# Lecture 07

-Half Adder : S(sum), C(carry)

Sum=x XOR y

Carry=xy

-Full Adder : 올림수를 처리할 수 있도록 한 회로. (올림수도 더한다.)

Sum=x XOR y XOR c 

Carry=xy + xc+ yc

-carry ripple adder : output은 ADD 2개의 4-bit 숫자의 sum, carry를 포함한 5bit.

앞의 carry값이 영향을 끼치므로 시간이 필요하다. ( 4FA delays)

Multiply by 3 → L+L+L / 2L+L (shift left, 마지막자리수에 0추가)

-subtraction using addition

: using 2's complement → Only 1 adder required to perform addition and subtraction
Addition은 그냥 X+Y.
Subtraction은 X+Y'+1.

➔ Control bit를 하나 놓아서 addition이면 0, subtraction이면 1을 넣어준다.(carry로)

-Arithmetic overflow
: -2^(n-1) 부터 2^(n-1)-1까지 나타낼 수 있음. 이 범위를 벗어난다면 overflow.

-Determining Arithmetic Overflow
: c3는 계산한 비트 중 가장 상위 비트.
C4는 sign-bit (올림수)
C3와 c4값이 다르다면 overflow. → xor해서 1이라면 overflow.

-carry lookahead adder → full adder의 지연시간을 예방하기 위해 나옴.
carry값을 미리 예견함.

C1=b0c0+a0c0+a0b0  c2=b1c1+a1c1+a1b1

# Lecture 08

**-combinational logic vs sequential logic**

: **combinational**은 **memory**를 갖고 있지 않다.

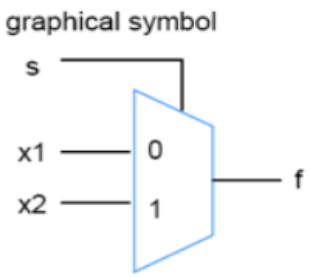
**Sequential**은 이전의 작동을 알아야 한다. **(**현재의 **input**값과 현재의 상태**(**이전 값**))**

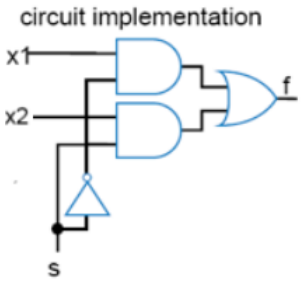
**-combinational logic**

**MUX : Multiplexer → 2^n**개의 입력 데이터로부터 **N**개의 선택 입력에**(N**비트짜리**)** 의해 선택된 정보다 단일 출력선을 통해 신호를 전송 **( if 8 input mux, 3 select inputs )**

logic expression (optimized)

f = s'x1 + sx2

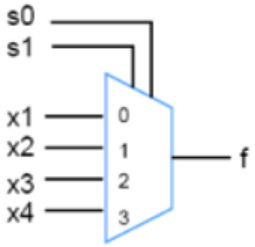



**- 4-1 MUX**

**f=s1's0'x1+s1's0x2+s1s0'x3+s1s0x4**

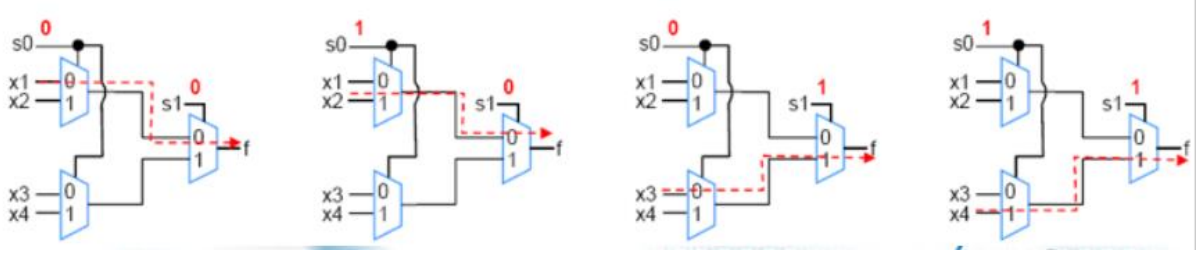
- Truth table to circuit
  - If S1 = 0 and S0 = 0, output x1
  - If S1 = 0 and S0 = 1, output x2
  - If S1 = 1 and S0 = 0, output x3
  - If S1 = 1 and S0 = 1, output x4


4x1 mux

| s1 | s0 | f |
|---|---|---|
| 0 | 0 | x1 |
| 0 | 1 | x2 |
| 1 | 0 | x3 |
| 1 | 1 | x4 |

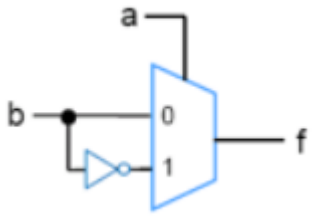
**-4-to-1 MUX using smaller MUXes**



**-XOR Gate implementation Using MUX**

| a | b | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| a | f |
|---|---|
| 0 | b |
| 1 | b' |




***XOR** → 홀수의 **1**의 개수를 가지고 있으면 **1**, 짝수의 **1**의 개수를 가지고 있으면 **0**

**-Shannon Expansion Theorem**

$$f(X_1, X_2, \ldots, X_n) = X_1 \cdot f(1, X_2, \ldots, X_n) + X_1' \cdot f(0, X_2, \ldots, X_n)$$

**F(a, b, c) = a'bc + ab'c + abc' + abc**

$F = a'F_{a'} + aF_a$
$= a'(bc) + a(c+b)$

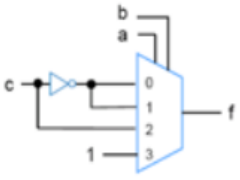
➔ **Expand for variable a → a가 select signal이 됨.**

**-expand the function with respect to multiple variables.**

**:** 4-to-1 MUX에서는 2개의 signal이 있기 때문에 2개를 바꿀 수 있겠거니 생각할 수 있음.

**Expand F(a, b, c) = a'c' + ab + ac with respect to a and b**

$F = a'b'F_{a'b'} + a'bF_{a'b} + ab'F_{ab'} + abF_{ab}$
$= a'b'(c') + a'b(c') + ab'(c) + ab(1)$

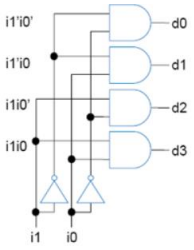



**Decoder = One-hot encoded**

: 해독기, 입력 단자의 어느 조합에 신호가 가해졌을 때 그 조합에 대응하는 하나의 출력.

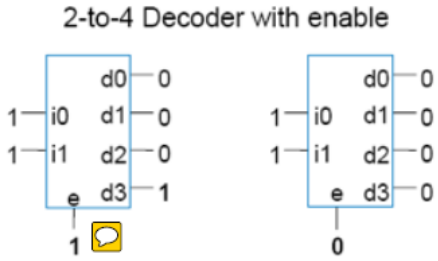
**N-input, 2^N output**

➔ **Input의 combination으로 output이 정해진다.**



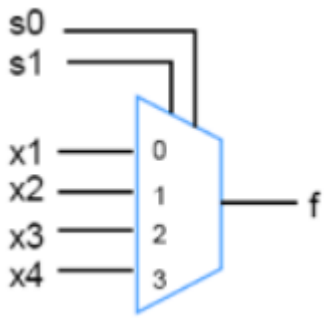

**: AND gate for each output to detect input combination.**

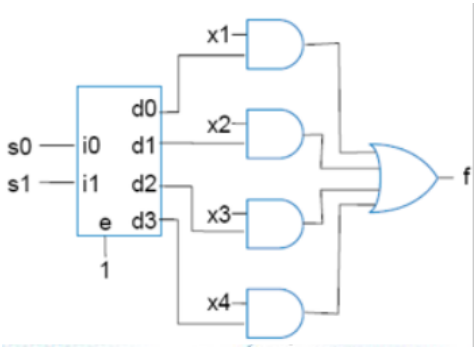
Enable e: 스위치 같은 것, e가 0이면 모든 output이 0이다.




**-4-to-1 MUX Using Decoder and Logic Gates**

**: Enable is not needed, set to 1**



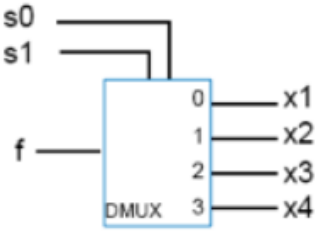



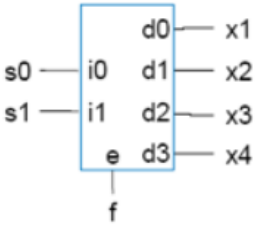

**-ROM Decoder**

: 10-1024 decoder 사용. address 10bit를 던지면 8bit의 memory로 구성된 것이 나온다.

1024개 * 8bit = 1k byte 짜리.

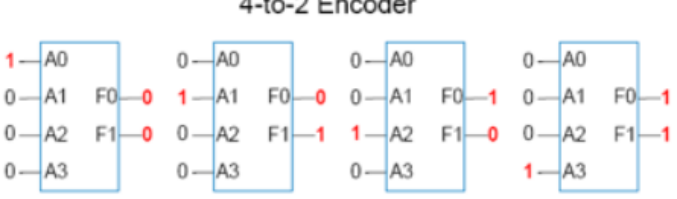
**-1-to-4 DEMUX Using Decoder**




1-to-4 Demultiplexer implemented with decoder

**-Encoder ( Opposite of Decoder )**



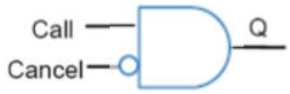

| A3 | A2 | A1 | A0 | F1 | F0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

# Lecture 09 (Memory, Sequential Logic)

* 래치는 레벨 동작(enable)의해 회로가 동작하는 타입, 플립플롭은 클록 엣지(CLK)에 의해 동작하는 타입입니다.

**-Bit Storage**

**Button-light stay doesn't work.**

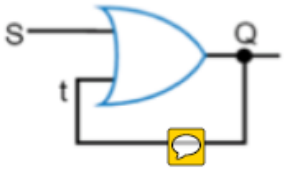



Doesn't work. Q=1 when Call=1, but
doesn't stay 1 when Call returns to 0

손을 떼는 순간 light이 꺼져버리지만 우리는 손을 뗀 순간에도 memory가 남아서 상태를 기억하여 불이 계속 켜져있길 원한다.

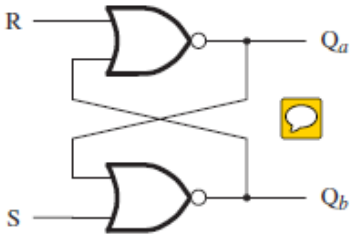
-> 이것을 logic으로 표현할수있는가? combinational로는 되지 않는다.

**-Feedback**

1. 

→ 뗐을 때도 1을 유지하지만 0으로 다시 변할 수가 없다.

2. SR Latch (2 nor gate)




| S | R | $Q_a$ | $Q_b$ | |
|---|---|---|---|---|
| 0 | 0 | 0/1 | 1/0 | (No change) |
| 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | |

| S | R | Q | 동작 |
|---|---|---|---|
| 0 | 0 | $Q_0$ | 상태유지 |
| 0 | 1 | 0 | Reset |
| 1 | 0 | 1 | Set |
| 1 | 1 | X | 사용불가 |

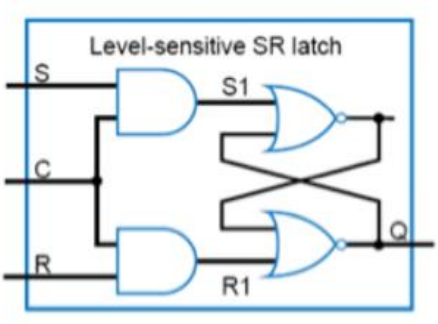
표. 1 SR NOR latch의 진리표

➔ S가 1이고 R도 1이면 output이 0 0 이 나온다. 0 0으로 놓으면 output이 1 1이 나오기 때문에 왔다갔다하며 폭주해버린다

➔ Q와 Q'는 보수 관계여야 하는데 둘다 0이 나온다.

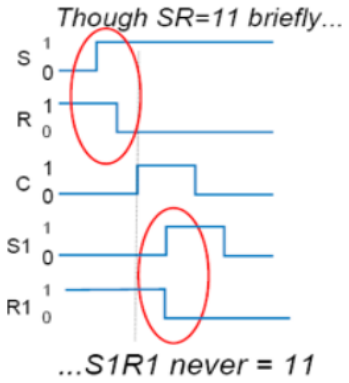
3. Gated / Level-Sensitive Latch

: Enable SR Latch 회로에 Enable 신호를 사용할 수 있게 만든 회로.




| S | R | E | Q | 동작 |
|---|---|---|---|---|
| 0 | 0 | 0 | $Q_0$ | 상태유지 |
| 0 | 0 | 1 | $Q_0$ | 상태유지 |
| 0 | 1 | 0 | $Q_0$ | 상태유지 |
| 0 | 1 | 1 | 0 | Reset |
| 1 | 0 | 0 | $Q_0(0)$ | 상태유지 |
| 1 | 0 | 1 | 1 | Set |
| 1 | 1 | 0 | $Q_0(1)$ | 상태유지 |
| 1 | 1 | 1 | X | 사용불가 |

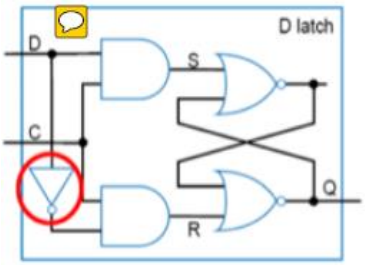
:E가 0이면 모두 상태 유지.




➔ S=1, R=1이라면 delay를 줘서 C를 0을 유지. 안정이 되면 C를 1로 만든다.

3. Level-Sensitive D-Latch (CLOCK이 1이면 D값 그대로 복사, 0이면 0)

: D value means set or reset / 기어조작을 수동에서 자동으로 바꾼 것.(D하나로다룰수있음)




| Truth table | | |
|---|---|---|
| Input | | Output |
| E | D | Q |
| 0 | X | Q |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

➔ C가 1이 되어야 변하기 때문에 시간이 오래 걸린다 ** latches을 계속 진행해야 함. 1이 될때까지. (클럭이 계속 켜져있는 것보다 클락이 변할 때의 q값을 저장)

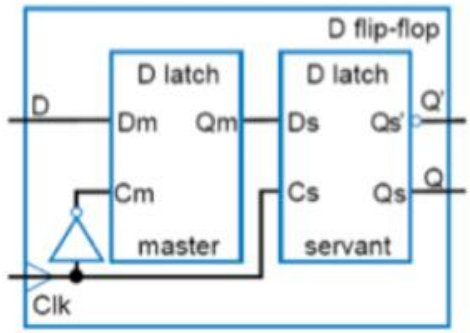
➔ Clock edge의 rising edge만 저장할 순 없을까? (level sensitive vs edge-triggered)

클록이 켜져있는 대신 클록 값이 변하는 순간에 Q가 변하게 하는 회로 구성

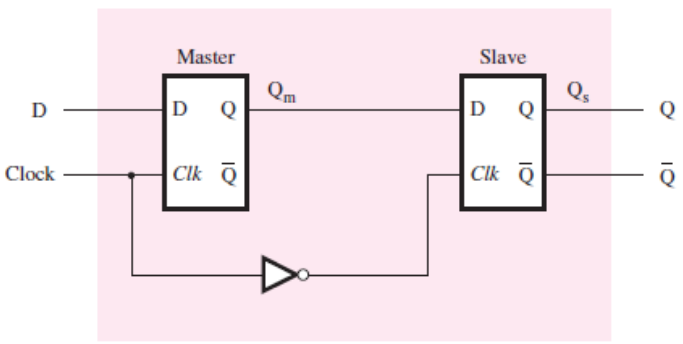
→

4. D Flip-Flop (rising edge) (clock edge , not level)

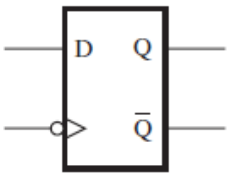
: 하나의 입력 단자가 있고 클록 펄스가 인가되었을 때 입력 신호가 1이면 1로 0이면 0으로 자리잡는 플립플롭. ( clock이 0에서 1이 될 때, rising할 때만.)




D Flip-Flop (falling edge) ( clock이 1에서 0이 될 때, falling할 때만.)



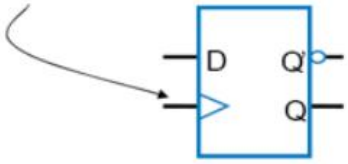

< Circuit >



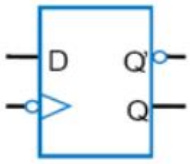

< Symbol >

*Edge Triggering

*The triangle means clock input, edge triggered*




Symbol for rising-edge triggered D flip-flop




Symbol for falling-edge triggered D flip-flop



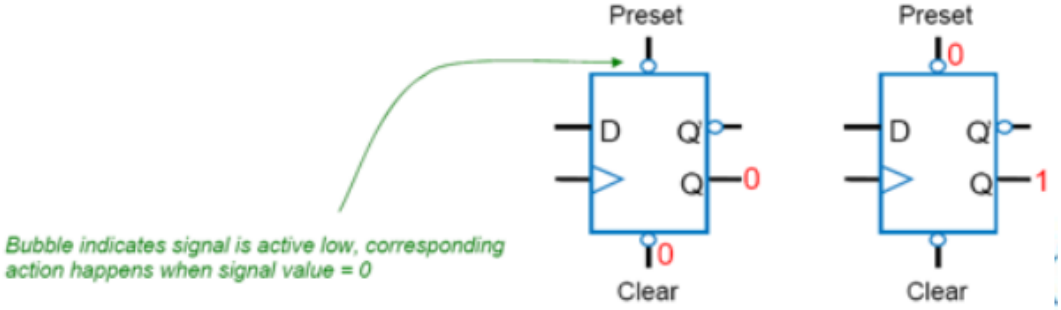
➔ Edge의 개수에 반응하는 회로를 만들 수 있다.

➔ 신호가 켜져있을 때 얼마나 많은 래치를 거치는지 모르는 문제를 해결.

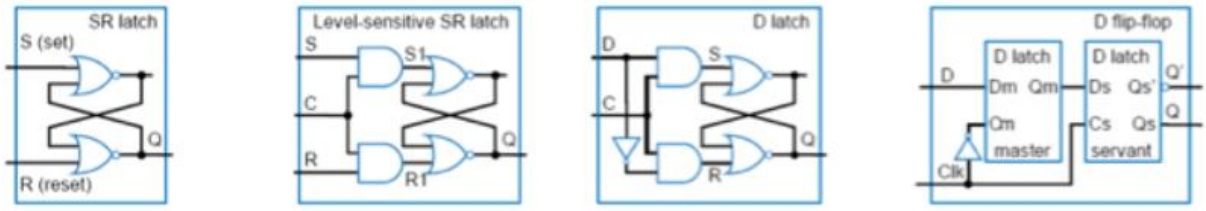
**D Flip-Flops with clear and preset(bubble 동그라미가 붙는다. 0일 때 진행)**

**: preset**해주고 싶으면 **0**을 넣으면 **1**로 이니셜라이즈

**clear**해주고 싶으면 **0**을 넣으면 **0**으로 이니셜라이즈.




**********중요**********




**Feature**: S=1 sets Q to 1, R=1 resets Q to 0.

**Problem**: SR=11 yield undefined Q.

**Feature**: S and R only have effect when C=1. We can design outside circuit so SR=11 never happens when C=1.

**Problem**: avoiding SR=11 can be a burden.

**Feature**: SR can't be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1.

**Problem**: C=1 too long propagates new values through too many latches: too short may not enable a store.

**Feature**: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle.

**Tradeoff**: uses more gates internally than D latch, and requires more external gates than SR – but gate count is less of an issue today.

4. T Flip-Flop

: T가 1이면 현재의 출력 값을 Toggle.

## -Register **(Multiple flip-flops sharing clock signal)**


***flip-flop**은 **clock**신호를 유지해주면 입력된 신호를 유지할 수 있다.

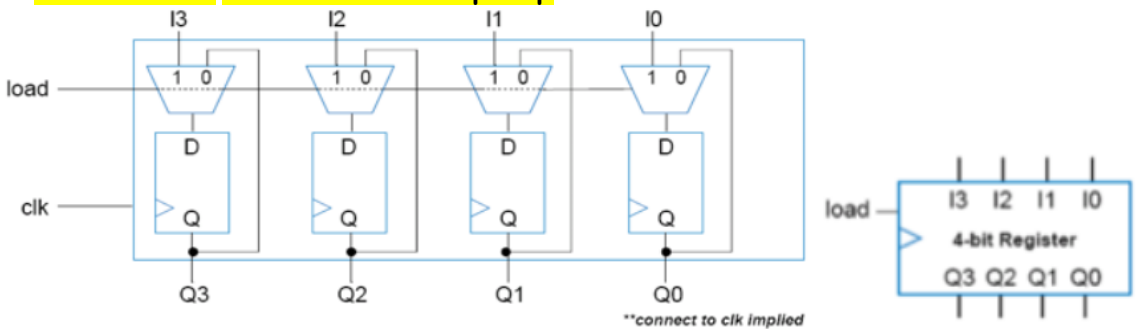
따라서 **n**개의 **flip-flop**을 병렬로 구성하고 클럭신호를 유지한다면 데이터를 **n**비트 저장할 수 있다.**-->** 이것을 **register**라고 부른다.

*4비트 병렬 레지스터는 데이터를 입력하면 상승 클럭일 때 바로 출력.-> 기존 입력된 데이터가 제거된다.

→ 데이터를 입력하고 저장을 해야 출력으로 반영이 되는 로드 기능 필요.

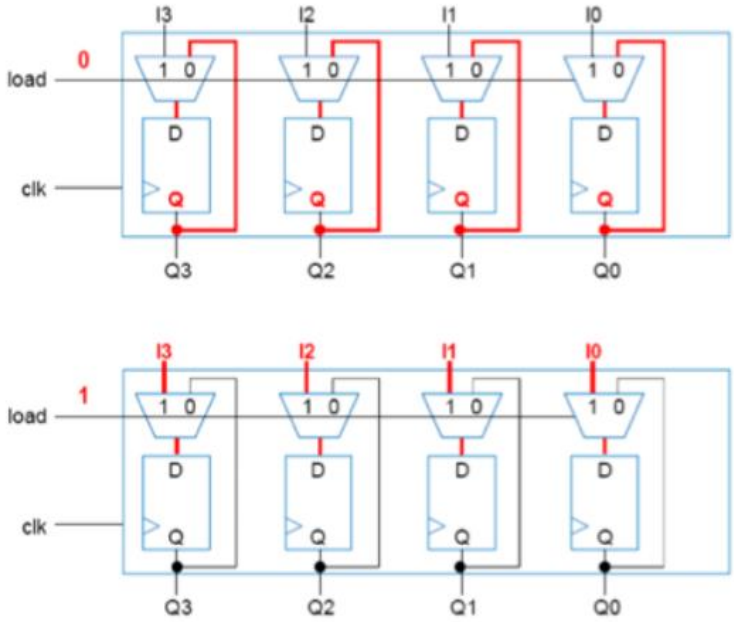
**-Register with Parallel Load**

**<Add 2x1 mux to front of each flip-flop>**




- **load = 0, existing flip-flop value**
- **load =1, new value to load**

## Load Selection




**-Mirror Display**

**4 8-bit values register**가 있다. 컴퓨터는 한번에 한 **value**만 **load**한다.

**2-to-4 decoder**로 어떤 것을 **load**할지 정한다.(**load**가 **set**되어있을 경우)

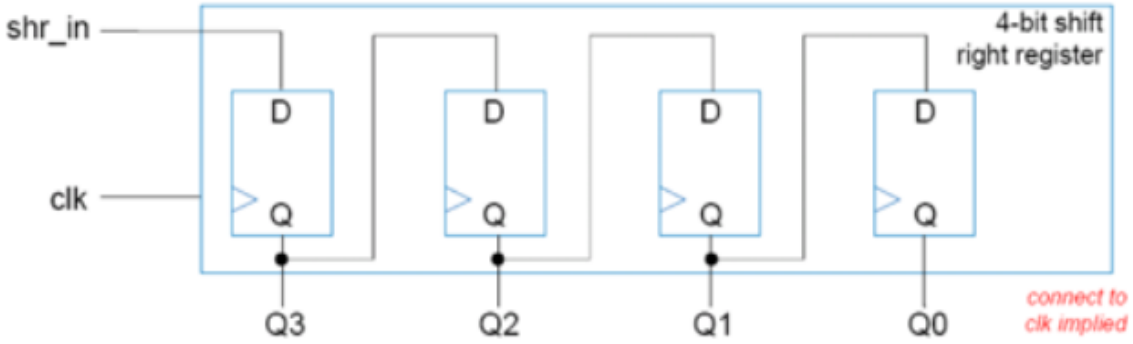
**4**개의 **register**중 하나만 **load**.

**4-to-1 MUX**를 이용해 어떤 **VALUE**를 보여줄지 정한다.

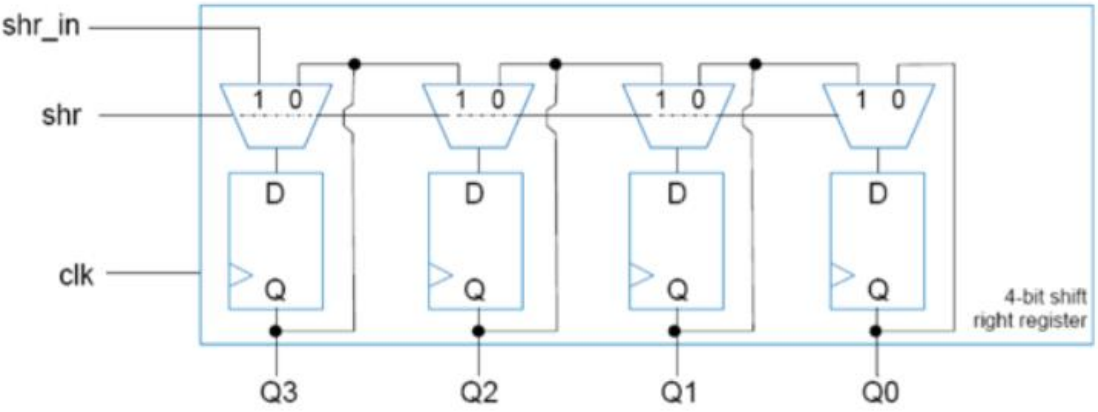
# Lecture 10

**-Shift Register**

: 플립플롭의 출력을 뒤에 있는 플립플롭의 입력에 차례로 연결하면 클럭이 들어올 때마다 저장된 데이터는 한 자리씩 이동한다.

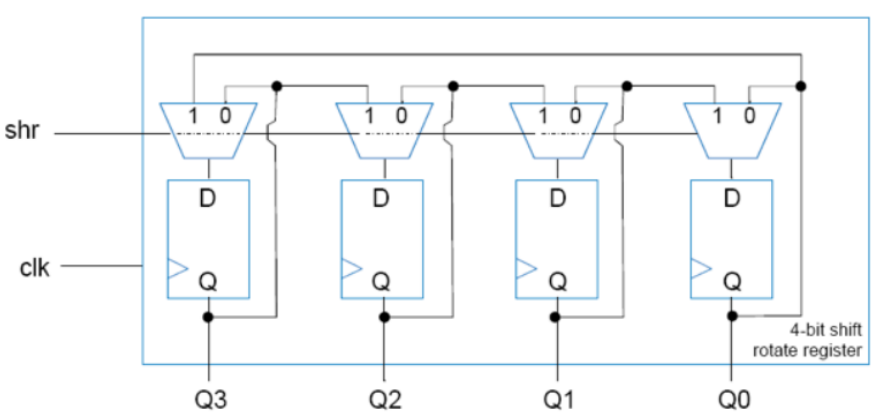



use 2x1 muxes→ retain할지 shift할지 결정해준다.

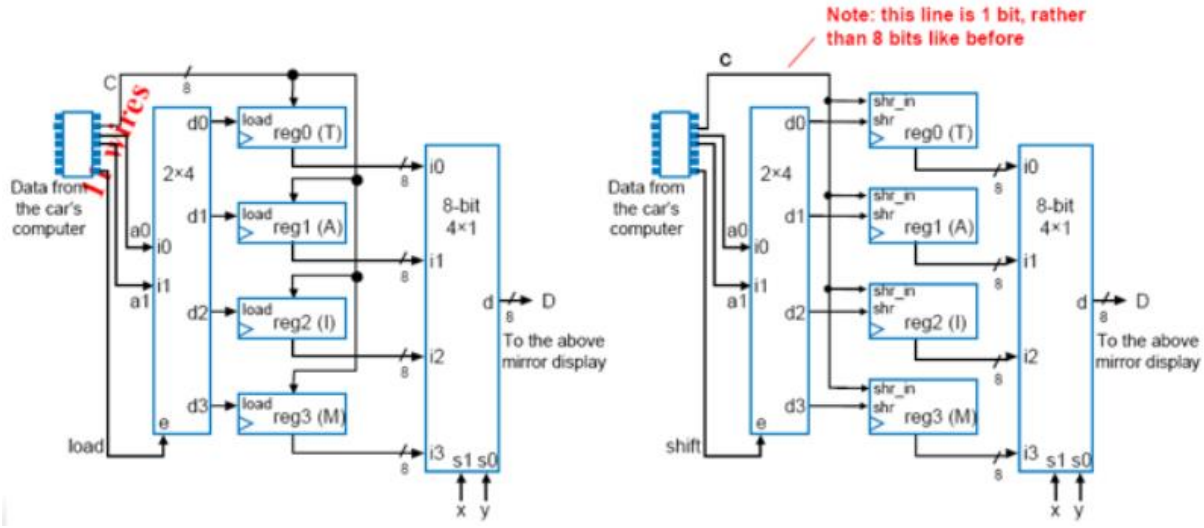



**-Rotate Register**

: 돌아가는 걸 볼 수 있음. 초기화 불가능. 무조건 뺑뺑이.



**-Mirror Display**




➔ **Shift register**을 사용함으로서 **8**비트를 **1**비트로 줄일 수 있다.

**-Multi Function Register**

Functions:

| s1 | s0 | Operation |
|---|---|---|
| 0 | 0 | Maintain present value |
| 0 | 1 | Parallel load |
| 1 | 0 | Shift right |
| 1 | 1 | (unused - let's load 0s) |

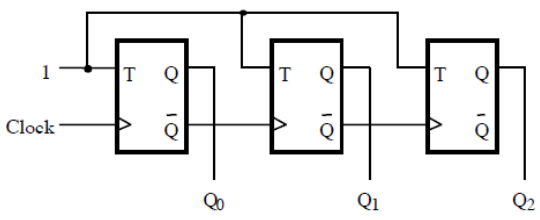
**s1=1,s0=1 will be reset to all 0**

**-Counters( T 플립플랍이용)**

: 클록의 펄스 엣지에 따라 플립플롭들에 의해 **2**진수의 숫자가 하나씩 증가하는 회로.

**-Asynchronous Counter :** 동시에 클럭이 입력되지 않음

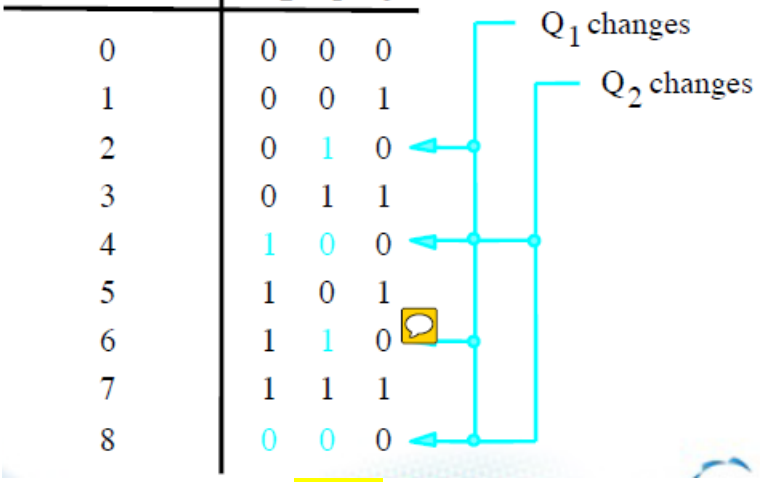
**0**에서 **1**로바뀔때는 다음 **clock**은 **0, 1**에서 **0**으로 바뀔때는 한자리가 올라가기 때문에 **clock**에 **1.**



모든 플립플롭이 클럭펄스에 의해 트리거 되지 않고 첫번째만 클럭에 의해 트리거되고 나머지는 앞단의 출력에 의해 트리거.--> 지연시간 발생.

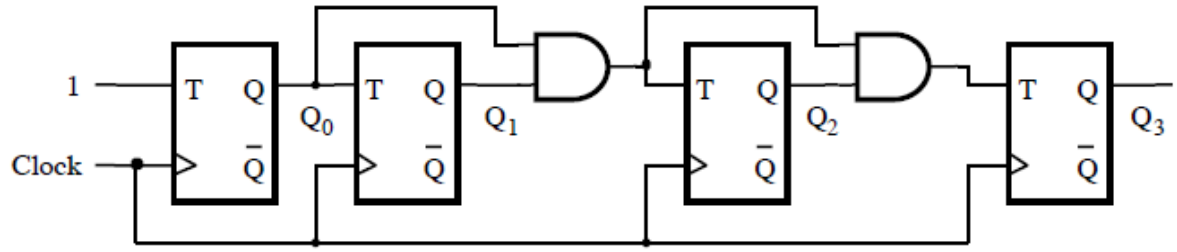
**-Synchronous Counter :** 클록을 모든 플립플롭에 동시에 인가.

| Clock cycle | $Q_2$ $Q_1$ $Q_0$ |
|---|---|
| 0 | 0 0 0 |
| 1 | 0 0 1 |
| 2 | 0 1 0 |
| 3 | 0 1 1 |
| 4 | 1 0 0 |
| 5 | 1 0 1 |
| 6 | 1 1 0 |
| 7 | 1 1 1 |
| 8 | 0 0 0 |




➔ **Q0**이 **1**일 때 다음에 **q1**은 **toggle**된다. **Q2**는 **q0**과 **q1**모두 **1**일 때 **toggle**된다.

➔ 변했다면 다음 **clock**에 **toggle**

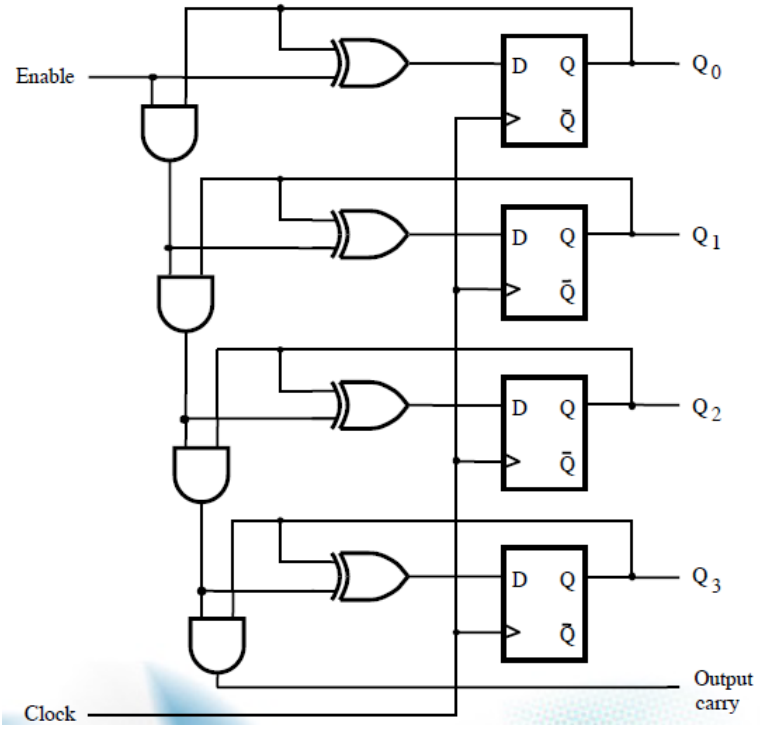



**-Counter with D-FF**

**Q0=D0 XOR E, Q1=D1 XOR (D0E), Q2=D2 XOR (D1D0E)**

**-modulo 6**이면 **0**부터 **5**까지 표현하면됨. → 비트가 **3**개필요.

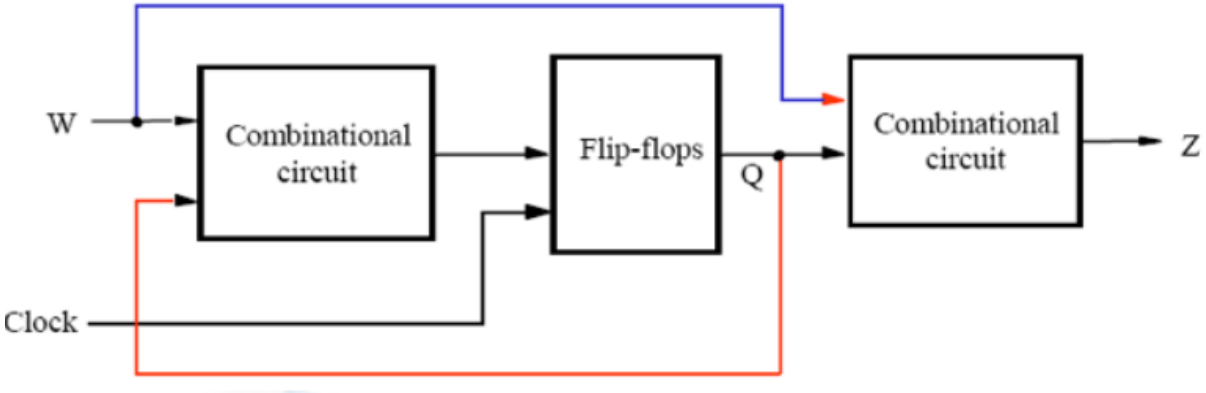
**5**가 되면 다시 **reset!**




# Lecture 11

- Two categories of sequential circuits
  - Synchronous – clock used to control operation of circuit
  - Asynchronous – no clock

**-The General Form of Sequential Circuit**

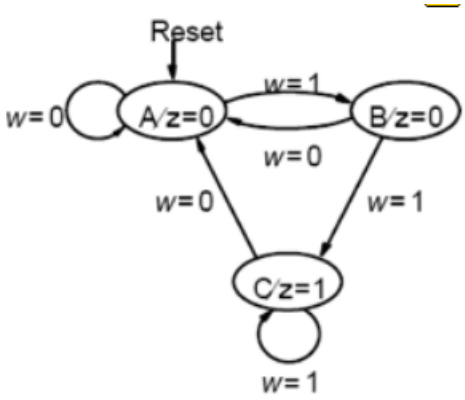



- **Sequential building blocks** : Latch (SR or D), Flip-flops (D, JK, T)
  - latch는 'control input = 1'이기만 하면 input의 상태에 따라 output값이 즉시 바뀜.
  - 하지만, flip-flop은 clock에 맞추어서만 바뀜.

그래서 latch에서 알아두어야 할 중요한 한가지는 clock이 0이면 Output Q값은 이전 상태의 값을 그대로 유지하고 clock이 1인 상태에서 입력 D값에 따라 Output Q값이 0(reset)아니면 1(set)값으로 바뀐다는 사실입니다.

**-FSM Definition**

**State, transition, input, action(output)**




| Present state | Next state | | Output |
|---|---|---|---|
| | $w = 0$ | $w = 1$ | $z$ |
| A | A | B | 0 |
| B | A | C | 0 |
| C | A | C | 1 |

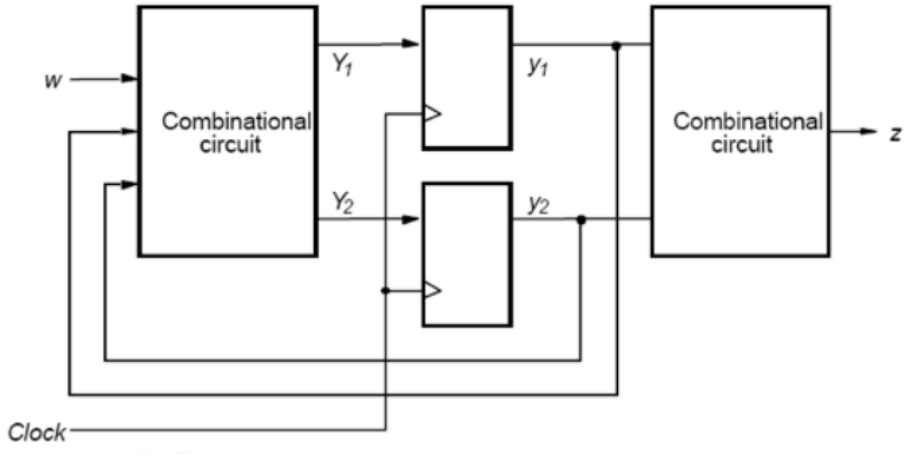
- State diagram defines three states: A, B, C
- Three states implies the use of two variables
  - *Y2, Y1*

| | Present state $y_2y_1$ | Next state $w = 0$ $Y_2Y_1$ | Next state $w = 1$ $Y_2Y_1$ | Output $z$ |
|---|---|---|---|---|
| A | 00 | 00 | 01 | 0 |
| B | 01 | 00 | 10 | 0 |
| C | 10 | 00 | 10 | 1 |
| | 11 | *dd* | *dd* | *d* |

Output logic - function of present state only

Output logic - function of present state and FSM inputs

*__output__이 __moore__는 현재 상태만을 고려하지만 __mealy__는 현재상태와__input__값도 고려한다. 무어가 더안전. __Mealy__는 __asynchronous__의 문제가 생길 수 있다. 하지마ㄴ __moore__가 더 복잡.




- A general sequential circuit with input w, output z, and two state flip-flops