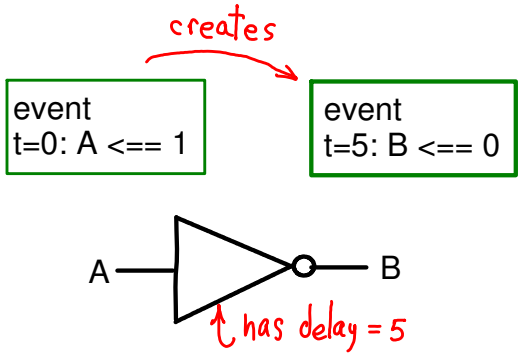
Verilog

1. **Produce a Verilog file:** foo.v
2. **compile it for simulation:** iverilog foo.v
3. **simulate system:** vvp a.out > testResult.txt → output to file
4. **see what happened:** vi testResult.txt → see content

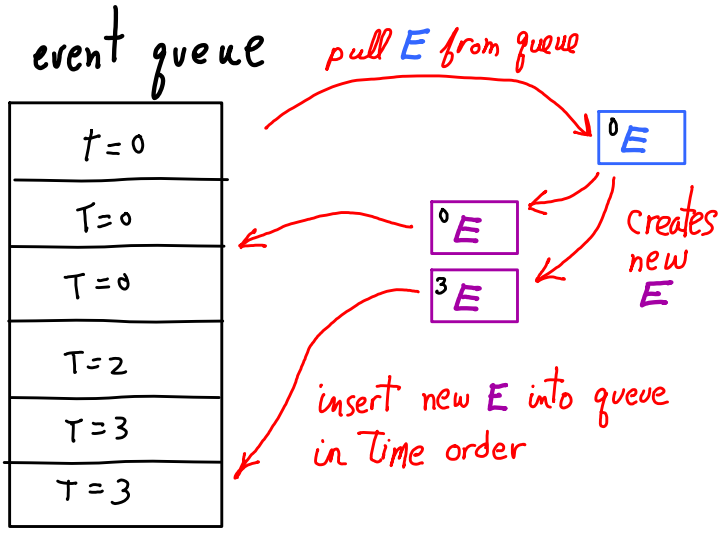
Discrete Event Simulation

--- create an event which occurs at time t
--- check what that event will cause to happen:
  --- create new events at t (zero delay)
  --- create new events at t + d (d delay)



Event Processing

WHILE(1)
  --- get earliest event E from queue
  --- E creates new events, insert into queue
  --- delete E
end_WHILE



Q. Which t=0 event is processed first?

Verilog statements, making things happen / creating events

Only two types of "action" statements:

--- initial
--- always

all initial and always statements run in parallel
they all start creating events at t=0

```
initial begin

  A = 0;
  #1
  A = 1;
  #1
  A = 0;

end
```

create event, t=0
wait 1 tick
create event, t=1
wait 1

go once through

```
always begin

  A = 0;
  #1
  A = 1;
  #1
  A = 0;

end
```

go through

start over, forever

Q. How many events created at t=2?

Q. If there were no delays (#1), how many at t=0?

```
reg clock;

initial begin
   clock = 0;
end

always begin
   #10 clock = ~ clock;
end
```

this is a free running clock.
Starts = 0 at t = 0.
Switches to 1 at t = 10,
back to 0 at t = 20,
...
forever

```
always @( clock ) begin
   $display( "time=%d", $time );
   $display( "clock = %b", clock );
end
```

This waits for clock to change.
Then creates two events to call system functions $display + $time.

Q. At what simulation time will the first $display occur? The second? The third? Which $display runs first?

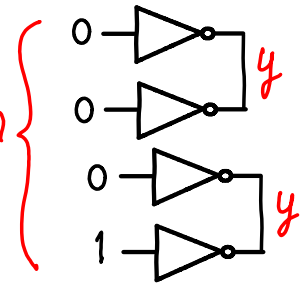
Signal Values

All wires are x or z until some event changes it to 1 or 0.

Verilog knows three signal values:

--- 0, logic 0 (~ GND)
--- 1, logic 1 (~ +5v for us)
--- x, unknown logic value; wire is driven, but cannot determine value.
--- z, high-impedance, nothing driving the wire to a specific value.

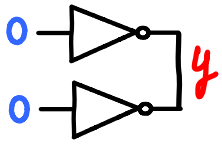
what value makes sense for y in each case?




```
initial begin
   A = 0;
   $display( "A = %b", A);
  #5
   $display( "time=%0d", $time );
  ...
```

Q. What value will $display show for A?

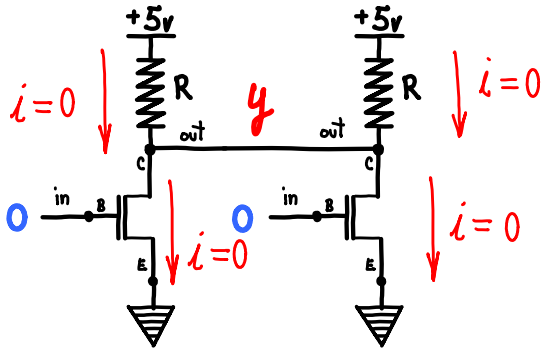
what value makes sense for y?



Verilog reasons it this way:

NOT(0) == 1 == NOT(0)

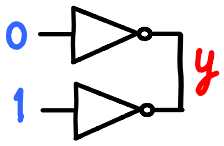
Which, works out electrically as well: y = 1.

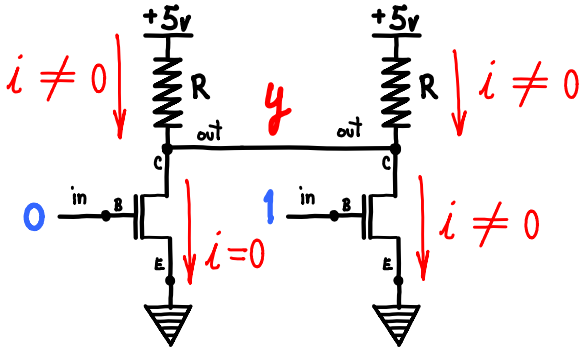


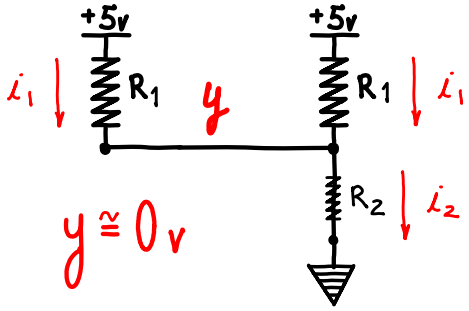
ΔV across R = iR = 0R = 0

y = +5v

what value makes sense for y ?







As a logical statement, the NOT-NOT circuit says this,

NOT(0) == NOT(1)

This does not make sense, mathematically.

Electrically, the value of y might be a perfectly good logic value near 0v. But that depends on the device's characteristics (see at right).

Verilog will simply say it cannot guess: y = x.

actually, $i_2 = i_1 + i_1 = 2 i_1$

$\Delta V_2 = i_2 R_2 = 2 i_1 R_2$

$i_1 = (5v - \Delta V_2) / R_1$

$\Delta V_2 = (5v - \Delta V_2) 2 R_2 / R_1$

$\Rightarrow \quad \Delta V_2 = 10 R_2 / (R_1 + 2R_2)$

if $R_2 \ll R_1$, then $\approx 0v$

but, what happens in real device?
measure it!

How to stop the simulation:

```
initial begin
    #1000 $finish;
end
```

This runs in parallel w/ other statements.
It waits until $t = 1000$, then calls $finish

Other Verilog Language Elements

--- Definitions, e.g., reg A, assign, ...

--- Helper code, helps you write actions using less text
integer, for, ...

--- Behaviorial modeling
while, if, then, ...

documentation:
see projects/LC3trunk/docs/verilog
or google "verilog tutorial"

Work Flow

Electric Window

shell commandline

^Tools.Simulation(Verilog).
WriteVerilogDeck

%> iverilog bar.v

%> vvp a.out > res.txt

Testbench cell "bar"

lib / foo.lib

Verilog Source

bar.v

Sim. Code

a.out

Sim. output

res.txt

Verilog Code Structure from Electric Cells

T.jelib

L.jelib

instance of foo

instances of bar

cell "test{sch}"

cell "foo{sch}"

cell "bar{sch}"

bar's icon

res

sig

x

y

Test code

net1

foo's icon

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

= verilog code box

o = Export

— = wire

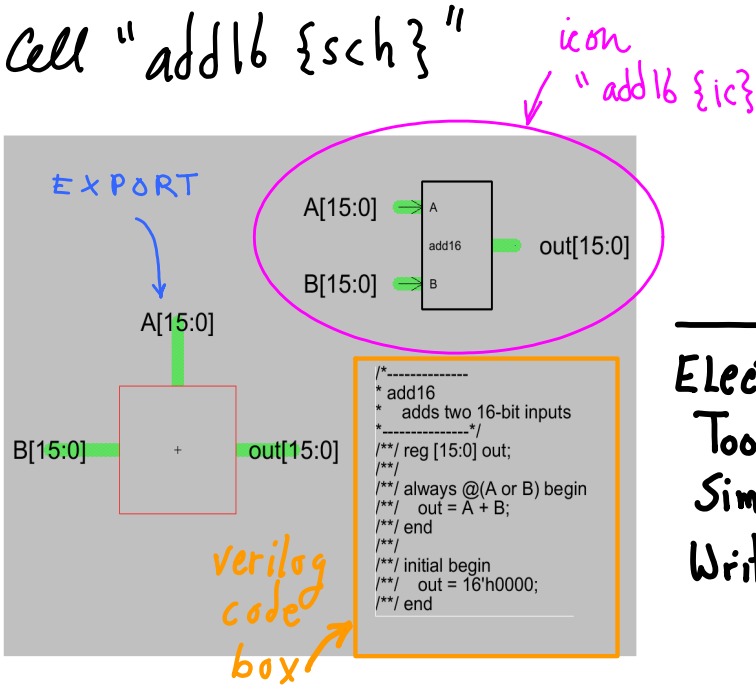
```
module test();

   wire res;
   reg sig;

   L__foo foo0( res, sig );

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

   always begin
      #1 sig = ~sig;
   end

end module
```

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

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

endmodule;
```

```
module L_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
```

Electric...WriteVerilogDeck
saved as, "test.v"

Cell "add16{sch}"



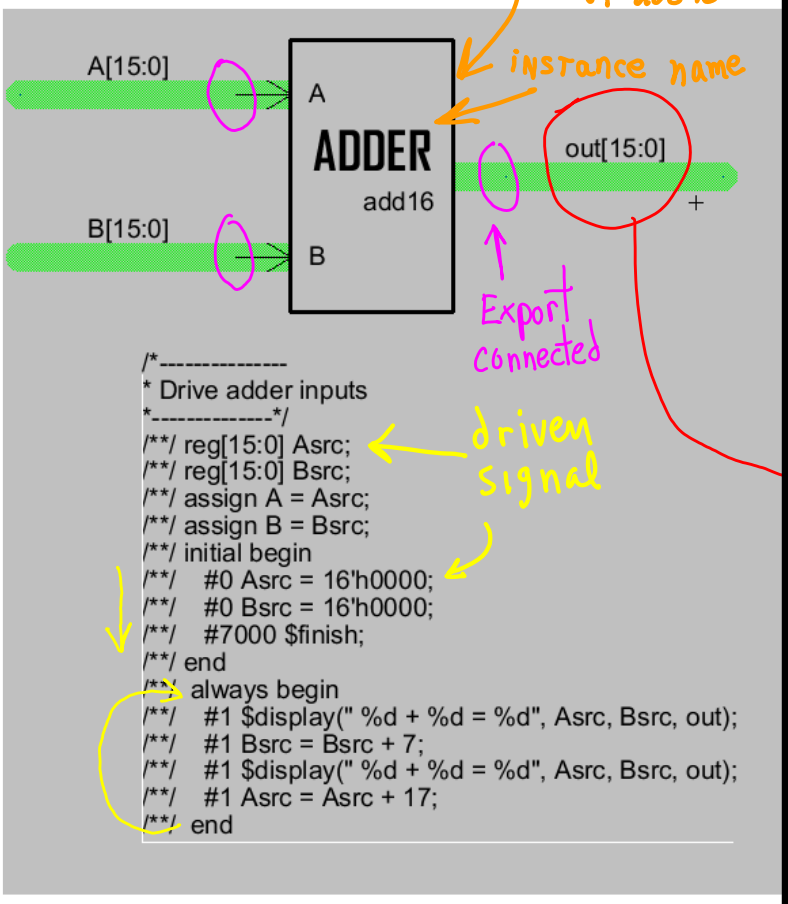
add16.v

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 */
```

Write Verilog

cell "add16_test{sch}"




```
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 */
```

defn of add16

add16_test's code

```
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

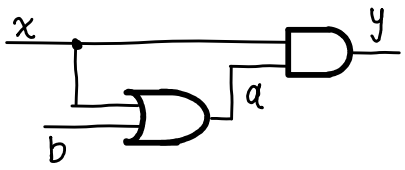
BUS

Structural vs. Behavioral

Structural — **wires and gates:** verilog wire, reg, and, or, buff, ... (other primitives)

Behavioral — **description of behavior:** verilog if(), while(), case(), ... (other constructs)

Gate-level Modeling



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
```

Data Flow via "continuous assignment"

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

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

endmodule
```

Delays

--- We can specify delays for devices and wires (modeling real life). But then we have to check circuit timing issues.

--- We can have 0 delay devices (math/logic). But then we don't know how things actually occur.

--- If you cannot be sure how events will be ordered, put in a delay.

Q. Will the $dispay() and $write() events occur before or after signal changes?

--- This will make you more certain,
#1 $display();

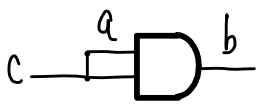
```
module f();
  reg A, B, C;

  initial begin
    #2 A = 0;
    #2 B = 0;
    #2 C = 0;
    #2 $finish;
  end

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

endmodule
```

Q. Trace all the output events, showing what is printed.

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

ordering 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

Which is the correct, i.e., more realistic?

Depends on what we are modeling.

USE **<=** (non-blocking, for parallel occuring assignments)

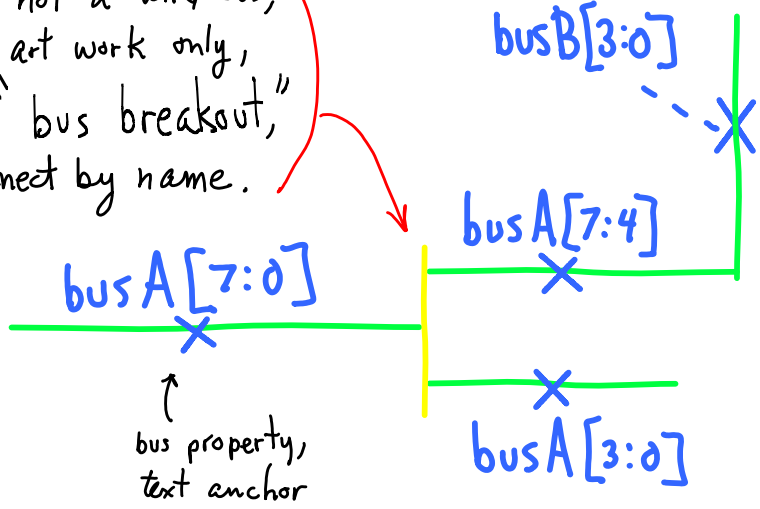
USE **=** (blocking or procedural, for behavioral modeling)

Busses, Arrays

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

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

wire order assumed

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



Expressing multi-wire signal values

size(# bits) ' format number

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

interpret number as an expression in:

d = decimal
h = hex
b = binary

Combining busses

```
input  [3:0]  a, b;
output [9:0] y;

assign y = { 3 { a[1:0] },  b[3:2],  2'b00 };
```

concatenate

(duplicate 3 times) and (concatenate)

```
{ a[1], a[0], a[1], a[0], a[1], a[0], b[3], b[2], 1'b0, 1'b0 }
```

y[9] y[8] ... y[0]

4-bit numbers

decimal	hex	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

$$\text{hex } 2D = 2\times16^1 + D\times16^0$$
$$= (0010)\times 2^4 + (1101)\times(2^4)^0$$
$$= 00100000 + 1101$$
$$= 00101101 \text{ binary}$$

⇒ easy to convert hex ⟷ binary

definitions, parameters

```
`define BUSWIDTH  16

reg [ (`BUSWIDTH - 1) : 0 ]  busA;
```

macro substitution.

Equivalently: `reg [ 15 : 0 ] busA;`

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

```
`include header.vh
```

(" \ " 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;
```

override parameter explicitly affects following def

or

```
F #(16) bar;
```

implicit override: by order parameters were defined. E.g., #(10, 16, 2) means 1st param gets 10, 2nd gets 16, 3rd gets 2

Arrays

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

type | id | array size

e.g.,

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

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

Q. What's in regWords[1] now?

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

"helper" code = shorthand

otherwise, 16 assignments

```
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, ...); — (text file contains hex notation)

optional: Begin/end indices (both binary and hex)

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

**$stop;** — goes to interactive mode, then enter "$finish" to quit.
(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)
```

(implicit set parameters)

A or B changes, causing Y change | Y changes @

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?

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)

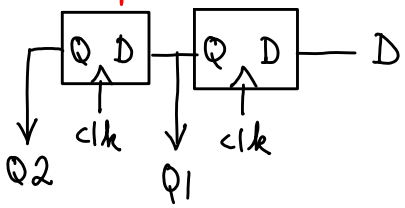
```
initial begin

   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
```

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

always @( posedge clk)  Q2 = Q1;

initial begin

   @( posedge clk )  D = 0;

   @( negedge clk )  D = 1;

end
```

what happens to Q1, Q2?



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
```