

Introduction to Design With Verilog

Dr. Paul D. Franzon

Outline

1. HDL-based Design Flow
2. Introduction to the Verilog Hardware Description Language
3. A complete example: count.v
4. Verilog features to be used in this class

References

1. Sutherland, Quick Reference Guide
2. Smith, Chapter 11
2. See Course Outline for further list of references

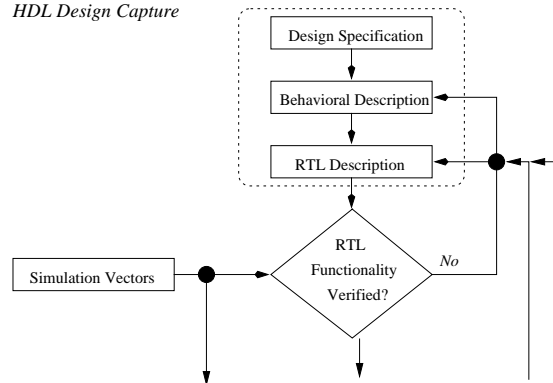
Role of Hardware Description Languages

Modern digital chip and system design centers on the use of Hardware Description Languages (HDL) to capture the design at the Register Transfer Level (RTL)

- RTL specifies all registers (flip-flops) and the combinational logic between the flip-flops
- Capturing the design in RTL is much faster than drawing a schematic
- Modern design depends heavily on the use of Computer-Aided Design tools:
 - ♦ To synthesize the RTL design into a schematic (or netlist)
 - ♦ To turn the schematic into a chip layout, FPGA mapping or board layout
 - ♦ To verify the original design, and verify that the more detailed designs are consistent with the original design
- Good designers depend critically on their ability to operate effectively with the CAD tools
 - ♦ Just knowing how to design logic is not enough
 - ♦ Unfortunately, you must learn a lot of tools and learn how to deal with their complexity and bugs
 - ♦ Its important to form a good understanding of the tool's methodology

ASIC Design Flow

HDL Design Capture

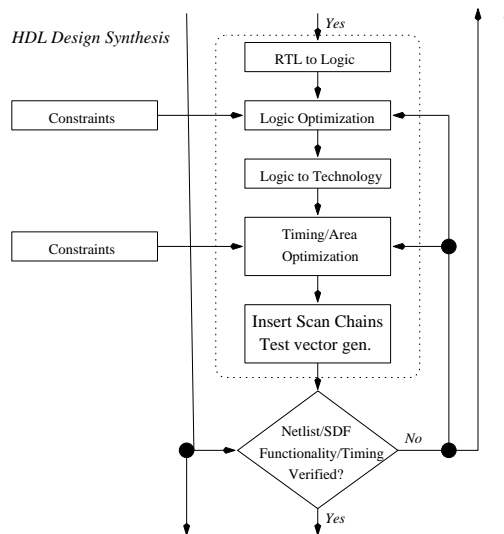


The specification should be:

- Simulatable (captures the behavior)
- Include a block diagram showing all blocks, their functions, and all wires connecting them

...ASIC Design Flow

HDL Design Synthesis



Hardware Description Languages

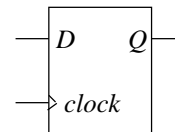
- Verilog
 - Based on C, originally Cadence proprietary, now an IEEE Standard
 - Quicker to learn, read and design in than VHDL
 - Has more tools supporting its use than VHDL
- VHDL
 - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
 - Developed by the Department of Defence, based on ADA
 - Now an IEEE Standard
 - More formal than Verilog
 - ◆ e.g. Strong typing
 - Has more features than Verilog

Verilog Model of a D-Flip Flop

```

module flipflop (D, clock, Q);
input D, clock;
output Q;
reg Q;
always@(posedge clock)
begin
  Q <= D;
end
endmodule

```



1. Module header, parameter list (=connected signals) and end
2. Declarations of parameter list
3. Local 'variables'
 - All assigned variables must be declared
4. Procedural block (or 'sequential block') forming 'main body'
 - 'main body' can consist of several procedural blocks and other statements

VHDL Model of a D-Flip Flop

```

entity flipflop is
  port (clock, D:in bit;
        Q: out bit);
end flipflop;

architecture test of flipflop is
begin
  process
  begin
    wait until clock'event and clock = `1';
    Q <= D;
  end process;
end test;

```

*Design Example Count Down Timer***Produce Specification**

- Simulatable
- Block (Module) Diagram
- All functions and wires

Step 1: Write Specification:

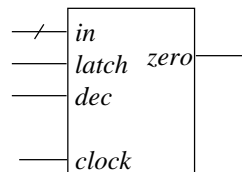
- 4-bit counter
- count value loaded from `in' on a positive clock edge when `latch' is high
- count value decremented by 1 on a positive clock edge when `dec' is high
- decrement stops at 0
- `zero' flag active high whenever count value is 0

Simulatable Specification (C):

```

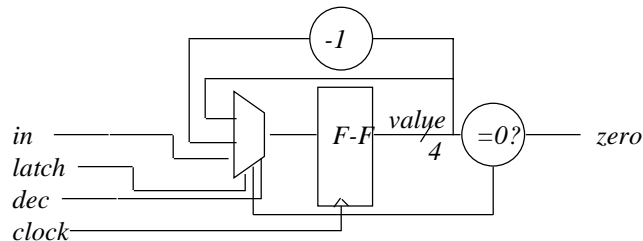
for (value=in; value>=0; value--)
  if (value==0) zero = 1
  else zero = 0;

```



*... Design Example***Step 2: Design the Hardware**

- Clearly identify
 - ♦ All registers
 - ♦ Chunks of combinational logic
 - ♦ All internal signals

ALWAYS DESIGN THE HARDWARE BEFORE CODING*... Design Example***Step 3. Write the Verilog**

```

module counter (clock, in, latch, dec, zero);
/* simple top down counter with zero flag */

input      clock; /* clock */
input [3:0] in;   /* starting count */
input      latch; /* latch 'in' when high */
input      dec;   /* decrement count when dec high */
output     zero;  /* high when count down to zero */

reg [3:0] value; /* current count value */
wire      zero;

// register 'value' and associated input logic
always@(posedge clock)
begin
    if (latch) value <= #1 in;
    else if (dec && !zero) value <= #1 value - 1'b1;
end

// combinational logic to produce 'zero' flag
assign zero = (value == 4'b0);
endmodule /* counter */

```

Features in Verilog Code

Note that it follows the hardware design, not the 'C' specification

Multibit variables:

```
reg [3:0] value;
4-bit `signal' [MSB:LSB]
```

Specifying constant values:

```
1'b1;          4'b0;
size `base value ;    size = # bits,  HERE: base = binary
                        NOTE: zero filled to left
```

Procedural Block:

```
always@(      ) Executes whenever variables in sensitivity list (  ) change value
begin        change as indicated
    ↓
end          Usually statements execute in sequence, i.e. procedurally
            begin ... end only needed if more than one statement in block
```

Design Example ... Verilog

Continuous Assignment:

`assign` is used to implement combinational logic directly

Questions

1. When is the procedural block following the `always@(posedge clock)` executed?
2. How is a comment done?
3. What does `1'b1` mean?
4. What does `reg [3:0] value;` declare?
5. What does `#1` mean and why do I use it?
6. Convert the continuous assignment statement to a procedural block.

A Basic Verilog Style

Verilog is a powerful and flexible language

- It is very easy to describe functions that do NOT map well (or synthesize into) hardware
- Its is also very easy to describe garbage that has no direct correlation to HW

Here is a suggested basic style for Verilog usage:

- For Registers and their associated input logic:

```
always@(posedge clock)
begin
    if (case) register_output1 <= #1 register_input1;
    else register_output2 <= #1 register_input2;
end
```

- For Selectors, Deselectors, etc:

```
always@(input1 or input2 or ...)
begin
    if-then-else or case statement
end
```

- For Simpler Combinational Logic:

```
assign output = inputs and operators
```

Lexical Conventions in Verilog

Logic values:

Logic Value	Description
0	Zero, low, or false
1	One, high or true
Z or z or ?	High impedance, tri-stated or floating
X or x	Unknown, uninitialized, or Don't Care

Integers:

```
1'b1;          4'b0;
```

*size 'base value ; size = # bits, HERE: base = binary
NOTE: zero filled to left*

Other bases: h = hexadecimal, d = decimal (which is the default)

```
Examples:  10                3'b1
           8'hF0              8'hF
           5'd11              2'b10
```

Procedural Blocks

Code of the type

```
always@(input1 or input2 or ...)
begin
    if-then-else or case statement
end
```

is referred to as *Procedural Code* because the statements between the `begin` and the `end` are executed *procedurally*, or in order.

- Note that all variables assigned (i.e. on the left hand side) in procedural code must be of a *register data type*. Here type `reg` is used.
 - Because a variable is of type `reg` does NOT mean it is a register or flip-flop.
- The procedural block is executed when triggered by the `always@` statement.
 - Then the statements that follow are executed once
 - The statements in parentheses (`...`) are referred to as the *sensitivity list*.
 - `always@(posedge clock)` executes whenever a positive edge on the clock occurs
 - `always@(value)` executes whenever any of the statements in parentheses change

... Procedural Code

Blocking vs. Non-Blocking Assignment

What is the difference between the following code segments?

```
initial
begin
    a = 4'h3; b = 4'h4;
end
```

```
always@(posedge clock)
begin
    c = a + b;
    d = c + a;
end;
```

```
initial
begin
    a = 4'h3; b = 4'h4;
end
```

```
always@(posedge clock)
begin
    c <= a + b;
    d <= c + a;
end;
```

- `=` is referred to as a blocking assignment, because execution of subsequent code is blocked until this statement completes. Essentially, `c = a + b;` and `d = c + a;` are performed in series.
- `<=` is referred to as non-blocking assignment. Essentially, `c <= a + b;` and `d <= c + a;` are performed in parallel

Always Use `<=` when building registers to prevent possible races.

Procedural Block Examples

What do the following code fragments synthesize to?

```
reg z;  
  
always @(x or y)  
begin  
    if (x)  
        z = y;  
end  
  
always@(w or x or y)  
begin  
    if (x)  
        z = y;  
    else  
        z = w;  
end
```

...Procedural Examples

```
always@(w or x)                always@(w)  
begin                          if (w) z = 1'b1;  
    if (x)                    else y = 1'b1;  
        z = y;  
    else  
        z = w;  
end
```

Important Points:

- What can be synthesized if you have if or case statements that do not have all cases for inputs and outputs completely covered?
- Why are complete 'sensitivity lists' (@()) important?
 - ♦ An incomplete sensitivity list is a common bug in a design.

Operators

14. VERILOG HDL 2.0 REFERENCE GUIDE 15

11.0 Operators (continued)

Language	Description
Equality Operators (Compare logic values of 0 and 1)	
<code>a == b</code>	1 if <code>a</code> is equal to <code>b</code> (1-bit True/False result)
<code>a != b</code>	1 if <code>a</code> is not equal to <code>b</code> (1-bit True/False result)
Relational Operators (Compare logic values 0, 1, X, and Z)	
<code>a < b</code>	1 if <code>a</code> is less than <code>b</code> (1-bit True/False result)
<code>a <= b</code>	1 if <code>a</code> is less than or equal to <code>b</code> (1-bit True/False result)
<code>a > b</code>	1 if <code>a</code> is greater than <code>b</code> (1-bit True/False result)
<code>a >= b</code>	1 if <code>a</code> is greater than or equal to <code>b</code> (1-bit True/False result)
Logical AND Operators	
<code>a & b</code>	1-bit result: 1 if both <code>a</code> and <code>b</code> are 1, else 0
<code>a & b & c</code>	1-bit result: 1 if all three are 1, else 0
<code>a & b & c & d</code>	1-bit result: 1 if all four are 1, else 0
Logical OR Operators	
<code>a b</code>	1-bit result: 1 if either <code>a</code> or <code>b</code> is 1, else 0
<code>a b c</code>	1-bit result: 1 if either <code>a</code> , <code>b</code> , or <code>c</code> is 1, else 0
<code>a b c d</code>	1-bit result: 1 if either <code>a</code> , <code>b</code> , <code>c</code> , or <code>d</code> is 1, else 0
Logical NOT Operator	
<code>!a</code>	1-bit result: 1 if <code>a</code> is 0, else 0
Unary Plus Operator	
<code>+a</code>	1-bit result: 1 if <code>a</code> is 0, else 0
Unary Minus Operator	
<code>-a</code>	1-bit result: 1 if <code>a</code> is 0, else 0
Unary Bitwise NOT Operator	
<code>~a</code>	1-bit result: 1 if <code>a</code> is 0, else 0
Unary Bitwise AND Operator	
<code>a & b</code>	1-bit result: 1 if both <code>a</code> and <code>b</code> are 1, else 0
Unary Bitwise OR Operator	
<code>a b</code>	1-bit result: 1 if either <code>a</code> or <code>b</code> is 1, else 0
Unary Bitwise XOR Operator	
<code>a ^ b</code>	1-bit result: 1 if either <code>a</code> or <code>b</code> is 1, but not both, else 0
Unary Bitwise XNOR Operator	
<code>a ^~ b</code>	1-bit result: 1 if either <code>a</code> or <code>b</code> is 1, but not both, else 0
Unary Bitwise NOR Operator	
<code>a ~ b</code>	1-bit result: 1 if neither <code>a</code> nor <code>b</code> is 1, else 0
Unary Bitwise NAND Operator	
<code>a ~ b</code>	1-bit result: 1 if neither <code>a</code> nor <code>b</code> is 1, else 0
Unary Bitwise NOR Operator	
<code>a ~ b</code>	1-bit result: 1 if neither <code>a</code> nor <code>b</code> is 1, else 0
Unary Bitwise NAND Operator	
<code>a ~ b</code>	1-bit result: 1 if neither <code>a</code> nor <code>b</code> is 1, else 0

Continuous Assignment

Use to build combinational logic:

Examples:

```
wire [31:0] mux_out;
tri [31:0] bus;
wire [7:0] A;
wire [31:0] B, C;
wire connect;

assign mux_out = &A ? {4{A}} : B ^ C;
assign bus = connect ? mux_out : 32'hzzzzzzzz;
```

Continuous assignment statements are executed whenever any of the operands on the right hand side change.

Continuous Assignment

Notes:

- Variables being assigned (i.e. on LHS) must be of a *Net* data type (e.g. wire, tri)
- Variables on RHS of statements can be a Net or Register data types
- Statement executes whenever anything on the RHS changes
- Tri-statable units can **ONLY** be implemented in this way
- Flip-flops and latches can **NOT** be created with continuous assignment

The easiest Verilog coding style is to use procedural code when creating flip-flops and latches and for complex logic (e.g. Finite State Machines) and use continuous assignment when specifying small pieces of logic. (Safe to mix the two types in the same module.)

- Minimizes potential for messed up sensitivity lists.

Exercise -- Use Continuous Assignment to Make an even Parity Generator:

```
wire [31:0] A;  
wire      even_parity;
```

Structural Verilog

Complex modules can be put together by 'building' (instantiating) a number of smaller modules.

e.g. Given the 1-bit adder module with module definition as follows, build a 4-bit adder with carry_in and carry_out

```
module OneBitAdder (CarryIn, In1, In2, Sum, CarryOut);
```

4-bit adder:

```
module FourBitAdder (Cin, A, B, Result, Cout);
```

```
input    Cin;  
input    [3:0] A, B;  
output   [3:0] Result;  
output   Cout;  
wire [3:1] chain;
```

```
OneBitAdder u1 (.CarryIn(Cin), .In1(A[0]), .In2(B[0]),  
               .Sum(Result[0]), .CarryOut(chain[1]));  
OneBitAdder u2 (.CarryIn(chain[1]), .In1(A[1]), .In2(B[1]),  
               .Sum(Result[1]), .CarryOut(chain[2]));  
OneBitAdder u3 (.CarryIn(chain[2]), .In1(A[2]), .In2(B[2]),  
               .Sum(Result[2]), .CarryOut(chain[3]));  
OneBitAdder u4 (Chain[3], A[3], B[3], Result[3], Cout); // in correct  
order  
endmodule
```

Structural Verilog

Features:

Four copies of the same module (OneBitAdder) are built ('instantiated') *each with a unique name* (u1, u2, u3, u4).

Module instance syntax:

All nets connecting modules must be of *wire* type (wire or tri):

```
wire [3:1] chain;
```

...Design Example

Step 3. Test Fixture must be written in order to verify correctness.

```
module test_fixture;
  reg    clock;
  reg    latch, dec;
  reg    [3:0] in;
  wire zero;
  `include ("count.v");
  initial //following block executed only once
  begin
    $shm_open("all.shm"); //to save waveform
    $shm_probe ("AS");    //save all signals
    clock = 0;
    latch = 0;
    dec = 0;
    in = 4'b0010;
    #16 latch = 1;        // wait 16 ns
    #10 latch = 0;        // wait 10 ns
    #10 dec = 1;
    #100 $shm_close();    //close data base
    $finish;              //finished with simulation
  end

  always #5 clock = ~clock; // 10ns clock

  // instantiate modules -- call this counter u1
  counter u1(.clock(clock), .in(in), .latch(latch), .dec(dec),
    .zero(zero));
endmodule /*test_fixture*/
```

Test Fixture Features

```
initial //following block executed only once
begin
...
end
```

is a procedural block that is executed *once* at the start of the simulation.

```
$shm_open("all.shm"); //to save waveform
$shm_probe ("AS"); //save all signals
...
$shm_close();
```

opens and closes the database that stores the waveforms and specifies what signals to save. (This test fixture is written as if you were using *Stand-Alone Verilog*, not *Verilog Integration* - databases managed through the GUI in the latter.

```
#10
```

are delays in terms of pre-defined units. (Here ns, but it is not always ns).

... Test Fixture Features

What does the following line of code do?

```
always #5 clock = ~clock; // 10ns clock
```

One copy of the module counter is *instantiated* in the test fixture:

```
counter u1(clock, in, latch, dec, clear, zero);
```

How would you 'build' (instantiate) a second counter?

... Example ... Simulation

Step 4. Invoke the Verilog Simulator (see tutorials) and Observe the results with Simwave

The essential difference between a HDL Simulator and compile C program is that the HDL simulator must capture the intrinsic parallelism of hardware.

Cadence Verilog-XL is an *event-based simulator*

Event Sequence in the simulation of `test_fixture` and `counter` together.....

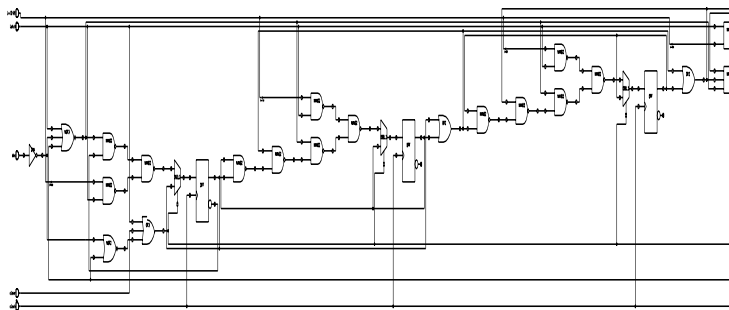
Synthesis

Step 5. After verifying correctness, the design can be synthesized to optimized logic with the Synopsys tool

Synthesis Script run in Synopsys (`test_fixture` is NOT synthesized):

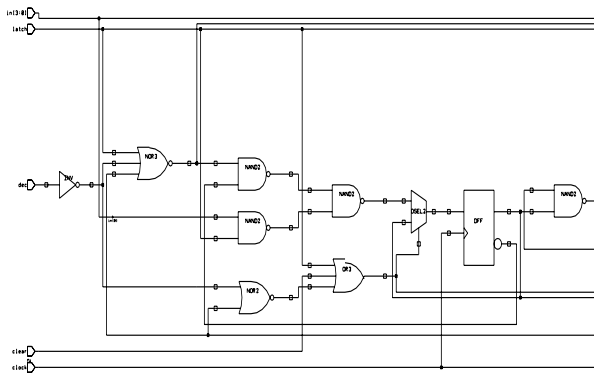
(See attached script file)

The result is a gate level design:



Detail of Design

First bit of design:



Post Synthesis Steps

- After synthesis, the design is checked for timing with a timing analyzer and correctness is checked with Verilog.
- When the design is checked, the standard cells describing the gates are placed and routed, or the FPGA is placed and routed. The result is checked and simulated again.
- Test (Scan chain) insertion and test vector generation

Verilog Use Models

1. Fully integrated with Cadence

- Design Hierarchy is captured with the Composer Schematic Capture tool.
- No need to declare module headings, ports, and ends.
- Use simwave and Verilog integration to view waveforms
- Best approach to handling large designs

2. 'Stand-alone' Verilog with Simwave

```
eos> add cadence
eos> verilog <file.v>
eos> wd &
```

- Must capture hierarchy with module instantiations. e.g. Create a module count2 that implements 2 counters with separate decrements dec1 and dec2.

3. Verilog on a PC

- Found in /ncsu/ece520_info/pc_tools/verilog
- must use \$display, etc. to view results

Sample Problem

● Accumulator:

- Design an 8-bit adder accumulator with the following properties:
- While 'accumulate' is high, adds the input, 'in1' to the current accumulated total and add the result to the contents of register with output 'accum_out'.
 - ♦ use absolute (not 2's complement) numbers
- When 'clear' is high ('accumulate' will be low) clear the contents of the register with output 'accum_out'
- The 'overflow' flag is high is the adder overflows

Summary

- *Behavioral Verilog* is based on procedural blocks starting with an `always` or `initial` statement.
 - ♦ Code executed in sequence (unless `<=` is used) whenever sensitivity list conditions are satisfied
 - ♦ assigned variables must be of register type (usually `reg`)
 - ♦ flip-flops created with `always@(posedge clock)` or `always@(negedge clock)`
 - ♦ latches created with incomplete if-then or case statements in non-edge-clocked blocks
 - take care not to inadvertently create unwanted latches.
- *Continuous assignment* creates combinational logic
- *Structural Verilog* is used to put together existing modules
- Sequence in Design
 - ♦ Understand Specification
 - ♦ Sketch the design, identifying all registers
 - ♦ Write Verilog
 - ♦ Write test fixture, and debug Verilog
 - ♦ Synthesize, verify, place-and-route, and verify