CSE241
VLSI Digital Circuits
Winter 2003

Recitation 1: RTL Coding in Verilog
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
Introduction

- Learn Verilog basics
  - Hardware Description Language Semantics
  - Verilog Syntax
  - Features

- How to use Verilog for behavioral design

- How to use Verilog for structural design

- How to write Verilog for synthesis (brief)

- Examples!
Verilog from 20,000 Feet

- Verilog Descriptions look like programs:

<table>
<thead>
<tr>
<th>C / Pascal</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedures/Functions</td>
<td>Modules</td>
</tr>
<tr>
<td>Procedure parameters</td>
<td>Ports</td>
</tr>
<tr>
<td>Variables</td>
<td>Wires / Regs</td>
</tr>
</tbody>
</table>

- Block structure is a key principle
  - Use hierarchy/modularity to manage complexity

- But they aren’t ‘normal’ programs
  - Module evaluation is concurrent. (Every block has its own “program counter”)
  - Model is really communicating blocks
Introduction - Motivation

- Generic HDL uses:
  - Simulation
    - Test without build
  - Synthesis
    - Real hardware (gates)
  - Documentation
    - Self documentating
    - Portable
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
Quick Verilog History

- The Verilog Language
- Verilog HDL (Hardware Description Language) was concocted by Gateway Design Automation
- Later put in the public domain by Cadence Design Systems in order to promote the language as a standard
HDL and High Level Languages

- Verilog models look like programs
- Descriptions are partitioned into Verilog modules
- Modules resemble subroutines in that you can write one description and use (instantiate) it in multiple places.

- HDLs represent:
  - Electricity
    - Ultimately a physical entity
  - Parallelism - Concurrency
  - Time
Hardware Description Languages

- Need a description level up from logic gates.

- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an ALU, or could be a microprocessor

- The description consists of functions blocks and their interconnections
  - Describe functional block (not predefined)
  - Support hierarchical description (function block nesting)

- To make sure the specification is correct, make it executable.
  - Run the functional specification and check what it does
HDLs vs High Level Software Languages

- **High Level Languages**
  - Matlab or C are algorithmic
  - Execute in hardware

- **Verilog**
  - Blocks operate concurrently
  - Similar to real hardware blocks (chips)
  - Have interface signals (similar to variables)

- **Example:**

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Module</td>
</tr>
<tr>
<td>Parameters</td>
<td>Ports</td>
</tr>
<tr>
<td>~Variables</td>
<td>Wires</td>
</tr>
<tr>
<td>Variables</td>
<td>Reg</td>
</tr>
</tbody>
</table>
Topic Outline

- Introduction
- Verilog Background
- Syntax
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Examples
Verilog Naming Conventions

- The following must be used in all code:
  - Two slashes “//” are used to begin single line comments
  - A slash and asterisk “/*” are used to begin a multiple line comment and an asterisk and slash “*/” are used to end a multiple line comment.
  - Names can use alphanumeric characters, the underscore “_” character, and the dollar “$” character
  - Names must begin with an alphabetic letter or the underscore.
  - Spaces are not allowed within names

- Strings:
  - “hello world” //string
## Reserved Keywords

The following is a list of the Verilog reserved keywords:

<table>
<thead>
<tr>
<th>Always</th>
<th>Endmodule</th>
<th>Medium</th>
<th>Reg</th>
<th>Tranif0</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>Endprimitive</td>
<td>Module</td>
<td>Release</td>
<td>Tranif1</td>
</tr>
<tr>
<td>Assign</td>
<td>Endspecify</td>
<td>Nand</td>
<td>Repeat</td>
<td>Tri</td>
</tr>
<tr>
<td>Attribute</td>
<td>Endtable</td>
<td>Negedge</td>
<td>Rnmos</td>
<td>Tri0</td>
</tr>
<tr>
<td>Begin</td>
<td>Endtask</td>
<td>Nmos</td>
<td>Rpmos</td>
<td>Tri1</td>
</tr>
<tr>
<td>Buf</td>
<td>Event</td>
<td>Nor</td>
<td>Rtran</td>
<td>Triand</td>
</tr>
<tr>
<td>Bufif0</td>
<td>For</td>
<td>Not</td>
<td>Rtranif0</td>
<td>Trior</td>
</tr>
<tr>
<td>Bufif1</td>
<td>Force</td>
<td>Notif0</td>
<td>Rtranif1</td>
<td>Triireg</td>
</tr>
<tr>
<td>Case</td>
<td>Forever</td>
<td>Notif1</td>
<td>Scalared</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Casex</td>
<td>Fork</td>
<td>Or</td>
<td>Signed</td>
<td>Vectored</td>
</tr>
<tr>
<td>Casez</td>
<td>Function</td>
<td>Output</td>
<td>Small</td>
<td>Wait</td>
</tr>
<tr>
<td>Cmos</td>
<td>Highz0</td>
<td>Parameter</td>
<td>Specify</td>
<td>Wand</td>
</tr>
</tbody>
</table>
Reserved Keywords (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>deassign</td>
<td>highz1pmos</td>
<td>param</td>
<td>spec</td>
<td>weak0</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>if</td>
<td>posedge</td>
<td>strength</td>
<td>weak1</td>
<td></td>
</tr>
<tr>
<td>defparam</td>
<td>ifnone</td>
<td>primitive</td>
<td>strong0</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>disable</td>
<td>initial</td>
<td>pull0</td>
<td>strong1</td>
<td>wire</td>
<td></td>
</tr>
<tr>
<td>edge</td>
<td>inout</td>
<td>pull1</td>
<td>supply0</td>
<td>wor</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>input</td>
<td>pulldown</td>
<td>supply1</td>
<td>xnor</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>integer</td>
<td>pullup</td>
<td>table</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>endattribute</td>
<td>join</td>
<td>remos</td>
<td>task</td>
<td></td>
<td></td>
</tr>
<tr>
<td>endcase</td>
<td>large</td>
<td>real</td>
<td>time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>endfunction</td>
<td>macromodule</td>
<td>realtime</td>
<td>tran</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Numbers

- Example of number notation:

  - `<size> ‘ <base format><number>`

- Sized
  - 4’b1111  // 4bit binary number
  - 12’habc  //12 bit hexadecimal number
  - 16d2’55  //16 bit decimal number

- Unsized
  - 234  // decimal number
  - ‘hc3  //32 bit hexadecimal number
  - ‘o21  //32 bit octal number

- X or Z
  - 12’h13x  // 12 bit hex number
Operators

- **Arithmetic**
  - * multiply
  - / divide
  - + add
  - - subtract
  - % modulus

- **Logical**
  - ! Not
  - && and
  - || or

- **Relational**
  - > greater
  - < less
  - >= greater-equal
  - <= less-equal

- **Equality**
  - == equal
  - != not equal
  - === (case equality)

- **Bitwise**
  - ~ negation
  - & and
  - \ or
  - ^ xor
  - ^~ xnor

- **Others**
  - Reduction
  - Shift
  - Replication
Topic Outline

- Introduction
- Verilog Background
- Syntax
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Examples
Ports

- Keywords:
  - input - input
  - output - output
  - inout - bi-directional

- Ports do not store information

- Example
  
  ```verilog
  module ex (a, b, c, out)
  output out;
  input a, b, c;
  endmodule
  ```
Wires and Nets

- **Wires**
  - Connection between hardware elements
  - Module connections
  - Used to connect signals from sensitivity list
  - Memoryless

- **Example:**
  
  ```
  wire a; //declared wire net
  wire b = 1'b0 // tied to zero at declaration
  ```

- **Advanced data types**
  - tri
  - trireg
Memory Elements

- **Register**
  - Keyword = reg
  - Represents storage
  - Same as a variable (as opposed to wire)

- **Examples:**
  - reg clock; // clock
  - reg [0:4] vec_reg // 5 bit register vector
Topic Outline

- Introduction
- Verilog Background
- Syntax
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Examples
Modules

- Main lexical unit in Verilog
  - Functional block
  - Keyword = module/ endmodule
  - Is used for all dataflow types
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Examples
Procedural Statements

- Control statements
  - This type of control statement implies **sequential ordering**
  - keyword *always* provides functionality of a tiny program that executes sequentially

- Inside an *always* block, can use standard control flow statements:
  - *if* (*<conditional>*)) then *<statements>* else *<statements>*;
  - *case* (*<var>*): *<value>*: *<statements>*; … *default*: *<statements>*

- Case statements are prioritized
  - The second case entry can’t happen unless the first does not match.
  - May not be what the actual hardware implies – especially when cases are mutually exclusive.
  - Need additional directives (parallel-case) to indicate this
  - Statements can be compound (use *begin* and *end* to form blocks)

**Example:**

```verilog
always @(Activation List)
begin
  if (x==y) then
    out= in1
  else
    out = in2;
end
```
Initial Block

- Another type of procedural block
  - Does not need an activation list
  - It is run just once, when the simulation starts

- Used at the very start of simulation
  - Initialize simulation environment
  - Initialize design
    - This is usually only used in the first pass of writing a design
    - NOT Synthesizable, real hardware does not have initial blocks
  - Allows testing of a design (outside of the design module)

- Use initial blocks only for non-hardware statements (like $display or $gr_waves)
Module vs. Procedure

- Module is a method of building structural hierarchy
- Procedure (function)
  - Method of building behavioral hierarchy
  - Value within procedure unchanged, until next assignment
Blocking vs Non-blocking

- Scheduling of events
  - Procedural code

- Blocking
  - Block execution of following statement
  - Ex:
    ```
    begin
    count = 0;
    #10 b = 1;
    #15 count = count + 1;
    count = count + 2;
    ```
  - Count will execute at time 25, because of b assignment blocking
Non-blocking

- Block execution of following statement
- Ex:
  ```verilog
  begin
      count = 0;
      #10 b <= 1;
      #15 count <= count + 1;
      count <= count + 2;
  ```
- Count will execute at time 0, because of b non-blocking assignment

- Note: <= is only a relational operator in assign statement
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Examples
Structural Description

- Modules
  - Represent macros
  - Simulate some wanted function
  - Can contain hierarchy
## Modules

### Structure

- **Module Name:**
  - Port list, port declaration
  - Parameters.

- **Variables:**
  - Wires, regs.

- **Dataflow:**
  - Assign statement

- **Blocks:**
  - Initial, always.
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.
    - You could describe your lab1 circuit this way

- Identify:
  - Gate instances, wire names, delay

```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
```
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Delay
- Behavioral
- Testbenches
- Simulation
- Examples
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 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.

  ```verilog
  8'b00x0zx10
  ```

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

```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

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

```verilog
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.
Activation Lists

- Contained in `always` block

- Definition: Activation List
  - Tells the simulator when to run this block
    - NOTE!! If every input is not sensitized, a latch is created
  - But also enables subtle errors to enter into the design

- Activation lists in Verilog:
  - `@(signalName or signalName or ...)`
    - Evaluate this block when any of the named signals change (either positive or negative change)
  - `@(posedge signalName); or @(negedge signalName);`
    - Makes an edge triggered flip-flop
    - Evaluates only on one edge of a signal
    - Can have `@(posedge signal1 or negedge signal2)`
    - Only allow “or” not “and” because edges are singular events
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Examples
Testbenches

- Behavioral blocks
  - Also stimulus blocks
  - Are used to feed signals to main block
Delay Models

- **Verilog simulation time**
  - Execution time of the verilog model
    - When the computer completes with all the “events” that occur at the current simulated time
    - The computer increases time until another signal is scheduled to change values.

- **Behavioral delay assignments within the blocks**
  - # delayAmount
    - Simulator sees this symbol, and stops evaluation
    - Pause delayAmount of simulated time (# of ticks).
    - Delays are often used to model the delay in functional units.
    - Can be tricky to use properly
  - RTL is delay-less for synthesis
    - Use only in testbench
    - The standard cell library we use can annotate delay information.
Declarative Delay Control

- A way to specifying delay of a signal

- Make out a delayed version of the input (by 10 ticks)
  - `assign #10 out = in;`
  - Delayed assignment.

- Anywhere else to put delay is not allowed
  - `assign out = #10 in; //is not allowed`

![Diagram showing input and output signals with 10 tick delay]
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”)

```verilog
module design (a, b, c);
    input a, b;
    output c;
...
module test (q, r);
    output q, r;
...
module testbench ();
    wire l, m, n;
    design d (l, m, n);
    test t (l, m);
    initial begin
        //monitor and display
        ...
    end
```
Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation
- Example
- Reference: Coding Style
Creating Code

- **Example:**
  - Given a specification – “build full adder”
  - Name signals:
    - Inputs: carry_in, A, B
    - Outputs: carry_out, sum
  - Math:
    - Sum = (A xor B xor carry_in)
    - Carry_out = (A ?B) + carry_in ? (A ?B)
  - Need:
    - Module name
    - Algorithm (see math)
Full-adder Code

- Sample Code

```verilog
module full_adder (a, b, ci, sum, co);  // lists full input/output signal list

input  a, b, ci; //input declaration
output sum, co;   //output declaration

assign sum = a ^ b ^ ci;
assign co = (a & b) | (a & ci) | (b & ci);

endmodule
```

Sensitivity List
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;    assumes a is instantiated in q
            e = 3;
        end
endmodule