

Verilog Coding Style

We will talk about Verilog coding style

- Issues for EE272
- Issues for larger projects
 - Complexity management very important
 - Many projects in industry use flops
 - Talk about flop rules too

What is Verilog

(I assume that everyone has seen Verilog before in EE271)

There is an excellent reference manual in EE271 notes

- Refer to it if you have syntax questions

Verilog is really two things:

- Hardware description language
- Simulator
 - Program used to do function validation
 - Need to know how to control simulation
 - Talk more about running the simulator on Thurs

Why Write A Verilog Model?

It is required

Simulation

- Determine if your part does the right thing
 - Do you understand the problem you are trying to solve
 - Did you implement it correctly

Synthesis

- Input to synopsys
- Pseudo logic design

Conflict

- For simulation you want to do something quick to see if it works
 - For synthesis, you need to think about implementation to make sure the partitioning and functional design will lead to good implementations
 - There is NEVER enough time to write it twice
-

Real World Constraints

(Heard this before)

Need to make your best guess when you start

- Try to estimate what hardware and wires you will need
 - Do this while you are figuring out what to build.
 - Makes the initial going a little more difficult
- Then you get to iterate only the sections where your guess was not good
- Also need to build the structure needed to test your design

In 272 you will have

- Testing structure
 - Control (synthesis)
 - One or more datapaths
-

Levels of Description in Verilog

Verilog provides you the ability to describe functions in different levels of detail

How much detail

- Enough to let you sleep at night
- Too much detail will get in the way (over constrain problem)

Structural Level

- Specify the cells that you will use (too low a level)

Synthesis subset

- What is used in EE271. Some structure, but use always and assign
- Datapath and control of the chip should use this level

Process

- More like a parallel program view -- communicating processes
- No so good for hardware, but great for writing the test infrastructure you need

Levels of Abstraction--Example

Structural code. Explicitly describe what are the contents of a module. The contents can be other modules, a set of primitive supported by verilog, user defined primitives (UDP), or transistors. We will only go down to the gate level. For example,

```
module FIFO (Phil, In_s1, Out_s1);
    input  Phil;
    input  [15:0] In_s1;
    output [15:0] Out_s1;
    wire   [15:0] in_s2, in1_s1, in1_s2;

    latch_16 Latch1(in_s2, In_s1, Phil);
    latch_16 Latch2(in1_s1, in_s2, Phi2);
    latch_16 Latch3(in1_s2, in1_s2, Phil);
    latch_16 Latch4(Out_s1, in1_s2, Phi2);
endmodule
```

Levels of Abstraction, cont.

Synthesis subset.

- Describe how my design behaves, but not exactly how it will be implemented.
This is a good way to describe cells and control blocks

```
module adder_16(Phil, InA_s1, InB_s1, Out_s2);
    input  Phil;
    input  [15:0] InA_s1, InB_s1;
    output [15:0] Out_s2;
    reg    [15:0] out_s2;
    wire   [15:0] out_v1;

    assign #20 out_v1 = InA_s1 + InB_s1;
    assign #1 Out_s2 = out_s2;
    always @(Phil or out_v1)
        begin
            if (Phil == 1'b1)
                out_s2 = out_v1;
        end
endmodule
```

Mixed Descriptions

Can Mix both forms of verilog. Adding an additional latch to the adder

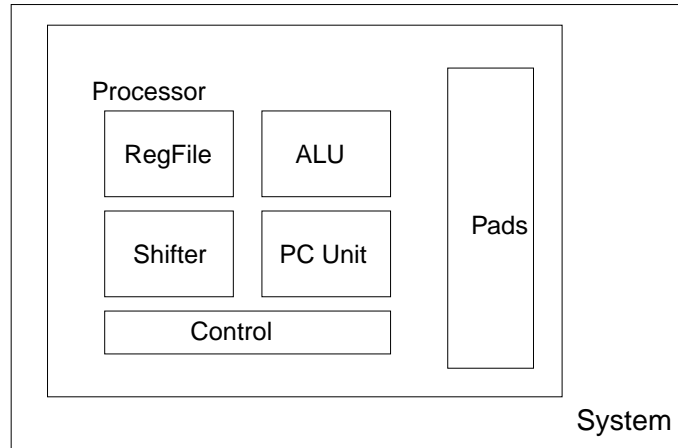
```
module adder_16(Phil, Phi2, InA_s1, InB_s1, Out_s1);
    input  Phil, Phi2;
    input  [15:0] InA_s1, InB_s1;
    output [15:0] Out_s1;
    reg    [15:0] out_s2;
    wire   [15:0] out_v1;

    assign #20 out_v1 = InA_s1 + InB_s1;
    always @(Phil or out_v1)
        begin
            if (Phil == 1'b1)
                out_s2 = out_v1;
        end
    latch_16 OutLatch(Out_s1, out_s2, Phi2);
endmodule
```

Level of Hierarchy

Divide and conquer approach -- completely orthogonal to levels of abstraction

Example for a 272 type project



Can use different levels of abstraction in the different blocks

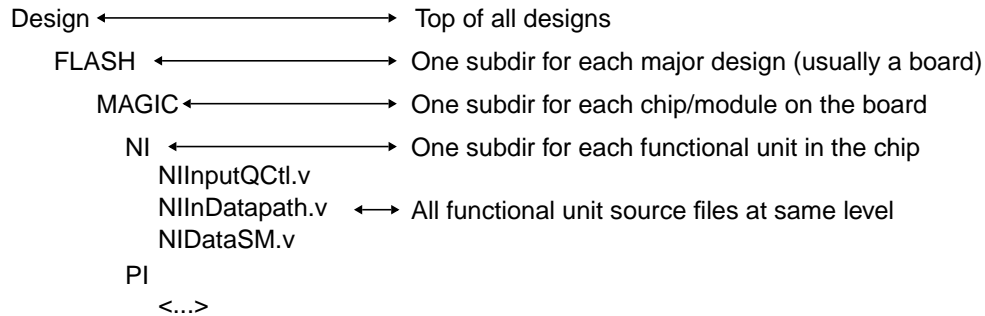
Hierarchical Design, cont.

Some things to keep in mind.

- Partition according to physical design. Keep together stuff that will be laid-out together. Also keep together logic that is closely related, or has many wires connecting them
- Use hierarchy to hide implementation details. Find objects for which I do not want to give a structural description and encapsulate them in a level of hierarchy. For example to describe an ALU,

Verilog Hierarchy

In larger projects need to prevent deep design trees



Ensure that no module names conflicts can occur

- Source file name == module name
- Prefix source file/module with name of unit (e.g. NIInputQCtrl.v)

Coding Guidelines

Use lots of comments

```
//-----  
// ModuleName.v with revision information (from CVS)  
// Hierarchy: ParentModuleName.v  
// Module Function:  
//     What the module does, and tricky issues that one needs  
//     to watch for  
// Revision History:  
//     Time, Author and explanation of changes  
//-----
```

Use a source control system (RCS, SCCS, CVS)

Use macros to remind you of what is going on:

- Bit-widths, bit-positions, array sizes, delays, etc.

```
'define RD_FIELD      15:11  
assign #1 DestSpec_s2 = Instr_s2[ 'RD_FIELD];
```

Naming Conventions

All signal names consist of a name, an optional polarity, and a mandatory timing type

- Signal name
 - Global (inter-unit) signals begin with a capital letter
 - Local (intra-unit) signals begin with a lowercase letter
 - capEachNewWor
- Polarity indicator
 - If none, signal is asserted iff == 1 (high-true)
 - If “_b”, signal is asserted iff == 0 (low-true)

Propagate names through hierarchy

Use the same name for instance, module, and file

Easier to find module declaration

Two Phase Timing Types

Use timing types

- signalname_s1Stable 1
- signalname_v1Valid 1
- signalname_s2Stable 2
- signalname_v2Valid 2
- signalname_q1qualified clock

All signals end with a timing type:

`_[s,v,q,w][1,2][a-z]`

Active low signals are signal_b_s1e

Verilog Timing

Don't bother modeling timing at RTL level

- You'll never get it correct
- You can (will) backannotate it anyway
- It slows down simulation speed (for vcs, at least)

In 2 phase designs

- Don't need to worry about timing at all
- Clock generator should be only system with delay information
- Because sequence is set by clock edges

In single clock designs, you need to be more careful

- A small clock-to-Q delay will solve most simulation problems

Timing in FLASH

FLASH conventions

- Combinational logic (including memory) evaluates in zero time
- Only sequential elements have delay
- Flops/latches have a delay of one 'TICK (global 'define)
- Clock cycle time === 100 time units; 'define TICK #5

Examples

- `assign bar_v = sigA_v | (sigB_s & sigC_v);`
- `always @(posedge Phi1) foo_s <= 'TICK bar_v;`
- `always @(posedge Phi1) // more conservative`
`foo_s <= 'TICK ((Phi1 === 1) ? bar_v : 1'bx)`

Synthesis Issues

Code for synthesis from the start

- Avoid unsynthesizable constructs (implied memories, variable shifts, dividers, etc.)
- Separate combinational logic from flops/latches
- Avoid using verilog functions/tasks (not sure how well Synopsys handles these)

Keep implementation technology in mind (stats for LSI LCB500K)

- Raw nominal gate speed (180ps for FO4 inverter)
- Worst-case derating factor (1.6 for WCPVT at 85C)
- Clock skew (500ps for balanced clock tree)
- Flop overhead (WC setup: 0.75ns; WC clk-to-q: 1.3ns)
- Not much time for logic $(10 - 2.1) / 1.6 = 4.9\text{ns}$ nominal for logic!

Synthesis

- Plan to let Synopsys deal with control
- Plan to deal differently with datapaths. Often use vendor datapath tools.
- Plan to hand-tweak critical control paths at the source level
- Sometimes better logic results from higher abstraction level in RTL description, sometimes not. Will need to experiment.
- Plan to deal with testability from the start of synthesis (scan insertion, additional delay of scannable flops, etc.)

Simulation Issues

Sequential program modeling parallel hardware

- Careful about the execution order

```
// This model of a pipeline will never work
always @(posedge Phil) begin
    stageA = input;
    stageB = stageA;
    stageC = stageB;
end

// This will
always @(posedge Phil)
    stageA <= #'TICK input
always @(posedge Phil)
    stageB <= #'TICK stageA
always @(posedge Phil)
    stageC <= #'TICK stageB
```

Simple Do and Don't

Don't use

- More than one clock in always block.
- Embedded constraints and attributes.
- //synopsys translate_on/off (debug code).
- //synopsys full_case. Use default: instead.

Do use

- Sizes of constants (4'bf except for X's i.e. 4'bxxxx).
- Explicit clock qualification in datapath.
- //synopsys parallel_case
- Lots of comments -- it will help you remember what this logic is really supposed to do.

Mixed Signal Designs

You can never reach the same modeling fidelity as in digital designs¹. Your main goals should be:

- Verify the digital blocks
- Ensure that the interactions between analog and digital blocks are predictable

Divide and conquer:

- Define a clear boundary between Analog and Digital blocks and accurately model the digital blocks following all the rules outlined before.
 - Even if the digital portion of your chip is very simple model it anyway
- Write a very simple model of what your analog block is supposed to do
 - Be very accurate w.r.t. digital signals into analog blocks
 - Be very careful with outputs of analog blocks into digital blocks

If you have such signals talk to us ASAP.

¹ An extended version of verilog (verilog-A) promises to solve this problem. Not available in this class so we will try to survive the old-fashioned way

Modeling Analog Blocks in verilog

Three main tools:

- *real* variables
- *time* variables
- PLI code (C-code linked into the simulator). Not in EE272.

Example. D/A converter:

```
'define NBITS 8
'define LEVELS 2^8
'define VMAX 3
module DAC(outv, in)
output outv;
input ['NBITS-1:0] in;

real outv;

always @(in) begin
    outv = in / 'LEVELS * 'VMAX;
end

endmodule
```

Modeling analog blocks in verilog (cont'd)

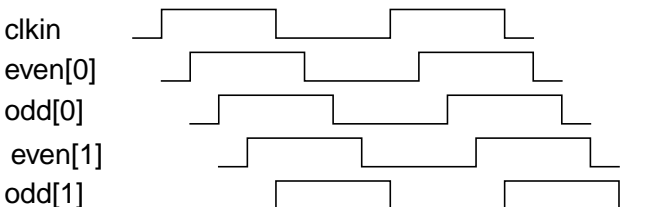
Situation is easier with analog timing blocks (e.g. PLL's, DLL's. Example: Four stage delay line generating multiphase overlapping clocks

```
'define TCYCLE 100
moduleDL(even, odd, clkkin, enable);
output [1:0] even, odd;
input clkkin, enable;

parameter stageDly = 'TCYCLE/8;

assign #stageDly even[0] = clkkin & enable;
assign #stageDly odd[0]  = even[0] & enable;
assign #stageDly even[1] = odd[0] & enable;
assign #stageDly odd[1]  = even[1] & enable;

endmodule
```



Verilog Verification Tools

In order to verify that the circuits you designed are working (as intended), we will use Verilog to generate IRSIM vectors. This will be done using snoopgen and snooper modules.

In order to “snoop” correctly, the following are necessary:

- Verilog accurately describes what you want your circuit to do
- Nets you want to verify exist in the Verilog description
- Direction of bidirectional nodes are specified by a control_signal

Some additional tips to facilitate verification:

- Append timing types for all signal names
- Use the same name for Verilog, schematics and layout
- Use bracketed numbers for buses (e.g. bus_v1[3], ..., bus_v1[0])

Why Snoop?

The first phase of the design process is to write a description of the chip (and overall system) in Verilog. The chips that we are designing are simply a translation of this Verilog code to circuit layout form. So, we can use Verilog to generate the test vectors to make sure that the layout operates correctly and matches the functionality of the Verilog. This saves us a lot of time since we don't have to generate all the vectors by hand.

How to Snoop

There are a few simple steps that you need to follow:

- Use snoopgen to create a snooper module
- Instantiate the snooper module in your Verilog
- Run Verilog (with -x option or rsimverilog)
- Run IRSIM on the .sim file of circuit(s) you wish to verify
- Set up the analyzer window, Vdd and Gnd
- Run "versim.cmd" (generated by Verilog)
- Check to see if there were any assertion failures

Snoopgen

Snoopgen is a tool that takes a list of node names specified as inputs, outputs or binputs and generates a module called snoopgen. The snoopgen module calls on PLI functions such as "\$rsim_log_input" and "\$rsim_log_output" to generate a file (versim.cmd) with a list of commands to toggle inputs and to check outputs.

An example of an input file for snoopgen is available on the website:

ee272 website -> Documentation -> Snoopgen Documentation

- Remember to use timing types with signal names since snoopgen will use this to figure out when to check outputs (or a segmentation fault will result)
- To run snoopgen:

```
snoopgen system.in >! snoopgen.v
```

Snooper Module

Here are a couple of hints on how and where to use the snooper module:

- Use multiple snooper modules during the design process
- Once you have completed a major block, use Verilog and a snooper module to generate test vectors for that block
 - Instantiate a snooper module in the Verilog module which calls that block
 - Use hierarchy to snoop into internal nodes

```
system.ALU.in_A_s1[7:0]
```

Vcheck

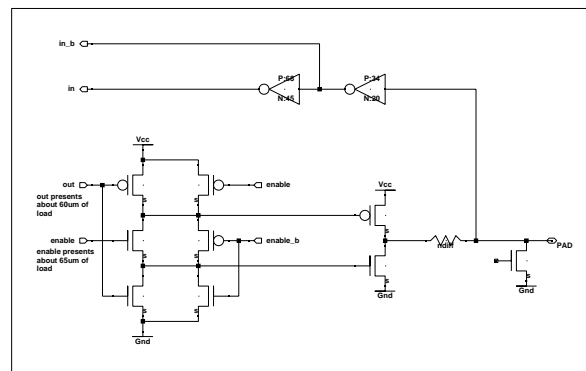
What can we say? Use it! It's good for you.

In order to run vcheck on your Verilog code, you need to preprocess it. There are a couple of ways that you can do this, but an easy way is:

```
vpp code.v | vcheck
```

Look at the messages that are generated and fix the Verilog or convince yourself that the code is okay.

Pad info



Remember to tristate the “in” signal if you’re driving data off-chip.

The “in” signal is strong enough to drive cross-chip (fanout of 16).

The “out” and “enable” signals present a relatively light load.