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. Verification
5. Synthesis

References
1. Quick Reference Guides
2. Smith & Franzon, Chapter 2-6

Attachments Required: Standard Synthesis Script (count.dc)
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
  - It’s important to form a good understanding of the tool’s methodology
ASIC Design Flow

Develop C/Matlab/? Model
Design Hardware
Develop Timing Diagram
Write Verilog Code

- High level simulatable design
  - Permits debugging of design intent
  - Reduces design cycle time
- Design before writing the Verilog code
  - HDLs speed up detailed gate design, not design capture
  - Design = hardware blocks + timing

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 Defense, based on ADA
  - 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;

©2004, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
Design Example .... Count Down Timer

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

Step 2: Design the Hardware
- Clearly identify
  - All registers
  - Chunks of combinational logic
  - All internal signals
- Develop timing diagram

ALWAYS DESIGN THE HARDWARE BEFORE CODING
**Design Example**

**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 <= in;
  else if (dec && !zero) value <= 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

**Procedural Block:**

```
always@(    )
begin
end
```

Executes whenever variables in sensitivity list ( ) change value

change as indicated

Usually statements execute in sequence, i.e. procedurally

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. When is ‘zero’ evaluated?
3. How is a comment done?
4. What does `1’b1` mean?
5. What does `reg [3:0] value;` declare?

Design Style

- Combine input logic to register with register

```verilog
clear  in  latch  dec  clock
always@(posedge clock)
begin
  if (latch) value = in;
  else if (dec && !zero) value = value - 1’b1;
end
assign zero = ~|value;
```

©2004, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
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 */
reg zero;

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

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

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
Summary so Far

- How do you build a flip-flop in Verilog?

- How does Verilog handle the intrinsic parallelism of hardware?

- What is a procedural block?

- What is continuous assignment?

...Design Flow

Verify Design
   Perform Pre-synthesis Simulation
   Synthesize Verilog into Hardware
Perform Post-synthesis Simulation (or Equivalency Check)
Timing Verification
Back-end Processing...

- Verification over 50% of total chip effort
  - Mainly verifying logical correctness
- Post-synthesis simulation used in this class as a “sanity check” on synthesis procedure … often skipped in practice (use logical equivalency check and timing verifier instead)
- Back end processing include “Place and Route” and final “timing verification”
**Design Example**

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

```verilog
module test_fixture;
reg clock;
reg latch, dec;
reg [3:0] in;
wire zero;
`include "count.v"
initial //following block executed only once
begin
    $dumpfile("count.vcd"); // save waveforms in this file
    $dumpvars; // saves all waveforms
    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 $finish; //finished with simulation
end
always #5 clock = ~clock; // 10ns clock
// instantiate modules -- call this counter ul
counter ul1(.clock(clock), .in(in), .latch(latch), .dec(dec), .zero(zero));
endmodule /*test_fixture*/
```

---

**Test Fixture Features**

initial //following block executed only once
begin
    //is a procedural block that is executed once at the start of the simulation.
    $dumpfile("count.vcd"); // save waveforms in this file
    $dumpvars; // saves all waveforms
end

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).
What does the following line of code do?

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

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

```verilog
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.....
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 (netlist):

<table>
<thead>
<tr>
<th>Visual</th>
<th>Textual Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND2 U37 ( .Y(n109), .A0(n105), .A1(n103) );</td>
<td>NAND2 U37 ( .Y(n109), .A0(n105), .A1(n103) );</td>
</tr>
<tr>
<td>NAND2 U38 ( .Y(n114), .A0(\value[1] ), .A1(\value[0] ) );</td>
<td>NAND2 U38 ( .Y(n114), .A0(\value[1] ), .A1(\value[0] ) );</td>
</tr>
<tr>
<td>‘n107’, etc. are nets, i.e. wires that connect the gates together.</td>
<td>‘n107’, etc. are nets, i.e. wires that connect the gates together.</td>
</tr>
</tbody>
</table>
Timing Annotation

- In this class we use SDF to annotate the synthesis outputs with timing:
- Extract from SDF file…

```plaintext
(CELL
  (CELLTYPE "DSEL2")
  (INSTANCE U49)
  (DELAY
    (ABSOLUTE
      (IOPATH D0 Y (0.896:0.919:0.919) (0.806:0.820:0.820))
      (IOPATH D1 Y (0.886:0.886:0.886) (0.791:0.791:0.791))
      (COND D1 == 1'b1 & D0 == 1'b0 (IOPATH S0 Y (1.195:1.195:1.195) (1.219:1.219:1.219)))
      (COND D1 == 1'b0 & D0 == 1'b1 (IOPATH S0 Y (1.224:1.224:1.224) (1.080:1.080:1.080))
      (IOPATH S0 Y (1.224:1.255:1.255) (1.064:1.080:1.080))
    )
  )
)
```

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

  Hint:
  8-bit adder produces a 9-bit result:
  \[ \{\text{carry\_out, sum}\} = A + B; \]

Sketch Design

1. Determine and name registers.
2. Determine combinational logic
Code Verilog

module accum (clock, accumulate, clear, in1, accum_out, overflow);
input clock, accumulate, clear;
input [7:0] in1;
output [7:0] accum_out;
output overflow;
reg [7:0] accum_out;
wire [7:0] accum_in;
wire overflow;
always@(posedge clock)
begin
if (clear) accum_out <= 8'b0;
if (accumulate) accum_out <= accum_in;
delay
assign {overflow, accum_in} = accum_out + in1;
endmodule /* counter */

Summary

- What are our two “matras” so far?

- What is built for all assignments after always@(posedge clock)?

- What is built after always@(A or B)

- What is built with assign C =
Summary

- In Synthesis with Synopsys
  - What is important after the “read” statement?
  - Which timing library do we use for the first compile?
  - What does “compile” do?
  - What is important to do after every incremental compile?