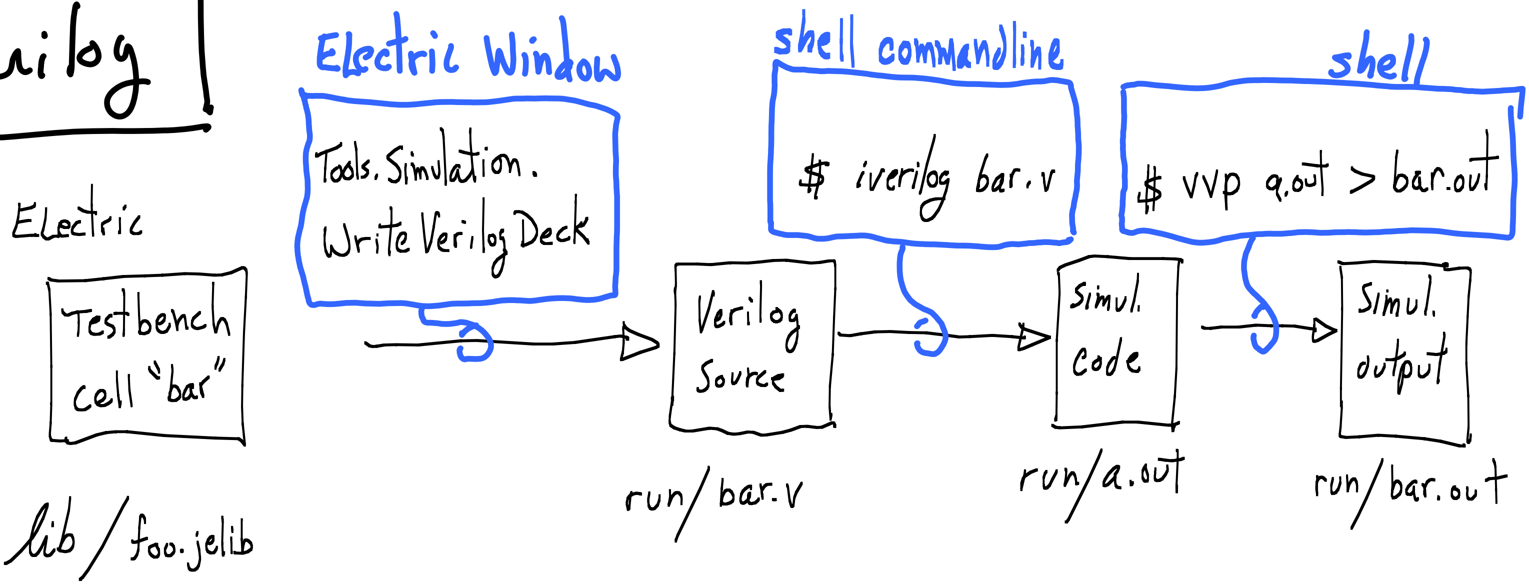
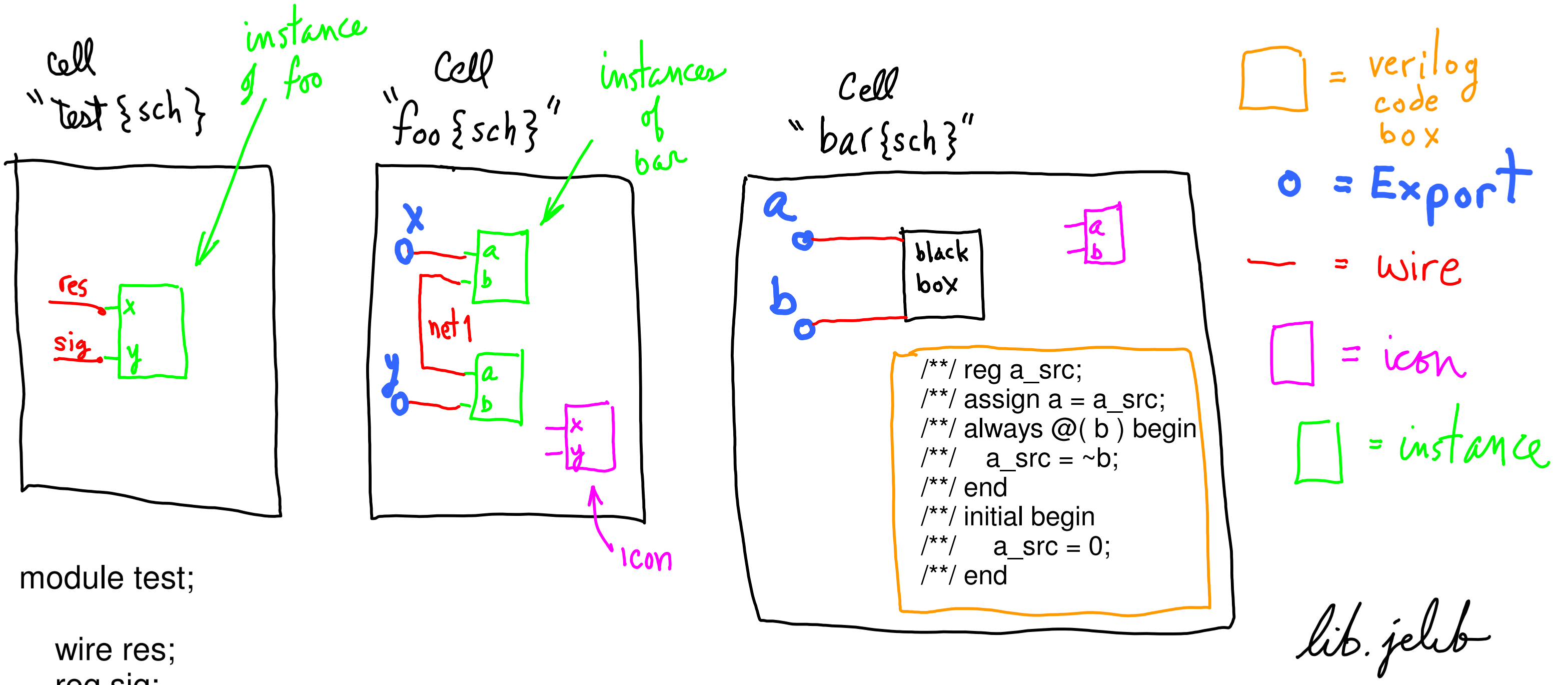
Verilog



Verilog Code Structure from Electric Cells



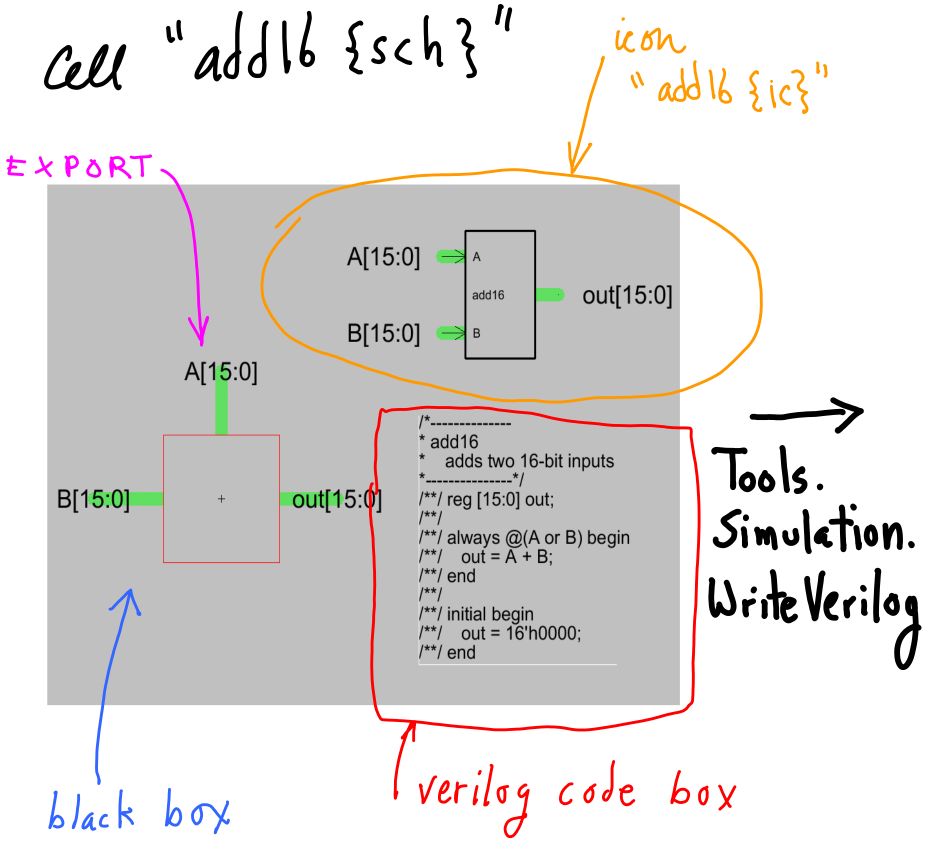
```
/**/ reg a_src;
/**/ assign a = a_src;
/**/ always @( b ) begin
/**/    a_src = ~b;
/**/ end
/**/ initial begin
/**/     a_src = 0;
/**/ end
```

```
module test;

   wire res;
   reg sig;

   lib__foo foo0( res, sig );

   initial begin
      sig = 0;
      #100 $finish;
   end

   always begin
      #1 sig = ~sig;
   end

end module
```

```
module foo( x, y);
   output x; wire x;
   input y; wire y;
   wire net1;

   lib__bar bar0( x, net1);
   lib__bar bar1( net1, y);

endmodule;
```

```
module bar( a, b);
   output a; wire a;
   input b; wire b;

  /**/ reg a_src;
  /**/ assign a = a_src;
  /**/ always @( b ) begin
  /**/    a_src = ~b;
  /**/ end
  /**/ initial begin
  /**/     a_src = 0;
  /**/ end

endmodule
```

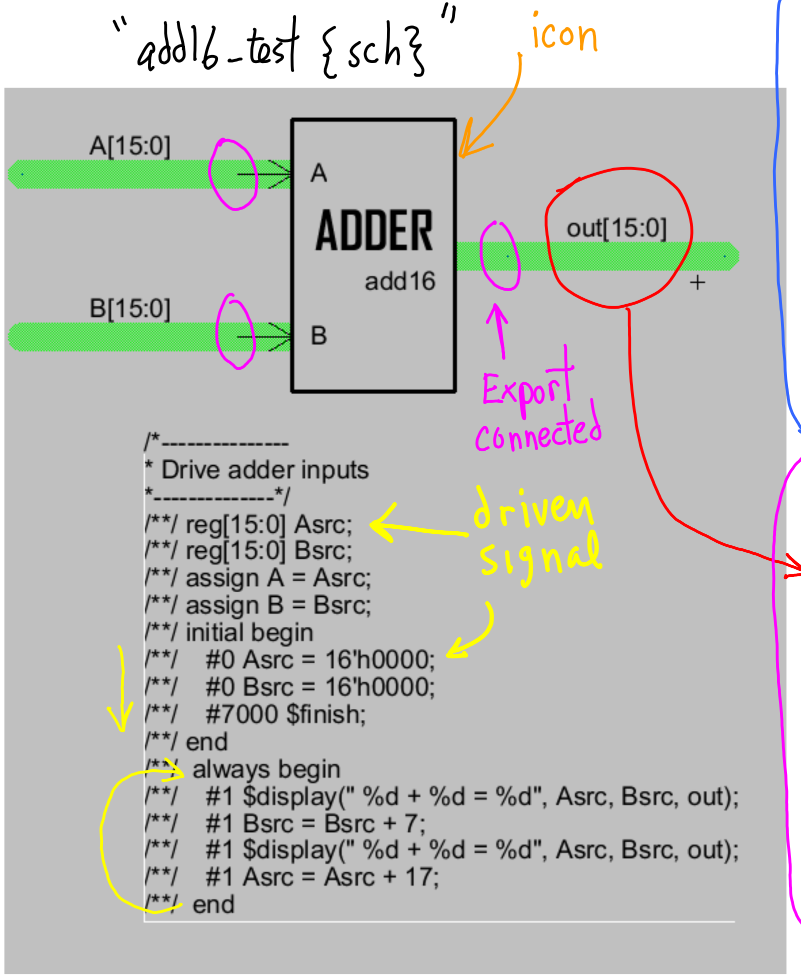
Cell "add16 {sch}"



```
/*--------------
* add16
*    adds two 16-bit inputs
*--------------*/
/**/ reg [15:0] out;
/**/
/**/ always @(A or B) begin
/**/    out = A + B;
/**/ end
/**/
/**/ initial begin
/**/    out = 16'h0000;
/**/ end
```

```
/* Verilog for cell 'parts:add16{sch}' from library 'parts'*/

module add16(A, B, out);
  input [15:0] A;
  input [15:0] B;
  output [15:0] out;

  /* user-specified Verilog code */
  /*--------------
  * add16
  *    adds two 16-bit inputs
  *---------------*/
  /**/ reg [15:0] out;
  /**/
  /**/ always @(A or B) begin
  /**/    out = A + B;
  /**/ end
  /**/
  /**/ initial begin
  /**/    out = 16'h0000;
  /**/ end

endmodule   /* add16 */
```

Cell
"add16_test {sch}"



```
/*--------------
* Drive adder inputs
*--------------*/
/**/ reg[15:0] Asrc;
/**/ reg[15:0] Bsrc;
/**/ assign A = Asrc;
/**/ assign B = Bsrc;
/**/ initial begin
/**/    #0 Asrc = 16'h0000;
/**/    #0 Bsrc = 16'h0000;
/**/    #7000 $finish;
/**/ end
/**/ always begin
/**/    #1 $display(" %d + %d = %d", Asrc, Bsrc, out);
/**/    #1 Bsrc = Bsrc + 7;
/**/    #1 $display(" %d + %d = %d", Asrc, Bsrc, out);
/**/    #1 Asrc = Asrc + 17;
/**/ end
```

```
module parts__add16(A, B, out);
  input [15:0] A;
  input [15:0] B;
  output [15:0] out;

  /* user-specified Verilog code */
  /*--------------
  * add16
  *    adds two 16-bit inputs
  *---------------*/
  /**/ reg [15:0] out;
   ...
  /**/ end

endmodule   /* parts__add16 */
```

defn of add16

Export

```
module add16_test();
  wire [15:0] A;
  wire [15:0] B;
  wire [15:0] out;

  /* user-specified Verilog code */
  /*---------------
  * Drive adder inputs
  *--------------*/
  /**/ reg[15:0] Asrc;
  ...
  /**/  end

  parts__add16 ADDER(.A(A[15:0]), .B(B[15:0]), .out(out[15:0]));
endmodule   /* add16_test */
```

Top-level, no args

BUS

instance of add16

Export

BUS

Structural vs. Behavioral

Structural ⟷ wire, gates, devices: wire, reg, AND, or, ...

Behavioral ⟷ if( ) then ( ), while, wait, ..., case, ...

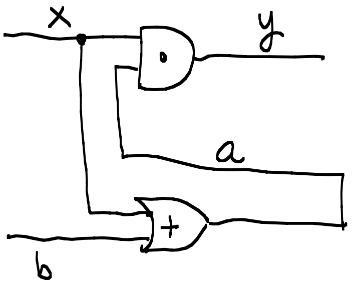
"Helper" Language
integer, ..., while, ...

Gate-level

STRUCTURAL

```
module foo ( y, x, b );

   input x, b;
   output y;
   wire y, x, a, b;

  and    and_0( y, x, a );
  or        or_0( a, b, x);

endmodule
```

via "primitives"



OR via "continuous assignment"

```
module foo ( y, x, b );
   input x, b;
   output y;
   wire y, x, b;

   assign y = (x & (x | b));

endmodule
```

Delays

When (what simulation time) does

a become 0?

b become 0?

c become 0?

Signal values
----

x == don't know
z == disconnected
0
1

(x & 0) = ?
(x | 0) = ?

```
initial begin
        #1 a = 0;
#1 $display()
        #1 b = 0;
        #1 c = 0;
       #1
end
```

what are the values of a, b, c for every tic?

$display("time=0%d a=%b", $time, a);
$display(" ... " b);
$display(" ... " c);

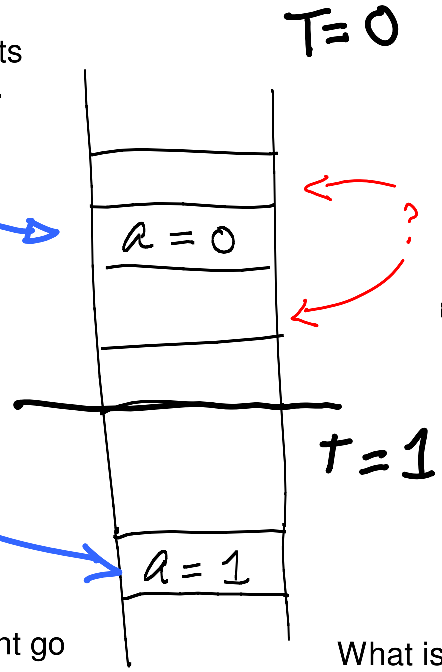
Event Queue

All "initial" and "always" statements execute in parallel.

```
initial begin
  a = 0;
  #1 a = ~a;
end
```

```
always @(a) begin
  c = ~a;
end
```

Where does this event go in queue?



```
initial begin
  b = 1;
end
```

```
initial begin
  $display(...a)
end
```

What is the value printed by $display("a=%d", a)?

What timestamp does the event $display() have?

Events cause other events: signal "a" changes, creates event that changes "c".

Events placed in queue, pulled from queue for current step,new events added, until no events left in this time step.

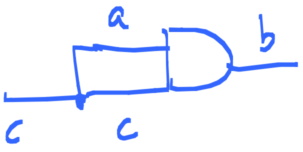
← no ordering of events *

Discrete Event Simulation

(versus Discrete Time Simulation)

Ordering of Events*

c = 0
a = 0
event: c ← 1



blocking assignment

```
input c;
wire c;
output a, b;
reg a, b;

initial begin
   a = 0;
   b = 0;
end
always @(c) begin
   a = c;
   b = a & c;
end
```

c ← 1
a ← 1
b ← 1

a's new value used for b.

RHS evaluated in order, after preceding LHS assignment

non-blocking assignment

```
input c;
wire c;
output a, b;
reg a, b;

initial begin
   a = 0;
   b = 0;
end
always @(c) begin
   a <= c;
   b <= a & c;
end
```

c ← 1
a ← 1
b ← 0

a's old value used for b

RHS evaluated in parallel, globally, before LHS assignment

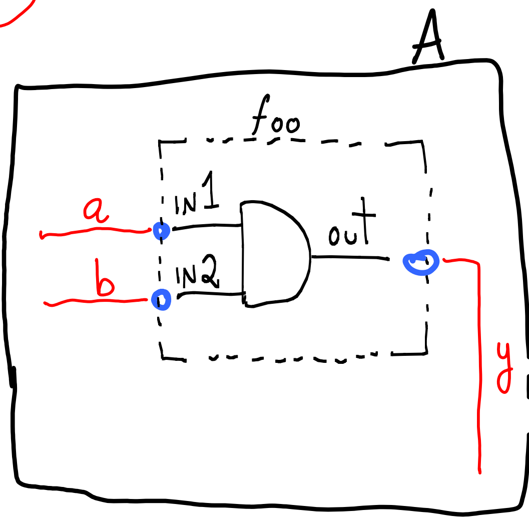
Ports by name

```
module A;

   wire a, b, y;

   MY_AND foo( .in1( a ), .in2( b ), .out( y ) );

endmodule

module MY_AND( out, in1, in2 );

   out <= #1 (in1 & in2);

endmodule
```

connect by name: order doesn't matter

Should def'n of MY_AND be inside def'n of A?

How deep can nesting be?



OR

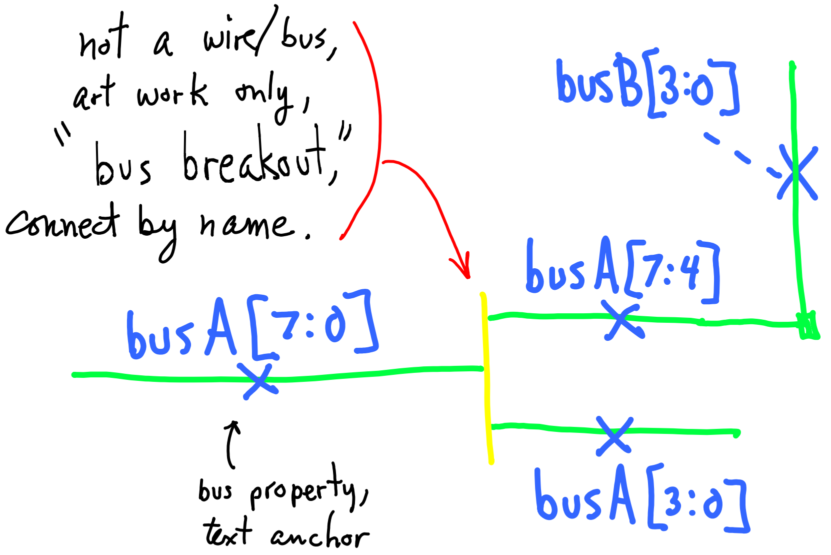
order matters

MYAND foo(y , a, b)

Busses, Arrays

```
wire [7:0] busA;
wire [3:0] busB;
reg busAsrc;
assign busA = busAsrc;
assign busB = busA[7:4];

initial begin
   busAsrc = 8'd255;
end
//-- busAsrc = 8'hff;
//-- busAsrc = 8'b11111111;
```

order assumed



not a wire/bus, art work only, "bus breakout", connect by name.

bus property, text anchor

(See, Edit.SelectObject)

nums format

size(#bits)		format	number
8	,	d	255
		h	FF
		b	1111 1111

(d = decimal format
h = hex format
b = binary format)

Combining busses

```
input  [3:0]  a, b;
output [15:0] y;

assign y = { 3 { a[3:0] },  b[3:2],  2'b00 };
```

concatenate

duplicate

what is connected to:

y[0]? →	0
y[3]? →	b[3]
y[4]? →	a[0]
y[12]? →	a[0]
y[15]? →	a[3]

definitions, parameters

```
`define  busWidth  16
reg [(`busWidth -1): 0]  busA;

`include foo.vh
```

(put defs in header)

```
module f
   parameter  WIDTH = 8;
   reg [(WIDTH -1) : 0]  busA;
   ...
endmodule

module h
   defparam  f1.WIDTH = 32;
   f  f1;

   f #(16)  f0;
```

} explicit

← by order declared :
#(p1, p2, p3)

ARRAYS

reg [7:0] regWords [15:0];

$$\underbrace{\text{reg [7:0]}}_{\text{type}}\ \underbrace{\text{regWords}}_{\text{id}}\ \underbrace{\text{[15:0]}}_{\text{array size}}$$

```
regWords[1] = 8`b01010000;
regWords[1][0] = 1'b1;
```

What's in
RegWords[1] ?

→ 01010001

Tasks = methods

```
module memory(...);
   ...

   reg [7:0] regWords [15:0];
   integer i;

   task clear;
        begin
           for ( i = 0; i < 15; i = i + 1) begin
                regWords[i] = 8'd0;
           end
        end
   endtask

endmodule
```

Task

invoke Task

```
module top ();

   memory mem;

   initial begin
        mem.clear;
        mem.regWords[0] = 8'b000111;
   end

endmodule
```

Names from above

in Top : instance.instance.thing

Pre-defined procedures | VPI

- $display("   ", ...);   has eoln
- $write ("   ", ...);   no eoln
- $time   sim. time step
- $monitor("   ", x);   like $display, but w/ always @(x)
- $strobe   (broken?)
- $fwrite   also Multi-Channel Descriptor: multiple files
- $fopen   or File-Descriptor: 1 file

⋮

- $readmemb("file", dataArray); text file contains binary notation
- $readmemh("file", dataArray□); text file contains hex notation

optional, Begin/end indices

- $dump(   )   — use w/ GTKwave
  dumps every signal @ every change
- $stop   goes into interactive mode

^C sends "kill" signal to process, interactive mode, use $finish

- $finish   ends simulation

More Signal propagation delays

```
#2 A = B
```

B sampled and A changes at T+2

```
A = #2 B
```

B sampled at t,
A changes at t+2

```
and #(3,2) and1(Y,A,B)
```

A or B changes, causing Y change	Y changes @
Y → 1	t+3
Y → 0	t+2
Y → z	t + min(3,2)

```
wire A
assign #2 A = B & C
```

B + C sampled at t
A changes at t+2
(unless b, c change → canceled)

```
wire #2 A;
assign A = B & C
```

wire A takes 2 ticks to see (B&C),
same as above.

```
always @(posedge clk) begin
    a = b;
    @(negedge clk)
    a = ~b;
    @(c or clk)
      a = 0;
end
```

What's in "a" if c doesn't change?

```
always begin
    wait(a);
  #1
      c = ~c;
end
```

← only waits if $a \neq 1$ (or FALSE)

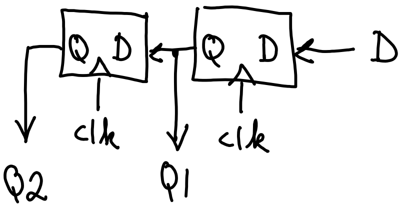
```
initial begin
    c = 0;
    a = 0;
  #3
    a = 1;
  #3
    a = 0;
  #1 $finish;
end
```

How many times does c change?

→ easy way:

```
always @(c) begin
    $display("%b", c);
end
```

```
always @(posedge clk) Q1 = D;
always @(posedge clk) Q2 = Q1;
```



```
initial begin
  @(posedge clk)
  D = 0;
  @(posedge clk)
  D = 1;
end
```

what happens to Q1, Q2?

MODELS

STRUCTURAL

```
and A(y, x, z)
or  B(z, y, x)
```

Dataflow

```
assign z = (y | x);
assign y = (x & z);
```

Behavioral

```
if (x == 0)
   z = 1;
   y = 0;
```

Other language elements

---- Looping (forever, while, for)

These can appear inside an "initial" or "always", and can thus start at times other than 0. Conditionals are T if they evaluate to 1, F if they evaluate to 0, x, or z. "Forever" is the same as "while(1)".

---- Control (fork, join)

Creates parallel event streams that synchronize at the "join": all enclosed "begin-end" blocks run in parallel and the last to finish exits the "fork-join". E.g.,

```
     fork
        begin ... end
        begin ... end
    join
```