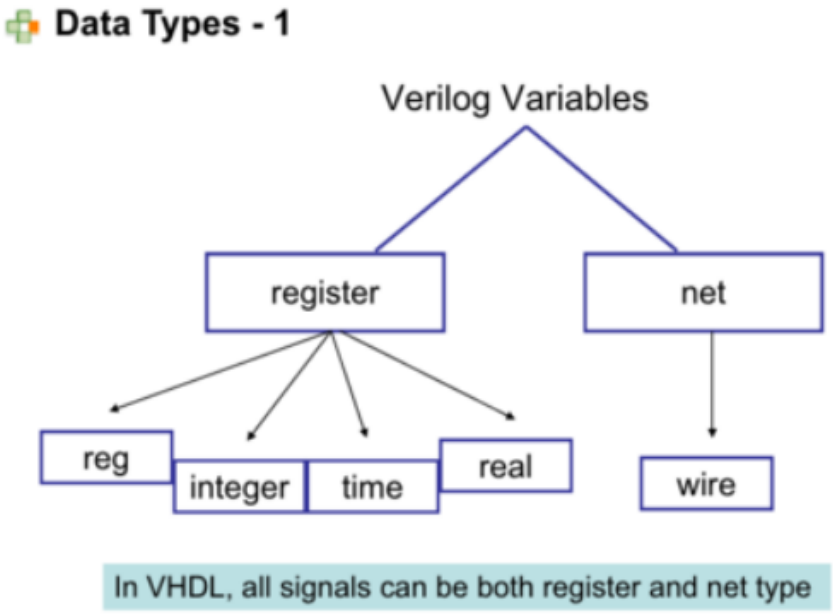
-verilog 1일차

VHDL은 Hardware Desciption Language로 디지털 회로 및 혼합 신호를 표현하는 하드웨어 언어이다.
디지털 회로 관점으로 말해보자면, 디지털 회로를 배우기 앞서서 디지털 논리를 배우고, 그리고 거기서 더 나아가서 AND, OR, NOR, XOR, FF 이렇게 더 관점이 확대된다. 그리고 디지털 논리는 간단한 식으로 표현이 가능하고, 이게 심볼로 확대되고 더 나아가 이를 회로적으로 구현하려면 이제 mos 이런 tr들을 이용해서 표현하는 것이다.

VHDL, verilog의 기본적인 사용법부터 기본연산, FF, 가능하다면 덧셈, 뺄셈, 곱셈, 나눗셈기를 만들어보고
마지막으로 FSM까지... 가능하면 해볼까한다ㅎㅎ



1. 가장 중요한 데이터 타입을 알아보자
먼저 디지털 논리 회로에는 항상 input과 output이 존재한다.
그리고 input에는 저장소인 register가 있어야 값을 받아서 연산을 하고, output 이 나온다.
이 output을 출력할 때 사용하는 것이 wire 변수이다. 연결해주는 선이라고 생각하면 쉬울 것 같다.
→ wire는 in/out/연결선(값을 단순히 출력해주는), reg는 레지스터(연산에 이용하는) 로 이해하자.
그렇다면 wire변수로는 연산과정을 할 수 없을까? 나중에 설명하고 넘어가보자.

Verilog는 로직을 구성하는 module부분과 해당 모듈을 시뮬레이션하기 위해 값을 입력하는 test bench부분이 있다.
 module의 기본적인 틀은 module의 이름을 정하고, 그 외에 어떤 input, output 신호를 사용할지 정한 후
그 안에서 in,out 간의 관계, state의 변화, 동작 등을 서술하고 module을 끝낸다. Test bench는 실습에서 진행하도록 하겠다.

디지털 논리회로는 회로 구성에 있어서 조합 회로와 순차회로에 대해 알아보도록 하겠다.

*** 순차회로와 조합회로**

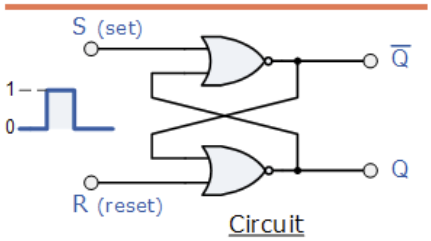
조합논리회로는 상태에 대한 정보를 갖고 있지 않으며, 오로지 입력신호에 따라 출력을 하게 되어 있으므로, 입력신호가 동일하다면 출력신호가 다를 수 없다.

순차논리회로는 이전의 상태에 대한 정보를 갖고 있고, 이에 따라 같은 입력을 받는 경우에도 다른 결과를 출력할 수 있다.
(순차논리회로는 조합논리회로와 플립플롭을 가지고 구성한 회로라고 보면됨)

순차라는 말의 의미는 시간에 따라 변화할 수 있음을 암시하고, 조합이라는 말의 의미는 회로의 구성조합에만 의존하는 회로임을 암시한당~~ 다시 말해서 순차논리회로란, 입력값이 들어가서 출력으로 나오고 또 그 출력이 다시 입력으로 작용하는 회로이다. 플립플롭이 그것이다. 플립플롭을 써서 회로를 구성하면 순차논리회로가 되는 것이다.(always문에서 사용, 나아가 FSM(Finite State Machine)을 실습파트에서 해보려고 한다. (순차회로의 꽃이라고 한다링)

*** 플립플롭 종류 (기억 장치)**
:플립플롭은 동기식과 비동기식으로 나누어 진다.동기식은 클럭이 있어서 클럭이 들어갈 때만 동작을 하는것이고, 비동기식은 클럭이 없는것.

★ SR Flip-Flop



Circuit

S	R	Q	$\overline{Q}$
0	0	No change	
0	1	0	1
1	0	1	0
1	1	0	0
		(Invalid)	

(Q와 ^Q는 보수이다.)

1) RS플립플롭: 기본이 됨. 두개의 NAND 게이트로 이루어져 있다. 두 개의 입력과 두개의 출력으로 되었고, 출력이 다시 입력으로 들어가게 된다. 입력이 0,0 일 때 출력은 이전 값의 보수가 되는 것이다. 나머지 플립플롭은 기본적으로 RS플립플롭을 가지고 만든다. 입력 S와 R에 0이 입력되면 출력 Q와 Q' 는 이전 값의 보수가 된다. 즉 값을 기억하는 것이다. 입력 S = 0, R = 1이 입력되면 Q = 0, Q' = 1로 변한다. 이를 리셋(Reset)이라한다. 리셋(Reset) 되었다는 소리는 출력 Q의 값이 0으로 되었을 경우를 말한다. 입력 S = 1, R = 0 이 입력되면 Q = 1, Q' = 0 으로 변한다. 이를 셋(Set)이라한다. 셋(Set) 되었다는 소리는 출력 Q의 값이 1로 되었을 경우를 말한다. 입력 S=1, R=1 이 입력되면 Q = 0 , Q' = 0 로 변하지만 문제점이 발생한다. 0도 1도 아닌 중간값을 갖는 상태가 지속되기 때문이다.
EN은 쉽게 말하면 클럭이라 할 수 있다.0이면 AND 게이트는 항상 0이므로 입력 S와 R의 값에 신경 쓰지 않는다.즉 EN이 1이 되어야 입력S와 R 값에 의해 결과 값이 변경된다.

2. verilog언어에서의 조합회로, 순차회로문 사용
always문은 해당 입력을 반복한다는 뜻이고, initial은 프로그램이 시작할 때, 변수를 초기화하는 역할로 이해하면 된다. 또 대표적으로 assign문이 있는데, 간단히 말하자면 조합회로에서 값을 할당할 때 사용한다.

2.1 always문과 assign문은 많이 쓰이므로 주의점
assign문은 값을 받는 signal은 wire로 선언해야하며 "=" 기호를 사용한다. assign문은 조합회로에서만 쓸 수 있는 반면에 always문은 조합회로와 순차회로에 둘다 쓰일 수 있다.

2.1.1 조합회로에서의 always문
특정 조건문을 사용하고자 할 때 쓰이며, 값을 받는 signal은 reg로 선언되어야한다. 같은 Signal을 2개 이상의 always문에서 값을 바꾸어서는 안되고, if문 case문 사용시 else와 default 문장을 빠트리지말자.
그리고 always문에는 이벤트 리스트라는 것이 있다. 이벤트 리스트에는 always문이 동작하기 위해 필요한 signal이 모두 포함되어야 한다!! (이벤트 리스트를 쉽게 구분하는 방법은 "=" 을 중심으로 우측항에 있는 signal과 if문 또는 case문의 조건에 사용되는 siganl를 포함시키면된다.)
(A=A+1 불가 조합회로에서는 같은 signal을 사용할 수 없다.)

2.1.2 순차회로에서의 always문
순차회로는 조합회로를 가지고 만든다. 순차회로를 만들 때는 반드시 always문을 사용하고, 마찬가지로 값을 할당 받는 signal은 reg로 선언한다. 이벤트 리스트에는 clock과 reset 만을 사용한다. 논-블락킹 대입문("<=")을 사용한다. "<=" 의 특징은 begin ~ end 블록의 여러 문장들이 동시에 정말 동시에 수행된다는 것이다. (fpga 를 사용하는 이유아닐까)
reset의 동작은 가능한 negative edge를 사용한다. always문에서 사용하는 reg는 플립플롭이기 때문에 반드시 reset이 필요하다. 또한 always문 내의 시작은 항상 이 reset 동작을 수행하도록 한다. 주의 점으로는
하나의 Module에서 여러 개의 always문이 사용 가능하기 때문에 같은 signal(reg)을 서로 다른 always문에서 값을 할당하면 안된다.

3. Blocking and non-blocking

testbench 에서 비교해보도록 해보겠다. 그전에 testbench 사용법을 익혀보자.

3.1 `timescale문

`timescale 1ns/1ps (테스트벤치 스텝 단위/시뮬레이션 스텝 단위)
→ 테스트 벤치 상단에서 시간단위를 지정해준다.
→ 문장의 끝에 세미콜론이 붙지 않는다.
→ 테스트 벤치 스텝 단위는 시간 지연등을 사용할때의 단위이다.

예를 들어 1ns/1ps 로 선언을 하였을 경우 테스트 벤치에서 #30의 의미는 30ns를 의미한다.

시뮬레이션 스텝 단위는 시뮬레이션 후 timing diagram에서 보여지는 시간의 resolution을 결정한다.

3.2 initial문
→ 순차적으로 한번만 실행 시킬 때 사용된다. 즉 testbench말고 코드작성시 값 초기화 시킬때 많이 사용한다.

이제 블록킹 렛츠기릿

blocking 은 y = x 와 같은 형식이고, non-blocking은 y <= x 와 같은 형식으로 표현한다.
신호의 흐름을 'blocking' 한다 라고 생각하면 된다. 논블럭은 그 반대. 따라서 코드 내에서 blocking구문으로 사용할 경우 이는 c 언어처럼 순차 실행을 의미하게 된다.

```
initial begin
  #10 a = b;
  #5   c = a;
end
```

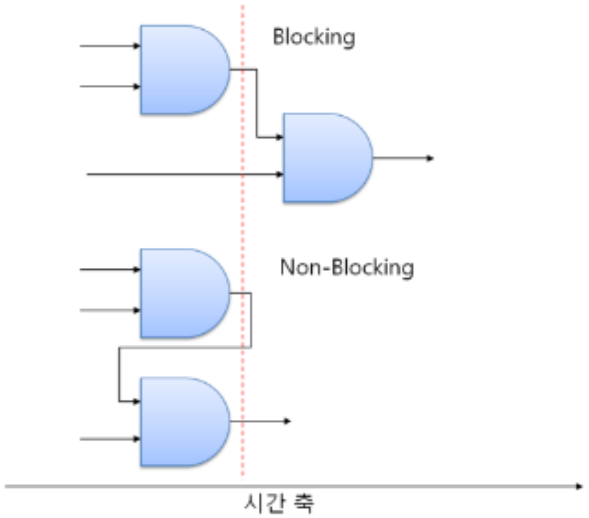
위와 같은 경우 처음 10 단위 시간 뒤에 b를 a에 넣고, 다시 5 단위 시간 뒤에 a를 c에 넣으므로 두번째 라인은 15 단위 시간 뒤에 동작하게 된다. (단위시간 ns 설정시)

<= 를 사용하는 non-blocking 같은 경우 말 그대로 신호의 흐름을 막지 않는 것이다.

```
initial begin
    a <= #10 b;
    c <= #5 a;
end
```

위와 같은 경우 처음 5단위 시간에서 c에 a값이 들어가고 10단위 시간에서 a에 b값이 들어가므로 순서와 상관없이 동시에 실행하게 된다.

그림으로 마무리하자면



위와 같이 blocking과 non-blocking을 설명할 수 있는 것이다.

#Tip " always구문 안에서는 가능하면 non-blocking 구문으로 작성할 것"
왜 그럴까? 경쟁조건(race condition) 때문이다.

코드로 설명하겠다.

보기1.

```
always@(posedge clock)
   a = b;
always@(posedge clock)
   b = a;
```

보기2.

```
always@(posedge clock)
 a <= b;
always@(posedge clock)
  b <= a;
```

위의 보기1과 보기2를 보면 보기1 blocking 구문부터 보면 **a = b 와 b = a의 구문이 clock 상승 에지에서 동시에 실행**되게 되는데 이것을 **경쟁 상태** 라고 한다. 이것이 어느것이 먼저 실행되느냐는 실제 게이트 레벨에서 어떻게 합성되느냐에 따라 선의 길이가 누가 더 긴가(?)와 같은 애매한 관계에 따라 달라지게 된다. 한마디로 예측이 어려운 상태가 되는 것이다. 이것을 **경쟁 조건(race condition)** 이라 한다.

하지만 보기2 에서 보면 둘은 동시에 실행되기 때문에 클럭이 발생했을 때 **a는 현재 b값**을 가지고 **b는 이전의 a값**을 가지고 있게 된다. 한마디로 서로 부딪칠 일이 없다.
그래서!!**왜 always 구문 안에서는 가능하면 non-blocking으로 설계해야 하는가?**바로 경쟁 조건이 발생하지 않기 위해서 이다. 저런 동시에 실행되는 많은 always 문이 생기게 되면 자신도 모르게 어디선가 경쟁조건이 발생하게 되고 시뮬레이션 상에서는 알 수 없지만 나중에 칩으로 갔을 때 제대로 동작 되지 않는 상태가 되는 것이다!!!!

(참조 : 책에는 non-blocking으로 파이프라인과 상호 배타적인 데이터 전송을 하지만 단점으로 시뮬레이터의 성능 하락과 메모리 사용량을 늘리는 것의 원인이라고 나와 있음)