


32bit pipeline mips cpu

상세 내용

+ 코드는 git에 별도 첨부하였습니다!

어떤 명령어 구성으로 검증?

명령어 메모리 저장 내용



1.	lw	\$t0, 0(\$zero)	
2.	lw	\$t1 , 4(\$zero)	
3.	add	\$t2 , \$t0, \$t1	Mem hazard
4.	sw	\$t2 , 8(\$zero)	Data hazard
5.	beq	\$t0, \$t1, L1	
6.	sub	\$t3, \$t0, \$t1	
7.	jump	L2	
8.	L1 : add	\$t3, \$t0, \$t1	
9.	sw	\$t3, 12(\$zero)	

5 stage 명령어 사이클 단계

IF → ID → EX → MEM → WB

Mem hazard 해결 단계

1. Hazard detection 유닛에서 **\$t1** 을 감지하여 lw와 add 명령어 사이클 사이에 1 bubble을 생성
2. Forwarding 유닛에 의해 **\$t1** 을 감지하여 add의 EX cycle에 lw의 WB cycle의 **\$t1** 을 받아와 사용
>> 정확한 데이터 갱신을 1 bubble만으로 가능, bubble이 없는

Data hazard 로 설계 가능하지만 이 경우 조합로직이 커짐
즉, 전체 시스템 주파수가 느려짐

Data hazard 해결 단계

1. Forwarding 유닛에 의해 **\$t2** 을 감지하여 sw의 EX cycle에 add의 MEM cycle의 **\$t2** 을 받아와 사용
>> 정확한 데이터 갱신을 bubble 없이 가능

검증 포인트는?

1. 분기 명령어가 정확히 ID cycle에서 EX되는지
2. 점프 명령어가 정확히 ID cycle에서 EX되는지
3. 1,2번 검증에서 각각 hazard 처리와 stall도 정확히 수행되는지

1. 분기 명령어 검증

1) 데이터 메모리 값 세트

mem[3:0] = 0x00000014

mem[7:4] = 0x00000014

: 분기가 일어나도록 하기 위해
beq 조건을 참으로 만들기

datamemfile - Windows 메모장	
파일(F)	편집(E) 서식(O) 보기(V) 도
14	
00	
00	
00	
14	
00	
00	
00	



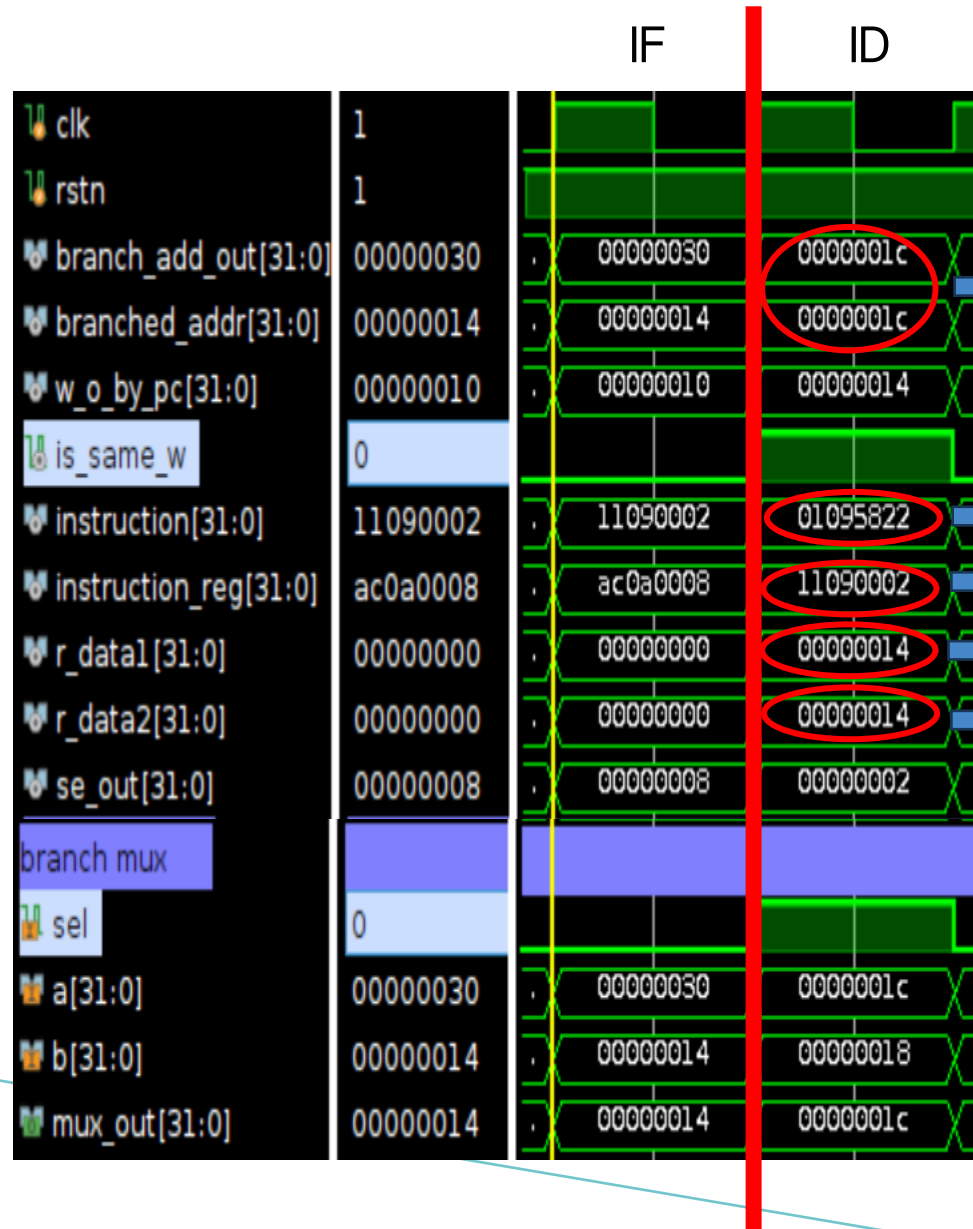
L1 : add \$t3, \$t0, \$t1

sw \$t3, 12(\$zero)에 의해

mem[15:12] = 0x00000028

이 되는지 확인하기

2) 분기 시뮬레이션 수행



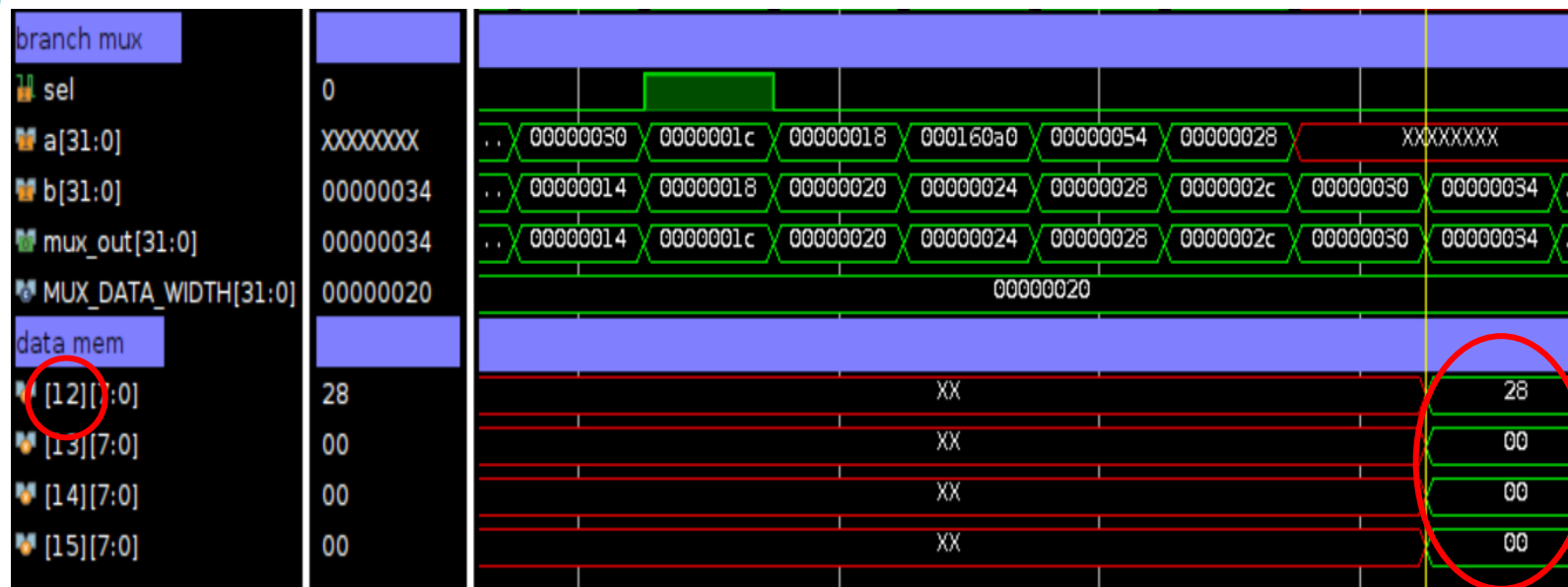
+ 부가 설명 : word 단위를 byte 단위의 addressing 으로 바꾸기 위해 2칸 좌측 시프트 수행

분기 대상 명령어 메모리의 주소
= L1 : add \$t3, \$t0, \$t1 의 주소

Flush 관련해서 drop됨(다음장에서 설명)
mips 명령어 코드 = beq \$t0, \$t1, L1

\$t0 저장 값

\$t1 저장 값



L1 : add \$t3, \$t0, \$t1
 sw \$t3, 12(\$zero)
 확인 완료

+ flush 기능 확인하기 : IF 에서는 beq가 거짓이라 가정하여, 우선 next 주소의 명령어를 fetch 후 beq가 참이면 그 명령어를 ID에서 flush(=drop) 함 IF ID



beq가 거짓이라고 가정한 후 fetch된 next 주소의 명령어

Branch 발생 이후 이전 사이클에서 fetch된 명령어가 ID에서 flush(=drop)된 부분

2. 점프 명령어 검증

1) 데이터 메모리 값 세트

mem[3:0] = 0x00000015

mem[7:4] = 0x00000014

: 점프가 일어나도록 하기 위해
beq 조건을 거짓으로 만들기

datamemfile - Windows 메모장	
파일(F)	편집(E) 서식(O) 보기(V) 도
15	
00	
00	
00	
14	
00	
00	
00	



sub \$t3, \$t0, \$t1

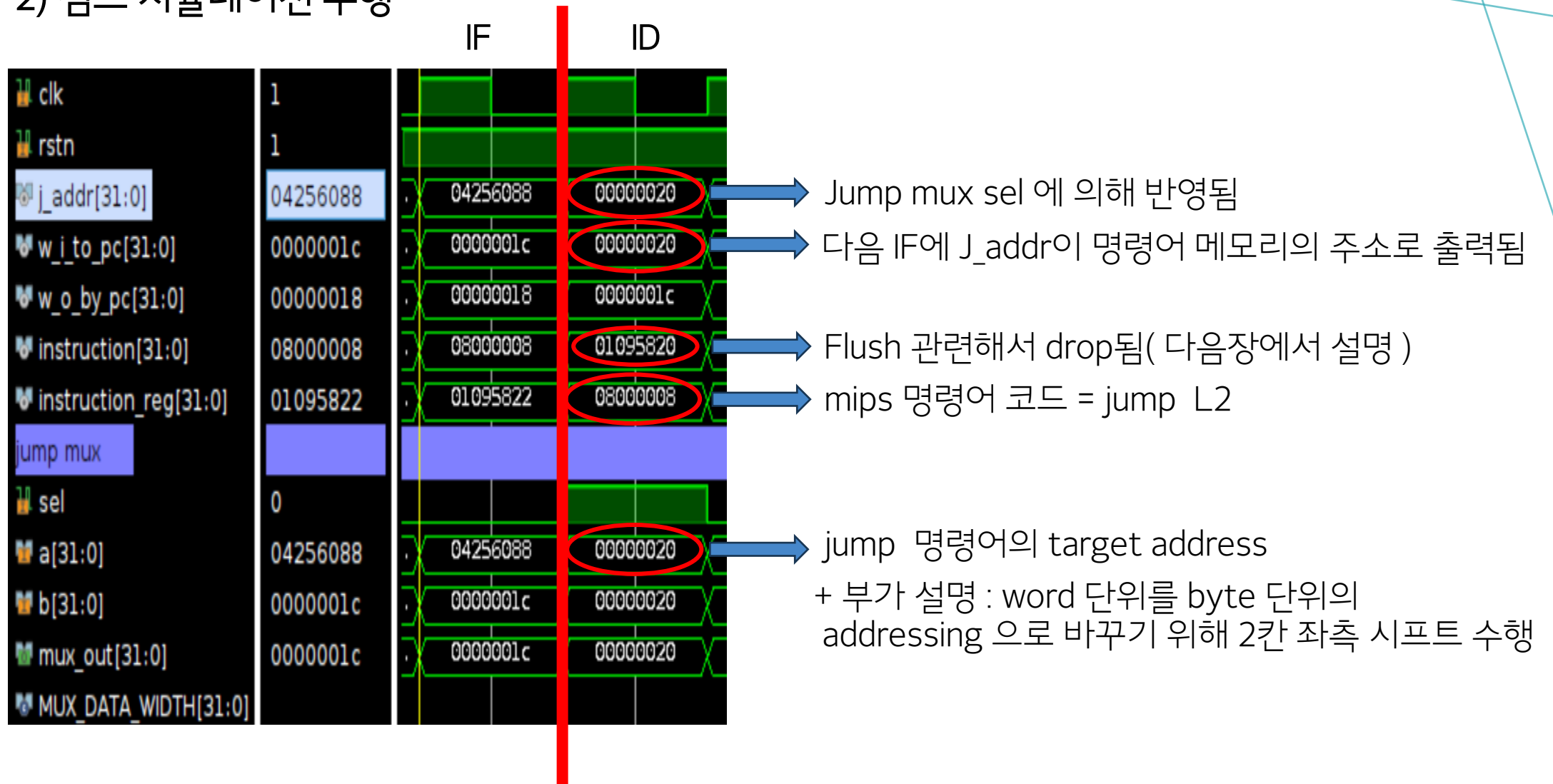
jump L2

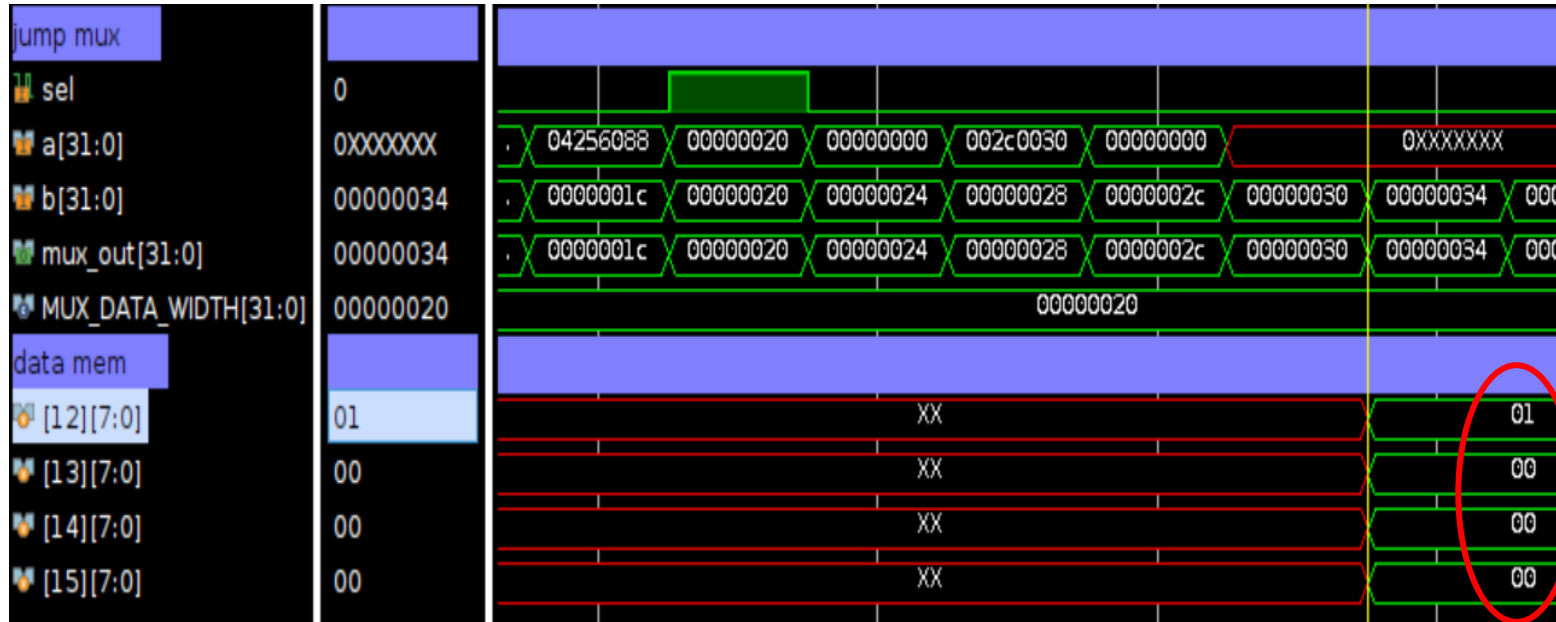
sw \$t3, 12(\$zero) 에 의해

mem[15:12] = 0x00000001

이 되는지 확인하기

2) 점프 시뮬레이션 수행





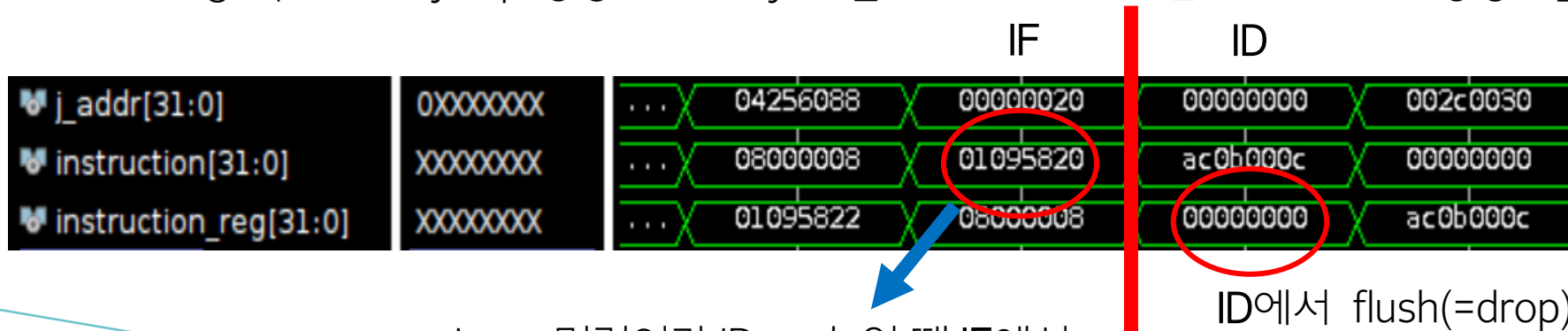
sub \$t3, \$t0, \$t1

jump L2

sw \$t3, 12(\$zero)

확인 완료

+ flush 기능 확인하기 : jump 명령어가 ID cycle일 때 IF에서 fetch된 next 주소의 명령어를 ID에서 flush(=drop) 함



jump 명령어가 ID cycle일 때 IF에서
fetch된 next 주소의 명령어

3. Hazard 와 stall 검증



Mem hazard 에 의해 pipeline이 Stall 됨.

Forwarding 에 의해 add 명령어의 EX 에서 정확히 갱신된 앞의 lw로 읽은 값이 반영

add, sw에 의한 data hazard를 처리 하기 위해 sw의 저장할 값이 add의 결과값으로 갱신되게 함.

alu_acc with mul_div_control

상세 내용

+ 코드는 git에 별도 첨부하였습니다!

Verilog 코드(서브 모듈)

add_subNbit

: 뺄셈도 수행할 수 있도록 감수의 2의 보수를 취해주는 과정을 XOR을 이용하여 설계 즉, cin이 1이면 뺄셈, 그 외는 덧셈을 수행하게 됨
⇒ ALU의 사칙연산을 담당하는 하위 모듈

```
module add_subNbit_v2 #(parameter N = 4)(
    input [N-1:0] A, // 4bit bus A [Msb:Lsb]
    input [N-1:0] B,
    input Cin,
    output [N-1:0] S,
    output Cout
);

    assign {Cout,S} = A + ( B ^ {N{Cin}} ) + Cin;
    // if add, cin=0 or if sub, cin=1
endmodule
```

reg_1bit

: ALU의 연산결과를 바탕으로 SREG의 상태를 저장하기 위한 하위 모듈

```
module reg_1bit (
    input clk, rstb, in_en, d,
    output reg q
);

    always @(posedge clk or negedge rstb)begin
        if(!rstb)begin
            q <= 0;
        end
        else begin
            if(in_en) q <= d;
            else q <= q;
        end
    end
endmodule
```

Verilog 코드

ALU 설계 명시

- sub,div 제어신호가 1이면 뺄셈,
그 외는 덧셈을 수행하도록 Cin에 매핑
하여 설계

- 컴퓨터 구조 때 배운 SREG의 일부
상태 표현 기능(s,c,z비트) 구현

```
module alu_v1(
    input clk, rstb,
    input op_add, op_sub, op_mul, op_div,
    input op_and, op_or,
    input [3:0] acc_high_data,
    input [3:0] bus_reg_data,
    output [3:0] alu_out,
    output Cout, zero_flag, sign_flag, carry_flag
);
    wire [3:0] alu_out_w;

    assign alu_out = (op_and==1) ? acc_high_data & bus_reg_data :
                      (op_or==1) ? acc_high_data | bus_reg_data : alu_out_w ;
    // logic

    add_subNbit_v2 #(N(4)) add_sub4bit(
        .A(acc_high_data),
        .B(bus_reg_data),
        .Cin(op_sub|op_div),
        .S(alu_out_w),
        .Cout(Cout)
    );

    reg_1bit sign_f(
        .clk(clk), .rstb(rstb), .in_en(op_sub), .d( (op_sub&(!Cout)) ), .q(sign_flag)
    );

    reg_1bit carry_f(
        .clk(clk), .rstb(rstb), .in_en(1), .d( (op_add|op_div)&Cout ), .q(carry_flag)
    );

    reg_1bit zero_f(
        .clk(clk), .rstb(rstb), .in_en(1), .d( (alu_out_w == 4'b0000) ), .q(zero_flag)
    );

endmodule
```

Verilog 코드

ACC 설계 명시

- 곱셈, 나눗셈은 중간 연산 값의 상위 4비트와 하위 4비트의 동작이 달라 분리 필요

- 곱셈, 나눗셈은 연산 수행 전 acc의 하위 4비트에 승수 or 피제수 데이터가 저장 되어야 함.

⇒ 처음 중간 연산값이 저장되기전 sel 신호가 11 즉, load 기능이 활성화 되고, 그 후에 시프트 하도록 sel이 바뀌는 과정을 설계 해야함
(이는 top모듈에서 구체적으로 설명)

- 곱셈은 중간 연산값의 우측시프트 과정의 연속으로 설계

- 나눗셈은 alu의 cout값이 0 즉, 음수라면 결과값이 acc에 반영 x, 시프트 시 fill_data도 0

- 나눗셈은 alu의 cout값이 1 즉, 양수라면 결과값이 acc에 반영, 시프트 시 fill_data도 1

```
module acc_v2 (
    input clk, rstb,
    input fill_data, alu_sel,
    input [3:0] alu_out, bus_reg_data,
    input [1:0] high_sel, low_sel,
    output reg [3:0] acc_high_data, acc_low_data
    // 00 : maintain, 01 : left shift, 10 : right shift, 11 : load
);
wire [3:0] data_in;
assign data_in = (alu_sel) ? alu_out : bus_reg_data;
always @(posedge clk or negedge rstb)begin
    if(!rstb)begin
        acc_high_data <= 0;
    end
    else begin
        if(high_sel==2'b00)begin
            acc_high_data <= acc_high_data;
        end
        else if(high_sel==2'b01)begin // about div
            if(fill_data) acc_high_data <= {data_in[2:0], acc_low_data[3]};
            else acc_high_data <= {acc_high_data[2:0], acc_low_data[3]};
        end
        else if(high_sel==2'b10)begin // about mul
            acc_high_data <= {fill_data, data_in[3:1]};
        end
        else acc_high_data <= fill_data; // fill_data is about carry
    end
end
always @(posedge clk or negedge rstb)begin
    if(!rstb)begin
        acc_low_data <= 0;
    end
    else begin
        if(low_sel==2'b00)begin // sel's separating is needed for optimization
            acc_low_data <= acc_low_data;
        end
        else if(low_sel==2'b01)begin // about div
            acc_low_data <= {acc_low_data[2:0], fill_data};
        end
        else if(low_sel==2'b10)begin // about mul
            acc_low_data[3] <= data_in[0];
            acc_low_data[2:0] <= acc_low_data[3:1];
        end
        else acc_low_data <= data_in;
    end
end
endmodule
```

Verilog 코드

Mul_controller 설계 명시

- 곱셈 알고리즘에 의해 제수의 lsb부터 차례대로 판단하여 1이면 피제수를 alu로 보내고 0이면 0을 보내는 과정을 설계

- 4비트끼리의 곱셈은 총 4번의 덧셈과정으로 설계

⇒ 4번을 세는 카운터를 설계하여 덧셈 제어 신호와 피승수를 alu로 보내고, 이 4번의 클럭주기동안 피승수 데이터가 내부에 저장되도록 설계

- complete 제어 신호는 else 구문에 위치하여 곱셈 or 나눗셈이 완료된 후에는 acc값을 시프트가 아닌 유지시키기 위해 생성

```
module mul_control_v2(
    input clk, rstb,
    input acc_lsb,
    input mul_en, // from CU
    input [3:0] multiplicand,
    output reg alu_sel, add_en, mul_complete, // if mul is complete
    output wire [3:0] multiplicand_to_alu
);
reg [2:0] mul_count;
reg [3:0] multiplicand_save;

assign multiplicand_to_alu = (acc_lsb) ? multiplicand_save : 0;

always @(posedge clk or negedge rstb) begin
    if(!rstb) begin
        mul_count <= 0;
        add_en <= 0;
        mul_complete <= 1;
        alu_sel <= 0;
        multiplicand_save <= 0;
    end
    else begin
        if(mul_en)begin // mul_en is 1 clock
            mul_count <= mul_count + 1;
            add_en <= 1; // to alu
            mul_complete <= 0;
            alu_sel <= 1;
            multiplicand_save <= multiplicand;
        end
        else if((mul_count)&&(mul_count != 3'b100))begin
            mul_count <= mul_count + 1;
            add_en <= 1;
            mul_complete <= 0;
            alu_sel <= 1;
            multiplicand_save <= multiplicand_save;
        end
        else begin
            mul_count <= 0;
            add_en <= 0;
            mul_complete <= 1;
            alu_sel <= 0;
            multiplicand_save <= 0;
        end
    end
end
endmodule
```

Verilog 코드

Div_controller 설계 명시

- 4비트끼리의 나눗셈은 총 5번의 뺄셈
과정으로 설계

⇒ 5번을 세는 카운터를 설계하여 뺄셈
제어신호와 제수를 alu로 보내고, 이 5번
의 클럭주기동안 제수 데이터가 내부에
저장되도록 설계

- 5번의 뺄셈이 끝나면 complete 신호
생성 및 초기화 과정을 카운터의 else 구
문에 설계

```
module div_control_v1(
    input clk, rstb,
    input div_en, // from CU
    input [3:0] divisor,
    output reg alu_sel, sub_en, div_complete,
    output wire [3:0] divisor_to_alu
);

reg [2:0] div_count;
reg [3:0] divisor_save;

assign divisor_to_alu = divisor_save;

always @(posedge clk or negedge rstb) begin
    if(!rstb) begin
        div_count <= 0;
        sub_en <= 0;
        div_complete <= 1;
        alu_sel <= 0;
        divisor_save <= 0;
    end
    else begin
        if(div_en) begin // div_en is 1 clock
            div_count <= div_count + 1;
            sub_en <= 1;
            div_complete <= 0;
            alu_sel <= 1;
            divisor_save <= divisor;
        end
        else if( (div_count)&&(div_count != 3'b101) ) begin
            div_count <= div_count + 1;
            sub_en <= 1;
            div_complete <= 0;
            alu_sel <= 1;
            divisor_save <= divisor_save;
        end
        else begin
            div_count <= 0;
            sub_en <= 0;
            div_complete <= 1;
            alu_sel <= 0;
            divisor_save <= 0;
        end
    end
end
endmodule
```


Verilog 코드 (TOP 모듈)

ALU_ACC(top 모듈) 설계 명시 1

- 곱셈, 나눗셈 제어신호를 받게 되면
그에 맞는 acc의 제어신호를 보내게
됨

- acc의 sel 신호는 첫 클럭주기에 3을
나타내어 데이터를 로드 시킨 후,
Complete 신호를 판단하여 곱셈, 나
눗셈의 완료 여부를 따져 완료되지 않
으면 좌 or 우측 시프트 제어신호를
acc에 보냄

```
module alu_acc_v1(  
    input clk, rstb,  
    input op_add, op_sub, // op_mul and op_div enter in mul or div_control module  
    input op_and, op_or,  
    input [3:0] bus_reg_data1, // multiplicand or divisor from CU  
    input [3:0] bus_reg_data2, // multiplier or dividend from CU  
    input mul_en, div_en, // from CU  
    output wire [3:0] acc_high_data, acc_low_data,  
    output zero_flag, sign_flag, carry_flag  
);  
  
wire Cout_w;  
wire [1:0] high_sel_w, low_sel_w;  
wire [3:0] alu_out_w;  
wire alu_sel_w_mul, alu_sel_w_div;  
wire mul_en_r, div_en_r; // when mul or div  
wire [3:0] multiplicand_to_alu, divisor_to_alu;  
wire mul_complete, div_complete;  
  
assign high_sel_w = (mul_en|div_en) ? 3 : (!mul_complete) ? 2 :  
    (!div_complete) ? 1 : 0; // + more coding  
assign low_sel_w = (mul_en|div_en) ? 3 : (!mul_complete) ? 2 :  
    (!div_complete) ? 1 : 0; // + more coding
```

모듈 이어서 계속

Verilog 코드 (TOP 모듈)

ALU_ACC(top 모듈) 설계 명시 2

- 원래 덧셈기능을 상실하지 않기 위해 bus_reg_data1이 bus_reg_data에 OR게이트로 매핑 되어있음
- \times, \div *controller* 에서 나온 피승수, 제수를 alu에 전달하기 위해 wire로 매핑
- acc에 저장되는 데이터를 결정하는 제어신호는 \times, \div *controller* 에서 출력되기 때문에 wire로 alu_sel에 매핑
- 승수 or 피제수 데이터를 bus_reg_data2로 입력 시켜 acc의 bus_reg_data 핀에 매핑되도록 함

```
alu_v1 U1(  
    .clk(clk),  
    .rstb(rstb),  
    .op_add(op_add),  
    .op_sub(op_sub),  
    .op_mul(mul_en_r),  
    .op_div(div_en_r),  
    .op_and(op_and),  
    .op_or(op_or),  
    .acc_high_data(acc_high_data),  
    .bus_reg_data(bus_reg_data1 | multiplicand_to_alu | divisor_to_alu),  
    .alu_out(alu_out_w),  
    .zero_flag(zero_flag),  
    .sign_flag(sign_flag),  
    .carry_flag(carry_flag),  
    .Cout(Cout_w)  
);  
  
acc_v2 U2(  
    .clk(clk),  
    .rstb(rstb),  
    .fill_data(Cout_w),  
    .alu_sel(alu_sel_w_mul | alu_sel_w_div),  
    .alu_out(alu_out_w),  
    .bus_reg_data(bus_reg_data2),  
    .high_sel(high_sel_w),  
    .low_sel(low_sel_w),  
    .acc_high_data(acc_high_data),  
    .acc_low_data(acc_low_data)  
);
```

모듈 이어서 계속

Verilog 코드 (TOP 모듈)

ALU_ACC(top 모듈) 설계 명시 3

- acc의 lsb는 말 그대로 acc_low_data[0]으로 매핑

- 피승수 or 제수를 controller 내부에 저장하고 있어야 하므로 bus_reg_data1에 매핑하여 신호를 입력받음

```
mul_control_v2 U3(  
    .clk(clk),  
    .rstb(rstb),  
    .acc_lsb(acc_low_data[0]),  
    .mul_en(mul_en),  
    .multiplicand(bus_reg_data1),  
    .alu_sel(alu_sel_w_mul),  
    .add_en(mul_en_r),  
    .mul_complete(mul_complete),  
    .multiplicand_to_alu(multiplicand_to_alu)  
);  
  
div_control_v1 U4(  
    .clk(clk),  
    .rstb(rstb),  
    .div_en(div_en),  
    .divisor(bus_reg_data1),  
    .alu_sel(alu_sel_w_div),  
    .sub_en(div_en_r),  
    .div_complete(div_complete),  
    .divisor_to_alu(divisor_to_alu)  
);  
  
endmodule
```

RTL 시뮬레이션

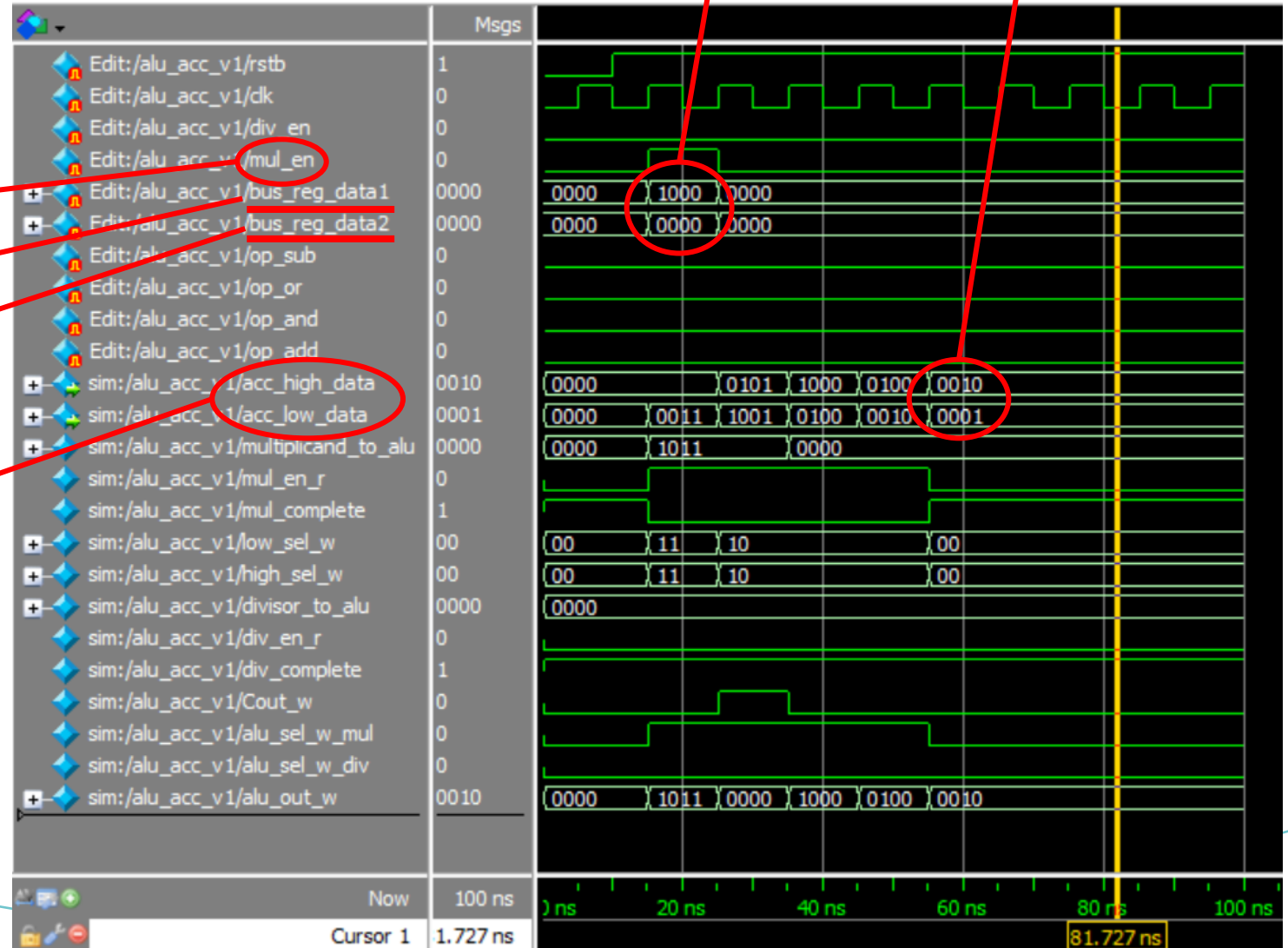
1. $1011 \times 0011 = ?$

곱셈 제어 신호

피승수

승수

곱셈 출력 결과값



(값 표시 오류정정)

1011
0011

곱셈 최종결과
(00100001)

RTL 시뮬레이션

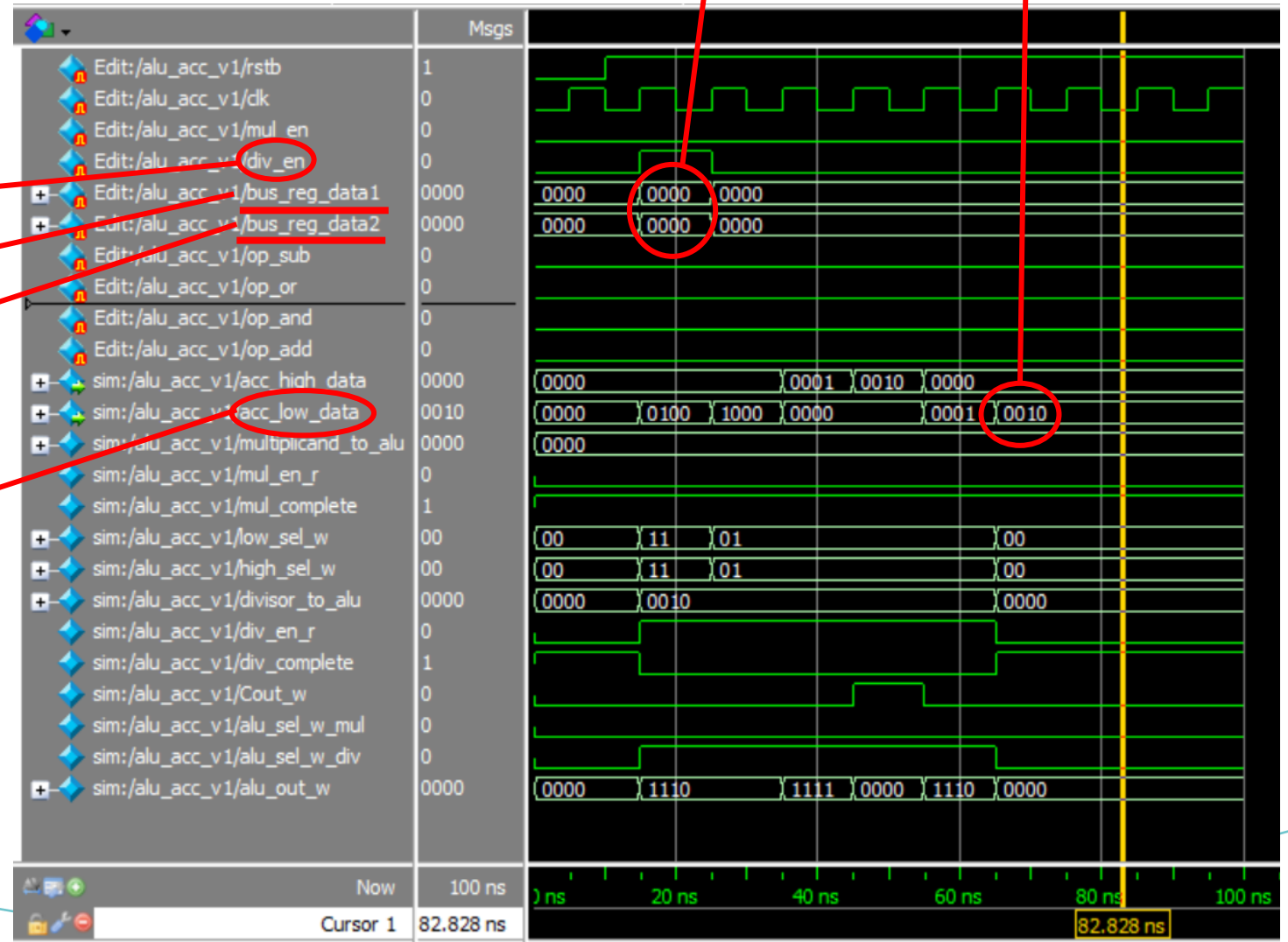
2. $0100 \div 0010 = ?$

나눗셈 제어 신호

제수

피제수

나눗셈 출력 결과값



감사합니다!