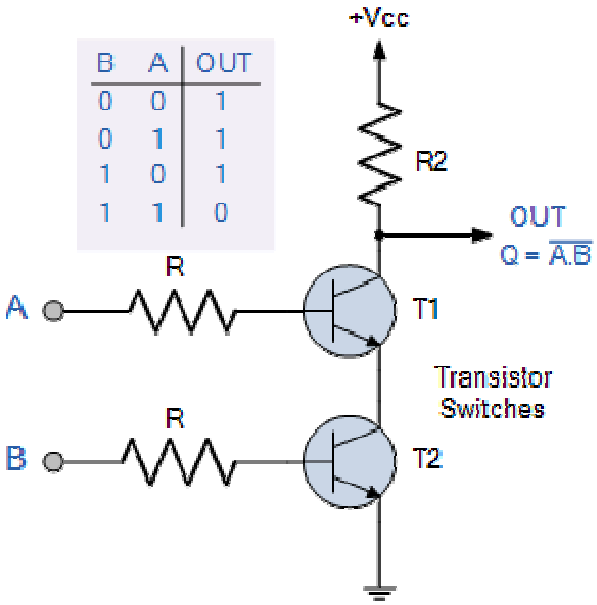
중간 정리

Transistor NAND Gate

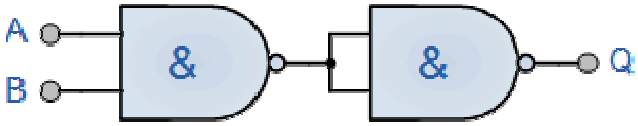


The “Universal” NAND Gate

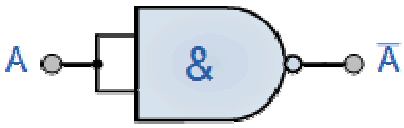
The Logic NAND Gate is generally classed as a “Universal” gate because it is one of the most commonly used logic gate types. They can also be used to produce any other type of logic gate function, but in practice being universal it forms the basis of most practical logic circuits.

By connecting them together in various combinations the three basic gate types of AND, OR and NOT function can be formed using only NAND gates, for example.

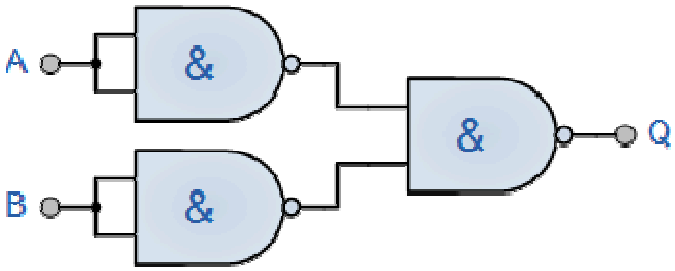
Various Logic Gates



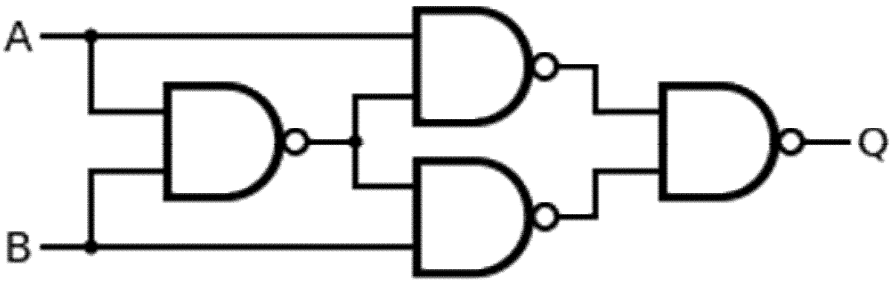
2-input AND Gate



NOT Gate



2-input OR Gate



2-input XOR Gate

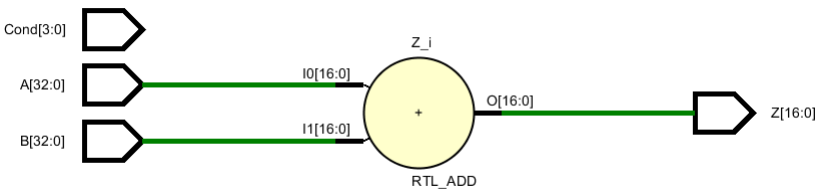
HA->FA->ADD4->SUB4->CMP4

&, |, ~, ^, +, -, (*, /)

ALU 구현해 보기

덧셈기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output [31:0] Z,
    input [3:0] Cond
    );

    assign Z = A + B;

endmodule
```

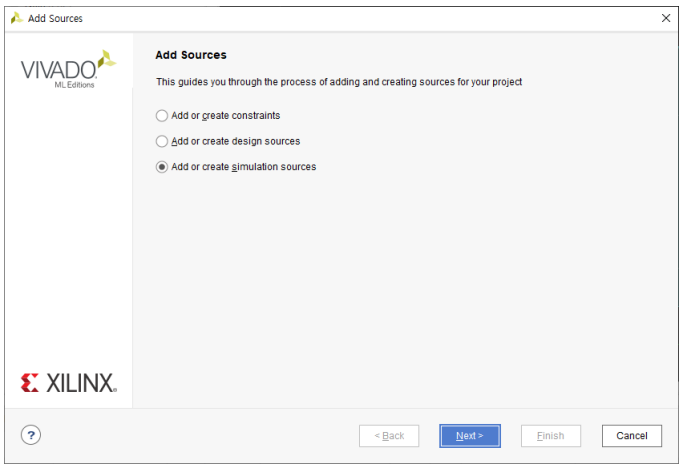


RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

테스트 벤치로 결과 확인

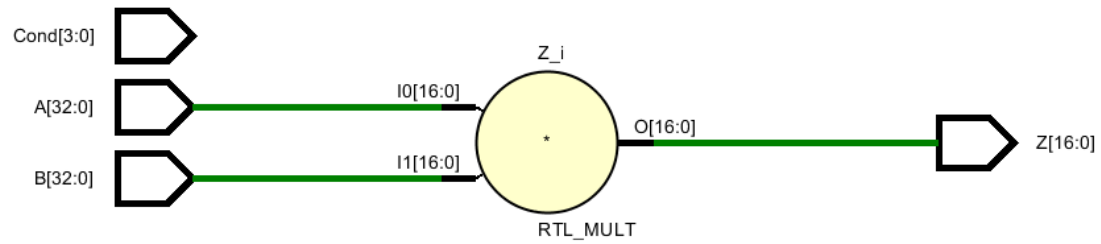
```
module tb_ALU_32(

    );
```

```
    reg [31:0] A = 0;
    reg [31:0] B = 0;
    wire [31:0] Z;
    wire [3:0] Cond = 0;

    ALU_32 alu_32(.A(A), .B(B), .Z(Z), .Cond(Cond));

    always #5 A = A + 1;
    always #10 B = B + 1;

endmodule
```



곱셈기 변경해보기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output [31:0] Z,
    input [3:0] Cond
    );

    assign Z = A * B;

endmodule
```

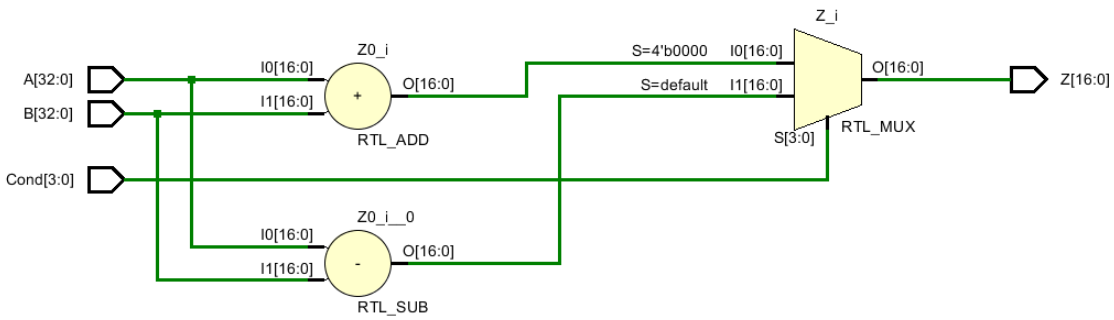


RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

덧셈기와 뺄셈기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output [31:0] Z,
    input [3:0] Cond
    );

    assign Z = Cond==4'b0000 ? A + B :
                               A - B ;

endmodule
```

삼항 연산자를 이용한 간단한 조건문
삼항 연산자는 간단한 if-else 문으로 mux로 변환됩니다.
mux는 and와 or 게이트를 이용하여 구성할 수 있습니다.
*** 물음표는 숫자에서 한 칸 떼어주어야 합니다.



RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

테스트 벤치로 결과 확인

```
module tb_ALU_32(

    );

    reg [31:0] A = 0;
    reg [31:0] B = 0;
    wire [31:0] Z;
    reg [3:0] Cond = 0;

    ALU_32 alu_32(.A(A), .B(B), .Z(Z), .Cond(Cond));

    always #5 A = A + 1;
    always #10 B = B + 1;
    always #20 Cond = Cond + 1;

endmodule
```

Mux 구현해보기 1 : 3항 연산자

```
module MYMUX32(
    input A,
    input B,
    input Cond,
    output Z
);

    assign Z=(A&~Cond)|(B&Cond);

endmodule
```

테스트 벤치로 결과 확인

```
module tb_MYMUX32(

    );

    reg A=0;
    reg B=0;
    reg Cond=0;
    wire Z;
```

```
    MYMUX32 mymux32_0(.A(A),.B(B),.Cond(Cond),.Z(Z));
    always #1 A=~A;
    always #2 B=~B;
    always #4 Cond=~Cond;

endmodule
```

Mux 구현해보기 2 : 3항 연산자

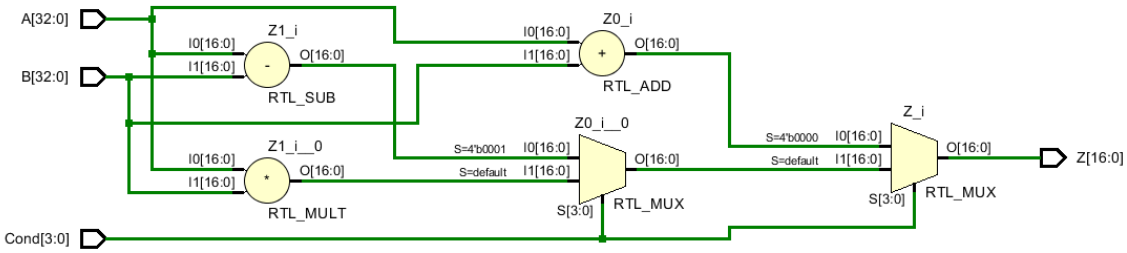
```
module MYMUX32(
    input [31:0] A,
    input [31:0] B,
    input Cond,
    output [31:0] Z
);

    wire [31:0] tCond={32{Cond}};
    assign Z=(A&~tCond)|(B&tCond);

endmodule
```

테스트 벤치로 결과 확인

```
module tb_MYMUX32(

    );

    reg [31:0] A=0;
    reg [31:0] B=0;
    reg Cond=0;
    wire [31:0] Z;

    MYMUX32 mymux32_0(.A(A),.B(B),.Cond(Cond),.Z(Z));
    always #1 A=A+1;
    always #2 B=B+1;
    always #4 Cond=~Cond;

endmodule
```

곱셈기 추가해보기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output [31:0] Z,
    input [3:0] Cond
    );

    assign Z = Cond==4'b0000 ? A + B :
               Cond==4'b0001 ? A - B :
                               A * B ;

endmodule
```



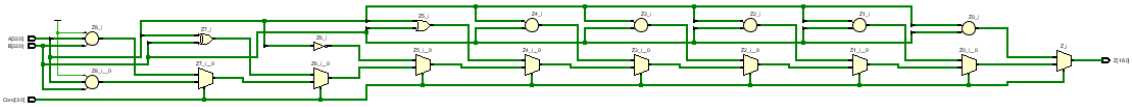
RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

테스트 벤치로 결과 확인

ALU 구현하기 1

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output [31:0] Z,
    input [3:0] Cond
    );

    assign Z = Cond==4'b0000 ? A + B :
               Cond==4'b0001 ? A - B :
               Cond==4'b0010 ? A * B :
               Cond==4'b0011 ? A / B :
               Cond==4'b0100 ? A % B :
               Cond==4'b0101 ? A | B :
               Cond==4'b0110 ? ~A :
```

```
                Cond==4'b0111 ? A ^ B :
                Cond==4'b1000 ? A << B :
                                A >> B;

endmodule
```



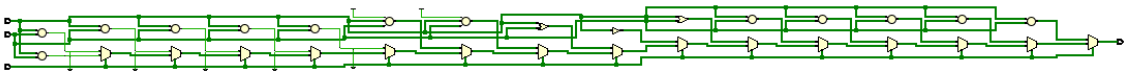
RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

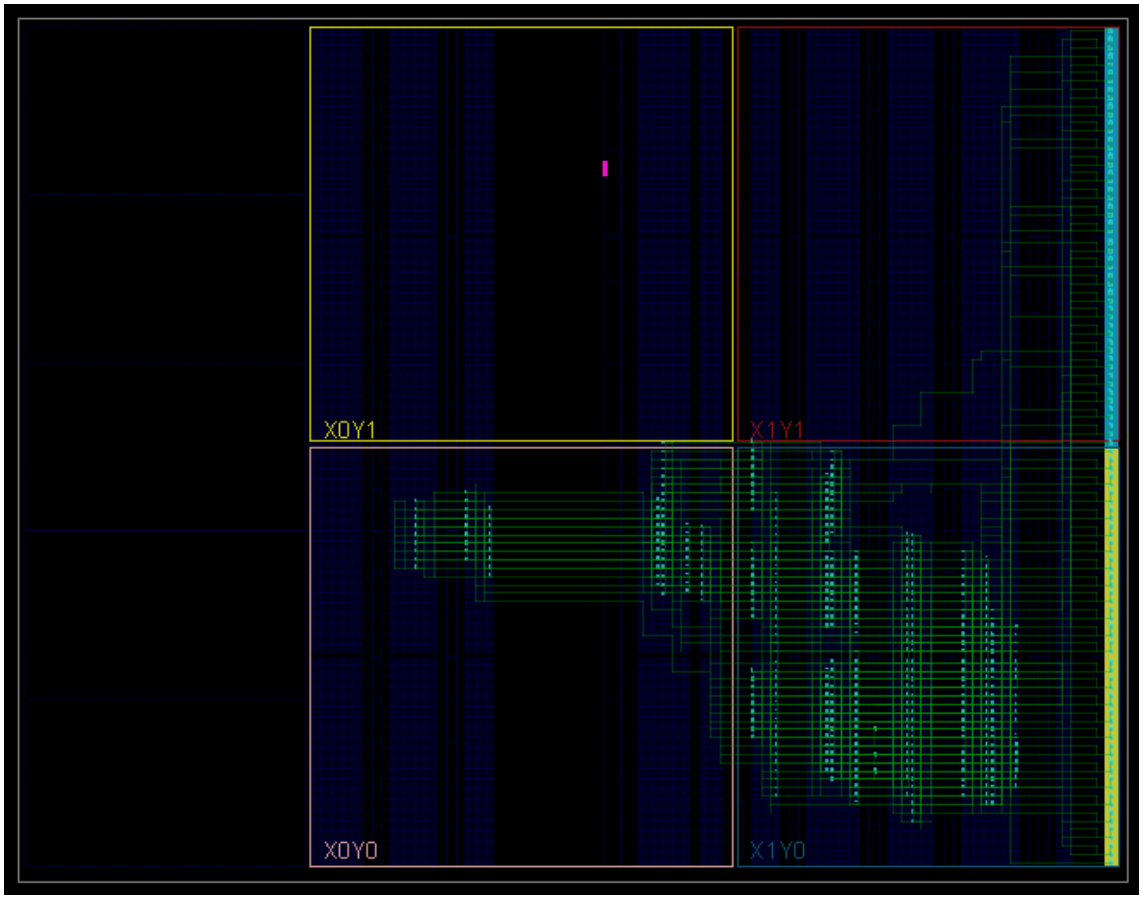
테스트 벤치로 결과 확인

ALU 구현하기 2

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output [31:0] Z,
    input [3:0] Cond
    );

    assign Z = Cond==4'b0000 ? A + B :
               Cond==4'b0001 ? A - B :
               Cond==4'b0010 ? A * B :
               Cond==4'b0011 ? A / B :
               Cond==4'b0100 ? A % B :
               Cond==4'b0101 ? A | B :
               Cond==4'b0110 ? ~A :
               Cond==4'b0111 ? A ^ B :
               Cond==4'b1000 ? A << B :
               Cond==4'b1001 ? A >> B :
               Cond==4'b1010 ? { {31{1'b0}}, A == B } :
               Cond==4'b1011 ? { {31{1'b0}}, A != B } :
               Cond==4'b1100 ? { {31{1'b0}}, A >  B } :
               Cond==4'b1101 ? { {31{1'b0}}, A >= B } :
               Cond==4'b1110 ? { {31{1'b0}}, A <  B } :
                               { {31{1'b0}}, A <= B };

endmodule
```

62페이지 참조





RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

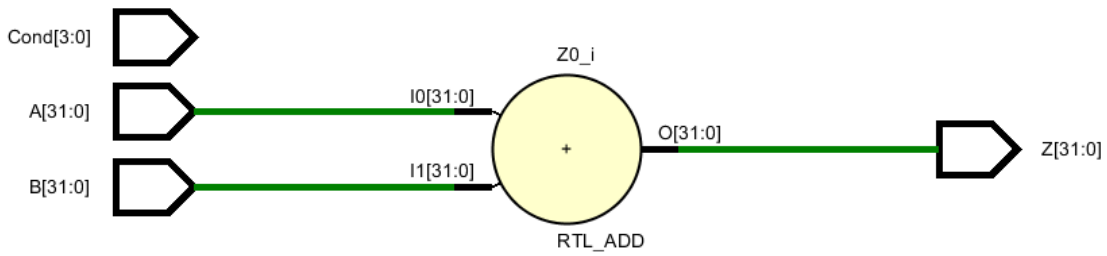
테스트 벤치로 결과 확인

Mux 구현해보기

always 문 써보기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] Z,
    input [3:0] Cond
    );

    always @(A, B, Cond)
    begin
        Z = A + B;
    end

endmodule
```

if 문을 사용하기 위한 always 문



RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

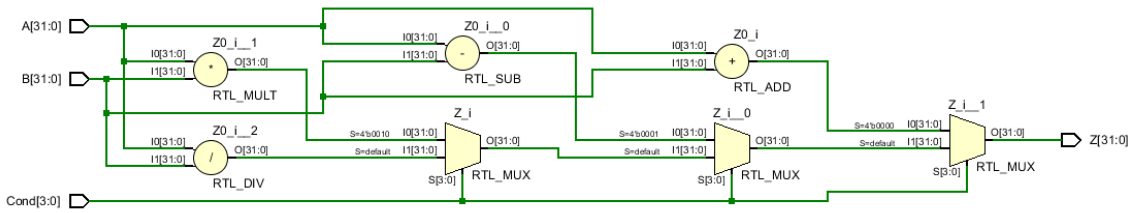
테스트 벤치로 결과 확인

if 문 써보기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] Z,
    input [3:0] Cond
```

```
    );

    always @(A, B, Cond)
    begin
        if(Cond==4'b0000)
            Z = A + B;
        else if(Cond==4'b0001)
            Z = A - B;
        else if(Cond==4'b0010)
            Z = A * B;
        else
            Z = A / B;
    end

endmodule
```

if 문 추가 - always 안에서만!!!



RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

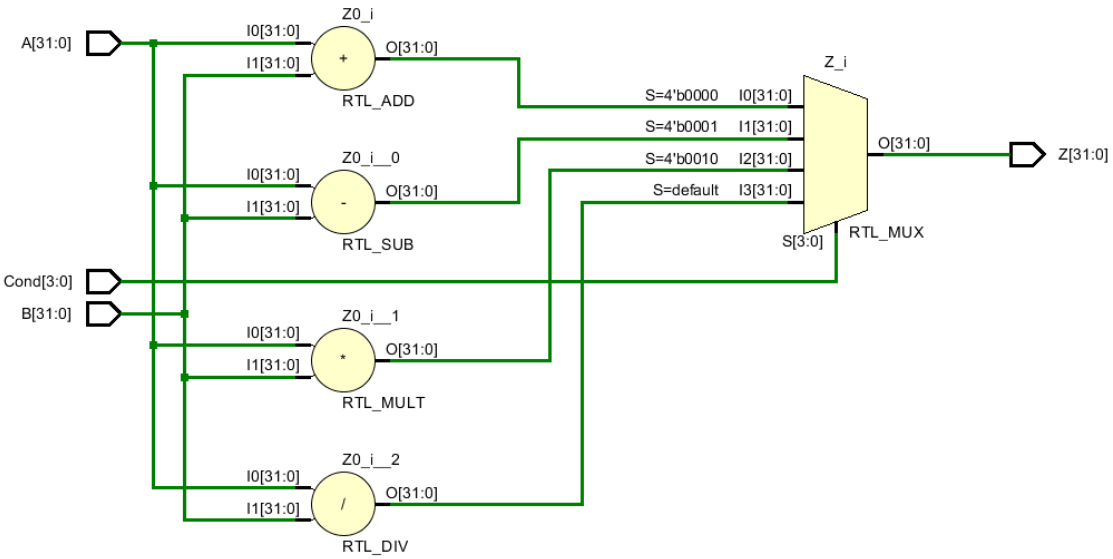
테스트 벤치로 결과 확인

case 문 써보기

```
module ALU_32(
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] Z,
    input [3:0] Cond
    );

    always @(A, B, Cond)
    begin
        case(Cond)
        4'b0000: Z = A + B;
```

```
        4'b0001: Z = A - B;
        4'b0010: Z = A * B;
        default: Z = A / B;
        endcase
    end

endmodule
```

case 문도 always 문 안에서만!!!



RTL, 합성, 구현 스키매틱 확인, 디바이스 확인
각 창 닫기

테스트 벤치로 결과 확인