This is not one cohesive presentation on Verilog. The slides contained here are collected from different CMU classes at various academic levels.

These slides are provided as an alternate aid to learning the language. You may find them helpful.

Send bug reports to the above address — there are some!


University faculty wanting access to a PowerPoint version of the slides should contact the author at the above address.
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.

always
mumble
mumble
blah
blah

Synthesizable Verilog

Synthesis
gates, gates, gates, ...

Technology
Mapping

Place
and
Route

clb 1
clb 2

© Don Thomas, 1998
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.
  - You could describe your lab1 circuit this way

Identify:
- Gate instances, wire names, delay from \( a \) or \( b \) to \( f \).

```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```
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 #delay instance-name (out, in1, in2, in3, ...);
```

optional

“many”
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.
Four-Valued Logic

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.

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

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
</tr>
</tbody>
</table>
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 design (a, b, c);
    input a, b;
    output c;
    ...
```

```
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
    ...
```
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
Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was 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

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
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, sum=%b, cOut=%b”,a, b, sum, cOut);

  extra commas
  print a spaces
  %b is binary (also, %h, %d and others)

  The above will print:
  2 a=0, b=0, sum=0, cOut=0<return>

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

  display contents of data item called “hexvalue” using hex digits (0-9,A-F)

What if what you print has the value x or z?

newline automatically included
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 (Lab 2)
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

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 a = 1;
    #10 b = 0;
    #10 $finish;
  end
endmodule
Two initial statements?

... initial begin
  a = 0; b = 0;
  #5 b = 1;
  #13 a = 1;
end

... initial begin
  out = 1;
  #10 out = 0;
  #8 out = 1;
end

- 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?
What do we mean by “Synthesis”?  

Logic synthesis
- A program that “designs” logic from abstract descriptions of the logic
  - takes constraints (e.g. size, speed)
  - uses a library (e.g. 3-input gates)

How?
- You write an “abstract” Verilog description of the logic
- The synthesis tool provides alternative implementations
An example

What’s cool?
- You type the left, synthesis gives you the gates
- It used a different library than you did. (2-input gates only)
- One description suffices for a variety of alternate implementations!

Hmmm …
- … but this assumes you know a gate level implementation — that’s not an “abstract” Verilog description.

module gate (f, a, b, c);
    output f;
    input a, b, c;
    and A (a1, a, b, c),
        B (a2, a, ~b, ~c),
        C (a3, ~a, o1);
    or D (o1, b, c),
        E (f, a1, a2, a3);
endmodule

```verilog
module gate (f, a, b, c);
  output f;
  input a, b, c;
  and A (a1, a, b, c),
      B (a2, a, ~b, ~c),
      C (a3, ~a, o1);
  or D (o1, b, c),
      E (f, a1, a2, a3);
endmodule
```
What Do We Want Here...?

Goal
- To specify a combination ckt, inputs->outputs...
- ... in a form of Verilog that synthesis tools will correctly read
- ... and then use to make the right logic

And...
- We know the function we want, and can specify in C-like form...
- ... but we don’t now the exact gates; we want the tool to do this.

Combinational Logic

A →
B →
C →
F →
Behavioral Modeling

- Procedural statements are used
  - Statements using “always” Verilog construct
  - 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)

- Current technology
  - You can do combinational (and later, sequential) design
  - Sizable designs can take hours … days … to run
  - Companies pay $50K - 80K per copy for such software
    - This ain’t shrink-wrap software!
  - The software we’ll use is more like $10-15K
## 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>initial begin ... end</td>
<td>Starts when simulation starts</td>
<td>Execute once and stop</td>
<td>Not used in synthesis</td>
</tr>
<tr>
<td>always</td>
<td>always begin ... end</td>
<td>Continually loop—while (power on) do statements;</td>
<td>Used in synthesis</td>
<td></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

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

Can’t do — why?
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.
    
    $8'b00x0zx10$

```verilog
if (select == 1) 
  f = in1;
else 
  f = in0;
```

```verilog
case (selector)
  2'b00: a = b + c;
  2'b01: q = r + s;
  2'bx1: r = 5;
  default: r = 0;
endcase
```

assume f, a, q, and r are registers for this slide
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 — see the book

```
reg [3:0] testOutput, i;
...
for (i = 0; i <= 15; i = i + 1) begin
    testOutput = i;
    #20;
end
```

```
reg [3:0] testOutput, i;
...
i = 0;
while (i <= 15) begin
    testOutput = i;
    #20 i = i + 1;
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.
Bit Selects and Part Selects

This expression extracts bits or ranges of bits or a wire or register.

The individual bits of register \(i\) are made available on the ports. These are later connected to individual input wires in module design.

Alternate:

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

```verilog
module top;
wire [3:0] w;
testgen t (w);
design d (w[0], w[1], w[2], w[3]);
end
```

```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
```

```verilog
module top;
wire w0, w1, w2, w3;
testgen t (w0, w1, w2, w3);
design d (w0, w1, w2, w3);
end
```

```verilog
module design (a, b, c, d);
input a, b, c, d;

mumble, mumble, blah, blah;
end
```
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 that 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.

- Why are these concurrent?

  - Because the event being waited for can only occur as a result of the
  concurrent execution of some other always/initial block or gate.

  - They’re happening concurrently
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 behavioral constructs

- Logic for a simple MUX is specified procedurally here
- This example is synthesizable

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

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

Read this as follows:
Wait for any change on a, b, or c, then execute the begin-end block containing the if. Then wait for another change.

This “if” functionally describes the MUX
Is it really correct?

Problem?

- Where’s the register?
  The synthesis tool *figures out that this is a combinational circuit*. Therefore, it doesn’t need a register.

  The register is there as an “artifact” of the descriptions — things on the left-hand side have to be registers.

- How does it figure out that this is combinational?
  - The output is only a function of the inputs (and not of previous values)
  - Anytime an input changes, the output is re-evaluated

- Think about the module as being a black box …
  - Could you tell that there is a register in there?

```verilog
module mux (f, sel, b, c);
  output f;
  input sel, b, c;
  reg f;
  always @ (sel or b or c)
    if (sel == 1)
      f = b;
    else
      f = c;
endmodule
```
Using procedural statements in Verilog

- Logic is specified in "always" statements ("Initial" statements are not allowed).
- Each "always" statement turns into Boolean functions.

```verilog
module blah (f, a, b, c);
    output f;
    input a, b, c;
    reg f;

    always @ (a or b or c)
        begin
            stuff...
            stuff...
            stuff...
        end
endmodule
```

You have to declare the combinational outputs like this, for synthesis. i.e., tool needs to think you are putting these computed outputs someplace.

You have to list all the block’s inputs here in the “sensitivity list”.

Actually do logic in here. There are a bunch of subtle rules to ensure that synthesis won’t mess this up... We’ll see how...
How? ... A Few Definitions

There are some restrictions on specification

- **Input set** of an “always” statement — the set of all variables that are used on the right-hand side of procedural assignments or in conditionals. i.e. anything “sourced”.

- **Sensitivity list** of an “always” statement — the set of all names that appear in the event (“@”) list.

```verilog
module mux (f, sel, b, c);
  output f;
  input sel, b, c;
  reg f;
  always @ (sel or b or c)
    if (sel == 1)
      f = b;
    else
      f = c;
endmodule
```

The elements in these lists are:
More Definitions...

A control path of an “always” statement — a sequence of operations performed when executing the always statement

Combinational output of an “always” statement — a variable (or variables) assigned to in every control path

```verilog
module mux (f, sel, b, c);
  output f;
  input sel, b, c;
  reg f;
  always @ (sel or b or c)
    if (sel == 1)
      f = b;
    else
      f = c;
endmodule
```
The Basic Rules

- The rules for specifying combinational logic using procedural statements
  - Every element of the input set must be in the sensitivity list
  - The combinational output must be assigned in every control path

So, we’re saying that if any input changes, then the output is re-evaluated. — That’s the definition of combinational logic.

Walking this narrow line allows you to specify and synthesize combinational logic.

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

  always @ (sel or b or c)
    if (sel == 1)
      f = b;
    else
      f = c;
endmodule
```
What If You Mess Up?

If you don’t follow the rules...? … you’re dead meat

- Verilog assumes you are trying to do something clever with the timing
- It’s legal, but it won’t be combinational
- The rules for what it does make sense -- but not yet for us.

```verilog
module blah (f, g, a, b, c);
    output f, g;
    input a, b, c;
    reg f, g;

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

This says: as long as a==1, then f follows b. (i.e. when b changes, so does f.) But, when a==0, f remembers the old value of b.

Combinational circuits don’t remember anything!

What’s wrong?

f doesn’t appear in every control path in the always block (neither does g).
Your Verilog for combination stuff will look like this:

```verilog
module blah (<output names>, <input names>);
    output <output names>;
    input <input names>;
    reg <output names>;

    always @( <names of all input vars>)
    begin
        < LHS = RHS assignments>
        < if ... else statements>
        < case statements>
    end
endmodule
```

Yes...it’s a pretty restricted subset of the language...
A Difficulty

Assigning in every control path

- If the function is complex, you don’t know if you assigned to the outputs in every control path.
- So, set all outputs to some known value (zero here) and write the code to set them to other values as needed.
- Synthesis tools will figure it out.

```verilog
always @(coke or cola) begin
  if (coke)
    blah1 = 1;
  else if (cola > 2'b01)
    blah2 = coke;
  else if ( ... ...
end
```

```verilog
always @(coke or cola) begin
  blah1 = 0;
  blah2 = 0;
  if (coke)
    blah1 = 1;
  else if (cola > 2'b01)
    blah2 = coke;
  else if ( ... ...
end
```
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'bx
then \{a, b, c\} = 12'b01111x0001zx

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

Check the rules ...
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'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
```

check the rules...

```
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
```

Important: every control path is specified
Don’t Cares in Synthesis

### Rules

- You can’t say “if (a == 1’bx)…” — this has meaning in simulation, but not in synthesis.
- However, an unknown x on the right-hand side will be interpreted as a don’t care.

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

The inverse function was implemented; x’s taken as ones.
Alternatively…

These aren’t quite equivalent to the previous slide…why?

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Module Fred1**

```verilog
define Fred1 (f, a, b, c);
    output f;
    input a, b, c;
    reg f;

    always @(a or b or c)
        f = ~(a & b & ~c);
endmodule
```

**Module Fred2**

```verilog
define Fred2 (f, a, b, c);
    output f;
    input a, b, c;
    reg f;

    always @(a or b or c)
        f = ~a | c | ~b;
endmodule
```

**Module Fred3**

```verilog
define Fred3 (f, a, b, c);
    output f;
    input a, b, c;
    reg f;

    always @(a or b or c)
        begin
            if (c ==0)
                f = a~&b;
            else f = 1’b1;
        end
endmodule
```
Two inputs, Three outputs

```verilog
reg [1:0] newJ;
reg out;
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
```

Works like the C conditional operator.

`(expr) ? a : b;
If the expr is true, then the resulting value is a, else it’s b.
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 D input
- What order will the gates and behavioral model execute in.

```verilog
always @ (D)
begin
  yadda yadda
end

always @ (F)
begin
  yadda yadda
  b = D;
end
```
Could we have described the module as shown here?

- Note the delays. There is a different delay from the b input than from the c input.
- Yes, you could write this

But,

- Synthesis tools will ignore the time delays.
- Generally, they try to minimize the propagation from any combinational input to any combinational output in the system.

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

always @ (sel or b or c)
  if (sel == 1)
    #5 f = b;
  else
    #88 f = c;
endmodule```
Here's an always block for a combinational function.

- What Boolean functions can it model?
- Can I have more than one of these always blocks in a module? Yes
- Can two separate always calculate function f? No

```vhdl
module xyzzy (ports);
  ...
  always @(b1 or b2 or b3)
  begin
    f = yadda;
  end
  always @(r1 or r2 or r3)
  begin
    f = yadda yadda;
  end
  ...
endmodule
```

```vhdl
module xyzzy (ports);
  ...
  always @(b1 or b2 or b3)
  begin
    q = b1 ... b2 ... b3
    r = b2 ... b3
  end
  always @(r1 or r2 or r3)
  begin
    s = yadda yadda yadda
  end
endmodule
```
Module partitioning can affect logic optimizations

- Here are two modules
- The output of blob1 is connected to blob2
- The synthesis tool will optimize them separately
  - No common prime implicants, etc, will be shared or optimized between the two modules.

```vhd
module blob1(inputs, outputs1)
  ...
  always @(inputs)
  outputs1 = blah & inputs;
endmodule
```

```vhd
module blob2(inputs, outputs2)
  ...
  always @(inputs)
  outputs2 = blah & inputs;
endmodule
```

- Alternate
  - Put everything in one module
  - Now there's a possibility for optimization between functions

```vhd
module blob1_2(inputs, outputs)
  always @(inputs)
  outputs1 = blah & inputs;
  always @(outputs1)
  outputs = blah & outputs1;
endmodule
```
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
Discrete Event Simulation

- Quick example
  - Gate A changes its output. This causes gates B and C to execute
    - But as we’ll see, A doesn’t call B and C (as in a function call)
    - Rather, they execute because they’re connected

- Observation
  - The elements in the diagram don’t need to be logic gates
  - SimCity is a discrete event simulator, Verilog too

- Discrete Event Simulation
  - Events — changes in state — occur at discrete times. These cause other events to occur.
  - Time advances in discrete (not continuous) steps
Non-discrete Event Simulation

- Continuous systems — all elements and state are updated at every simulation time
- Could you do logic circuits that way too?
  - ...
- e.g. analog circuits, numerical integration …
  - differential equations to solve
Discrete Event Simulation

- Basic models — things not found in C
  - gate level — built-in models for AND, OR, ...
    - When an input to one of these changes, the model executes to see if its output should change
  - behavioral level — sort-of C-like programs but with a few extra operators
    - Executes until it blocks for one of three reasons — #delay, wait(level), or @(event) — when the reason for blocking is resolved, it continues executing
    - Does C have any notion of these?

- Gate and behavioral models can advance time
How does it keep track of time?

Explicitly

- Events are stored in an event list (actually a 2-D list) ordered by time
- Events *execute* at a time and possibly *schedule* their output to change at a later time (a new event)
- When no more events for the current time, move to the next
- Events within a time are executed in arbitrary order

Let's say A changes to 0 here. B and C have delay 2.
Two pieces of a simulation

- **The model** — an executable specification including timing, interconnect, and input vectors
  - Written in a language like Verilog or VHDL
  - What’s a VHDL?

- **The simulation scheduler** —
  - keeps track of when events occur,
  - communicates events to appropriate parts of the model,
  - executes the model of those parts, and
  - as a result, possibly schedules more events for a future time.

  - it maintains “simulated time” and the event list.
Verilog Levels of Abstraction

- **Gate modeling**
  - the system is represented in terms of primitive gates and their interconections
    - NANDs, NORs, ...

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

---

**Gate-level model**

always
@posedge clock
Q = #5 D

**behavioral model**

© Don Thomas, 1998, 52
Mixing Levels

Generally there is a mix of levels in a model

- e.g. part of the system is at the gate level and another part is at the behavioral level.
- Why?
  - Early in design process you might not have fully-detailed models — you don’t actually know all the gate implementations of the multipliers, adders, register files
  - You might want to think of the design at a conceptual level before doing all the work to obtain the gate implementations
  - There might be a family of implementations planned
- Levels — switch, gate, functional block (e.g. ALUs), register-transfer, behavioral
  - for now, we’ll deal with gate and behavioral models
- These are all modeled as discrete systems — no continuous modeling of analog behavior
A Gate Level Model

- A Verilog description of an SR latch
A Gate Level Model

A Verilog description of an SR latch

A module is defined

module nandLatch (q, qBar, set, reset);

output q, qBar;
input set, reset;

g1 (q, qBar, set),
g2 (qBar, q, reset);

endmodule

primitive gates with names and interconnections

name of the module
the module has ports

ports are typed
type and delay of gates

© Don Thomas, 1998, 55
A Gate Level Model

Things to note:

- It doesn’t appear “executable” — no for loops, if-then-else, etc.
  - it’s not in a programming sense, rather it describes the interconnection of elements
- A new module made up of other modules has been defined
  - software engineering aspect — we can hide detail

```plaintext
module nandLatch (q, qBar, set, reset);
  output q, qBar;
  input set, reset;

  nand #2
    g1 (q, qBar, set),
    g2 (qBar, q, reset);
endmodule
```
But, there is an execution model

Gate-level timing model

- **Timing model** — how time is advanced, what triggers new processing in the model
- Here — when any of the inputs of a primitive gate change, the output is re-evaluated. If there is a new result, it is passed on to other gates on its fanout.

```verilog
module nandLatch (q, qBar, set, reset);
  output q, qBar;
  input set, reset;
  nand #2
    g1 (q, qBar, set),
    g2 (qBar, q, reset);
endmodule
```
Why not describe a module’s function and delay using a language like C?

- Sound like fun, here goes

```verilog
module d_type_FF (q, clock, data);
    output q;
    reg q;
    input clock, data;
    always @(negedge clock) q = #10 data;
endmodule
```

@ … — wait for a negative edge on clock, evaluate “data” now and wait 10 time units. Then assign q to that value and wait for the next negedge

reg — declares a one-bit register. Can be thought of as being similar to a variable in programming. BTW, each instantiation of this module will have a separate register q.

always — “while TRUE” Continuously do the following statement.
Comparison

These two models are interchangable — either could have been instantiated into a register
- ports in same order
- same delay from clock to q
- one is abstract, clear
- one is specific
- there are subtle differences

```verilog
module d_type_FF (q, clock, data);
  output q;
  reg q;
  input clock, data;

  always @(negedge clock) q = #10 data;
endmodule
```

```verilog
module d_type_FF (q, clock, data);
  input clock, data;
  output q;
  wire q, qBar, r, s, r1, s1;
  nor #10 a (q, qBar, r);
  nor b (qBar, q, s),
  c (s, r, clock, s1),
  d (s1, s, data),
  e (r, r1, clock),
  f (r1, s1, r);
endmodule
```

Behavioral

Structural
At first look, it is a lot like C

Most of the operators are the same as C
  - ^ is XOR, etc.
  - makes it easy to read

But there are major differences (quick list, we’ll get to these)
  - statements like #delay, @event, wait(level)
    - the language is concurrent — can specify many things that can happen at the same time.
  - four-valued logic (1, 0, x, z) and the operators to go with them
  - arbitrary bit width specification
  - there are a couple of procedural assignments (=, <=) with subtle differences
  - a different 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 "no" here too. — not always
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
```

This would execute

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

This wouldn’t
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.

```verilog
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?

Oops — The order of execution can matter!

*film at 11*

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

module sreg (...);
    ...
    dff a (q0, shiftin, clock),
        b (q1, q0, clock),
        c (shiftout, q1, clock);
endmodule
```
Some more gate level examples

- An adder

```vhdl
module adder (carryOut, sum, alnput, blnput, carryIn);
    output carryOut, sum;
    input alnput, blnput, carryIn;
    xor (sum, alnput, blnput, carryIn);
    or (carryOut, ab, bc, ac);
    and (ab, alnput, blnput),
        (bc, blnput, carryIn),
        (ac, alnput, carryIn);
endmodule
```

- List of gate instances of same function (AND)
- Implicit wire declarations

- No instance names or delays
Adder with delays

An adder with delays

```verilog
module adder (carryOut, sum, alinput, blinput, carryIn);
  output carryOut, sum;
  input alinput, blinput, carryIn;
  xor #(3, 5) (sum, alinput, blinput, carryIn);
  or #2 (carryOut, ab, bc, ac);
  and #(3, 2) (ab, alinput, blinput),
            (bc, blinput, carryIn),
            (ac, alinput, carryIn);
endmodule
```

- Each AND gate instance has the same delay.
- Alternate timing.
Adder, continuous assign

Using “continuous assignment”

- **Continuous assignment** allows you to specify combinational logic in equation form
- Anytime an input (value on the right-hand side) changes, the simulator re-evaluates the output
- No gate structure is implied — logic synthesis can design it.
  - the description is a little more abstract
- A behavioral function may be called — details later

```verbatim
module adder (carryOut, sum, aInput, bInput, carryIn);
  output carryOut, sum;
  input aInput, bInput, carryIn;

  assign sum = aInput ^ bInput ^ carryIn,
               carryOut = (aInput & bInput) | (bInput & carryIn) |
                           (aInput & carryIn);
endmodule
```
Continuous assignment assigns continuously

- delays can be specified (same format as for gates) on whole equation
- no instances names — nothing is being instantiated.
- given the same delays in this and the gate-level model of an adder, there is no functional difference between the models
  - FYI, the gate-level model gives names to gate instances, allowing back annotation of times.

module adder (carryOut, sum, aInput, bInput, carryIn);
output carryOut, sum;
input aInput, bInput, carryIn;
assign #(3, 5) sum = aInput ^ bInput ^ carryIn;
assign #(4, 8) carryOut = (aInput & bInput) | (bInput & carryIn) | (aInput & carryIn);
endmodule
Using continuous assign vs gate instantiations

- drives wires
- abstract
- random logic
- used for specifying unknowns
- detailed, specific
- loads registers
- typically no notion of wire delays
- multibit datapath elements

which goes with which?
Gate level timing model

Execution model

- *execution model* — how time advances and new values are created
- a fundamental concept in any language

Gate level timing model

- applies to both primitive instantiations and continuous assigns

Definition —

- when an *input* changes, the simulator will evaluate the primitive or continuous assign statement, calculating a new output
- if the *output* value is different, it is propagated to other primitive and assign inputs
- nothing said yet about behavior.
Gate level timing model

What’s an input?
- an input to a gate primitive
- anything on the right-hand side of the “=” in a continuous assign

What’s an output?
- the output of a gate primitive
- anything on the left-hand side of the “=” in a continuous assign

Outputs on this “side” of the language are all …
- … wires
- no registers are latched/loaded, no need to know about a clock event
- i.e. the left-hand sides are all wires

Contrast
- The left-hand sides on the behavioral “side” of the language are all registers
Event-Driven Simulation

- How does the simulator execute a gate-level model
- Event-driven simulation
  - Event — a value-change occurs at a given time
  - The event-driven simulator only executes models when events occur
    - (some simulators execute every model every time unit)
Two types of events

- **Evaluation events** — evaluate, or execute, a gate model or continuous assign.
  - produce update events
  - i.e. if the output changes, schedule an update event
- **Update events** — propagate new values along a fanout.
  - produce evaluation events
  - for each element on the fanout, schedule an evaluation event

We’ll treat these as separate types of events

- gate level simulators generally combine them for efficiency
- i.e. when an output is updated, instead of scheduling an evaluation, just do the evaluation and schedule any updates resulting from it.
- We’ll keep them separate for now — it will help in the later discussion of behavioral models
Event-Driven Simulation

while something in time-ordered event list {
    advance simulation time to top event's time
    retrieve all events for this time

    For each event in *arbitrary* order

    If it's an update event
        Update the value specified.
        Follow fanout and evaluate gate models.
        Schedule any new updates from gates.
        Schedule eval events for behavioral models

    else // it's an evaluation event
        evaluate the model
        schedule resulting update events

}
Event-Driven Simulation

the event list

Update A=1 at 25
init values as shown

Update A=1 at 25
init values as shown

Update A=1 at 25
init values as shown

A=0
1
1
A=0
B=1
C=0
D=1

A=1
1
1
A=1
B=0
C=0
D=1

B=1
1
1
B=1
G=3
C=0
D=1

C=0
1
1
C=0
G=3
G=5
D=1

D=1
1
1
D=1
G=3
G=5

Event-driven simulation

Update $A=1$ at 25
init values as shown

Update $A=1$ at 25
init values as shown

Update $A=1$ at 25
init values as shown

1
$g1 \#2$
$B=0$

1
$g2 \#3$
$C=1$

0
$g3 \#5$
$D=1$

final
What if an update event is already scheduled for an output?

- If the value being scheduled is different, the currently scheduled value is removed from the event list; the new is not scheduled.
- Thus, any input pulse shorter than the propagation delay will not be seen (inertial delay).

\[a \land b \land c\]

Update scheduled

Update removed, final value

\[\text{nand #5 (c, a, b);}\]

What happens in four-valued logic?
Scheduling and event list management

- Can think of the event list as a 2-D linked list
  - One dimension links all the events for a given time
  - The second dimension links these lists in ascending order

- Problem
  - inefficient — most events are near in time to the current one, thus lots of linked list bashing
Scheduling and event list management

Hack of the rich and famous — “Timing wheel”

- M nearest time slots stored in an array — M is a power of two
- Access a list by (time mod M) — a table lookup
- Essentially turned first linked list access into an array access saving time
- Further out times are kept in linked list. As time is advanced, further out times are brought into wheel
Asides

- Can a gate model be executed several times in a time step?

- Does the order of execution of the gates in a combinational circuit matter?
Summary on gate evaluation

- **Timing model**
  - *timing-execution model*
    - how time is advanced and new values created
  - Any gate input or assign righthand-side change causes the model to be evaluated during the time step
    - this is not the case for behavioral models — they have a different timing model
  - Fanout list is static — design never changes

- **Gate level modeling**
  - detailed timing

- **Continuous assignment**
  - abstract

- **What if you don’t like these models?**
  - e.g., inertial delays?
  - use behavioral models
Review Stuff

- **Update Events**
  - A new value appears at some simulated time

- **Evaluation Events**
  - A model is executed (evaluated) at some simulated time

- **Event List**
  - A time-ordered list of events

- **Simulation scheduler**
  - Software program that manages the event list by scheduling update and evaluation events, tracing fanouts to propagate values, and manages simulated time
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
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);
  ...
endmodule

module no (in, ...);
  input in;
  ...
  while (in != 1);
  ...
endmodule
```
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 ck)
  q <= d;

always @(hello or goodbye)
  a = b;

always @(hello)
  a = b;

always begin
  yadda = yadda;
  @(posedge hello or negedge goodbye)
  a = b;
  ...
end
```
Sensitivity Lists

In the gate level timing model...

- model execution was sensitive to any change on any of the inputs at any time.
- **sensitivity list** — a list of inputs that a model is sensitive to
  - a change on any of them will cause execution of the model
- In the gate level timing model, the lists don’t change.
- Ditto with continuous assign

In procedural models ...

- the sensitivity list changes as a function of time and execution

```verilog
module d_type_FF (q, clock, data);
    input clock, data;
    output q;

    nor #10
        a (q, qBar, r);
    nor
        b (qBar, q, s),
        c (s, r, clock, s1),
        d (s1, s, data),
        e (r, r1, clock),
        f (r1, s1, r);

    endmodule
```

Structural
**Fanout Lists**

- Outputs of things are connected to inputs of other things
  - No surprise
  - The simulator maintains a list of inputs driven by each “output”

- Why?
  - When the output changes, it’s easy to figure out what other models need (to be) evaluated

- What’s an “output” in the above sense?

- Because of procedural models ...
  - Fanout lists change

- Fanout lists <-> Sensitivity lists
Behavioral Timing Model

What is the behavioral model sensitive to?
- The behavioral statements execute in sequence (one then the next)
- Therefore, what a behavioral model is sensitive to is context specific
  - i.e. it is only sensitive to what it is currently waiting for
    - time, edge, level — (#, @, wait)
- The model is not sensitive to a change on y, or w.

```verbatim
always begin
  @ (negedge clock1)
    q = y;
  @ (negedge clock2)
    q = w;
  @ (posedge clock1)
    /*nothing*/;
  @ (posedge clock2)
    q = 3;
end
```

- Here, it is only sensitive to clock1
- Here, it is only sensitive to clock2. A posedge on clock1 will have no effect when waiting here.
- It is never sensitive to changes on y or w
Scheduling #, @, and Wait

How are #, @, and wait tied into the event list?

- **# delay**
  - schedule the resumption of the process — put it in the event queue delay units into the future. Essentially an evaluation event scheduled in the future.

- **@ change**
  - when suspended for an @v, the behavioral model is put on the fanout list of the variable v. i.e., the behavioral model is now sensitive to v.
  - When an update event for v occurs, (e.g. posedge), then the behavioral model is scheduled to **resume** at the current time — an evaluation event.

- **Wait (exp)**
  - if exp is TRUE, don’t stop
  - if exp is FALSE, then the behavioral model is put on the fanout list(s) of the variable(s) in exp. (it’s now sensitive to the variable(s))
  - When there is an update event for any of the variables in exp, exp is evaluated. If exp is TRUE, **resume** executing in the current time (schedule an eval event), else go back to sleep.
Non-blocking assignments (\(\leq\))

- Two important aspects to these
  - an intra-assignment time delay doesn’t stop them (they’re non-blocking)
  - they implement a concurrent assignment
- Example — intra-assignment time delay
  - non-blocking assignments use “\(\leq\)”
    \[
    a \leq \#10 \ b + \ c;
    \]
- What happens?
  - \(b + c\) is calculated
  - an update event for \(a\) is scheduled \(\#10\) in future
  - execution of the always continues in the current time
    - the execution of the always is not blocked by the delay
  - there is also a subtle difference in how \(a\) is updated …
    - we’ll get to it, but first, an example
What’s the difference?

module procAnd1 (a1, b, c);
    input  b, c;
    output a1;
    always @(b or c)
        a1 = #5 b & c;
endmodule

module procAnd2 (a2, b, c);
    input  b, c;
    output a2;
    always @(b or c)
        a2 <= #5 b & c;
endmodule

Which is similar to an AND primitive?

assume c = 1
**Non-Blocking Concurrent Assignment**

### Concurrent Assignment — primary use of <=
- The assignment is “guarded” by an edge
- All assignments guarded by the edge happen concurrently
  - All right-hand sides are evaluated before any left-hand sides are updated
  - Like this

```verilog
module fsm (Q1, Q0, in, clock);
  output Q1, Q0;
  input clock, in;
  reg Q1, Q0;

  always @(posedge clock) begin
    Q1 <= in & Q0;
    Q0 <= in | Q1;
  end
endmodule
```
module fsm (Q1, Q0, in, clock);
    output Q1, Q0;
    input clock, in;
    reg Q1, Q0;

    always @(posedge clock) begin
        Q1 <= in & Q0;
        Q0 <= in | Q1;
    end
endmodule

Values after the clock edge (t+)
— calculated in response to the clock edge, using values at the clock edge

Values at the clock edge. (At t−)
Alternates — not all equivalent

module fsm (Q1, Q0, in, clock);
  output Q1, Q0;
  input clock, in;
  reg Q1, Q0;

  always @(posedge clock) begin
    Q1 <= in & Q0;
    Q0 <= in | Q1;
  end
endmodule

module fsm (Q1, Q0, in, clock);
  output Q1, Q0;
  input clock, in;
  reg Q1, Q0;

  always @(posedge clock) begin
    Q1 = in & Q0;
    Q0 = in | Q1;
  end
endmodule

module fsm (Q1, Q0, in, clock);
  output Q1, Q0;
  input clock, in;
  reg Q1, Q0;

  always @(posedge clock) begin
    Q0 <= in | Q1;
    Q1 <= in & Q0;
  end
endmodule

A very different animal?
The same?
module fsm1 (Q1, Q0, in, clock);
  output Q1;
  input clock, in, Q0;
  reg Q1;

  always @(posedge clock) begin
    Q1 <= in & Q0;
  end
endmodule

module fsm0 (Q1, Q0, in, clock);
  output Q0;
  input clock, in, Q1;
  reg Q0;

  always @(posedge clock) begin
    Q0 <= in | Q1;
  end
endmodule

Will these work?

module fsm1 (Q1, Q0, in, clock);
  output Q1;
  input clock, in, Q0;
  reg Q1;

  always @(posedge clock) begin
    Q1 = in & Q0;
  end
endmodule

module fsm0 (Q1, Q0, in, clock);
  output Q0;
  input clock, in, Q1;
  reg Q0;

  always @(posedge clock) begin
    Q0 = in | Q1;
  end
endmodule

These?
The Important Aspect …

- Non-Blocking Concurrent transfers
  - Across the whole design,
    - all right-hand sides are evaluated
    - before any left-hand sides are updated.
  - Thus, the order of r-hs’s evaluated and l-hs’s updated can be arbitrary (but separate)
- This allows us to …
  - handle concurrent specification in major systems
  - reduce the complexity of our descriptions
  - attach lots of actions to one event — the clock
A State Change

- Find all of your “state” variables
  - Not just FSM state, but registers in a datapath too
  - They’re probably all keyed to an edge of a clock
  - Use <= to assign to them at the edge
  - You’re guaranteed they’ll all be sampled before any of them are updated.

- A check: in many cases, the only #delay operator you need is in the clock (for functional specification)
**Event List: We told a fib**

- This is what we told you before:

- In a concurrent language, there are some very dirty issues regarding the “arbitrary order” of execution.

- In software, such issues are handled by synchronization primitives
  - Some of you have probably seen semaphores in the OS or real-time (embedded systems) course
  - They only allow other concurrent parts of a system to see full state changes, not partial. State changes appear “atomic”
  - These provide a very clean way to enforce order (actually, mutual exclusion) within “zero time”
Differences in the Event List Scheduling

- Previous picture of doubly linked event list

- More detailed view
  - Three lists per time
  - Regular events, gate outputs, continuous assign outputs, updates of blocking procedural assignments
  - Non-blocking procedural updates
  - Monitor events

© Don Thomas, 1998
What gets scheduled when/where

Now

- While there are *regular* events:
  - “retrieve all regular events for current time and execute in arb. order”
  - Note: These may produce more regular events for current time
- Retrieve all *non-blocking* events for the current time and execute
  - these may produce more regular events for current time, if so
- When no more events, do *monitor* events. No new events produced
A picture of the event list

What happens?

\[ a \leq b + c \]

\$\text{monitor} (\ldots q)\$
module fsm (Q1, Q0, in, clock);
  output Q1, Q0;
  input clock, in;
  reg Q1, Q0;

  always @(posedge clock) begin
    Q1 <= in & Q0;
    Q0 <= in | Q1;
  end
endmodule

always #10 clock = ~clock;

clock becomes 0

time 20

time 30

time 40
Follow the Execution

module dff (Q, D, clock);
  output Q;
  input clock, D;
  reg Q;
  
  always @(posedge clock)
    Q <= D;
  
  always #10 clock = ~clock;
endmodule
previous values:
  A = 1
  B = 0
  S = 0
new values at time 10:
  A = 0
  B = 1
  S = 1

U: A = 0
B = 1
S = 1

10
More Scheduling

previous values: x

and (c, a, b);
always begin
a = 0;
#0 q = 1;
...
#10 ...
initial
b = 1;
Other strange things you can do

- A 4-stage pipelined multiplier
  - every clock edge, the a and b inputs are read and their product is scheduled to appear three clock periods later

```verilog
module pipeMult (product, a, b, ck);
  input ck;
  input [9:0] a, b;
  output [19:0] product;
  reg [19:0] product;

  always @(posedge ck)
    product <= repeat (3) @(posedge ck) a * b;
endmodule
```
Some ugly ramifications

You need to be careful when mixing blocking and non-blocking assignments

- **blocking** — you can read it like regular C language assignments. The value of the variable on the left-hand side can be used in the next statement on the right-hand side.

- **non-blocking** — the assignment is scheduled to appear at a later time. The value on the left-hand side is not available in the next statement.

The Verilog Police say: “careful on how you mix these!”

General rule: for “state” use “<=”. For intermediate values and combinational elements, use “=”}

```plaintext
a = 3
b = 4
a <= 3 + 4
c = a
```

What value is assigned to c? who cares
while (there are events in the event list) {
    if (there are no events for the current time
        advance currentTime to the next event time
    if (there are no regular events for the current time)
        if (there are non-blocking assignment update events)
            turn these into regular events for the current time
        else
            if (there are any monitor events)
                turn these into regular events for the current time
    Unschedule (remove) all the regular events scheduled for currentTime
    For each of these events, in arbitrary order {
        if (this is an update event) {
            Update the value specified
            Evaluate gates on the fanout of this value and Schedule update
            events for gate outputs that change
            Schedule evaluation events for behaviors waiting for this value
        }
        else { // it’s an evaluation event
            Evaluate the model
            Schedule any update events resulting from the evaluation
        }
    }
}
Gate-Level Modeling

- 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:
      ```
      and #delay name (out, in1, in2, in3, ...)  
      ```
      - multi-level logic used in some models to represent:
        - values, edges, unknowns, high impedances, ...
Logic Values

- Verilog Logic 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. Generally treated as an x on an input.

- Off-the-wall, but important, values (a partial list)
  - rising edge — posedge
    - 0->x; x->1; 0->1
  - falling edge — negedge
    - 1->x; x->0; 1->0
  - switch-transistor values
    - strong 1; weak 1; ...

- Logic with multi-level logic values
  - note: z treated as an x on input
  - some languages allow you to define a function based on multi-level logic values (Verilog does)

<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>
Delay models for gates: views and definitions

- Basic view: the function and delay are separate
  - The function is handled by model execution, the delay by the simulator scheduler
**Kinds of delays**

**Definitions**

- **Zero delay** models — functional testing
  - there’s no delay, not cool for circuits with feedback!
- **Unit delay** models — all gates have delay 1. OK for feedback
- **Transport** delay — input to output delay
- **Inertial** delay — how long must an input spike be to be seen?
  - in Verilog, inertial == transport

![Diagram of transport delay](image)

Inertial delay — too small, no output change
Delay Models

- Other factors
  - Delay can be a function of output transition
  - Need a number for each of the arrowheads

- Verilog example

```
primitive gate instantiation
  rising delay 0->1 0->x x->1 z->1
  falling delay 1->0 1->x x->0 z->0

not # (3, 5, 7) (nbar, n);
```

Delay to z (tristate gates only)
Delay Models

Unknown Delays — different simulators do different things

- $d = \text{randomize (min, max, distribution)}$
  - delay is determined per gate at simulator startup time, same time used for gate throughout
  - this might model TTL chips, but not gates on an IC
    - Why?

- $d = (\text{min, typical, max})$
  - delay to use is determined by simulator command at simulator startup time (i.e. one is selected)
  - for Verilog, each of the three timing values can be replaced by a triple (min:typ:max)

not # (2:3:4, 4:5:6, 7:8:9) (nbar, n)
Overridden Delays

Delays Overridden

- Use “actual” delays to override specified model delays
- Most importantly, delay due to loading and path lengths is made more accurate
  - generally, this adds to the wire delay accuracy
Delays on Wires

How do you drive wires?

- gate outputs can drive wires
  - gate outputs implicitly define wires
- wires can also be defined — with or without delay

```
wire <size> <delay> name;
wire  #5       LM;
and   #6  a (LM, L, M);
not   b (MA, MB, LM);
```

- The delay on a wire is added to any delay in the gate(s) that drive the wire

Gate b sees an input change 11 time units after a change on L or M
Model Evaluation

- **Gate evaluation**
  - the design is made up of primitive gates and behaviors
  - we’re only considering primitive gates now

- **Approach #1**
  - Performing (A nand B nand …) is slow
    - especially in multi-valued logic
  - Use table lookup — takes up memory, but it’s fast
    - Encode 0 as 00, 1 as 01, x as 10, z as 11

```
Truth Table for Nand
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

{A,B} → Truth Table → result
Model Evaluation

Oh that was fun, let’s do more of it

- Variation on table lookup — “Zoom table”
  - the table includes all primitive functions as well as variables

- Essentially this is the “programming pearl” that says:
  - If it takes too much time to calculate something, precalculate it, store the results in a table, and look up the answer

Truth Table
for Nand and And

```
<table>
<thead>
<tr>
<th>func</th>
<th>A</th>
<th>B</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Zoom
Truth Table

\{func,A,B\} → result
Model Evaluation

Approach #2 — Input counting method

- input width independent (as compared to Zoom tables)
- represents functions by controlling and inversion values
  - output is $c \oplus i$
- Evaluation function:

```plaintext
x_val = FALSE
for every input v of G {
    if (v == c) return (c \oplus i)
    if (v == x) x_val = TRUE
}
if (x_val) return x
return (c' \oplus i)
```

- requires scanning of the inputs

If any input is controlling, you know the output
Simulation: Model Evaluation

Approach #3: Input counting

- An update event keeps count of various features
  - when 1 -> 0 on AND gate, increment c_count
    - (the number of controlling inputs)
  - when 0 -> x on AND gate, decrement c_count, increment x_count

- an evaluation event becomes

  ```
  if (c_count > 0) return c \oplus i
  if (x_count > 0) return x
  return c’ \oplus i
  ```

- Can you make this work with XORs?
Behavioral Models

- Interpreted
  - Compile to an intermediate representation
  - To execute, interpret it — slow

- Code Generation
  - Compile behavioral Verilog directly to assembly code — treat it as a programming language
  - Long compile times, but fast execution
    - Still slower than regular C — why?
  - Not limited to behavioral models

- Who said computer engineers don’t need to know how a compiler works!
Tying behavior and gate models together

Real designs mix behavior and gate models

```vhdl
module beh (i1, i2, o1, clk);
  input [] i1, i2, clk;
  output [] o1;
  reg [] a, b, o1;
  always begin
    @ (posedge clk);
    a = b + i1;
    o1 = a + i2;
  endmodule

module str (in1, out1, out2);
  input [] in1;
  output [] out1, out2;
  assign #10 out1 = in1 | a;
  nand #(2, 5) (out2, in1, b);
  nand #(6, 3) (xxx, in1, b);
  ...
endmodule

module putTogether ();
  wire [] w1, w2, w3, w4;
  beh inst1 (w1, w2, w3, w4);
  str duh (w3, w2, w1);
endmodule
```
Tying behavior and gate models together

An alternate version

- modules may contain mixture of behavior and gate models

```verilog
module behstr (clk);
    reg [ ] a, b, o1in1;
    input clk;

    wire [ ] #10 out1i2 = o1in1 | a;
    nand #(2, 5) (out2in1[0], o1in1[0], b[0]);
    nand #(2, 5) (out2in1[1], o1in1[1], b[1]);

    always begin
        @ (posedge clk);
        a = b + out2in1;
        o1in1 = a + out1i2;
    ...
```

note that the assign turned into a wire declaration with an assign.

changes will be propagated to a and o1in1 after the behavioral model stops again at the “@”
Names of things

Thus far, we’ve seen names of...
- registers, variables, inputs, outputs, instances, integers
- Their scope is the begin-end block within which they were defined
  - module — endmodule
  - task — endtask
  - function — endfunction
  - begin:name — end
- ... nothing else within that scope may already have that name

Types of references
- **Forward referenced** — Identifiers for modules, tasks, functions, and named begin-end blocks may be used before being defined
- **Not Forward referenced** — must be defined before use
  - wires and registers
- **Hierarchical references** — named through the instantiation hierarchy
  - “a.b” references identifier b in namespace a
  - forward referenced
Identifiers

- **Forward referenced**
  - Identifiers of modules, tasks, functions, named-blocks
  - Hierarchical search tree defined by module instantiation
    - Identifiers within each higher scope are known
  - After all instantiations are known, search upward for the first identifier
    - a.b.c.d
    - When found go down through the rest of the name

- **Non-Forward referenced**
  - Identifiers for registers and wires (non-hierarchical)
  - Hierarchical search tree defined by nested procedural blocks
    - rooted in module
    - Search doesn’t cross module instantiation boundaries

- **Hierarchical — registers and wires**
  - These are forward referenced — see above
Scope of functions and tasks

Where defined
- functions and tasks are defined within modules

Scope
- As with other names, the scope of the functions and tasks is the begin-end block (module-endmodule) within which they are defined
- They can also be accessed hierarchically
  - define “global” functions and tasks in the “top” module
  - they’ll be accessible from any (recursively) instantiated module.
A few examples

module a (...);
    reg e;
    task b;
    reg c;
    begin : d
        reg e;
        e = 1;
        a.e = 0;
    end
    endtask
always
    begin : f
        reg g;
        a.b.d.e = 2;
        g = q.a.b.d.e;
        e = 3;
    end
endmodule

e’s hierarchical name is ...a.b.d.e

Chapter 2.6