The Verilog Hardware Description Language

- These slides were created by Prof. Don Thomas at Carnegie Mellon University, and are adapted here with permission.

Verilog Overview

- Verilog is a concurrent language
  - Aimed at modeling hardware — optimized for it!
  - Typical of hardware description languages (HDLs), it:
    - provides for the specification of concurrent activities
    - stands on its head to make the activities look like they happened at the same time
    - Why?
      - allows for intricate timing specifications
  - A concurrent language allows for:
    - Multiple concurrent “elements”
    - An event in one element to cause activity in another. (An event is an output or state change at a given time)
      - based on interconnection of the element’s ports
    - Further execution to be delayed
      - until a specific event occurs

Simulation of Digital Systems

- Simulation —
  - What do you do to test a software program you write?
    - Give it some inputs, and see if it does what you expect
    - When done testing, is there any assurance the program is bug free? — NO!
      - But, to the extent possible, you have determined that the program does what you want it to do
  - Simulation tests a model of the system you wish to build
    - Is the design correct? Does it implement the intended function correctly? For instance, is it a UART
      - Stick in a byte and see if the UART model shifts it out correctly
      - Also, is it the correct design?
      - Might there be some other functions the UART could do?

Simulation checks two properties

- functional correctness — is the logic correct
  - correct design, and design correct
- timing correctness — is the logic/interconnect timing correct
  - e.g. are the set-up times met?

It has all the limitations of software testing

- Have I tried all the cases?
- Have I exercised every path? Every option?
Modern Design Methodology

Simulation and Synthesis are components of a design methodology:
- Synthesis
- Technology Mapping
- Place and Route

Verilog Levels of Abstraction

- Gate modeling (Structural modeling)
  - the system is represented in terms of primitive gates and their interconnections
    - NANDs, NORs, ...

- Behavioral modeling
  - the system is represented by a program-like language

Representation: Structural Models

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.

- Identify:
  - Gate instances, wire names, delay from a or b to f
  - This is a multiplexer — it selects one of n inputs (2 here) and passes it on to the output

```verilog
module MUX (f, a, b, sel);
    output f;
    input a, b, sel;
    and #5 g1 (f1, a, nsel),
    g2 (f2, b, sel);
    or #5 g3 (f, f1, f2);
    not g4 (nsel, sel);
endmodule
```

Representation: Gate-Level Models

- Need to model the gate’s:
  - Function
  - Delay

- Function
  - Generally, HDLs have built-in gate-level primitives
    - Verilog has NAND, NOR, AND, OR, XOR, XNOR, BUF, NOT, and some others
  - The gates operate on input values producing an output value
    - typical Verilog gate instantiation is:

```verilog
and #5 g1 (f1, a, nsel),
and #5 g2 (f2, b, sel);
and #5 g3 (f, f1, f2);
and #5 g4 (nsel, sel);
endmodule
```
Four-Valued Logic

Verilog Logic Values
- The underlying data representation allows for any bit to have one of four values
  - 1, 0, x (unknown), z (high impedance)
  - x — one of: 1, 0, z, or in the state of change
  - z — the high impedance output of a tri-state gate.

What basis do these have in reality?
- 0, 1 ... no question
- z ... A tri-state gate drives either a zero or one on its output. If it's not doing that, its output is high impedance (z). Tri-state gates are real devices and z is a real electrical affect.
- x ... not a real value. There is no real gate that drives an x on to a wire. x is used as a debugging aid. x means the simulator can't determine the answer and so maybe you should worry!

BTW ...
- some simulators keep track of more values than these. Verilog will in some situations.

Logic with multi-level logic values
- Logic with these four values make sense
  - NAND anything with a 0, and you get a 1. This includes having an x or z on the other input. That's the nature of the nand gate
  - NAND two x's and you get an x
- Note: z treated as an x on input. Their rows and columns are the same
- If you forget to connect an input ... it will be seen as an z.
- At the start of simulation, everything is an x.

<table>
<thead>
<tr>
<th>Nand</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

A 4-valued truth table for a Nand gate with two inputs

How to build and test a module

Construct a “test bench” for your design
- Develop your hierarchical system within a module that has input and output ports (called “design” here)
- Develop a separate module to generate tests for the module (“test”)
- Connect these together within another module (“testbench”)

module testbench ();
  wire l, m, n;
  design d (l, m, n);
  test t (l, m);
  initial begin
    //monitor and display ...
  
module design (a, b, c);
  input a, b, c;
  output ...;

module test (q, r);
  output q, r;
  initial begin
    //drive the outputs with signals ...

module testbench ();
  wire l, m, n;
  design d (l, m, n);
  test t (l, m);
  initial begin
    //monitor and display ...

Another view of this

3 chunks of Verilog, one for each of:

- TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...
- Another piece of hardware, called TEST, to generate interesting inputs
- Your hardware called DESIGN
An Example

Module testAdd generates inputs for module halfAdd and displays changes. Module halfAdd is the design.

module tBench;
    wire su, co, a, b;
    halfAdd ad(su, co, a, b);
    testAdd tb(a, b, su, co);
endmodule

module halfAdd (sum, cOut, a, b);
    output sum, cOut;
    input a, b;
    xor #2 (sum, a, b);
    and #2 (cOut, a, b);
endmodule

module testAdd (a, b, sum, cOut);
    input sum, cOut;
    output a, b;
    reg a, b;
    initial begin
        $monitor ($time, ,"a=%b, b=%b, sum=%b, cOut=%b", a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule

The test module

It’s the test generator

$monitor
- prints its string when executed.
- after that, the string is printed when one of the listed values changes.
- only one monitor can be active at any time.
- prints at end of current simulation time

Function of this tester
- at time zero, print values and set a=b=0
- after 10 time units, set b=1
- after another 10, set a=1
- after another 10 set b=0
- then another 10 and finish

Another Version of a Test Module

Multi-bit constructs
- test is a two-bit register and output
- it acts as a two-bit number (counts 00-01-10-11-00...)
- module tBench needs to connect it correctly — mod halfAdd has 1-bit ports.

module tBench;
    wire su, co;
    halfAdd ad(su, co, t[1], t[0]);
    testAdd tb(t, su, co);
endmodule

module testAdd (test, sum, cOut);
    input sum, cOut;
    output [1:0] test;
    reg [1:0] test;
    initial begin
        $monitor ($time, ,"test=%b, sum=%b, cOut=%b", test, sum, cOut);
        test = 0;
        #10 test = test + 1;
        #10 test = test + 1;
        #10 test = test + 1;
        #10 $finish;
    end
endmodule

Yet Another Version of testAdd

Other procedural statements
- You can use "for", "while", "if-then-else" and others here.
- This makes it easier to write if you have lots of input bits.

module tBench;
    wire su, co;
    halfAdd ad(su, co, t[1], t[0]);
    testAdd tb(t, su, co);
endmodule

module testAdd (test, sum, cOut);
    input sum, cOut;
    output [1:0] test;
    reg [1:0] test;
    initial begin
        $monitor ($time, ,"test=%b, sum=%b, cOut=%b", test, sum, cOut);
        for (test = 0; test < 3; test = test + 1) #10;
        #10 $finish;
    end
endmodule

hmm… “<3” …?
Other things you can do

More than modeling hardware

- $monitor — give it a list of variables. When one of them changes, it prints the information. Can only have one of these active at a time.
  
  e.g.,
  
  $monitor ($time,, "a=%b, b=%b, sums=%b, cOut=%b", a, b, sum, cOut);
  
  - The above will print:
    2   a=0, b=0, sum=0, cOut=0

- $display — sort of like printf()
  
  - $display ("Hello, world — %h", hexvalue)

- $display() — sort of like printf()
  
  - $display ("Hello, world — %h", hexvalue)
  
- $display() — sort of like printf()
  
  - $display ("Hello, world — %h", hexvalue)

Structural vs Behavioral Models

- Structural model
  
  - Just specifies primitive gates and wires
  
  - i.e., the structure of a logical netlist
  
  - You basically know how to do this now.

- Behavioral model
  
  - More like a procedure in a programming language
  
  - Still specify a module in Verilog with inputs and outputs...
  
  - ...but inside the module you write code to tell what you want to have happen, NOT what gates to connect to make it happen
  
  - i.e., you specify the behavior you want, not the structure to do it

Why use behavioral models

- For testbench modules to test structural designs
- For high-level specs to drive logic synthesis tools

How do behavioral models fit in?

How do they work with the event list and scheduler?

- Initial (and always) begin executing at time 0 in arbitrary order
- They execute until they come to a "#delay" operator
- They then suspend, putting themselves in the event list 10 time units in the future (for the case at the right)

- At 10 time units in the future, they resume executing where they left off.

- Some details omitted
  
  - ...more to come

Two initial statements?

- Two initial statements?

  module testAdd(a, b, sum, cOut);
  input sum, cOut;
  output a, b;
  reg a, b;
  initial begin
    $monitor ($time,, "a=%b, b=%b, sums=%b, cOut=%b", a, b, sum, cOut);
    a = 0; b = 0;
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
    #10 $finish;
  end
endmodule

Things to note

- Which initial statement starts first?
- What are the values of a, b, and out when the simulation starts?
- These appear to be executing concurrently (at the same time). Are they?
Behavioral Modeling

- Procedural statements are used
  - Statements using "initial" and "always" Verilog constructs
  - Can specify both combinational and sequential circuits

- Normally don’t think of procedural stuff as “logic”
  - They look like C: mix of ifs, case statements, assignments ...
  - ... but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)

Behavioral Constructs

- Behavioral descriptions are introduced by initial and always statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Looks like</th>
<th>Starts</th>
<th>How it works</th>
<th>Use in Synthesis?</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>begin ... end</td>
<td>Starts when simulation starts</td>
<td>Continually loop—while (power on) do statements;</td>
<td>Not used in synthesis</td>
</tr>
<tr>
<td>always</td>
<td>always begin ... end</td>
<td></td>
<td></td>
<td>Used in synthesis</td>
</tr>
</tbody>
</table>

Points:
- They all execute concurrently
- They contain behavioral statements like if-then-else, case, loops, functions, ...

Statements, Registers and Wires

- Registers
  - Define storage, can be more than one bit
  - Can only be changed by assigning value to them on the left-hand side of a behavioral expression.

- Wires (actually “nets”)
  - Electrically connect things together
  - Can be used on the right-hand side of an expression
  - Thus we can tie primitive gates and behavioral blocks together!

- Statements
  - left-hand side = right-hand side
  - left-hand side must be a register
  - Four-valued logic

Behavioral Statements

- if-then-else
  - What you would expect, except that it’s doing 4-valued logic. 1 is interpreted as True; 0, x, and z are interpreted as False

- case
  - What you would expect, except that it’s doing 4-valued logic
  - If "selector" is 2 bits, there are 4^2 possible case-items to select between
  - There is no break statement — it is assumed.

- Funny constants?
  - Verilog allows for sized, 4-valued constants
  - The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.
  - 8'b0000dx10

Logic with registers and wires

Multi-bit registers and wires

module silly (q, r);
reg [3:0] a, b;
wire [3:0] q, r;
always begin
  a = (b & r) | q;
  q = b;
end
endmodule
Behavioral Statements

- **Loops**
  - There are restrictions on using these for synthesis — don’t.
  - They are mentioned here for use in test modules
- **Two main ones — for and while**
  - Just like in C
  - There is also repeat and forever

```verilog
reg [3:0] testOutput, i;

for (i = 0; i <= 15; i = i + 1) begin
  testOutput = i;
  #20;
end
```

**Important:** Loops must have a delay operator (or as we’ll see later, an @ or wait(FALSE)). Otherwise, the simulator never stops executing them.

Test Module, continued

- **Bit Selects and Part Selects**
  - This expression extracts bits or ranges of bits or a wire or register

```verilog
module testgen (i[3], i[2], i[1], i[0]);
reg [3:0] i; output i;
always
  for (i = 0; i <= 15; i = i + 1)
    #20;
endmodule
```

Alternate:

```verilog
module testgen (i);
reg [3:0] i; output i;
always
  for (i = 0; i <= 15; i = i + 1)
    #20;
endmodule
```

Concurrent Constructs

- We already saw #delay
- **Others**
  - @ ... Waiting for a change in a value — used in synthesis
    - @ (var) w = 4;
    - This says wait for var to change from its current value. When it does, resume execution of the statement by setting w = 4.
  - Wait ... Waiting for a value to be a certain level — not used in synthesis
    - wait (f == 0) q = 3;
    - This says wait if f is equal to zero, then continue executing and set q = 3.
    - But if f is not equal to zero, then suspend execution until it does. When it does, this statement resumes by setting q = 3.

FAQs: behavioral model execution

- How does an always or initial statement start
  - That just happens at the start of simulation — arbitrary order
- Once executing, what stops it?
  - Executing either a #delay, @event, or wait(FALSE).
  - All always blocks need to have at least one of these. Otherwise, the simulator will never stop running the model — (it’s an infinite loop!)
- How long will it stay stopped?
  - Until the condition that stopped it has been resolved
    - #delay ... until the delay time has been reached
    - @ (var) ... until var changes
    - wait (var) ... until var becomes TRUE
- Does time pass when a behavioral model is executing?
  - No. The statements (if, case, etc) execute in zero time.
  - Time passes when the model stops for #, @, or wait.
- Will an always stop looping?
  - No. But an initial will only execute once.
Using a case statement

- Truth table method
  - List each input combination
  - Assign to output(s) in each case item.

- Concatenation
  - (a, b, c) concatenates a, b, and c together, considering them as a single item
  - Example:
    - a = 4'b0111
    - b = 6'b 1x0001
    - c = 2'bzx
    - then {a, b, c} = 12'b01111x0001zx

Example:

```verilog
module fred (f, a, b, c);
    output f;
    input a, b, c;
    reg f;
    always @ (a or b or c)
        case ({a, b, c})
            3'b000: f = 1'b0;
            3'b001: f = 1'b1;
            3'b010: f = 1'b1;
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b101: f = 1'b0;
            3'b110: f = 1'b0;
            3'b111: f = 1'b1;
        endcase
endmodule
```

How about a Case Statement Ex?

- Here's another version ...

```verilog
module fred (f, a, b, c);
    output f;
    input a, b, c;
    reg f;
    always @ (a or b or c)
        case ({a, b, c})
            3'b000: f = 1'b0;
            3'b101: f = 1'b0;
            3'b110: f = 1'b0;
            default: f = 1'b1;
        endcase
endmodule
```

Check the rules...

Two inputs, Three outputs

```verilog
reg [1:0] newJ;
reg input i, j;
always @(i or j)
    case (j)
        2'b00: begin
            newJ = (i == 0) ? 2'b00 : 2'b01;
            out = 0;
        end
        2'b01: begin
            newJ = (i == 0) ? 2'b10 : 2'b01;
            out = 1;
        end
        2'b10: begin
            newJ = 2'b00;
            out = 0;
        end
        default: begin
            newJ = 2'b00;
            out = 1'bx;
        end
    endcase
endmodule
```

Behavioral Timing Model

- How does the behavioral model advance time?
  - # — delaying a specific amount of time
  - $@$ — delaying until an event occurs ("posedge", "negedge", or any change) — this is edge-sensitive behavior
  - wait — delaying until an event occurs ("wait (f == 0)") — this is level sensitive behavior

- What is a behavioral model sensitive to?
  - any change on any input? — No
  - any event that follows, say, a "posedge" keyword
    - e.g. @posedge clock
    - Actually "not" here too. — not always

Important: every control path is specified

Check the rules...

Works like the C conditional operator. (exp) ? a : b;

If the expr is true, then the resulting value is a, else it's b.
What are behavioral models sensitive to?

Quick example
- Gate A changes its output, gates B and C are evaluated to see if their outputs will change, if so, their fanouts are also followed...
- The behavioral model will only execute if it was waiting for a change on the A input

```
always @(A)
begin
  B = ~A;
end
```

```
always @(posedge clock)
  Q <= A;
```

This would execute

This wouldn't

Order of Execution

In what order do these models execute?
- Assume A changes. Is B, C, or the behavioral model executed first?
- Answer: the order is defined to be arbitrary
- All events that are to occur at a certain time will execute in an arbitrary order.
- The simulator will try to make them look like they all occur at the same time — but we know better.

```
always @(A)
begin
  yadda yadda
end
```

Arbitrary Order? Oops!

Sometimes you need to exert some control
- Consider the interconnections of this D-FF
- At the positive edge of c, what models are ready to execute?
- Which one is done first?

```
module dff(q, d, c);
  ...
  always @(posedge c)
    q = d;
  endmodule
```

```
module sreg (...);
  ...
  always @ (posedge clock)
    a (q0, shiftin, clock),
    b (q1, q0, clock),
    c (shiftout, q1, clock);
  endmodule
```

Oops — The order of execution can matter!

Behavioral Timing Model

How does the behavioral model advance time?
- # — delaying a specific amount of time
- @ — delaying until an event occurs — e.g. @v
  - "posedge", "negedge", or any change
  - this is edge-sensitive behavior
  - When the statement is encountered, the value v is sampled. When v changes in the specified way, execution continues.
- wait — delaying until an event occurs ("wait (f == 0)")
  - this is level sensitive behavior
- While one model is waiting for one of the above reasons, other models execute — time marches on
**Wait**

- **Wait** — waits for a level on a line
  - How is this different from an “@”?

**Semantics**

- `wait (expression)` statement:
  - e.g. `wait (a == 35) q = q + 4;`
  - if the expression is FALSE, the process is stopped
  - when `a` becomes 35, it resumes with `q = q + 4`
  - if the expression is TRUE, the process is not stopped
  - it continues executing

**Partial comparison to @ and #**

- @ and # always “block” the process from continuing
- wait blocks only if the condition is FALSE

**An example of wait**

```verilog
module handshake (ready, dataOut, ...)
    input ready;
    output [7:0] dataOut;
    reg [7:0] someValueWeCalculated;
    always begin
        wait (ready);
        dataOut = someValueWeCalculated;
        ...
    end
endmodule
```

Do you always get the value right when ready goes from 0 to 1? Isn’t this edge behavior?

**Wait vs. While**

- **Are these equivalent?**
  - No: The left example is correct, the right one isn’t — it won’t work
  - Wait is used to wait for an expression to become TRUE
    - the expression eventually becomes TRUE because a variable in the expression is changed by another process
  - While is used in the normal programming sense
    - in the case shown, if the expression is TRUE, the simulator will continuously execute the loop. Another process will never have the chance to change “in”. infinite loop!
    - while can’t be used to wait for a change on an input to the process. Need other variable in loop, or # or @ in loop.

```verilog
module yes (in,...);
    input in;
    ...
    wait (in == 1);   // intra-assignment time delay
    ...
endmodule
```

```verilog
module no (in,...);
    input in;
    ...
    while (in != 1); // how is this done?
    ...
endmodule
```

**Blocking procedural assignments and #**

- We’ve seen blocking assignments — they use =
  - Options for specifying delay
    - `#10 a = b + c;`
    - `a = #10 b + c;`
    - The differences:

  - Note the action of the second one:
    - an *intra-assignment* time delay
      - execution of the always statement is blocked (suspended) in the middle of the assignment for 10 time units.
      - how is this done?
Events — @something

Action
- when first encountered, sample the expression
- wait for expression to change in the indicated fashion
- This always blocks

Examples

```verilog
always @(posedge clk)
  q <= d;
always @(hello or goodbye)
  a = b;
always @(@posedge hello or negedge goodbye)
  a = b;
always begin
  yadda = yadda;
  @(@posedge hello or negedge goodbye)
  a = b;
  ...
end
```