Verilog - a "Hardware Description Language" (HDL)

=> originally invented in early 1980s for simulation of discrete component networks

Verilog => synthesis of verification & logic

Originally owned by Cadence, INC but made available in public domain ~1990

VHDL => originally from DOD military for ASIC (application specific integrated circuits) production

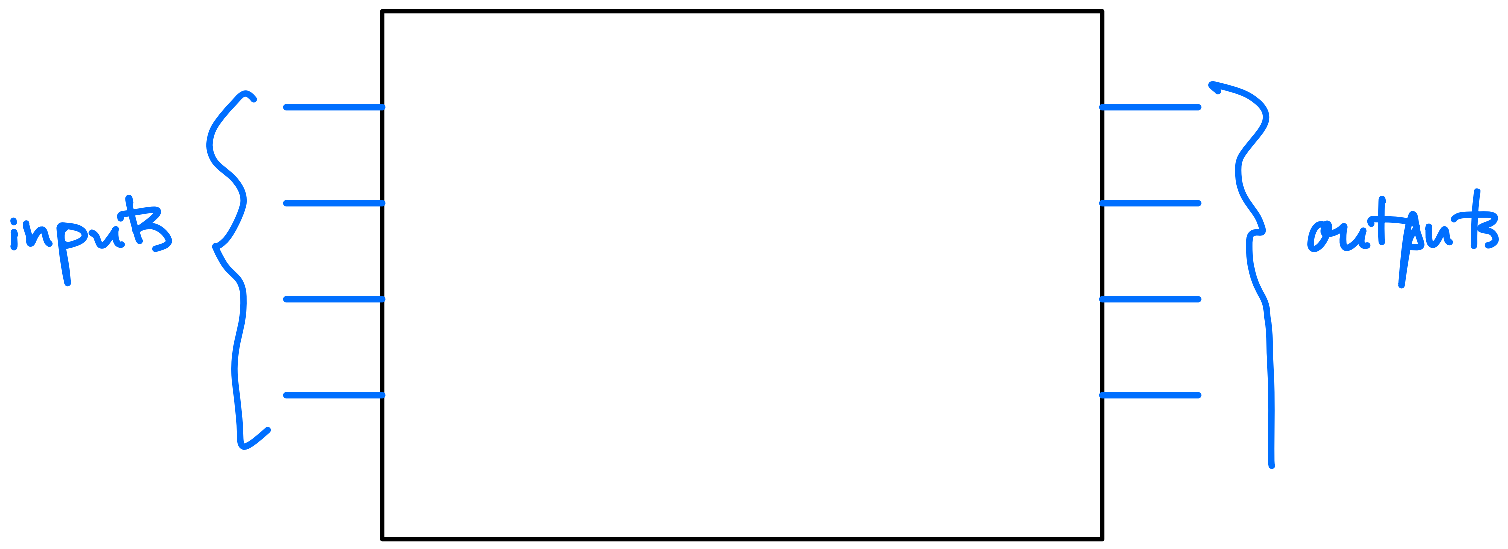
VHDL allows very specific details of everything down to hardware level

ex transistor rise times, etc

=> it is too detailed & Verilog is easier to learn

Verilog is used to describe circuits

For example, here's an FPGA chip with inputs & outputs



Inside you want to instantiate a circuit
⇒ this is your "Top level" circuit, or "module"

Verilog code:

arbitrary name

~all lines end with ;

```
module TOP(inputs, outputs);
   :
endmodule
```

Inputs: these are inputs FROM somewhere

Outputs: these drive signals TO somewhere

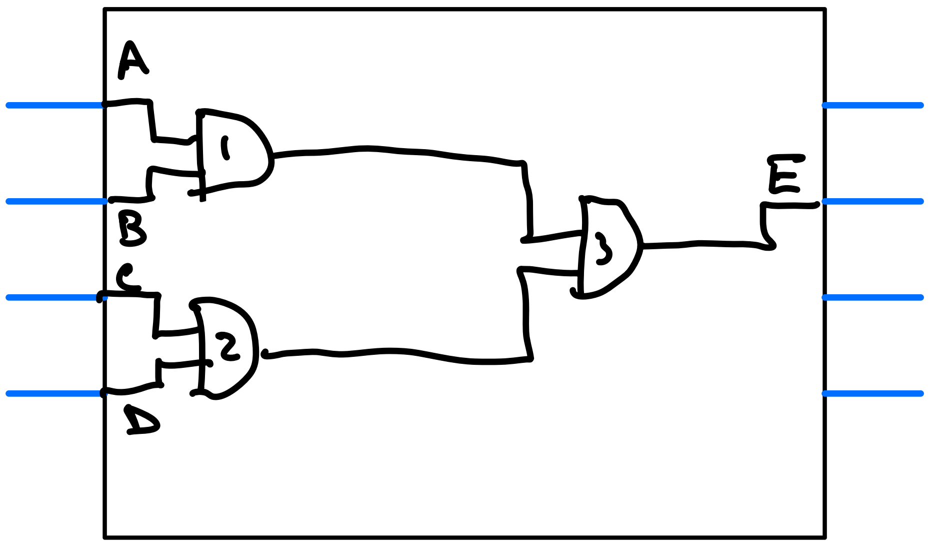
⇒ will discuss external to FPGA later

Specifying inputs & outputs:

old way: name the inputs A, B, C, D
and the output E

```
module TOP (A, B, C, D, E);
 input A, B, C, D;
 output E;
  :
 endmodule
```

lets code this circuit:



we have 3 AND gates

Verilog: use data type "WIRE" to specify connections → like a real wire that connects components on a circuit board

```
module TOP (A,B,C,D,E);
 input  A,B,C,D;
 output  E;
 wire  A,B,C,D;        or     wire  A;
                              wire  B;
                                 :
 wire  E;                     wire  D;
```

now we need a wire to connect gates ① & ② to ③

```
wire  C1;
wire  C2;
```

next, perform logic using operators

&	=	AND
|	=	OR
^	=	XOR
~	=	NOT

and use assign to perform logic

```
assign  C1 = A & B;
assign  C2 = C & D;
```

shortcut:

```
wire  C1 = A & B;
wire  C2 = C & D;
```
} implicit assign

or

```
wire  C1, C2;
C1 = A & B;
C2 = C & D;
```

both are ok: 1st method is used to keep declaration together, separated from assigments

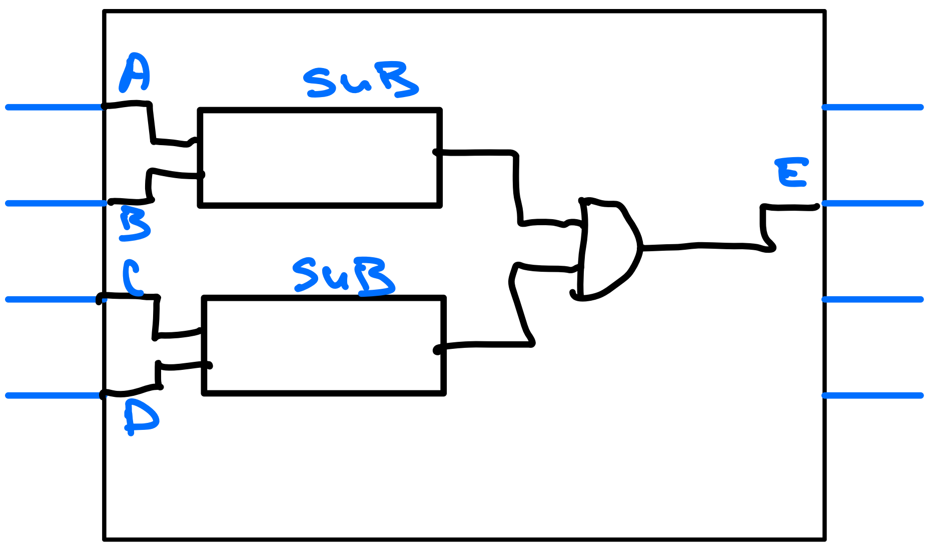
Then for output E:

```
assign  E = C1 & C2;
```

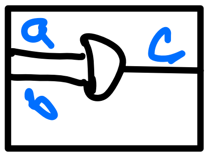
code:

```
module TOP (A, B, C, D, E);
 input  A, B, C, D;
 output  E;
 wire  A, B, C, D;
 wire  E;
wire c1, c2;
assign c1 = A & B;
assign c2 = C & D;
assign E = c1 & c2;
end module
```

Subcircuits: lets put the 1st 2 AND gates into a subcircuit



subcircuit name is SUB & there are 2 here



code is:

```
module SUB(a,b,c)
 input a,b;
 output c;
 assign c= a & b;
end module
```

now I place 2 copies of SUB inside TOP
and specify connections
⇒ have to specify a "local name"
for SUB inside TOP

```
module TOP(A,B,C,D,E);
 input A,B,C,D;
 output E;
 wire A,B,C,D;
 wire E;
 wire c1;
 SUB local1 (A,B,c1);
 wire c2;
 SUB local2 (C,D,c2);
 assign E = c1 & c2;
endmodule
```

follows order of arguments in SUB

completely arbitrary names

variation on instantiating SUBs:

```
SUB local1 (.a(A), .b(B), .c(c1));
```

$$\begin{tikzcd}
\text{module arg name} \arrow[u] \arrow[ur] \arrow[urr]
\end{tikzcd}$$

this says connect A to a, B to b, etc.

⟹ no longer depends on order!

note: assign is mostly just for simulation

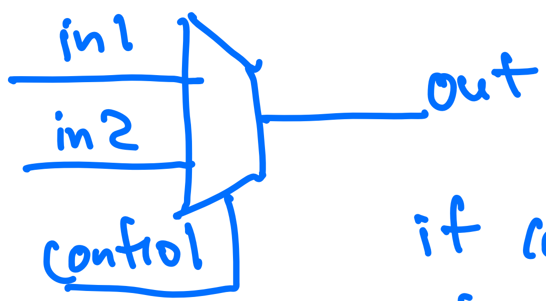
assign A = B & C;

means if B or C changes, then change A

for making FPGA circuits it's often not necessary

exception: for <u>driving</u> an output

<u>Verilog mux</u>



if control = 0, out = in1
else out = in2

in verilog, use a "conditional"

```
wire out;
assign out = control ? in1 : in2;
```

$$\texttt{assign out = control ? }\underbrace{\texttt{in1}}_{1}\texttt{ : }\underbrace{\texttt{in2}}_{0}\texttt{;}$$

Verilog bus

bus ⇒ collection of wires

```
wire [1:0] N;
```

MSB ↑ ↑ LSB

then there are 2 wires: N[0] is LSB
N[1] is MSB

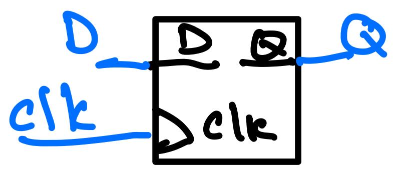
Verilog register

mostly used as output of DFF or latch,
⇒ holds data

```
reg A;
```

can think of a reg as a DFF or a "driver" that drives current
(but mostly for DFFs)

Verilog flip-flops



Q always follows D at posedge of the clock clk

```
wire D;
reg Q;
always @(posedge clk) Q = D;
```

if we have 2 DFF's, can use this syntax:

```
wire D1, D2;
reg Q1, Q2;
always @(posedge clk) Q1= D1;
always @(posedge clk) Q2= D2;
```

or

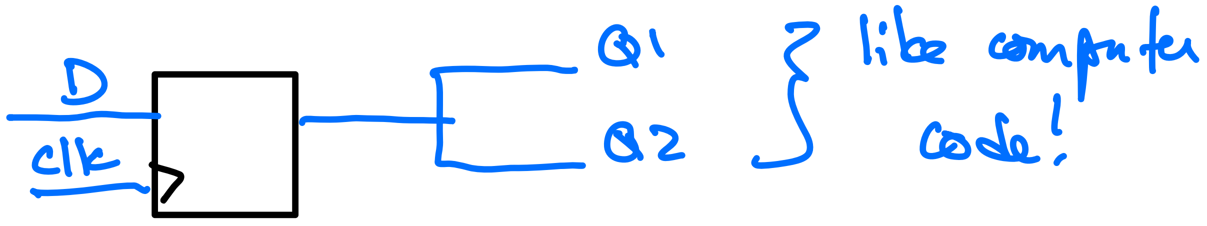
```
wire D1, D2;
reg Q1, Q2;
always @ posedge (clk) begin
        Q1= D1;
        Q2= D2;
end
```

matched!

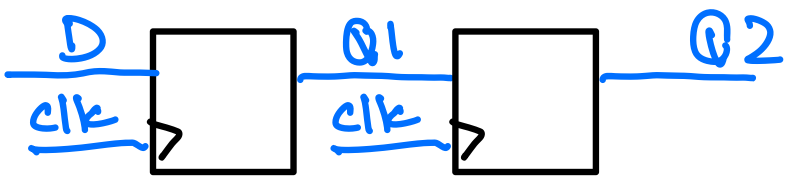
problem: take following

```
wire D;
reg Q1, Q2;
always @(posedge clk) begin
        Q1 = D;
        Q2 = Q1;
end
```

ambiguous!



or



what we probably want is:

1. 1st edge, Q1 = D
2. 2nd edge, Q2 = Q1

using
```
Q1 = D;
Q2 = Q1;
```
executing in series or parallel?

BLOCKING : $Q1 = D;$ evaluate
$Q2 = Q1;$ continuously

NON-BLOCKING $Q1 \Leftarrow D;$ evaluate 1st,
$Q2 \Leftarrow Q1;$ then assign all at once

ex:

```
reg A,B,C;
always @ (posedge clk) begin
        A = 1;
        B = A;
        C = B;
end
```

if this was a computer code, then things would happen sequentially
=> after C = B; they would all be = 1

so each statement "blocks" the next one from occuring until executed

In FPGA we want things to happen in parallel!

so after 1st clock: A=1

B=?

C=?

2nd " A=1

B=1

C=?

3rd " A=1

B=1

C=1

each statement is evaluated and assigned simultaneously, and are non-blocking

For programmable logic, need to distinguish:

BLOCKING Assignment: use =

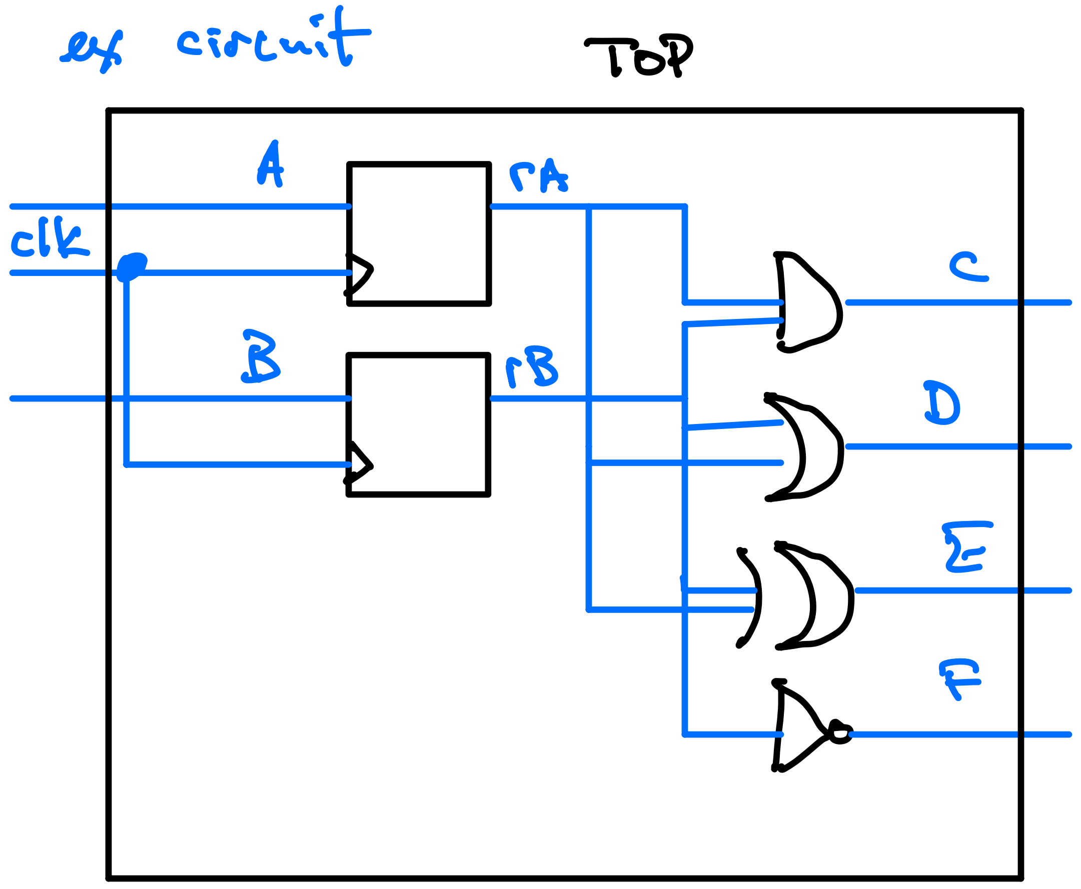
alway for combinatorial logic

NON-BLOCKING Assignment: use <=

always for sequential login

note: for DFF inside always, use <=

ex circuit



```
module TOP (
 // declare in/out
   input A, B, clk,
   output C, D, E, F
  );
 reg rA, rB;
 always @(posedge clk) begin
     rA <= A;
     rB <= B;
 end
 /*
   combinatorial logic
```

```
*/
assign C = rA & rB;
assign D = rA | rB;
assign E = rA ^ rB;
assign F = ~ rA;
endmodule
```

} optional assign (C, D, E)

} not optional, drives output (F)

Conditionals

a wire that determines an output depending on if it is 0 or 1

ex: x is a wire (binary)
if x is 0, y=2, else y=5

```
wire x;
wire [2:0] y;      (need 3 bits to hold values)
if (x) y = 5;      (or y = {1,0,1})
else y = 2;        (or y = {0,1,0}
```

alternative:

$$y = x \; ? \; \underset{\text{true}}{5} : \underset{\text{false}}{2};$$

Math

verilog synthesizer can implement math !

ex:
```
wire [1:0] a, b;
wire [2:0] c;
c = a * b;
```
synthesizer will build the logic to implement "*"

⇒ will also do this for +, -, /