

## Hardware System Design Mid-term Exam

2017. 4. 25

ID :

이름 :

**Q1. (Verilog Basics, 10pt)** A=4'b0011, B=3'b011, C=3'b101일 때 다음 Verilog expression의 결과를 구하시오.

A + (B | C); \_\_\_\_\_

~ | A; \_\_\_\_\_

(A[2:0] == B) ? B : C; \_\_\_\_\_

{A, {2{C}}}; \_\_\_\_\_

A > B; \_\_\_\_\_

### ANSWER

A + (B | C); → 1010

~ | A; → 0

(A[2:0] == B) ? B : C; → 011

{A, {2{C}}}; → 0011\_101\_101

A > B; → 0

**Q2. (Synthesizable code, 10pt)** Verilog를 이용한 하드웨어 구현에서 합성 중에 발생하는 의도치 않은 latch inference는 시스템에 예측할 수 없는 버그를 만들어 낼 수 있다. 아래 Verilog code들은 각각 오른쪽에 있는 truth table을 구현한 combinational logic 혹은 flip-flop 의 code중 일부이다. (1) 그 중 latch inference가 발생한 code를 고르고, (2) 본래 의도 (combinational logic 혹은 flip-flop)대로 합성될 수 있도록 code를 수정하시오.

(a)

```
always @(enable or data)
  if (enable) y = data;
```

input	output
enable	y
0	x
1	data

(b)

```
always @(posedge clk)
  if (enable) y = data;
```

input		output
clk	enable	y
posedge	0	x
	1	data

(c)

```
always @(select or data)
  case (select)
    2'b00: y = data[select];
    2'b01: y = data[select];
    2'b10: y = data[select];
  endcase
```

input	output
select	y
2'b00	data[0]
2'b01	data[1]
2'b10	data[2]
2'b11	x

## ANSWER

(a)

```
always @(enable or data)
  if (enable) y = data;
  else y = 0;
```

or

```
always @(enable or data)
  y = 0;
  if (enable) y = data;
```

(c)

```
always @(select or data)
  case (select)
    2'b00: y = data[select];
    2'b01: y = data[select];
    2'b10: y = data[select];
    default: y = 0;
  endcase
```

**Q3. (BRAM modeling, 15pt)** 다음은 BRAM을 modeling한 Verilog code이다. Code는 simulation을 위한 부분과 BRAM의 동작을 위한 부분 두 부분으로 나뉘어 있다. 각각의 부분의 동작은 다음과 같다.

**<Simulation>**

‘INIT\_FILE’이 null string이 아니면 ‘INIT\_FILE’로부터 값을 읽어 register ‘mem’에 저장한다. ‘done’이 high가 될 때까지 기다렸다가 외부로부터 ‘done’ 신호가 high가 되면, register ‘mem’에 저장된 데이터를 ‘OUT\_FILE’에 text형태로 출력한다.

**<BRAM 동작>**

BRAM은 read에 2 cycle이 소모되고, write에 1 cycle 이 소모된다. 다음은 input/output port와 동작에 대한 자세한 설명이다.

- **BRAM\_ADDR:** 외부에서 들어오는 주소 값.
- **BRAM\_CLK:** clock.
- **BRAM\_WRDATA:** BRAM 의 data input port.
- **BRAM\_RDDATA:** BRAM 의 data output port.
- **BRAM\_EN:** ‘BRAM\_EN’ 이 1 이면 BRAM 이 read 혹은 write 동작을 수행한다. 만약 ‘BRAM\_WE’ 이 0 이고, ‘BRAM\_EN’ 이 1 이면 ‘mem[addr]’ 을 ‘BRAM\_RDDATA’ 로 출력한다.
- **BRAM\_WE:** ‘BRAM\_WE[i]’ 이 1 이면 ‘BRAM\_WRDATA[8\*i-1:8\*(i-1)]’을 ‘mem[addr][8\*i-1:8\*(i-1)]’ 에 저장한다.
- **BRAM\_RST:** ‘BRAM\_RST’ 가 1 이면 ‘BRAM\_RDDATA’ 는 0 을 출력한다.

위와 같이 동작할 수 있도록 빈칸을 채워 code를 완성 하시오.

```
module my_bram # (  
    parameter integer BRAM_ADDR_WIDTH = 15, // 4x8192  
    parameter INIT_FILE = "input.txt",  
    parameter OUT_FILE = "output.txt"  
)  
(  
    input wire [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR,  
    input wire BRAM_CLK,  
    input wire [31:0] BRAM_WRDATA,  
    output reg [31:0] BRAM_RDDATA,  
    input wire BRAM_EN,  
    input wire BRAM_RST,  
    input wire [3:0] BRAM_WE,  
    input wire done  
);  
    reg [31:0] mem[0:8191];  
    wire [BRAM_ADDR_WIDTH-3:0] addr = BRAM_ADDR[BRAM_ADDR_WIDTH-1:2];  
    reg [31:0] dout;  
  
//Simulation  
    initial begin
```

```

    if (INIT_FILE != "")
        _____;
        _____;
    end

//BRAM 동작
    always @(posedge BRAM_CLK)
        if (BRAM_WE[0] && BRAM_EN)
            _____;
    always @(posedge BRAM_CLK)
        if (BRAM_WE[1] && BRAM_EN)
            _____;
    always @(posedge BRAM_CLK)
        if (BRAM_WE[2] && BRAM_EN)
            _____;
    always @(posedge BRAM_CLK)
        if (BRAM_WE[3] && BRAM_EN)
            _____;

    //dout
    always @(posedge BRAM_CLK)
        _____;

    //BRAM RDDATA
    always @(posedge BRAM_CLK)
        if (BRAM_RST)
            _____;
        else if (BRAM_EN)
            _____;

endmodule

```

## ANSWER

```
module my_bram # (
    parameter integer BRAM_ADDR_WIDTH = 15, // 4x8192
    parameter INIT_FILE = "input.txt",
    parameter OUT_FILE = "output.txt"
)(
    input wire [BRAM_ADDR_WIDTH-1:0] BRAM_ADDR,
    input wire BRAM_CLK,
    input wire [31:0] BRAM_WRDATA,
    output reg [31:0] BRAM_RDDATA,
    input wire BRAM_EN,
    input wire BRAM_RST,
    input wire [3:0] BRAM_WE,
    input wire done
);

reg [31:0] mem[0:8191];
wire [BRAM_ADDR_WIDTH-3:0] addr = BRAM_ADDR[BRAM_ADDR_WIDTH-1:2];
reg [31:0] dout;

initial begin
    if (INIT_FILE != "") begin
        $readmemh(INIT_FILE, mem);
    end
    wait(done)
    $writememh(OUT_FILE, mem);
end

always @(posedge BRAM_CLK)
    if (BRAM_WE[0] && BRAM_EN)
        mem[addr][7:0] <= BRAM_WRDATA[7:0];
always @(posedge BRAM_CLK)
    if (BRAM_WE[1] && BRAM_EN)
        mem[addr][15:8] <= BRAM_WRDATA[15:8];
always @(posedge BRAM_CLK)
    if (BRAM_WE[2] && BRAM_EN)
        mem[addr][23:16] <= BRAM_WRDATA[23:16];
always @(posedge BRAM_CLK)
    if (BRAM_WE[3] && BRAM_EN)
        mem[addr][31:24] <= BRAM_WRDATA[31:24];

always @(posedge BRAM_CLK)
    dout <= mem[addr];
always @(posedge BRAM_CLK)
    if (BRAM_RST)
        BRAM_RDDATA <= 'd0;
    else if (BRAM_EN)
        BRAM_RDDATA <= dout;

endmodule
```

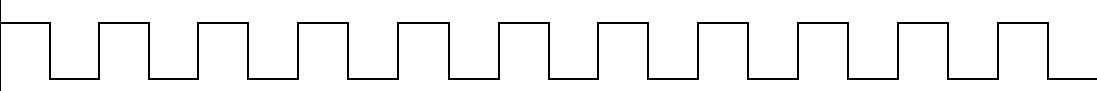
(M\*V 곱셈기 설계) 본 문제의 목표는 여러 개의 PE를 이용해 하나의 M\*V 곱셈기를 설계하는 것이다. 편의를 위해 모든 vector의 크기는 N이며 모든 matrix의 크기는 N\*N으로 가정한다.

**Q4. (15pt)** M\*V 곱셈기는 외부 BRAM으로부터 데이터를 가져온다. 아래는 Xilinx에서 제공하는 BRAM simulation model에 입력 벡터를 넣은 실험 결과이다. 입력 벡터의 의도는 데이터가 제대로 쓰이고 읽히는지 확인하는 것인데, 먼저 데이터 4개를 쓴 후 쓴 데이터를 다시 읽어서 확인하는 식이다.

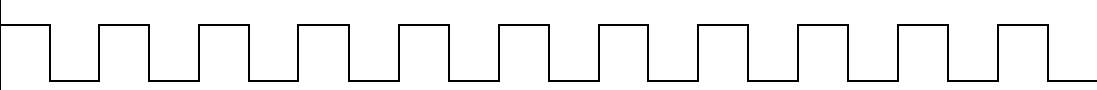
입력 벡터)

Time	1	2	3	4	5	6	7	8
Type	WR	WR	WR	WR	RD	RD	RD	RD
ADDR	0x30	0x34	0x38	0x3C	0x30	0x34	0x38	0x3C
DIN	0x1234	0x5678	0x9ABC	0xDEF0	0x0808	0x0808	0x0808	0x0808

출력 레지스터 OFF) ADDR를 주면 #delay 이후 DOUT이 출력된다.

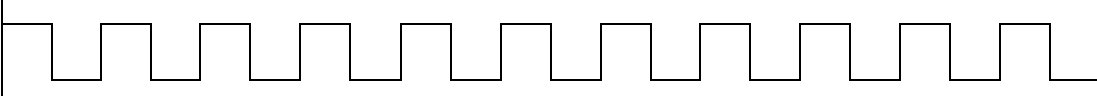
CLK											
ADDR	0000	0030	0034	0038	003C	0030	0034	0038	003C	003C	003C
DIN	0000	<b>1234</b>	<b>5678</b>	<b>9ABC</b>	<b>DEF0</b>	0808	0808	0808	0808	0808	0808
DOUT	0000	0000	5678	9ABC	DEF0	<b>0808</b>	<b>5678</b>	<b>9ABC</b>	<b>DEF0</b>	DEF0	DEF0
WE	0	F	F	F	F	0	0	0	0	0	0

출력 레지스터 ON) ADDR를 주면 다음 @posedge CLK에서 DOUT이 출력된다.

CLK											
ADDR	0000	0030	0034	0038	003C	0030	0034	0038	003C	003C	003C
DIN	0000	<b>1234</b>	<b>5678</b>	<b>9ABC</b>	<b>DEF0</b>	0808	0808	0808	0808	0808	0808
DOUT	0000	0000	0000	5678	9ABC	DEF0	<b>0808</b>	<b>5678</b>	<b>9ABC</b>	<b>DEF0</b>	DEF0
WE	0	F	F	F	F	0	0	0	0	0	0

실험자는 출력 레지스터가 ON 또는 OFF일 때의 DOUT 출력 특성을 확인했다. 그런데, 출력 레지스터 설정과 무관하게 데이터 출력이 이상했다. 예를 들면 1234 대신 0808이 읽혔다. 이 문제로 Xilinx에서 BRAM을 잘 못 만들었는지 고민하는 실험자를 본 한 친구가 BRAM의 clock을 반전시켜보라고 권했다.

출력 레지스터 ON + BRAM 클락 반전) 컨트롤러의 @negedge CLK에서 DOUT이 출력된다.

CLK												
ADDR	0000	0030	0034	0038	003C	0030	0034	0038	003C	003C	003C	
DIN	0000	1234	5678	9ABC	DEF0	0808	0808	0808	0808	0808	0808	
DOUT		0000	0000	1234	5678	9ABC	DEF0	1234	5678	9ABC	DEF0	
WE	0	F	F	F	F	0	0	0	0	0	0	

BRAM 클락 반전 후 실험자는 IP에 문제가 없다는 사실 뿐만 아니라 이 경우 BRAM이 2-cycle read delay를 가진다는 것을 알 수 있었다. 그렇다면 초기 실험은 BRAM의 어떤 특성 때문에 생긴 오해인가?

## ANSWER

WE 신호가 Clock의 head가 아닌 tail에 입력되는 특성 때문이다. 1234가 무시되고 0808이 30번지에 쓰였다.



**Q5. (10pt)** Zed board의 총 resource 용량과 Processing Element 개당 소모량이 아래와 같을 때, 이론상 최대로 사용 가능한 Processing Element 개수를 구하여라.

Resource	Zed-board	PE
LUT	53200	681
LUTRAM	17400	41
FF	106400	1118
BRAM	140	0.5
DSP	220	2
MMCM	4	0

**ANSWER**

**78개** ( $53200/681 = 78.12$ )

**Q6. (15pt)** 우리에게 주어진 clock generator는 10MHz의 배수만 설정 가능하다고 하자. 아래는 Zed board에서 N=64 M\*V 연산기 합성 결과 보고서의 일부이다.

- (1) Critical path delay 가 아래와 같을 때, 이론상 최대로 동작 가능한 clock frequency 와 그 때의 이론상 최대 Operations Per Second (OPS) 를 구하여라.
- (2) Critical path 가 BRAM 과 controller 사이의 연결을 포함한다고 할 때, 만약 BRAM clock 반전을 사용할 경우 위의 계산 값은 어떻게 되는가?

	Delay type	Delay (ns)
Source Clock Path	PS7	0
	Net (fo=1, routed)	1.193
	BUFG (Prop bufg I O)	0.101
	Net (fo=80352, routed)	1.648
	<b>Total Clock Delay</b>	<b>2.942 (ns)</b>
Data Path	FDRE (Prop fdre_C O)	0.419
	Net (fo=299, routed)	2.004
	MUXF7 (Prop muxf7_S O)	0.451
	Net (fo=1, routed)	1.530
	LUT6 (Prop lut6 IO O)	0.299
	Net (fo=1, routed)	1.074
	LUT6 (Prop lut6 I1 O)	0.124
	Net (fo=2, routed)	0.899
	<b>Total Data Delay</b>	<b>6.801 (ns)</b>
Source Clock + Data Path	Arrival Time	9.743 (ns)

#### ANSWER

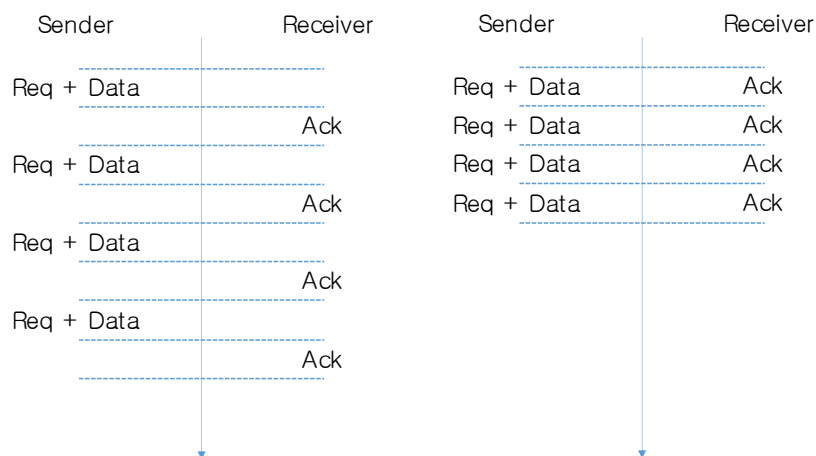
(1) 100MHz, 6.4GOPS

(2) 50MHz, 3.2GOPS

**Q7. (10pt)** 수업시간에 버스 시스템 예제로 ARM AMBA3 AXI 버스 동작과 간단한 구현 예를 공부하였다. AXI 버스에서는 각 채널 별로 valid, ready 시그널을 갖는다. 이 2개 시그널의 용도에 대해 설명하고, 매 cycle 마다 채널의 master 쪽에서 slave 쪽으로 정보(예, address, data or response)를 전달할 수 있으려면 이 2개 시그널들을 어떻게 사용해야 하는지 설명하자.

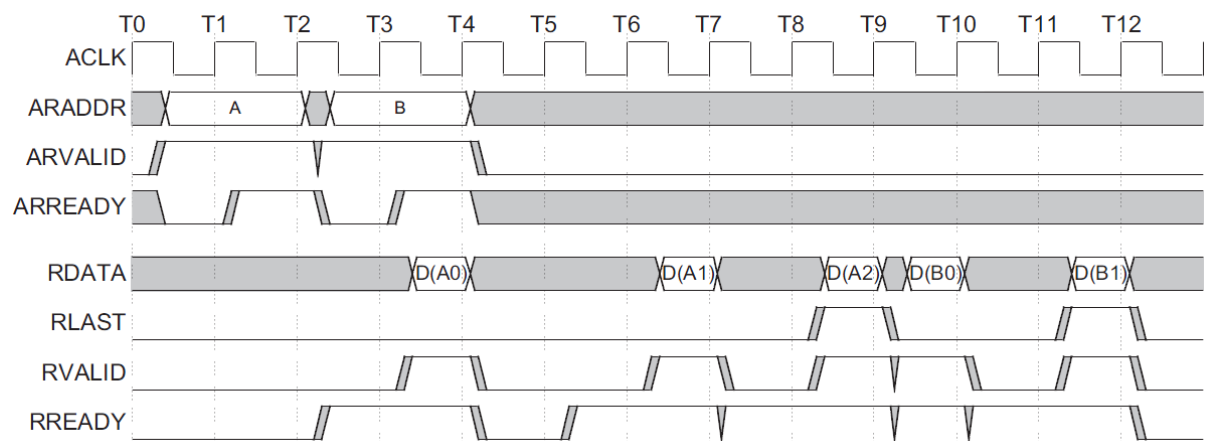
답) Master 쪽에서 정보를 보내려 할 때 valid를 '1'로 해야 한다. Clock rising edge에서 master의 출력(slave의 입력)인 valid가 '1'이고, master의 입력(slave의 출력)인 ready가 '1'이면, master에서 slave로 정보가 넘어간다(예, 읽기 요청이 넘어간다).

매 cycle 마다 master 쪽에서 slave 쪽으로 정보를 전달하려면 master는 매 cycle (clock rising edge)마다 valid 시그널을 '1'로 하여야 하고 (보내는 정보를 매 cycle 새로운 내용으로 하면서), slave에서도 매 cycle 마다 ready 시그널을 '1'로 하면 된다. 그러면 아래 그림의 오른쪽처럼 매 cycle 정보를 전달할 수 있다.



**Q8. (15pt)** AXI 버스의 하나의 port (예, CPU가 버스와 연결된 port)에는 5개의 채널이 있다. 여러 채널을 이용하면, 앞의 요청이 처리되기 전에 여러 개의 요청(읽기 또는 쓰기 요청)을 보낼 수 있어서, 하드웨어 블록들 간의 통신(예, CPU에서 메모리로부터 데이터를 받는 통신)의 성능을 향상시킬 수 있다고 한다. Read address channel (AR channel)과 read data channel (R channel)의 예를 들어 어떻게 성능 향상이 가능한지 설명해 보자.

답) 아래 그림과 같은 예에서 시간 T2에 master가 보낸 주소 A에 대한 읽기 요청이 완전히 처리되기 전에, 시간 T2에 주소 B에 대한 읽기 요청을 보내어서, slave 쪽에서 2개의 요청을 동시에 처리할 수 있는 경우 요청 처리의 평균 시간을 줄일 수 있다.



아래 그림과 같은 걸 제시하고 slave 쪽(이 경우는 메모리)에서 여러 개의 읽기 요청을 동시에 처리해서 평균 처리시간을 줄일 수 있다는 것을 설명해도 OK 임.

