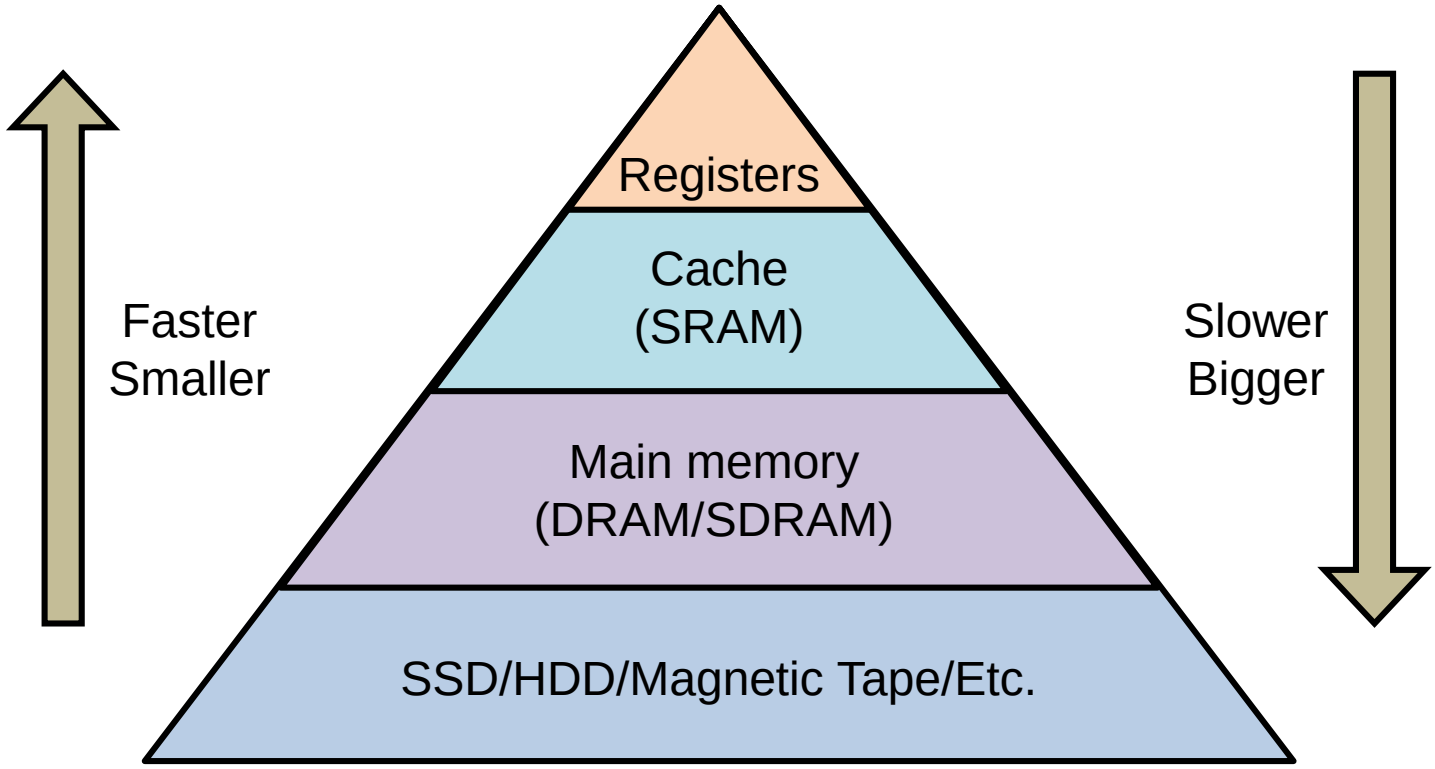
Lecture 09~10

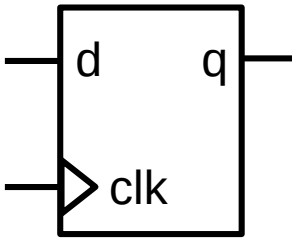
RAM

메모리 계층

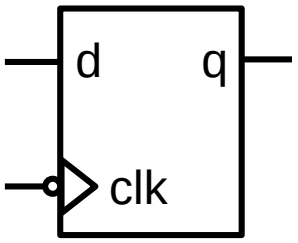

Registers
Cache
(SRAM)
Main memory
(DRAM/SDRAM)
SSD/HDD/Magnetic Tape/Etc.
Faster
Smaller
Slower
Bigger

기본 메모리 소자

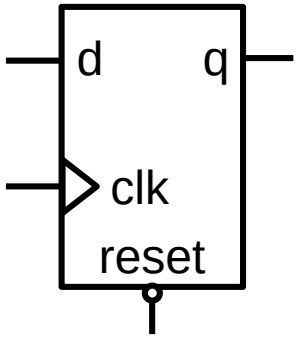
- Positive-edge-triggered D FF
- Negative-edge-triggered D FF
- 비동기식 reset D FF



clk	q^{next}
0	q
1	q
↑(rising edge)	d



clk	q^{next}
0	q
1	q
↓(falling edge)	d



reset	clk	q^{next}
0	-	0
1	0	q
1	1	q
1	↑(rising edge)	d

기본 메모리 소자

- Verilog 코드

D FF

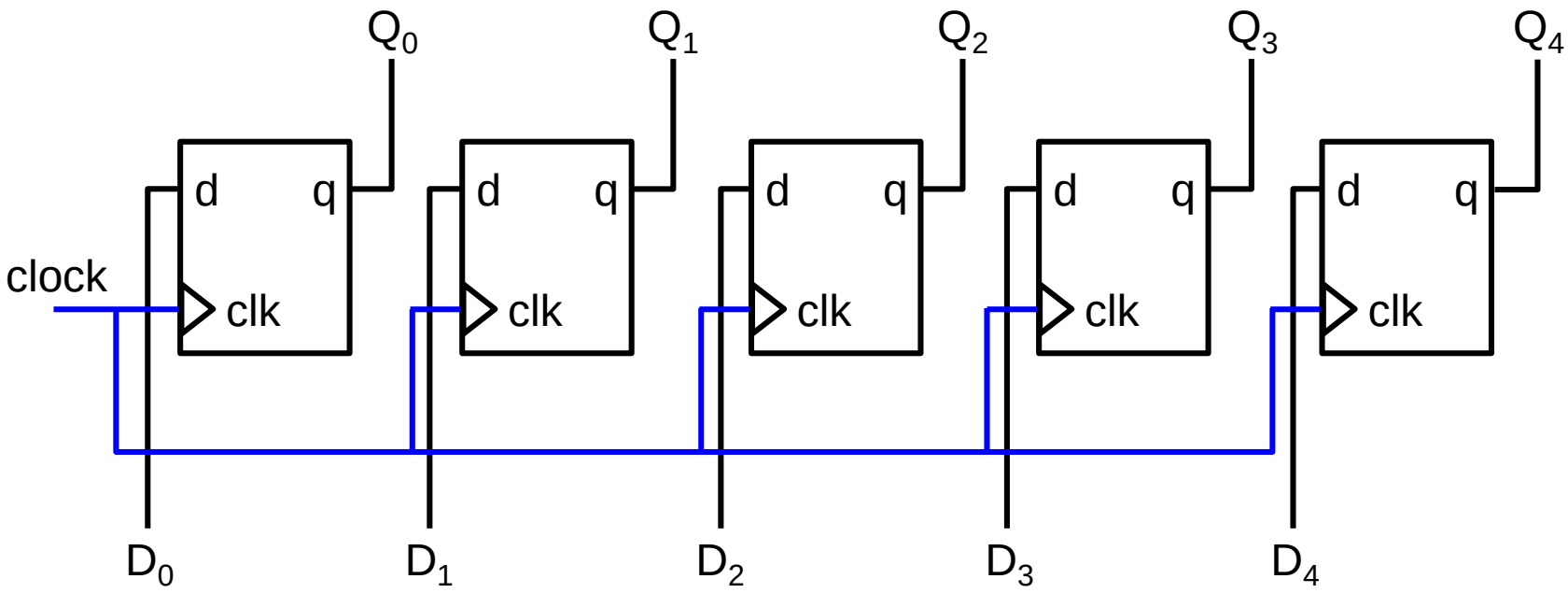
```
module d_ff (
    input       clk,
    input       d,
    output reg q
);

always @(posedge clk) begin
    q <= d;
end

endmodule
```

비동기식 reset D FF

```
module d_ff_rstb (
    input       clk,
    input       rstb,
    input       d,
    output reg q
);

always @(posedge clk or negedge rstb) begin
    if (~rstb) q <= 0;
    else       q <= d;
end

endmodule
```

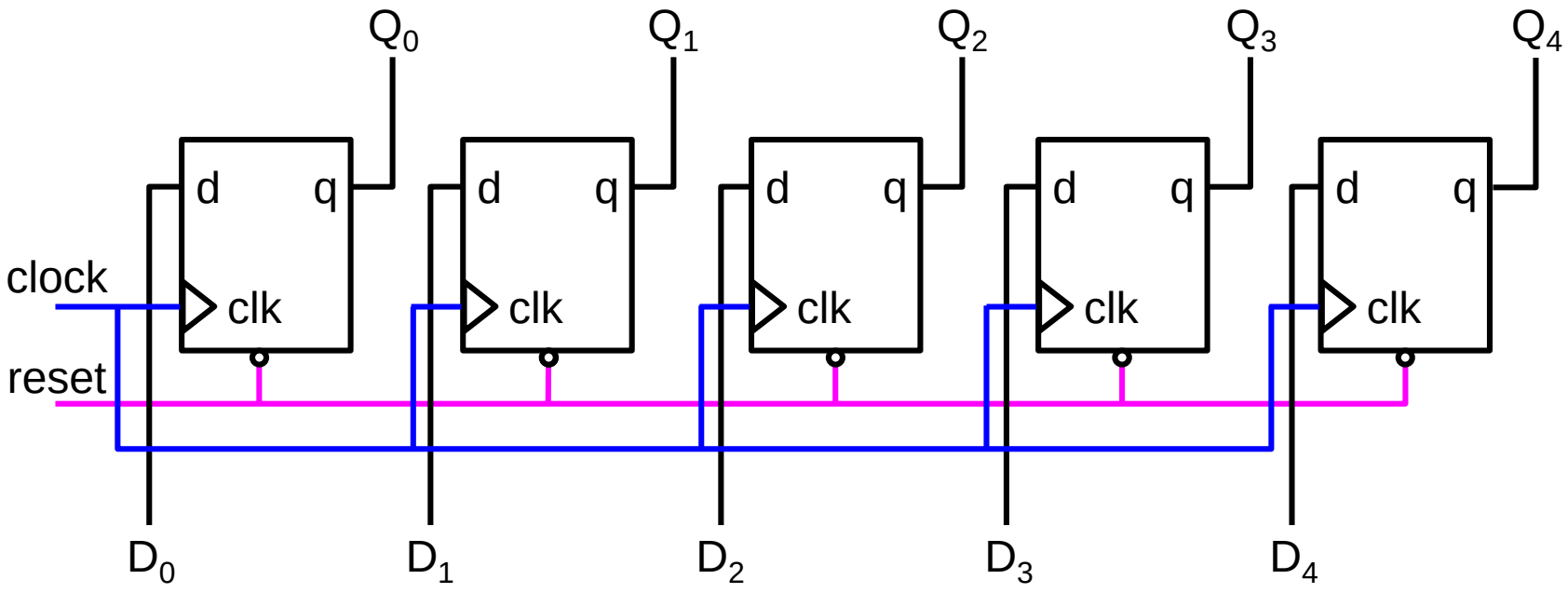
레지스터

- D FF 하나 ↔ 1비트 register
- D FF N개 ↔ N비트 register



레지스터

- D FF 하나 ↔ 1비트 register
- D FF N개 ↔ N비트 register



레지스터

- Verilog 코드

D FF로 구성된 레지스터

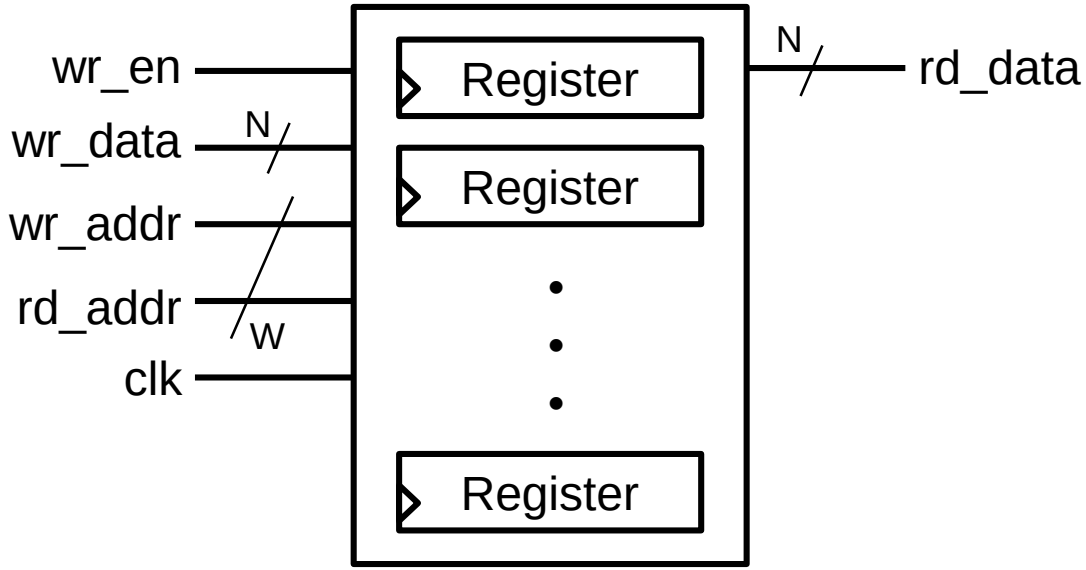
```
module reg
#(
    parameter N = 8
) (
    input               clk,
    input       [N-1:0] d,
    output reg  [N-1:0] q
);

always @(posedge clk) begin
    q <= d;
end

endmodule
```

비동기식 reset D FF로 구성된 레지스터

```
module reg_rstb
#(
    parameter N = 8
) (
    input               clk,
    input               rstb,
    input       [N-1:0] d,
    output reg  [N-1:0] q
);

always @(posedge clk or negedge rstb) begin
    if (~rstb) q <= 0;
    else       q <= d;
end

endmodule
```

레지스터 파일

- 일시 저장을 위한 빠른 메모리
- 용량이 크지 않음



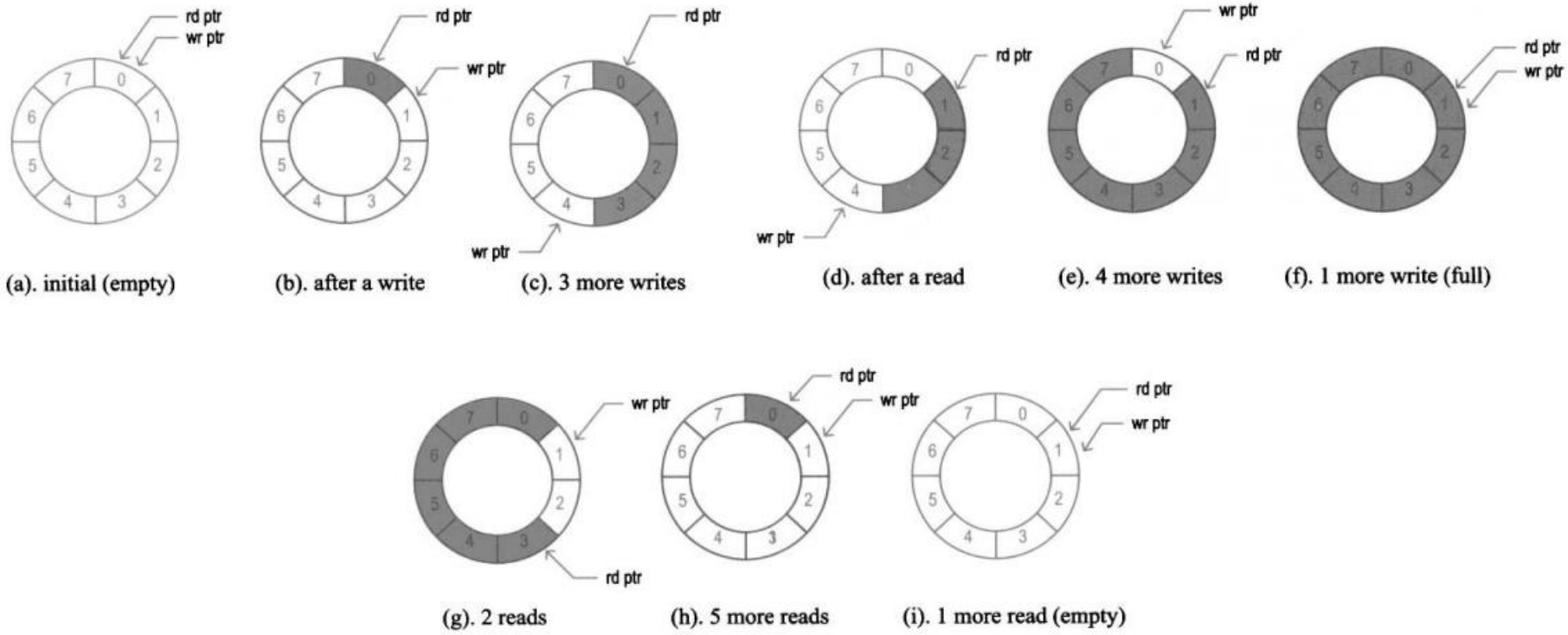
레지스터 파일

- Verilog 코드

```
module register_file
#(
    parameter N = 8,
              W = 2
) (
    input           clk,
    input           wr_en,
    input  [W-1:0] wr_addr, rd_addr,
    input  [N-1:0] wr_data,
    output [N-1:0] rd_data
);

// signal declaration
reg [N-1:0] array_reg[2**W-1:0];

// write operation
always @(posedge clk) begin
    if (wr_en)
        array_reg[wr_addr] <= wr_data;
end

// read operation
assign rd_data = array_reg[rd_addr];

endmodule
```

레지스터 파일

- 레지스터 파일 기반 FIFO 구현



레지스터 파일

- 레지스터 파일 기반 FIFO 구현

```
module fifo #(
   parameter N = 8, // number of bits
             W = 4  // number of addr bits
) (
   input           clk, rstb,
   input           rd, wr,
   input  [N-1:0] w_data,
   output          empty, full,
   output [N-1:0] r_data
);

// signal declaration
reg [N-1:0] register_file[2**W-1:0];
reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
reg         full_reg, empty_reg, full_next, empty_next;
wire        wr_en;
```

```
// register file operation
always @(posedge clk) begin
   if (wr_en)
      register_file[w_ptr_reg] <= w_data;
   end
assign r_data = register_file[r_ptr_reg];
assign wr_en = wr & ~full_reg; // registers

always @(posedge clk or negedge rstb) begin
   if (~rstb) begin
      w_ptr_reg <= 0;
      r_ptr_reg <= 0;
      full_reg <= 0;
      empty_reg <= 1;
   end
   else begin
      w_ptr_reg <= w_ptr_next;
      r_ptr_reg <= r_ptr_next;
      full_reg <= full_next;
      empty_reg <= empty_next;
   end
end
```

레지스터 파일

- 레지스터 파일 기반 FIFO 구현

```
// next-state logic
always @* begin
   w_ptr_succ = w_ptr_reg+1;
   r_ptr_succ = r_ptr_reg+1;
   // default values
   w_ptr_next = w_ptr_reg;
   r_ptr_next = r_ptr_reg;
   full_next = full_reg;
   empty_next = empty_reg;

   case ({wr, rd})
      // 2'b00: no operation
      2'b01: // read
         if (~empty_reg) begin
            r_ptr_next = r_ptr_succ;
            full_next = 0;
            if (r_ptr_next==w_ptr_reg)
               empty_next = 1;
         end
      2'b10: // write
         if (~full_reg) begin
            w_ptr_next = w_ptr_succ;
            empty_next = 0;
            if (w_ptr_next==r_ptr_reg)
               full_next = 1;
         end

      2'b11: // write and read
         begin
            w_ptr_next = w_ptr_succ;
            r_ptr_next = r_ptr_succ;
         end
   endcase
end

assign full = full_reg;
assign empty = empty_reg;

endmodule
```

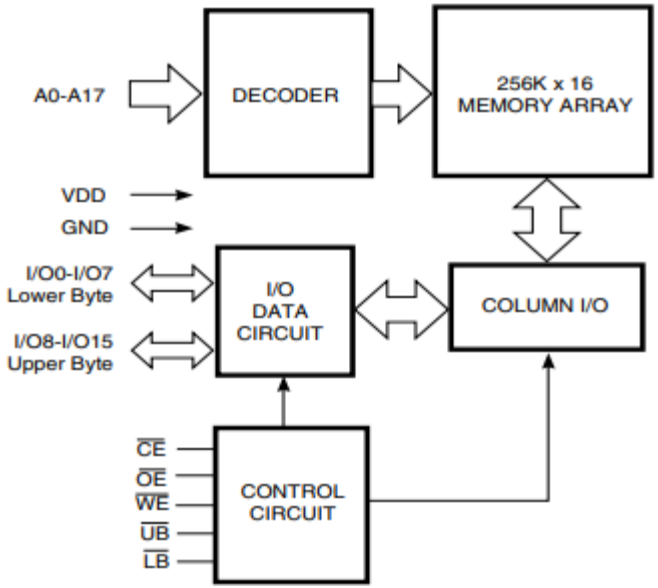
RAM(Random Access Memory)

- 대용량 저장을 위한 메모리 장치
- RAM 소자는 D FF보다 훨씬 간단함
- RAM은 크게 **DRAM(dynamic RAM)**와 **SRAM(static RAM)**로 나누며, SRAM이 널리 쓰인 RAM임
- RAM은 칩 속에 포함되면 **내장 메모리(on-chip memory)**, 칩 박에 있으면 **외장 메모리(external memory)**라고 함
- 메모리 접근의 복잡도 : 레지스터 파일 < on-chip RAM < external RAM
- External RAM을 접근하기 위해 **메모리 컨트롤러**를 사용해야 함

RAM(Random Access Memory)

- 실제 external RAM 장치
  - IS61LV25616AL, 제작사 ISSI(Integrated Silicon Solution Inc.)
  - 256k-by-16 SRAM, 용량 512kB
  - Datasheet : https://www.issi.com/ww/pdf/61lv25616al.pdf

$\overline{CE}$ (Chip Enable)	칩 접근 허용 (0 : **O**, 1 : **X**)
$\overline{OE}$ (Output Enable)	데이터 출력 허용 (0 : **O**, 1 : **X**)
$\overline{WE}$ (Write Enable)	데이터 쓰기 허용 (0 : **O**, 1 : **X**)
$\overline{UB}$ (Upper Byte Enable)	출력의 상위 byte 허용 (0 : **O**, 1 : **X**)
$\overline{LB}$ (Lower Byte Enable)	출력의 하위 byte 허용 (0 : **O**, 1 : **X**)



RAM(Random Access Memory)

- 실제 external RAM 장치
  - IS61LV25616AL, 제작사 ISSI(Integrated Silicon Solution Inc.)
  - 256k-by-16 SRAM, 용량 512kB
  - Datasheet : https://www.issi.com/ww/pdf/61lv25616al.pdf

TRUTH TABLE

Mode	$\overline{WE}$	$\overline{CE}$	$\overline{OE}$	$\overline{LB}$	$\overline{UB}$	I/O PIN I/O0-I/O7	I/O PIN I/O8-I/O15	V_{DD} Current
Not Selected	X	H	X	X	X	High-Z	High-Z	I_{SB1}, I_{SB2}
Output Disabled	H	L	H	X	X	High-Z	High-Z	I_{CC}
	X	L	X	H	H	High-Z	High-Z	
Read	H	L	L	L	H	D_{OUT}	High-Z	I_{CC}
	H	L	L	H	L	High-Z	D_{OUT}	
	H	L	L	L	L	D_{OUT}	D_{OUT}	
Write	L	L	X	L	H	D_{IN}	High-Z	I_{CC}
	L	L	X	H	L	High-Z	D_{IN}	
	L	L	X	L	L	D_{IN}	D_{IN}	

RAM(Random Access Memory)

- 실제 external RAM 장치
  - 읽기 관련 타이밍

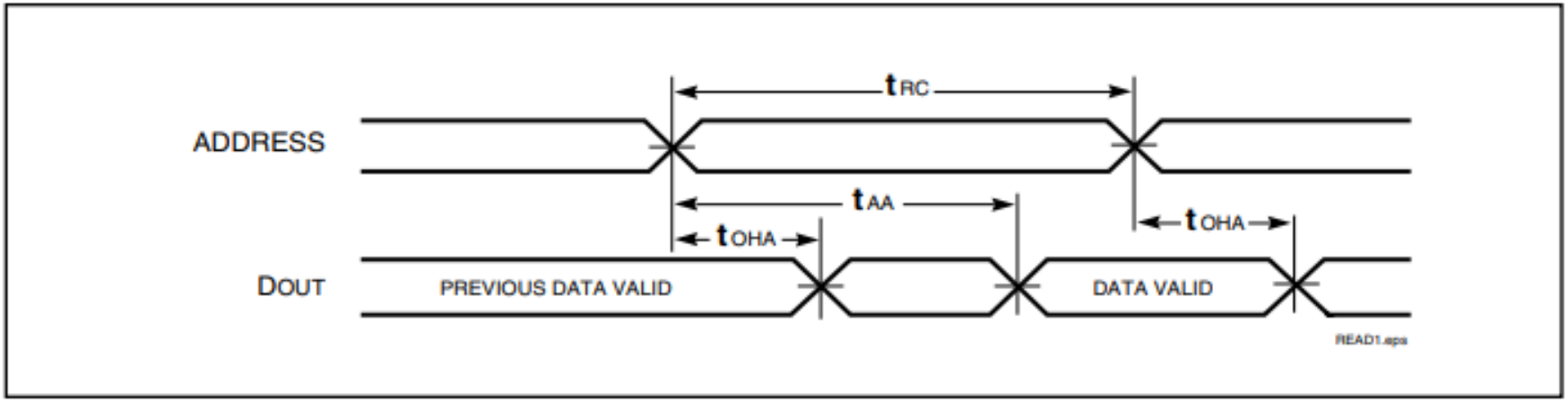
READ CYCLE SWITCHING CHARACTERISTICS[1] (Over Operating Range)

Symbol	Parameter	-10 Min.	-10 Max.	-12 Min.	-12 Max.	Unit
t_{RC}	Read Cycle Time	10	—	12	—	ns
t_{AA}	Address Access Time	—	10	—	12	ns
t_{OHA}	Output Hold Time	2	—	2	—	ns
t_{ACE}	$\overline{CE}$ Access Time	—	10	—	12	ns
t_{DOE}	$\overline{OE}$ Access Time	—	4	—	5	ns
t_{HZOE}[2]	$\overline{OE}$ to High-Z Output	—	4	—	5	ns
t_{LZOE}[2]	$\overline{OE}$ to Low-Z Output	0	—	0	—	ns
t_{HZCE}[2]	$\overline{CE}$ to High-Z Output	0	4	0	6	ns
t_{LZCE}[2]	$\overline{CE}$ to Low-Z Output	3	—	3	—	ns
t_{BA}	$\overline{LB}$, $\overline{UB}$ Access Time	—	4	—	5	ns
t_{HZB}[2]	$\overline{LB}$, $\overline{UB}$ to High-Z Output	0	3	0	4	ns
t_{LZB}[2]	$\overline{LB}$, $\overline{UB}$ to Low-Z Output	0	—	0	—	ns
t_{PU}	Power Up Time	0	—	0	—	ns
t_{PD}	Power Down Time	—	10	—	12	ns

RAM(Random Access Memory)

- 실제 external RAM 장치
  - 읽기 관련 타이밍

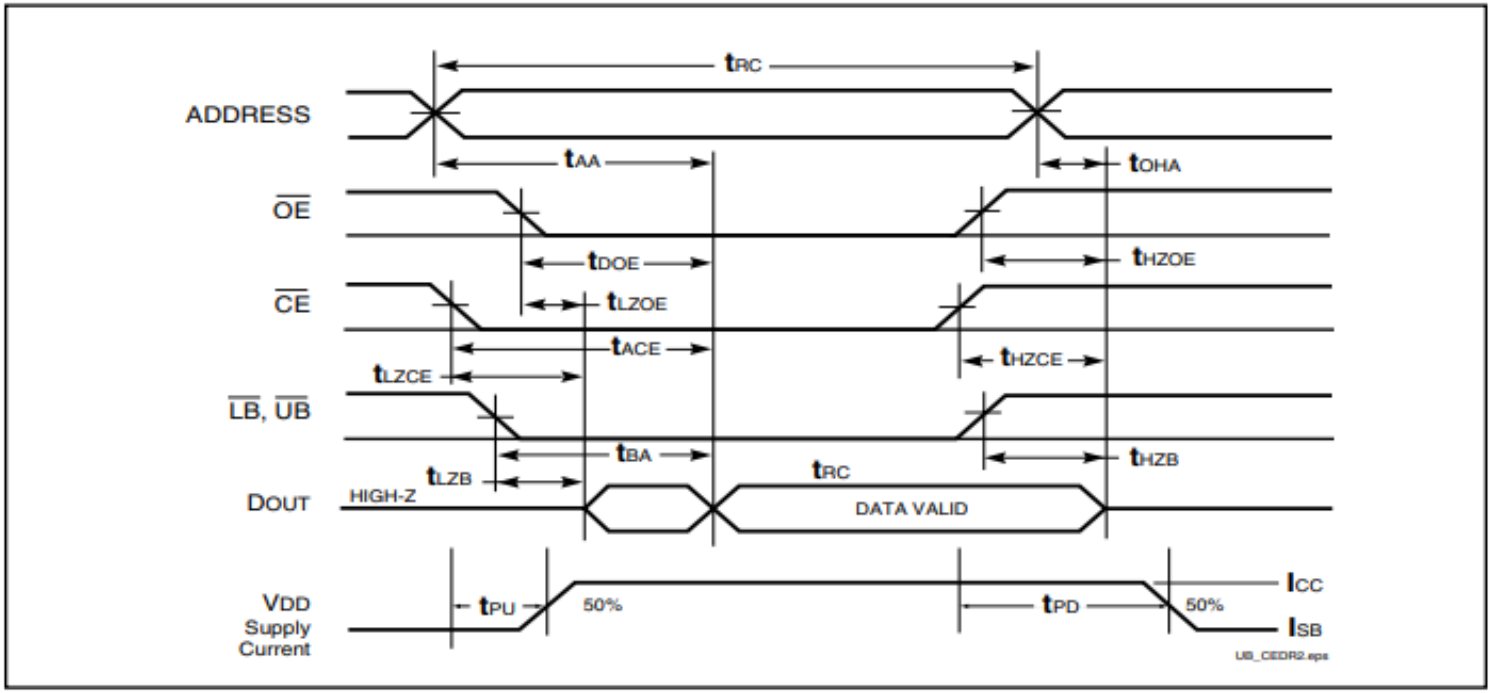
READ CYCLE NO. 1[1,2] (Address Controlled) ($\overline{CE}$ = $\overline{OE}$ = V_{IL}, $\overline{UB}$ or $\overline{LB}$ = V_{IL})



RAM(Random Access Memory)

- 실제 external RAM 장치
  - 읽기 관련 타이밍

READ CYCLE NO. 2(1,3)



RAM(Random Access Memory)

- 실제 external RAM 장치
  - 쓰기 관련 타이밍

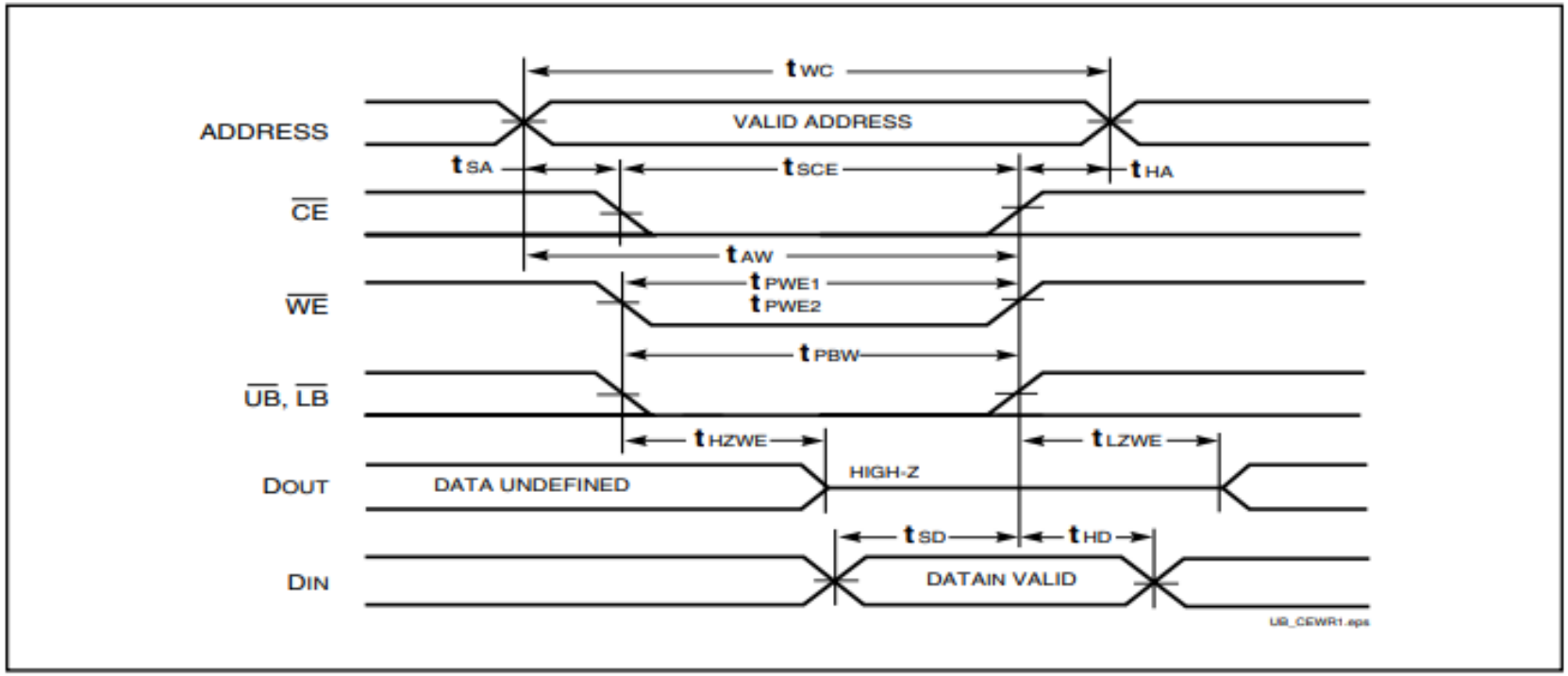
WRITE CYCLE SWITCHING CHARACTERISTICS[1,3] (Over Operating Range)

Symbol	Parameter	-10 Min.	-10 Max.	-12 Min.	-12 Max.	Unit
t_{WC}	Write Cycle Time	10	—	12	—	ns
t_{SCE}	$\overline{CE}$ to Write End	8	—	8	—	ns
t_{AW}	Address Setup Time to Write End	8	—	8	—	ns
t_{HA}	Address Hold from Write End	0	—	0	—	ns
t_{SA}	Address Setup Time	0	—	0	—	ns
t_{PWB}	$\overline{LB}$, $\overline{UB}$ Valid to End of Write	8	—	8	—	ns
t_{PWE1}	$\overline{WE}$ Pulse Width	8	—	8	—	ns
t_{PWE2}	$\overline{WE}$ Pulse Width ($\overline{OE}$ = LOW)	10	—	12	—	ns
t_{SD}	Data Setup to Write End	6	—	6	—	ns
t_{HD}	Data Hold from Write End	0	—	0	—	ns
t_{HZWE}[2]	$\overline{WE}$ LOW to High-Z Output	—	5	—	6	ns
t_{LZWE}[2]	$\overline{WE}$ HIGH to Low-Z Output	2	—	2	—	ns

RAM(Random Access Memory)

- 실제 external RAM 장치
  - 쓰기 관련 타이밍

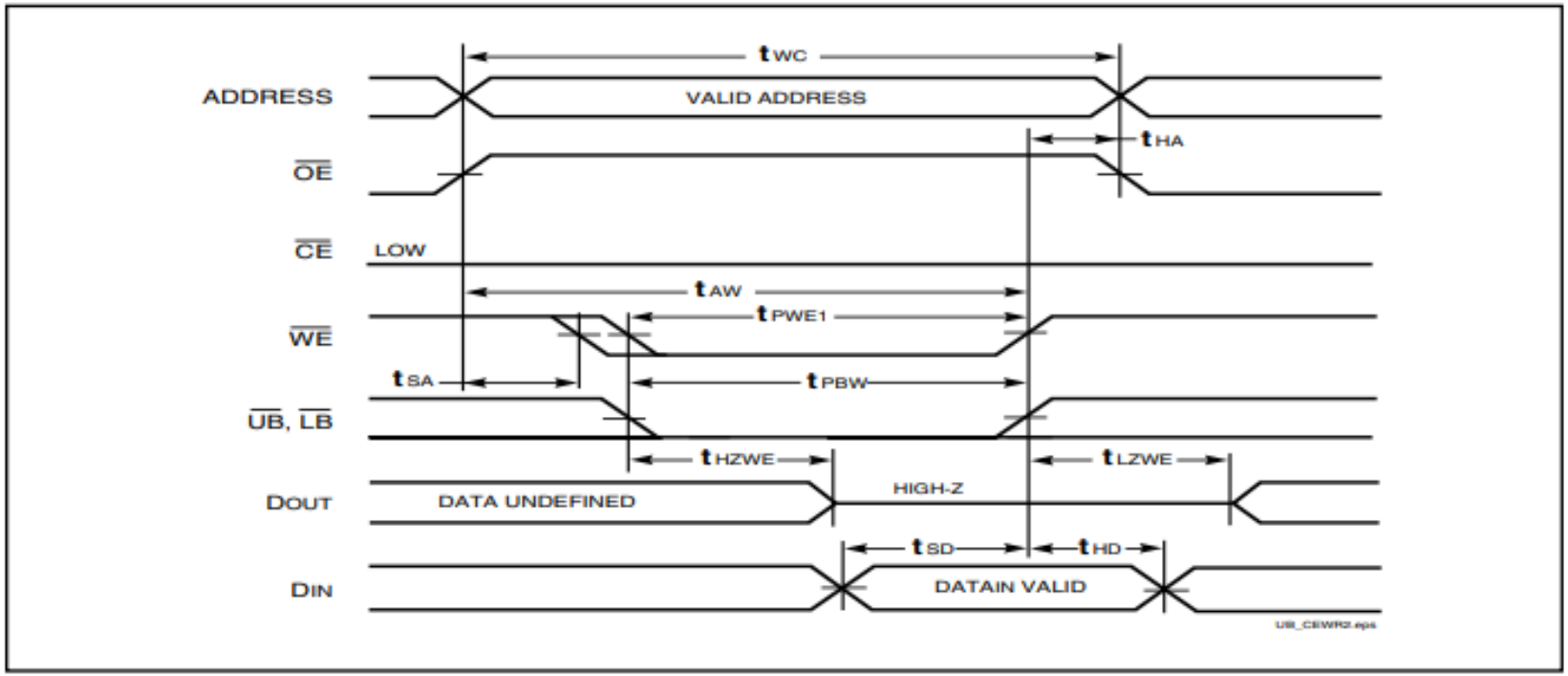
WRITE CYCLE NO. 1 ($\overline{CE}$ Controlled, $\overline{OE}$ is HIGH or LOW) (1)



RAM(Random Access Memory)

- 실제 external RAM 장치
  - 쓰기 관련 타이밍

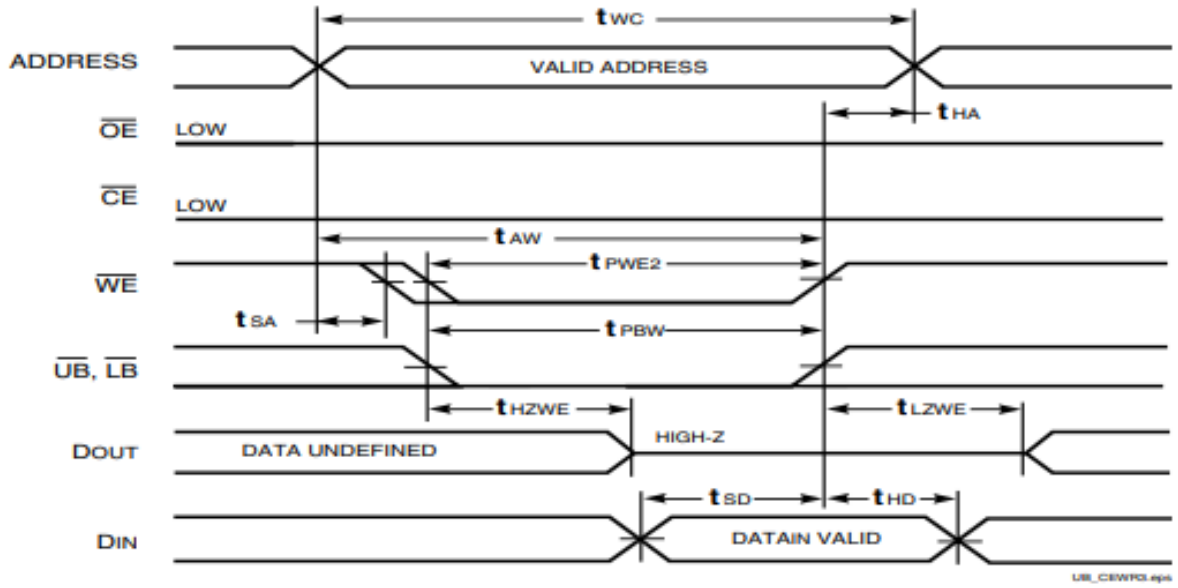
WRITE CYCLE NO. 2 ($\overline{WE}$ Controlled. $\overline{OE}$ is HIGH During Write Cycle) (1,2)



RAM(Random Access Memory)

- 실제 external RAM 장치
  - 쓰기 관련 타이밍

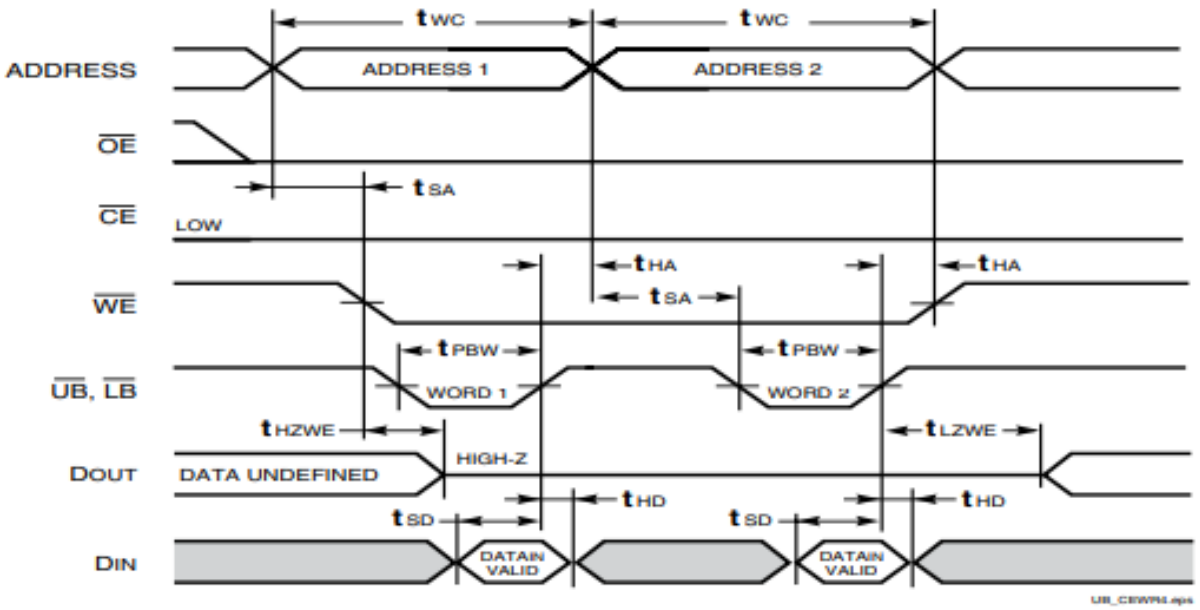
WRITE CYCLE NO. 3 ($\overline{WE}$ Controlled. $\overline{OE}$ is LOW During Write Cycle) (1)



RAM(Random Access Memory)

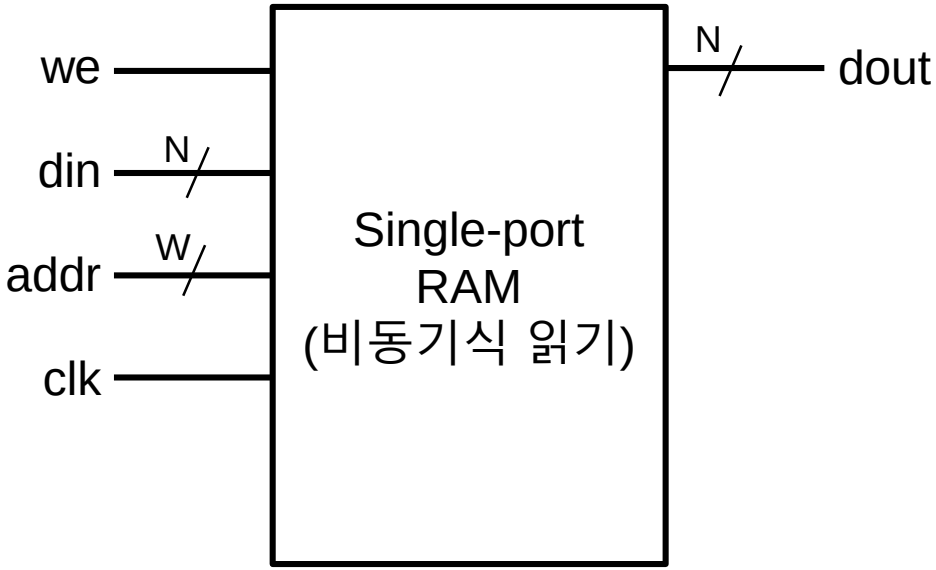
- 실제 external RAM 장치
  - 쓰기 관련 타이밍

WRITE CYCLE NO. 4 ($\overline{LB}$, $\overline{UB}$ Controlled, Back-to-Back Write) [1,3]



RAM(Random Access Memory)

- On-chip RAM
  - 비동기식 읽기 single-port RAM



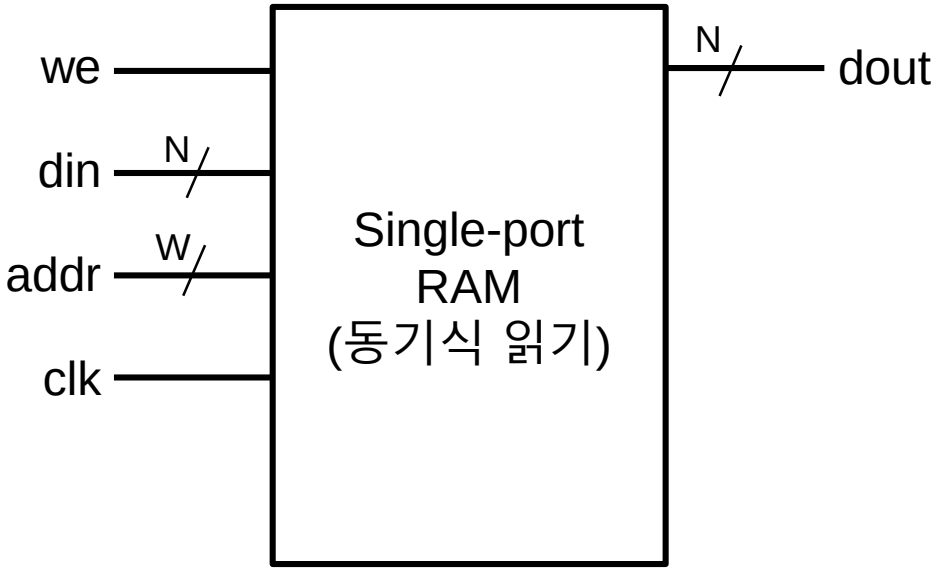
RAM(Random Access Memory)

- Verilog 코드로 on-chip RAM 합성
  - 비동기 읽기 single-port RAM

```
module sp_ram_async_read
#(
   parameter N = 8,
             W = 2
) (
   input           clk,
   input           we,
   input  [W-1:0]  addr,
   input  [N-1:0]  din,
   output [N-1:0]  dout
);

// signal declaration
reg [N-1:0] ram[2**W-1:0];

// write operation
always @(posedge clk) begin
   if (we)
      ram[addr] <= din;
end

// read operation
assign dout = ram[addr];

endmodule
```

RAM(Random Access Memory)

- On-chip RAM
  - 동기식 읽기 single-port RAM



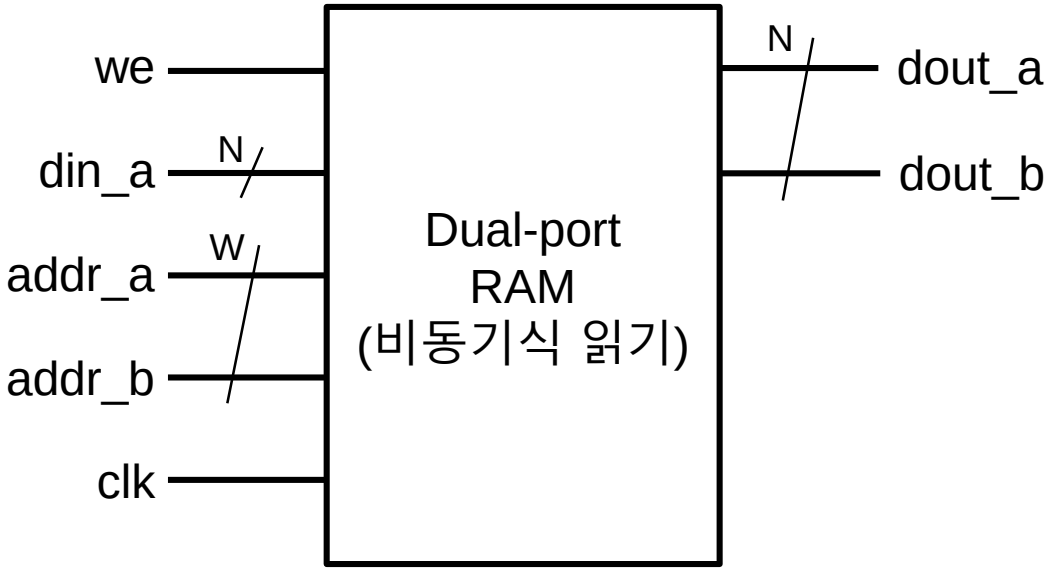
RAM(Random Access Memory)

- Verilog 코드로 on-chip RAM 합성
  - 동기 읽기 single-port RAM

```
module sp_ram_sync_read
#(
   parameter N = 8,
             W = 2
) (
   input           clk,
   input           we,
   input  [W-1:0]  addr,
   input  [N-1:0]  din,
   output [N-1:0]  dout
);

// signal declaration
reg [N-1:0] ram[2**W-1:0];
reg [W-1:0] addr_reg;

// write operation
always @(posedge clk) begin
   if (we)
      ram[addr] <= din;
   addr_reg <= addr;
end

// read operation
assign dout = ram[addr_reg];

endmodule
```

RAM(Random Access Memory)

- On-chip RAM
  - 비동기식 읽기 dual-port RAM



RAM(Random Access Memory)

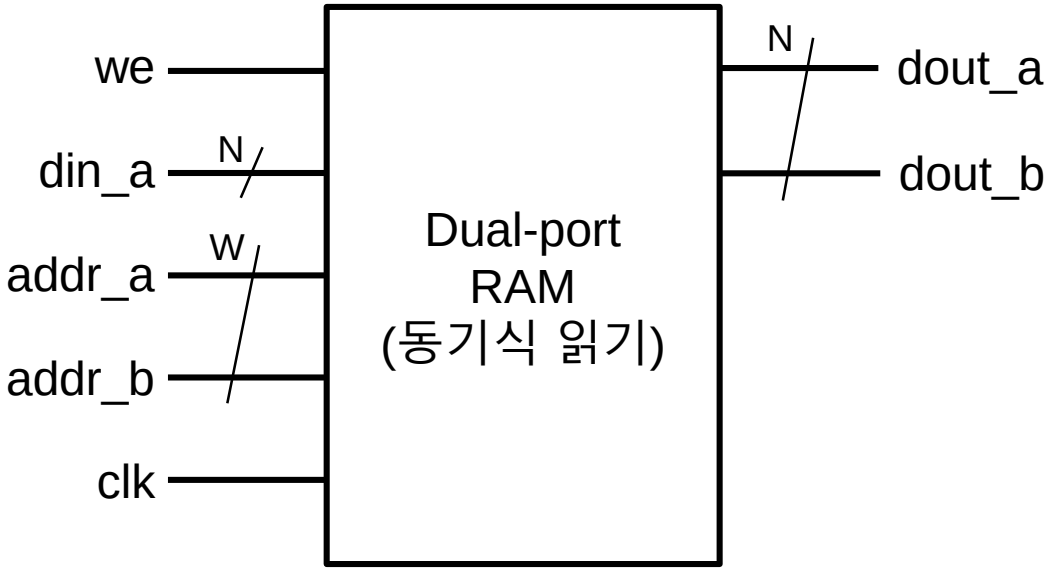
- Verilog 코드로 on-chip RAM 합성
  - 비동기 읽기 dual-port RAM

```
module dp_ram_async_read
#(
   parameter N = 8,
             W = 2
) (
   input           clk,
   input           we,
   input  [W-1:0] addr_a, addr_b,
   input  [N-1:0] din_a,
   output [N-1:0] dout_a, dout_b
);
```

```
// signal declaration
reg [N-1:0] ram[2**W-1:0];

// write operation
always @(posedge clk) begin
   if (we)
      ram[addr_a] <= din_a;
end

// read operation
assign dout_a = ram[addr_a];
assign dout_b = ram[addr_b];

endmodule
```

RAM(Random Access Memory)

- On-chip RAM
  - 동기식 읽기 dual-port RAM



RAM(Random Access Memory)

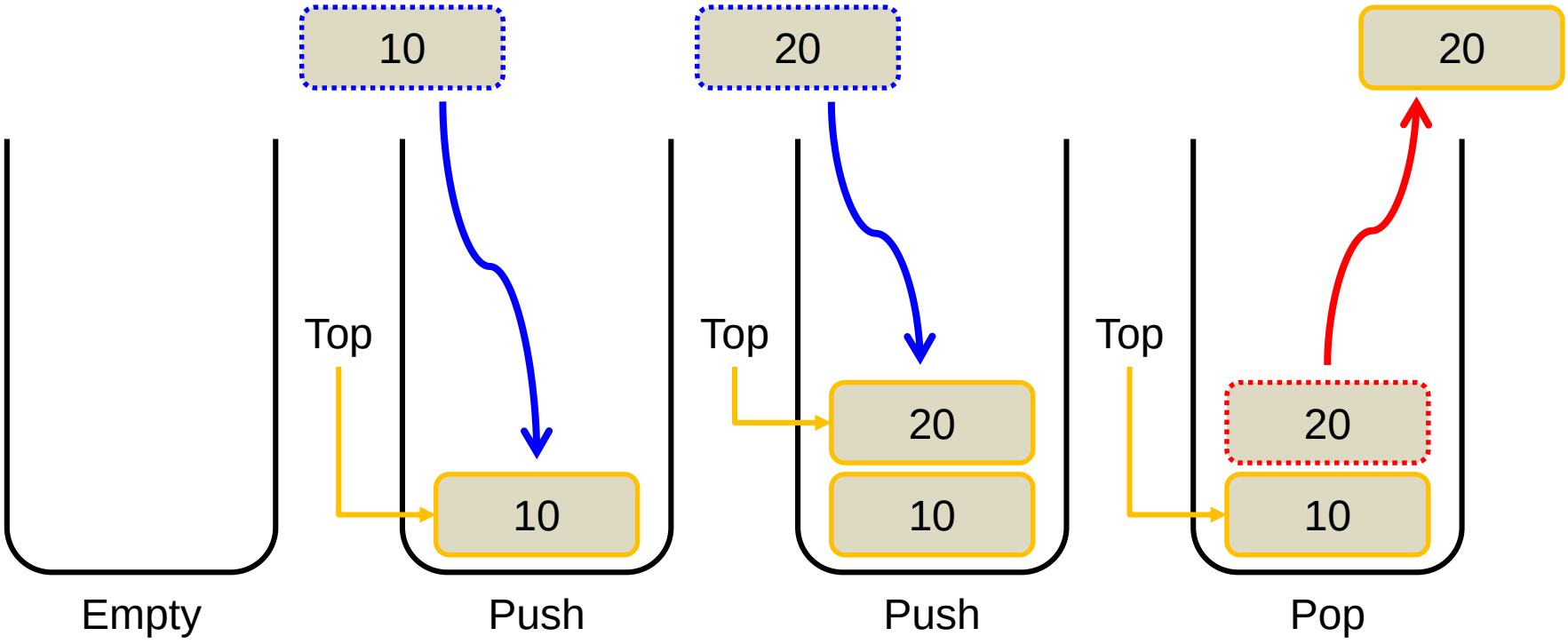
- Verilog 코드로 on-chip RAM 합성
  - 동기 읽기 dual-port RAM

```
module dp_ram_sync_read
#(
    parameter N = 8,
              W = 2
) (
    input           clk,
    input           we,
    input  [W-1:0] addr_a, addr_b,
    input  [N-1:0] din_a,
    output [N-1:0] dout_a, dout_b
);
```

```
// signal declaration
reg [N-1:0] ram[2**W-1:0];
reg [W-1:0] addr_a_reg, addr_b_reg;

// write operation
always @(posedge clk) begin
   if (we)
      ram[addr_a] <= din_a;
   addr_a_reg <= addr_a;
   addr_b_reg <= addr_b;
end

// read operation
assign dout_a = ram[addr_a_reg];
assign dout_b = ram[addr_b_reg];

endmodule
```

RAM(Random Access Memory)

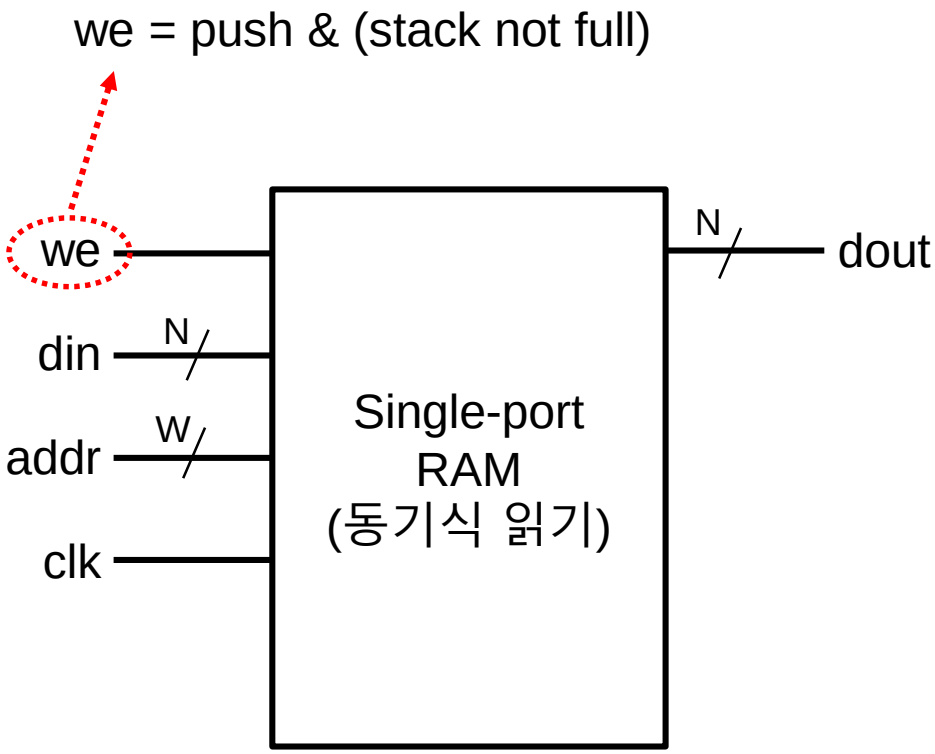
- RAM 기반 stack 구현



RAM(Random Access Memory)

- RAM 기반 stack 구현
  - Stack 용량 : S
  - Top = –1 → stack empty
  - Top = S – 1 → stack full

Push	Pop	RAM 동작
0	0	No op.
0	1	읽기
1	0	쓰기
1	1	No op.



RAM(Random Access Memory)

- RAM 기반 stack 구현

```
module stack
#(
   parameter N = 8,  // number of bits
             S = 10, // stack size
             W = $clog2(S+1)
) (
   input           clk, rstb,
   input           push, pop,
   input  [N-1:0] w_data,
   output          empty, full,
   output [N-1:0] r_data
);

// signal declaration
reg  [W-1:0] top_reg, top_next,
             top_dec, top_inc;
reg          full_reg, empty_reg,
             full_next, empty_next;
wire         ram_we;
wire [W-1:0] ram_addr;
wire [N-1:0] ram_din, ram_dout;
```

```
// RAM
sp_ram_sync_read
#(
   .N   (N       ),
   .W   (W       )
) mem (
   .clk (clk     ),
   .we  (ram_we  ),
   .addr(ram_addr),
   .din (ram_din ),
   .dout(ram_dout)
);

assign ram_we = push & ~full_reg;
assign ram_addr = push ? top_inc
                       : top_reg;
assign ram_din = w_data;
assign r_data = (push & pop) ? w_data
                             : ram_dout;
```

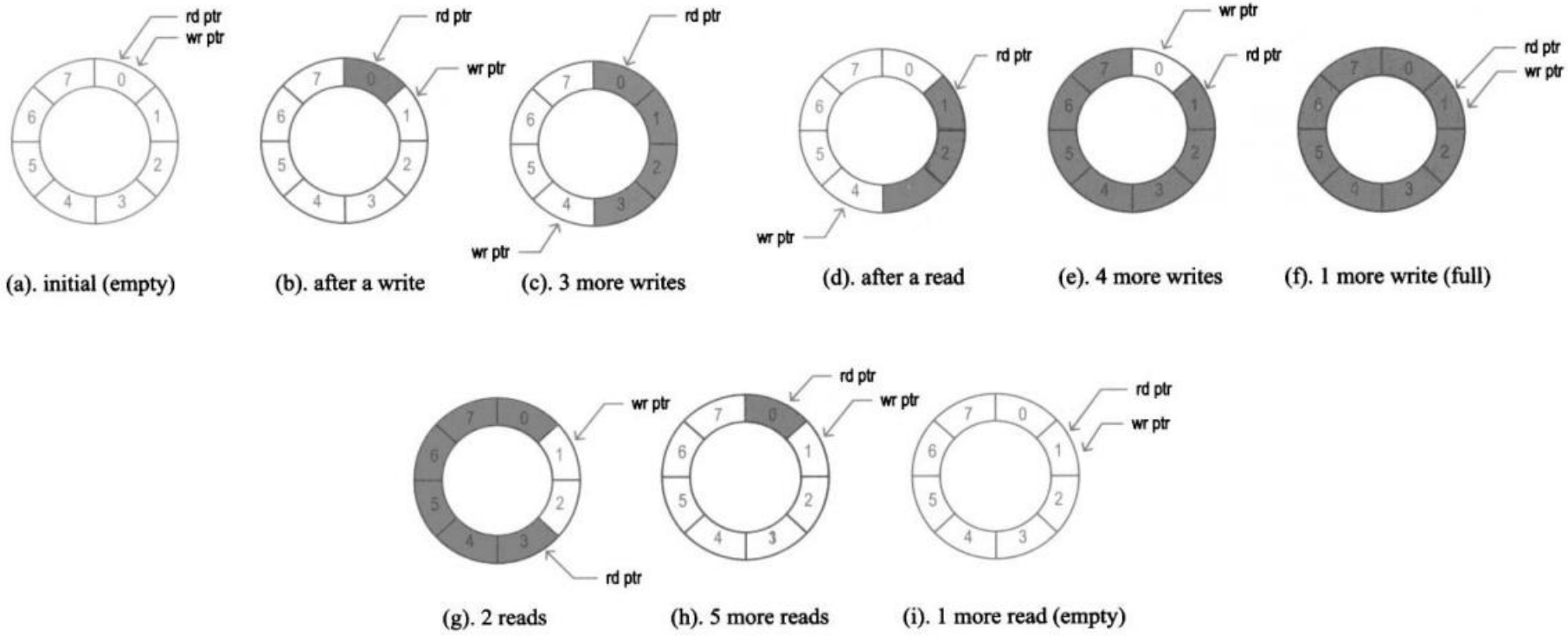
RAM(Random Access Memory)

- RAM 기반 stack 구현

```
// registers
always @(posedge clk or negedge rstb) begin
   if (~rstb) begin
      top_reg <= 2**W-1;
      full_reg <= 0;
      empty_reg <= 1;
   end
   else begin
      top_reg <= top_next;
      full_reg <= full_next;
      empty_reg <= empty_next;
   end
end

// next-state logic
always @* begin
   top_dec = top_reg-1;
   top_inc = top_reg+1;
   // default values
   top_next = top_reg;
   full_next = full_reg;
   empty_next = empty_reg;
   case ({push, pop})
      // 2'b00: no operation
      2'b01: // pop (read)
         if (~empty_reg) begin
            top_next = top_dec;
            full_next = 0;
            if (top_dec==2**W-1) empty_next = 1;
         end
      2'b10: // push (write)
         if (~full_reg) begin
            top_next = top_inc;
            empty_next = 0;
            if (top_inc==S-1) full_next = 1;
         end
      // 2'b11: // push and pop (do nothing)
   endcase
end

assign full = full_reg;
assign empty = empty_reg;

endmodule
```

RAM(Random Access Memory)

- RAM 기반 FIFO 구현



RAM(Random Access Memory)

- RAM 기반 FIFO 구현

```
module fifo
#(
   parameter N = 8, // number of bits
             S = 10, // FIFO size
             W = $clog2(S+1)
) (
   input           clk, rstb,
   input           rd, wr,
   input  [N-1:0] w_data,
   output          empty, full,
   output [N-1:0] r_data );

// signal declaration
reg  [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
reg  [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
reg          full_reg, empty_reg,
             full_next, empty_next;
wire         ram_we;
wire [W-1:0] ram_addr;
wire [N-1:0] ram_din, ram_dout;
```

```
// RAM
sp_ram_sync_read
#(
   .N   (N       ),
   .W   (W       )
) mem (
   .clk (clk     ),
   .we  (ram_we  ),
   .addr(ram_addr),
   .din (ram_din ),
   .dout(ram_dout)
);

assign ram_we = wr & ~full_reg;
assign ram_addr = ram_we ? w_ptr_reg
                         : r_ptr_reg;
assign ram_din = w_data;
assign r_data = ram_dout;
```

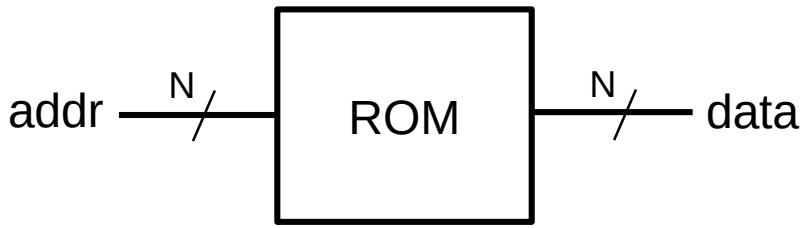
RAM(Random Access Memory)

- RAM 기반 FIFO 구현

```
// registers
always @(posedge clk or negedge rstb) begin
   if (~rstb) begin
      w_ptr_reg <= 0;
      r_ptr_reg <= 0;
      full_reg <= 0;
      empty_reg <= 1;
   end
   else begin
      w_ptr_reg <= w_ptr_next;
      r_ptr_reg <= r_ptr_next;
      full_reg <= full_next;
      empty_reg <= empty_next;
   end
end

// next-state logic
always @* begin
   w_ptr_succ = w_ptr_reg+1;
   r_ptr_succ = r_ptr_reg+1;
   // default values
   w_ptr_next = w_ptr_reg;
   r_ptr_next = r_ptr_reg;
   full_next = full_reg;
   empty_next = empty_reg;
   case ({wr, rd})
      // 2'b00: no operation
      2'b01: // read
         if (~empty_reg) begin
            r_ptr_next = r_ptr_succ;
            full_next = 0;
            if (r_ptr_next==w_ptr_reg) empty_next = 1;
         end
      2'b10: // write
         if (~full_reg) begin
            w_ptr_next = w_ptr_succ;
            empty_next = 0;
            if (w_ptr_next==r_ptr_reg) full_next = 1;
         end
      2'b11: // write and read
         begin
            w_ptr_next = w_ptr_succ;
            r_ptr_next = r_ptr_succ;
         end
   endcase
end

assign full = full_reg;
assign empty = empty_reg;

endmodule
```

ROM(Read Only Memory)

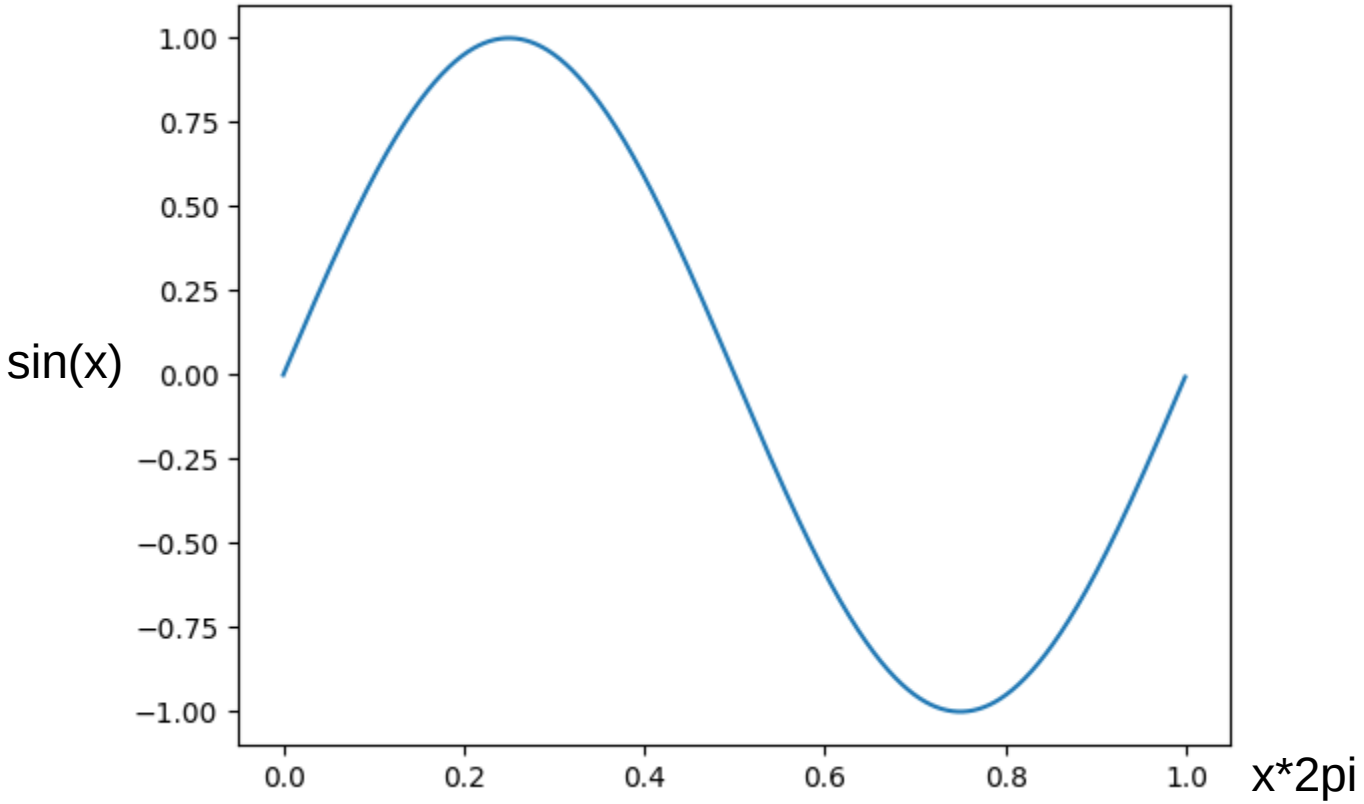
- ROM
  - 전기공급이 끊긴 상태에서도 장기간 기억하는 비휘발성(**non-volatile**) 메모리
  - FPGA 설계 분야에서 복잡한 연산/함수 구현을 위한 LUT용으로 쓰임
  - Verilog의 case 문장으로 합성




```
module rom (
   input      [2:0] addr,
   output reg [7:0] data
);
always @* begin
   case (addr)
      3'd0: data = 8'd10;
      3'd1: data = 8'd10;
      3'd2: data = 8'd20;
      3'd3: data = 8'd30;
      3'd4: data = 8'd40;
      3'd5: data = 8'd50;
      3'd6: data = 8'd60;
      3'd7: data = 8'd70;
   endcase
end
endmodule
```

ROM(Read Only Memory)

- ROM 기반 sin(x) 함수 구현



ROM(Read Only Memory)

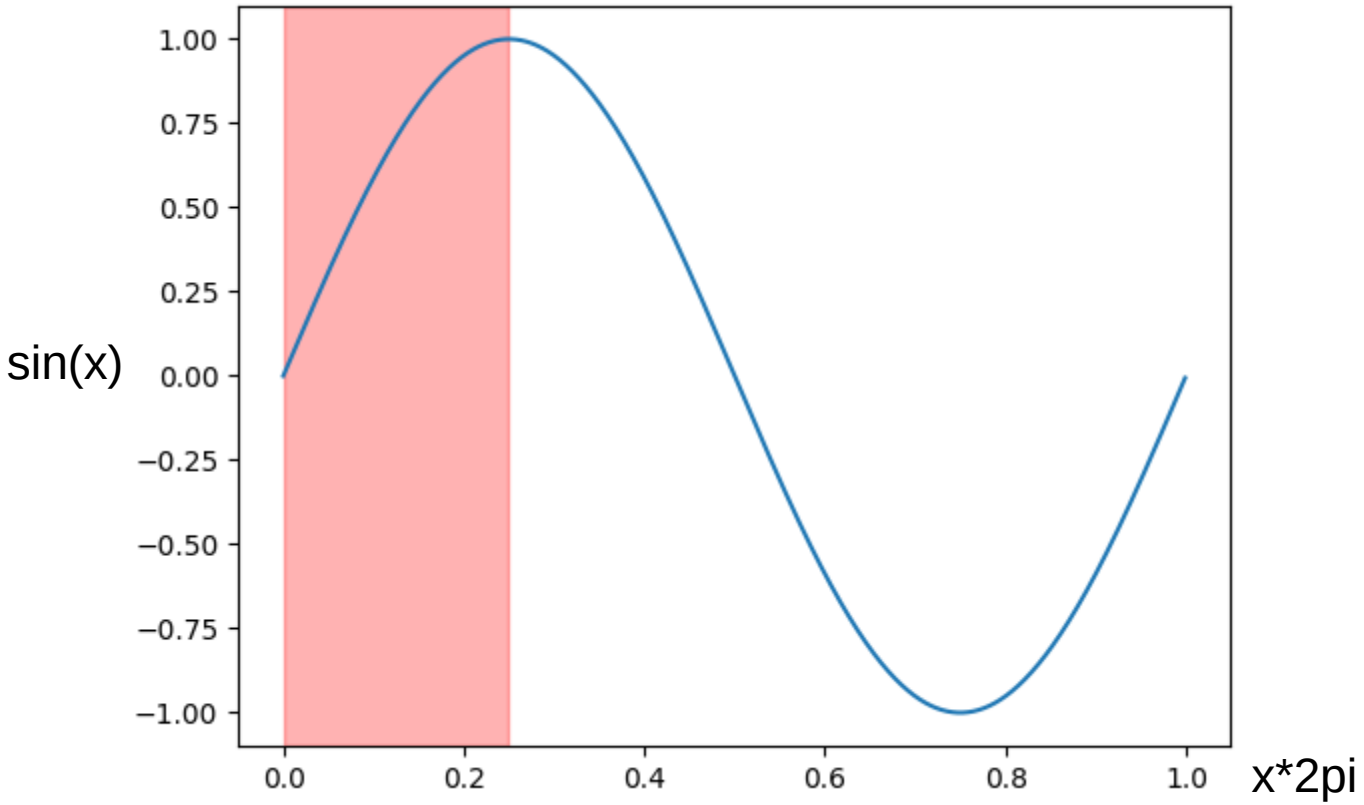
- ROM 기반 sin(x) 함수 구현
  - 입력 : 10비트 (0 → 1023)
  - 출력 : 8비트 (0 → 255)
  - case 문장이 너무 길어서 코드를 자동 생성해야 함

```
import numpy as np

x = np.arange(0, 1, 1/1024)
y = np.sin(2*np.pi*x)
y = np.round((y+1)*128)
y[y==256] = 255
```

```
f = open('rom1024.v', 'w')
f.write('module rom (\n')
f.write('input [9:0] addr,\n')
f.write('output [7:0] data\n')
f.write(');\n\n')
f.write('always @* begin\n')
f.write('case(addr)\n')
for i in range(0, 1024, 1):
    f.write('10\'h{:03x}: data = 8\'h{:02x}\n'.format(i, int(y[i])))
f.write('endcase\n')
f.write('end\n')
f.write('endmodule\n')
f.close()
```

ROM(Read Only Memory)

- ROM 기반 sin(x) 함수 구현



ROM(Read Only Memory)

- ROM 기반 sin(x) 함수 구현
  - 입력 : 10비트 (0 → 1023)
  - 출력 : 8비트 (0 → 255)
  - case 문장이 너무 길어서 코드를 자동 생성해야 함
  - **ROM 크기 4배 줄임**

```
import numpy as np

x = np.arange(0, 1, 1/1024)
y = np.sin(2*np.pi*x)
y = np.round((y+1)*128)
y[y==256] = 255

y256 = y[0:256]
```

```
f = open('rom256.v', 'w')
f.write('module rom (\n')
f.write('input [9:0] addr,\n')
f.write('output [7:0] data\n')
f.write(');\n\n')
f.write('always @* begin\n')
f.write('case(addr)\n')
for i in range(0, 256, 1):
    f.write('10\'h{:03x}: data = 8\'h{:02x}\n'.format(i, int(y256[i])))
f.write('endcase\n') f.write('end\n') f.write('endmodule\n')
f.close()
```