Verilog 기초

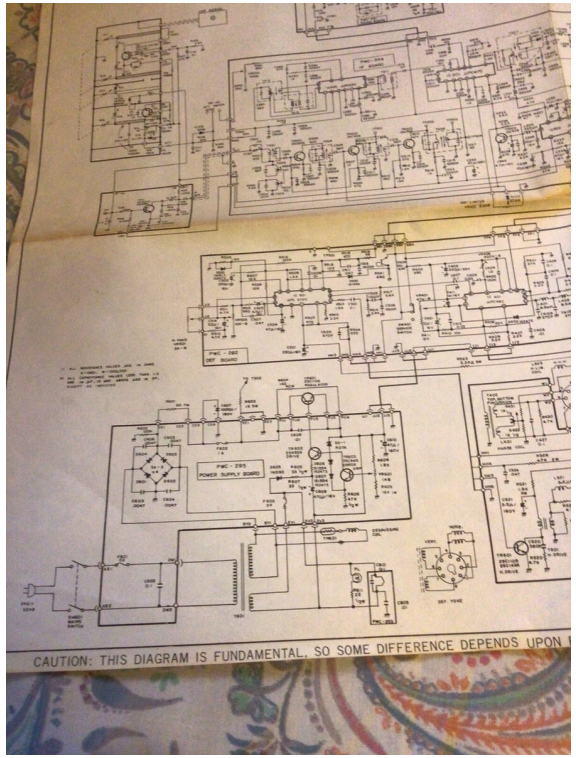
https://youtu.be/Rw-1eq51S2k

Verilog

- Verilog는 디지털 시스템과 회로를 설계하기 위해 사용되는 하드웨어 기술 언어이다.

- Verilog와 같은 HDL이 없을 때는 하드웨어 설계자가 직접 회로도를 그려 도면으로써 하드웨어를 설계함
  - HDL 하드웨어 기술 언어 – Hardware Description Language

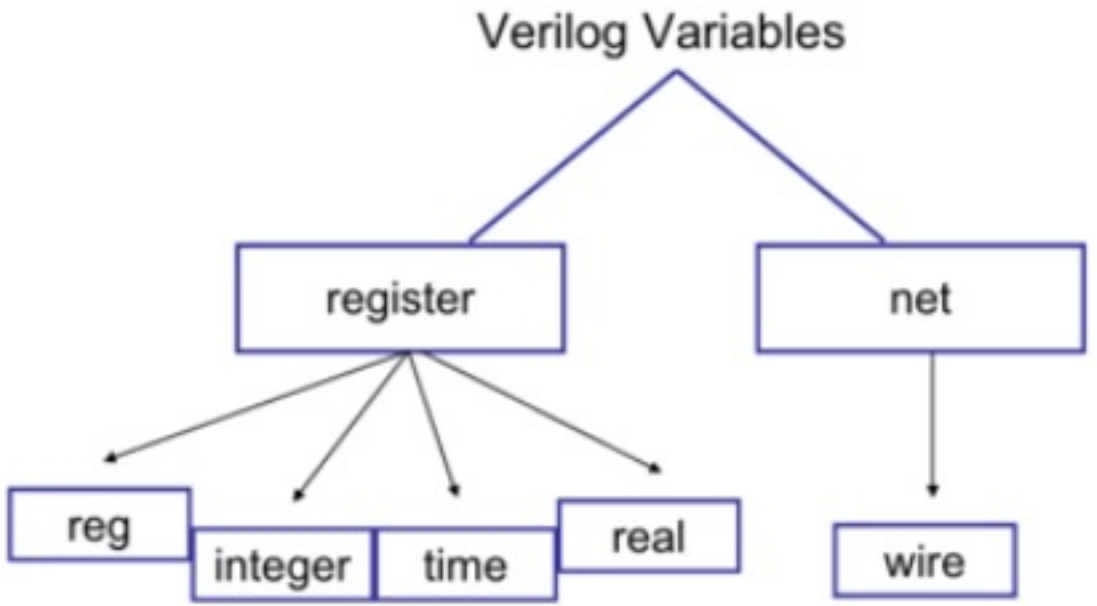
- 그림으로 그리는 것 대신 HDL을 통해서 하드웨어 구조 또는 작동의 정의를 하고 시뮬레이션으로 검증을 수행.



Verilog

- Verilog의 변수에는 register와 net 두 가지 변수 종류가 존재.

- register는 값을 저장하고 기억하는 역할
- net은 회로에서 이동하는 경로 역할



Verilog

- register의 타입
  - reg : 기본적인 저장 변수
  - integer : 정수 값을 저장하는 변수
  - time : 시간 값을 저장하는 변수, 주로 시뮬레이션에서 사용
  - real : 실수 값을 저장하는 변수, 부동소수점 연산이 필요할 때 사용.

- net의 타입
  - wire : net의 기본적인 형태.

- 이외에도 여러가지 타입이 존재
  - register – byte, shortint, longint
  - net– wand, wor, tri, trireg 등

Verilog

- 회로를 만들 때, 모듈을 사용함. 즉, c언어의 함수랑 비슷함.
  - c언어의 함수와 차이는 병렬로 처리되는 것과 항상 동작하는 점.

- 모듈을 구현할 때는 module로 시작해서, endmodule로 끝냄.

표 2.8 Verilog HDL의 연산자

연산자	기 능	연산자	기 능
{}, {{}}	결합, 반복	^	비트 exclusive or (비트 xor)
+, -, *, /, **	산술	^~ 또는 ~^	비트 등가 (비트 xnor)
%	나머지	&	축약 and
>, >=, <, <=	관계	~&	축약 nand
!	논리 부정	|	축약 or
&&	논리 and	~|	축약 nor
||	논리 or	^	축약 xor
==	논리 등가	^~ 또는 ~^	축약 xnor
!=	논리 부등	<<	논리 왼쪽 시프트
===	case 등가	>>	논리 오른쪽 시프트
!==	case 부등	<<<	산술 왼쪽 시프트
~	비트 부정	>>>	산술 오른쪽 시프트
&	비트 and	? :	조건
|	비트 inclusive or	or	Event or

Verilog

- 기본적인 and_gate 코드 (.v 확장자를 사용)

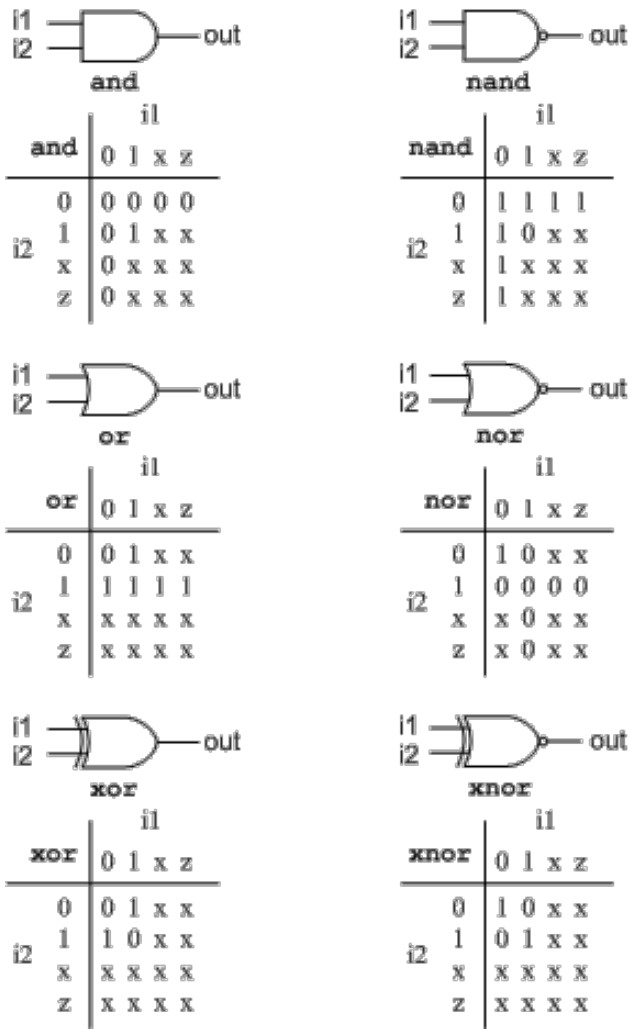
```
module 전구_회로(스위치, 전구);
    input 스위치;
    output 전구;

    assign 전구 = 스위치;
endmodule
```

```
// and_gate.v
module and_gate ();
    input wire a;       // 입력 신호 a
    input wire b;       // 입력 신호 b
    output wire y;      // 출력 신호 y

    // AND 연산을 수행하여 y에 결과를 할당
    assign y = a & b;
endmodule
```

```
// and_gate.v
module and_gate (
    input wire a,     // 입력 신호 a
    input wire b,     // 입력 신호 b
    output wire y     // 출력 신호 y
);
    // AND 연산을 수행하여 y에 결과를 할당
    assign y = a & b;
endmodule
```



Verilog 테스트벤치

- 실제 하드웨어는 멈추지 않고 계속 동작함.
- initial block 내부에 있는 코드는 시간 순서에 따라서 위에서 아래로 실행됨. 즉, 병렬로 실행되지 않음.
  - 이러한 특징으로, 절차적인 block 이라고 함

- 시뮬레이션에서 한번만 실행되는 내용으로 초기화나 신호 값을 보깅 위해서 사용됨.

```
reg A, B, SEL;
initial begin
    #0  SEL = 0; A = 0; B = 0;
    #10 A = 1;
    #10 SEL = 1;
    #10 B = 1;
end
```

Verilog 테스트벤치

- Icarus Verilog
  - verilog 코드를 컴파일하고 시뮬레이션 할 수 있는 프로그램.
  - .v 코드를 실행하고 결과 파일을 생성. 결과 파일에는 파형 정보를 담고 있어 시간에 따라 어떻게 동작하는지 확인 가능
  - brew install icarus-verilog

- GTKWave
  - 파형을 확인하는 프로그램, 시뮬레이션 결과를 시간에 따른 신호 변화로 시각화해서 보여줌.
  - Icarus Veilog로 생성된 결과 파일을 불러와서, Verilog 회로에서 신호가 어떻게 변하는지 파형 그래프로 보여줌.
  - brew install gtkwave

Verilog 테스트벤치

```v
module and_gate (
    input wire a,
    input wire b,
    output wire y
);
    assign y = a & b;
endmodule

module or_gate (
    input wire a,
    input wire b,
    output wire y
);
    assign y = a | b;
endmodule
```

```v
module top_module;
    reg a, b;
    wire and_y, or_y;

    and_gate uut_and (
        .a(a),      // top_module의 a를 and_gate의 a에 연결
        .b(b),      // top_module의 b를 and_gate의 b에 연결
        .y(and_y)   // and_gate의 y를 top_module의 and_y에 연결
    );

    or_gate uut_or (
        .a(a),      // top_module의 a를 or_gate의 a에 연결
        .b(b),      // top_module의 b를 or_gate의 b에 연결
        .y(or_y)    // or_gate의 y를 top_module의 or_y에 연결
    );

    initial begin
        $display("Time\t a b | AND y | OR y");
        $display("------------------------------");

        a = 0; b = 0;
        #10;
        $display("%g\t %b %b | %b      | %b", $time, a, b, and_y, or_y);

        a = 0; b = 1;
        #10;
        $display("%g\t %b %b | %b      | %b", $time, a, b, and_y, or_y);

        a = 1; b = 0;
        #10;
        $display("%g\t %b %b | %b      | %b", $time, a, b, and_y, or_y);

        a = 1; b = 1;
        #10;
        $display("%g\t %b %b | %b      | %b", $time, a, b, and_y, or_y);

        $finish;
    end
endmodule
```

```
% iverilog -o result.out main.v
% vvp result.out
```

```
Time     a b | AND y | OR y
------------------------------
10       0 0 | 0     | 0
20       0 1 | 0     | 1
30       1 0 | 0     | 1
40       1 1 | 1     | 1
main.v:53: $finish called at 40 (1s)
```

Verilog

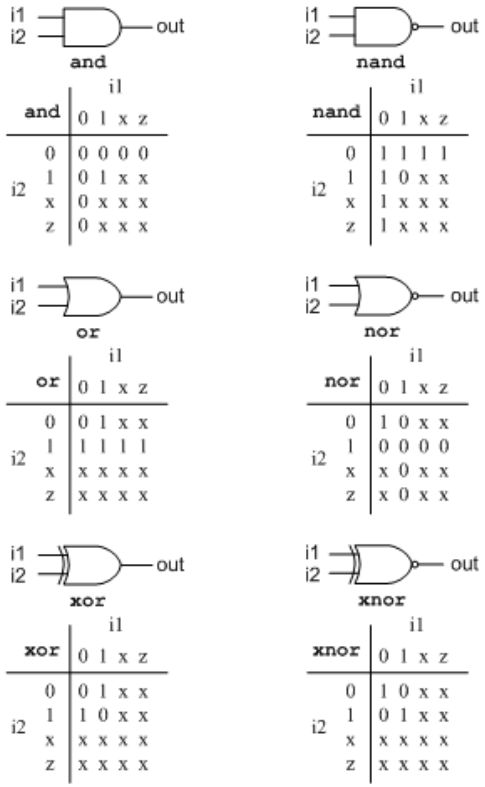
AND / OR 게이트

출력: 1개 / 입력: 多개

관련 모듈: and / or / xor / nana / nor / xnor

and | **or** | **nand** | **xor** | **nor** | **xnor** [instance name] (output, input1, ..., inputn);

논리연산에 가장 기초가 되는 게이트이다.



and	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	0	0	0	0
i2 = 1	0	1	x	x
i2 = x	0	x	x	x
i2 = z	0	x	x	x

nand	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	1	1	1	1
i2 = 1	1	0	x	x
i2 = x	1	x	x	x
i2 = z	1	x	x	x

or	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	0	1	x	x
i2 = 1	1	1	1	1
i2 = x	x	x	x	x
i2 = z	x	x	x	x

nor	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	1	0	x	x
i2 = 1	0	0	0	0
i2 = x	x	0	x	x
i2 = z	x	0	x	x

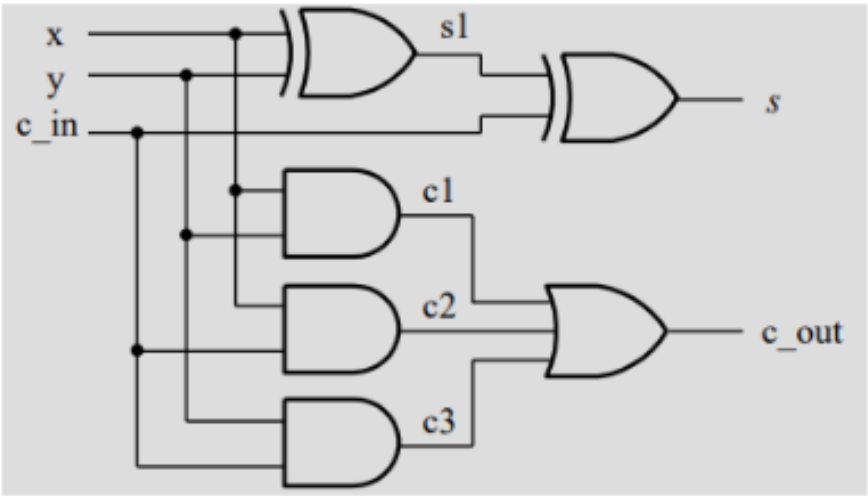
xor	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	0	1	x	x
i2 = 1	1	0	x	x
i2 = x	x	x	x	x
i2 = z	x	x	x	x

xnor	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	1	0	x	x
i2 = 1	0	1	x	x
i2 = x	x	x	x	x
i2 = z	x	x	x	x

1-bit Full Adder

```
module full_adder_structural(x, y, c_in, s, c_out);
    input x, y, c_in;
    output s, c_out;
    wire s1, c1, c2, c3;

    xor xor_s1(s1, x, y);
    xor xor_s2(s, s1, c_in);
    and and_c1(c1, x, y);
    and and_c2(c2, x, c_in);
    and and_c3(c3, y, c_in);
    or or_cout(c_out, c1, c2, c3);
endmodule
```



감 사 합 니 다