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

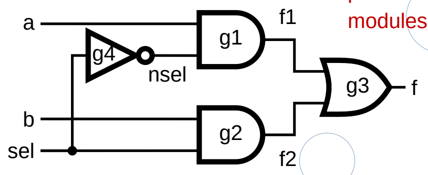
Multiplexer Built From Primitives

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

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

endmodule
```

Verilog programs built from modules

Each module has an interface

Module may contain structure: instances of primitives and other modules

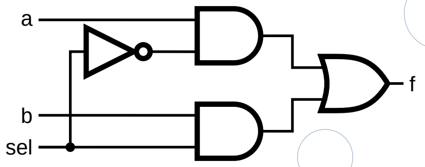


Multiplexer Built with Always

```
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

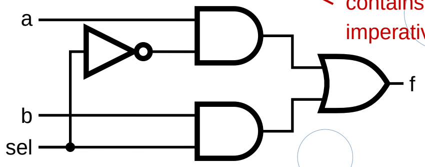


Multiplexer Built with Always

```
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

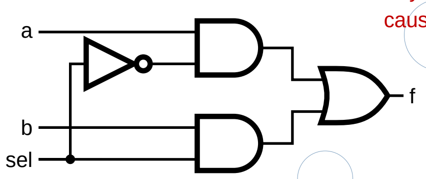


Mux with Continuous Assignment

```
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

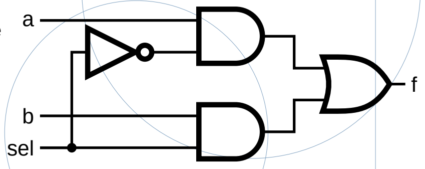


Mux with User-Defined Primitive

```
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

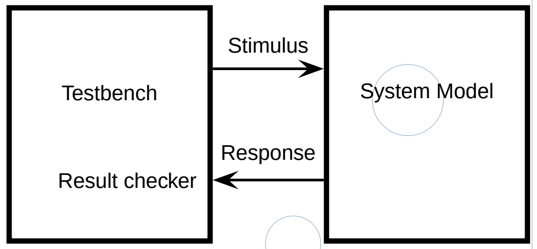


How Are Simulators Used?

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

Discrete-event Simulation

Basic idea: only do work when something changes

Centered around an event queue that contains events labeled with the simulated time at which they are to be executed

Basic simulation paradigm

- Execute every event for the current simulated time
- Doing this changes system state and may schedule events in the future
- When there are no events left at the current time instance, advance simulated time soonest event in the queue

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

	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
trireg (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
```

Switch-level Primitives

Verilog also provides mechanisms for modeling CMOS transistors that behave like switches

A more detailed modeling scheme that can catch some additional electrical problems when transistors are used in this way

Now, little-used because circuits generally aren't built this way

More seriously, model is not detailed enough to catch many of the problems

These circuits are usually simulated using SPICE-like simulators based on nonlinear differential equation solvers

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	always
begin	begin
// imperative statements	// imperative statements
end	end
Runs when simulation starts	Runs when simulation starts
Terminates when control reaches the end	Restarts when control reaches the end
Good for providing stimulus	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 ˆ 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;`
	`a <= 1;`
	`b <= a;`
	`c <= b;`

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

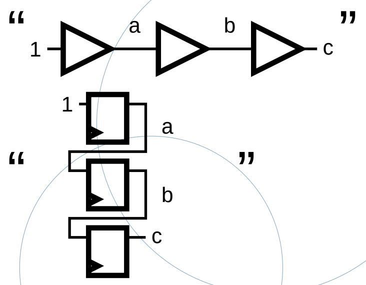
Nonblocking Looks Like Latches

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 combinationally 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
```

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;

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

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
```

Inputs to device under test

Device under test

$monitor is a built-in even-driven “printf”

Stimulus generated by sequence of assignments and delays

Simulating Verilog

Simulation Behavior

Scheduled using an event queue

Non-preemptive, no priorities

A process must explicitly request a context switch

Events at a particular time unordered

Scheduler runs each event at the current time, possibly scheduling more as a result

Two Types of Events

Evaluation events compute functions of inputs

Update events change outputs

Split necessary for delays, nonblocking assignments, etc.

```
a <= b + c
```

Update event writes new value of a and schedules any evaluation events that are sensitive to a change on a

Evaluation event reads values of b and c, adds them, and schedules an update event

Simulation Behavior

Concurrent processes (initial, always) run until they stop at one of the following

- #42
  Schedule process to resume 42 time units from now
- wait(cf & of)
  Resume when expression “cf & of” becomes true
- @(a or b or y)
  Resume when a, b, or y changes
- @(posedge clk)
  Resume when clk changes from 0 to 1

Simulation Behavior

Infinite loops are possible and the simulator does not check for them This runs forever: no context switch allowed, so ready can never change

```
while (~ready)
  count = count + 1;
```

Instead, use

```
wait(ready);
```

Simulation Behavior

Race conditions abound in Verilog

These can execute in either order: final value of a undefined:

```
always @(posedge clk) a = 0;
always @(posedge clk) a = 1;
```

Simulation Behavior

Semantics of the language closely tied to simulator implementation

Context switching behavior convenient for simulation, not always best way to model

Undefined execution order convenient for implementing event queue

Compiled-Code Discrete-Event Sim.

Most modern simulators use this approach

Verilog program compiled into C

Each concurrent process (e.g., continuous assignment, always block) becomes one or more C functions

Initial and always blocks split into multiple functions, one per segment of code between a delay, a wait, or event control (@)

Central, dynamic event queue invokes these functions and advances simulation time

Verilog and Logic Synthesis

Logic Synthesis

Verilog is used in two ways

Model for discrete-event simulation

Specification for a logic synthesis system

Logic synthesis converts a subset of the Verilog language into an efficient netlist

One of the major breakthroughs in designing logic chips in the last 20 years

Most chips are designed using at least some logic synthesis

Logic Synthesis Tools

Mostly commercial tools

- Very difficult, complicated programs to write well
- Limited market
- Commercial products in $10k – $100k price range

Major vendors

- Synopsys Design Compiler, FPGA Express
- Cadence BuildGates
- Synplicity (FPGAs)
- Exemplar (FPGAs)

Academic tools

- SIS (UC Berkeley)

Logic Synthesis

Takes place in two stages:

1. Translation of Verilog (or VHDL) source to a netlist

   Register inference performed here

2. Optimization of the resulting netlist to improve speed and area

   Most critical part of the process

   Algorithms very complicated and beyond the scope of this class: Take Prof. Nowick's class for details

Logic Optimization

Netlist optimization the critical enabling technology

Takes a slow or large netlist and transforms it into one that implements the same function more cheaply

Typical operations:

- Constant propagation
- Common subexpression elimination
- Function factoring

Time-consuming operation. Can take hours for large chips

Translating Verilog into Gates

Parts of the language easy to translate

Structural descriptions with primitives is already a netlist

Continuous assignment expressions turn into little datapaths

Behavioral statements the bigger challenge

What Can Be Translated

Every structural defi nition

Behavioral blocks

- Depends on sensitivity list
- Only when they have reasonable interpretation as combinational logic, edge, or level-sensitive latches
- Blocks sensitive to both edges of the clock, changes on unrelated signals, changing sensitivity lists, etc. cannot be synthesized

User-defi ned primitives

- Primitives defi ned with truth tables
- Some sequential UDPs can't be translated (not latches or flp-fbps)

What Is Not Translated

Initial blocks

- Used to set up initial state or describe fi nite testbench stimuli
- Don't have obvious hardware component

Delays

- May be in the Verilog source, but are simply ignored

A variety of other obscure language features

- In general, things heavily dependent on discrete-event simulation semantics
- Certain "disable" statements
- Pure events

Register Inference

The main trick

A **reg** is not always a latch or flip-flop

Rule: Combinational if outputs always depend exclusively on sensitivity list

Sequential if outputs may also depend on previous values

Register Inference

Combinational:

```
reg y;
always @(a or b or sel)   <- Sensitive to changes on all the variable it reads
  if (sel) y = a;
  else y = b;             <- y is always assigned
```

Sequential:

```
reg q;
always @(d or clk)
  if (clk) q = d;         <- q only assigned when clk is 1
```

Register Inference

A common mistake is not completely specifying a **case** statement

This implies a latch:

```
always @(a or b)
case ({a, b})
  2'b00 : f = 0;
  2'b01 : f = 1;
  2'b10 : f = 1;
endcase   <- f is not assigned when {a,b}= 2'b11
```

Register Inference

The solution is to always have a default case

```
always @(a or b)
case ({a, b})
  2'b00 : f = 0;
  2'b01 : f = 1;
  2'b10 : f = 1;
  default : f = 0;   <- f is always assigned
endcase
```

Inferring Latches with Reset

Latches and Flip-flops often have reset inputs

Can be synchronous or asynchronous

Asynchronous positive reset:

```
always @(posedge clk or posedge reset)
  if (reset)
   q <= 0;
  else q <= d;
```

Simulation-synthesis Mismatches

Many possible sources of conflict

- Synthesis ignores delays (e.g., #10), but simulation behavior can be affected by them
- Simulator models X explicitly, synthesis does not
- Behaviors resulting from shared-variable-like behavior of regs is not synthesized:

  ```
  always @(posedge clk) a = 1;
  ```

  New value of a may be seen by other @(posedge clk) statements in simulation, never in synthesis

Summary

Summary of Verilog

Systems described hierarchically

- Modules with interfaces
- Modules contain instances of primitives, other modules
- Modules contain initial and always blocks

Based on discrete-event simulation semantics

- Concurrent processes with sensitivity lists
- Scheduler runs parts of these processes in response to changes

Modeling Tools

Switch-level primitives: CMOS transistors as switches that move around charge

Gate-level primitives: Boolean logic gates

User-defined primitives: Gates and sequential elements defined with truth tables

Continuous assignment: Modeling combinational logic with expressions

Initial and always blocks: Procedural modeling of behavior

Language Features

Nets (wires) for modeling interconnection

- Non state-holding
- Values set continuously

Regs for behavioral modeling

- Behave exactly like memory for imperative modeling
- Do not always correspond to memory elements in synthesized netlist

Blocking vs. nonblocking assignment

- Blocking behaves like normal “C-like” assignment
- Nonblocking delays update, modeling synchronous behavior

Language Uses

Event-driven simulation

- Event queue containing things to do at particular simulated times
- Evaluate and update events
- Compiled-code event-driven simulation for speed

Logic synthesis

- Translating Verilog (structural and behavioral) into netlists
- Register inference: whether output is always updated
- Logic optimization for cleaning up the result

Little-used Language Features

Switch-level modeling

- Much slower than gate or behavioral-level models
- Insufficient detail for modeling most electrical problems
- Delicate electrical problems simulated with a SPICE-like differential equation simulator

Little-used Language Features

Delays

- Simulating circuits with delays does not improve confidence enough
- Hard to get timing models accurate enough
- Never sure you have simulated the worst case
- Static timing analysis has taken its place

Compared to VHDL

Verilog and VHDL are comparable languages

VHDL has a slightly wider scope

- System-level modeling
- Exposes even more discrete-event machinery

VHDL is better-behaved: Fewer sources of nondeterminism (e.g., no shared variables)

VHDL is harder to simulate quickly

VHDL has fewer built-in facilities for hardware modeling

VHDL is a much more verbose language: Most examples don’t fit on slides

In Conclusion

Verilog is a deeply flawed language

- Nondeterministic
- Often weird behavior due to discrete-event semantics
- Vaguely defined synthesis subset
- Many possible sources of simulation/synthesis mismatch

In Conclusion

Verilog is widely used because it solves a problem

- Good simulation speed that continues to improve
- Designers use a well-behaved subset of the language
- Makes a reasonable specification language for logic synthesis
- Logic synthesis one of the great design automation success stories