보안 회로 설계

실습 Introduction & Verilog 기초문법

Dong Kyue Kim
Hanyang University
dqkim@hanyang.ac.kr

Verilog 기초문법

Contents

- Verilog HDL
- Verilog type
- Verilog 연산자
- Verilog 순차 처리문

Verilog HDL(1/2)

- Verilog HDL 특성
  - 국제 표준(공정, 설계에 제한x)
  - C 프로그램 언어와 유사
  - 대소문자 구분 : 예약어는 반드시 소문자
- 예약어(키워드)
  - 사전에 예약된 식별자(always, and, assign, wire, reg,…)
  - 변수(variable), 식별자(identifier)로 사용x

Verilog HDL(2/2)

- 어휘 규칙
  - 여백 : 빈칸, 탭, 줄 바꿈 등을 포함
    문자열 에서만 의미를 가짐
  - 주석 : // 단일 라인 주석,
    /* */ 블록 주석
  - 숫자 표현
    - [비트수]'〈Base_format〉〈number_value〉
    - integer : simple decimal, 32bit
    - ex)
      - 2017 : signed number, 32bits
      - –2 : signed number, 32 bits in 2's complement
      - 2'b10 : size of 2bits
      - 8'h9a = 8'b10011010

Design Entities – Verilog

- Module
  - Verilog 기본 구성단위
  - “module” ~ “endmodule”로 종료
  - 몸체부 : 회로의 기능, 동작, 구조 표현

```
1  module half_adder(a,b,sum,cout);
2      input a, b;
3      output sum, cout;
4      wire cout_bar;
5
6      xor(sum,a,b);
7      nand(cout_bar, a, b);
8      not(cout, cout_bar);
9
10 endmodule
~
~
```

module 이름은 대소문자 구분, _사용 가능

와이어 선언;
파라미터 선언;

회로 기능 표현

endmodule로 종료

Design Entities – Verilog

- Gate primitive를 이용한 모델링

```
 1 module half_adder(a,b,sum,cout);
 2     input a, b;
 3     output sum, cout;
 4     wire cout_bar;
 5
 6     xor(sum,a,b);
 7     nand(cout_bar, a, b);
 8     not(cout, cout_bar);
 9
10 endmodule
```

예약어(primitive)를 사용한 module 구현

  - 병행문이므로 순서에 무관, output 동일
  - 코딩시 주로 사용되는 구조x

Design Entities – Verilog

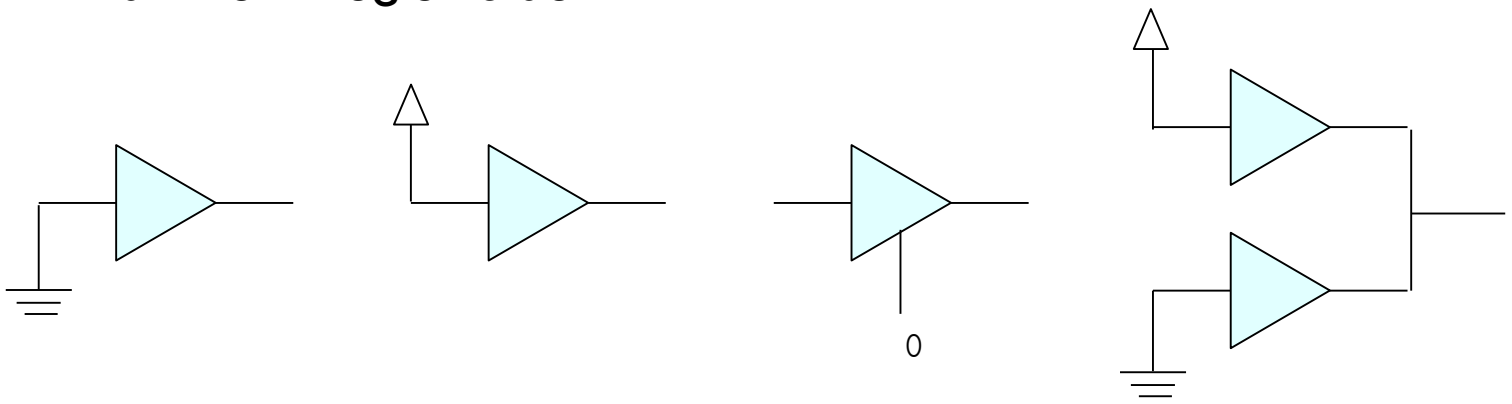
- 연속 할당문을 이용한 모델링
  - Assign문으로 표현

```
1 module half_adder2(a,b,sum,cout)
2     input a, b;
3     output sum, cout;
4
5     assign cout = a&b;
6     assign sum = a^b;
7
8 endmodule
```

비트 연산자를 이용한 module구현

Verilog type

- Verilog 논리값 : 0 1 X Z
  - 0 : logic zero, false condition
  - 1 : logic one, true condition
  - Z : high-impedance state
  - X : unknown logic value



논리값 0　　논리값 1　　논리값 z　　논리값 x

- **설계에 있어서 x, z는 존재하면 안됨**

Verilog type

- Verilog의 net type
  - 하드웨어 요소들 사이의 연결
  - wire : 단순한 열결
  - wor : 다중 구동자, or gate로 연결
  - wand : 다중 구동자, and gate로 연결

  - supply0 : Ground net
  - supply1 : Vdd net

Verilog type

- Verilog의 reg type
  - always, initial 구문 에서 사용
  - reg type은 할당 사이의 값을 유지

- Verilog의 벡터
  - 다중비트의 net, reg type 선언 시 사용
  - ex) wire [7 : 0] ess; = 8비트 벡터 ess

- Verilog의 배열
  - 자료형의 요소를 다차원으로 묶기 위해 사용
  - ex) reg lab[0 : 255] = 1비트 lab 256개

Verilog 연산자

- 산술 연산자

	operator	example
Multiplication	*	a = b * c
Division	/	a = b / c
Addition	+	a = b + c
Subtraction	–	a = b – c
Modulus	%	a = b % c

- Modulus연산 결과

수식	result
10%3	1
–10%3	–1
11%–3	2

- 관계 연산자

	operator	example
less than	〈	a = b 〈 c
less than or equal to	〈=	a = b〈=c
greater than	〉	a = b 〉 c
greater than or equal to	〉=	if(a〉=b)
equality	==	if(b==c)
inequality	!=	if(b!=c)

– 회로의 복잡도를 높여 사용 지양
*, /, %

Verilog 연산자

- 비트 연산자

	operator	example
unary negation Not	~	a = ~b
binary AND	&	a = b & c
binary OR	|	a = b | c
binary XOR	^	a = b ^ c
binary XNOR	^~, ~^	a = ~^ c

- 축약 연산자

	operator	example
AND	&	a = &b
OR	|	a = |b
NAND	~&	a =~&b
NOR	~|	a = ~|b
XOR	^	a = ^b

```
1 module nand4_op2(a, y);
2     input [3:0] a;
3     output y;
4
5     assign y = ~(a[0] & a[1] & a[2] & a[3]);
6 endmodule
7
```

```
1 module nand4_op2(a, y);
2     input [3:0] a;
3     output y;
4
5     assign y = ~&a;
6 endmodule
7
```

Verilog 연산자

- 논리 연산자

	operator	Example
Not	!	if(!a)
AND	&&	if((a 〉 b) && (a 〉 c))
OR	||	if((a 〉 b) || (a 〉 c))

- 조건 연산자
  - 기호 ?, : 사용
  - 3항 연산자
  - ex) condition = a?b:c

- 결합 및 반복 연산자

	operator	Example
concatenation	{}	a = {b, c, c}
replication	{{}}	a = {b, {2{c}}

Verilog 연산자

- 결합 및 반복 연산자 : { }
  - { }에 묶인 피연산자를 순서대로 결합
  - ex) 8bit + 8bit = 16bit

```
1 wire [15:0] addr_bus;
2 wire [7:0] addr_h, addr_l;
3
4 assign addr_bus = {addr_h, addr_l};
```

  - {a{b}} a횟수 만큼 반복
  - ex) {4{w}}={w, w, w, w}

Verilog 연산자

- Shift 연산자

	operator	Example
Right shift	〉〉	01011_1001 = 1011_0011 〉〉 1
Left shift	〈〈	0110_0110 = 1011_0011 〈〈 1

  - 직관성이 떨어지므로 반복연산자로 대체
  - ex) left shift : a = {a[7:0], a[8]};

순차 처리문(1/2)

- Initial 문
  - 블록 부분 순차적 실행
  - 시뮬레이션 때만 사용

```
Initial
begin
    “assign” 문
    “if, case” 문 등
    .
    .
end
```

순차적으로 1회 진행

순차 처리문(2/2)

- always 문
  - @(event)신호로 수행여부 결정

```
always @(event or … or event)
begin
    “if, case” 문 등
    .
    .
end
```

순차적으로 1회 진행

  - always 문 내의 신호는 reg혹은 integer로 선언
  - @(*) : always 문 내의 모든 변수를 포함

조건문(1/2)

- if 문
  - (조건문)신호로 수행여부 결정

```
if(조건문)
begin
  “할당문들” ;
end
else if(조건문)
begin
  “할당문들” ;
end
else if(조건문)
begin
  “할당문들” ;
end
```

```
ex)
always @(a or b or sel)
  if(sel==1'b0)
        out = a;
  else
        out = b;
```

If 문을 활용한 mux2to1 예시

  - 할당문이 1줄 –〉 begin…end 생략 가능

조건문(2/2)

- case 문
  - case문을 신호로 수행여부 결정

```
case(조건식)
    case_item1 : 수행문
    case_item2 : 수행문
    case_item3 : 수행문
    (default)
endcase
```

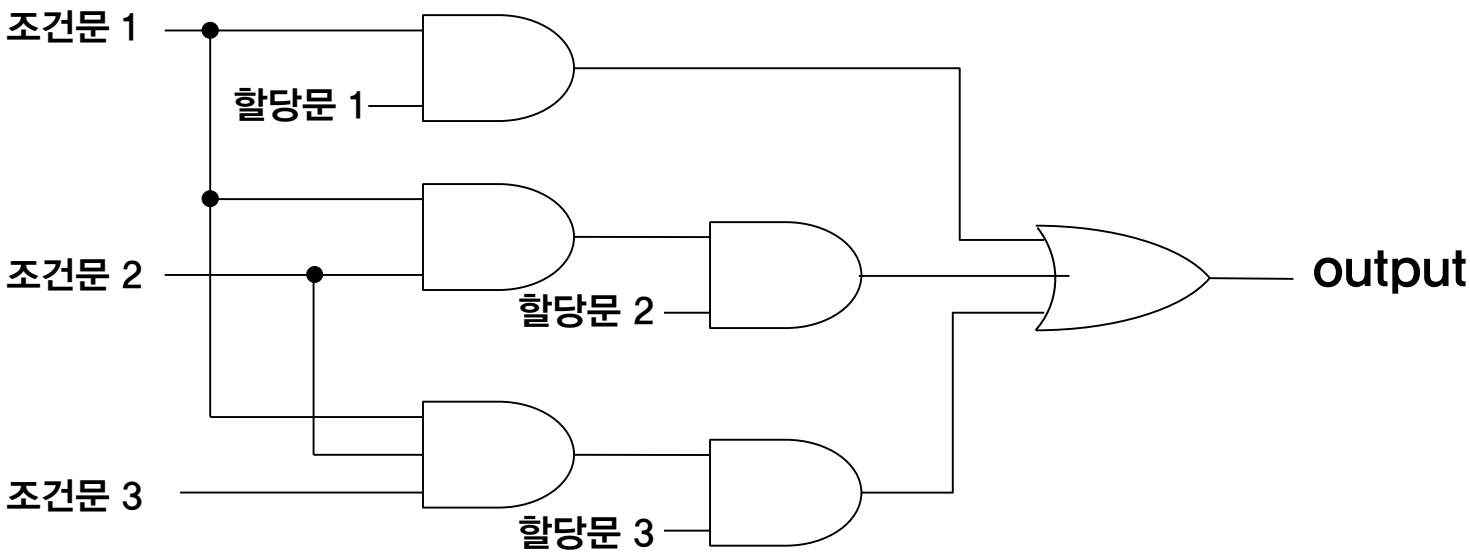
```
ex)
always @(a or b or sel)
begin
    case(sel)
        0 : out = a;
        1 : out = b;
    endcase
end
```

case 문을 활용한 mux2to1 예시

  - case_item 나열된 순서로 비교
  - 만족하는 case_item x –〉 default
  - default 없을시 기존 값 유지

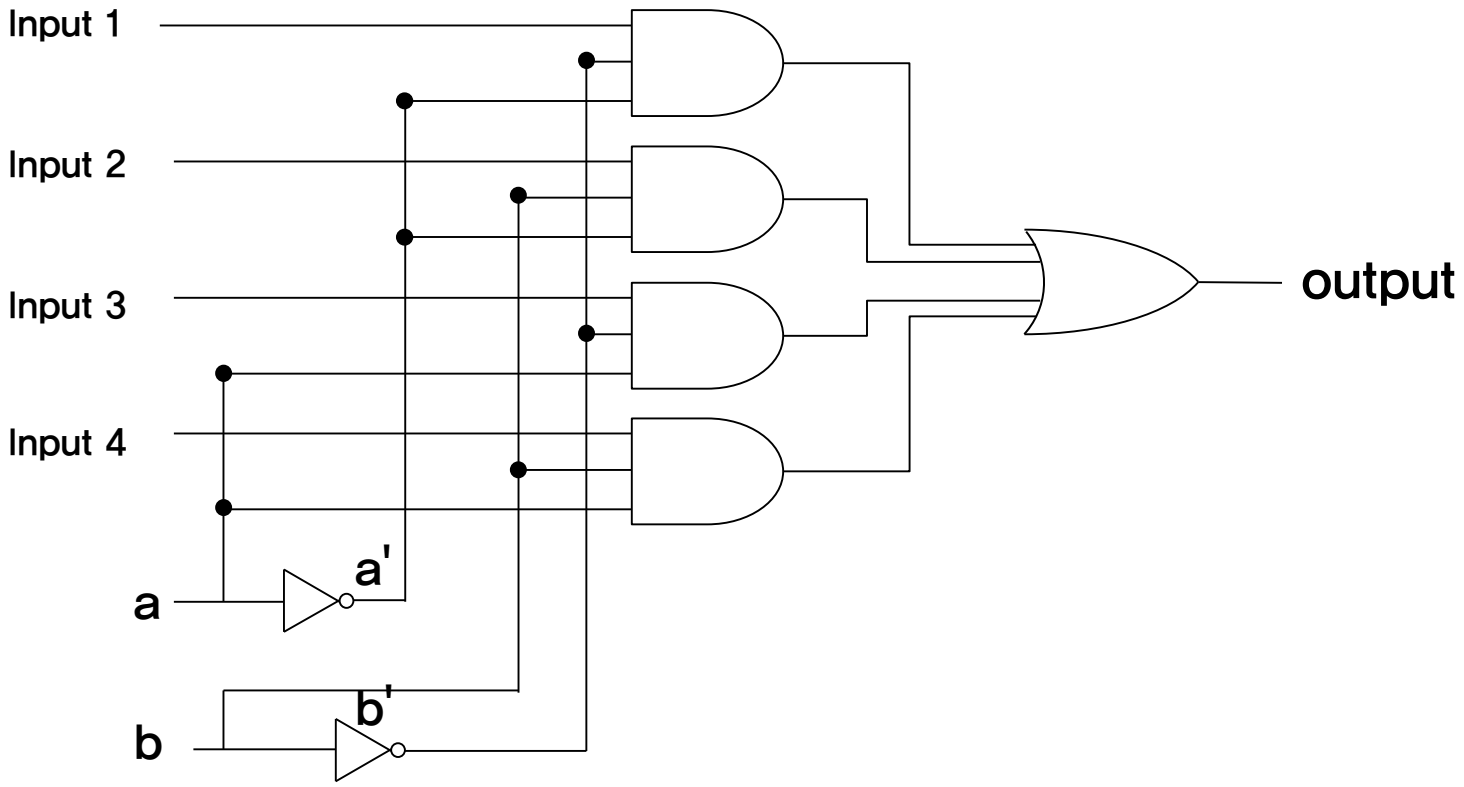
조건문 회로도(1/2)

- 회로표현
  - MUX를 활용한 if문 회로



조건문 회로도(2/2)

- 회로표현
  - MUX를 활용한 case문 회로



반복문

- for 문
  - case문을 신호로 수행여부 결정

```
for(초기값 ; 조건문 ; 증감)
    begin
        “순차구문”
    end
end
```

```
ex)
always @(Address)
    for(J=3 ; J>=0 ; J=J-1)
        if(Address == J)
                Line[J]=1;
         else
                Line[J] =0;
end
```

for 문을 활용한 예시

  - 코드의 양을 줄이기 위해 사용

Contents

- Logic Circuit
  - Combinational Circuit & Sequential Circuit
- Latch
  - Inferred Latch
- Flip-Flop
  - Synchronous vs Asynchronous Reset
  - Enable vs Reset
  - Blocking/Non-blocking Assignment
- The Finite State Machine

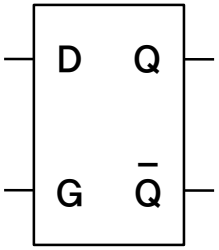
Logic Circuit

- Combinational Logic Circuit (조합회로)
  - 입력 → 출력이 결정 됨
  - 기억소자 포함 X
- Sequential Logic Circuit (순차회로)
  - 입력 & 회로에 기억된 상태 값 → 출력이 결정 됨
  - 기억소자 포함 O
  - 동작 방식
    - Synchronous : 하나의 공통 clock신호에 의해 동작
    - Asynchronous : 서로 다른 clock신호에 의해 동작

Latch vs Flip-Flop

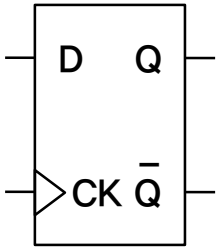
- comparison

Latch		Flip-Flop	
동작 O	동작 X	동작 O	동작 X
Enable 신호의 정해진 level (1 or 0)	반대 lever (0 or 1)	정해준 clock edge (positive or negative)	정해진 clock edge 제외
➔ Level sensitive		➔ Edge sensitive	



D	G	Q_+	$\bar{Q}_+$
X	0	$Q\_$	$\bar{Q}\_$
0	1	0	1
1	1	1	0

D Latch

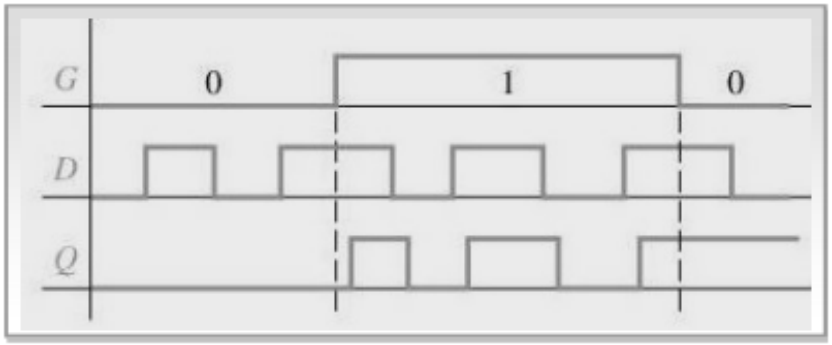


D	CK	Q_+	$\bar{Q}_+$
0	↑	0	1
1	↑	1	0
X	0	$Q\_$	$\bar{Q}\_$
X	1	$Q\_$	$\bar{Q}\_$

D Flip-flip

Latch

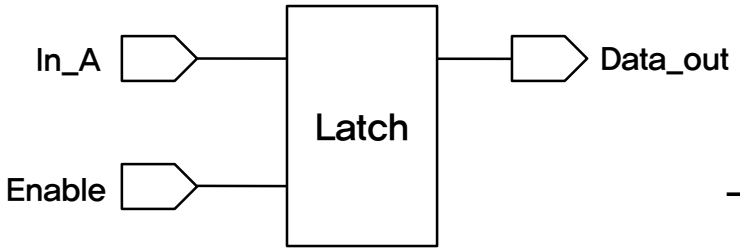
- Example



  - Clock 1 : 입력 D가 출력 Q로 전달
  - Clock 0 : 입력이 출력에 영향 X, 출력 유지

Inferred Latch

- 정의
  - Combinational always문은 모든 신호에 대해 구현되어야 함. 그렇지 않았을 때 발생하는 latch. Ex) If & else if, case
- Asynchronous latch



→ clock이 없음

- 막아야 하는 이유
  - 모든 flip-flop이 같은 clock에서 컨트롤 되는 synchronous 선호
    - Difficulty with accurate timing analysis
    - Unpredictable behavior

Solution of Inferred Latch : if

- If 문
  - 모든 branch에 대해 정의
  - 초기값 설정

```
always @(Enable or In_A)
begin
   if(Enable)
      begin
         Data_Out = In_A;
      end
end
```

해결방법

```
if(Enable)
   begin
      Data_Out = In_A;
   end
else
   begin
      Data_Out = In_B;
   end
end
```

```
Data_Out = In_B;
if(Enable)
   begin
      Data_Out = In_A;
   end
end
```

Solution of Inferred Latch : Case

- Case 문
  - default 사용
  - 초기값 설정

```
always @(Data_In)
begin    case(Data_In)
       0 : Data_Out = In_A;
       1 :  Data_Out = In_B;
   endcase
end
```

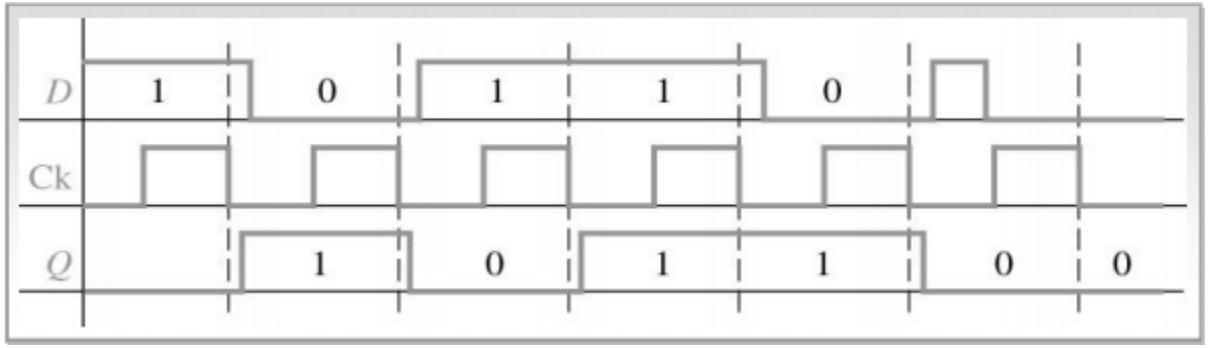
해결방법

```
always @(Data_In)
begin
   case(Data_In)
      0 : Data_Out = In_A;
      1 :  Data_Out = In_B;
      default : Data_Out = In_A;
   endcase
end
```

```
always @(Data_In)
begin
Data_out = In_A;
   case(Data_In)
      0 : Data_Out = In_A;
      1 :  Data_Out = In_B;
   endcase
end
```

Flip-flop

- Example

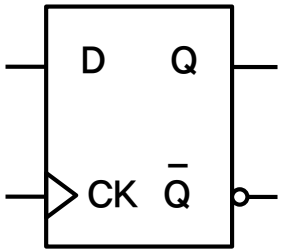


  - Clock신호 하강 시 : 입력D가 출력Q로 전달
  - 그 이외 기간 : 입력이 출력에 영향 X, 출력 유지

Flip-flop 사용법

- Rising edge

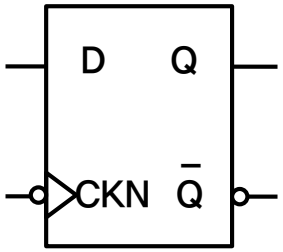
```
always@(posedge Clock)
begin
   ...
end
```



Positive edge
D Flip-Flop

- Falling edge

```
always@(negedge Clock)
begin
   ...
end
```



Negative edge
D Flip-Flop

Synchronous vs Asynchronous Reset (Code)

- 정의

```
always @(posedge Clock)
   if(!SynReset)
      Data_Out <= 0;
   else
      Data_Out <= Data_In;
end
```

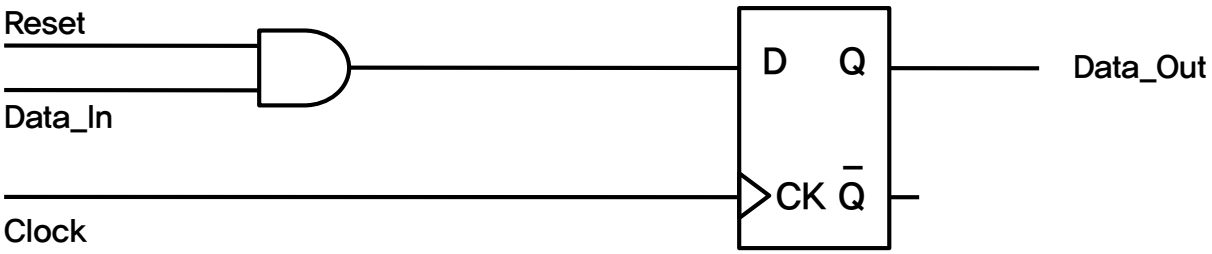
```
always @(posedge Clock or negedge AsynReset)
   if(!AsynReset)
      Data_Out <= 0;
   else
      Data_Out <= Data_In;
end
```

- Synchronous Reset : clock의 edge 에서 reset의 상태에 따라 reset발생

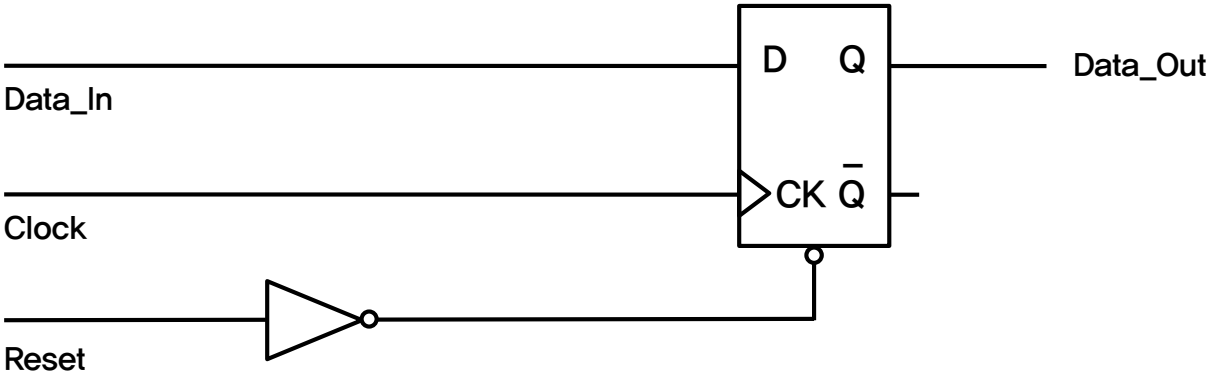
- Asynchronous Reset : reset의 edge 에서 reset 발생
  - edge condition과 if 조건문의 극성이 일치해야 함
    - always@(posedge clk or negedge AsynReset) → if(!AsynReset)
    - always@(posedge clk or posedge AsynReset) → if(AsynReset)

Synchronous vs Asynchronous Reset (Circuit)

- Synchronous Reset



- Asynchronous Reset



Enable & Reset (Code)

- 정의

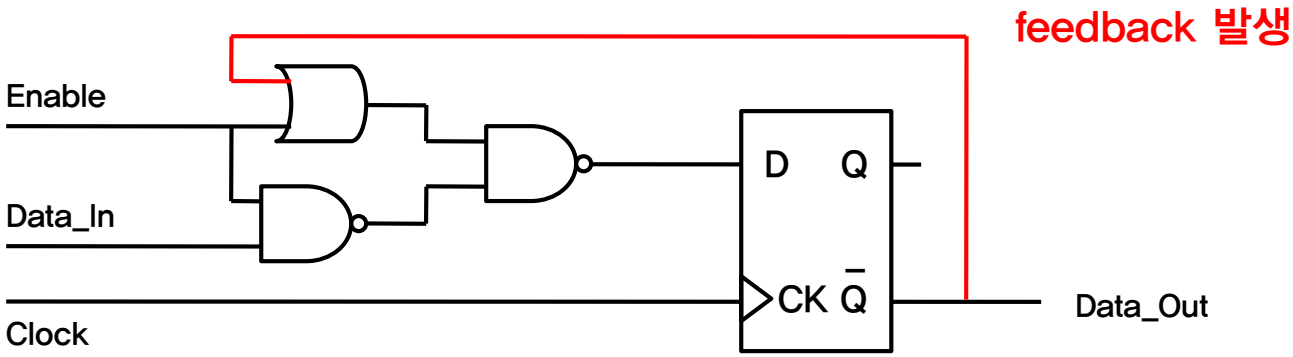
```
always @(posedge Clock)
   if(Enable)
      Data_Out <= Data_In;
end
```

```
always @(posedge Clock)
   if(!SynReset)
      Data_Out <= 0;
   else
      Data_Out <= Data_In;
end
```

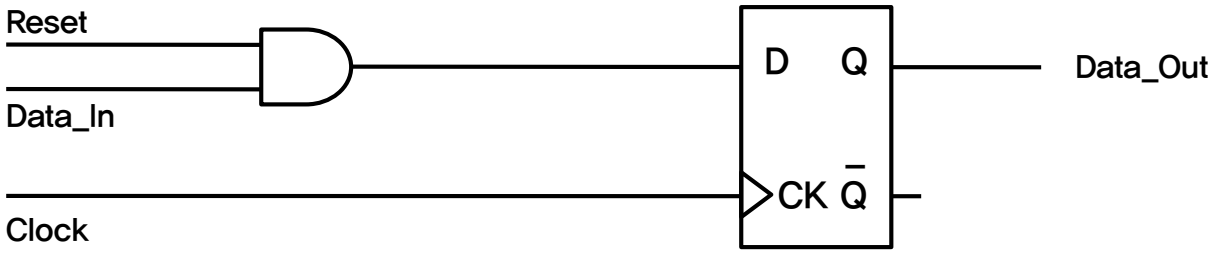
- Enable : Clock의 edge에서
  Enable = 1 : Data ← Data_in
  Enable = 0 : Data 유지 (출력 → 입력 피드백 발생)
- Synchronous Reset : clock의 edge 에서
  reset의 상태에 따라 reset발생

Enable vs Reset (Circuit)

- Enable



- Reset



Blocking/Non-Blocking Assignment

- Blocking Assignment : '=' 사용
  - 하나의 대입이 끝난 후 다음 대입 (기술순서에 영향 O)
  - assignment 바로 발생

- Non-blocking Assignment : '<=' 사용
  - 대입식의 오른쪽 모두 처리 후 왼쪽에 대입 (기술순서에 영향 X)
  - clock cycle의 끝 단에서 assignment

Blocking/Non-Blocking Assignment (Code)

- 실제 사용시 차이점
  - combination circuit에서는 차이점이 없음
  - sequential circuit에서 두 개 이상의 assignment가 일어날 때 차이점 발생

```
always @(posedge Clock)
begin
   Intermediate_Variable = In_A & In_B;
   Data_Out <= Intermediate_Variable;
end
```

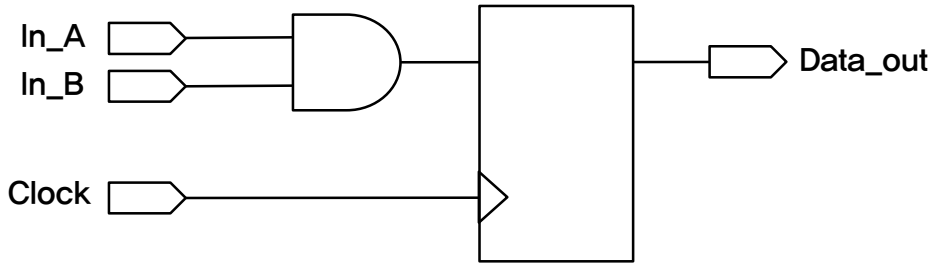
blocking assignment
→ Intermediate_Variable 바로 갱신
→ 갱신된 값이 Data_Out에 저장

```
always @(posedge Clock)
begin
   Intermediate_Variable <= In_A & In_B;
   Data_Out <= Intermediate_Variable;
end
```

non-blocking assignment
→ cycle의 끝에서 할당
→ Data_Out에 갱신 전
Intermediate_Variable 값이 저장

Blocking/Non-Blocking Assignment (Circuit)

- 실제 사용시 차이점
  - blocking assignment



  - non-blocking assignment