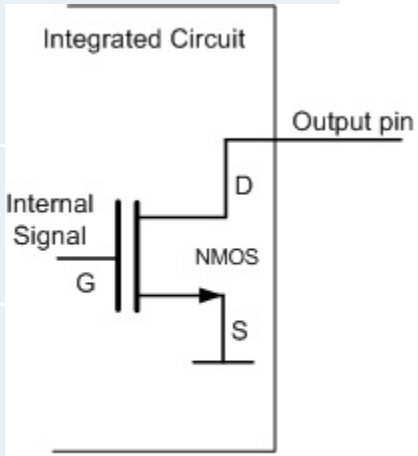
항목	설명
구성	PMOS와 NMOS 트랜지스터가 상보적으로 연결된 출력 구조
작동 방식	- PMOS: 출력이 HIGH일 때 활성화되어 VDD와 연결 - NMOS: 출력이 LOW일 때 활성화되어 GND와 연결
장점	- 외부 풀업/풀다운 저항 불필요 - 강력한 전류 소스 및 싱크 능력 제공 - 빠른 신호 전환 속도 - 낮은 출력 임피던스로 안정적인 신호 전달
단점	- 여러 출력 핀이 같은 라인을 공유할 경우 신호 충돌 발생 가능 - 단일 출력 핀에서만 사용 권장
적용 분야	- 마이크로컨트롤러 GPIO 핀 - LED 드라이버 - 디지털 신호 전송 등
특징 요약	능동적으로 HIGH 또는 LOW 상태를 출력하며, Hi-Z 상태는 지원하지 않음





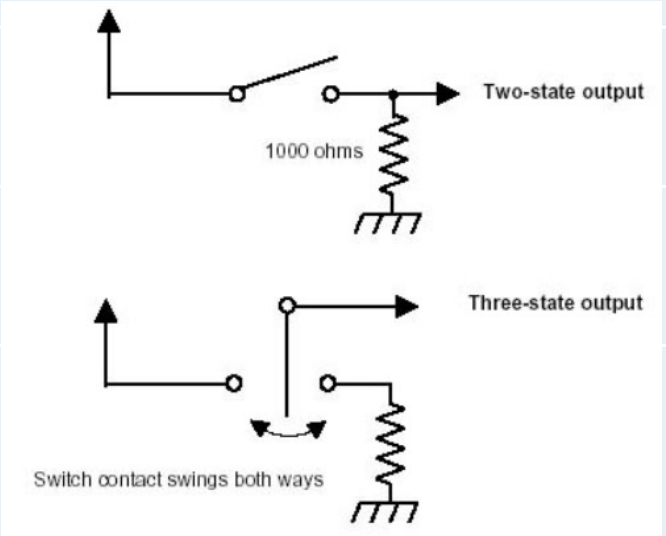
SoC에서 사용되는 Open Collector

항목	설명
정의	출력 핀이 단일 트랜지스터(BJT 또는 NMOS)에 의해 GND로 연결될 수 있는 출력 방식.
구조	트랜지스터의 컬렉터 또는 드레인이 출력 핀과 연결되며, 외부 풀업 저항을 통해 HIGH 상태를 유지.
작동 방식	- LOW 출력: 트랜지스터가 활성화되어 출력 핀이 GND에 연결. - HIGH 출력: 트랜지스터가 비활성화되어 외부 풀업 저항을 통해 Vcc로 유지.
특징	- 여러 장치가 동일한 라인을 공유 가능. - 와이어드 AND/OR 논리 구현 가능. - 전류 소싱 능력이 낮음.
장점	- 다중 장치 연결에 적합. - 간단한 회로 구조. - 외부 풀업 저항으로 전압 레벨 조정 가능.
단점	- 외부 풀업 저항 필요. - 신호 전환 속도가 상대적으로 느림. - 전류 소싱 능력이 부족함.
응용 분야	- I2C 통신. 인터럽트 라인. - 멀티 드롭 버스. - 와이어드 논리 구현.
와이어드 논리 구현	- 와이어드 AND: 여러 Open Collector 출력이 같은 라인에서 LOW일 때만 LOW 상태 유지. - 와이어드 OR: 여러 출력 중 하나라도 HIGH일 때 HIGH 상태 유지.
전압 레벨 조정	외부 풀업 저항을 통해 원하는 전압 레벨(Vcc)을 설정 가능.



Tri-state Output

항목	설명
정의	출력 핀이 HIGH, LOW, Hi-Z(High-Impedance) 상태 중 하나를 가질 수 있는 출력 방식.
구조	추가적인 Enable 신호를 통해 출력이 활성화(HIGH/LOW)되거나 비활성화(Hi-Z 상태)될 수 있음.
작동 방식	- Enable = 1: 출력이 HIGH 또는 LOW로 동작. - Enable = 0: 출력이 Hi-Z 상태로 플로팅.
특징	- Hi-Z 상태에서는 출력 핀이 회로에서 분리되어 전류가 흐르지 않음. - 여러 장치가 동일한 버스를 공유 가능.
장점	- 다중 장치 간 데이터 공유 가능. - Hi-Z 상태에서 회로 간섭 없음. - 외부 풀업 저항 불필요.
단점	- Hi-Z 상태에서 외부 노이즈에 취약. - 충돌 방지를 위해 정확한 제어 필요. - 복잡한 제어 로직 요구.
응용 분야	- 데이터 버스. - 메모리 인터페이스. - 다중 장치 간 데이터 공유. - FPGA 및 마이크로컨트롤러 설계.
Hi-Z 상태의 역할	- 출력 핀을 비활성화하여 다른 장치가 동일한 라인을 사용할 수 있도록 허용. - 전력 소모 감소.
전압 레벨 조정	Hi-Z 상태에서는 외부 회로(풀업/풀다운 저항 등)를 통해 출력 전압 레벨 유지 가능.



SoC 기본 - 3상태 게이트 버퍼 입출력 제어, 실습3

- SoC에서의 3상태
- 게이트 버퍼
- 다음을 설계해 보시오

```
1  `timescale 1ns / 1ps
2  module tri_io_example (
3      input             clk,       // 클럭 신호
4      input             reset,
5      input             oe,        // 출력 활성화 신호 (Output Enable)
6      input       [7:0] data_in,   // 출력 데이터
7      output reg  [7:0] data_out,  // 입력 데이터 저장
8      inout  wire [7:0] io_pin     // 양방향 핀
9  );
10
11     // 트라이스테이트 제어
12     assign io_pin = (oe) ? data_in : 8'bz; // 출력 모드일 때만 데이터 전송
13
14     // 입력 데이터를 읽어오는 로직
15     always @(posedge clk, posedge reset) begin
16         if (reset) begin
17             data_out <= 0;
18         end else begin
19             if (!oe) begin
20                 data_out <= io_pin;  // 입력 모드일 때만 데이터 읽기
21             end
22
23         end
24     end
25 endmodule
```

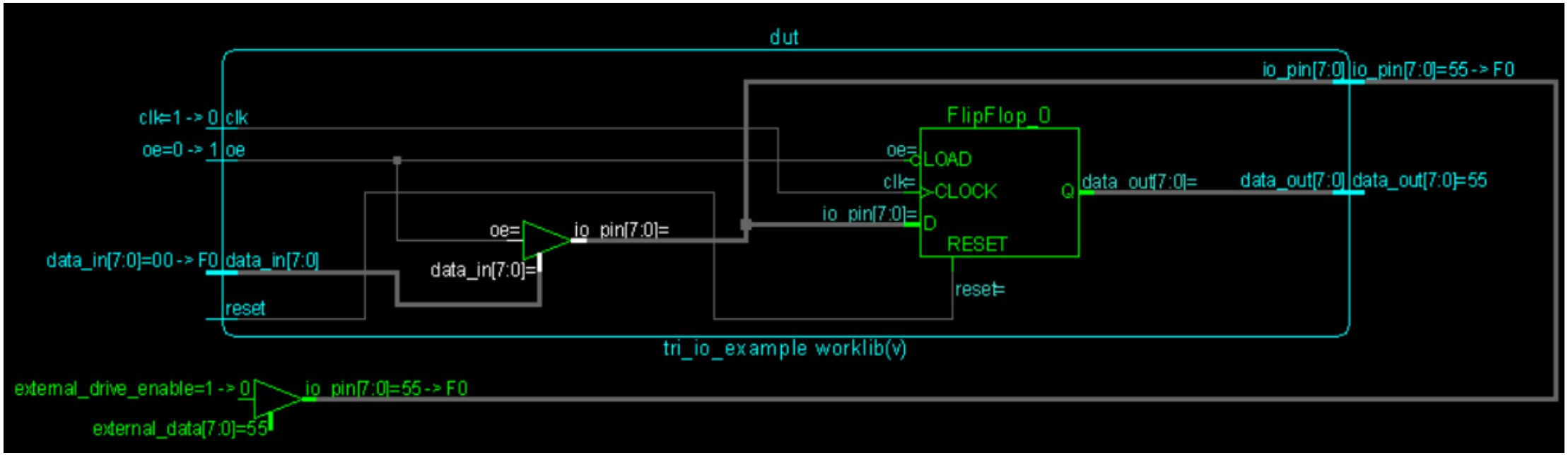
3상태 게이트 버퍼 입출력 제어, 실습3 시뮬레이션

- SoC에서의 3상태 게이트 버퍼, 다음을 시뮬레이션 해 보시오

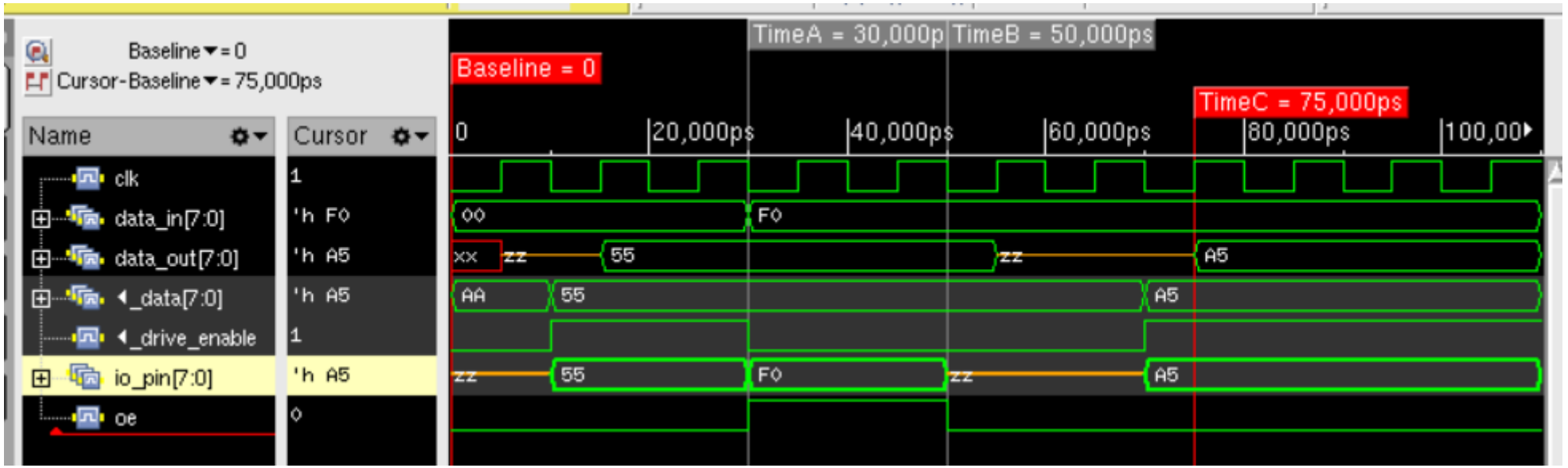
```
`timescale 1ns / 1ps

module tri_io_example_tb;

    reg clk;
    reg oe;
    reg [7:0] data_in;
    wire [7:0] io_pin;
    wire [7:0] data_out;

    // 외부 환경 모델링
    reg [7:0] external_data;
    reg external_drive_enable;

    // 양방향 핀 연결 (외부 환경 구동)
    assign io_pin = external_drive_enable ? external_data : 8'bz;

    // DUT 인스턴스화
    tri_io_example dut (
        .clk(clk),
        .oe(oe),
        .data_in(data_in),
        .data_out(data_out),
        .io_pin(io_pin)
    );

    // 클럭 생성 (10ns 주기)
    always #5 clk = ~clk;
```

```
    initial begin
        // 초기화
        clk = 0;
        oe = 0;
        data_in = 8'h00;
        external_data = 8'hAA;          // 외부에서 보낼 초기값
        external_drive_enable = 0;

        $display("Simulation Start");
        // Step 1: 외부 데이터를 DUT로 입력하는 테스트 (입력 모드)
        #10;
        external_drive_enable = 1;      // 외부가 io_pin을 구동 (입력 모드)
        external_data = 8'h55;          // 외부 데이터 설정
        #20;
        oe = 1;                         // DUT를 출력 모드로 전환
        external_drive_enable = 0;      // 외부 구동 중지
        data_in = 8'hF0;                // DUT에서 출력할 데이터 설정
        #20;
        oe = 0;                         // DUT를 입력 모드로 전환
        #20;
        external_drive_enable = 1;      // 다시 외부 데이터를 구동 (입력 모드)
        external_data = 8'hA5;          // 새로운 외부 데이터 설정
        #40;
        $display("Simulation End");
        $finish;
    end

    initial begin
        $dumpfile("tri_io_example_tb.vcd");
        $dumpvars(0, tri_io_example_tb);
    end
```

3상태 게이트 버퍼 입출력 제어, 실습3 로직 이해



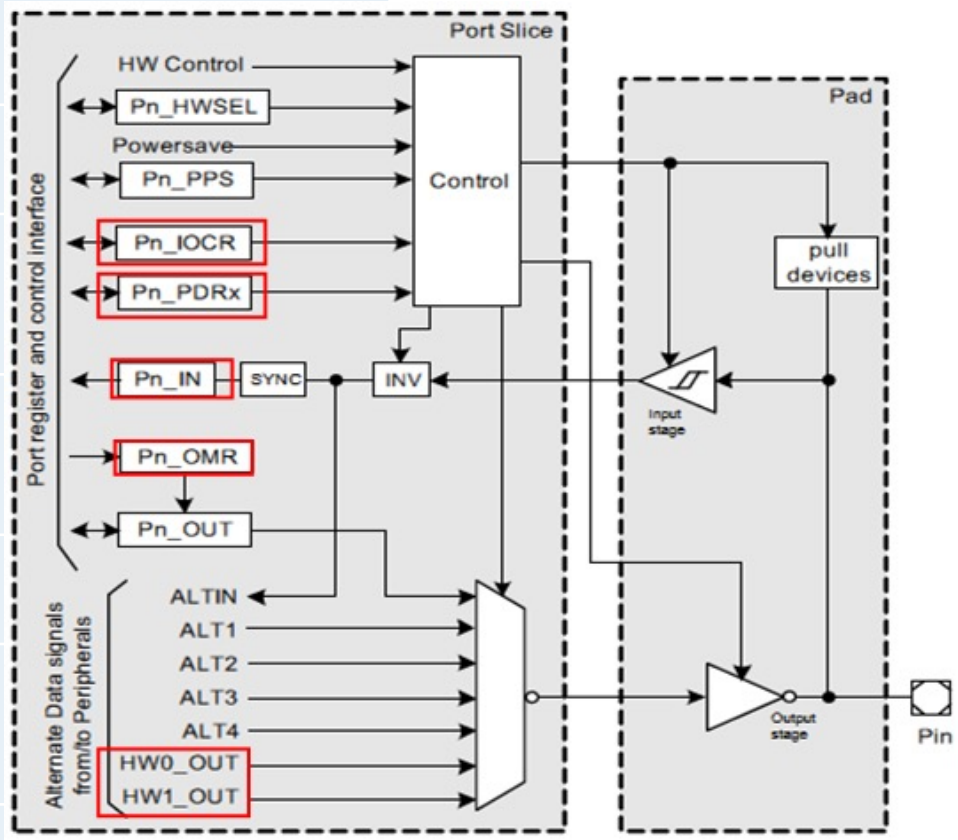
3상태 게이트 버퍼 입출력 제어, 실습3 시뮬레이션 결과



(ns)	clk	oe	data_in	data_out	external_ drive_enable	external_ data	io_pin
0	0	0	0	XX	1	AA	AA
20	1	0	0	55	1	55	55
30	0	1	F0	55	0	ZZ	F0
50	1	1	F0	55	0	ZZ	F0
75	0	0	F0	A5	1	A5	A5

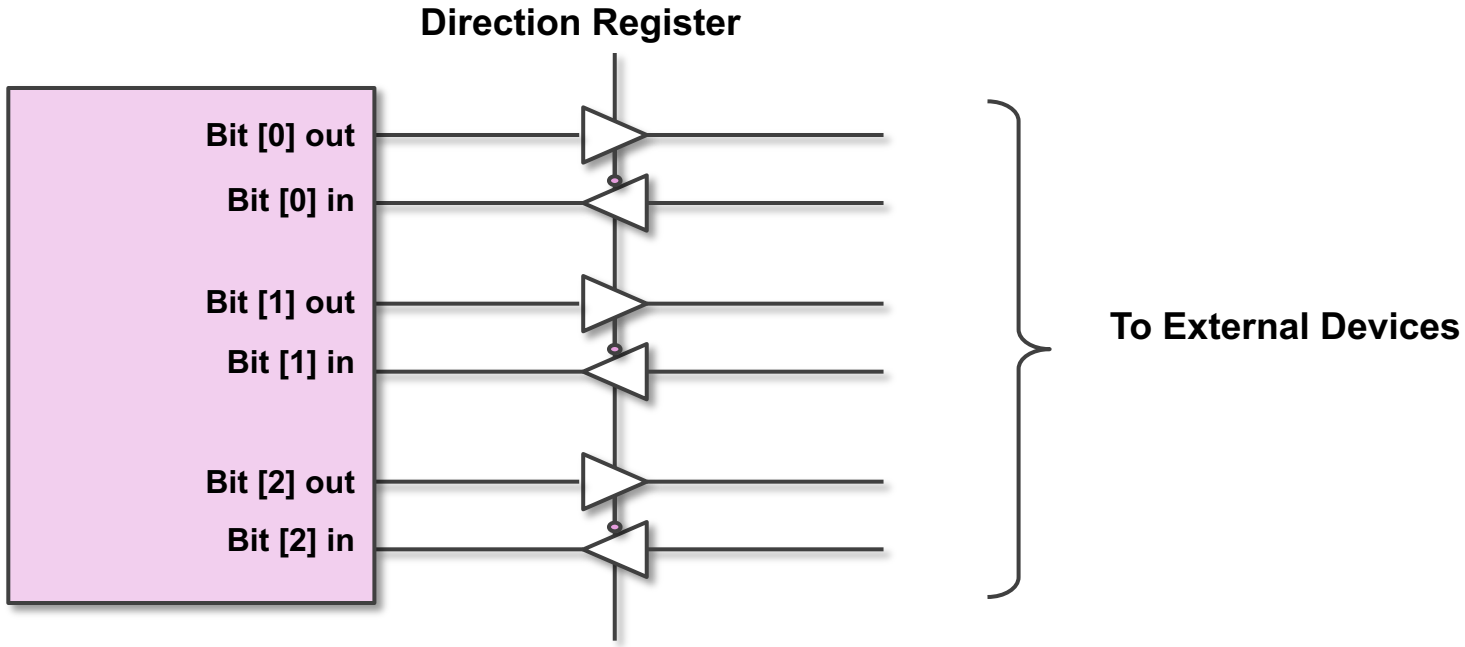
SoC에서 사용되는 GPIO (General-Purpose Input/Output)

항목	설명
정의	GPIO는 SoC에서 디지털 신호의 입출력을 처리하는 다목적 핀으로, 입력 또는 출력으로 구성 가능.
구조 및 동작	- 입력 모드: 외부 신호를 감지하고 CPU에 전달. - 출력 모드: CPU 명령에 따라 외부 장치를 제어.
특징	- 각 핀은 독립적으로 입력, 출력, 또는 양방향으로 설정 가능. - Active-High 또는 Active-Low로 동작.
구성 요소	- 내부 레지스터를 통해 핀 상태 제어. - 풀업/풀다운 저항 내장 가능. - 인터럽트 생성 기능 제공.
장점	- 유연한 핀 구성. - 다양한 전압 레벨 지원(예: 1.8V, 3.3V). - 저속 신호 처리에 적합.
단점	- 고속 신호 처리에는 부적합. - 외부 회로 설계 시 노이즈와 간섭 방지 필요.
응용 분야	- LED 제어, 센서 데이터 수집. - 버튼 입력 처리. - I2C, SPI 통신 인터페이스.
고급 기능	- 인터럽트 지원(엣지 또는 레벨 감지). - 하드웨어 디바운싱. - PWM 및 Blink 기능 지원.



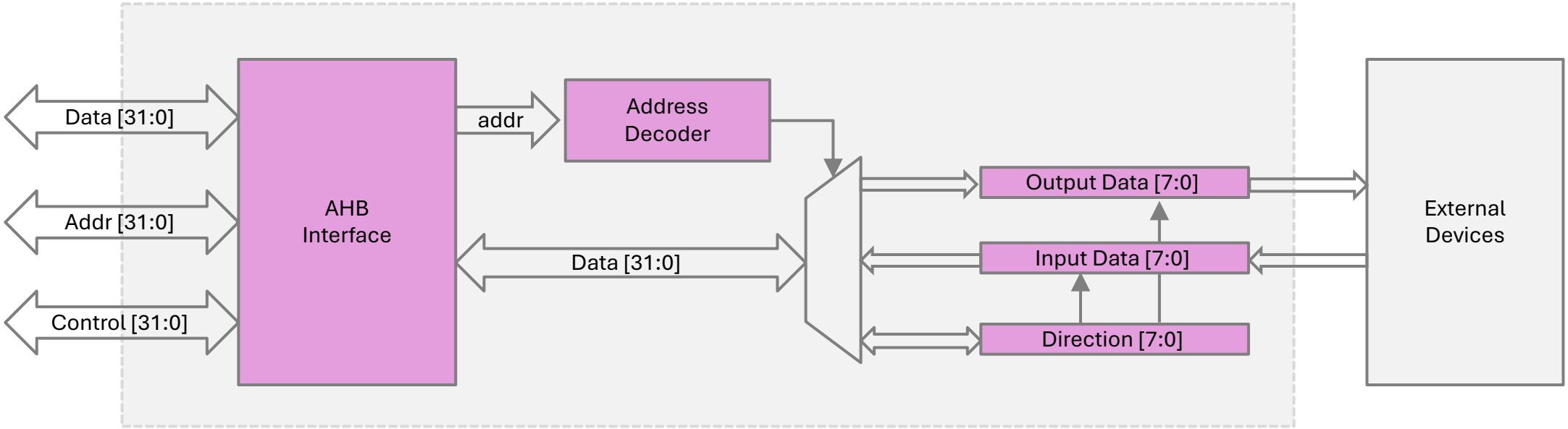
GPIO Overview

- General-purpose input/output (GPIO)
  - 일반적인 목적으로 사용되며, 특별한 용도가 정의되지 않음.
  - 대부분의 응용 프로그램에서 널리 사용됨.
  - 입출력 방향은 방향 레지스터(direction register)에 의해 제어됨.
  - 특정 비트를 마스킹하기 위해 마스크 레지스터(mask register)가 자주 사용됨.



SoC AHB GPIO

- Basic hardware architecture
  - 기본 레지스터인 데이터 입력(data in), 데이터 출력(data out), 방향 레지스터(direction register)만을 가지고 있음.



GPIO Registers

- The GPIO peripheral registers include
  - 데이터 레지스터
  - 입력 데이터(Input data): 외부 장치로부터 읽어온 데이터를 저장.
  - 출력 데이터(Output data): 외부 장치로 전송할 데이터를 저장.
  - 방향 레지스터(Direction register): 읽기 또는 쓰기 동작을 제어.

Register	Address	Size
GPIO base address	0x5300_0000	
Input Data	0x5300_0000	4 Byte
Output Data	0x5300_0004	4 Byte
Direction	0x5300_0008	4 Byte

SoC 기본 GPIO 이해하기 – AHB GPIO 실습 4

- SoC에서의 GPIO, 다음을 설계해 보시오

```
1   `timescale 1ns/1ns
2   module ahb_gpio (
3       input wire clk,                 // 시스템 클럭
4       input wire reset_n,             // 비동기 리셋 (Active Low)
5       input wire [31:0] addr,         // AHB 주소
6       input wire [31:0] wdata,        // AHB 쓰기 데이터
7       input wire write_enable,        // AHB 쓰기 활성화 신호
8       output reg [31:0] rdata,        // AHB 읽기 데이터
9       input wire [7:0] gpio_in,       // 외부 입력 데이터
10      output reg [7:0] gpio_out,      // 외부 출력 데이터
11      output reg [7:0] gpio_dir       // GPIO 방향 제어 (1: 출력, 0: 입력)
12  );
13
14      // 내부 레지스터 정의
15      reg [7:0] input_data;           // 입력 데이터 레지스터
16      reg [7:0] output_data;          // 출력 데이터 레지스터
17      reg [7:0] direction;            // 방향 레지스터
18
19      // 주소 매핑 상수 정의
20      localparam ADDR_INPUT  = 32'h5300_0000;  // 입력 레지스터 주소
21      localparam ADDR_OUTPUT = 32'h5300_0004;  // 출력 레지스터 주소
22      localparam ADDR_DIR    = 32'h5300_0008;  // 방향 레지스터 주소
23
24      // 읽기 동작 처리
25      always @(*) begin
26          case (addr)
27              ADDR_INPUT:  rdata = {24'b0, input_data};   // 입력 데이터 읽기
28              ADDR_OUTPUT: rdata = {24'b0, output_data};  // 출력 데이터 읽기
29              ADDR_DIR:    rdata = {24'b0, direction};    // 방향 레지스터 읽기
30              default:     rdata = 32'b0;                 // 기본값
31          endcase
32      end
34      // 쓰기 동작 처리
35      always @(posedge clk or negedge reset_n) begin
36          if (!reset_n) begin
37              input_data <= 8'b0;
38              output_data <= 8'b0;
39              direction <= 8'b0;
40          end else if (write_enable) begin
41              case (addr)
42                  ADDR_OUTPUT: output_data <= wdata[7:0];   // 출력 데이터 쓰기
43                  ADDR_DIR:    direction <= wdata[7:0];     // 방향 레지스터 쓰기
44                  default: ;                                // 기본값 유지
45              endcase
46          end
47      end
48
49      // GPIO 동작 처리 (입출력 제어)
50      always @(posedge clk or negedge reset_n) begin
51          if (!reset_n) begin
52              gpio_out <= 8'b0;
53              gpio_dir <= 8'b0;
54          end else begin
55              gpio_out <= output_data & direction;          // 출력 모드일 때만 값 설정
56              gpio_dir <= direction;                       // 방향 설정 유지
57          end
58
59          input_data <= gpio_in & ~direction;              // 입력 모드일 때만 값 읽기
60      end
61  endmodule
```

1
2
3
4
5

SoC 기본 IO 이해하기 - 3상태 게이트 버퍼 입출력 제어, 실습 시뮬레이션

- SoC에서의 GPIO, 다음을 시뮬레이션 해 보시오

```
1   `timescale 1ns/1ns
2   module tb_ahb_gpio;
3
4       reg clk;
5       reg reset_n;
6       reg [31:0] addr;
7       reg [31:0] wdata;
8       reg write_enable;
9       wire [31:0] rdata;
10      reg [7:0] gpio_in;
11      wire [7:0] gpio_out;
12
13      ahb_gpio uut (
14          .clk(clk),
15          .reset_n(reset_n),
16          .addr(addr),
17          .wdata(wdata),
18          .write_enable(write_enable),
19          .rdata(rdata),
20          .gpio_in(gpio_in),
21          .gpio_out(gpio_out)
22      );
23
24      initial begin
25          clk = 1'b0;
26          forever #5 clk = ~clk;  // 클럭 생성 (10ns 주기)
27      end
```

```
29      initial begin
30          reset_n = 1'b0;
31          addr = 32'b0;
32          wdata = 32'b0;
33          write_enable = 1'b0;
34          gpio_in = 8'b10101010;
35
36          #10 reset_n = 1'b1;        // 리셋 해제
37
38          #10 addr = 32'h53000008; wdata = 8'b11110000;
39          write_enable = 1'b1; #10 write_enable = 1'b0;
40              // 방향 설정(출력)
41
42          #10 addr = 32'h53000004; wdata = 8'b11001100;
43          write_enable = 1'b1; #10 write_enable = 1'b0;
44              // 출력 데이터 설정
45
46          #10 addr = 32'h53000000; write_enable = 1'b0;
47              $display("Input Data Read: %b", rdata[7:0]);
48              // 입력 데이터 확인
49
50          #50 $stop;
51      end
52
53  endmodule
```

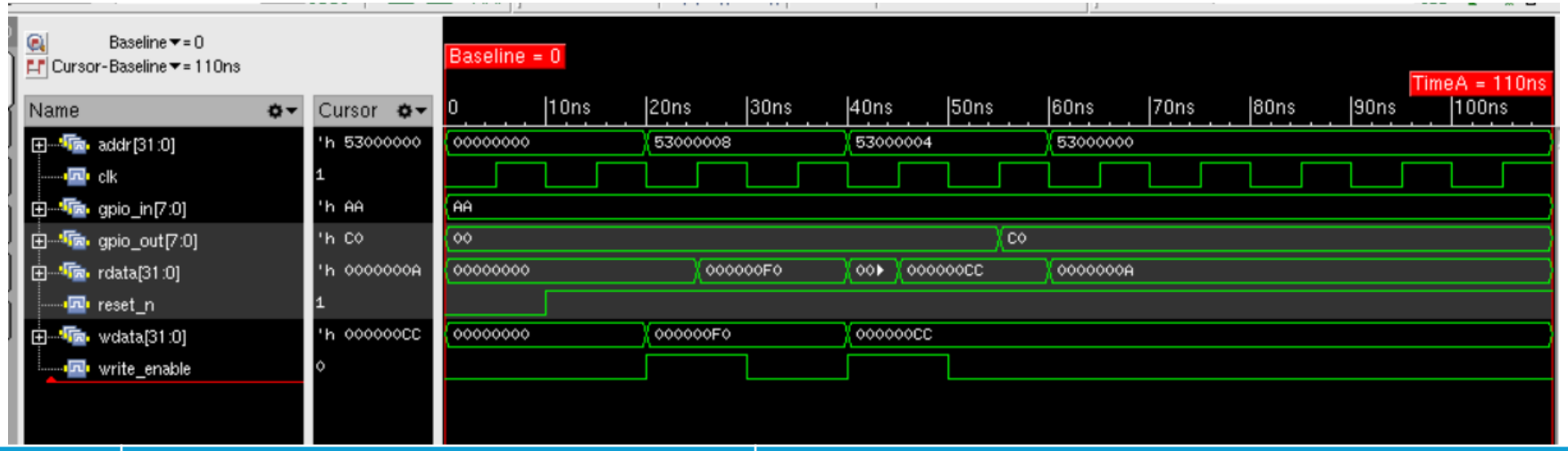
1

2

3

4

3상태 게이트 버퍼 입출력 제어, 실습 시뮬레이션 결과



테스트 단계	입력 조건	예상 결과	결과 상태
초기화	- reset_n = 0 - 모든 레지스터 초기화	- gpio_out = 0 - gpio_dir = 0 // 내부 레지스터 초기화	성공
방향 레지스터 설정	- 주소: 0x5300_0008 - 데이터: 0b11110000	- 상위 4비트는 출력, 하위 4비트는 입력으로 설정	성공
출력 데이터 설정	- 주소: 0x5300_0004 - 데이터: 0b11001100	- 출력 핀(gpio_out)에 1100이 설정되고, 입력 핀은 변경되지 않음	성공
입력 데이터 읽기	- 외부 입력 데이터: gpio_in = 0b10101010	- 입력 핀(gpio_in)에서 읽힌 값이 출력(rdata)으로 반환	성공
출력 확인 (출력 모드)	- 방향 레지스터(gpio_dir)에 따라 상위 4비트만 출력 모드로 설정	- 출력 핀(gpio_out)의 상위 4비트는 설정된 값 유지 (1100), 하위 4비트는 입력 모드로 유지	성공
입력 확인 (입력 모드)	- 방향 레지스터(gpio_dir)에 따라 하위 4비트만 입력 모드로 설정	- 입력 핀(gpio_in)의 하위 4비트가 읽히고, 출력 핀의 상위 4비트는 변경되지 않음	성공

Thanks