03 Intro to CAD tools and Verilog HDL

- CAD tools
- Design Entry
- Simulation
- Synthesis
- Hardware description language (HDL)
- Verilog code examples
- Design flow recap

CAD tools:

CAD stands for Computer-Aided Design

Why CAD?

- 100-100,000,000 times quicker to design
- Easier to test and debug
- Easier to turn design into real circuits
- Easier to re-use old designs
- Some tools are standardized
    - Learn once, work on many projects

Design entry:

How we tell CAD tools what design we are doing

2 popular ways:

1. Schematic capture: draw gates/wires
   - old method - can be very tough to debug
   - no standard → switch tools, lose all
   + good for “big picture” view
2. Hardware description language (HDL)

Specialized programming language to describe hardware operation

- some learning curve
+ much faster to design and/or debug
+ STANDARDIZED

Simulation, synthesis, and manufacturing:

Simulation:

- Test run on software to catch design errors (design errors are called bugs)
- Very critical for any design
- Verification (Testing) is mostly done here

Synthesis:

- Turn design into circuits

Manufacturing:

- Turn prototype into products

Hardware description language (HDL):

- A bit like “software” – C or Pascal
- Specially designed to describe hardware
- 2 popular choices

1. Verilog ← We will use this
   - popular in profitable ☺ companies in US
   - easier to learn and write than VHDL
   - has some similarity to C
2. VHDL (Very Hard to Design and Learn)
   - developed by US government
   - has some similarity to Ada / Pascal

Verilog HDL:

- Design consists of modules
- A module describes the relationship between its inputs and outputs

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

port = input or output
→interface to outside world

Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

port = input or output
→interface to outside world

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

module name

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

module name

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

foo

Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

port list

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

foo

Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

port list

port declaration

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

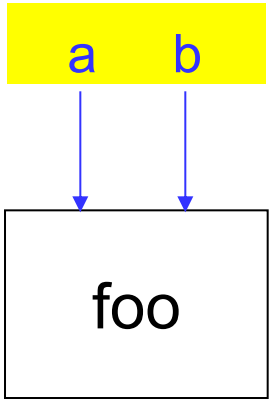
foo

Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

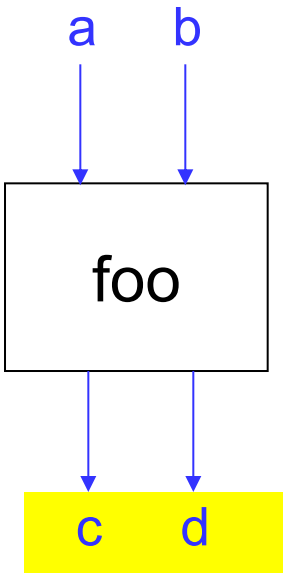


Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```

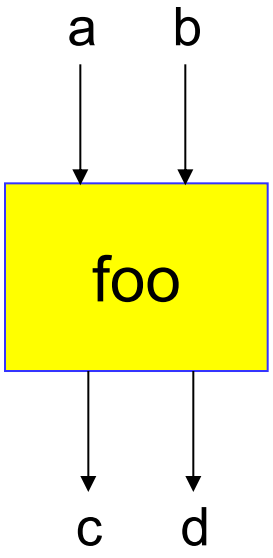


Verilog HDL:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

```
module foo (a, b, c, d);
    input a, b;
    output c, d;

    // some design here

endmodule
```



Verilog HDL:

Some design inside 1: gate primitives

and (out, in1, in2);	2-input AND gate
or (out, in1, in2, in3);	3-input OR gate
not (out, in);	inverter

Some design inside 2: equations

assign c = a & b;	& is AND
assign d = e | f;	| is OR
assign g = ~h;	~ is NOT
assign j = (i & k) | (~m & p);	parentheses possible

Verilog HDL:

Some design inside 1: gate primitives

and (out, in1, in2);	2-input AND gate
or (out, in1, in2, in3);	3-input OR gate
not (out, in);	inverter

Some design inside 2: equations (also called assignments)

assign c = a & b;	& is AND
assign d = e | f;	| is OR
assign g = ~h;	~ is NOT
assign j = (i & k) | (~m & p);	parentheses possible

will tell you about this soon

More Verilog vocabulary:

Lecture:	x y + z
Verilog:	(x & y) | z
Lecture:	z’
Verilog:	~z
Lecture:	(xy)’
Verilog:	~(x & y)
Lecture:	x ⊕ y
Verilog:	x ^ y

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

All above are bitwise.
Use ( ) to group which to do first

Verilog supports && (logical AND), || (logical OR), and ! (logical NOT) but be careful using them. (not recommended)

Verilog HDL:

Some design inside 3: procedural statements

```
always@ (s or a or b)
    if (s == 1)
        f = b;
    else
        f = a;
```

VERY POWERFUL WAY

WILL USE THIS FOR MOST OF THE CLASS

Verilog HDL:

Some design inside 3: procedural statements

```
always@ (s or a or b)
    if (s == 1)
        f = b;
    else
        f = a;
```

will tell you about this soon

Verilog HDL:

Some design inside 3: procedural statements

```
always@ (s or a or b)
    if (s == 1)
        f = b;
    else
        f = a;
```

Some design inside 4: other modules

```
module many_cpus (...);
    ...
    cpu first_core(core1_input, core1_output);
    cpu second_core(core2_input, core2_output);
```

Verilog HDL:

Some design inside 3: procedural statements

```
always@ (s or a or b)
    if (s == 1)
        f = b;
    else
        f = a;
```

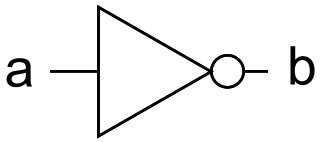
Some design inside 4: other modules

```
module many_cpus (...);
    ...
    cpu first_core(core1_input, core1_output);
    cpu second_core(core2_input, core2_output);
```

will tell you about this later

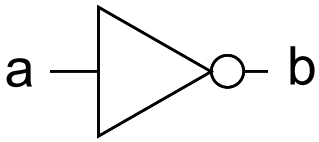
Verilog HDL:

First Example



Verilog HDL:

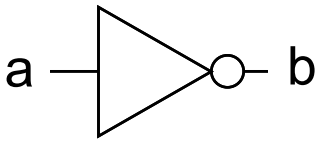
First Example




```
module my_not
```

Verilog HDL:

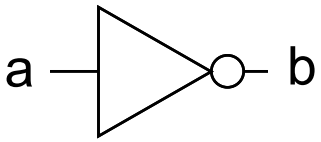
First Example




```
module my_not (a, b);
      input a;
      output b;
```

Verilog HDL:

First Example

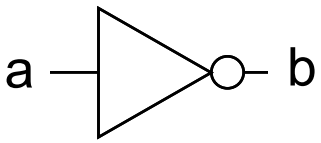


DESIGN STYLE 1: GATE PRIMITIVE

```
module my_not (a, b);
      input a;
      output b;

      not(b, a);
```

Verilog HDL:

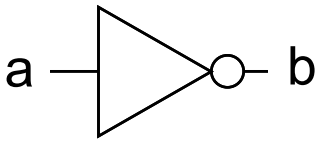
First Example




```
module my_not (a, b);
      input a;
      output b;

      not(b, a);

endmodule
```

Verilog HDL:

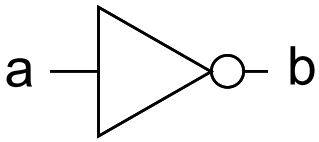
First Example




```
module my_not (a, b);
      input a;
      output b;
```

Verilog HDL:

First Example

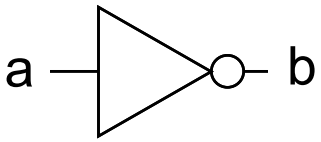


DESIGN STYLE 2: ASSIGNMENTS

```
module my_not (a, b);
      input a;
      output b;

      assign b = ~a;
```

Verilog HDL:

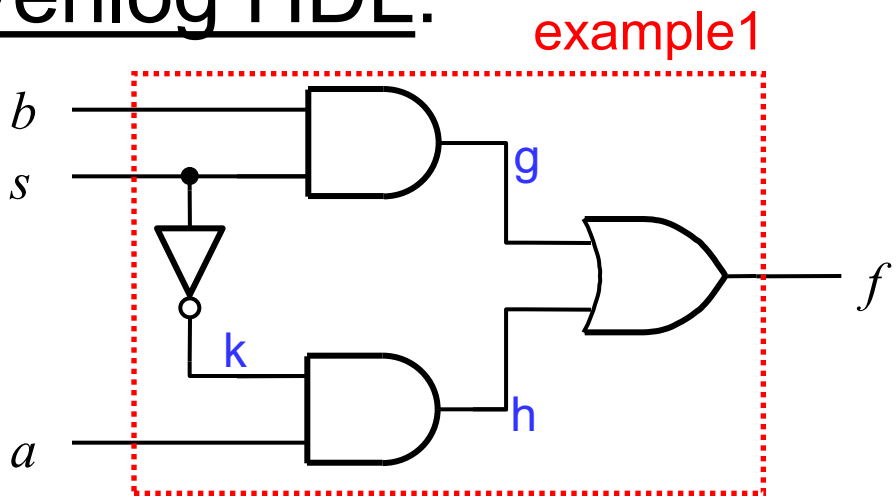
First Example




```
module my_not (a, b);
      input a;
      output b;

      assign b = ~a;

endmodule
```

Verilog HDL:

[BV]

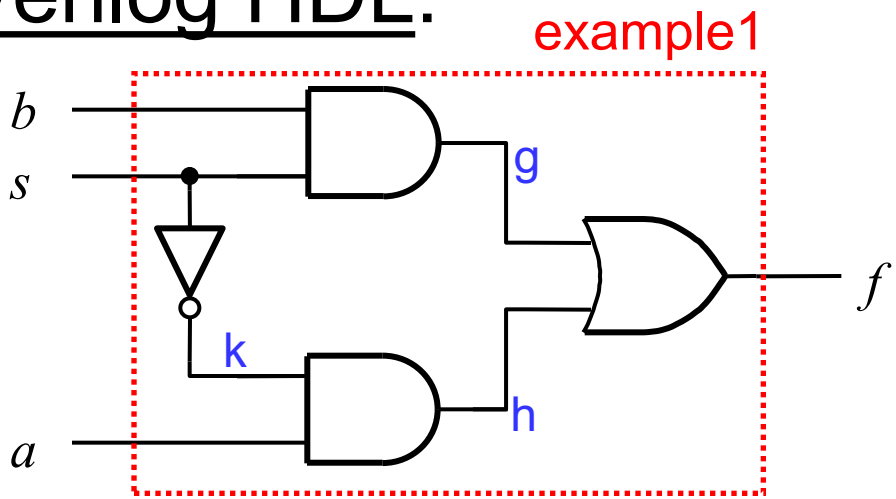


HANDS ON!

Let’s try to write module name and ports

Verilog HDL:

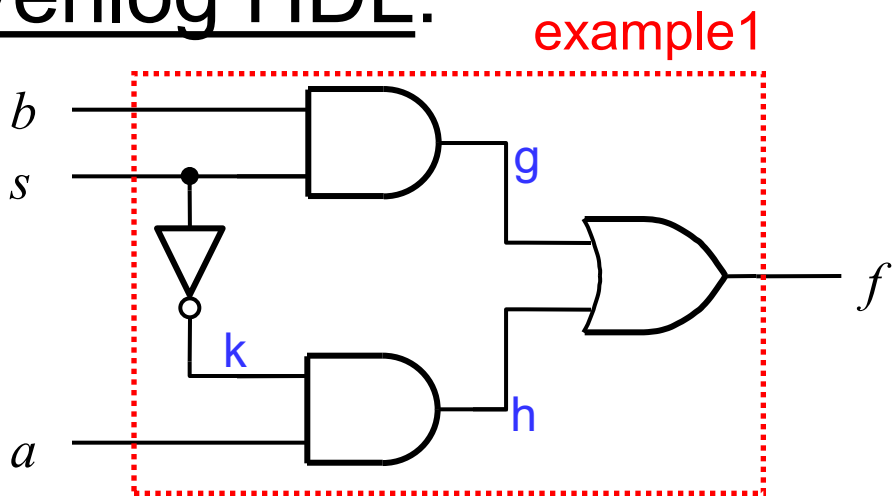
[BV]




```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
```

Verilog HDL:

[BV]



module name

port list

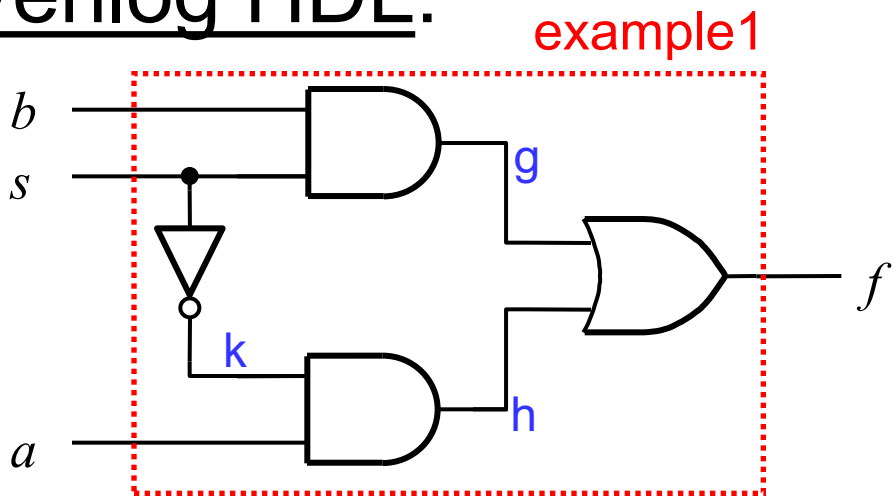
```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
```

port declarations

THERE ARE 3 INTERNAL WIRES (g, h, k) INSIDE
USE “wire” KEYWORD to DECLARE THEM

Verilog HDL:

[BV]

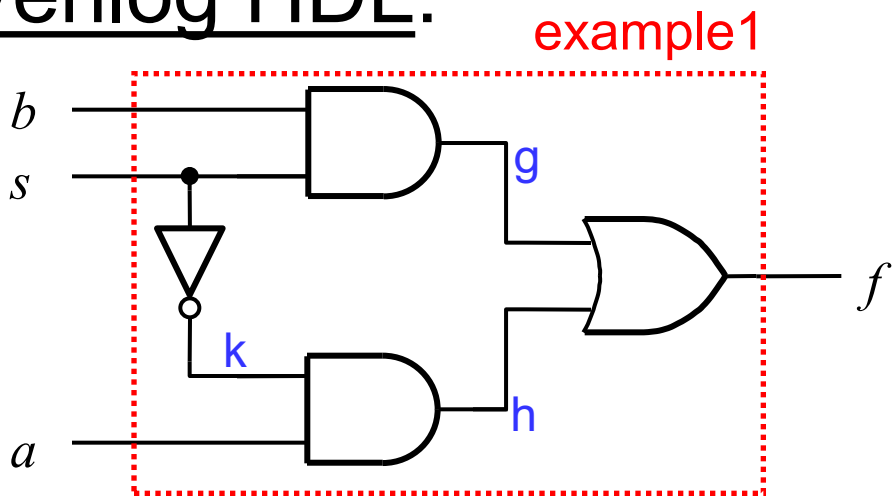



```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;
```

inside design, we can now use g, h, k

Verilog HDL:

[BV]

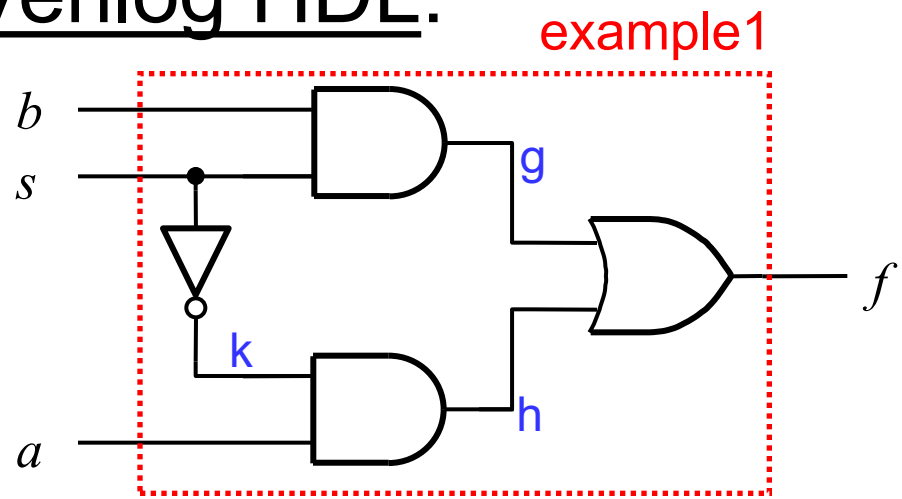



```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;
        and (g, b, s);
        not (k, s);
        and (h, k, a);
        or (f, g, h);

endmodule
```

design

Verilog HDL:

[BV]



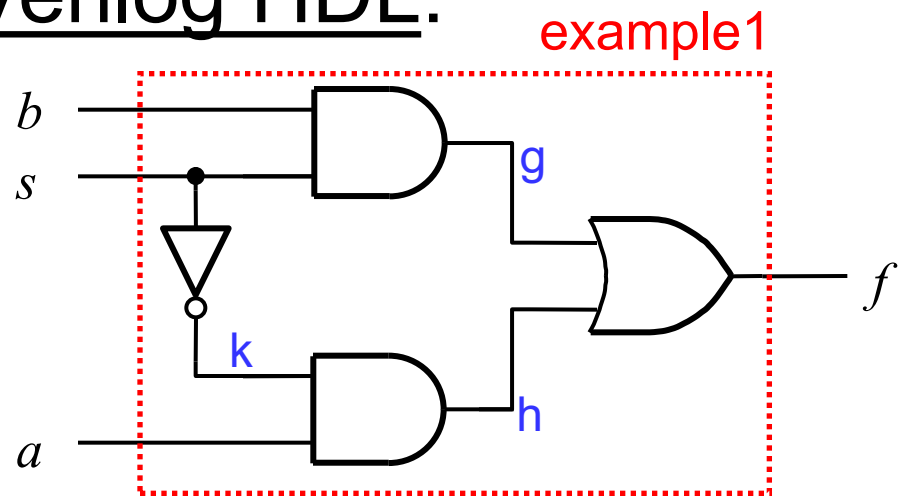

```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;
        assign g = s & b;
        assign k = ~s;
        assign h = k & a;
        assign f = g | h;

endmodule
```

design

OR WE CAN USE ASSIGNMENTS

Verilog HDL:

[BV]

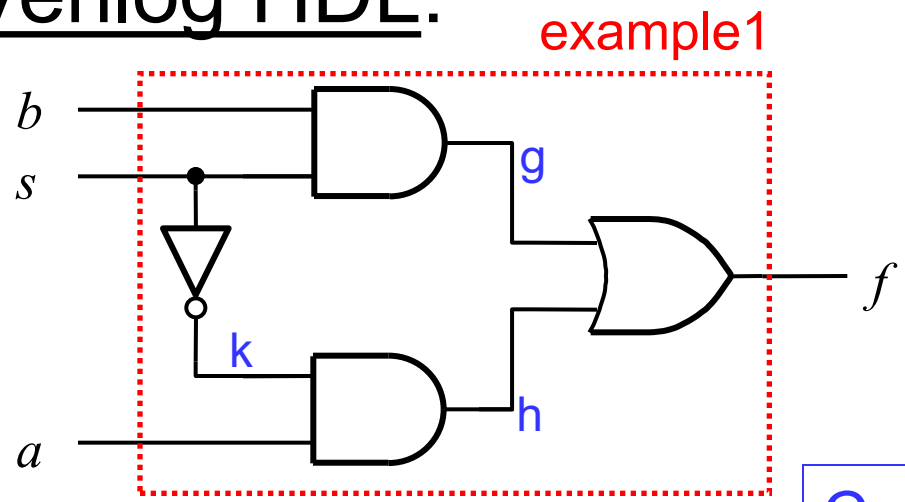



```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;
        assign g = s & b;   1
        assign k = ~s;      2
        assign h = k & a;   3
        assign f = g | h;   4

endmodule
```

QUESTION:
Can we swap 1 2 3 4 into different orders?

Verilog HDL:

[BV]



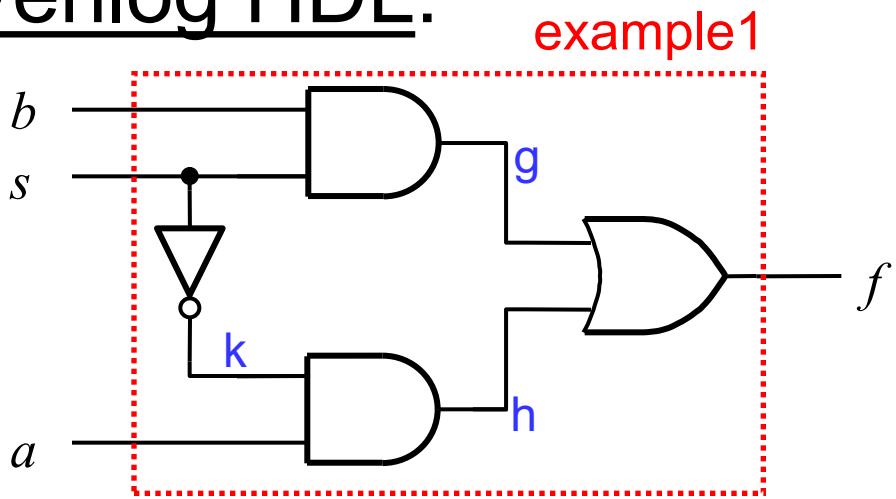

```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;
        assign g = s & b;   1
        assign k = ~s;      2
        assign h = k & a;   3
        assign f = g | h;   4

endmodule
```

Can we swap 1, 2, 3, 4?
YES

- 1 2 3 4 are parallel (orders do not matter)
- any time RHS changes, LHS will be re-evaluated
- This matches real hardware behavior

Verilog HDL:

[BV]

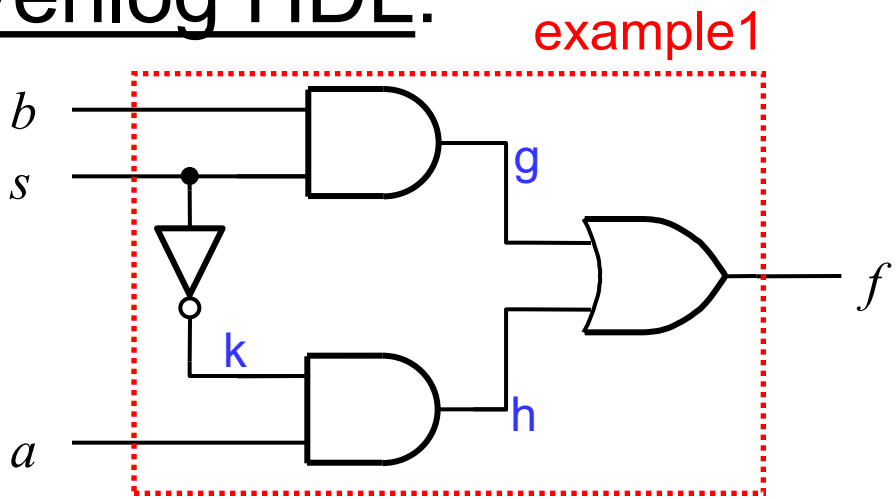


you notice that
f = (sb) + (s'a)

```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;
```

Verilog HDL:

[BV]

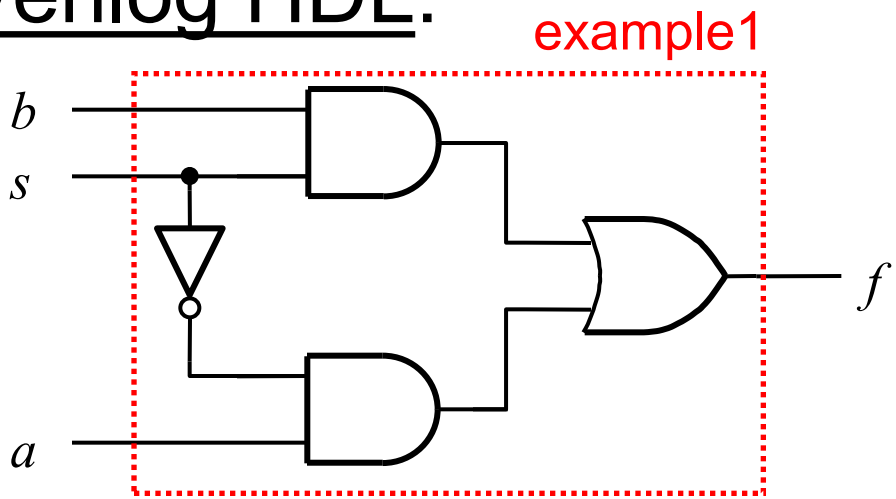


you notice that
f = (sb) + (s'a)

```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;

        assign f = (b & s) | (~s & a);

endmodule
```

Verilog HDL:

[BV]

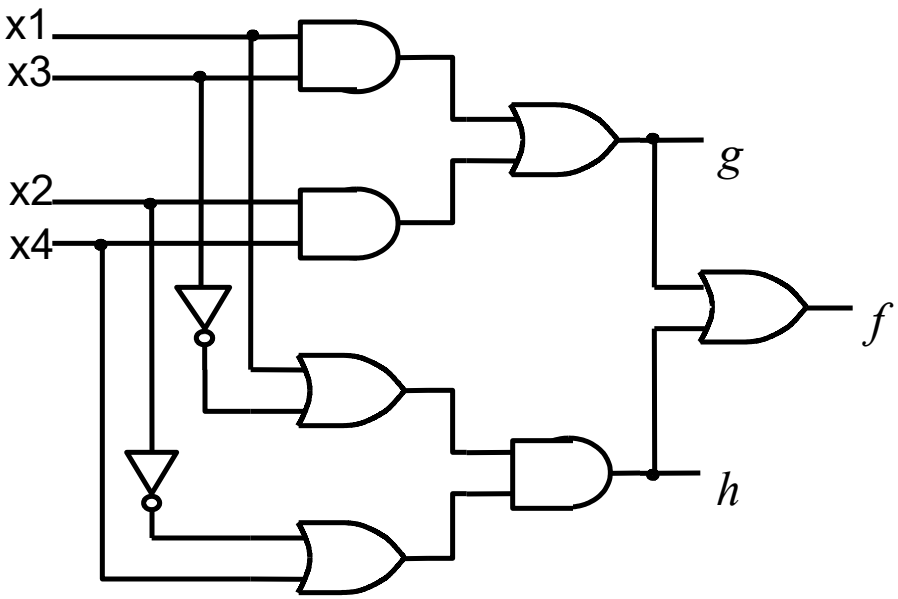



```
module example1 (s, a, b, f);
        input s, a, b;
        output f;
        wire g, h, k;

        assign f = (b & s) | (~s & a);

endmodule
```

no longer need g, h, k, because we write f from s, a, b directly

Verilog Example 2:

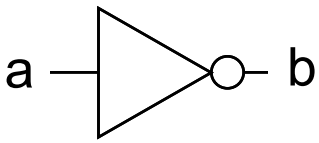
[BV]




```
module example2 (x1, x2, x3, x4, f, g, h);
        input x1, x2, x3, x4;
        output f, g, h;

        assign g = (x1 & x3) | (x2 & x4);
        assign h = (x1 | ~x3) & (~x2 | x4);
        assign f = g | h;

endmodule
```

Verilog HDL:

First Example

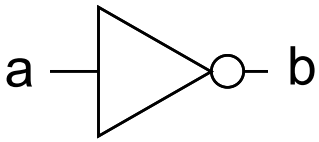



```
module my_not (a, b);
      input a;
      output b;
```

BACK TO OUR VERY SIMPLE EXAMPLE

Verilog HDL:

First Example

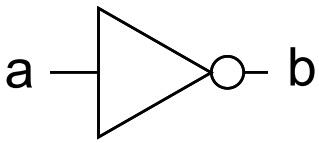



```
module my_not (a, b);
      input a;
      output b;

      not(b, a);

endmodule
```

DESIGN STYLE 1
GATE PRIMITIVE

Verilog HDL:

First Example

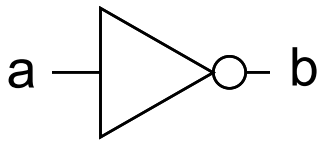



```
module my_not (a, b);
      input a;
      output b;

      assign b = ~a;

endmodule
```

DESIGN STYLE 2
ASSIGNMENTS

Verilog HDL:

First Example




```
module my_not (a, b);
      input a;
      output b;
```

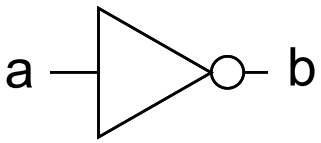
WE WILL DO PROCEDURAL DESCRIPTION

procedural description

- human readable and understandable
- computer programming like language

Verilog HDL:

First Example



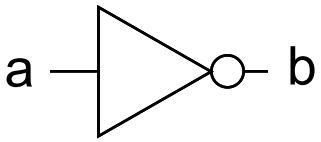

```
module my_not (a, b);
      input a;
      output b;
```

PROCEDURAL DESCRIPTION OF “NOT”

b is the inverse of a.

Verilog HDL:

First Example



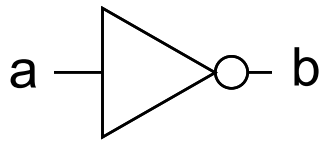

```
module my_not (a, b);
      input a;
      output b;
```

PROCEDURAL DESCRIPTION OF “NOT”

Do the following every time “a” changes:
b is the inverse of a.

Verilog HDL:

First Example




```
module my_not (a, b);
      input a;
      output b;
```

PROCEDURAL DESCRIPTION OF "NOT"

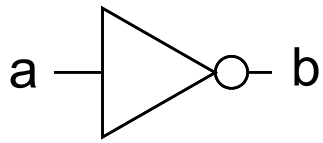
Do the following every time "a" changes:

b is the inverse of a.

This description is said to be sensitive to a

Verilog HDL:

First Example




```
module my_not (a, b);
      input a;
      output b;
```

PROCEDURAL DESCRIPTION OF “NOT”

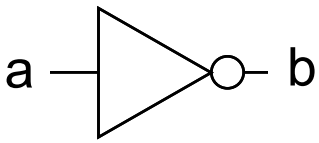
Do the following every time “a” changes:

b is the inverse of a.

This description is **always** sensitive to its input

Verilog HDL:

First Example

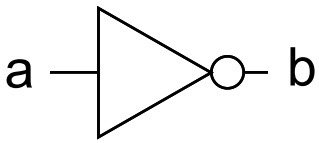



```
module my_not (a, b);
      input a;
      output b;


      always@(a)
```

every time “a” changes:
b is the inverse of a.

Verilog HDL:

First Example

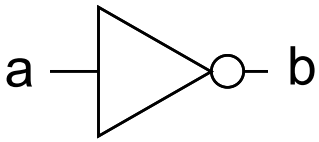



```
module my_not (a, b);
      input a;
      output b;


      always@(a)
            b = ~a;
```

every time "a" changes:
b is the inverse of a.

Verilog HDL:

First Example

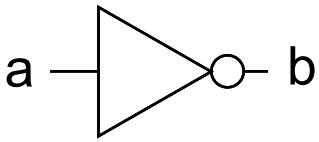



```
module my_not (a, b);
      input a;
      output b;

      always@(a)
            b = ~a;
```

This is called
an always block
or a procedural description

Verilog HDL:

First Example



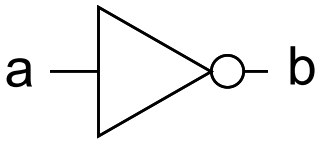

```
module my_not (a, b);
      input a;
      output b;


      always@(a)
            b = ~a;
```

The stuff inside parentheses is called the sensitivity list of the always block

This always block is sensitive to “a” and produces output “b”

Verilog HDL:

First Example




```
module my_not (a, b);
      input a;
      output b;

      reg b;
      always@(a)
            b = ~a;
```

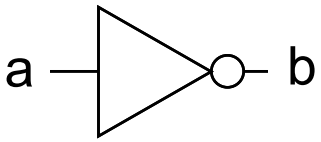
One more thing:

Every output of the always block must be declared as a **reg** before it is valid.

Why? Answer later on.

Verilog HDL:

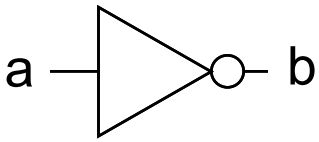
First Example




```
module my_not (a, b);
      input a;
      output b;

      reg b;
      always@(a)
          b = ~a;

endmodule
```

Verilog HDL:

First Example

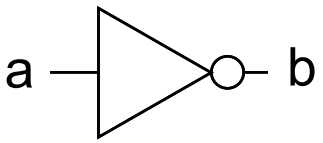



```
module my_not (a, b);
      input a;
      output b;

      reg b;
      always@(a)
            b = ~a;

endmodule
```

much more complicated than
assign b = ~a;
Why do we do it?

Verilog HDL:

First Example




```
module my_not (a, b);
        input a;
        output b;
```

PROCEDURAL DESCRIPTION OF “NOT”

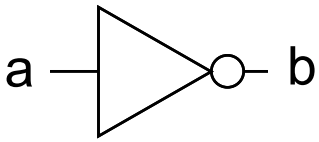
Do the following every time “a” changes:

if a is zero, then b is one

else, b is zero.

Verilog HDL:

First Example

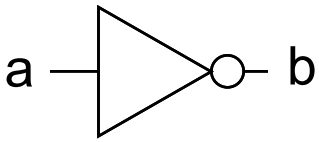



```
module my_not (a, b);
        input a;
        output b;


        always@(a)
```

every time “a” changes:
if a is zero, then b is one
else, b is zero.

Verilog HDL:

First Example

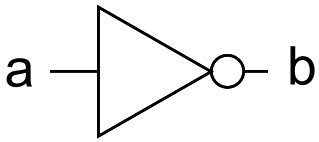



```
module my_not (a, b);
        input a;
        output b;


        always@(a)
          if (a == 0)
            b = 1;
          else
            b = 0;
```

every time “a” changes:
if a is zero, then b is one
else, b is zero.

Verilog HDL:

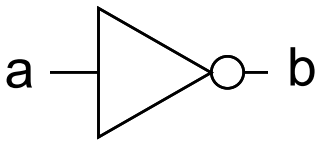
First Example




```
module my_not (a, b);
      input a;
      output b;

      reg b;
      always@(a)
        if (a == 0)
          b = 1;
        else
          b = 0;

endmodule
```

Verilog HDL:

First Example

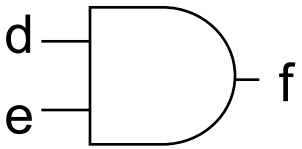



```
module my_not (a, b);
        input a;
        output b;

        reg b;
        always@(a)
          if (a == 0)
            b = 1;
          else
            b = 0;

endmodule
```

always block allows us to write human readable description.

Verilog HDL:

Another Example

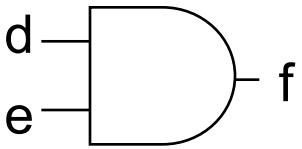



```
module my_and
```

HANDS ON
WRITE PORT LIST AND PORT DECLARATIONS

Verilog HDL:

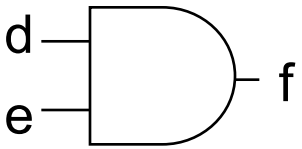
Another Example




```
module my_and (d, e, f);
      input d, e;
      output f;
```

Verilog HDL:

Another Example



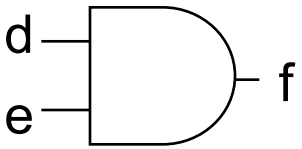

```
module my_and (d, e, f);
      input d, e;
      output f;
```

Behavioral description:

Every time d or e changes:
f is d AND e

Verilog HDL:

Another Example

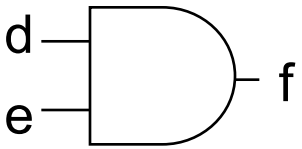



```
module my_and (d, e, f);
      input d, e;
      output f;


      always @(d or e)
```

Every time d or e changes:
f is d AND e

Verilog HDL:

Another Example

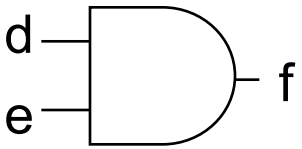



```
module my_and (d, e, f);
      input d, e;
      output f;


      always @(d or e)
            f = d & e;
```

Every time d or e changes:
f is d AND e

Verilog HDL:

Another Example



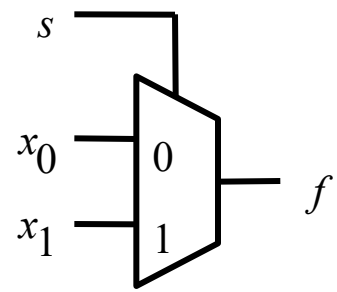

```
module my_and (d, e, f);
      input d, e;
      output f;

      reg f;
      always @(d or e)
          f = d & e;
endmodule
```

f is an output of always block

Verilog Example 3 (back to 2:1 mux):

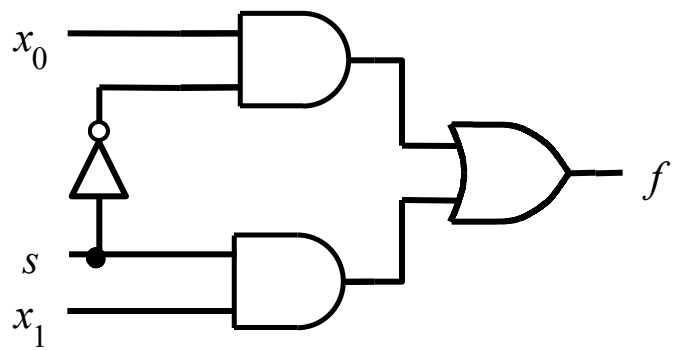
[BV]

Description:
When s=0, f = x0
When s=1, f = x1



symbol

If we designed by schematic capture:

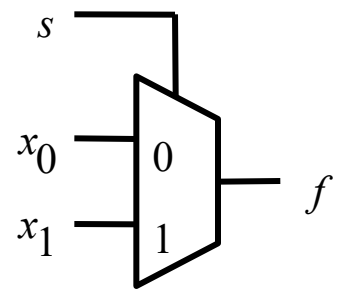


schematic

[BV]

Verilog Example 3 (2:1 mux):

Description:
When s=0, f = x0
When s=1, f = x1



symbol

Verilog behavioral description:

```
if (s == 0)
        f = x0;
else
        f = x1;
```

s == 0 is called an *equality test*
(s==0) is 1 (true) if s is 0
(s==0) is 0 (false) if s is not 0
== is called *equality operator*

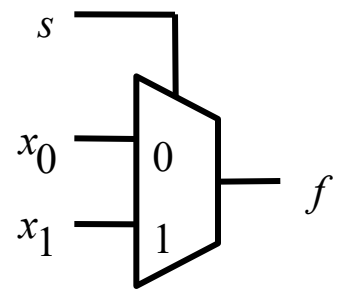
f = … is called an *assignment*
f = x0 means copy x0 into f
f = x1 means copy x1 into f

if … else … is a *procedural statement*

[BV]

Verilog Example 3 (2:1 mux):

Description:
When s=0, f = x0
When s=1, f = x1



symbol

Verilog behavioral description:

```
always @ (s or x0 or x1)
    if (s == 0)
        f = x0;
    else
        f = x1;
```

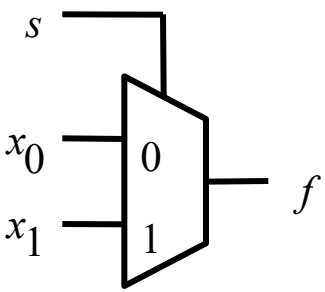
procedural statements (if - else) must be inside an **always** block

sensitivity list controls when an **always** block is executed
This case, if s or x0 or x1 changes, code inside **always** will run.
→ s, x0, x1 can affect output f

[BV]

Verilog Example 3 (2:1 mux):

Description:
When s=0, f = x0
When s=1, f = x1



symbol

Verilog behavioral description:

```
reg f;

always @ (s or x0 or x1)
    if (s == 0)
        f = x0;
    else
        f = x1;
```

signal f is assigned value inside an **always** block.

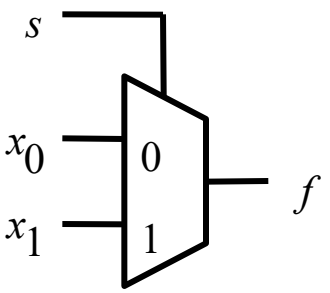
Only *variables* can be assigned inside an **always** block.

reg f declares f as a *variable*.

[BV]

Verilog Example 3 (2:1 mux):

Description:
When s=0, f = x0
When s=1, f = x1




```
module mux21 (s, x0, x1, f);
        input s, x0, x1;
        output f;
        reg f;

        always @ (s or x0 or x1)
            if (s == 0)
                f = x0;
            else
                f = x1;
endmodule
```

Now add *module name*, *port list*, *port declarations*, and *endmodule* to finish up the design

f is the module output and a variable at the same time.

design is done
testing has not started!

Now compare

1. Behavioral Description (always block)

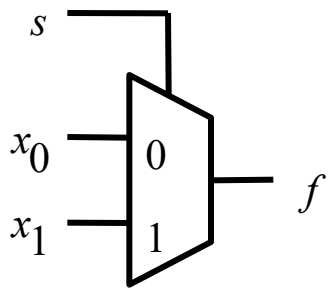
with

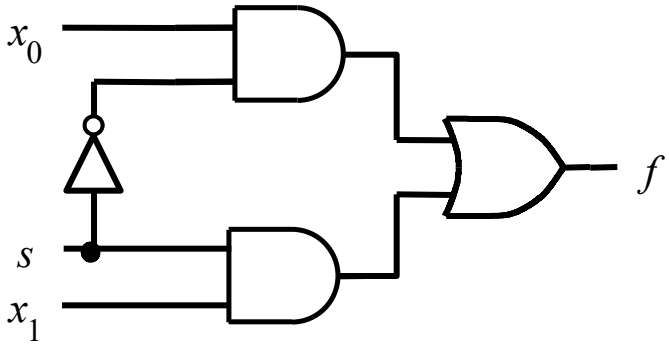
2. Assignments

Verilog Example 3 (2:1 mux):

[BV]

Description:
When s=0, f = x0
When s=1, f = x1






```
module muxb(s, x0, x1, f);
   input s, x0, x1;
   output f;
   reg f;

   always @ (s or x0 or x1)
      if (s == 0)
         f = x0;
      else
         f = x1;
endmodule
```

human readable

```
module muxa(s, x0, x1, f);
   input s, x0, x1;
   output f;

   assign f = (~s&x0)|(s&x1);

endmodule
```

what is s'x0 + sx1?

Example 3 review

Example 3 review:

```
module mux21 (s, x0, x1, f);
        input s, x0, x1;
        output f;
        reg f;

        always @ (s or x0 or x1)
            if (s == 0)
                f = x0;
            else
                f = x1;
endmodule
```

reg is a *declaration of variable(s)* for use inside *procedural statement*

always block is used to contain *procedural statements*

sensitivity list controls when an **always** block is executed

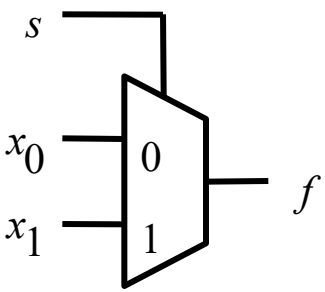
Verilog *procedural statement* (if, else, for, while, etc.)

Concept of *sensitivity list* :
Whenever s or x0 or x1 changes value, the always block is executed. List everything that should trigger execution of the *procedural statement* in this **always** block

Verilog Example 3 (2:1 mux):

[BV]

Description:
When s=0, f = x0
When s=1, f = x1




```
module mux21 (s, x0, x1, f);
        input s, x0, x1;
        output f;
        reg f;

        always @ (s or x0 or x1)
            if (s == 0)
                f = x0;
            else
                f = x1;
endmodule
```

Verification:
1. How do we know that the code compiles w/o any error?

2. How can we be sure that code matches the description above?

HARD part!

Verification:

Description:
When s=0, f = x0
When s=1, f = x1

```
module mux21 (s, x0, x1, f);
        input s, x0, x1;
        output f;
        …

endmodule
```

3 inputs: s, x0, and x1
each input is 1-bit

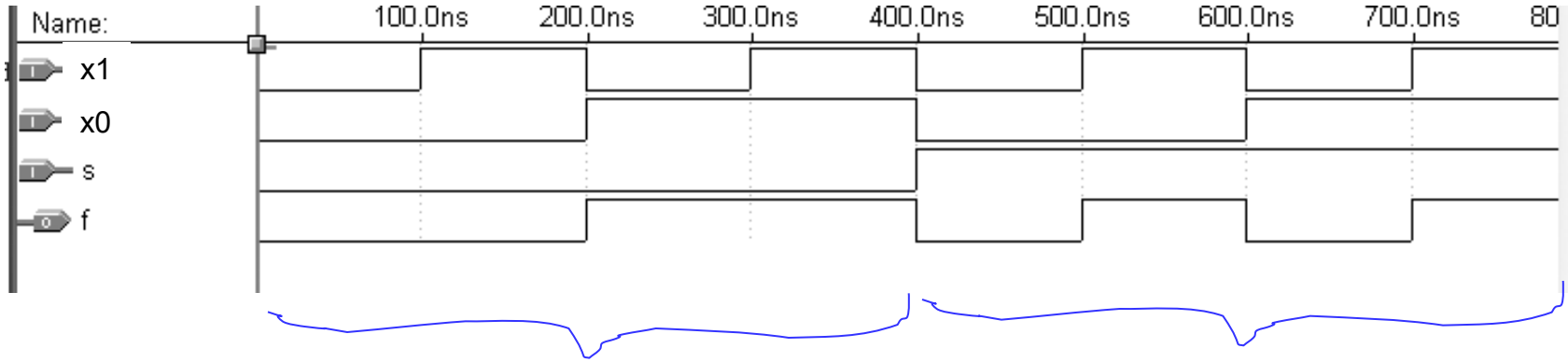
1 output, f, size is 1-bit

3 inputs → only $2^3 = 8$ input combinations are possible.

IDEA: Put in every input combination and see whether f is correct.

Each input combination to test design is called a *stimulus* (plural: *stimuli*).

Verification result:



s == 0
so f = x0

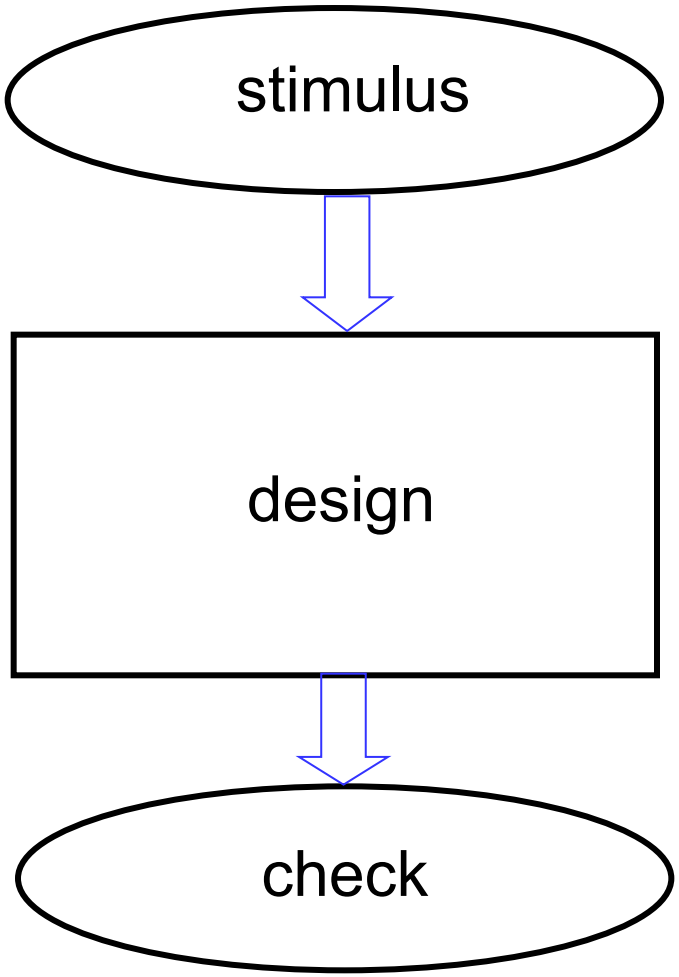
s == 1
so f = x1

We supplied 8 different *stimuli*, covering every possible input combination.

Ex: From 500nS to 600nS, x1=1, x0=0, and s=1
We *expect* f to follow x1, we *expect* f to be 1.

We looked at output and see that output f is what we want it to be. → DESIGN IS WORKING! DONE!

Design flow recap:



1. design
2. build test stimuli
   (also called *test vector*)
3. run simulation
4. *check* for *expected* result
5. *debug* design if *bug* exists
6. repeat until all *bugs* are fixed.

PITFALL:
Most people DON'T spend enough time testing. The end result is a very buggy design.

The spirit of hardware description language:

- Nobody writes modules that consists purely of gates!
- People write hardware description language (HDL) to specify the “description” or “behavior” of the hardware, **NOT** to duplicate schematics.
- Practice writing behavioral model.

A note to software experts:

- Writing HDL as if it were a computer program rarely works well, especially **for** and **while**.
- Think in terms of hardware.
  - You must “see” the schematics of your design