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;
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;
```

**Produce Specification**
- Simulatable
- Block (Module) Diagram
- All functions and wires
**Design Example**

**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

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 <= 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:

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

Specifying constant values:

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

Procedural Block:

```verilog
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. 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
classic
always@ (posedge clock)
begin
    if (latch) value = in;
    else if (dec && !zero) value = value - 1'b1;
end
assign zero = ~|value;
```
... Alternative Coding

Step 3. Write the Verilog

```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 */
```

©2004, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
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
- It 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**:
  ```verilog
  always@(posedge clock)
  begin
    if (case) register_output1 <= #1 register_input1;
    else register_output2 <= #1 register_input2;
  end
  ```

- **For Selectors, Deselectors, etc**:
  ```verilog
  always@(input1 or input2 or ...)
  begin
    if-then-else or case statement
  end
  ```

- **For Simpler Combinational Logic**:
  ```verilog
  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?
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”
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
$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 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 \textit{once} at the start of the simulation.

```
$dumpfile("count.vcd"); // save waveforms in this file
$dumpvars; // saves all waveforms
```

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 \textit{Stand-Alone Verilog}, not \textit{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.....
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):

Visual:

Textual Form:

`n107`, etc. are nets, i.e. wires that connect the gates together.
Netlist:

```Verilog
module counter (clock, in, latch, dec, zero);
  input [3:0] in;
  input clock, latch, dec;
  output zero;
  wire [3:0] value [3], value [2], value [1], value [0],
  [5:0] n54 [0], n54 [1], n54 [2], n54 [3],
  n103, n104, n105, n106, n107, n108, n109, n110, n111, n112, n113, n114, n115;
NOR2 U36 (.Y(n107), .A0(n109), .A1(value [2]));
NAND2 U39 (.Y(n115), .A0(value [3]), .A1(value [2]));
NOR2 U38 (.Y(n114), .A0(value [1]), .A1(value [0]));
NAND2 U37 (.Y(n113), .A0(value [2]), .A1(value [1]));
NOR2 U35 (.Y(n108), .A0(value [0]), .A1(value [3]));
INV U43 (.Y(n106), .A(n106));
INV U45 (.Y(n108), .A(n108));
AOI21 U44 (.Y(n106), .A0(n112), .A1(dec), .B0(latch));
NAND2 U46 (.Y(n112), .A0(n115), .A1(n108));
DSEL2 U48 (.Y(value [3]), .D0(n111), .D1(in [3]), .S0(latch));
DSEL2 U49 (.Y(value [2]), .D0(n110), .D1(in [2]), .S0(latch));
DSEL2 U50 (.Y(value [1]), .D0(n113), .D1(in [1]), .S0(latch));
DSEL2 U51 (.Y(value [0]), .D0(n105), .D1(in [0]), .S0(latch));
EDFF \value_reg [3] ( .Q(value [3]), .QBAR(n104), .CP(clock), .D(n54 [3]), .E(n54 [0]));
EDFF \value_reg [2] ( .Q(value [2]), .QBAR(n103), .CP(clock), .D(n54 [2]), .E(n54 [0]));
EDFF \value_reg [1] ( .Q(value [1]), .QBAR(n102), .CP(clock), .D(n54 [1]), .E(n54 [0]));
EDFF \value_reg [0] ( .Q(value [0]), .QBAR(n101), .CP(clock), .D(n54 [0]), .E(n54 [0]));
endmodule
```
Timing Annotation

- In this class we use SDF to annotate the synthesis outputs with timing:
- Extract from SDF file....
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:
  \[
  \{\text{carry\_out}, \text{sum}\} = A + B;
  \]

Module accumulator (accumulate, clear, in1, accum_out, overflow);

input accumulate, clear;
input [7:0] in1;
output [7:0] accum_out;
output overflow;

reg [7:0] accum_out;
wire [7:0] accum_in;
wire carry, overflow, enable;

Fourbitadder u1 (1’b0, in1[3:0], accum_out[3:0], result[3:0], carry);
Fourbitadder u1 (carry, in1[7:4], accum_out[7:4], result[3:0], overflow);

assign enable = accumulate || clear;
assign accum_in = clear ? 8’b0 : result;

always@(posedge clock)
  if (enable) accum_out <= accum_in;
endmodule
Sketch Design

1. Determine and name Registers.
2. Determine combinational logic
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;
end
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?