14. Introducton to Verilog

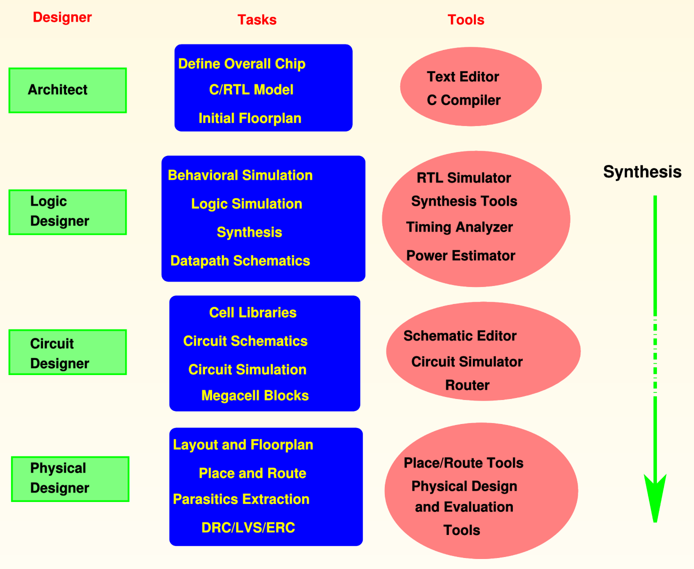
Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin

VLSI Design
Fall 2020

October 15, 2020

Synthesis in the Design Flow



Logic Synthesis

- Design described in a Hardware Description Language (HDL)
  - Verilog, VHDL
- Simulation to check for correct functionality
  - Simulation semantics of language
- Synthesis tool
  - Identifies logic and state elements
  - Technology-independent optimizations (state assignment, logic minimization)
  - Map logic elements to target technology (standard cell library)
  - Technology-dependent optimizations (multi-level optimization, gate strengths, etc.)

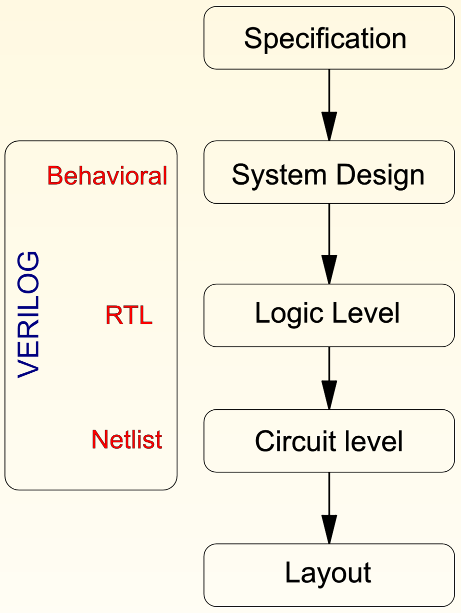
Features of Verilog

- A **concurrent** language (syntax similar to C)
- Models hardware
- Provides a way to specify concurrent activities
- Allows timing specifications
- **Originally developed by Gateway Design Automation, acquired by Cadence, then became IEEE Standard 1364-1995 (Open Verilog International)**
- **Updated, now Verilog 2005 (IEEE Standar 1364-2005)**
- **System Verilog 2009 (superset of Verilog), (IEEE Standard 1800-2009)**

Applications of Verilog

- Description of design at a higher level
- Development of formal models
- System documentation
- Simulation to uncover errors (bugs) in design
- Synthesis of designs
- Design reuse

Design Process Cycle

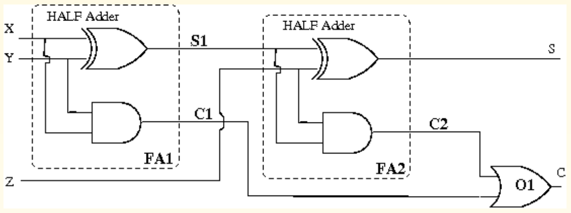


Verilog Modes

1. **Structural:** describes the structure of the hardware components, including how ports or modules are connected together
   - module contents are built in gates (`and, or, xor, not, nand, nor, xnor, buf`) or other modules previously declared
2. **Behavioral:** describes what should be done in a module
   - module contents are C-like assignment statements, loops

Structural Model of Adder

Structural models: interconnections of primitive gates (AND, OR, NAND, NOR, etc.) and other modules



```
module HA(a, b, s, c);
input a,b;
output s,c;

xor G1(s, a, b);
and G2(c, a, b);

endmodule
```

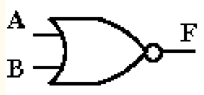
```
module FA(X, Y, Z, S, C);
input X, Y, Z;
output S, C;

HA FA1(X, Y, S1, C1);
HA FA2(S1, Z, S, C2);
or O1(C, C2, C1);

endmodule
```

Specification of Gate-Level Model

- Gate function
  - Verilog built-in: `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `buf`, `not`, `etc.`
- Gate specification format example:
  - `and #delay inst-name (out, in1, in2,..., ink);`
  - delay and inst-name are optional

Logic Values

- A bit can have any of these values
  - 0 representing logic low (false)
  - 1 representing logic high (true)
  - X representing either 0, 1, or Z
  - Z representing high impedance for tri-state
  - (unconnected inputs are set to Z)

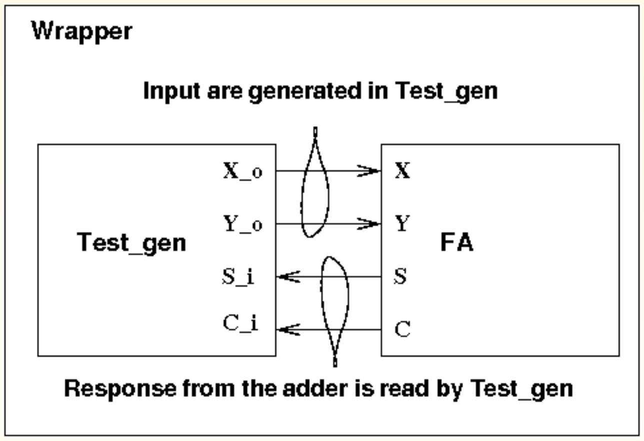


		B			
		0	1	x	z
A	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

F

Test Generation

- Test_gen: Generates test and captures the response
- FA: The design under test
- Wrapper: Wraps Test_gen and FA into one module


Wrapper
Input are generated in Test_gen
Test_gen
X_o
Y_o
S_i
C_i
X
Y
S
C
FA
Response from the adder is read by Test_gen

Test Generation, Cont'd

```
module HA(a,b,s,c};
input a,b;
output s,c;

xor G1
and G2(c,a,b);
endmodule

module FA(X,Y,Z,S,C);
input X,Y,Z;
output S,C;

HA FA1(X,Y,S1,C1);
HA FA2(S1,Z,S,C2);
or O1(C,C2,C1);
endmodule
```

```
module test_gen(a,b,c,sum,cout);
output a,b,c;
input sum,cout;
reg a,b;
initial begin
$monitor($time," A=%b B=%b Sum=%b Cout=%b",
                     a,b,sum,cout);
a = 0; b = 0;
#5 a = 1;
#5 b = 1;
#5 c = 1;
end
endmodule
```

```
module wrapper;
wire a,b,sum,cout;

FA(a,b,c,sum,cout);
test_gen(a,b,c,sum,cout);
endmodule
```

`initial`

- tells Verilog to execute all statements within `begin` and `end` once it starts

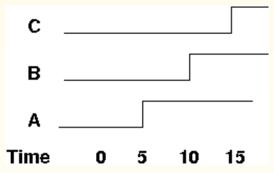
`monitor`

- tells Verilog to monitor the list of variables and every time a variable changes, it prints the string

Test Generation, Cont'd

```
module test_gen(a,b,c,sum,cout);
output a,b,c;
input sum,cout;
reg a,b;
initial begin
$monitor($time," A=%b B=%b Sum=%b Cout=%b",
                        a, b, sum, cout);
a = 0; b = 0; c = 0;
    #5 a = 1;
    #5 b = 1;
    #5 c = 1;
end
endmodule
```

```
module wrapper
   wire a, b, sum, cout;

   FA(a,b,c,sum,cout);
   test_gen(a,b,c,sum,cout);
endmodule
```



Behavioral Modeling

- Behavioral model consists of `always` and `initial` constructs
- All behavioral statement must be within these blocks
- Many `initial/always` statements can exist within a module

`initial` constructs execute once at the start of the simulation

```
initial
   begin
      <statements>
   end
```

`always` constructs execute at the beginning of the simulation and continually loop

```
always @(sensitivity-list)
   begin
      <statements>
   end
```

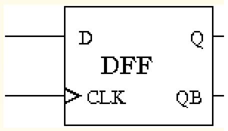
Behavioral Timing

- Advance time when
  - #20 delay 20 time units
  - @(list) delay until an event occurs
  - wait: delay until a condition is satisfied

```
@r rega = regb;  // load rega when r changes

@(posedge r) rega = regb;
                 // load rega on positive edge of r

@(negedge r) rega = regb;
                 // load rega on negative edge of r

wait(!r) rega = regb;
                 // execution is suspended until r is 0
```

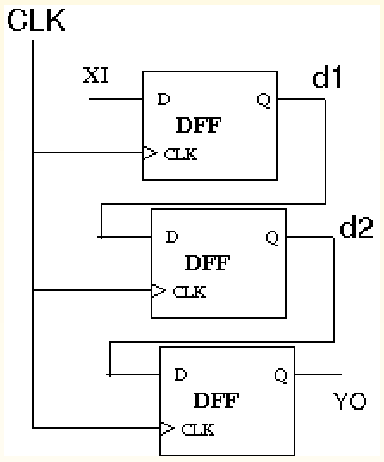
Edge Triggered D-Flip-Flop

```
module dff(q,qb,d,clk);
input d,clk;
output q,qb;

reg q;

always @(posedge clk)
   begin
      q = d;
        // left hand side must be a register
   end

not G1(qb,q);

endmodule
```



Shift Register

```
module dff(q,d,clk);
input d,clk;
output q;
reg q;
   always @(posedge clk)
      q=d;
endmodule

module shift(YO,X1,CLK);
   input CLK, X1;
   output YO;

   dff(d1,X1,CLK);
   dff(d2,d1,CLK);
   dff(YO,d2,CLK);
endmodule
```



Behavioral Statements

```
if( expr ) then <statement>; else <statement>;

case(selector) val0: <statement>; val1: <statement>;
            default: <statement>; endcase;

for(i=0; i <=10; i=i+1) A = B + i;

i=0; while(i <= 15)
       begin A = B + i; i = i + 1; end
```

Concurrent Constructs

- @ means wait for a change in value
  - `@(a) w=4;` Wait for ‘a’ to change to resume execution
- wait(condition)
  - `wait(a==1) w=4;` Wait for ‘a’ to become 1 before resuming execution

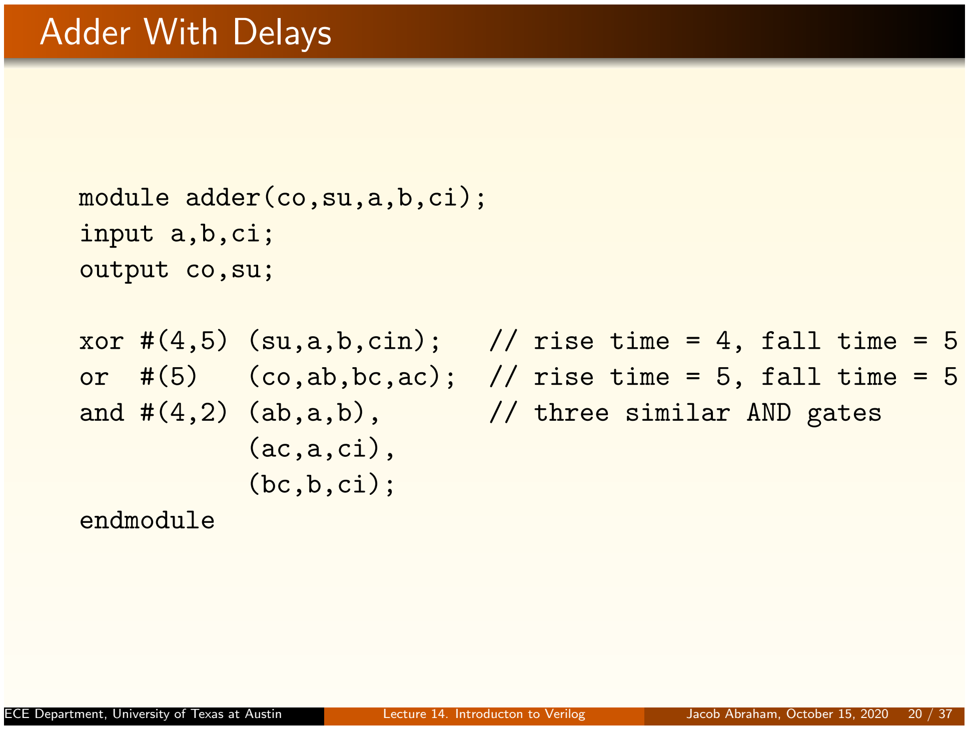
Another Example

```
module mux(f,sel,b,a);
   input sel, b, c;
   output f;

   reg f;
```

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

assign f=sel ? b:a;
```

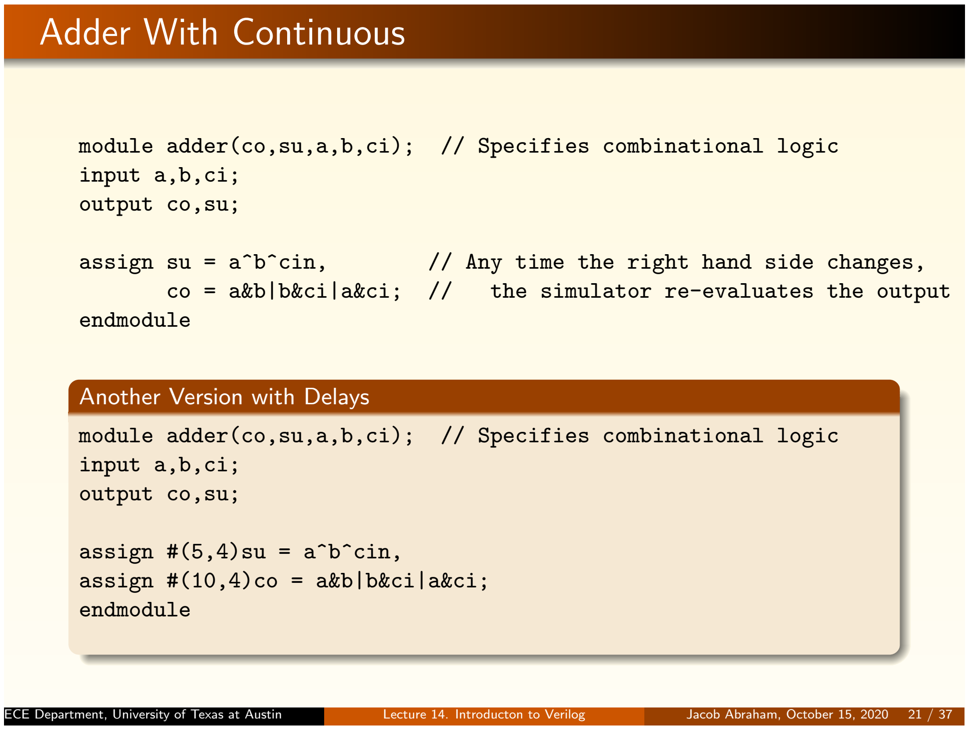
Case Statement

```
module function_table(f,a,b,c);
input a,b,c;
output f;

reg f;

always @(a or b or c)
   case({a,b,c}) // concatenate a,b,c to form a 3-bit number
      3'b000: f=1'b0;
      3'b001: f=1'b1;
      3'b010: f=1'b0;
      3'b011: f=1'b0;
      3'b100: f=1'b0;
      3'b101: f=1'b1;
      3'b110: f=1'b0;
      3'b111: f=1'b0;
   endcase
endmodule
```



Adder With Delays

```
module adder(co,su,a,b,ci);
input a,b,ci;
output co,su;

xor #(4,5) (su,a,b,cin);   // rise time = 4, fall time = 5
or  #(5)   (co,ab,bc,ac);  // rise time = 5, fall time = 5
and #(4,2) (ab,a,b),       // three similar AND gates
           (ac,a,ci),
           (bc,b,ci);
endmodule
```

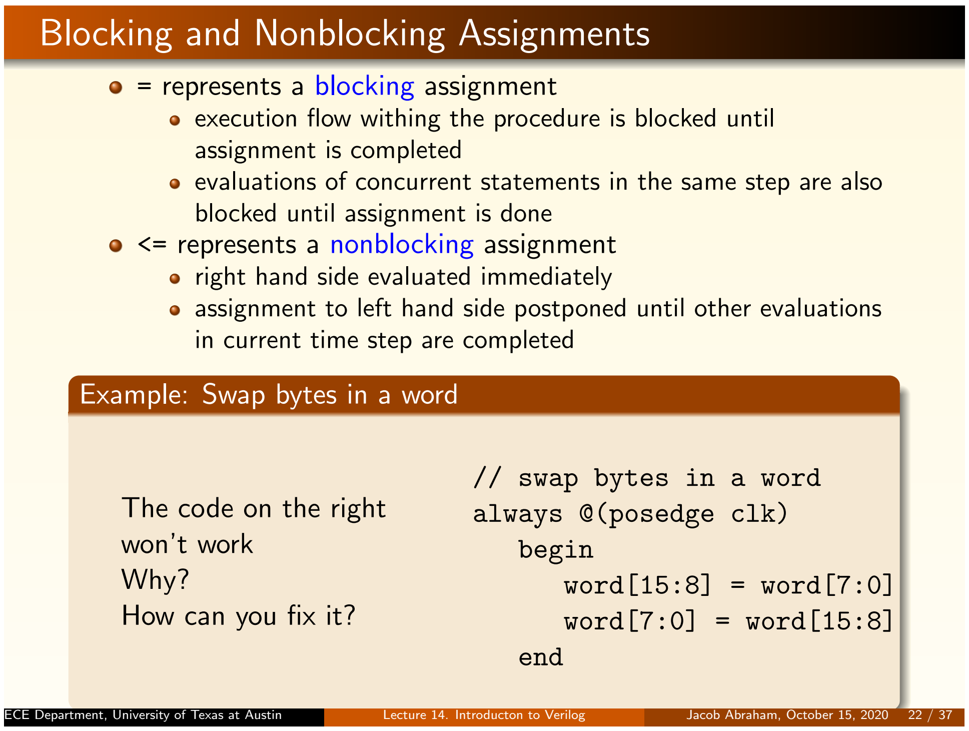


Adder With Continuous

```
module adder(co,su,a,b,ci);  // Specifies combinational logic
input a,b,ci;
output co,su;

assign su = a^b^cin,        // Any time the right hand side changes,
       co = a&b|b&ci|a&ci;  //   the simulator re-evaluates the output
endmodule
```

Another Version with Delays

```
module adder(co,su,a,b,ci);  // Specifies combinational logic
input a,b,ci;
output co,su;

assign #(5,4)su = a^b^cin,
assign #(10,4)co = a&b|b&ci|a&ci;
endmodule
```



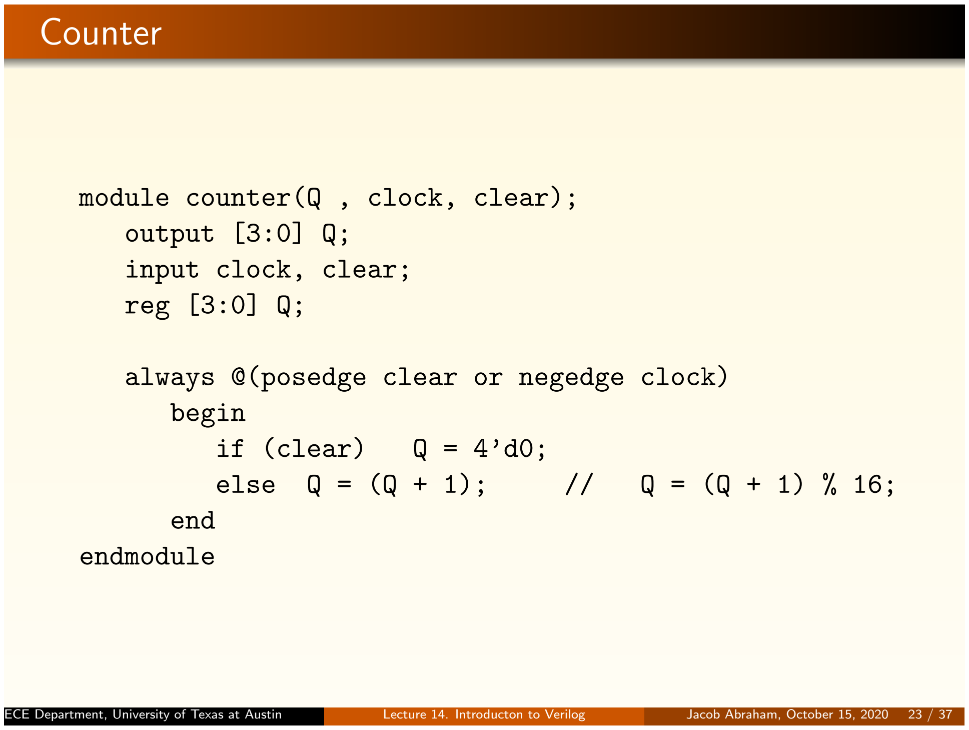
Blocking and Nonblocking Assignments

- = represents a blocking assignment
  - execution flow withing the procedure is blocked until assignment is completed
  - evaluations of concurrent statements in the same step are also blocked until assignment is done
- <= represents a nonblocking assignment
  - right hand side evaluated immediately
  - assignment to left hand side postponed until other evaluations in current time step are completed

Example: Swap bytes in a word

The code on the right won't work
Why?
How can you fix it?

```
// swap bytes in a word
always @(posedge clk)
   begin
      word[15:8]  = word[7:0]
      word[7:0]  = word[15:8]
   end
```



Counter

```
module counter(Q , clock, clear);
   output [3:0] Q;
   input clock, clear;
   reg [3:0] Q;

   always @(posedge clear or negedge clock)
      begin
         if (clear)   Q = 4'd0;
         else  Q = (Q + 1);     //   Q = (Q + 1) % 16;
      end
endmodule
```

Counter, Cont'd

```
module stimulus;
reg CLOCK, CLEAR;
wire [3:0] Q;

counter c1(Q, CLOCK, CLEAR);
initial
begin
$monitor($time, " Count Q = %b Clear= %b",  Q[3:0],CLEAR);
CLEAR = 1'b1;
#34 CLEAR = 1'b0;
#200 CLEAR = 1'b1;
#50 CLEAR = 1'b0;
#400 $stop;
end

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

endmodule
```

Bus Select

```
module slect_bus( out, b0, b1, b2, b3, enable, s);
parameter    n = 16;
parameter    Z = 16'bz;  // define a 16 bit of z
output [1:n] out;        // n-bit output
input  [1:n] b0, b1, b2, b3;  // n-bit inputs
input  enable;
input  [1:2] s;

tri   [1:n] data;            // tri-state net
tri   [1:n] out = enable ? data: Z; // net declaration
                          // with continuous assignment

assign
      data = (s==0) ? bus0 : Z,// 4 continuous assignments
      data = (s==1) ? bus1 : Z,
      data = (s==2) ? bus2 : Z,
      data = (s==3) ? bus3 : Z;
endmodule
```

Tri-State Latch

```
module tri_latch( q, nq, clk, data, enable);
output q, nq;
input  clk, data, enable;
tri    q, nq;

not #5 (ndata, data);
nand #(3, 5) (wa, data, clk), (wb, ndata, clk);
nand #(12,15) (ql, nql, wa), (nql, ql, wb);

bufif1 #(3,5,13)  // rise, fall, change to z
   q_dr ( q, ql, enable),     // when enable = 1, q=ql
   nq_dr (nq, nql, enable);   // when enable = 0, q=ql=z
endmodule
```

User Defined Primitives

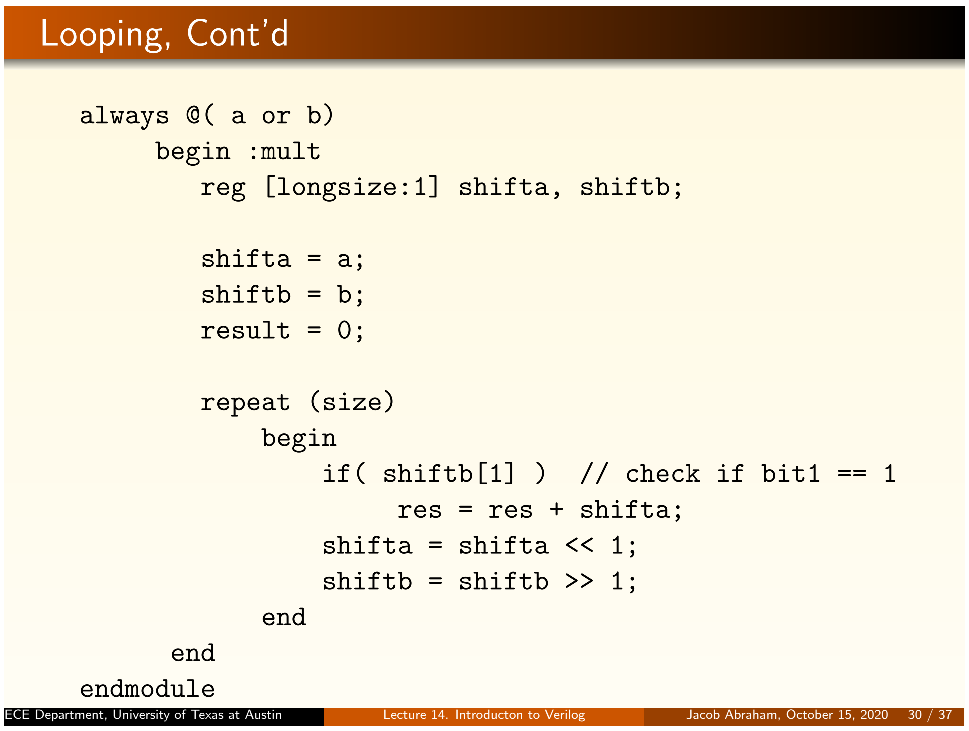
```
primitive mux(mux,cntr,A,B);
output mux;
input  cntr, A, B;
    table
    // cntr A B   mux
        0   1 ? : 1 ;  // ? represents don't care
        0   0 ? : 0 ;
        1   ? 0 : 0 ;
        1   ? 1 : 1 ;
        x   0 0 : 0 ;
        x   1 1 : 1 ;
    endtable
endprimitive
```

User Defined Primitives, Cont'd

```
primitive latch(q, clk, data);
output    q; reg q;
input     clk, data;

    table
       //  clk  data state output/nxtstate
            0    1  : ?   :   1 ;
            0    0  : ?   :   0 ;
            1    ?  : ?   :   - ; // represents no change
    endtable
endprimitive
```

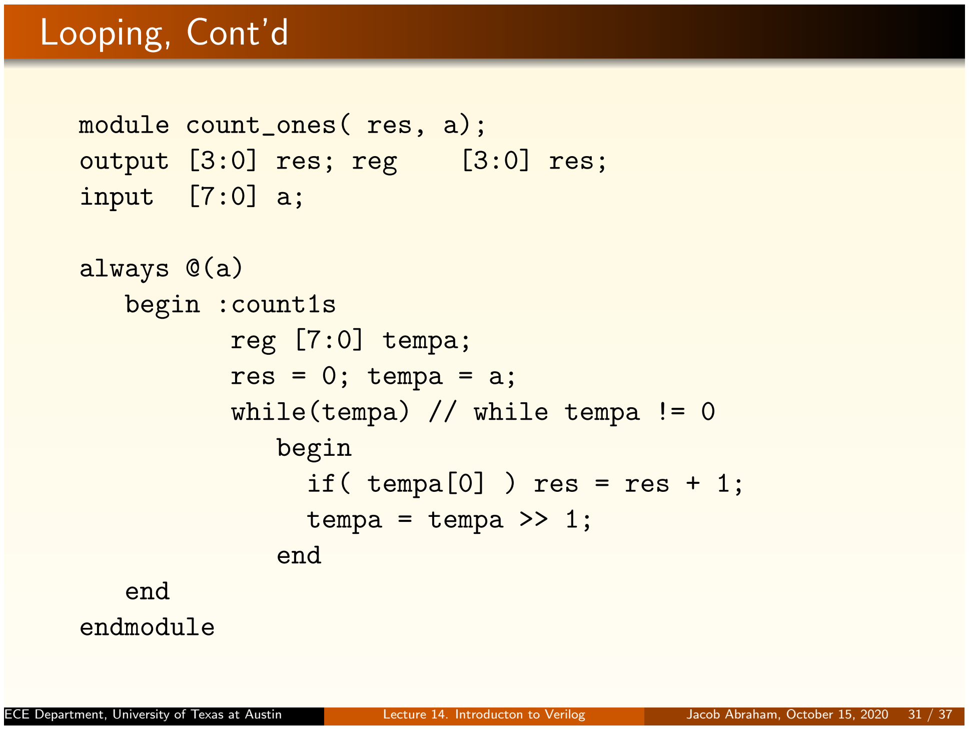
Looping

```
module mult( res, a, b);
parameter size = 8, longzise = 16;
input [size:1] a, b;
output [longsize:1] res;

reg [size:1] a, b;
reg [longsize:1] res;
```



Looping, Cont'd

```
always @( a or b)
     begin :mult
        reg [longsize:1] shifta, shiftb;

        shifta = a;
        shiftb = b;
        result = 0;

        repeat (size)
            begin
                if( shiftb[1] )  // check if bit1 == 1
                     res = res + shifta;
                shifta = shifta << 1;
                shiftb = shiftb >> 1;
            end
      end
endmodule
```



Looping, Cont'd

```
module count_ones( res, a);
output [3:0] res; reg    [3:0] res;
input  [7:0] a;

always @(a)
   begin :count1s
          reg [7:0] tempa;
          res = 0; tempa = a;
          while(tempa) // while tempa != 0
             begin
               if( tempa[0] ) res = res + 1;
               tempa = tempa >> 1;
             end
   end
endmodule
```

Parallel Blocks

- Its statement are executed concurrently
- Delay values for each statement are relative to the simulation when control enter block
- Control passed out of the block when the last time-ordered statement is executed

Order of statements is not important

Parallel Block

```
fork
     #100 r = 100;
     #50 r = 50;
     #150 r = 150;
   join
```

Equivalent Sequential Block assuming c changes every 50 units

```
begin
     @c r = 50;
     @c r = 150;
     @c r = 200;
end
```

Another Example

areg is loaded when both A and B occur in any order

```
begin
   fork
      @A;
      @B;
   join
   areg = breg;
end
```

Tasks

- Tasks provide a way to execute common procedures for different places
- They provide a means of breaking large procedure into smaller
- Tasks make it easy to debug and read the source code

```
task proc_name;
     input a, b;
     inout c;
     output d, e;

        <statement>

endtask
```

This can be enabled by:
`proc_name(v, w, x, y, z);`
which performs the following assignment:
`a=v; b=w; c=x;`
The following is performed on completion of the task:
`x=c; y=d; z=e;`

Functions

- They differ from task is the following way
  - They return a value that to be used in an expression
  - They must of of zero simulation time duration
  - They must have at least one input

```
function [7:0] func_name;
     input [15:0] a;

     begin
        <statement>
        func_name = expression;
     end
endfunction
```

To enable this:
`new_address = a * func_name(old_address)`

Latches and Flip-Flops (Flops)

A Latch is synthesized if you write:

```
always @(CLK) begin
     if (CLK) begin
          LatchOut = LatchInput
     end
end
```

A Flop is synthesized with:

```
always @(posedge CLK) begin
     LatchOut = LatchInput
end
```

Constructs Not Supported in Synthesis

initial	used only in test benches
events	for synchronizing test bench components
real	real data type not supported
time	time data type not supported
force/release	not supported
assign/deassign	not supported for `reg` data types but assign on `wire` data type supported
fork/join	use non-blocking assignments to get the same effect
primitives	only gate-level primitives supported
table	UDP and tables not supported