

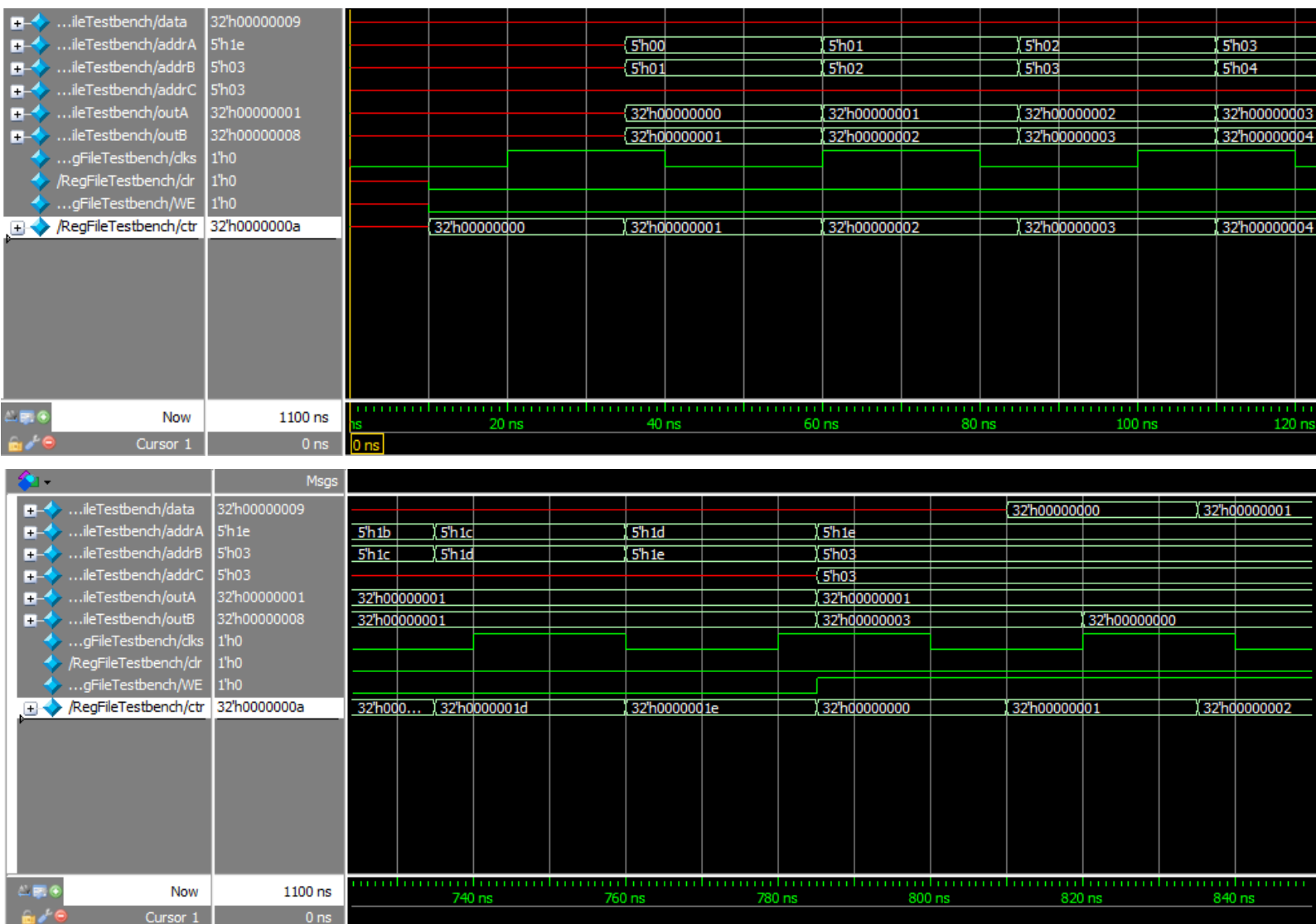
Computer Structrue

~ Lab 08 ~

2013210111 남세현

RegisterFile Waveform 분석

10ns 부터 785ns 까지는 WE 가 0 입니다. 그리고 초기에 reg.dat 에 있던 데이터가 register 에 입력이 되어지기 때문에, read port A 와 B 에 해당되는 register 의 값의 출력은 reg.dat 과 같음을 볼 수 있습니다.



0ns ~ 785ns 까지의 데이터

1	@000	00000000
2	@001	00000001
3	@002	00000002
4	@003	00000003
5	@004	00000004
6	@005	00000005
7	@006	00000006
8	@007	00000007
9	@008	00000008
10	@009	00000009
11	@00a	0000000a
12	@00b	0000000b
13	@00c	0000000c
14	@00d	0000000d
15	@00e	0000000e
16	@00f	0000000f
17	@010	00000010
18	@011	00000001
19	@012	00000001
20	@013	00000001
21	@014	00000001
22	@015	00000001
23	@016	00000001
24	@017	00000001
25	@018	00000001
26	@019	00000001
27	@01a	00000001
28	@01b	00000001
29	@01c	00000001
30	@01d	00000001
31	@01e	00000001
32	@01f	00000001
33		

Reg.dat의 데이터.

위 waveform의 output부분과 정확히 일치함을 볼 수 있음.

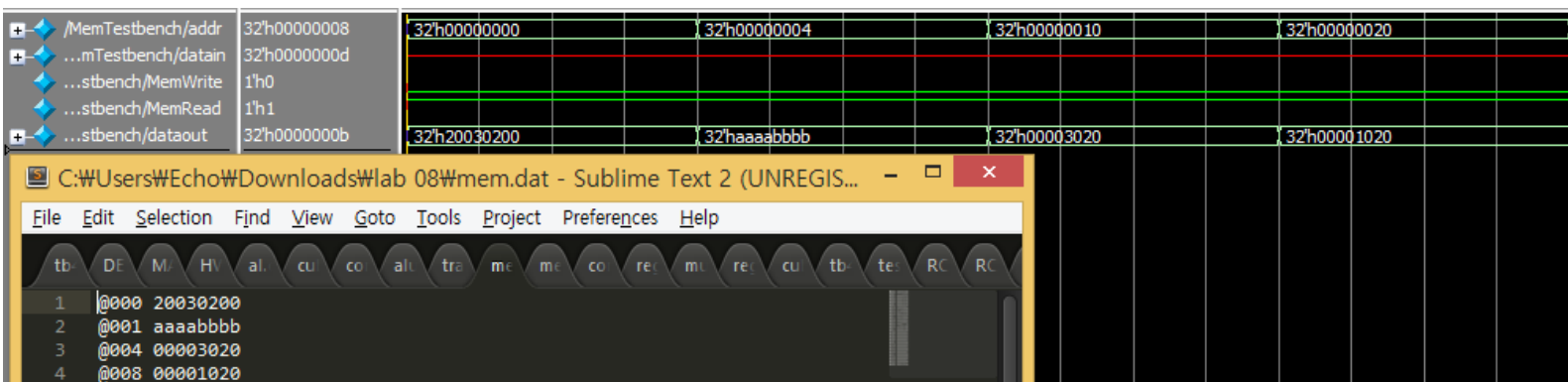
하지만 785ns부터 1035ns까지는 WE가 1이 되면서, 1개의 write포트로 값들을 레지스터에 넣어주고 있음을 볼 수 있습니다. (addrC 와 ctr 이 바로 그것)

1035ns부터 나머지는 WE가 0이 되고, 나머지 부분은 register가 새로 갱신이 안되고 reg.dat에서 가져온 데이터 그대로 출력됨을 볼 수 있습니다.

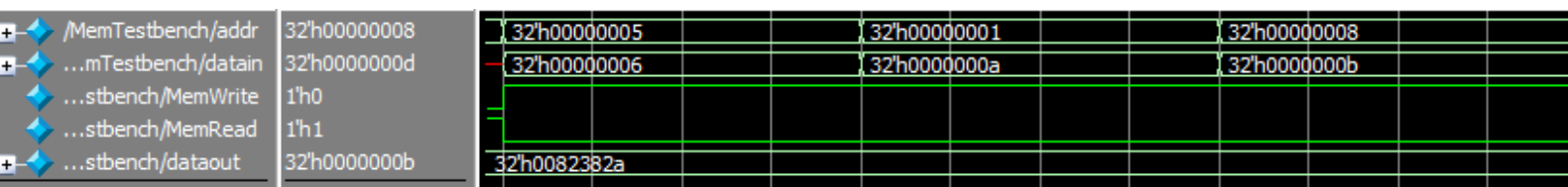
MEM Waveform 분석

Memory read가 1이면 memory에서 read를, memory write가 1이면 memory에 write를 하는 모듈입니다.

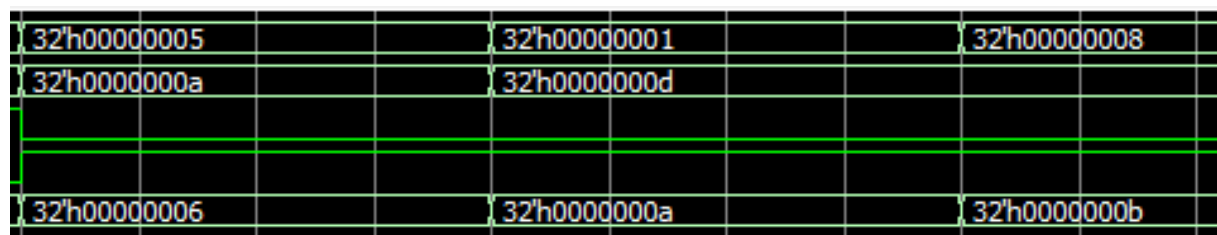
0ns부터 480ns까지는 memread가 1이므로, mem.dat에 있는 데이터와 동일하게 load됨을 볼 수 있습니다.



반면에 480ns부터 540ns까지는 memwrite가 1이 되면서 memory에 새로운 값들이 올라갑니다.



480ns ~ 540ns.



540ns ~ 600ns.

540 ~ 600ns 구간에서는 480~540ns 구간에서 write했던 메모리를 다시 읽어옵니다.

480~540ns 구간의 dain과 540~640ns 구간의 dataout을 비교하면 똑같은 것을 확인할 수 있습니다. 이로서 메모리에 잘 들어갔음을 볼 수 있습니다.

MUX2to1은 생략하겠습니다.(MUX4to1을 이 보고서에서 다루기 때문)

ALU

OpCode 3bit를 입력받아서, 아래의 명령어를 수행하는 코드를 작성해야 합니다.

- X00 : AND operation
- X01 : OR operation
- 010 : Addition operation
- 110 : Subtraction operation
- X11 : SLT operation

Casex 문을 이용하여 위에 X에 해당되는 부분을 ? 로 처리하여, 어떤 비트가 들어와도 처리할 수 있도록 하였습니다.

아래에 해당부분 코드만 첨부합니다.

[alu_ben.h] 의 module ALU 중

전략...

```
/*whenever input or ALUop changes*/  
always @(inputA or inputB or ALUop)  
begin
```

```
casez(ALUop)
```

```
    3'b110: result = inputA - inputB;  
    3'b010: result = inputA + inputB;  
    3'bz00: result = inputA & inputB;  
    3'b?01: result = inputA | inputB;  
    3'b?11:  
        if ( inputA - inputB >= 0 )  
            result = 1;  
        else  
            result = 0;  
endcase
```

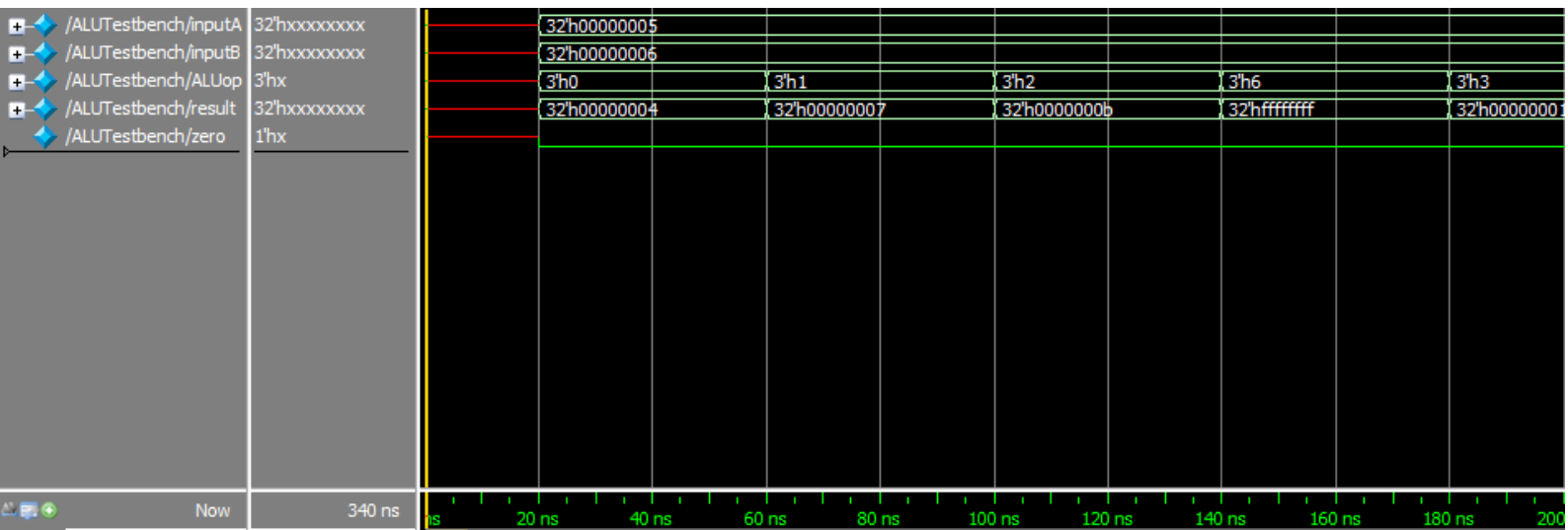
```
    if (inputA == inputB)
```

```

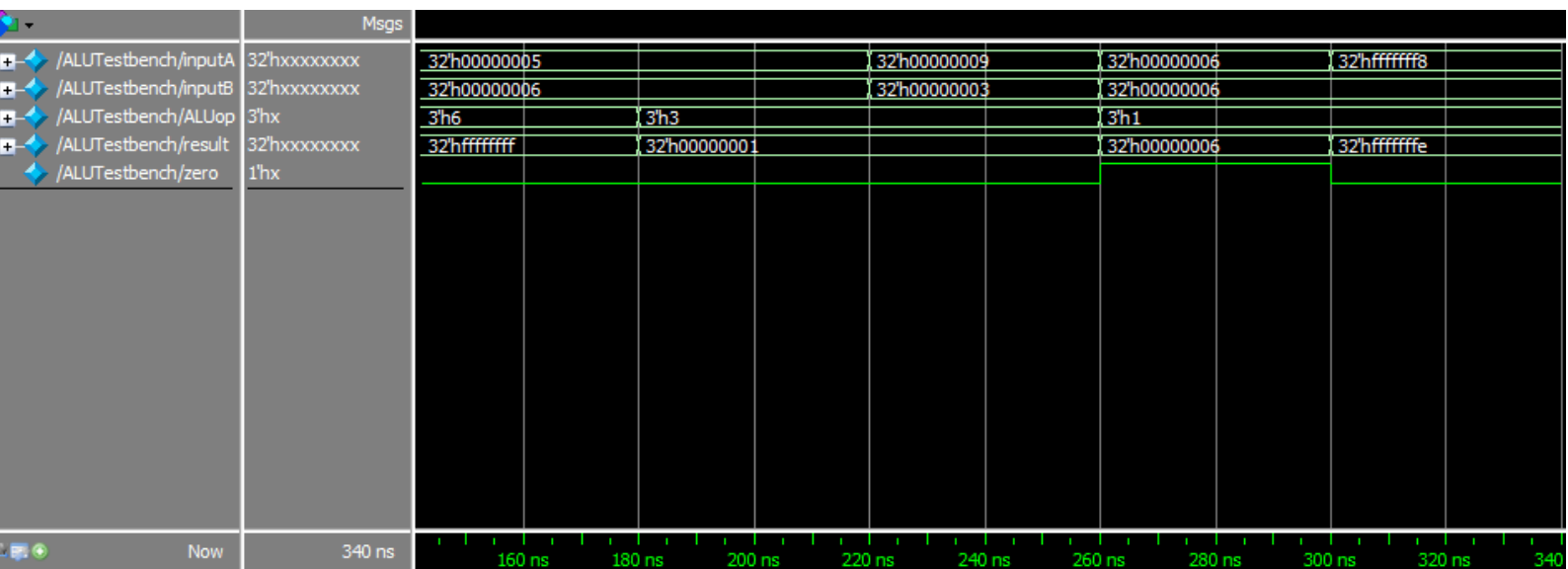
        zero = 1;
    else
        zero = 0;
    후략...

```

실행 결과



20ns : ALU op = 000 이므로 and 연산으로 4가,

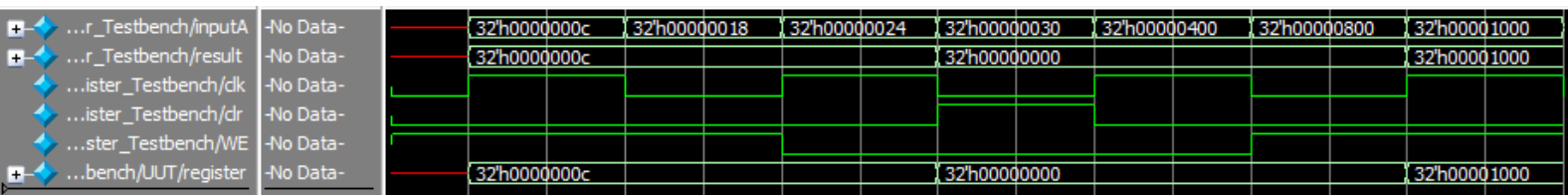


60ns : ALU op = 001 이므로 or 연산으로 7이. (5 | 6 = 7. 1001 | 1010 = 1011(7))
 그 외에도 나머지 값들이 정상적으로 나온 것을 확인할 수 있었습니다.

Single Register

clr 이나 clk 이 pos-edge 일때만 register 에 값을 넣거나 지우는 코드를 작성했습니다.

모든 상황에서만 register 에 값을 넣을 수 있는게 아니고, WE(wite enable)이 1 일때만 가능합니다.



(실행 화면)

WE 가 1 이고, clk 이 pos-edge 가 될 때만 값이 register 에 써지는 것을 확인할 수 있습니다.

또한 clr 가 pos-edge 될 때도 값이 지워짐을 8 번째 clock 에서 확인이 가능합니다.

[Components.v 중 single_register]

```
module single_register(datain, dataout, clk, clr, WE);
```

```
    input [31:0] datain;
```

```
    output [31:0] dataout;
```

```
    input clk, clr, WE;
```

```
    reg [31:0] register;
```

```
always @(posedge clk or posedge clr)
```

```
begin
```

```
    if(clr)
```

```
        register = 0;
```

```
    else
```

```
        if(WE == 1)
```

```
            register = datain;
```

```
end
```

```
    assign dataout = register;
```

```
endmodule
```

MUX 4 to 1

Select 값을 통해 4 개 중 1 개의 값으로 출력을 해주는 동작을 합니다.

[illegible]

```
module mux4to1(datain0, datain1, datain2, datain3, dataout, select);
```

```
input [31:0] datain0, datain1, datain2, datain3;
```

```
input[1:0] select;
```

```
output [31:0] dataout;
```

```
reg [31:0] dataout;
```

always@(select)

begin

case(select)

```
2'b00: dataout = datain0;
```

```
2'b01: dataout = datain1;
```

```
2'b10: dataout = datain2;
```

```
2'b11: dataout = datain3;
```

endcase

end

endmodule

Sign Extension 16 to 32

16bit를 입력받아서, \$signed() 구문을 통해 32bit로 변환하였습니다.

변환할 비트들 중 맨 앞 비트가 1이면 변환 후 나머지 비트들이 그 앞 비트로 설정됩니다.

즉, dataout = {16{datain[15] }, datain[15:0]} 과 같다고 할 수 있습니다.

/signextd_Testbenc...	16'h0023	16'h04d2				16'hfa00				16'h0023
/signextd_Testbenc...	32'h00000023	32'h000004d2				32'hffffffa00				32'h00000023

위에서 말한 대로, 두번째 때 맨 앞 1 비트가 1 이므로 FFFF... 로 시작됨을 볼 수 있습니다.

[Components.v 중 signextd]

```
module signextd(datain, dataout);
```

```
input [15:0] datain;
```

```
output [31:0] dataout;
```

```
reg [31:0] dataout;
```

```
integer i;
```

```
always @ (datain)
```

```
begin
```

```
    dataout = $signed(datain);
```

```
end
```

```
endmodule
```


Shift Left 2

Always @ (datain) 으로, datain이 바뀔때마다 체크해서, datain에 2를 곱한 값을 dataout에 넣어주고 있습니다.

Shift 연산이 곧 2의 지수승의 곱과 같이 때문입니다.

테스크벤치 코드는 전부 올리지 않고, ShiftLeft Module 코드만 올리겠습니다.

[Components.v 중 shiftleft2]

```
module shiftleft2(datain, dataout);
```

```
input [31:0] datain;
```

```
output [31:0] dataout;
```

```
reg [31:0] dataout;
```

```
always @ (datain)
```



```
begin
```

```
    dataout = datain * 4;
```

```
end
```

```
endmodule
```

실행 결과 (아래가 output, 위가 input)

		Msgs																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
 +  /shiftleft2_Testben...	32'b1001100110...																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													

Concatenate 4 to 28

..._Testbench/inputA	32'h99999999	32'h0000aada	32'h99999999	32'h08104225	32'hf0f0f0f0
..._Testbench/inputB	32'h99999999	32'h0000ffff	32'h99999999	32'h0f0f0f0f	32'h63c63c63
..._Testbench/result	32'h99999999	32'h0000aada	32'h99999999	32'h08104225	32'h60f0f0f0

PC In 의 상위 4비트와, datain의 하위 28비트를 합쳐서 PC Out으로 내보내야 합니다.

Pcout = {{pcin[31:28]},datain[27:0]} 연산으로 쉽게 할 수 있습니다.

위 결과물을 보면 상위 4비트는 항상 두번째 inputB(pcin)을 따르는 것을 볼 수 있습니다.

```
/* concatenate pcin[31:28] with datain[27:0] to form a jump address*/
```

```
module concatenate4to28(datain, pcin, pcout);
```

```
input [31:0] datain, pcin;
```

```
output [31:0] pcout;
```

```
reg [31:0] pcout;
```

```
always @ (datain or pcin)
```

```
begin
```

```
    pcout = {pcin[31:28],datain[27:0]};
```

```
end
```

```
endmodule
```