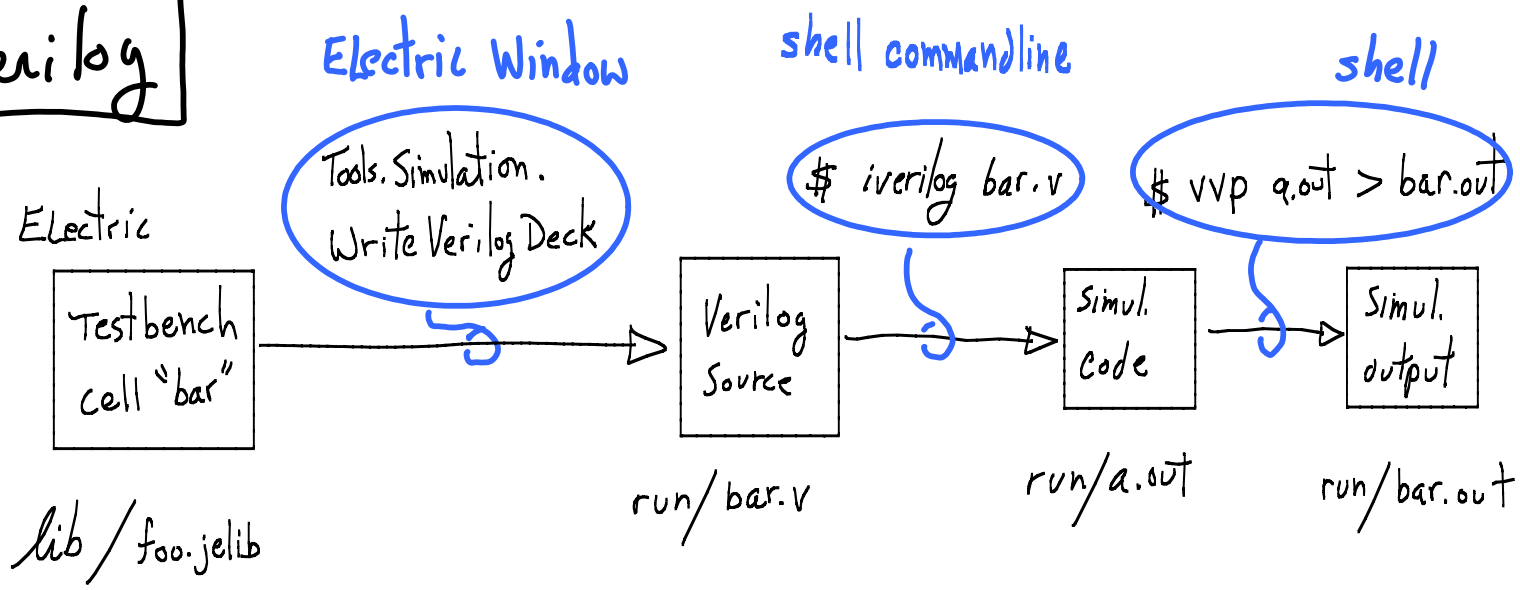
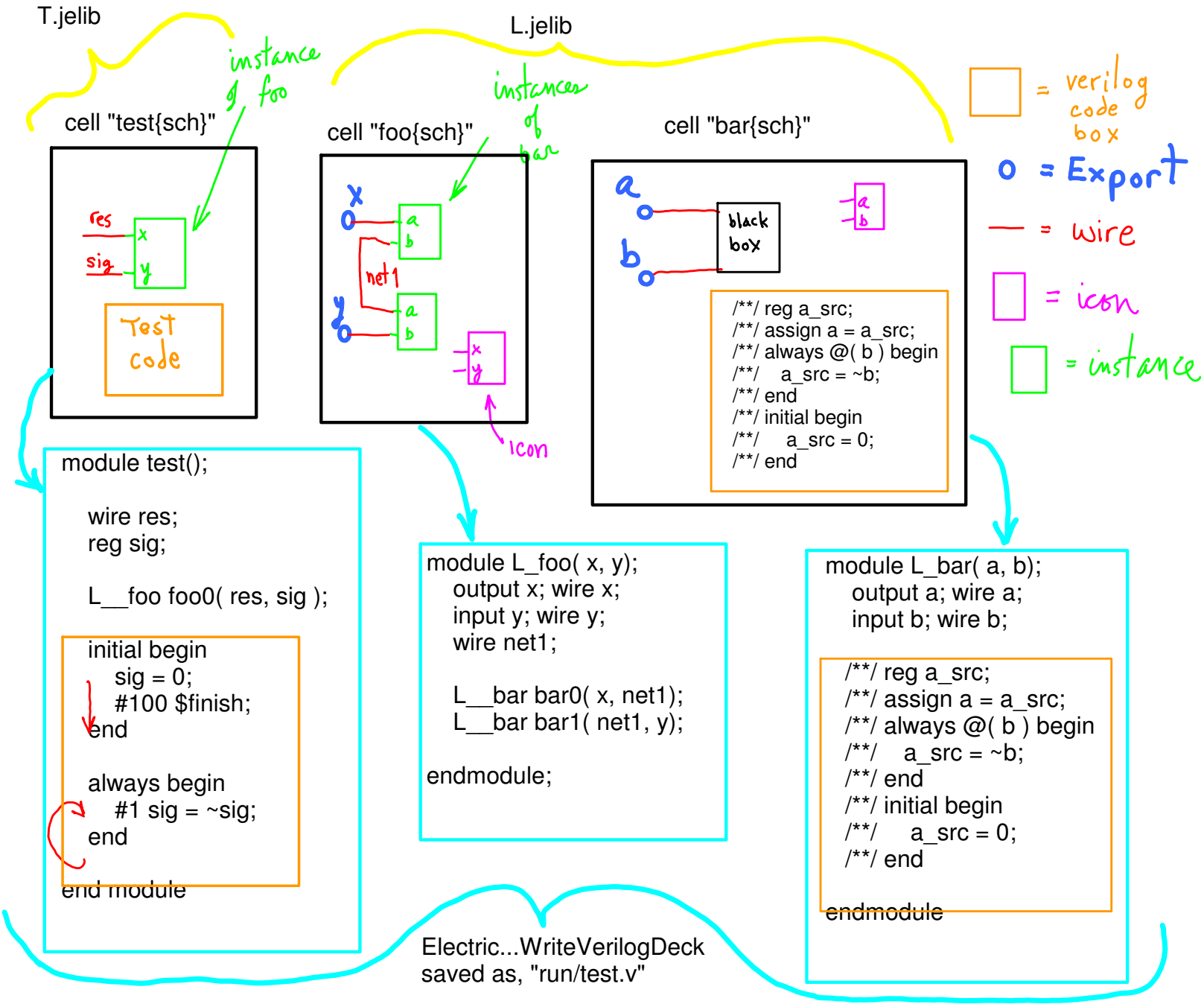
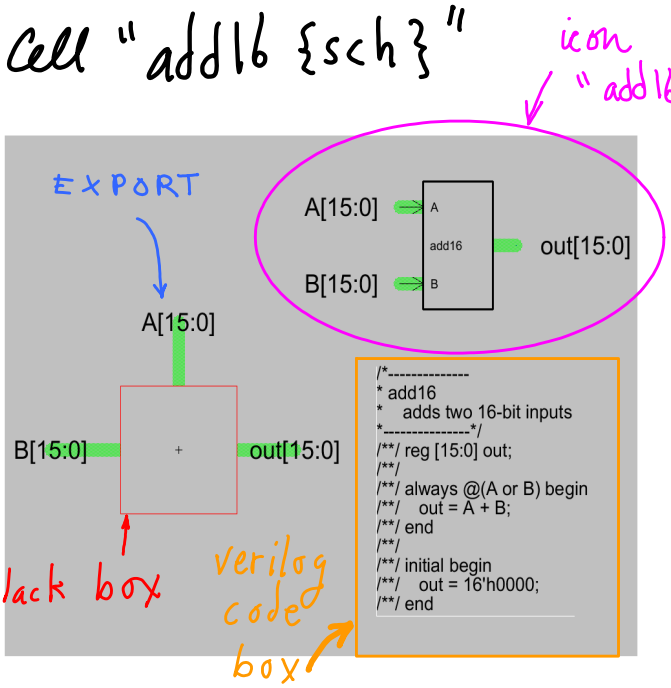
Verilog



Verilog Code Structure from Electric Cells



Cell "add16{sch}"

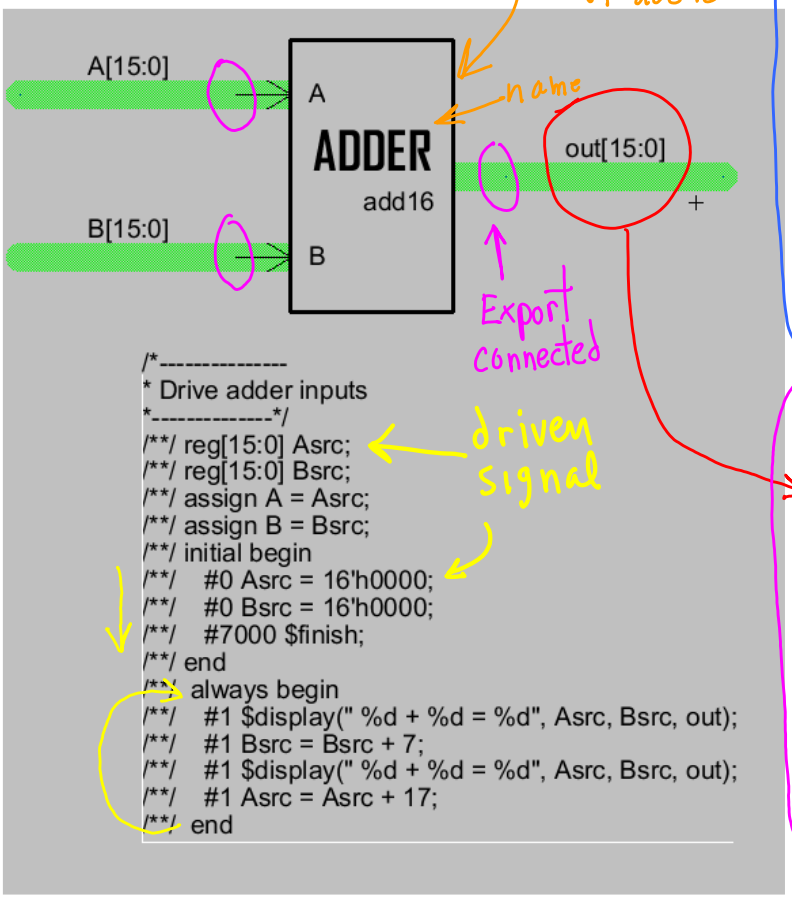


add16.v

Electric. Tools. Simulation. Write Verilog

```
/* 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}"



defn of add16

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

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

endmodule   /* parts__add16 */

module add16_test();
  wire [15:0] A;
  wire [15:0] B;
  wire [15:0] out;
  ← BUS
  /* 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

instance of add16

Export BUS

Structural vs. Behavioral

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

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

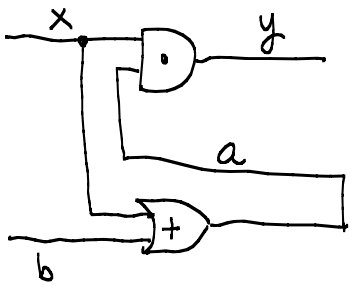
"Helper" Language
integer, ..., while, ...

Gate-level

STRUCTURAL

via "primitives"

```
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
```



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

at what simulation time does
a become 0?
b become 0?
c become 0?

possible signal values
---
x == don't know
z == disconnected
0
1

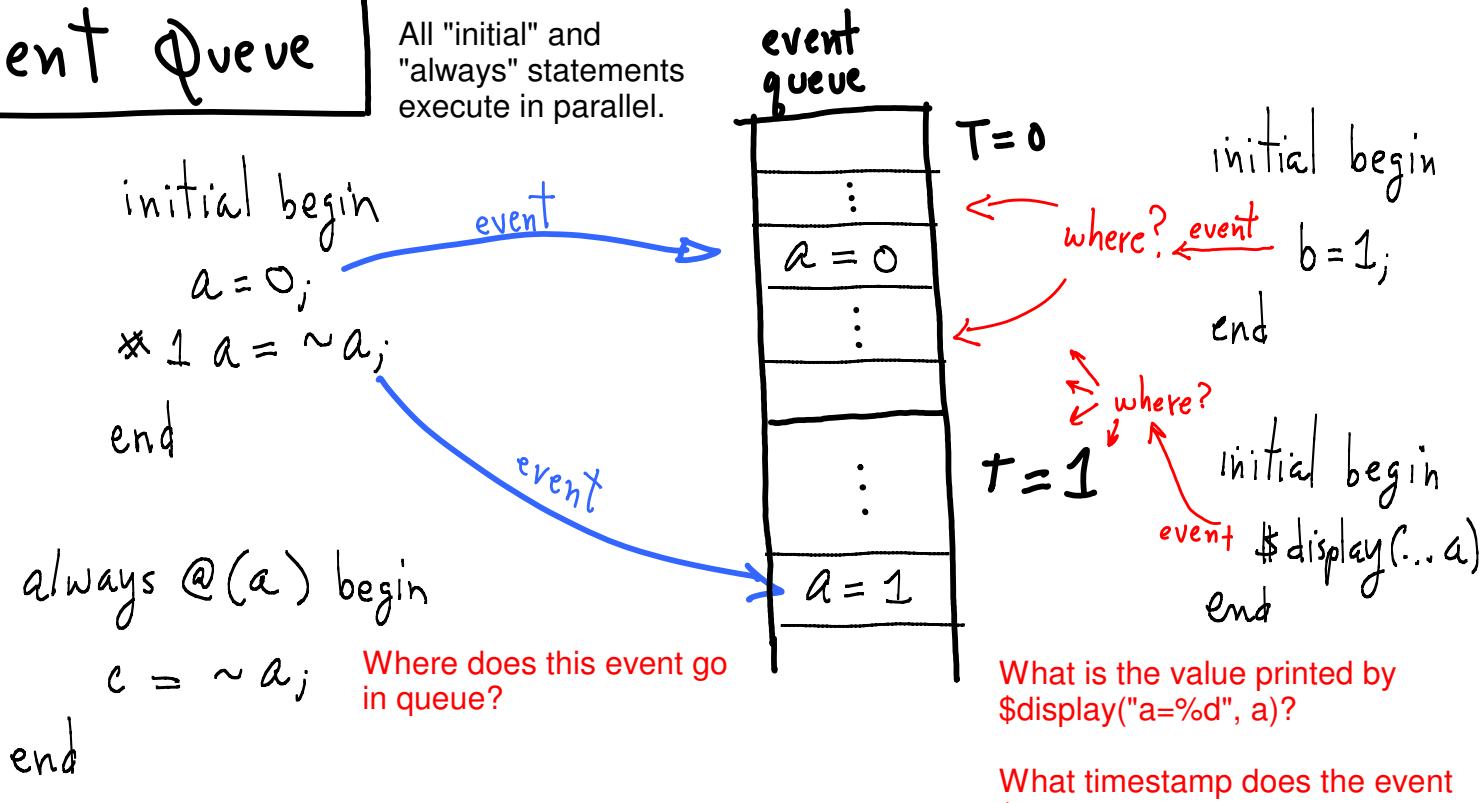
```
module f();
  reg a, b, c;

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

  always @(a or b or c) begin
    $display("--- t=%0d ---", $time);
    $write("a=%b ",a);
    $write("b=%b ",b);
    $write("c=%b ",c);
    $write("\n");
  end

endmodule
```

at what time do these events occur? Are they ordered, i.e., blocking?

```
--- t=1 ---
a=0 b=x c=x
--- t=2 ---
a=0 b=0 c=x
--- t=3 ---
a=0 b=0 c=0
```

Event Queue

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



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.

Discrete Event Simulation

(versus Discrete Time Simulation)

← no ordering of events *

Ordering of Events

c = 0
a = 0
event: c ← 1


```
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
```

blocking assignment

c ← 1
a ← 1
b ← 1

a's new value used for b.

RHS evaluated in order, after preceding LHS 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
```

non-blocking assignment

c ← 1
a ← 1
b ← 0

a's old value used for b

RHS evaluated in parallel, globally, before LHS assignment

Ports by name

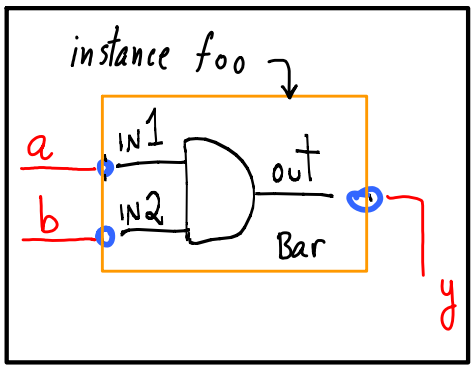
Should def'n of bar be inside def'n of A?

How deep can nesting be?

One level.

```
module A;
  wire a, b, y;
  Bar foo( .in1( a ), .in2( b ), .out( y ) );
endmodule

module Bar( out, in1, in2 );
  out <= #1 (in1 & in2);
endmodule
```

connect by name: order doesn't matter



OR

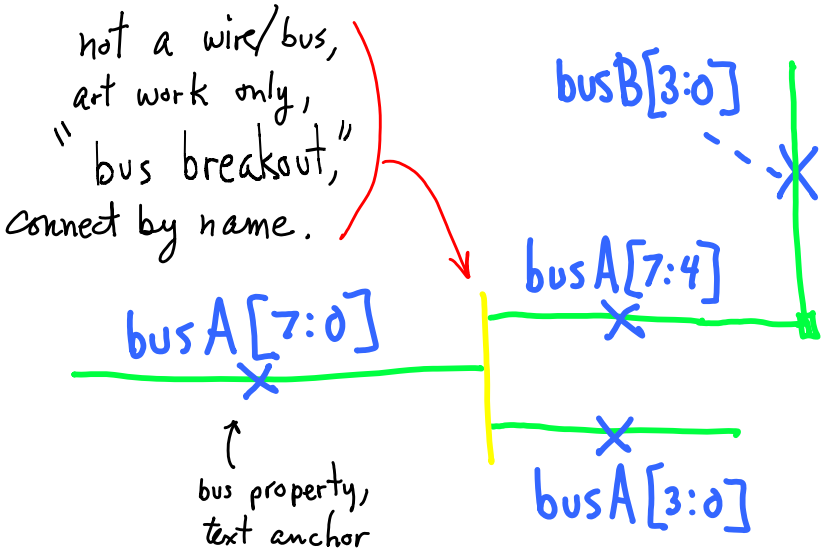
order matters

```
Bar 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;
```

wire order assumed



(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 and concatenate

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;
```

macro substitution.

(put defs in a header file and `include it)

```
`include header.vh
```

(These are " ` ", ie., back single quote)

Parameters

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

Seems redundant, but here's how we use it.

```
module H();
  defparam foo.WIDTH = 32;
  F foo;
  F #(16) bar;
```

set (override) parameter explicitly

} ← or set implicitly by order parameters were declared, e.g., #(10, 16, 2) sets parameters P1 = 10, P2 = 16, P3 = 2

Arrays

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

type | id | array size

e.g.,

```
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

```
module top ();

   memory mem;

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

endmodule
```

invoke task

Naming in hierarchy

name1.name2.name3

top-level instance name | instance inside name1 | instance inside name2

Pre-defined Tasks

$display("..." , ...); — has eoln

$write("..." , ...); — no eoln

$time; — simulation time step

$monitor("...", ...);
$strobe(...); } — like $display, but w/ implicit "always @(x)" (broken?)

$fopen(...);
$fwrite(...); } — uses Multi-Channel Descriptor: multiple files, or use a File-Descriptor: 1 file

$readmemb("filename", dataArray); — (reads data into model, text file contains binary notation)

$readmemh("filename", dataArray, ...); — (reads data into model, text file contains hex notation)

optional: Begin/end indices for both binary and hex

$dump(...); — dumps every signal @ every change, use w/ GTKwave

**$stop;** — goes into interactive mode, then enter "$finish" to quit.
NB- entering ctl-c during simulation also goes to interactive mode.

$finish; — quit simulation

More Signal propagation delays

```
#2 A = B
```
} B sampled and A changes both 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 @

implicit set parameters

Y changes	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?

How many times does c change?

```
always begin

   wait( a );
   #1
   c = ~c;

end
```

only waits if $a \neq 1$ $(0, x, z)$

$\cong$ wait-until $(a == 1)$

```
initial begin

   c = 0;
   a = 0;
   #3
   a = 1;
   #3
   a = 0;
   #1
   $finish;

end
```

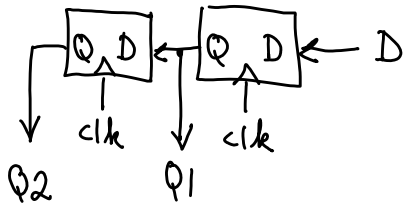
→ easy way to find the answer :

```
always @(c) begin
   #1 $display("%b", c);
end
```

---

what happens to Q1, Q2 ?

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

always @( posedge clk)  Q2 = Q1;

initial begin

   @( posedge clk )  D = 0;

   @( negedge clk )  D = 1;

end
```



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
```