A Verilog Primer

An Overview of Verilog for Digital Design and Simulation

John Wright   Vighnesh Iyer

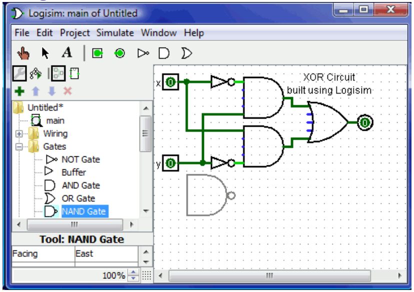
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

What is Verilog?

Visual Circuit Design

Digital circuits can be designed by laying out circuit elements. This was done in CS 61C. Schematic entry.

Logisim

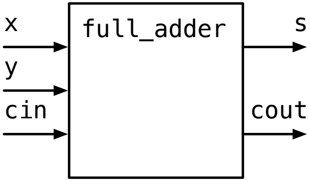


- Verilog is a HDL (hardware definition language) that can describe digital circuits with C-like syntax.
- Defines circuits at the RTL (register-transfer level) of abstraction
- The circuit on the left could be written in Verilog as `assign output = x ^ y`
- ASIC or FPGA toolchains translate Verilog to a gate-level netlist

Verilog Modules

- Modules are the building blocks of Verilog designs. They are a means of abstraction and encapsulation for your design.
- A module consists of a port declaration and Verilog code to implement the desired functionality.
- Modules should be created in a Verilog file (.v) where the filename matches the module name (the module below should be stored in full_adder.v)

```
module full_adder (input x, input y, input cin, output s, output cout);
endmodule
```



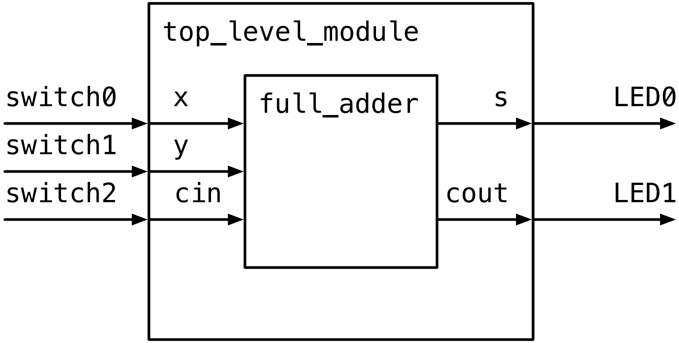
Verilog Module I/O Ports

- Ports allow communication between a module and its environment
- Each port has a name and a type
  - input
  - output
  - inout
- Ports for a module are declared in the port declaration

```
module full_adder (input x, input y, input cin, output s, output cout);
        // Verilog code here has access to inputs
        // Verilog code here can drive outputs
endmodule
```

The Top-Level Module

- Every Verilog design has a top-level module which sits at the highest level of the design hierarchy
- The top-level module defines the I/O for the entire digital system
- All the modules in your design reside inside the top-level module



Verilog Module Instantiation

- Modules can be instantiated inside other modules. The syntax used is `<module name> <instance name> (.port0(wire), .port1(wire), ...)`

```
module top_level (input switch0,
                  input switch1,
                  input switch2,
                  output LED0,
                  output LED1);
        full_adder add (
                .x(switch0),
                .y(switch1),
                .cin(switch2),
                .s(LED0),
                .cout(LED1)
        );
endmodule
```

Wire Nets

- Wires are analogous to wires in a circuit you build by hand; they are used to transmit values between inputs and outputs. Declare wires before they are used.

```
wire a;
wire b;
```

- The wires above are scalar (represent 1 bit). They can also be vectors:

```
wire [7:0]  d;  // 8-bit wire declaration
wire [31:0] e; // 32-bit wire declaration
```

Multi-bit Nets

- We can declare signals that are more than 1 bit wide in Verilog
- Use the syntax `[MSB bit index :  LSB bit index]` before a signal name to declare its bit-width
- When connecting multi-bit inputs or outputs to another module, the bit-widths of the signals need to match!

```
module two_bit_adder (input [1:0] x, input [1:0] y, output [2:0] sum);
        wire [1:0] partial_sum;
        wire carry_out;
endmodule
```

Structural Verilog with Gate Primitives

Gate-Level Circuit Construction

- The following gate primitives exist in Verilog: and, or, xor, not, nand, nor, xnor. In general, the syntax is:

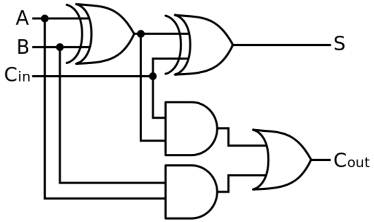
```
<operator> (output, input1, input2); // for two input gate
<operator> (output, input); // for not gate
```

Example of Verilog that implements the Boolean equation $f = a + b$:

```
wire a, b, f;
or (f, a, b);
```

Designing the Full Adder

Gate-Level Circuit



Designing the Full Adder

Using Structural Verilog

```
module full_adder (input a, input b, input cin, output s, output cout);
        xor(s, a, b, cin);

        wire xor_a_b;
        wire cin_and;
        wire and_a_b;

        xor(xor_a_b, a, b);
        and(cin_and, cin, xor_a_b);
        and(and_a_b, a, b);
        or(cout, and_a_b, cin_and);
endmodule
```

Behavioral Verilog

Letting the tools translate RTL to gates

- The full adder using structural Verilog was a pain to write, but you will never have to write it that way!
- Behavioral Verilog constructs allow you to describe what you want a circuit to do at the RTL level of abstraction
- The FPGA or ASIC toolchain will translate the Verilog code to the FPGA or ASIC primitives that implement your circuit description
- Verilog's gate primitives are typically not used in design

Assign Statements

- Wires can be assigned to logic equations, other wires, or operations performed on wires
- This is accomplished using the 'assign' statement
- The left argument of the assign statement must be a wire, and cannot be an input wire
- The right argument of the assign statement can be any expression created from Verilog operators and wires

```
module copy (input a, output b);
       assign b = a;
endmodule
```

Verilog Operators

- Verilog contains operators that can be used to perform arithmetic, form logic expression, perform reductions/shifts, and check equality between signals.

Operator Type	Symbol	Operation Performed
Arithmetic	+	Add
	-	Subtract
	*	Multiply
	/	Divide
	%	Modulus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or

Verilog Operators Cont.

Operator Type	Symbol	Operation Performed
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	Inequality
Bitwise	~	Bitwise negation
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR

Reduction operators also exist for AND, OR, and XOR that have the same symbol as the bitwise operators.

Verilog Operators Cont.

Shifts, Concatenation, Replication, Indexing multi-bit wires

Operator Type	Symbol	Operation Performed
Shift	<<	Shift left logical
	>>	Shift right logical
	<<<	Arithmetic left shift
	>>>	Arithmetic left shift
Concatenation	{}	Join bits
Replication	{{}}	Duplicate bits
Indexing/Slicing	[MSB:LSB]	Select bits

```
wire [7:0] d;
wire [31:0] e;
wire [31:0] f;
assign f = {d, e[23:0]}; // Concatenation + Slicing
assign f = { 32{d[5]} }; // Replication + Indexing
```

Designing the Full Adder

Using Behavioral Verilog

```
module full_adder (input x, input y, input cin, output s, output cout);
        assign s = x ^ y ^ cin;
        assign cout = (a && b) || (cin && (a ^ b));
endmodule
```

This is much better than the structural Verilog representation of a full adder! The Verilog synthesizer can take this opportunity to optimize logic expressions.

Designing an Adder

Using Behavioral Verilog

But it gets even better!

```
wire operand1 [31:0];
wire operand2 [31:0];
wire result [32:0];
assign result = operand1 + operand2;
```

We just described a 32-bit adder, but didn't specify the architecture or the logic! The Verilog synthesizer will turn the generalized adder into a FPGA specific netlist or an ASIC gate-level netlist.

Conditional (Ternary) Operator ?:

- The conditional operator allows us to define if-else logic in an assign statement.
- Syntax: (condition) ? (expression if condition is true) : (expression if condition is false)
- Example: an expression that implements min(a, 10)

```
assign out = a > 10 ? 10 : a;
```

- Example: output 0 if the two inputs are equal, output 1 if input a is less than input b, and output 2 if input a is greater than input b.

```
module equality_checker(input a [31:0], input b [31:0], output c [1:0]);
        assign c = a == b ? 2'd0 : (a < b ? 2'd1 : 2'd2);
endmodule
```

Verilog Literals

- Verilog defines a particular way of specifying literals
- Syntax: [bit width]'[radix][literal]
- Radix can be d (decimal), h (hexadecimal), o (octal), b (binary)
- It is critical to always match bit widths for operators and module connections, do not ignore these warnings from the tools

```
2'd1 2-bit literal (decimal 1)
16'hAD14 16-bit literal (hexadecimal 0xAD14)
8'b01011010 8-bit literal (binary 0b01011010)
```

Verilog Macros

Constant defines + Helper functions

- Macros in Verilog are similar to macros available in C
- `` `include `` is a preprocessor command to inject a Verilog source file at its location; often used to bring in a Verilog header file (.vh)
- `` `define <Constant Name> <Constant Value> `` is used to declare a synthesis-time constant; use these instead of putting 'magic' values in RTL
- `` `define <Macro function name>(ARGS) <Function body> `` is used to define a macro function that can generate RTL based on ARGS.
- `` `ifndef <NAME>, `define <NAME>, `endif `` is an include guard (used similarly in C).

Verilog Macros

Example

Inside constants.vh:

```
`ifndef CONSTANTS // guard prevents header file from being included more
↪    than once
`define CONSTANTS

`define ADDR_BITS 16
`define NUM_WORDS 32
`define LOG2(x) \
        (x <= 2) ? 1 : \
        (x <= 4) ? 2 : \
        (x <= 8) ? 3 : \
        (x <= 16) ? 4 : \
        (x <= 32) ? 5 : \
        (x <= 64) ? 6 : \
        -1
`endif
```

Verilog Macros

Example Cont.

Inside design.v:

```
`include "constants.vh"
module memory (input [`ADDR_BITS - 1:0] address,
↪  output [`LOG2(`NUM_WORDS) - 1:0] data);

        // implementation

endmodule
```

Reg Nets

Verilog has two types of net: `wire` and `reg`. Reg nets are required whenever a net must preserve state (i.e. in an always block). Wires are used for structural verilog (to connect inputs to outputs) and for continuous assignment.

```
reg x;
```

Combinational Logic Blocks using always@(*)

Verilog allows more complex logic through the use of `always` blocks. Combinational logic (i.e. no state elements) can be written using `always@(*)`. The value inside the parentheses is called the sensitivity list. Using a ∗ will tell the compiler to compute the sensitivity list automatically (recommended for combinational logic).

Only reg nets can be assigned in an always block

```
reg y;
reg x;
always @(*) begin
        x = ~y;
end
```

If-else statements

Like many programming languages, verilog includes `if-else` statements. These implicitly generate multiplexors in hardware. Multi-line code blocks require `begin` and `end` statements.

```
input w;
input y;
input x;
reg [1:0] z;
always @(*) begin
        if(x) begin
                z[0] = w;
                z[1] = y;
        end
        else begin
                z[0] = y;
                z[1] = x;
        end
end
```

Case statements

Like C, verilog supports case statements for generating multiplexor structures.

```
input [1:0] x;
reg [1:0] y;
always @(*) begin
    case(x)
        0: y = 2'd0;
        1: y = 2'd3;
        2: y = 2'd2;
        default: y = 2'd2;
    endcase
end
```

Non-Synthesizable Control Statements

WARNING: `for` and `while` loops can't be mapped to hardware!
These statements are valid verilog (and can be simulated), but cannot always be mapped to hardware.
Generate statements (more later) are the appropriate use for `for` loops.

Avoiding Unintentional Latch Synthesis

Every signal should have a default value. Assigning a value to a reg only under given conditions will result in latch synthesis. For example, this code generates a latch:

```
input [1:0] x;
reg [1:0] y;
always @(*) begin
    if(x == 2'b10) begin
        y = 2'd3;
    end else if(x == 2'b11) begin
        y = 2'd2;
    end
end
```

Avoiding Unintentional Latch Synthesis

This code has a default value and will not generate a latch:

```
input [1:0] x;
reg [1:0] y;
always @(*) begin
    y = 2'b00;
    if(x == 2'b10) begin
        y = 2'd3;
    end else if(x == 2'b11) begin
        y = 2'd2;
    end
end
```

Synchronous Logic Blocks

Synchronous logic blocks are generated using special identifiers in the sensitivity list. Here we only want to update on the positive edge of the clock, so we use `posedge`. This will generate a synchronous circuit that increments `x` every clock cycle.

```
input clk;
reg [1:0] x;

always @(posedge clk) begin
        x <= x + 1;
end
```

Blocking vs Non-Blocking Assignments

What was up with the <= operator on the last slide? Verilog has two assignment operators: blocking (=) and non-blocking (<=). For the purposes of this class, **always** use blocking (=) for combinational logic and non-blocking (<=) for sequential logic.

```
input clk;
reg [1:0] x;
reg [1:0] next_x;

always @(*) begin
    next_x = x + 1;
end

always @(posedge clk) begin
    x <= next_x;
end
```

localparam Declarations

Private module parameters are defined using the `localparam` directive. These are useful for constants that are needed only by a single module.

```
localparam coeff = 5;
reg [31:0] x;
reg [31:0] y;

always@(*) begin
        x = coeff*y;
end
```

Wire vs. Reg

What is a real register and what is just a wire?

Rules for picking a wire or reg net type:

- If a signal needs to be assigned inside an always block, it must be declared as a reg.
- If a signal is assigned using a continuous assignment statement, it must be declared as a wire.
- By default module input and output ports are wires; if any output ports are assigned in an always block, they must be explicitly declared as reg: `output reg <signal name>`

How to know if a net represents a register or a wire.

- A wire net always represents a combinational link
- A reg net represents a wire if it is assigned in an always `@ (*)` block
- A reg net represents a register if it is assigned in an always `@` (posedge/negedge clock) block

Code Generation with for-generate loops

Generate loops are used to iteratively instantiate modules. This is useful with parameters (next slide) or when instantiating large numbers of the same module.

```
wire [3:0] a, b;
genvar i;

core first_one (1'b0, a[i], b[i]);

// programmatically wire later instances
generate
    for (i = 1; i < 4 ; i = i + 1) begin:nameofthisloop
        core generated_core (a[i], a[i-1], b[i]);
    end
endgenerate
```

Verilog Module Parameterization

Verilog modules may include parameters in the module definition. This is useful to change bus sizes or with generate statements. Here we define and instantiate a programmable-width adder with a default width of 32.

```
module adder #(parameter width=32)
        (input [width-1:0] a,
         input [width-1:0] b,
         output [width:0] s);
        s = a + b;
endmodule

module top();
        localparam adder1width = 64;
        localparam adder2width = 32;
        reg [adder1width-1:0] a,b;
        reg [adder2width-1:0] c,d;
        wire [adder1width:0] out1;
        wire [adder2width:0] out2;
        adder #(.width(adder1width)) adder64 (.a(a), .b(b), .s(out1));
        adder #(.width(adder2width)) adder32 (.a(c), .b(d), .s(out2));
endmodule
```

Multidimensional Nets in Verilog

- It is often useful to have a net organized as a two-dimensional net to make indexing easier; this is particularly useful in any memory structure.
- Syntax: `reg [M:0] <netname> [N:0]`
- The line above creates a net called `<netname>` and describes it as an array of (N+1) elements, where each element is a (M+1) bit number.

```
// A memory structure that has eight 32-bit elements
reg [31:0] fifo_ram [7:0];
fifo_ram[2] // The full 3rd 32-bit element
fifo_ram[5][7:0] // The lowest byte of the 6th 32-bit element
```

Memory Module Design Example

High-Level Spec

Let's design a memory module with the following capabilities.

- Parameterized number of bits per word
- Parameterized number of words
- Asynchronous reads (as soon as an address is driven into the memory, after a combinational delay, the data at the address will come out of the memory)
- Synchronous writes (at a rising clock edge, if the write signal is high, write the memory with the contents on the data input line and at the address provided as input)

Memory Module Design Example

Module + Port Declaration

```
module memory #(
        parameter BITS_PER_WORD = 32,
        parameter NUM_WORDS = 128)(
        input clk,
        input [`LOG2(NUM_WORDS) - 1 : 0] write_address,
        input [BITS_PER_WORD - 1 : 0] write_data,
        input [`LOG2(NUM_WORDS) - 1 : 0] read_address,
        input write_enable,
        output [BITS_PER_WORD - 1 : 0] read_data
);
        // RTL Code
endmodule
```

Memory Module Design Example

RTL Code

```
reg [BITS_PER_WORD - 1 : 0] mem_array [NUM_WORDS - 1
 ↪  : 0];
assign read_data = mem_array[read_address];

always @ (posedge clk) begin
        if (write_enable) begin
                mem_array[write_address] <=
                 ↪  write_data;
        end
end
```

Initial Blocks and Test Benches

WARNING: NOT SYNTHESIZABLE!
Initial blocks are primarily used for test benches. They contain sequences of code to be executed at the beginning of the simulation. The delay operator (#) and at operator (@) are used in the inital block to step through time events.

```
`define clock_period 5
reg [31:0] a, b;

reg clk = 1'b0; // give it an initial value
always #(`clock_period/2) clk = ~clk; // toggle clock every half cycle

module dut mymodule(.clk(clk), .in0(a), .in1(b), .out(c));

initial begin
        clk = 1'b0;
        a = 32'h01234567;
        b = 32'h00000000;
        #1 a = 32'h10101010; // wait 1 time unit
        #1 b = 32'h01010101;
        #1 a = 32'h00000000;
           b = 32'h11111111; // a and b change together
        @(posedge clk);      // jump to next rising edge
        @(posedge clk);      // jump to next rising edge
        a = 32'hFEDCBA98;
        #10;
        // observe some output here.
end
```

Initial Blocks and Test Benches

Verilog test benches should include a `` `timescale `` directive. The first value defines the time unit, and the second defines the simulation resolution. In this example, the default time unit is 1 nanosecond and the simulation resolution is 1 picosecond.

```
`timescale 1ns/1ps
```

A delay of `#1` would result in a 1 ns step. Delays as low as `#0.001` would be supported due to the resolution of 1 ps.