EECS 470 Lab 1
Verilog: Hardware Description Language

Department of Electrical Engineering and Computer Science
College of Engineering
University of Michigan

Friday, 10th January, 2014
Overview

GSI’s Guide to 470
- Labs
- Projects

Administrivia

Verilog
- Behavioral v. Structural Semantics
- Data Types
- Operators
- Procedural Elements
- Assignments
- Flow Control
- Modules
- Synthesis

Testing Verilog Designs

Conclusion

Lab Assignment
Who?

- Contact Information
  - Jonathan Beaumont – jbbeau@umich.edu
  - William Cunningham – wcunning@umich.edu
  - Course Staff Email – eecs470w14staff@umich.edu
  - EECS 470 Piazza
    - Most of your project related questions should be asked here so that other people can benefit from the answer.
    - Please do not post code in public questions.

- Office Hours can be found in the course google calendar
Where? When?

▶ Two Lab Sections
  ▶ Both in 1620 BBB
  ▶ 012 – 10:30am to 12:30pm
  ▶ 011 – 2:30pm to 4:30pm

▶ These will become extra office hours after the midterm, for extra project help.
Lab 1 – Verilog: Hardware Description Language
Lab 2 – The Build System
Lab 3 – Writing Good Testbenches
Lab 4 – Revision Control
Lab 5 – Scripting
Lab 6 – SystemVerilog
Projects

Individual Verilog Projects

Project 1 – Priority Selector (2%)
Project 2 – Pipelined Multiplier, Integer Square Root (2%)
Project 3 – Verisimple 5-stage Pipeline (3%)

Group Project

Project 4 – Out-of-Order Alpha64 Processor (35%)
These projects will take a non-trivial amount of time, especially if you’re not a verilog guru.

You should start them early. Seriously...

Especially Project 3
Project 4

- Subset of the Alpha64 Instruction Set Architecture
- Groups of 4 to 5
  - Start thinking about your groups now
  - You’ll be spending hundreds of hours together this semester, so work with people you get along with.
- Heavy Workload
  - 100 hours/member, minimum
  - 150 to 300 hours/member, more realistically
  - This is a lower bound, not an upper...
- Class is heavily loaded to the end of the term
Homework 1 is due Wednesday, 15\textsuperscript{th} January, 2014, in lecture.

Project 1 is due Tuesday, 21\textsuperscript{st} January, 2014, turned in online.
What is Verilog?

- Hardware Description Language - IEEE 1364-2005
  - Replaced by SystemVerilog - IEEE 1800-2009
- Two Forms
  1. Behavioral
  2. Structural
- It can be built into hardware. If you can’t think of at least one (inefficient) way to build it, it might not be good.
Intro to Verilog

What is Verilog?

- Hardware Description Language - IEEE 1364-2005
  - Replaced by SystemVerilog - IEEE 1800-2009
- Two Forms
  1. Behavioral
  2. Structural
- It can be built into hardware. If you can’t think of at least one (inefficient) way to build it, it might not be good.

Why do I care?

We use Behavioral Verilog to do computer architecture here.

Semiconductor Industry Standard (in America at least, VHDL is widely used in Europe)
Intro to Verilog

What is Verilog?

- Hardware Description Language - IEEE 1364-2005
  - Replaced by SystemVerilog - IEEE 1800-2009
- Two Forms
  1. Behavioral
  2. Structural
- It can be built into hardware. If you can’t think of at least one (inefficient) way to build it, it might not be good.

Why do I care?

- We use Behavioral Verilog to do computer architecture here.
- Semiconductor Industry Standard (in America at least, VHDL is widely used in Europe)
The Difference Between Behavioral and Structural Verilog

Behavioral Verilog

- Describes function of design
- Abstractions
  - Arithmetic operations
    (+, -, *, /)
  - Logical operations
    (&, |, ^, ~)

Structural Verilog

- Describes construction of design
- No abstraction
- Uses modules, corresponding to physical devices, for everything

Suppose we want to build an adder?
Figure: 1-bit Full Adder
module one_bit_adder(
    input a, b, cin,
    output wire sum, cout);
wire w_0, w_1, w_2;
xor u0(w_0, a, b);
xor u1(sum, w_0, cin);
and u2(w_1, w_0, cin);
and u3(w_2, a, b);
or u4(cout, w_1, w_2);
endmodule
module one_bit_adder(
    input a,b,cin,
    output wire sum,cout);
assign sum = a ^ b ^ cin;
assign cout = ((a ^ b) & cin) | a & b;
endmodule
module one_bit_adder(
    input a, b, cin,
    output logic sum, cout);

always_comb
begin
    sum = a ^ b ^ cin;
    cout = 1'b0;
    if ((a ^ b) & cin) | (a & b))
        cout = 1'b1;
end
endmodule
Lexical

- Everything is case sensitive.
- Type instances must start with A−Z, a−z, _. They may contain A−Z, a−z, 0−9, _, $.
- Comments begin with // or are enclosed with /* and */.
Synthesizable Data Types

**wires**  Also called nets

```verilog
wire a_wire;
wire [3:0] another_4bit_wire;
```

- Cannot hold state

**logic**  Replaced reg in SystemVerilog

```verilog
logic [7:0] an_8bit_register;
reg a_register;
```

- Holds state, might turn into flip-flops
- Less confusing than using reg with combinational logic (coming up...)
Data Types

Unsynthesizable Data Types

- `integer`  Signed 32-bit variable
- `time`      Unsigned 64-bit variable
- `real`      Double-precision floating point variable
Types of Values

Four State Logic

0  False, low
1  True, high
Z  High-impedance, unconnected net
X  Unknown, invalid, don’t care
Values

Literals/Constants

- Written in the format `<bitwidth>’<base><constant>`
- Options for `<base>` are...
  - b  Binary
  - o  Octal
  - d  Decimal
  - h  Hexadecimal

```verilog
assign an_8bit_register = 8’b10101111;
assign a_32bit_wire = 32’hABCD_EF01;
assign a_4bit_logic = 4’hE;
```
## Verilog Operators

### Arithmetic
- `*` Multiplication
- `/` Division
- `+` Addition
- `-` Subraction
- `%` Modulus
- `**` Exponentiation

### Bitwise
- `~` Complement
- `&` And
- `|` Or
- `~|` Nor
- `^` Xor
- `~^` Xnor

### Logical
- `!` Complement
- `&&` And
- `||` Or

### Shift
- `>>` Logical right shift
- `<<` Logical left shift
- `>>>` Arithmetic right shift
- `<<<` Arithmetic left shift

### Relational
- `>` Greater than
- `>=` Greater than or equal to
- `<` Less than
- `<=` Less than or equal to
- `!=` Inequality
- `!==` 4-state inequality
- `==` Equality
- `===` 4-state equality

### Special
- `{,}` Concatenation
- `{n{m}}` Replication
- `?:` Ternary
Procedural Statements

**assign Statements**

- One line descriptions of combinational logic
- Left hand side must be a wire
- Right hand side can be any one line verilog expression
- Including (possibly nested) ternary (?:)

**Example**

```verilog
module one_bit_adder(
    input wire a, b, cin,
    output wire sum, cout);
assign sum = a ^ b ^ cin;
assign cout = ((a ^ b) & cin) | a & b;
endmodule
```
always Blocks

- Contents of `always` blocks are executed whenever anything in the sensitivity list happens
- Two main types in this class...
  - `always_comb`
    - implied sensitivity lists of every signal inside the block
    - Used for combinational logic. Replaced `always @*`
  - `always_ff @(posedge clk)`
    - sensitivity list containing only the positive transition of the `clk` signal
    - Used for sequential logic
- All left hand side signals need to be `logic` type.
Always Block Examples

Combinational Block

always_comb
begin
    x = a + b;
    y = x + 8’h5;
end

Sequential Block

always_ff @(posedge clk)
begin
    x <= #1 next_x;
    y <= #1 next_y;
end
Blocking vs. Nonblocking Assignments

**Blocking Assignment**
- Combinational Blocks
- Each assignment is processed in order, earlier assignments block later ones
- Uses the = operator

**Nonblocking Assignment**
- Sequential Blocks
- All assignments occur “simultaneously,” delays are necessary for accurate simulation
- Uses the <= operator
Blocking vs. Nonblocking Assignment by Example

Blocking Example

always_comb
begin
    x = new_val1;
    y = new_val2;
    sum = x + y;
end

- Behave exactly as expected
- Standard combinational logic

Figure: Timing diagram for the above example.
Blocking vs. Nonblocking Assignment by Example

Nonblocking Example

```verilog
always_ff @(posedge clock)
begin
    x <= #1 new_val1;
    y <= #1 new_val2;
    sum <= #1 x + y;
end
```

- What changes between these two examples?
- Nonblocking means that `sum` lags a cycle behind the other two signals.

*Figure: Timing diagram for the above example.*
Bad Example

```verilog
always_ff @(posedge clock)
begin
    x <= #1 y;
    z = x;
end
```

- z is updated after x
- z updates on negedge
- clock

![Timing diagram for the above example.](figure)

Figure: Timing diagram for the above example.
All Flow Control

- Can only be used inside procedural blocks (always, initial, task, function)
- Encapsulate multiline assignments with begin...end
- Remember to assign on all paths

Synthesizable Flow Control

- if/else
- case
Flow Control

Unsynthesizable Flow Control

- Most flow control is synthesizable
- Most is also dangerous/inefficient
- For example...
  - for
  - while
  - forever
  - repeat
Flow Control by Example

Synthesizable Flow Control Example

```verilog
always_comb
begin
  if (muxy == 1'b0)
    y = a;
  else
    y = b;
end
```

The Ternary Alternative

```verilog
wire y;
assign y = muxy ? b : a;
```
**Flow Control by Example**

**Casez Example**

```verilog
always_comb
begin
    casez(alu_op)
        3'b000: r = a + b;
        3'b001: r = a - b;
        3'b010: r = a * b;
        ...
        3'b1??: r = a ^ b;
    endcase
end
```

(University of Michigan) Lab 1: Verilog 10-01-2014 32 / 55
Intro to Modules

- Basic organizational unit in Verilog
- Can be reused

Module Example

```verilog
module my_simple_mux(
    input wire select_in, a_in, b_in; //inputs listed
    output wire muxed_out); //outputs listed
    assign muxed_out = select_in ? b_in : a_in;
endmodule
```
Modules

Writing Modules

- Inputs and outputs must be listed, including size and type format: `<dir> <type> <[WIDTH-1:0]> <name>;
  e.g. output logic [31:0] addr;
- In module declaration line or after it, inside the module

Instantiating Modules

- Two methods of instantiation
  1. e.g. `my_simple_mux m1(.a_in(a), .b_in(b), .select_in(s), .muxed_out(m));`
  2. e.g. `my_simple_mux m1(a,b,s,m);`
- The former is much safer...
- Introspection (in testbenches): `module.submodule.signal`
Synthesis

What is synthesis?

- Process of turning behavioral Verilog into structural
- Only applies to a subset of Verilog
What is synthesis?

- Process of turning behavioral Verilog into structural
- Only applies to a subset of Verilog

Why do I care?

- Used to determine viability of designs
- Used to compare design performance
Synthesis Tips

Latches

- What is a latch?

Memory device without a clock
Generated by a synthesis tool when a net needs to hold state without being clocked (combinational logic)
Generally bad, unless designed in intentionally
Unnecessary in this class
Synthesis Tips

Latches

► What is a latch?
  ► Memory device without a clock
► Generated by a synthesis tool when a net needs to hold state without being clocked (combinational logic)
► Generally bad, unless designed in intentionally
► Unnecessary in this class
Synthesis Tips

Latches

- Always assign every variable on every path
- This code generates a latch
- Why does this happen?

```verilog
always_comb
begin
    if (cond)
        next_x = y;
end
```
Synthesis Tips

Possible Solutions to Latches

```verilog
always_comb
begin
    next_x = x;
    if (cond)
        next_x = y;
end
```

```verilog
always_comb
begin
    if (cond)
        next_x = y;
    else
        next_x = x;
end
```
Keys to Synthesizability

Combinational Logic

- Avoid feedback (combinatorial loops)
- Always blocks should
  - Be always_comb blocks
  - Use the blocking assignment operator =
- All variables assigned on all paths
  - Default values
  - if(...) paired with an else
Keys to Synthesizability

Sequential Logic

- Avoid clock- and reset-gating
- Always blocks should
  - Be always_ff @(posedge clock) blocks
  - Use the nonblocking assignment operator, with a delay $\leq #1$
- No path should set a variable more than once
- Reset all variables used in the block
- //synopsys sync_set_reset "reset"
Testing

What is a test bench?

- Provides inputs to one or more modules
- Checks that corresponding output makes sense
- Basic building block of Verilog testing

Why do I care?

- Finding bugs in a single module is hard...
- But not as hard as finding bugs after combining many modules
- Better test benches tend to result in higher project scores
Features of the Test Bench

▶ Unsynthesized
  ▶ Remember unsynthesizable constructs? This is where they’re used.
  ▶ In particular, unsynthesizable flow control is useful in testbenches (e.g. repeat, while)

▶ Programmatic
  ▶ Many programmatic, rather than hardware design, features are available e.g. functions, tasks, classes (in SystemVerilog)
Anatomy of a Test Bench

A good test bench should, in order...

1. Declare inputs and outputs for the module(s) being tested
2. Instantiate the module (possibly under the name DUT for Device Under Test)
3. Setup a clock driver (if necessary)
4. Setup a correctness checking function (if necessary/possible)
5. Inside an initial block...
   5.1 Assign default values to all inputs, including asserting any available reset signal
   5.2 $monitor or $display important signals
   5.3 Describe changes in input, using good testing practice
Unsynthesizable Procedural Blocks

initial Blocks

- Procedural blocks, just like always
- Contents are simulated once at the beginning of a simulation
- Used to set values inside a test bench
- Should only be used in test benches
Unsythesizable Procedural Blocks

initial Block Example

```verilog
initial
begin
    @(negedge clock);
    reset = 1'b1;
    in0 = 1'b0;
    in1 = 1'b1;
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    in0 = 1'b1;
    ...
end
```
Tasks and Functions

**task**
- Reuse commonly repeated code
- Can have delays (e.g. #5)
- Can have timing information (e.g. @(negedge clock))
- Might be synthesizable (difficult, not recommended)

**function**
- Reuse commonly repeated code
- No delays, no timing
- Can return values, unlike a task
- Basically combinational logic
task exit_on_error;
    input [63:0] A, B, SUM;
    input C_IN, C_OUT;
    begin
        $display("@@@ Incorrect at time \%4.0f", $time);
        $display("@@@ Time:\%4.0f clock:\%b A:\%h B:\%h CIN:\%b SUM:\%h"
            "COUT:\%b", $time, clock, A, B, C_IN, SUM, C_OUT);
        $display("@@@ expected sum=\%b", (A+B+C_IN) );
        $finish;
    end
endtask
function Example

function check_addition;
    input wire [31:0] a, b;
    begin
        check_addition = a + b;
    end
endfunction

assign c = check_addition(a,b);
Intro to System Tasks and Functions

- Just like regular tasks and functions
- But they introspect the simulation
- Mostly these are used to print information
- Behave just like printf from C
List of System Tasks and Functions

$monitor  Used in test benches. Prints every time an argument changes. Very bad for large projects.
  e.g. $monitor("format",signal,...)

$display  Can be used in either test benches or design, but not after synthesis. Prints once. Not the best debugging technique for significant projects.
  e.g. $display("format",signal,...)

$strobe   Like display, but prints at the end of the current simulation time unit.
  e.g. $strobe("format",signal,...)

$time     The current simulation time as a 64 bit integer.

$reset   Resets the simulation to the beginning.

$finish Exit the simulator, return to terminal.

More available at ASIC World.
module testbench;
	numeric clock, reset, taken, transition, prediction;

two_bit_predictor(  
	.clock(clock),
	.reset(reset),
	.taken(taken),
	.transition(transition),
	.prediction(prediction));

always begin

clock = #5 ~clock;
end
Test Benches by Example

Test Bench Test Cases

```verilog
initial
begin
    $monitor("Time:%4.0f clock:%b reset:%b taken:%b transition:%b prediction:%b",
        $time, clock, reset, taken, transition, prediction);
    clock = 1'b1;
    reset = 1'b1;
    taken = 1'b1;
    transition = 1'b1;
    @(negedge clock);
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    taken = 1'b1;
    @(negedge clock);
    ...
    $finish;
end
```
How to Design with Verilog

- Remember – Behavioral Verilog implies no specific hardware design
- But, it has to be synthesizable
- Better be able to build it somehow
Test Bench Tips

Remember to...

▶ Initialize all module inputs
▶ Then assert reset
▶ Use @(negedge clock) when changing inputs to avoid race conditions
Assignment is posted to the course website as 470 tutorial

If you get stuck...
  ▶ Ask a neighbor, quietly
  ▶ Put yourself in the help queue

When you finish the assignment, sign up in the help queue and mark that you would like to be checked off.

If you are unable to finish today, the assignment needs to be checked off by a GSI in office hours before the next lab session.