Verilog is a *Hardware* Description Language (HDL)

- You'll design far better hardware if you think of it differently than a standard programming language
- A "standard programming language" such as C, C++, python, java, etc.:
  - Is a way to code an *algorithm* or *is a way to calculate a result*
  - Is written by a *software writer*
  - Often results in a more elegant solution when the programmer uses finer features of the language
- On the other hand, a hardware description language:
  - Is a way to describe *hardware*
  - Is written by a *hardware designer*
  - Results in a far better solution when the designer uses only the most basic features of the language

© B. Baas

Verilog vs. VHDL

- Verilog
  - Invented in 1983 at Automated Integrated Design Systems (later Gateway Design Automation) which was purchased by Cadence in 1990. It was transferred into the public domain in 1990 and it became IEEE Std. 1364-1995, or *Verilog-95.*
  - Later versions include
    - *Verilog-2001* aka *IEEE 1364-2001*
      - Added **`signed`** (2's complement) arithmetic support
      - Added support for combinational **`always @(*)`**
    - *Verilog-2005* aka *IEEE 1364-2005*
  - Strong similarities to C
  - Seems to be more commonly used in high-tech companies

Verilog vs. VHDL

- VHDL (VHSIC Hardware Description Language)
  - Published in 1987 with Dept. of Defense support as IEEE Std. 1076-1987. Updated in 1993 as IEEE Std. 1076-1993, which is still the most widely-used version.
  - Later versions in 2000, 2002, and 2008.
  - Strong similarities to Ada
  - The only(?) HDL language used in government and defense organizations, and seems to be more often used in east-coast companies. Widely taught in universities ↔ used in textbooks—who started it?!

© B. Baas

* 3 Ways to Specify Hardware *

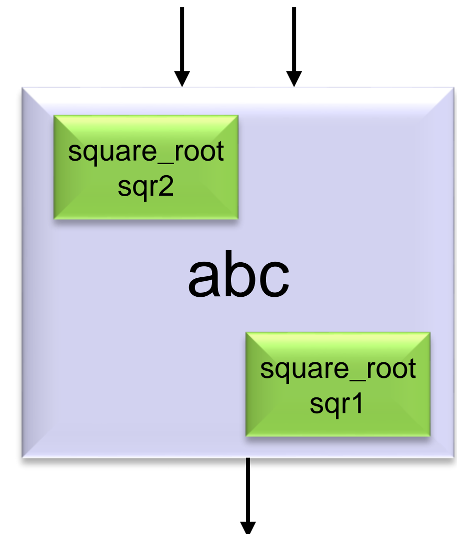
- There are three primary means to specify hardware circuits:
  1) Instantiate another module
  2) *wire* declared with an *assign* statement
  3) *reg* declared with an *always* block
- Example instantiating modules inside a main module

```
module abc (in1, in2, out);
   input  in1;
   input  in2;
   output out;

   assign...

   always...

   always...

   square_root sqr1 (clk, reset, in1, out1);

   square_root sqr2 (clk, reset, in2, out2);

endmodule
```

module name

instance names

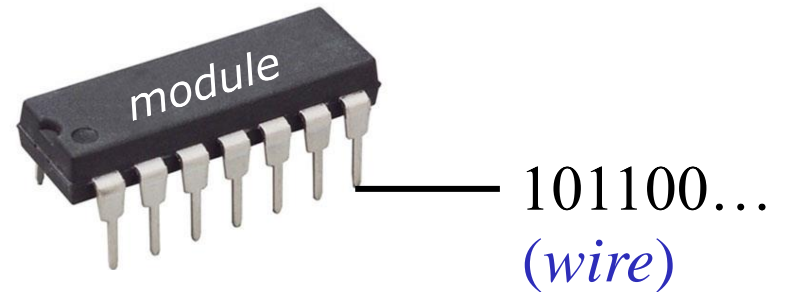
module name

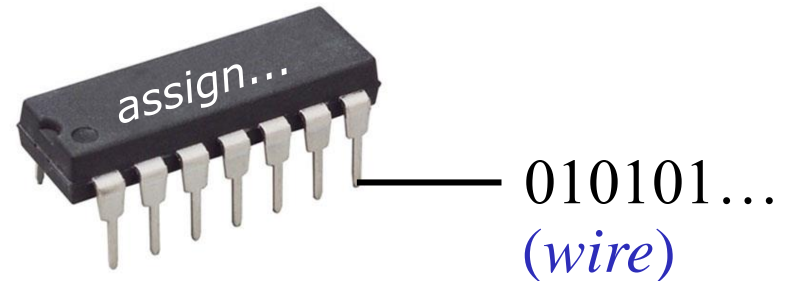


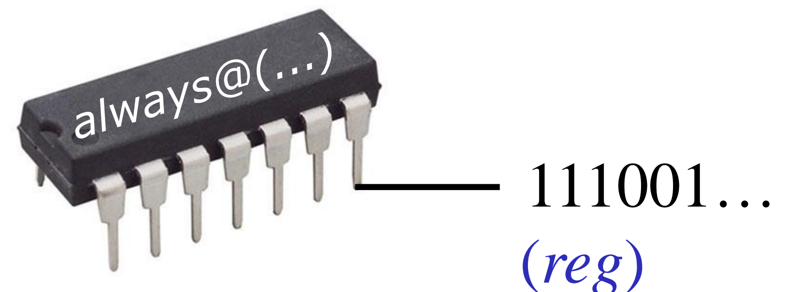
© B. Baas

Concurrency

- All circuits operate independently and concurrently
  - Different from most programming paradigms
- This is natural if we remember "hardware verilog" describes real circuit hardware: transistors and wires



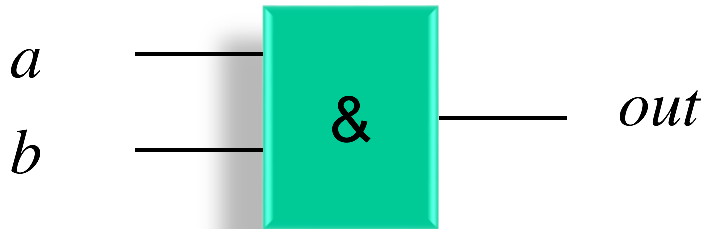


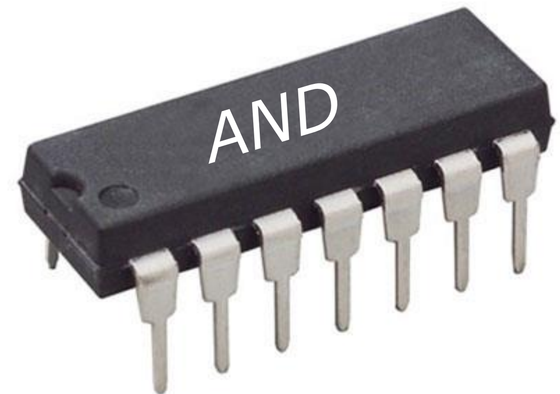


2) *wire, assign*

- Example:

```
wire out;
assign out = a & b;
```





© B. Baas

3) *reg, always*

- Picture a much more general way of assigning "wires" or "signals" or "buses"
- "if/then/else" and "case" statements are permitted
- You could, but don't use "for loops" in hardware blocks (use in testing blocks is ok)
- Sequential execution
  - *statements* execute in order to specify a *circuit*
- Syntax:

```
always @(sensitivity list) begin
    statements
end
```

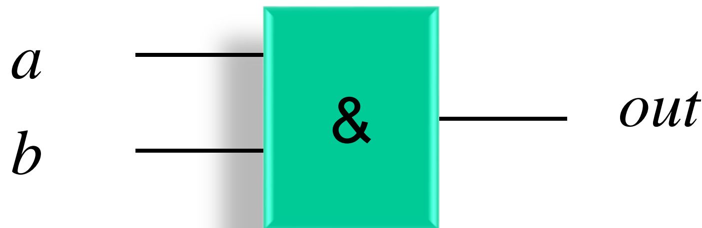
- Operation:
  statements are executed when any signal in *sensitivity list* changes

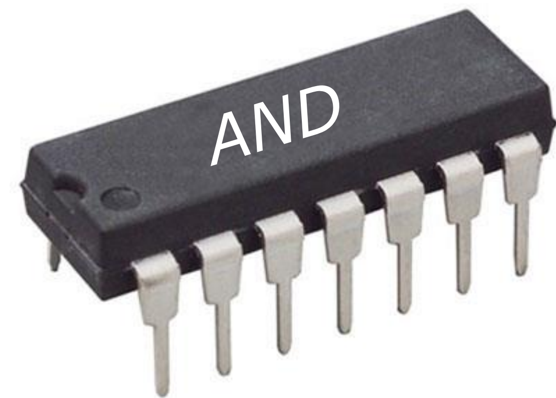
© B. Baas

3) *reg, always*

- Example: there is **no** difference whatsoever in this AND gate from the AND gate built using *assign*

```
reg out;
always @(a or b) begin
   out = a & b;
end
```



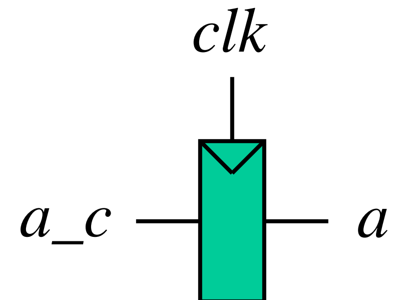


© B. Baas

Instantiating Flip-Flops/Registers

- One way to build a FF/register (do *not* use this)

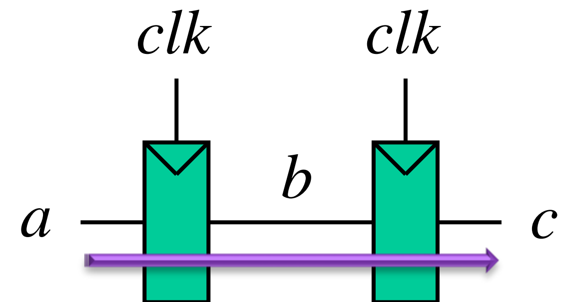
```
reg a;
always @(posedge clk) begin
  a = a_c;
end
```



- The "=" is a "blocking assignment" which causes the simulator to "block" on an assignment until the operation is completed, then it moves to the next statement
- It makes a race condition possible

```
reg b, c;
always @(posedge clk) begin
  b = a;
  c = b;
end
```

  – In this case, *a* races to *c* in one cycle!

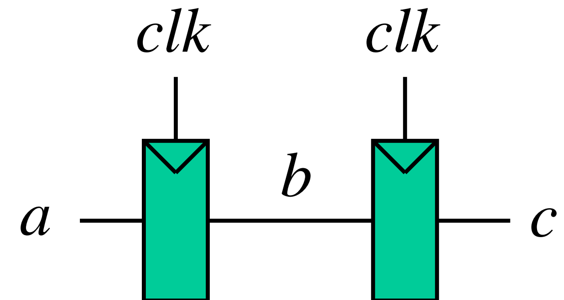


Instantiating Flip-Flops/Registers

- The correct solution is to use a "non-blocking assignment" written with "**<=**" which causes the simulator to evaluate the right side of the expression when the statement is encountered, but the assignment of the left side is not done until the end of that time step in "simulator time"
- With the verilog below, the registers perform as normal FFs behave without a race regardless of their ordering

```
reg b, c;
always @(posedge clk) begin
  b <= a;
  c <= b;
end
```

The key differentiator is the time when the value *b* is sampled. Here it is not "blocked" by the previous assignment **`b<=a`**



© B. Baas