Verilog: behavioral modeling, task and function

S. Palnitkar, “***Verilog HDL, A Guide to Digital Design and Synthesis***”, 2nd ed., , Sun Microsystems, Inc, 2003.

Behavioral Modeling

Procedural Assignments

- two structured procedures
  - **always** (SystemVerilog uses **always_ff, always_comb**)
  - **initial**
- procedural assignments update LHS values of
  - **reg** (SystemVerilog uses **logic**)
  - **integer**
  - **real**
  - **time**
- The value placed on a variable remain unchanged until another procedural assignment updates the variable with a different value
  - unlike continuous assignments (**assign**) in dataflow where one assignment can cause the value of RHS expression to be continuously placed on the LHS expression

initial statement

- start at time 0, executes exactly once during a simulation, and then does not execute again.
- If there are multiple initial blocks, each block starts to execute *concurrently* at time 0
- Multiple sequential behavioral statements must be grouped using keywords **begin** and **end**
- typically used for initialization (for simulation, *not for design*), monitoring, waveforms, …
  - Initial statement is NOT synthesizable, cannot be used to initialize values of DFF (instead, use synchronous or asynchronous set/reset, to be discussed later)

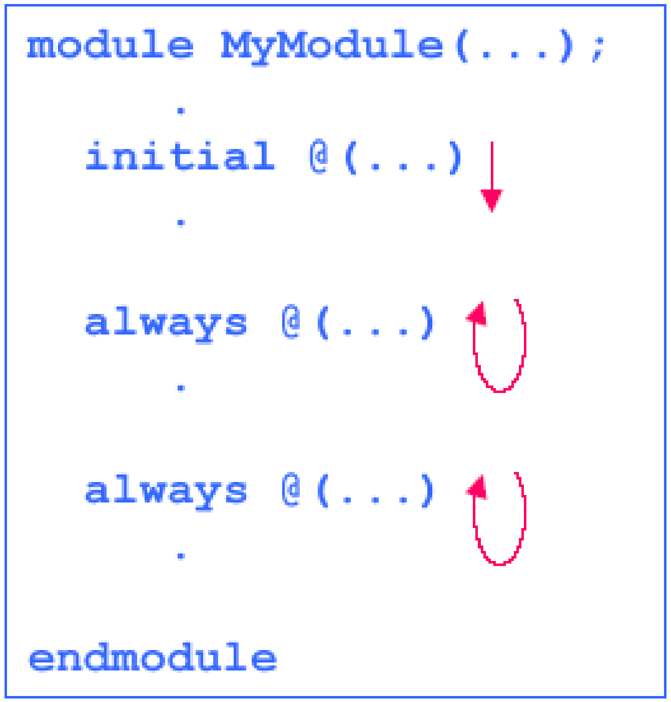
always statement

- starts at time 0 and executes the statements in the always block continuously in a looping fashion
- used to model a block of activity that is repeated continuously in a digital circuit

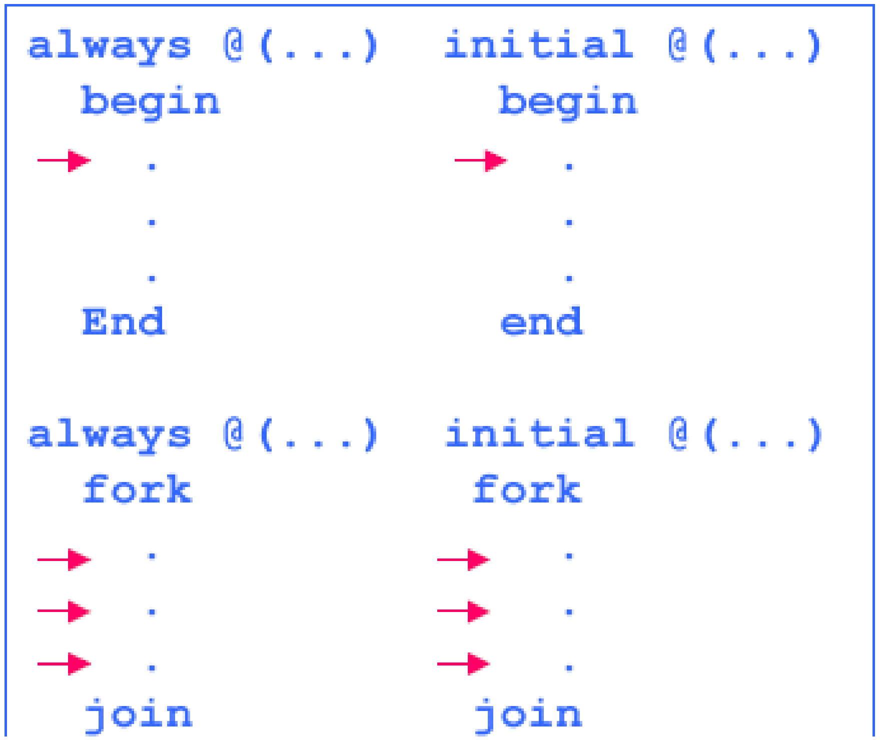
```
module clock_gen (output reg clock);
  initial clock = 1'b0;
  always #10 clock = ~clock;
  initial #1000 $finish;
endmodule
```

behavioral modeling

- Event-driven procedures: **always, initial**
- Sequential blocks: **begin...end**
- Parallel blocks: **fork...join**



```
module MyModule(...);
        .
   initial @(...)
        .

   always @(...)
        .

   always @(...)
        .

endmodule
```



```
always @(...)   initial @(...)
  begin            begin
      .                .
      .                .
      .                .
  End              end

always @(...)   initial @(...)
  fork           fork
      .                .
      .                .
      .                .
  join           join
```

behavioral statements (used inside **initial** or **always** blocks)

- procedural assignments
  - blocking (**=**) vs. non-blocking (**<=**)
- conditional statements (**if … else**…)
- multi-way branching (**case … endcase**)
- looping statement
  - **while, repeat, for, forever**
- sequential and parallel blocks
  - sequential block (**begin … end**)
  - parallel block (**for … join**)
- timing control (**#**)

Blocking Assignments (=)

- executed in the order they are specified in a sequential block

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

initial begin
        x=0;  y=1;  z=1;                          // scalar assignments
        count = 0;
        reg_a = 16'b0;   reg_b = reg_a;  // reg_b = 16`b0
#15   reg_a[2] =1'b1;                         // bit select assignment at time=15
# 10  reg_b[15:13] = {x,y,z};              // at time=25
        count = count +1;                     // at time=25
end
```

Combinational logic using blocking statements (=)

- it is reasonable to describe *combinational logic* using blocking statements
  - model cascade of combinational logic gates

```
always @(in1 or in2)
begin
   a = in1 & in2;      // no assign keyword
   b = in1 | in2;
   {cout, sum} = a + b;
end
```

Nonblocking Assignments (<=)

- allow scheduling of assignments without blocking execution of the statements that follow in a sequential block
  - typically, nonblocking assignments are executed last in the time step which they are scheduled, i.e., after all blocking assignments in the time step are executed

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
    x=0; y=1; z=1;                    // scalar assignments
    count = 0;
    reg_a = 16'b0; reg_b = reg_a;

    // the above statements are executed sequentially, but at time=0
    // the following statements are executed in parallel after
    reg_a[2]      <= #15 1'b1;        // bit select assignment at time=15
    reg_b[15:13] <= #10 {x,y,z};      // at time=10
    count        <= count +1;         // at time =0
end
```

Application of nonblocking assignments

- modeling concurrent data transfers that take place after a common event

```
always @(posedge clock)
begin
   reg1 <= #1 in1;
   reg2 <= @(negedge clock) in2^in3;
   reg3 <= #1 reg1;     // the old value of reg1
end
```

1. a read operation is performed on each right-hand-side (RHS) variable at positive edge of the clock
2. the write operations to the left-hand-side variables (LHS) are scheduled to be executed according to delay control
3. the write operations are executed at the scheduled time

4-bit Shift register using nonblocking statements

- each flip-flop shift to its neighbor, except for the two boundary FFs that acts as input and output registers

```
always @(posedge clock)
begin
   reg1 <= d;
   reg2 <= reg1;
   reg3 <= reg2;
   out <= reg3;
end
```

nonblocking assignments avoid race condition

```
// two concurrent always blocks with blocking statements
always @(posedge clk)
a = b;
always @(posedge clk)
b = a;
// either a=b are executed before b=a, or vice versa
// both registers will get same value (previous value of a or b)
```

```
// two concurrent always blocks with nonblocking statements
always @(posedge clk)
a <= b;
always @(posedge clk)
b <= a;
// at posedge clk, the values of all RHS variables are read,
// and stored in temporary variables, waiting to be evaluated
// reg a and reg b swap value
```

emulate nonblocking using blocking

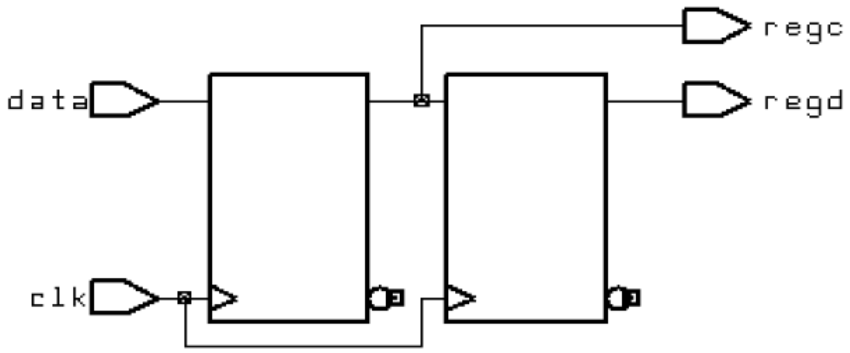
```
always @(posedge clk)
begin
// read operations
    temp_a = a;
    temp_b = b;
// write operations
    a = temp_b;
    b = temp_a;
end
```

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

synthesis of non-blocking and blocking

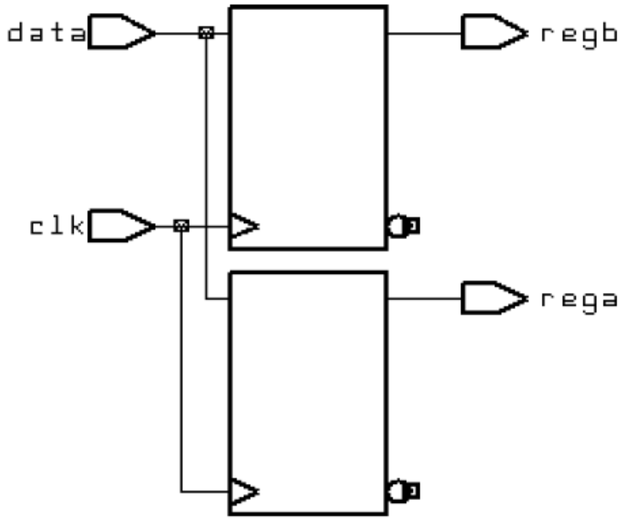
- non-blocking assignment (**<=**)

```
module rtl (clk, data, regc, regd);
input data, clk;
output regc, regd;

reg regc, regd;

always @(posedge clk)
begin
     regc <= data;
     regd <= regc;
end
endmodule
```



- blocking assignment (**=**)

```
module rtl (clk, data, rega, regb);
input data, clk;
output rega, regb;

reg rega, regb;

always @(posedge clk)
begin
     rega = data;
     regb = rega;
end
endmodule
```



Some rules of coding for **<=** and **=**

- In a procedural block (say **always**), never mix blocking (=) and non-blocking (<=) assignments
- For complicated *combinational* logic, use blocking assignments **=**
- Use non-blocking assignments **<=** for clocked *sequential* logic (with results stored in DFF)
- Never assign any data object in a module from more than one **always** block
  - Avoid race condition in *multiple assignment*
- DO NOT mix level and edge events in the sensitivity list of a procedural block
  - e.g. always @ (posedge clk, negedge reset_n) is good, but always @ (posedge clk, reset_n) is NOT good.

regular delay control (**#T c=a+b;**)

```
parameter latency = 20;
reg x, y, z, p, q;
begin
x = 0 ; // no delay
#10 y=1; // delay execution of y=1 by 10 units
#(latency) z=0;
#(4:5:6) q=0; // min, typical and max delays
end
```

Intra-assignment delays (c = #T a+b;)

```
reg x, y, z;

initial begin
x=0; z=0;    // initialization of x and z at time=0
y= #5 x+z; // take values of x and z at time=0, evaluate x+z,
                    // and then wait for 5 time units to assign the value of x+z to y
end

/* equivalent method with temporary variables and regular delay control */
initial begin
x=0; z=0;
temp_xz = x+z;  // x+z is executed at time=0
#5 y=temp_xz;   // y is assigned the value of x+z at time=5
end
```

regular event control

```
@(clk) q=d;
// q=d is executed when clk changes value

@(posedge clk) q=d;
// q=d is executed whenever clk does a
// positive transition (i.e., positive edge)

q = @(posedge clk) d;
// d is evaluated immediately,
// and assigned to q at the positive edge of clk
```

named event control

```
// define an event called received_data
event received_data;
always @(posedge clk)
begin
// an event is triggered by the symbol ->
    if (last_data_packet) -> received_data;
end

// await triggering of event received_data
always @(received_data)
    data_buf={packet[0], packet[1], packet[2]};
```

event OR control (sensitivity list)

```
// level-sensitive latch with asynchronous reset
always @ (rst or clk or d) // use or operator
begin
   if (rst) q=1’b0;                // if rst=1, set q to 0
   else if (clk) q=d;              // if clk is high, latch input
end
```

```
always @ (rst, clk, d)      // use comma (,) instead of or (IEEE 1364-2001)
begin
  if (rst) q=1’b0;
  else if (clk) q=d;
end
```

```
// a positive edge triggered D flip-flop with asynchronous falling reset
always @ (posedge clk, negedge rst_n)
if ( !rst_n) q <= 0;
else     q <= d;
```

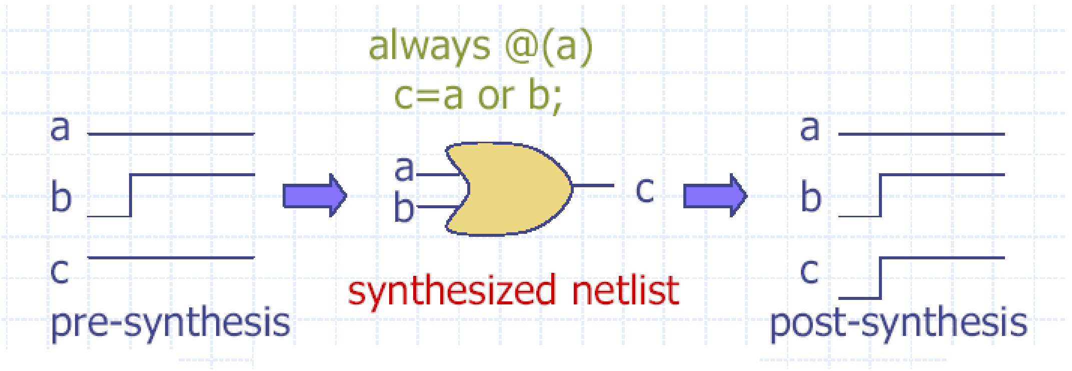
Use of @*, or @(*)

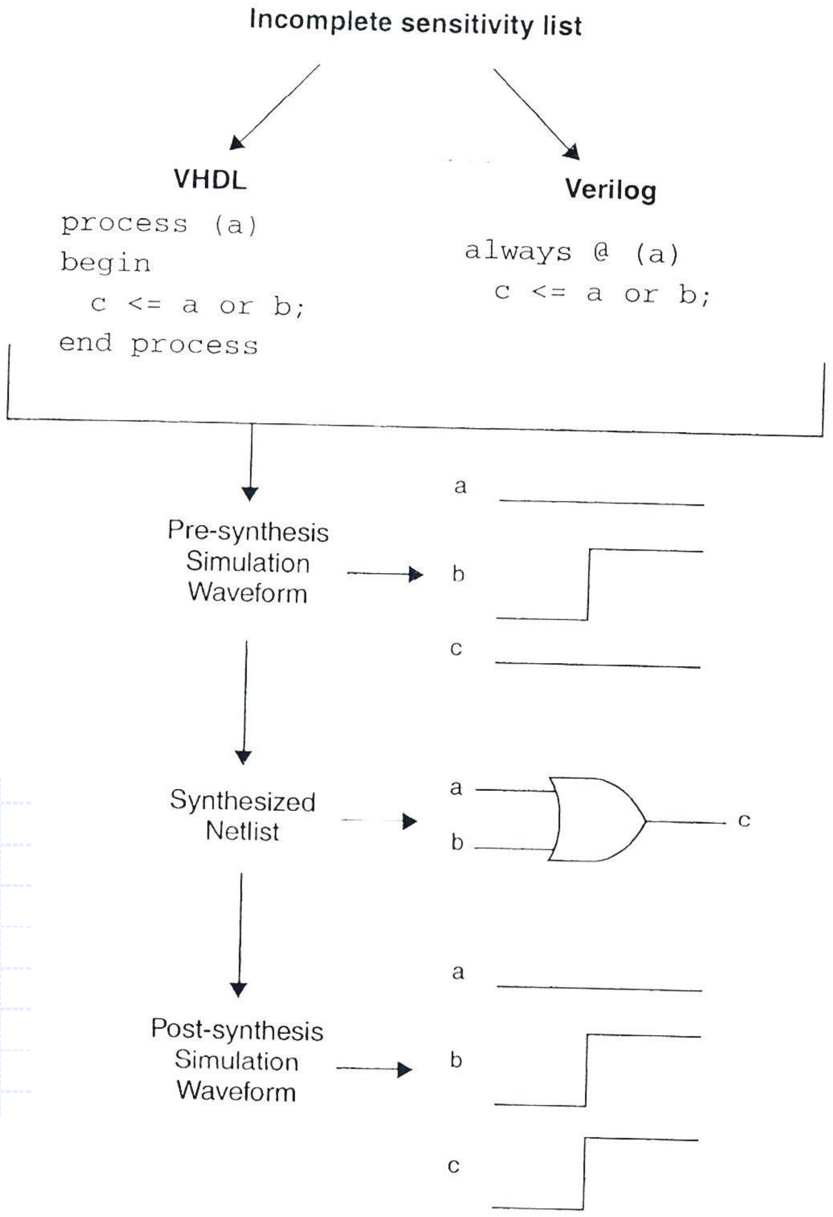
```
always @(a or b or c or d or e or f or g or h or p or m)
begin
out1 = a ? b+c: d+e;     // out1, out2 should be reg
out2 = f ? g+h: p+m;
end
```

```
// all input variables are included automatically
always @(*)   // IEEE 1364-2001
begin
out1 = a ? b+c: d+e;
out2 = f ? g+h: p+m;
end
```

simulation mismatch due to incomplete sensitivity list

- make sure your process sensitivity lists contain only necessary signals
- adding unnecessary sensitivity list slows down simulation




Incomplete sensitivity list
VHDL
Verilog
Pre-synthesis Simulation Waveform
a
b
c
Synthesized Netlist
a
b
c
Post-synthesis Simulation Waveform
a
b
c

```
process (a)
begin
  c <= a or b;
end process
```

```
always @ (a)
  c <= a or b;
```

Level-Sensitive Timing control

```
always
   wait (count_enable) #20 count = count+1;
// wait for count_enable = 1'b1
// count will be incremented (with delay=20)
// only when count_enable is at logic 1

// cp.  @ (event) …
// wait for the change of a signal value, or
// for the triggering of an event
```

conditional statements (**if** … **else** …)

```
if (!lock) buffer = data;
if (enable) out = in;  // infer a latch
```

```
if (number_queued < MAX_Q_DEPTH)
        begin   data_queue = data;
                number_queued = number_queued + 1;
        end
else
        $display (“Queue Full. Try Again”);
```

```
if (alu_control == 0)        y = x + z;
else if (alu_control == 1)   y = x – z;
else if (alu_control == 2)   y = x * z;
else $display (“Invalid ALU control sign”);
```

Verilog vs. VHDL (conditional)

```
// Verilog


if (...)
   begin ... ;  end
else if (…)
   begin …; end
else if (…)
   begin …; end
else
   begin …; end
```

```
-- VHDL


if (…) then
…
elsif (…) then
…
elsif (…) then
…
else
…
end if;
```

Multiway Branching (Case Statements)

```
reg [1:0] alu_control;
…
case (alu_control)
        2'd0     : y = x + z;
        2'd1     : y = x – z;
        2'd2     : y = x * z;
       default : $display("Invalid ALU control signal");
endcase
```

Verilog vs. VHDL (multi-way branching)

```
// Verilog


case ( )
     …     :  begin … end ;
     …     :  begin … end ;
  default :  begin … end ;
endcase;
```

```
-- VHDL


case … is
  when …        => … ;
  when …        => … ;
  when others => … ;
end case;
```

4-to-1 MUX with case

```
// 4-to-1 multiplexer (i1, i2, i3, s[1:0], out)

module mux4 (out, i0, i1, i2, i3, s0, s1);
output out;
input i0, i1, i2, i3, s0, s1;
reg out;

always @(s1, s0, i0, i1, i2, i3)
case ({s1, s0})
    2’d0: out = i0;
    2’d1: out = i1;
    2’d2: out = i2;
    2’d3: out = i3;
    default: $display (“Invalid control signals”);
endcase

endmodule
```

case statement with **x** and **z**

```
// 1-to-4 demultiplexer (in, s[1:0], out1, out2, out3, out4)

reg out0, out1, out2, out3;

always @ (s1 or s0 or in)
case ({s1, s0})
2’b00 : begin out0 = in;      out1 = 1’bz;  out2=1’bz;  out3=1’bz; end
2’b01 : begin out0 = 1’bz;  out1 = in;      out2=1’bz;  out3=1’bz; end
2’b10 : begin out0 = 1’bz;  out1 = 1’bz;  out2=in;      out3=1’bz; end
2’b11 : begin out0 = 1’bz;  out1 = 1’bz;  out2=1’bz;  out3=in;      end
2’bx0, 2’bx1, 2’bxz, 2’bxx, 2’b0x, 2’b1x, 2’bzx :
            begin out0=1’bx;   out1=1’bx;   out2=1’bx;   out3=1b’x; end
2’bz0, 2’bz1, 2b’zz, 2’b0z, 2’b1z :
            begin out0=1’bz;   out1=1’bz;   out2=1’bz;   out3=1b’z; end
default: $display (“unspecified control signals”);
endcase
```

full case and parallel case

- full case
  - all possible branches are specified
  - otherwise, latches are synthesized
- parallel case
  - no cases overlap (one and only one of the branches is executed each time)
  - hardware multiplexers are synthesized in parallel case
  - if not determined, hardware *priority encoder* is synthesized (such as 1xxx, x1xx, xx1x, xxx1)

both full case and parallel case

- a hardware multiplexer is synthesized

```
// full case, and parallel case
// infer multiplexer

input [1:0] a;
always @ (a, w, x, y, z) begin
   case (a)
      2'b11: b=w;
      2'b10: b=x;
      2'b01: b=y;
      2'b00: b=z;
end
```

parallel case, but not full case

- infer a latch for variable b to keep the value in cases that are not specified

```
// parallel case, but not full case
// infer a multiplexer with latched output b

input [1:0] a;
always @ (a, w,  z) begin
   case (a)
      2’b11: b=w;
      2’b00: b=z;
end
```

casex, casez

- **casez**
  - treat all **z** values as don’t care, **z** can also be represented by **?**
- **casex**
  - treat all **z** and **x** values as don’t care

```
reg [3:0] encoding;
integer state;

// not parallel case, but full case,
// infer priority encoder
casex (encoding)
     4’b1xxx : next_state = 3;
     4’bx1xx : next_state = 2;
     4’bxx1x : next_state = 1;
     4’bxxx1 : next_state = 0;
     default : next_state = 0;
endcase
```

4-to-1 multiplexer (4:1 MUX)

- input
  - in0, in1, in2, in3, s[1:0]
- output
  - out

```
reg out;

always @(s, in0, in1, in2, in3)
case (s)
    2’b00: out = in0;
    2’b01: out = in1;
    2’b10: out = in2;
    2’d11: out = in3;
    default: $display (“Invalid control signals”);
endcase
```

1-to-4 demultiplexer (1:4 DEMUX)

- input
  - in, s[1:0]
- output
  - out0, out1, out2, out3

```
reg  out0, out1, out2, out3;

always @(s, in0, in1, in2, in3)
case (s)
    2’b00: begin out0 = in;      out1=1’bz;  out2=1’bz;  out3=1’bz;   end
    2’b01: begin out0 = 1’bz;  out1=in;      out2=1’bz;  out3=1’bz;   end
    2’b10: begin out0 = 1’bz;  out1=1’bz;   out2=in;      out3=1’bz;   end
    2’d11: begin out0 = 1’bz;  out1=1’bz;   out2=1’bz;  out3=in;      end
    default: $display (“Invalid control signals”);
endcase
```

4:2 priority encoder

- input
  - in0, in1, in2, in3
- output
  - out[1:0]

```
reg [1:0] out;

always @(in3, in2, in1, in0)
casex ({in3, in2, in1, in0})
    4'b1xxx: out = 2'b11;
    4'b01xx: out = 2'b10;
    4'b001x: out = 2'b01;
    default  : out = 2'b00;
endcase
```

4:2 encoder

- input
  - in0, in1, in2, in3
- output
  - out[1:0]

```
reg [1:0] out;
reg        valid;

always @(in3, in2, in1, in0)
valid = 1;
case ({i3, i2, i1, i0})
    4'b1000: out = 2'b11;
    4'b0100: out = 2'b10;
    4'b0010: out = 2'b01;
    4'b0001: out = 2'b00;
    default  : begin out = 2'bxx;  valid = 0; end
endcase
```

2:4 decoder

- input
  - – in[1:0]
- output
  - – out0, out1, out2, out3

```
wire  [1:0] in;
reg  out0, out1, out2, out3;

always @(in)
case (in)
    2'b00: begin out0=1;   out1=0;    out2=0;   out3=0;   end
    2'b01: begin out0=0;   out1=1;    out2=0;   out3=0;   end
    2'b10: begin out0=0;   out1=0;    out2=1;   out3=0;   end
    2'd11: begin out0=0;   out1=0;    out2=0;   out3=1;   end
endcase
```

while Loop

```
integer count;

initial
begin
count = 0;
while (count < 128)
     count = count +1;
end
```

```
`define TRUE 1’b1;
`define FALSE 1’b0;
reg [15:0] flag;
integer i;
reg continue;

// find the leading 1 in flag
initial
begin
flag = 16’b 0010_0000_0000_0000
i = 0;
continue = `TRUE;
while ((i<16) && continue)
   begin
      if (flag[i]) continue = `FALSE;
      i=i+1;
   end
end
```

for Loop

```
integer count;

initial
for (count=0; count<128;  count=count+1)
   $display("Count=%d", count);
```

```
`define MAX_STATES 32
integer state[0:`MAX_STATES-1];
integer i;

initial
begin
for (i=0; i<32; i=i+2)
   state[i]=0;      //initialize even locations
for (i=1; i<32; i=i+2)
   state[i]=1;      //initialize odd locations
end
```

repeat Loop

- repeat construct must contain a number

```
integer count;

initial
begin
count=0;
repeat (128)
begin
    count = count+1;
end
end
```

```
module data_buffer(data_start, data, clk);
parameter cycles=8;
input data_start;
input [15:0] data;
input clk;
reg [15:0] buffer[0:7];
integer i;

always @(posedge clk) begin
        if (data_start) begin
                i=0;
                // store data at next 8 clock edges
                repeat (cycles)
                begin
                     // wait till next posedge to latch data
                    @(posedge clk) buffer[i]=data;
                    i=i+1;
                end
      end
end
endmodule
```

forever loop

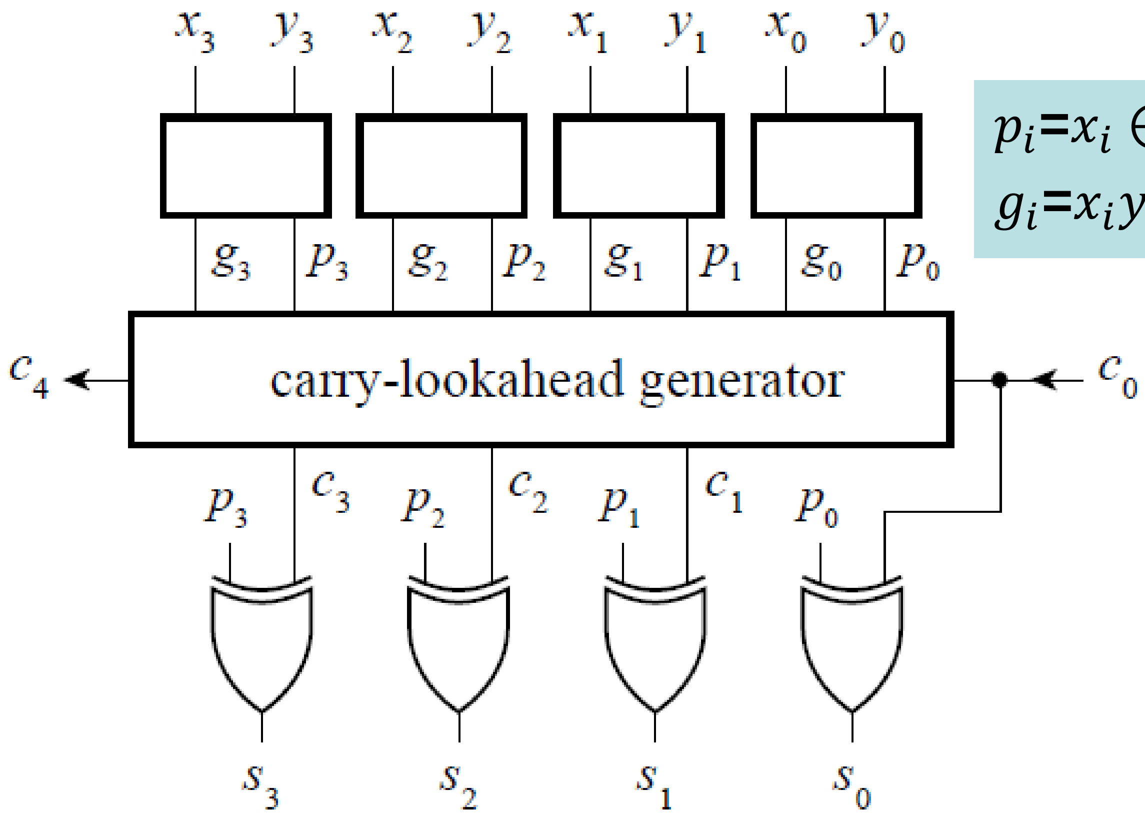
- execute forever until the **$finish** task

```
// clock generation using
// forever loop instead of
// always block
reg clk;

initial
begin
clk=1'b0;
forever #10 clk=~clk;
end
```

```
// synchronize two register values
// at every positive edge of clk
reg clk;
reg x, y;

initial
forever @(posedge clk)  x = y;
```

Example: Carry Lookahead Adder

- 4-bit CLA
  - – propagate p_i
  - – generate g_i



x_3 y_3 x_2 y_2 x_1 y_1 x_0 y_0
g_3 p_3 g_2 p_2 g_1 p_1 g_0 p_0
c_4
carry-lookahead generator
c_0
p_3 c_3 p_2 c_2 p_1 c_1 p_0
s_3 s_2 s_1 s_0

$p_i = x_i \oplus y_i$

$g_i = x_i y_i$

$c_{i+1} = g_i + p_i c_i$

$c_1 = g_0 + c_0 p_0$

$c_2 = g_1 + c_1 p_1 = g_1 + g_0 p_1 + c_0 p_0 p_1$

$c_3 = g_2 + c_2 p_2 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$

$c_4 = g_3 + c_3 p_3 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$

Example (16-bit carry lookahed adder)

$G_{3:0}=g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0$
$P_{3:0} = p_3p_2p_1p_0$

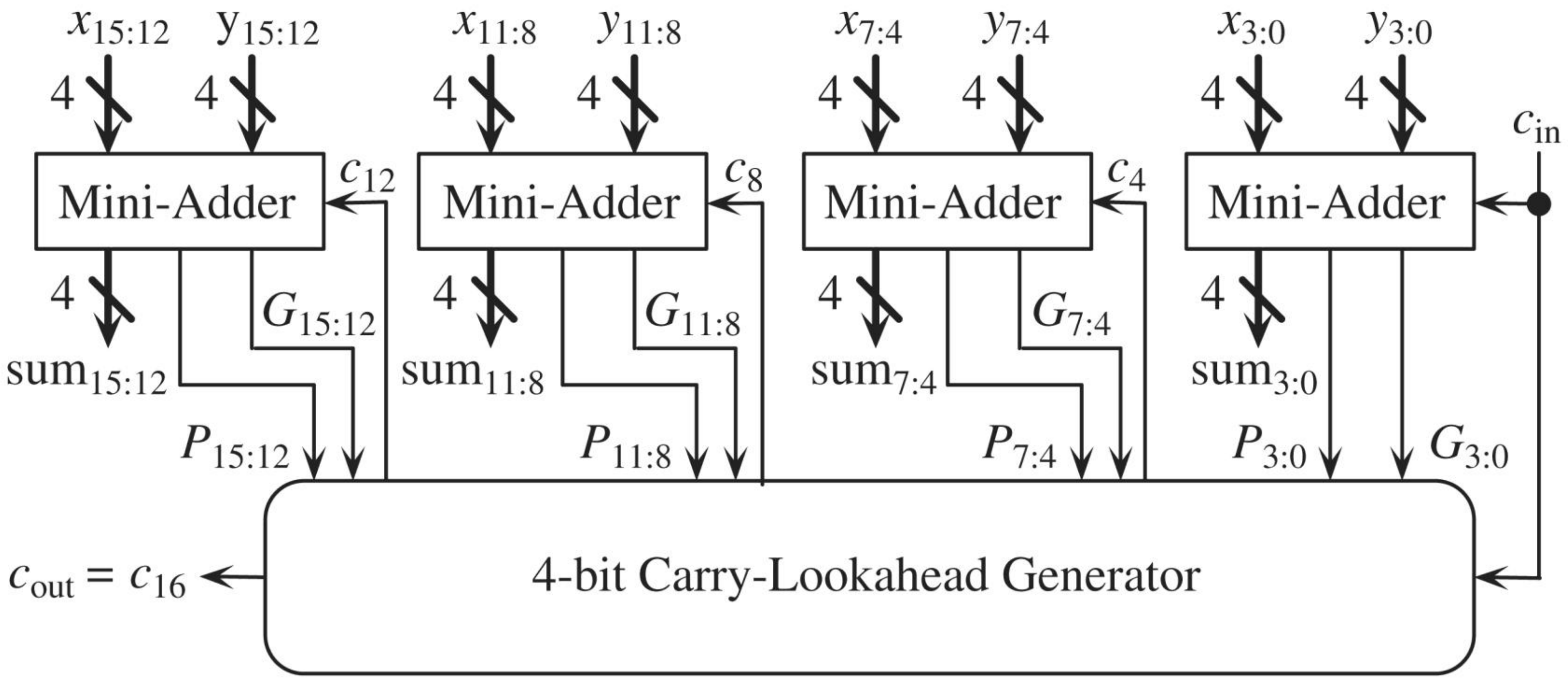
$G_{7:4}=g_7+p_7g_6+p_7p_6g_5+p_7p_6p_5g_4$
$P_{7:4} = p_7p_6p_5p_4$

…

$G_{11:8} = \cdots, P_{11:8}=\ldots$

$G_{15:12} = \cdots, P_{15:12}=\ldots$

$c_4 = G_{3:0} + P_{3:0}c_0$
$c_8 = G_{7:4} + P_{7:4}c_4$
$c_{12} = G_{11:8} + P_{11:8}c_8$
$c_{16} = G_{15:12} + P_{15:12}c_{12}$



Carry Look Ahead (CLA) Adder

propagate and **generate** signals:

$$p_i = x_i + y_i \ (p_i = \overline{x_i \oplus y_i})$$

$$g_i = x_i y_i$$

$$c_{i+1} = g_i + p_i c_i$$

x_i	y_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

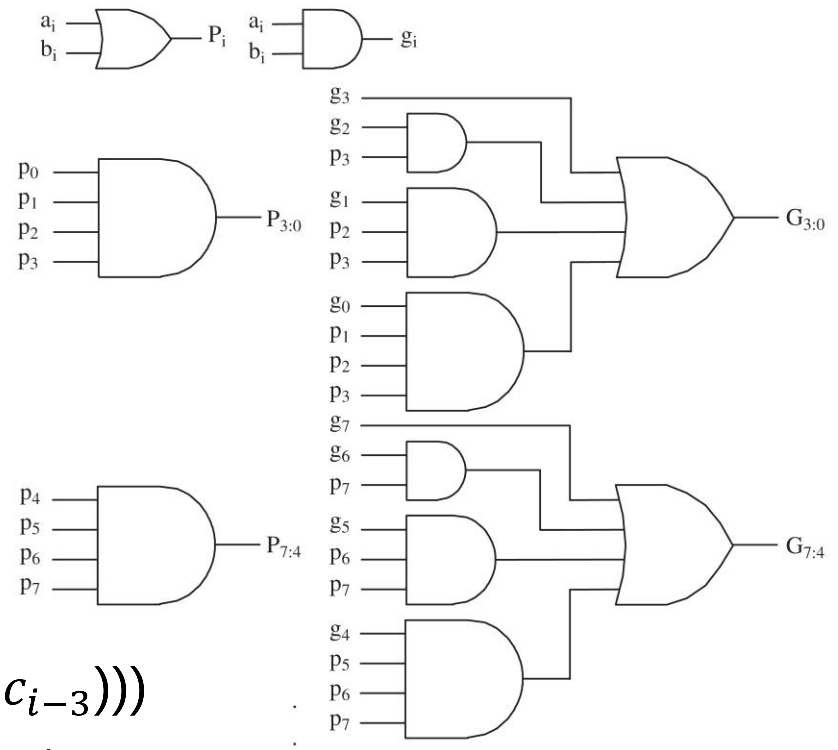
Group propagate and generate:

$$G_{3:0} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

$$P_{3:0} = p_3 p_2 p_1 p_0$$

$$G_{7:4} = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$$

$$P_{7:4} = p_7 p_6 p_5 p_4$$



$$c_i = g_i + p_i c_{i-1}$$

$$= g_i + p_i \left(g_{i-1} + p_{i-1} c_{i-2} \right)$$

$$= g_i + p_i \left(g_{i-1} + p_{i-1} (g_{i-2} + p_{i-2} c_{i-3}) \right)$$

$$= g_i + p_i \left(g_{i-1} + p_{i-1} (g_{i-2} + p_{i-2} (g_{i-3} + p_{i-3} c_{i-3})) \right)$$

$$= \boldsymbol{g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + p_i p_{i-1} p_{i-2} g_{i-3}} + \boldsymbol{p_i p_{i-1} p_{i-2} p_{i-3}} c_{i-3}$$

$$= \boldsymbol{G_{i:i-3}} + \boldsymbol{P_{i:i-3}} c_{i-3}$$

16-bit p_g_generator with 4-bit p_g block (1/2)

```
`define DATA_BITS  16  // number of bits in input and output data
`define NUM_GROUPS 4   // number of stages (or groups) used
`define GROUP_BITS (`DATA_BITS / `NUM_GROUPS)  // bits in 1 stage

// MODULE DECLARATION
module p_g_generator (P, G, a, b);

  output [`NUM_GROUPS-1:0] P;  // group-propagate signals
  output [`NUM_GROUPS-1:0] G;  // group-generate signals
  input [`DATA_BITS-1:0] a, b; // adder inputs

  // SIGNAL DECLARATIONS
  reg [`NUM_GROUPS-1:0] P;     // group-propagate signals
  reg [`NUM_GROUPS-1:0] G;     // group-generate signals
  reg [`DATA_BITS-1:0] p;      // propagate signals
  reg [`DATA_BITS-1:0] g;      // generate signals
  integer i, j;                // indices used in for loops
```

16-bit p_g_generator with 4-bit p_g block (2/2)

```
// always block used to model combinational logic
always @ (a or b) begin
    for (i = 0;  i < `NUM_GROUPS;  i = i + 1) begin
    P[i] = 1;
    G[i] = 0;

    for (j = `GROUP_BITS-1;  j >= 0;  j = j - 1) begin
        p[i*`GROUP_BITS+j] = (a[i*`GROUP_BITS + j] | b[i*`GROUP_BITS + j]);
        g[i*`GROUP_BITS+j] = (a[i*`GROUP_BITS + j] & b[i*`GROUP_BITS + j]);
        G[i] = G[i] | (g[i*`GROUP_BITS+j] & P[i]);
        P[i] = P[i] & p[i*`GROUP_BITS+j];
    end

    end
end

endmodule
```

$G_{3:0}=g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0$

$P_{3:0}=p_3p_2p_1p_0$

$G_{7:4}=g_7+p_7g_6+p_7p_6g_5+p_7p_6p_5g_4$

$P_{7:4}=p_7p_6p_5p_4$

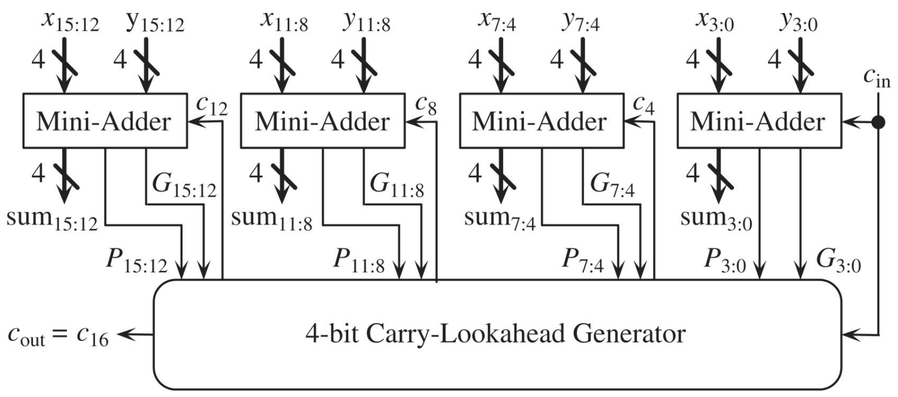
…

$G_{11:8}=\cdots, P_{11:8}=\ldots$

$G_{15:12}=\cdots, P_{15:12}=\ldots$

16-bit Hierarchical CLA (1/3)

```
`define DATA_BITS  16  // number of bits in input and output data
`define NUM_GROUPS 4    // number of stages (or groups) used
`define GROUP_BITS (`DATA_BITS / `NUM_GROUPS)  // bits in 1 stage

// MODULE DECLARATION
module hierarchical_cla_adder (sum, cout, x, y, cin);

  output [`DATA_BITS-1:0] sum; // the resulting sum value
  output cout;                  // carry-out bit
  input [`DATA_BITS-1:0] x, y; // the two numbers to be added
  input cin;                    // carry-in to the adder

  // Declaration of "reg" outputs
  reg [`DATA_BITS-1:0] sum;
  reg cout;

  // Declaration of internal signals
  reg [`DATA_BITS:0] carry;     // intermediate carry signals
  reg [`NUM_GROUPS-1:0] temp;  // temporary signal
  wire [`NUM_GROUPS-1:0] GP;    // group propagate signals
  wire [`NUM_GROUPS-1:0] GG;    // group generate signals
  integer i, j;                 // indices used in for loops
```

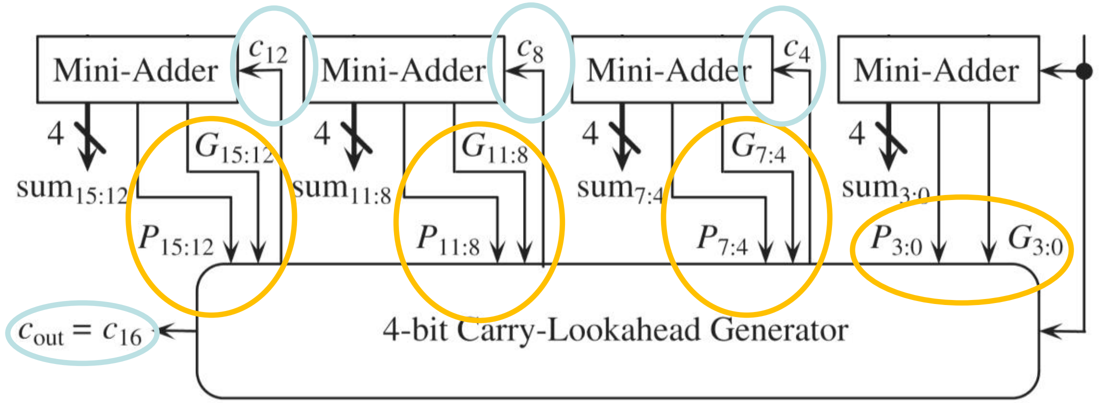


16-bit Hierarchical CLA (2/3)

```
// use a lower-level block for GP[i] and GG[i] signals
 p_g_generator UNIT0 (GP, GG, x, y);

 // create the carry-lookahead logic and ripple-carry adders

 always @ (x or y or GP or GG or cin) begin
   // first describe carry-lookahead generation block
   carry[0] = cin;
   for (i = 0;  i < `NUM_GROUPS;  i = i + 1) begin
     carry[(i+1)*`GROUP_BITS] = GG[i];
     carry[(i+1)*`GROUP_BITS] = carry[(i+1)*`GROUP_BITS] | (GP[i] & carry[i*`GROUP_BITS]);
   end  // of for (i)
  cout = carry[`NUM_GROUPS * `GROUP_BITS];
```

$G_{3:0}\ ,P_{3:0}$
$G_{7:4}, P_{7:4}$
$G_{11:8}, P_{11:8}$
$G_{15:12}, P_{15:12}$

$$c_4 = G_{3:0} + P_{3:0}c_0$$
$$c_8 = G_{7:4} + P_{7:4}c_4$$
$$c_{12} = G_{11:8} + P_{11:8}c_8$$
$$c_{16} = G_{15:12} + P_{15:12}c_{12}$$



16-bit Hierarchical CLA (3/3)

```
// next, generate the ripple-carry adder stages

for (i = 0;  i < `NUM_GROUPS;  i = i + 1) begin
     for (j = 0;  j < (`GROUP_BITS-1);  j = j + 1) begin
       {carry[i*`GROUP_BITS + j + 1], sum[i*`GROUP_BITS + j]} =
             x[i*`GROUP_BITS + j] + y[i*`GROUP_BITS + j] +
             carry[i*`GROUP_BITS + j];
     end  // of for (j)
     {temp[i], sum[(i+1)*`GROUP_BITS - 1]} =
           x[(i+1)*`GROUP_BITS - 1] + y[(i+1)*`GROUP_BITS - 1] +
           carry[(i+1)*`GROUP_BITS - 1];
end  // of for (i)

end  // of always block
endmodule
```

// use 4-bit RCA

$\{c_1, s_0\} = x_0 + y_0 + c_0$

$\{c_2, s_1\} = x_1 + y_1 + c_1$

$\{c_3, s_2\} = x_2 + y_2 + c_2$

$\{c_5, s_4\}, \{c_6, s_5\}, \{c_7, s_6\}$

$\{c_9, s_8\}, \{c_{10}, s_9\}, \{c_{11}, s_{10}\}$

$\{c_{13}, s_{12}\}, \{c_{14}, s_{13}\}, \{c_{15}, s_{14}\}$

// c_4, c_8, c_{12}, c_{16} been calculated

$\{temp, s_3\} = x_3 + y_3 + c_3$

$\{temp, s_7\} = x_7 + y_7 + c_7$

$\{temp, s_{11}\} = x_{11} + y_{11} + c_{11}$

$\{temp, s_{15}\} = x_{15} + y_{15} + c_{15}$

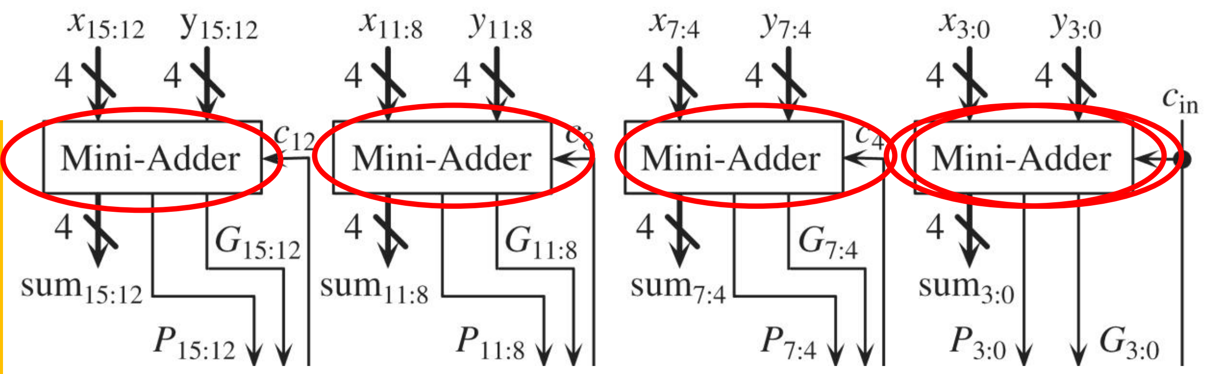
// use 4-bit CLA

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_{i-2}$

$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

…



Sequential Blocks (**begin ... end**)

- the statements are processed in the order they are specified
  - except for *non-blocking assignments* (**<=**) with intra-assignment timing control
- delay or event control is relative to the simulation time when the *previous statement* completed execution

Examples (Sequential Block)

```
// sequential block
// without delay
reg x, y;
reg [1:0] z, w;
initial
begin
     x=1'b0;
     y=1'b1;
     z={x,y};  // z=01
     w={y,x}; // w=10
end
```

```
// sequential block with delay
reg x, y;
reg [1:0] z, w;
initial
begin
           x=1'b0;   // complete at time=0
     #5    y =1'b1;  // complete at time=5
     #10 z={x,y};   // complete at time=15
     #20 w={y,x};  // complete at time=35
end
```

Parallel Blcoks (**fork ... join**)

- statements are executed concurrently
- ordering of statements is controlled by the delay or event control
- delay or event control is relative to the time the block was entered

Examples (Parallel Blocks)

```
reg x, y;      reg [1:0] z, w;
initial
fork
            x=1'b0;        // complete at time=0
      #5   y=1'b1;        // complete at time=5
      #10 z={x,y};        // complete at time=10
      #20 w={y,x};        // complete at time=20
join
```

```
// race condition, values of z and w are non-deterministic
reg x, y;   reg [1:0] z, w;
initial
fork
      x=1'b0;
      y=1'b1;
      z={x,y};  // z is unknown
      w={y,x};  // w is unknown
join
```

Named Block

```
module top;

initial
begin: block1     // sequential block named block1
   integer i;          // integer i is static and local to block1,
                       // can be accessed by hierarchical name, top.block1.i
   ....
end

initial
fork: block2     // parallel block named block2
   reg  i;             // register i is static and local to block 2
                       // can be accessed by hierarchical name, top.block2.i
   ...
join

endmodule
```

Disabling Named Block

```
reg [15:0] flag;
integer i;
initial
begin
flag = 16'b 0010_0000_0000_0000;
i = 0;

begin: block1
while (i<16) begin
     if (flag[i])  begin
          $display("encountered a TRUE bit at element number %d", i);
          disable blcok1;
     end
     i=i+1;
end
end

end
```

generate blocks

- allow Verilog code to be generated *dynamically* at the elaboration time (before simulation)
- help the creation of parameterized models
- applications
  - same operation or module instance is repeated
    - eg., 64-b ripple carry adder (cascaded of 64 FA cells)
  - certain Verilog code is conditionally included based on parameter definitions
    - eg., Booth or non-Booth multiplier based on bit-width

generate loop

- allow multiple instantiation
- note that generate should be used outside of procedural blocks (**always**, **initial**)

```
module bitwise_xor(out, i0, i1);
parameter N=32;
output [N-1:0] out;
input [N-1:0] i0, i1;

genvar j; // var. j is local
generate
        for (j=0; j<N; j=j+1)
             begin: xor_loop
                 xor g(out[j], i0[j], i1[j]);
             end
// hierarchical name referencing:
// xor_loop[0].g, xor_loop[1].g, ...,
endgenerate

endmodule
```

```
// alternate style (behavioral level)
reg [N-1:0]  out;

genvar j;
generate
        for (j=0; j<N; j=j+1)
        begin: bit
                always @ (i0[j] or i1[j])
                        out[j] = i0[j] ^ i1[j];
        end
endgenerate
```

Discussion of generate block

- before simulation, simulator elaborates (unrolls) the code in the generate blocks
- then, the unrolled code is simulated
  - a convenient way of replacing multiple repetitive Verilog statements with a single statement inside a loop
- **genvar** is to declare variables used in generate block, do not exist in simulation
- hierarchical name referencing:

  xor_loop[0].g, xor_loop[1].g, ..., xor_loop[31].g

generated ripple adder

```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
wire [N-1:0] carry;

assign carry[0]=ci;

genvar i;
generate
        for (i=0; i<N; i=i+1)
        begin: r_loop
                wire t1, t2, t3;
                xor g1(t1, a0[i], a1[i]);
                xor g2(sum[i], t1, carry[i]);
                and g3(t2, a0[i], a1[i]);
                and g4(t3, t1, carry[i]);
                or g5(carry[i+1], t2, t3);
        end
endgenerate

assign co = carry[N]

endmodule
```

generate conditional

```
module multiplier (product, a0, a1);
// 8-bit bus by default,
// could be changed during instantiation using defparam or #
parameter a0_width = 8;
parameter a1_width = 8;

// the following parameter cannot be modified using defparam
localparam product_width = a0_width + a1_width;

output [product_width-1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;

generate
	if (a0_width < 8) || (a1_width < 8)
	   cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
	else
	   tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate
endmodule
```

generate case

```
module adder (co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
generate
     case (N)
                  1 :  adder_1bit        adder1(co, sum, a0, a1, ci);
                  2 :  adder_2bit        adder2(co, sum, a0, a1, ci);
        default :  adder_cla #(N) adder3(co, sum, a0, a1, ci);
     endcase
endgenerate
endmodule
```

Tasks and Functions

Tasks vs. Functions

- task is similar to subroutine in FORTRAN
- function is similar to function in FORTRAN

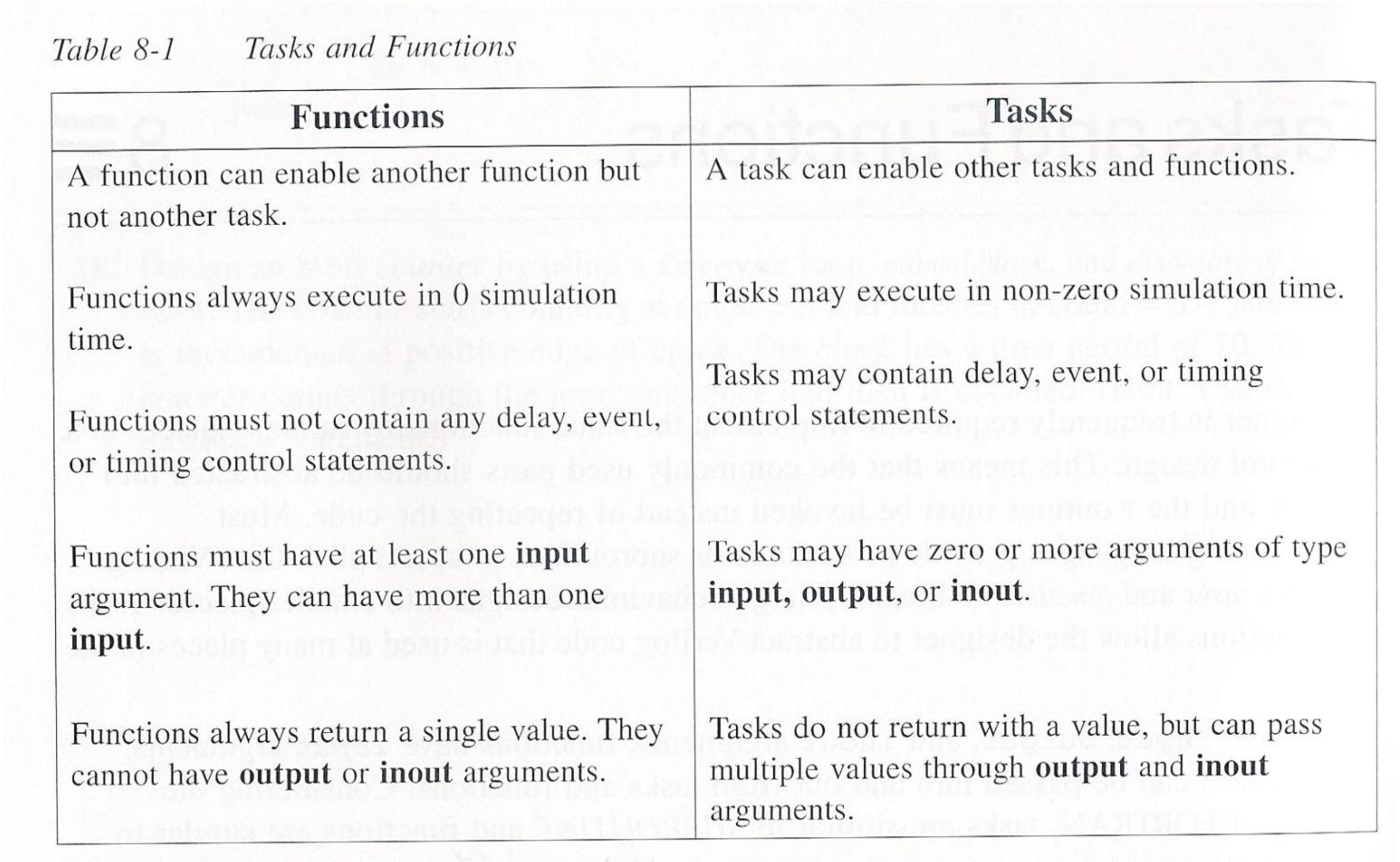


Table 8-1 Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output**, or **inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value, but can pass multiple values through **output** and **inout** arguments.

Tasks vs. Functions

- defined in a module and local to the module
  - contain *behavioral statements only*
  - called from **always** or **initial** blocks, or other tasks and functions
- tasks are used for commonly Verilog code
  - contain delays, timing, event constructs, or multiple output arguments
  - can have input, output and inout arguments
- functions are used for code that
  - is purely **combinational**
  - executes in zero simulation time
  - provides exactly one output
  - can have input arguments

Tasks (task … endtask)

```
module operation;
…
parameter delay=10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;
always @ (A or B)
begin
bitwise_oper(AB_AND, AB_OR, AB_XOR,
    A, B);
end
…
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor;
input [15:0] a, b;
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
end task
…
endmodule
```

```
// another task declaration
//
task bitwise_oper (
output [15:0] ab_and, ab_or, ab_xor,
input [15:0] a, b);
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
endtask
```

Example: Asymmetric Sequence Generator

```
module sequence;
reg clk;
initial  init_sequence;                    // invoke task init_sequence
always asymmetric_sequence; // invoke another task

task init_sequence; begin  clk= 1'b0; end endtask

task asymmetric_sequence; begin
#12 clk = 1'b0;
#5 clk = 1'b1;
#3 clk = 1'b0;
#10 clk = 1'b;
end endtask

endmodule
```

Automatic (Re-entrant) Tasks

```
// all normally declared task items are statically allocated and shared.
// automatic task allows for dynamic allocation for each invocation
// where each task call operates in an independent space.

module top;
reg [15:0] cd_xor, ef_xor;
reg [15:0] c, d, e, f;

task automatic bitwise_xor;
output [15:0] ab_xor;
input [15:0] a, b;
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end endtask

// two tasks are called concurrently in two procedural blocks
always @(posedge clk)   bitwise_xor (ef_xor, e, f);
always @(posedge clk2) bitwise_xor (cd_xor, c, d);  // clk2 twice frequency
endmodule
```

Function (**function** … **endfunction**)

```
module parity;
reg [31:0] addr;
reg parity;
always @(addr)
   parity = calc_parity(addr);

function calc_parity;
input [31:0] address;
begin
calc_parity = ^address;
end
endfunction

endmodule
```

```
// alternate function definition
function calc_parity
(input [31:0] address);
begin
calc_parity = ^address;
end
endfunction
```

Example: Left/Right Shifter

```
module shifter;
‘define LEFT_SHIFT 1’b0;
‘define RIGHT_SHIFT 1’b1;
reg [31:0] addr, left_addr, right_addr;
reg control;
always @(addr)
begin
left_addr = shift (addr, ‘LEFT_SHIFT);
right_addr = shift (addr, ‘RIGHT_SHIFT);
end

function [31:0] shift; // function output is a 32-bit value
input [31:0] address;
input control;
begin
shift = (control == ‘LEFT_SHIFT) ? (address << 1) : (address >>1);
end
endfunction

endmodule
```

Automatic (Recursive) Functions

```
// functions are normally used non-recursively
// automatic function allows all function declarations allocated dynamically
// for each recursive call
// each call to an automatic function operates in an independent variable space.

module top;

function automatic integer factorial;
input [31:0]  oper;
integer i;
begin
if (oper >=  2 ) factorial = factorial(oper-1) * oper;
else factorial = 1;
end endfunction

integer result;
initial  result = factorial(4);

endmodule
```

Constant Function

```
module ram (…);
parameter RAM_DEPTH = 256;
input [ clogb2(RAM_DEPTH) – 1 : 0] addr_bus;

function integer clogb2 (input integer depth);
begin
for (clogb2=0; depth >0; clogb2=clogb2+1)
depth = depth >> 1;
end endfunction

endmodule
```

signed function

```
module top;
...
function signed [63:0] compute_signed
(input [63:0] vector);
...          // returned function value is signed
endfunction
…
if (compute_signed(vector) < -3)
begin ... end
...
endmodule
```