



The Verilog Language

COMS W4995-02

Prof. Stephen A. Edwards

Fall 2002

Columbia University

Department of Computer Science

The Verilog Language

Originally a modeling language for a very efficient event-driven digital logic simulator

Later pushed into use as a specification language for logic synthesis

Now, one of the two most commonly-used languages in digital hardware design (VHDL is the other)

Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages

Combines structural and behavioral modeling styles

Multiplexer Built From Primitives

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

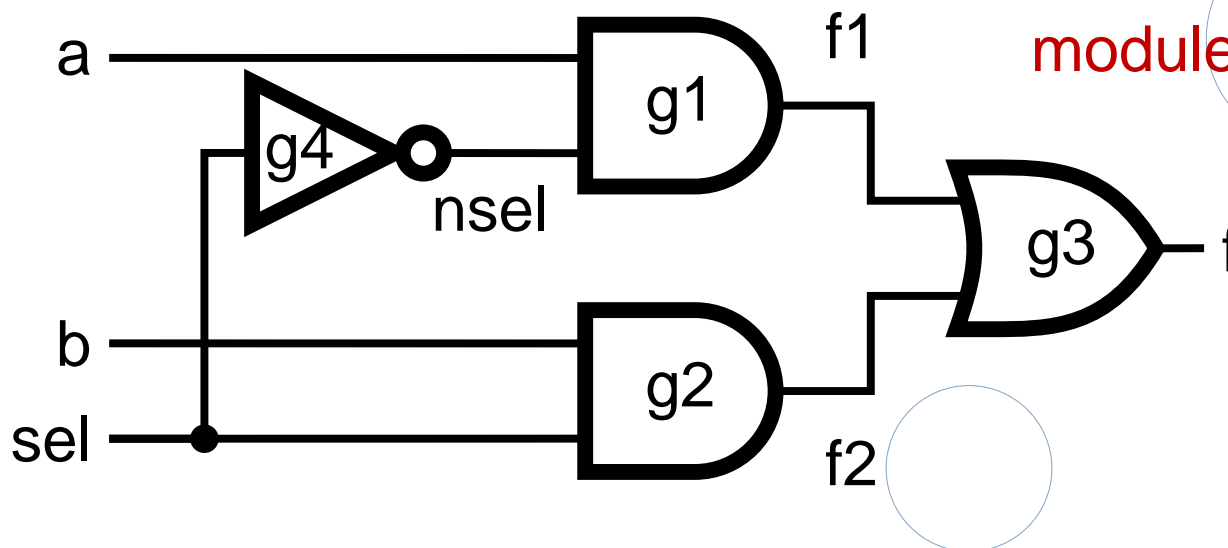
Verilog programs
built from modules

```
    and g1(f1, a, nsel),  
        g2(f2, b, sel);  
    or  g3(f, f1, f2);  
    not g4(nsel, sel);
```

Each module has
an interface

Module may contain
structure: instances of
primitives and other
modules

```
endmodule
```



Multiplexer Built with Always

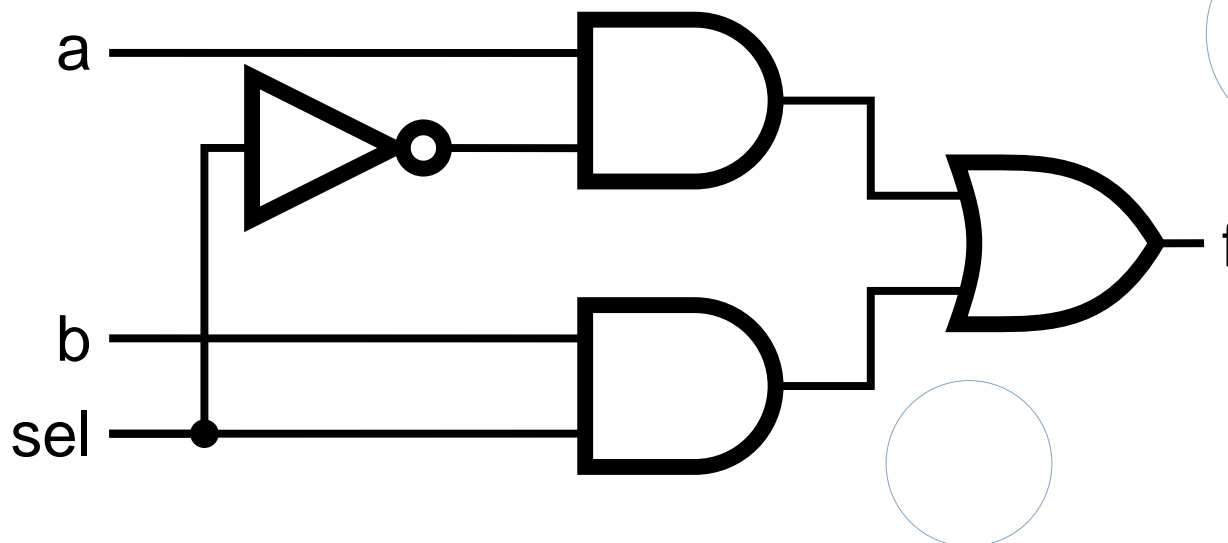
```
module mux(f, a, b, sel);  
output f;  
input a, b, sel;  
reg f;
```

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

```
endmodule
```

Modules may
contain one or more
always blocks

Sensitivity list
contains signals
whose change
makes the block
execute

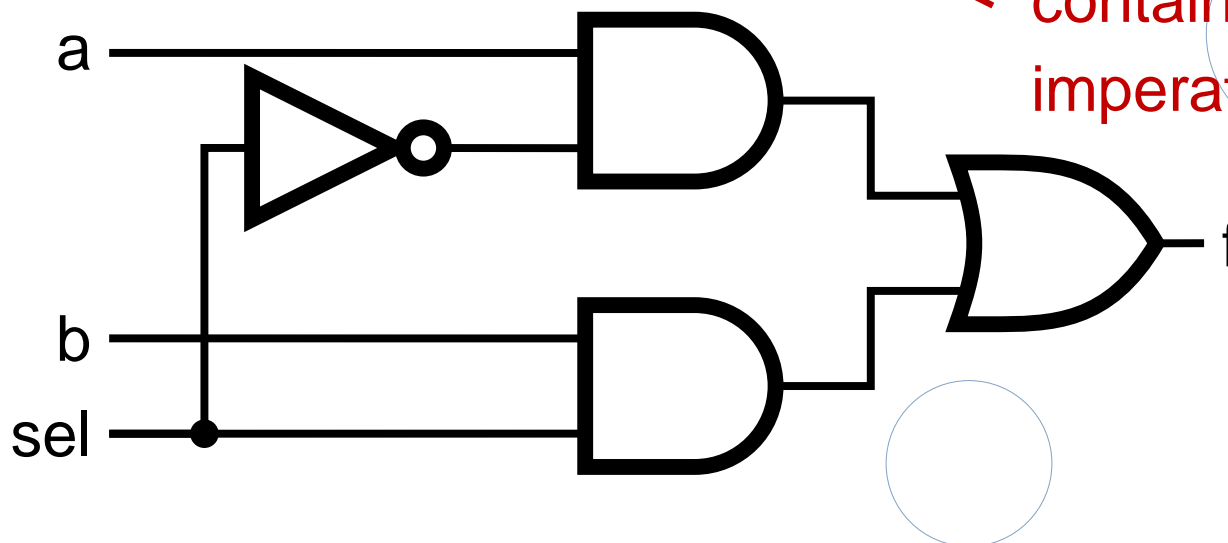


Multiplexer Built with Always

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

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

```
endmodule
```



A **reg** behaves like memory: holds its value until imperatively assigned otherwise

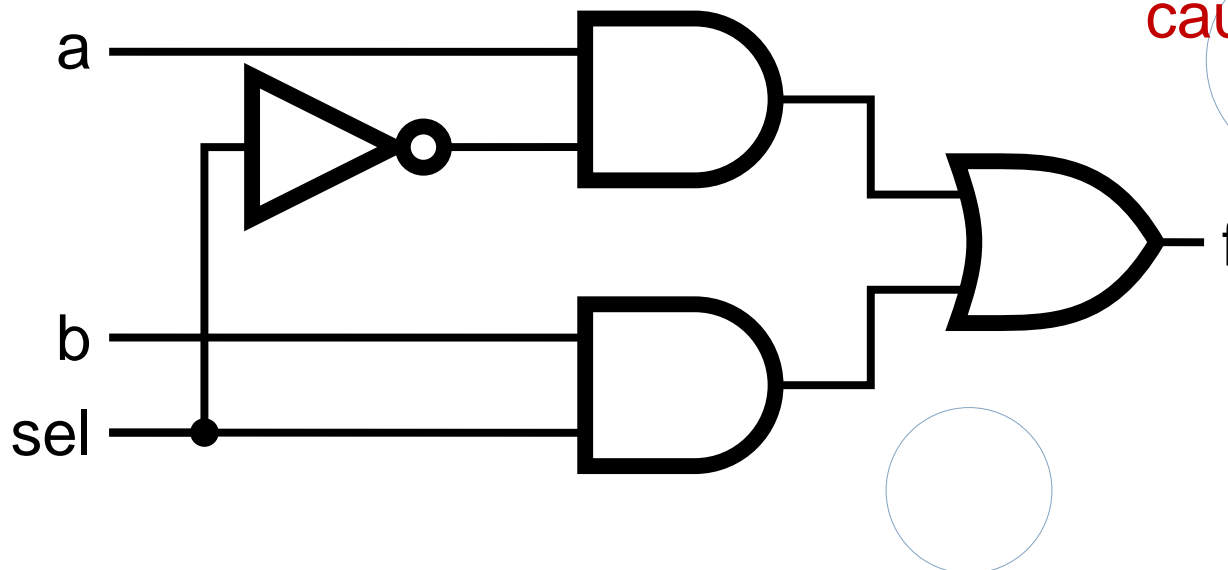
Body of an always block contains traditional imperative code

Mux with Continuous Assignment

```
module mux(f, a, b, sel);  
output f;  
input a, b, sel;  
  
assign ← f = sel ? a : b;  
  
endmodule
```

LHS is always set to
the value on the RHS

Any change on the right
causes reevaluation

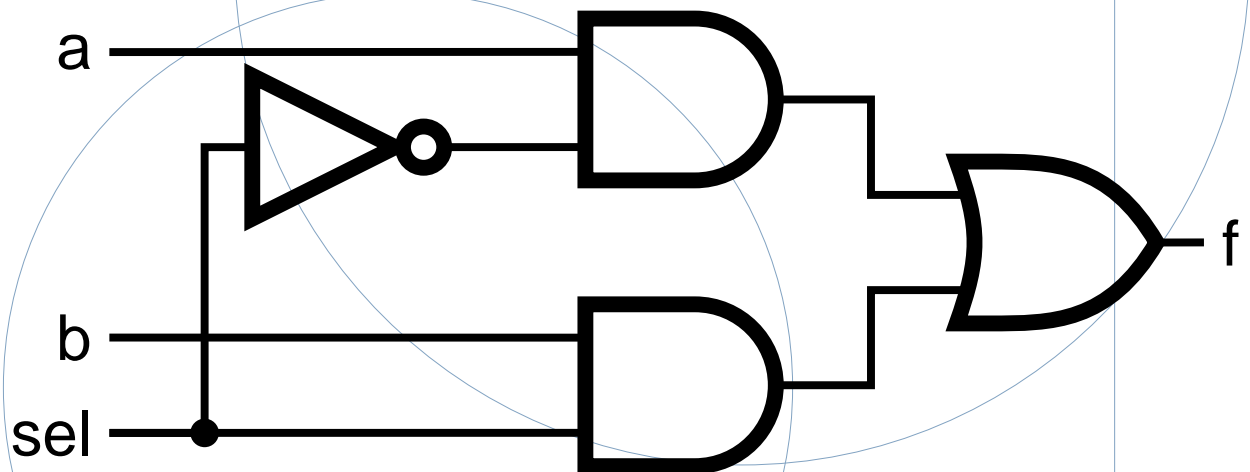


Mux with User-Defined Primitive

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

```
table  
  1?0 : 1;  
  0?0 : 0;  
  ?11 : 1;  
  ?01 : 0;  
  11? : 1;  
  00? : 0;  
endtable  
endprimitive
```

Behavior defined using
a truth table that
includes “don’t cares”
This is a less pessimistic than
others: when a & b match, sel is
ignored; others produce X

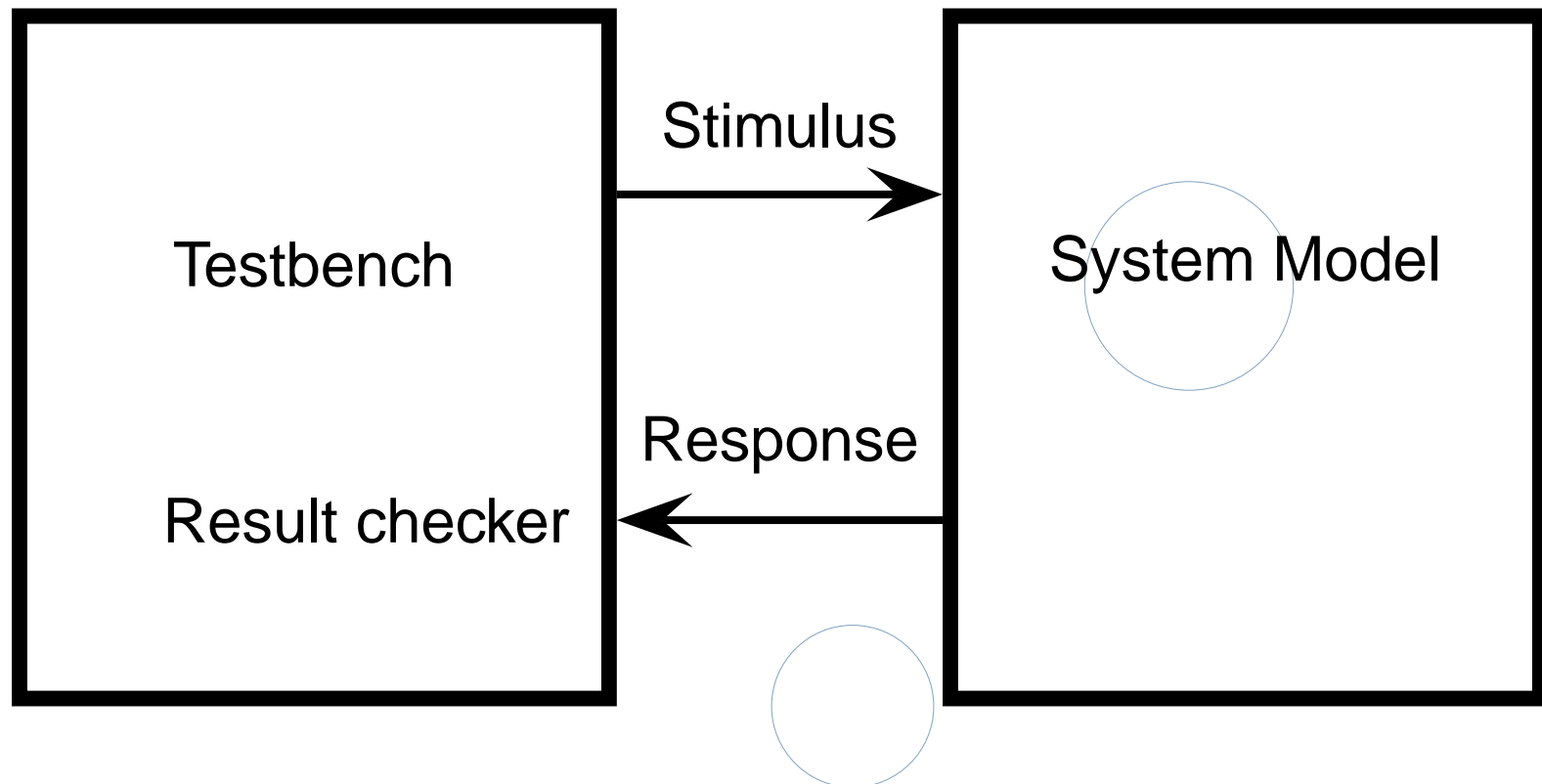


How Are Simulators Used?

Testbench generates stimulus and checks response

Coupled to model of the system

Pair is run simultaneously



Structural Modeling

When Verilog was first developed (1984) most logic simulators operated on netlists

Netlist: list of gates and how they're connected

A natural representation of a digital logic circuit

Not the most convenient way to express test benches

Behavioral Modeling

A much easier way to write testbenches

Also good for more abstract models of circuits

- Easier to write
- Simulates faster

More flexible

Provides sequencing

Verilog succeeded in part because it allowed both the model and the testbench to be described together

How Verilog Is Used

Virtually every ASIC is designed using either Verilog or VHDL (a similar language)

Behavioral modeling with some structural elements

“Synthesis subset” can be translated using Synopsys’ Design Compiler or others into a netlist

Design written in Verilog

Simulated to death to check functionality

Synthesized (netlist generated)

Static timing analysis to check timing

Two Main Components of Verilog: Behavioral

Concurrent, event-triggered processes (behavioral)

Initial and Always blocks

Imperative code that can perform standard data manipulation tasks (assignment, if-then, case)

Processes run until they delay for a period of time or wait for a triggering event

Two Main Components of Verilog: Structural

Structure (Plumbing)

Verilog program build from modules with I/O interfaces

Modules may contain instances of other modules

Modules contain local signals, etc.

Module configuration is static and all run concurrently

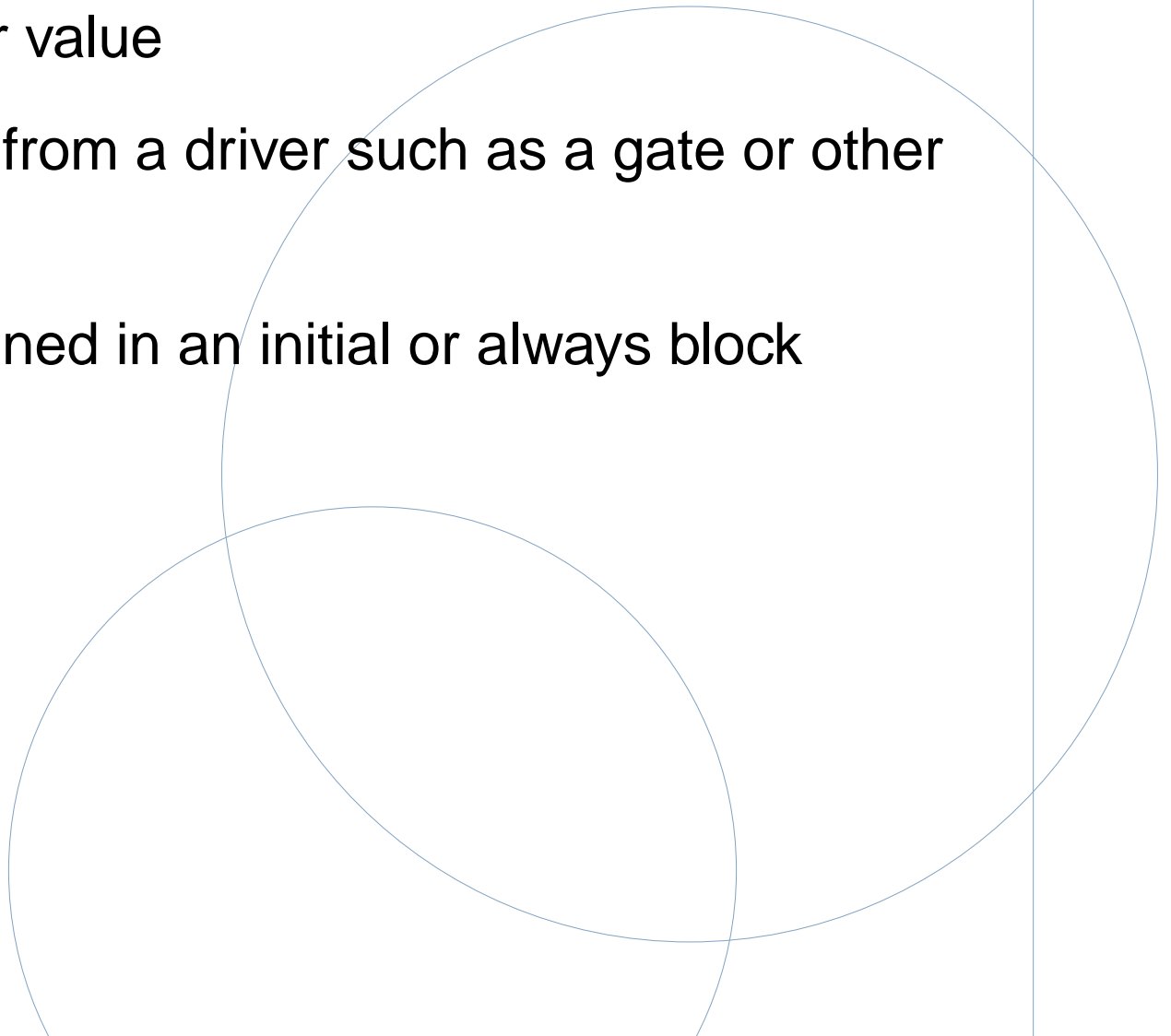
Two Main Data Types: Nets

Nets represent connections between things

Do not hold their value

Take their value from a driver such as a gate or other module

Cannot be assigned in an initial or always block



Two Main Data Types: Regs

Regs represent data storage

Behave exactly like memory in a computer

Hold their value until explicitly assigned in an initial or always block

Never connected to something

Can be used to model latches, flip-flops, etc., but do not correspond exactly

Actually shared variables with all their attendant problems

Four-valued Data

Verilog's nets and registers hold four-valued data

0, 1: Obvious

Z: Output of an undriven tri-state driver. Models case where nothing is setting a wire's value

X: Models when the simulator can't decide the value

- Initial state of registers
- When a wire is being driven to 0 and 1 simultaneously
- Output of a gate with Z inputs

Four-valued Logic

Logical operators work on three-valued logic

D	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Outputs 0 if either input is 0

Outputs X if both inputs are gibberish



Structural Modeling

Nets and Registers

Wires and registers can be bits, vectors, and arrays

wire a;	// Simple wire
tri [15:0] dbus;	// 16-bit tristate bus
tri #(5,4,8) b;	// Wire with delay
reg [-1:4] vec;	// Six-bit register
triereg (small) q;	// Wire stores a small charge
integer imem[0:1023];	// Array of 1024 integers
reg [31:0] dcache[0:63];	// A 32-bit memory

Modules and Instances

Basic structure of a Verilog module:

```
module mymod(out1, out2, in1, in2);
```

```
output out1;
```

```
output [3:0] out2;
```

```
input in1;
```

```
input [2:0] in2;
```

```
endmodule
```



Verilog convention
lists outputs first

Instantiating a Module

Instances of

```
module mymod(y, a, b);
```

look like

```
mymod mm1(y1, a1, b1); // Connect-by-position
```

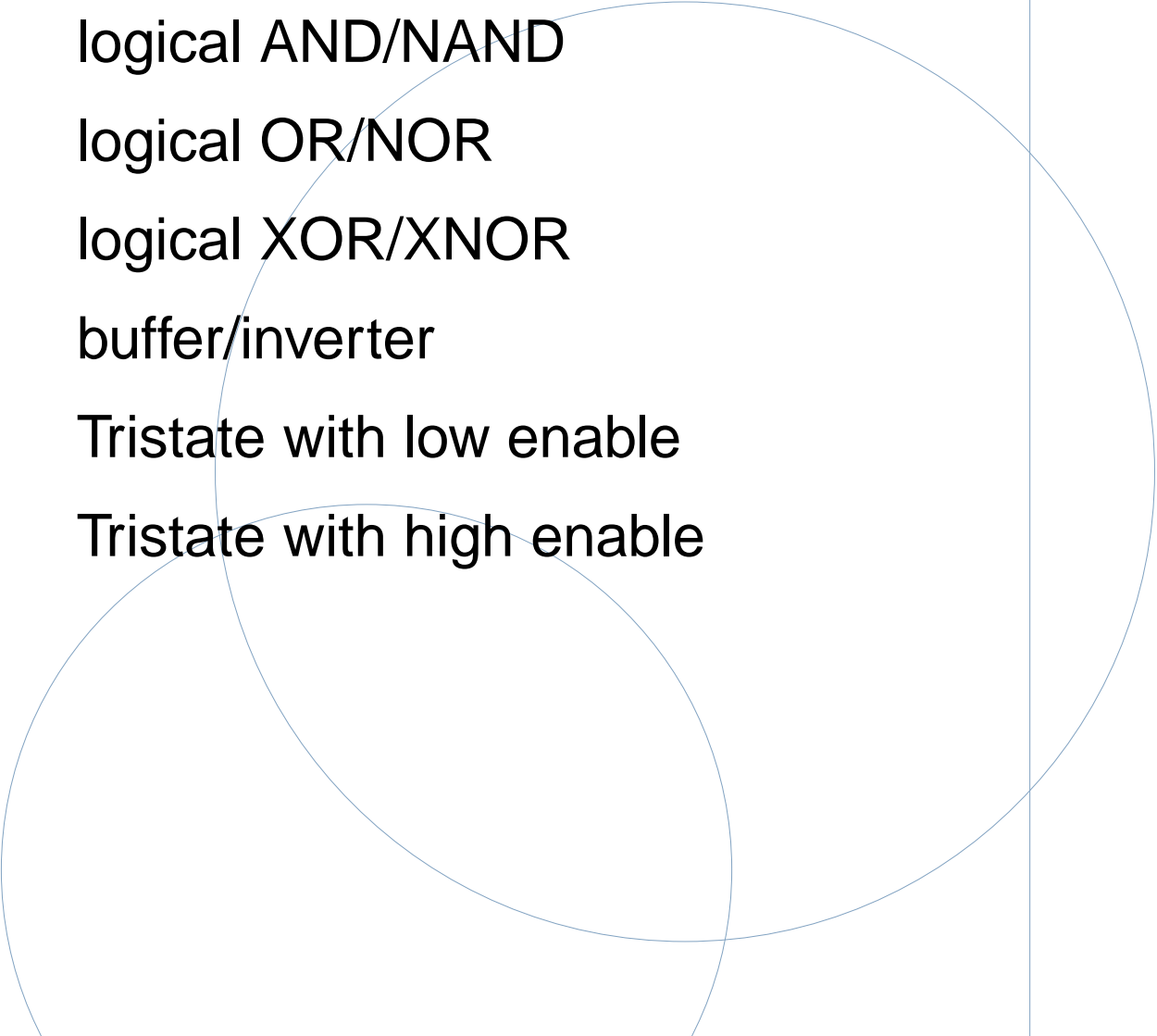
```
mymod (y2, a1, b1),  
      (y3, a2, b2);    // Instance names omitted
```

```
// Connect-by-name
```

```
mymod mm2(.a(a2), .b(b2), .y(c2));
```

Gate-level Primitives

Verilog provides the following:



and	nand	logical AND/NAND
or	nor	logical OR/NOR
xor	xnor	logical XOR/XNOR
buf	not	buffer/inverter
bufif0	notif0	Tristate with low enable
bifif1	notif1	Tristate with high enable

Delays on Primitive Instances

Instances of primitives may include delays

buf	b1(a, b);	// Zero delay
buf #3	b2(c, d);	// Delay of 3
buf #(4,5)	b3(e, f);	// Rise=4, fall=5
buf #(3:4:5)	b4(g, h);	// Min-typ-max

User-Defined Primitives

Way to define gates and sequential elements using a truth table

Often simulate faster than using expressions, collections of primitive gates, etc.

Gives more control over behavior with X inputs

Most often used for specifying custom gate libraries

A Carry Primitive

```
primitive carry(out, a, b, c);
```

```
output out;
```

```
input a, b, c;
```

```
table
```

```
  00? : 0;
```

```
  0?0 : 0;
```

```
  ?00 : 0;
```

```
  11? : 1;
```

```
  1?1 : 1;
```

```
  ?11 : 1;
```

```
endtable
```

```
endprimitive
```

Always has exactly
one output

Truth table may include
don't-care (?) entries

A Sequential Primitive

```
Primitive dff( q, clk, data);
```

```
output q; reg q;
```

```
input clk, data;
```

```
table
```

```
// clk data q new-q
```

```
(01)    0    : ?    : 0;    // Latch a 0
```

```
(01)    1    : ?    : 1;    // Latch a 1
```

```
(0x)    1    : 1    : 1;    // Hold when d and q both 1
```

```
(0x)    0    : 0    : 0;    // Hold when d and q both 0
```

```
(?0)    ?    : ?    : -;    // Hold when clk falls
```

```
?      (??)  : ?    : -;    // Hold when clk stable
```

```
endtable
```

```
endprimitive
```

Continuous Assignment

Another way to describe combinational function

Convenient for logical or datapath specifications

```
wire [8:0] sum;
```



Define bus widths

```
wire [7:0] a, b;
```

```
wire carryin;
```

```
assign sum = a + b + carryin;
```



Continuous assignment:
permanently sets the value of sum to be a+b+carryin.
Recomputed when a, b, or carryin changes



Behavioral Modeling

Initial and Always Blocks

initial

begin

// imperative statements

end

Runs when simulation starts

Terminates when control
reaches the end

Good for providing stimulus

always

begin

// imperative statements

end

Runs when simulation starts

Restarts when control
reaches the end

Good for modeling or
specifying hardware

Initial and Always

Run until they encounter a delay

```
initial begin
    #10 a = 1; b = 0;
    #10 a = 0; b = 1;
end
```

or a wait for an event

```
always @(posedge clk) q = d;
```

```
always begin
    wait(i);
    a = 0;
    wait(~i);
    a = 1;
end
```

Procedural Assignment

Inside an initial or always block:

```
sum = a + b + cin;
```

Just like in C: RHS evaluated and assigned to LHS before next statement executes

RHS may contain wires and/or regs

LHS must be a reg

(only primitives or continuous assignment may set wire values)

Imperative Statements

```
if (select == 1) y = a;  
else y = b;
```

```
case (op)  
  2'b00: y = a + b;  
  2'b01: y = a - b;  
  2'b10: y = a ^ b;  
  default: y = 'hxxxx;  
endcase
```


For Loops

Example generates an increasing sequence of values on an output

```
reg [3:0] i, output;  
  
for ( i = 0 ; i <= 15 ; i = i + 1 ) begin  
    output = i;  
    #10;  
end
```

While Loops

A increasing sequence of values on an output

```
reg [3:0] i, output;  
  
i = 0;  
while (i <= 15) begin  
    output = i;  
    #10 i = i + 1;  
end
```

Modeling A Flip-Flop With Always

Very basic: an edge-sensitive flip-flop

```
reg q;
```

```
always @(posedge clk)
```

```
    q = d;
```

q = d assignment runs when clock rises: exactly the behavior you expect

Blocking vs. Nonblocking

Verilog has two types of procedural assignment

Fundamental problem:

- In a synchronous system, all flip-flops sample simultaneously
- In Verilog, **`always @(posedge clk)`** blocks run in some undefined sequence

A Flawed Shift Register

This does not work as you would expect:

```
reg d1, d2, d3, d4;
```

```
always @(posedge clk) d2 = d1;
```

```
always @(posedge clk) d3 = d2;
```

```
always @(posedge clk) d4 = d3;
```

These run in some order, but you don't know which

Non-blocking Assignments

This version does work:


```
reg d1, d2, d3, d4;
```

```
always @(posedge clk) d2 <= d1;
```

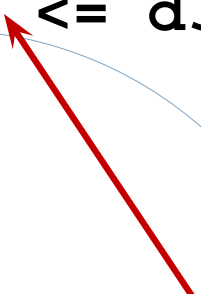
```
always @(posedge clk) d3 <= d2;
```

```
always @(posedge clk) d4 <= d3;
```

Nonblocking rule:
RHS evaluated
when assignment
runs



LHS updated only
after all events for
the current instant
have run



Nonblocking Can Behave Oddly

A sequence of nonblocking assignments don't communicate

```
a = 1;  
b = a;  
c = b;
```

Blocking assignment:
a = b = c = 1

```
a <= 1;  
b <= a;  
c <= b;
```

Nonblocking assignment:
a = 1
b = old value of a
c = old value of b

Nonblocking Looks Like Latches

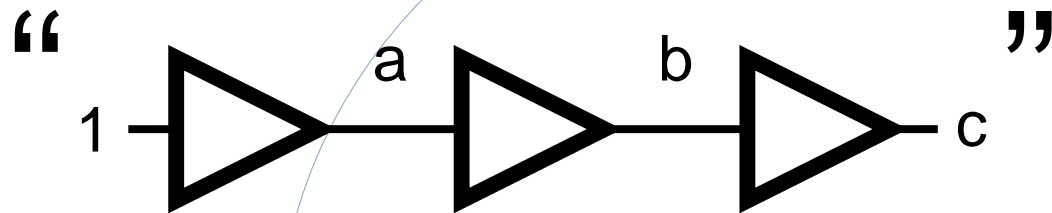
RHS of nonblocking taken from latches

RHS of blocking taken from wires

```
a = 1;
```

```
b = a;
```

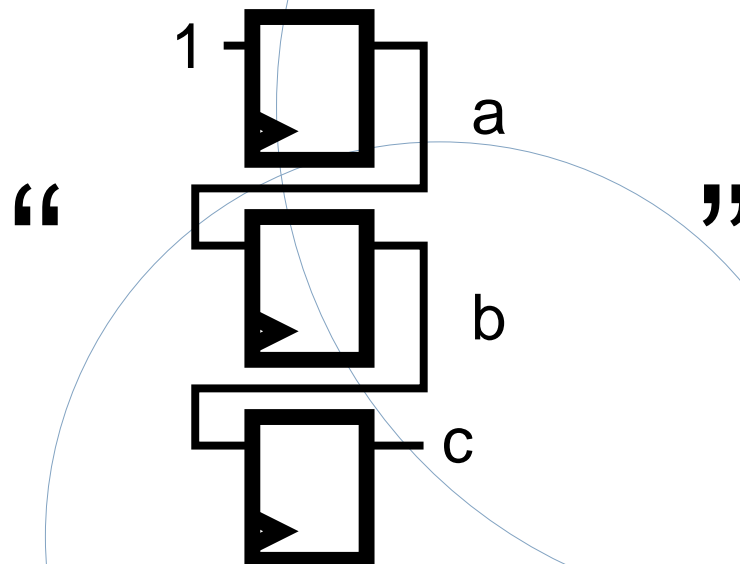
```
c = b;
```



```
a <= 1;
```

```
b <= a;
```

```
c <= b;
```





Building Behavioral Models

Modeling FSMs Behaviorally

There are many ways to do it:

- Define the next-state logic combinationaly and define the state-holding latches explicitly
- Define the behavior in a single `always @(posedge clk)` block
- Variations on these themes

FSM with Combinational Logic

```
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;

always @(a or b or state)
  case (state)
    2'b00: begin
      o = a & b;
      nextState = a ? 2'b00 : 2'b01;
    end
    2'b01: begin
      o = 0; nextState = 2'b10;
    end
  endcase

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;

endmodule
```

Output o is declared a reg because it is assigned procedurally, not because it holds state

FSM with Combinational Logic


```
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;

always @(a or b or state)
  case (state)
    2'b00: begin
      o = a & b;
      nextState = a ? 2'b00 : 2'b01;
    end
    2'b01: begin
      o = 0; nextState = 2'b10;
    end
  endcase

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;

endmodule
```

Combinational block must be sensitive to any change on any of its inputs (Implies state-holding elements otherwise)



Latch implied by sensitivity to the clock or reset only



FSM from a Single Always Block

```
module FSM(o, a, b);  
  output o; reg o;  
  input a, b;  
  reg [1:0] state;  
  
  always @(posedge clk or reset)  
    if (reset) state <= 2'b00;  
    else case (state)  
      2'b00: begin  
        state <= a ? 2'b00 : 2'b01;  
        o <= a & b;  
      end  
      2'b01: begin  
        state <= 2'b10;  
        o <= 0;  
      end  
    endcase  
endcase
```

Expresses Moore
machine behavior:
Outputs are latched.
Inputs only sampled
at clock edges

Nonblocking assignments
used throughout to ensure
coherency. RHS refers to
values calculated in
previous clock cycle

Writing Testbenches

```
module test;  
reg a, b, sel;
```

Inputs to device
under test

```
mux m(y, a, b, sel);
```

Device under test

```
initial begin
```

```
    $monitor($time,, "a=%b b=%b sel=%b y=%b",  
              a, b, sel, y);
```

```
    a = 0; b = 0; sel = 0;
```

```
    #10 a = 1;
```

```
    #10 sel = 1;
```

```
    #10 b = 1;
```

```
end
```

Stimulus generated by
sequence of
assignments and
delays