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)
- Combines structural and behavioral modeling styles

Outline

- Introduction
- Basics of the Verilog Language
- Operators
- Hierarchy/Modules
- Procedures and Assignments
- Timing Controls and Delay
- Control Statement
- Logic-Gate Modeling
- Modeling Delay
- Other Verilog Features
- Summary

Multiplexer Built From Primitives

```verilog
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
```
Multiplexer Built From Primitives

```verilog
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
```

Multiplexer Built With Always

```verilog
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
```

Multiplexer Built With Always

```verilog
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
```

Mux with Continuous Assignment

```verilog
module mux(f, a, b, sel);
  output f;
  input a, b, sel;
  assign f = sel ? a : b;
endmodule
```
Mux with User-Defined Primitive

```verilog
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

Introduction

Styles

- Structural - instantiation of primitives and modules
- RTL/Dataflow - continuous assignments
- Behavioral - procedural assignments

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

---

Style Example - Structural

```verilog
module full_add (S, CO, A, B, CI);
output S, CO;
input A, B, CI;
wire N1, N2, N3;
half_add HA1 (N1, N2, A, B), HA2 (S, N3, N1, CI);
or P1 (CO, N3, N2);
endmodule
```

```verilog
module half_add (S, C, X, Y);
output S, C;
input X, Y;
wire N1, N2, N3;
assign S = X ^ Y; //continuous assignment
assign C = X & Y; //continuous assignment
endmodule
```

---

Style Example – Dataflow/RTL

```verilog
module fa_rtl (S, CO, A, B, CI);
output S, CO;
input A, B, CI;
assign S = A ^ B ^ CI; //continuous assignment
assign CO = A & B | A & CI | B & CI; //continuous assignment
endmodule
```

---

Style Example – Behavioral

```verilog
module fa_bhv (S, CO, A, B, CI);
output S, CO;
input A, B, CI;
reg S, CO; // required to "hold" values between events.
always @(A or B or CI)
begin
  S = A ^ B ^ CI; //procedural assignment
  CO = A & B | A & CI | B & CI; //procedural assignment
endmodule
```

---
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

An Example: Counter

```verilog
module counter;
    reg clock; // declare reg data type for the clock
    integer count; // declare integer data type for the count

    initial // initialize things - this executes once at start
            begin
                clock = 0; count = 0; // initialize signals
                #340 $finish; // finish after 340 time ticks
            end

    /* an always statement to generate the clock, only one statement follows the always so we don't need a begin and an end */
    always
            #10 clock = ~ clock; // delay is set to half the clock cycle

    /* an always statement to do the counting, runs at the same time (concurrently) as the other always statement */
    always
            begin
                // wait here until the clock goes from 1 to 0
                @ (negedge clock);
                // now handle the counting
                if (count == 7)
                    count = 0;
                else
                    count = count + 1;
                $display("time = ", $time, " count = ", count);
            end

endmodule
```

Outline

- Introduction
- Basics of the Verilog Language
- Operators
- Hierarchy/Modules
- Procedures and Assignments
- Timing Controls and Delay
- Control Statement
- Logic-Gate Modeling
- Modeling Delay
- Other Verilog Features
- Summary
Basics of the Verilog Language

- Language Conventions
- Logic Values
- Data Types
- Wire Types
- Numbers
- Negative Numbers
- Strings

Language Conventions

- Case-sensitivity
  - Verilog is case-sensitive.
  - Some simulators are case-insensitive
  - Advice: Don't use case-sensitive feature!
  - Keywords are lower case
- Different names must be used for different items within the same scope
- Identifier alphabet:
  - Upper and lower case alphabeticals
  - decimal digits
  - underscore

Language Conventions (cont'd)

- Maximum of 1024 characters in identifier
- First character not a digit
- Statement terminated by ;
- Free format within statement except for within quotes
- Comments:
  - All characters after // in a line are treated as a comment
  - Multi-line comments begin with /* and end with */
- Compiler directives begin with // synopsys
- Built-in system tasks or functions begin with $
- Strings enclosed in double quotes and must be on a single line

Four-valued Logic

- Verilog's nets and registers hold four-valued data
- 0, 1
  - Logical Zero, Logical One
- Z
  - Output of an undriven tri-state driver – high-impedance value
  - Models case where nothing is setting a wire's value
- X
  - Models when the simulator can't decide the value – uninitialized or unknown logic 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 (cont'd)

- Logical operators work on three-valued logic

<table>
<thead>
<tr>
<th>Input</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Output 0 if one input is 0
Output X if both inputs are gibberish

Two Main Data Types

- 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

- 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
  - Shared variables with all their attendant problems

Data Types

- nets are further divided into several net types
  - wire, tri, supply0, ...
- registers - stores a logic value - reg
- integer - supports computation
- time - stores time 64-bit unsigned
- real - stores values as real numbers
- realtime - stores time values as real numbers
- event – an event data type

Wires and registers can be bits, vectors, and arrays

Nets and Registers (cont’d)

```verilog
module declarations_4;
wire Data;                            // a scalar net of type wire
wire [31:0] ABus, DBus;               // two 32-bit wide vector wires...
wire [31:0] Tbus, [15:0] BigBus;      // illegal
reg [3:0] vector;                     // a 4-bit vector register
reg [4:7] nibble;                     // msb index < lsb index
integer i;
initial begin
  i = 1;
  vector = 'b1010;                          // vector without an index
  nibble = vector;                         // this is OK too
  $display("T=%0g", $time, " vector=", vector," nibble="nibble);
  `$display("T=%0g", $time, " Bus=%b", DBus[15:0]);
end
assign DBus [1] = 1;                        // this is a bit-select
assign DBus [3:0] = 'b1111;                 // this is a part-select
// assign DBus [0:3] = 'b1111;               // illegal = wrong direction
endmodule
```
Nets and Registers (cont'd)

```
integer imem[0:1023];    // Array of 1024 integers
reg [31:0] dcache[0:63]; // A 64-word by 32-bit wide memory
time time_log[1:1000];  // as an array of regs
// real Illegal[1:10]; // Illegal. There are no real arrays.
```

Net Types

- **wire** - connectivity only
- **tri** - same as wire, but will be 3-state in hardware
- **wand** - multiple drivers - wired and
- **wor** - multiple drivers - wired or
- **triand** - same as wand, but 3-state
- **trior** - same as wor but 3-state
- **supply0** - Global net GND
- **supply1** - Global Net VCC (VDD)
- **trilo**, **tri1** - model resistive connections to VSS and VDD
- **trireg** - like wire but associates some capacitance with the net, so it can model charge storage

Declarations: An Example

```
module declarations_5;
    wire pwr_good, pwr_on, pwr_stable; // Explicitly declare wires
    integer i;                        // 32-bit, signed (2's complement)
    time t;                          // 64-bit, unsigned, behaves like a 64-bit reg
    event e;                         // Declare an event data type
    real r;                          // Real data type of implementation defined size

    // assign statement continuously drives a wire...
    assign pwr_stable = 1'b1; assign pwr_on = 1;      // 1 or 1'b1
    assign pwr_good = pwr_on & pwr_stable;

    initial begin
        $display("pwr_on=%b", pwr_on);
        i = 123.456;                     // There must be a digit on either side
        r = 123456e-3;                   // of the decimal point if it is present.
        t = 123456e-3;                   // Time is rounded to 1 second by default.
        $display("i=%g", i, " t=%6.2f", t, " r=%f", r);
        #2 $display("TIME=%d", $time, " ON=", pwr_on, " STABLE=", pwr_stable, " GOOD=", pwr_good);
    end
endmodule
```

Register Assignment

- A register may be assigned value only within:
  - a procedural statement
  - a user-defined sequential primitive
  - a task, or
  - a function.
- A reg object may never be assigned value by:
  - a primitive gate output or
  - a continuous assignment

Examples

```
reg a, b, c;
reg [15:0] counter, shift_reg;
integer sum, difference;
```
### Constants & Strings

**Constants**
- `parameter A = 2’b00, B = 2’b01, C = 2’b10;`
- `reg [regsize - 1:0]; /* illustrates use of parameter regsize */`

**Strings**
- No explicit data type
- Must be stored in reg (or array)

```verilog
reg [255:0] buffer; // stores 32 characters
parameter Tab = "\t"; // tab character
parameter NewLine = "\n"; // newline character
parameter BackSlash = "\\"; // back slash
```

### Number Representation

**Format:** `<size><base_format><number>`
- `<size>` - decimal specification of number of bits
  - default is unsized and machine-dependent but at least 32 bits
- `<base_format>` - `’` followed by arithmetic base of number
  - `<d> <D>` - decimal - default base if no `<base_format>` given
  - `<h> <H>` - hexadecimal
  - `<o> <O>` - octal
  - `<b> <B>` - binary
- `<number>` - value given in base of `<base_format>`
  - `_` can be used for reading clarity
  - If first character of sized, binary number 0, 1, x or z, will extend 0, 1, x or z (defined later)

**Examples:**
- `6b010_111` gives `010111`
- `8b0110` gives `000000110`
- `4bx01` gives `0001`
- `16'HAB` gives `0000001110101101`
- `24` gives `0.0011000`
- `5'O36` gives `11100`
- `16'Hx` gives `xxxxxxxxxxxxxxxxx`
- `8fhz` gives `xxxxxxxxxxxxxxxxxxxxxxx`

### Outline

- **Introduction**
- **Basics of the Verilog Language**
  - **Operators**
  - **Hierarchy/Modules**
  - **Procedures and Assignments**
  - **Timing Controls and Delay**
  - **Control Statement**
  - **Logic-Gate Modeling**
  - **Modeling Delay**
  - **Other Verilog Features**
- **Summary**
Operators

- Arithmetic (pair of operands, binary word)
  [binary: +, *, /, %]; [unary: +, -]
- Bitwise (pair of operands, binary word)
  [~, ~, &, |, ^, ~^, ^~]
- Reduction (single operand, bit) [&, ~&, |, ^, ~^, ^~]
- Logical (pair of operands, boolean value)
  [!, &&, ||, ==, !=, ===, !==]
- Relational (pair of operands, boolean value) [<, <=, >, >=]
- Shift (single operand, binary word) [>>, <<]
- Conditional ?: (three operands, expression)
- Concatenation and Replications {,} {int{}}

* unsupported for variables

---

Operators (cont’d)

- Relational
  - < (lt), <= (lte), > (gt), >= (gte)
- Shift
  - << (left shift), >> (right shift)
- Conditional
  - E.g.: Y = (A==B) ? A : B
  - wire[15:0] bus_a = drive_bus_a ? data : 16’bz;
- Concatenation
  - $a(a) = (a, a, a, a)$

---

Operators (cont’d)

- Arithmetic
  - + (addition), - (subtraction), * (multiplication), / (division), % (modulus)
- Bitwise
  - ~ (negation), & (and), | (or), ^ (xor), ~^ (xnor)
- Reduction
  - E.g.: &(0101) = 0
  - & (and), ~& (nand), | (or), ~| (nor), ^ (xor), ~^ (xnor)
- Logical
  - ! (negation), && (and), || (or), == (equality), != (inequality), === (case equality), !== (case inequality)
  - === : determines whether two words match identically on a bit-by-bit basis, including bits that have values “x” and “z”

---

Operators (cont’d)

- Relational
  - < (lt), <= (lte), > (gt), >= (gte)
- Shift
  - << (left shift), >> (right shift)
- Conditional
  - E.g.: Y = (A==B) ? A : B
  - wire[15:0] bus_a = drive_bus_a ? data : 16’bz;
- Concatenation
  - $a(a) = (a, a, a, a)$

---

Operators (cont’d)

- Arithmetic
  - + (addition), - (subtraction), * (multiplication), / (division), % (modulus)
- Bitwise
  - ~ (negation), & (and), | (or), ^ (xor), ~^ (xnor)
- Reduction
  - E.g.: &(0101) = 0
  - & (and), ~& (nand), | (or), ~| (nor), ^ (xor), ~^ (xnor)
- Logical
  - ! (negation), && (and), || (or), == (equality), != (inequality), === (case equality), !== (case inequality)
  - === : determines whether two words match identically on a bit-by-bit basis, including bits that have values “x” and “z”

---

Operators (cont’d)

- Relational
  - < (lt), <= (lte), > (gt), >= (gte)
- Shift
  - << (left shift), >> (right shift)
- Conditional
  - E.g.: Y = (A==B) ? A : B
  - wire[15:0] bus_a = drive_bus_a ? data : 16’bz;
- Concatenation
  - $a(a) = (a, a, a, a)$
Expression Bit Widths

- Depends on:
  - widths of operands and types of operators
  - Verilog fills in smaller-width operands by using zero extension.
  - Final or intermediate result width may increase expression width
  - Unsized constant number - same as integer (usually 32bit)
  - Sized constant number - as specified
  - x op y where op is +, *, /, %, &, |, ^, ~&:
    - Arithmetic binary and bitwise
    - Bit width = max (width(x), width(y))

- x ? y : z
  - Conditional
  - Bit width = max(width(y), width(z))

- {x, y}
  - Concatenation
  - Bit width = width(x) + ... + width(y)

- {x[y, ... z]}
  - Replication
  - Bit width = x * (width(y) + ... + width(z))

Expression Bit Widths (continued)
Expressions with Operands Containing x or z

- **Arithmetic**
  - If any bit is x or z, result is all x's.
  - Divide by 0 produces all x's.
- **Relational**
  - If any bit is x or z, result is x.
- **Logical**
  - == and !=: If any bit is x or z, result is x.
  - === and !==: All bits including x and z values must match for equality.

Expressions with Operands Containing x or z (cont’d)

- **Bitwise**
  - Defined by tables for 0, 1, x, z operands.
- **Reduction**
  - Defined by tables as for bitwise operators.
- **Shifts**
  - z changed to x. Vacated positions zero filled.
- **Conditional**
  - If conditional expression is ambiguous (e.g., x or z), both expressions are evaluated and bitwise combined as follows:
    - f(1,1) = 1, f(0,0) = 0, otherwise x.

Outline

- Introduction
- Basics of the Verilog Language
- Operators
- Hierarchy/Modules
- Procedures and Assignments
- Timing Controls and Delay
- Control Statement
- Logic-Gate Modeling
- Modeling Delay
- Other Verilog Features
- Summary

Modules

- Basic design units
  - Verilog program build from modules with I/O interfaces
- Modules are:
  - Declared
  - Instantiated
- Module interface is defined using ports
  - Each port must be explicitly declared as one of
    - input (wire or other net)
    - output (reg or wire; can be read inside the module)
    - inout (wire or other net)
- Modules declarations cannot be nested
- Modules may contain instances of other modules
- Modules contain local signals, etc.
- Module configuration is static and all run concurrently
Module Declaration

- Basic structure of a Verilog module:

```verilog
module mymod(output1, output2, ... input1, input2);
output output1;
output [3:0] output2;
input input1;
input [2:0] input2;
endmodule
```

Module Declaration (cont’d)

- Example:

```verilog
/* module_keyword module_identifier (list of ports) */
module C24DecoderWithEnable (A, E, D);
input [1:0] A; // input_declaration
input E; // input_declaration
output [3:0] D; // output_declaration
assign D = {4{E}} & ((A == 2’b00) ? 4’b0001 :
    (A == 2’b01) ? 4’b0010 :
    (A == 2’b10) ? 4’b0100 :
    (A == 2’b11) ? 4’b1000 :
    4’bxxxx) ; // continuous_assign
endmodule
```

Module Declaration (cont’d)

- Identifiers - must not be keywords!
- Ports
  - First example of signals
  - Scalar: e.g., E
  - Vector: e.g., A[1:0], A[0:1], D[3:0], and D[0:3]
    - Range is MSB to LSB
    - Can refer to partial ranges - D[2:1]
  - Type: defined by keywords
    - input
    - output
    - inout (bi-directional)

Module Instantiation

- Instances of

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

- look like

```verilog
mymod mm1(y1, a1, b1); // Connect-by-position
mymod (y2, a1, b1),
(y3, a2, b2); // Instance names omitted
mymod mm2(.a(a2), .b(b2), .y(c2)); // Connect-by-name
```
Example: 4/16 decoder using 2/4 decoders

```verilog
module C416DecoderWithEnable (A, E, D);
    input [3:0] A;
    input E;
    output [15:0] D;
    wire [3:0] S;
    C24DecoderWithEnable DE (A[3:2], E, S);
    C24DecoderWithEnable D0 (A[1:0], S[0], D[3:0]);
    C24DecoderWithEnable D1 (A[1:0], S[1], D[7:4]);
    C24DecoderWithEnable D2 (A[1:0], S[2], D[11:8]);
    C24DecoderWithEnable D3 (A[1:0], S[3], D[15:12]);
endmodule
```

Connections

- **Position association**
  - `C24DecoderWithEnable DE [A[3:2], E, S];`

- **Name association**

- **Empty Port Connections**
  - `E is at high impedance state (z)`
  - `// Outputs S[3:0] are unused`

Example:

- Single module instantiation for five module instances

```verilog
    C24DecoderWithEnable DE (A[3:2], E, S),
    D0 (A[1:0], S[0], D[3:0]),
    D1 (A[1:0], S[1], D[7:4]),
    D2 (A[1:0], S[2], D[11:8]),
    D3 (A[1:0], S[3], D[15:12]);
```

Empty Port Connections

- `// E is at high impedance state (z)`
- `C24DecoderWithEnable DE [A[3:2], E];`
- `// Outputs S[3:0] are unused`
- `C24DecoderWithEnable DE [A[3:2], E];`
Array of Instances

- `{ , }` is concatenate

Example

```verilog
module add_array (A, B, CIN, S, COUT);
input [7:0] A, B;
input CIN;
output [7:0] S;
output COUT;
wire [7:1] carry;
full_add FA[7:0] (A,B,{carry, CIN},S,{COUT, carry});
// full_add is a module
endmodule
```

Outline

- Introduction
- Basics of the Verilog Language
- Operators
- Hierarchy/Modules
- Procedures and Assignments
- Timing Controls and Delay
- Control Statement
- Logic-Gate Modeling
- Modeling Delay
- Other Verilog Features
- Summary

Procedures and Assignments

- Verilog procedures
  - initial and always statements
  - tasks
  - functions
- Sequential block: a group of statements that appear between a begin and an end
  - executed sequentially
  - considered as a statement - can be nested
- Procedures execute concurrently with other procedures
- Assignment statements
  - continuous assignments: appear outside procedures
  - procedural assignments: appear inside procedures

Assignments

- Continuous assignment
  ```verilog
  module holiday_1(sat, sun, weekend);
  input sat, sun; output weekend;
  assign weekend = sat | sun;     // outside a procedure
  endmodule
  ```
- Procedural assignment
  ```verilog
  module holiday_2(sat, sun, weekend);
  input sat, sun; output weekend; reg weekend;
  always #1 weekend = sat | sun;   // inside a procedure
  endmodule
  ```
  ```verilog
  module assignments
  // continuous assignments go here
  always begin
  // procedural assignments go here
  end
  ```
Continuous Assignment

- Another way to describe combinational function
- Convenient for logical or datapath specifications

```verilog
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;
assign sum = a + b + carryin;
```

Continuous assignment: permanently sets the value of sum to be \( a+b+\text{carryin} \)

Recomputed when \( a, b, \) or \( \text{carryin} \) changes

Sequential Block

- Sequential block may appear in an always or initial statement

```verilog
initial
begin
... imperative statements ...
end
```

Runs when simulation starts
Terminates when control reaches the end
(one time sequential activity flow)
Good for providing stimulus (testbenches); not synthesizable

```verilog
always @(posedge clk) q = d; // edge-sensitive ff
```

Initial and Always

- Run until they encounter a delay
- or a wait for an event

```verilog
initial begin
$10 a = 1; b = 0;
$10 a = 0; b = 1;
end
```

```verilog
always @(posedge clk) q = d; // edge-sensitive ff
```

```verilog
always begin
wait(i); a = 0;
wait(-i); a = 1;
end
```
module always_1; reg Y, Clk;
always // Statements in an always statement execute repeatedly:
begin: my_block // Start of sequential block.
@ (posedge Clk) #5 Y = 1; // At +ve edge set Y=1,
@ (posedge Clk) #5 Y = 0; // at the NEXT +ve edge set Y=0.
end // End of sequential block.
always #10 Clk = ~ Clk; // We need a clock.
initial Y = 0; // These initial statements execute
initial Clk = 0; // only once, but first.
initial $monitor("T=%2g,$time," Clk=", Clk," Y=",Y);
initial #70 $finish;
endmodule
Delay control

- **Timescale compiler directive**
  - `timescale 1ns/10ps // Units of time are ns. Round times to 10 ps.
  - Allowed unit/precision values: {110 | s | ms | us | ns | ps}

- **Intra-assignment delay vs. delayed assignment**
  - `x = $1 y; // Intra-assignment delay
  - `begin
  - `hold = y; // Sample and hold `y immediately.
  - `$1; // Delay.
  - `x = hold; // Assignment to `x. Overall same as `x = `$1 `y.
  - `end

  - `#1; // Delay.
  - `begin
  - `x = hold; // Assignment to `x. Overall same as x = #1 `y.
  - `end

Event control (cont’d)

```
module show_event;
    reg clock;
    event event_1, event_2; // Declare two named events.
    always @(posedge clock) -> event_1; // Trigger event_1.
    always @ event_1
begin
    $display("Strike 1!!");
    -> event_2; // Trigger event_2.
    end
    always @ event_2
begin
    $display("Strike 2!!");
    $finish; // Stop on detection of event_2.
    end
    always #10 clock = ~ clock; // We need a clock.
    initial clock = 0;
endmodule
```

```
T   Clk X Y
0   0  x x
10  1  1 1 1
20  0  1 1 1
30  1  1 1 1
40  0  1 1 1
50  0  1 1 1
60  1  1 1 1
70  1  1 1 1
80  0  1 1 1
90  1  1 1 1
```

Event control

- **posedge – 0 => 1, 0 => x, x => 1**
- **negedge – 1 => 0, 1 => x, x => 0**

```
event_control ::= @ event_identifier | @ (event_expression)

event_expression ::= expression | event_identifier
                    | posedge expression | negedge expression
                    | event_expression or event_expression
```

```
module delay_controls; reg X, Y, Clk, Dummy;
    always #1 Dummy=!Dummy; // Dummy clock, just for graphics.
```

```
module data_slip_1 ();
    reg Clk, D, Q1, Q2;
```

```
T   Clk D Q1 Q2
0   0  1 x x
50  1  1 1 1
100 0  1 1 1
150 1  1 1 1
200 0  1 1 1
250 1  1 1 1
300 0  1 1 1
350 1  1 1 1
```

```
always @ (posedge Clk)  Q1 = D; // The delays in the assgn.
always @ (posedge Clk)  Q2 = #1 Q1; // fix the data slip.
```

```
T   Clk D Q1 Q2
0   0  1 x x
50  1  1 x x
51  1  1 1  x
100 0  1 1  x
150 1  1 1  x
151 1  1 1  1
200 0  1 1  1
250 1  1 1  1
300 0  1 1  1
350 1  1 1  1
```

```
always @ (posedge Clk) Q1 = #1 D; // The delay in the assgn.
always @ (posedge Clk) Q2 = $1 Q1; // fix the data slip.
```

```
% Clk D Q1 Q2
0   0  1 a a
50  1  1 1 1
100 0  1 1 1
150 1  1 1 1
200 0  1 1 1
250 1  1 1 1
300 0  1 1 1
350 1  1 1 1
400 0  1 1 1
```
Wait Statement

- Suspends a procedure until a condition becomes true
  - there must be another concurrent procedure that alters the condition – otherwise we have an “infinite hold”

```verilog
df_wait test_dff_wait;
reg D, Clock, Reset;
df_wait u1(D, Q, Clock, Reset);
initial begin D=1; Clock=0;Reset=1'b1; #15 Reset=1'b0; #20 D=0; end
always #10 Clock = !Clock;
initial begin $display("T  Clk D Q Reset"); $monitor("%2g",$time,,Clock,,,,D,,Q,,Reset); #50 $finish;
end
endmodule
```

```verilog
df_wait D, Q, Clock, Reset;
output Q; input D, Clock, Reset; reg Q; wire D;
always @(posedge Clock) if (Reset !== 1) Q = D;
always begin wait (Reset == 1) Q = 0; wait (Reset !== 1);
end
endmodule
```

Blocking and Nonblocking Assignments

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

Blocking: assignments are evaluated in some order, but we do not know in what
Nonblocking: RHS evaluated when assignment runs

```verilog
reg d1, d2, d3, d4;
always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

Nonblocking assignment:

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

Blocking assignment:

```verilog
a = b = c = 1  a = 1;
b = old value of a  b <= a;
c = old value of b  c <= b;
```

Nonblocking Looks Like Latches

- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

Blocking: assignments are evaluated in some order, but we do not know in what
Nonblocking: RHS evaluated when assignment runs

```verilog
```

```
```
**Task and Functions**

- **Task** – type of a procedure called from another procedure
  - has inputs and outputs but does not return a value
  - may call other tasks and functions
  - may contain timing controls

- **Function** – procedure used in any expression
  - has at least one input, no outputs, and return a single value
  - may not call a task
  - may not contain timing controls

**Control Statements**

- **If statement**
  ```
  if (select == 1) y = a;
  else y = b;
  ```

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

- **Casez statement –** handles only z bits as don’t care
  ```
  casez (opcode)
    3'b??1: y = a + b;
    3'b?1?: y = a - b;
  endcase
  ```

- **Loop statements:**
  - for, while, repeat, forever
  ```
  integer i; reg [15:0] Dbus;
  initial Dbus = 0;
  // for loop
  for (i = 0 ; i <= 15 ; i = i + 1) begin
    Dbus[i] = 1;
  end
  // while loop
  while (i <= 15) begin
    Dbus[i] = 1;
    i = i + 1;
  end
  ```

- **Disable statement** - stops the execution of a labeled sequential block and skips to the end of the block
  ```
  forever
  begin: cpu_block
  @(posedge clock)
  if (reset) disable cpu_block;
  // Skip to end of block.
  else Execute_code;
  end
  ```

- **Fork statement and join statement** – allows execution of two or more parallel threads in a parallel block
  ```
  module fork_1
  event eat_breakfast, read_paper;
  initial begin
  fork
  @eat_breakfast; @read_paper;
  join
  end
  ```
Gate Level Modeling

- Verilog provides the following primitives:
  - `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
  - `bufif1`, `notif1` - Tristate with high enable
- No declaration; can only be instantiated
- All output ports appear in list before any input ports
- Optional drive strength, delay, name of instance

Gate-level Modeling (cont’d)

- Example:

```
and N25(Z, A, B, C); // instance name
and #10 (Z, A, B, X); // delay
(X, C, D, E); // delay
/* Usually better to provide instance name for debugging */
or N30(SET, Q1, AB, N5),
N41(N25, ABC, R1);
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 (UDPs)

- 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
  000 : 0;
  001 : 0;
  010 : 0;
  011 : 1;
  100 : 0;
  101 : 1;
  110 : 0;
  111 : 1;
endtable
endprimitive
```

Always have 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;

// 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

Shorthand notations:
- * is (??) - r is (01) - f is (10)
- p is (01), (0x), or (x1) - n is (10), (1x), (x0)

Switch-level Primitives (FIO)

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 these problems
These circuits are usually simulated using SPICE-like simulators based on nonlinear differential equation solvers
Switch Level
- * mos where * is p, c, m, rp, rc; pullup, pulldown; *tran*
  where * is (null), r and * (null), if0, if1 with both * and * not
  (null)

Delay Uses and Types

- Ignored by synthesizers; may be useful for simulation
- Uses
  - Behavioral (Pre-synthesis) Timing Simulation
  - Testbenches
  - Gate Level (Post-synthesis and Pre-Layout) Timing Simulation
  - Post-Layout Timing Simulation
- Types
  - Gate Delay (Inertial Delay)
  - Net Delay (Transport Delay)
  - Module Path Delay

Transport and Inertial Delay

- Transport delay - pure time delay
- Inertial delay
  - Multiple events cannot occur on the output in a time less than the delay.
- Example AND with delay = 2

A + 1 ns

B

C Transport Delay

C Inertial Delay
Gate Delay - Examples

- nd01 has a delay of 3 ns (assuming ns timescale) for both falling and rising delays
- nd02 has a triplet for the delay (min is 2.6 ns, typ is 3.0 ns, max is 3.4)
- nd03 has two triplets for the delay:
  - first triplet specifies min/typ/max for rising delay
  - second triplet specifies min/typ/max for falling delay
- For primitives which can produce high-impedance output, we can specify turn-off triplet

Net Delay (Transport)

- Example - Continuous Assignment
  - For rising output from x1 to N25, 200 + 40 = 240 ps

- Example - Implicit Continuous Assignment
  - For rising output from x1 to N25, 240 ps

Module Delay

- Example: norf201 - 3-input nor gate from a 1.2um CMOS

```
module norf201(o, a1, b1);
output o;
input a1, b1;
nor(o, a1, b1);
specify // module paths
  (a1, b1 **) o = (0.179:0.349:0.883, 0:084:0.169:0.466);
endspecify;
endmodule;
```

Outline

- Introduction
- Basics of the Verilog Language
- Operators
- Hierarchy/Modules
- Procedures and Assignments
- Timing Controls and Delay
- Control Statement
- Logic-Gate Modeling
- Modeling Delay
- Other Verilog Features
  - Summary

Modeling delay
### Altering Parameters

- **Use parameter**
  ```verilog
  module Vector_And(Z, A, B);
  parameter CARDINALITY = 1;
  input [CARDINALITY-1:0] A, B;
  output [CARDINALITY-1:0] Z;
  wire [CARDINALITY-1:0] z = A & B;
  endmodule
  ```

- **Override the parameter in instantiation**
  ```verilog
  module Four_And_Gates(OutBus, InBusA, InBusB);
  Vector_And #(4) My_AND(OutBus, InBusA, InBusB); // 4 AND gates
  endmodule
  ```

- **Or using defparam**
  ```verilog
  module And_Gates(OutBus, InBusA, InBusB);
  parameter WIDTH = 1;
  input [WIDTH-1:0] InBusA, InBusB; output [WIDTH-1:0] OutBus;
  Vector_And #(WIDTH) My_And(OutBus, InBusA, InBusB);
  endmodule
  ```

```
module Super_Size; defparam And_Gates.WIDTH = 4; endmodule
```

### 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

```verilog
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
      nextState = a ? 2'b00 : 2'b01;
      o = a & b;
    end
    2'b01: begin
      nextState = 2'b10;
      o = 0;
    end
  endcase
endmodule
```

#### FSM with Combinational Logic

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

Combined 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 Combinational Logic**

```
always @(a or b or state) 
  case (state) 
    2'b00: begin 
      nextState = a ? 2'b00 : 2'b01; 
      o = a & b; 
    end 
    2'b01: begin 
      nextstate = 2'b10; 
      o = 0; 
    end 
  endcase 
always @(posedge clk or reset) 
  if (reset) 
    state <= 2'b00; 
  else 
    state <= nextstate; 
```

This is a Mealy machine because the output is directly affected by any change on the input.

**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 
endmodule 
```

Expresses Moore machine behavior: Outputs are latched. Inputs are 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 event driven "printf" Stimulus generated by sequence of assignments and delays.

**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.

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

\[ a \leftarrow b + c \]

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

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 (cont’d)

- 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 (cont’d)

- 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;
```
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

- 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

- Takes place in two stages:
  - Translation of Verilog (or VHDL) source to a netlist
    - Register inference
  - 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

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
    - Already a netlist
  - Continuous assignment
    - Expressions turn into little datapaths
- Behavioral statements the bigger challenge

What Can Be Translated

- Structural definitions
  - Everything
- 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-defined primitives
  - Primitives defined with truth tables
  - Some sequential UDPs can’t be translated (not latches or flip-flops)

What Isn’t Translated

- Initial blocks
  - Used to set up initial state or describe finite 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
  - reg does not always equal latch
- Rule: Combinational if outputs always depend exclusively on sensitivity list
- Sequential if outputs may also depend on previous values
Register Inference

- Combinational:
  ```verilog
  reg y;
  always @(a or b or sel)
  if (sel) y = a;
  else y = b;
  ```

- Sequential
  ```verilog
  reg q;
  always @(d or clk)
  if (clk) q = d;
  ```

Other Verilog features

- A common mistake is not completely specifying a case statement.
- This implies a latch:
  ```verilog
  always @(a or b)
  case ({a, b})
  2'b00 : f = 0;
  2'b01 : f = 1;
  2'b10 : f = 1;
  endcase
  ```

Inferring Latches with Reset

- Latches and Flip-flops often have reset inputs
- Can be synchronous or asynchronous
- Asynchronous positive reset:
  ```verilog
  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 doesn't
- 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

Outline

- Introduction
- Basics of the Verilog Language
- Operators
- Hierarchy/Modules
- Procedures and Assignments
- Timing Controls and Delay
- Control Statement
- Logic-Gate Modeling
- Modeling Delay
- Other Verilog Features
- 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 updates later for 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
- Delays
  - Simulating circuits with delays does not improve confidence enough
  - Hard to get timing models accurate enough
  - Never sure you’ve 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