Verilog Tips, Pitfalls to Avoid

By marshallh 011315

Get into the Right Mindset

A. You are writing HDL, not C. Although Verilog may look vaguely similar in its syntax, that's where it ends. Experienced software monkeys may take more mental re-adjustment to write HDL than a newbie with no prior software experience.
B. CPUs execute instructions in sequence. HDL is describing everything and it is all happening at the same time.
C. Some of the same concepts used for safe multithreading are common to hardware, such as semaphores, race conditions, mutual exclusion, etc. Not all though.
D. Describe circuits, not programs.

What About Google?

A. Just like any other language, there is lots of bad code out there.
B. You'll see lots of course materials, that's fine, do remember that academic examples may not be the best or most reliable way of doing things, even though they may be elegant.
C. Fpga4Fun, UCSC, ArchVLSI, Peter's Intro

Testbenches

A. Approximately 80% of the Verilog language is not even synthesizable; that is, during compilation it will be either entirely ignored or error out depending on the design suite. Generally, such commands are labeled: **`$display`**, **`$monitor`**, etc
B. Although you won't be shipping anything with these constructs and commands, they will aid you if you choose to test your code via simulation. A common example would be if you're writing a SDRAM controller – if you violate timing as per the datasheet the chip is not going to tell you explicitly. However, by designing a testbench and instead of interacting with the physical chip, you plug it into Micron's behavioral model, it will log invalid commands, timing violations, and so on.
C. A benefit of simulation is that only the first step of compilation – Analysis & Synthesis – is required. Because fitting, timing analysis and bitstream generation are skipped, time saved can be over 50%.
D. Eventually your HDL needs to talk to the outside world, interfacing with other chips. When you can't fully model the entire system, add SignalTap to the design and watch

how the system reacts with your coded loaded into the FPGA. I tend to use this more than simulation.

Writing Synthesizable Code

A. You are only going to be using less than 10% of the Verilog language.
B. Stick with purely synchronous logic, and only trigger on rising edges.
C. Size and length of the code can be entirely unrelated to amount of device usage.
D. With Verilog there are options to write C-like code and offload more of the actual implementation onto the tools; this is a good way to waste logic resources.
E. Your F_{max} of a particular clock domain usually is brought down first by large amounts of combinational logic. Pipeline or break up the logic between several cycles.
F. Do not use tristate primitives or high impedance values (`1'bZ`) within modules, only in the top level. This is a holdover from the ASIC design and although most FPGA tools can work around it, you may be in for some nasty surprises. Verify this works as intended in your own application, and put a note for future reference if the design may be reused in a different environment later.
G. Identical RTL can be interpreted different ways by different vendor software if there is any ambiguity. In addition, test results can show different behavior between simulation and the physical implementation. Sunburst Design paper about this.

Clock Domains

A. As before, you should be writing largely synchronous logic, clocked off an either internal or external clock source (such as an internal PLL output, dedicated clock input, etc). Some combinational logic is fine too.
B. Never clock logic using other logic. The classical example of using a counter's MSB as a clock divider comes to mind. This will explode in your face if you try to apply it to anything more complex.
C. Any input that isn't edge-synchronous with your current module's clock should be treated as completely asynchronous. Whether it's an external output enable or data bus, you must synchronize the signal(s) with respect to the current clock or risk catching it between logic levels. This is explained down below.
D. Avoid having more than one clock per module. This is more a stylistic approach but it helps partition your code and avoid mixing registers up between clocks.

Avoiding Metastability

A. Any asynchronous signals or those coming from another clock domain *must* be synchronized with your local clocked logic.

BAD

```
input clk_50;
input ext_enable;

always @(posedge clk_50) begin
      if(ext_enable) …
end
```

A BETTER APPROACH

```
input clk_50;
input ext_enable;
reg ext_enable_1, ext_enable_2;

always @(posedge clk_50) begin
      ext_enable_1 <= ext_enable;
      ext_enable_2 <= ext_enable_1;

      if(ext_enable_2) …
end
```

There is no "one right way" to do this. The industry rule of thumb is to run each signal through 2 flip-flops to allow the signal time to settle and prevent glitching. For applications where even higher reliability is required, 3 flip-flops increases the MTBF exponentially.

B. Another cause of internal glitching is the classic asynchronous reset. This is another thing I've had to learn the hard way. Just because something may work with a blinking LED or in simulation does not guarantee it will work down the road in a more complex application.

BAD

```
input clk_50;
input rst_n;

always @(posedge clk_50 or negedge
rst_n) begin

      if(~rst_n) …
end
```

A BETTER APPROACH

```
input clk_50;
input rst_n; // must be synch

always @(posedge clk_50) begin
      if(~rst_n) …
      else begin
            …
      end
end
```

Again, in a design with multiple clock domains, you will need to synchronize the reset signal locally to that particular domain. More information in this comprehensive Sunburst Design paper.

C. You can adapt the synchronizers to use them as edge detection – **`if(~sig_latch_2 & sig_latch_3)`** triggers on the falling edge of **`sig_latch`**, but there is a 2 cycle delay. This of course assumes you have explicitly added your own synchronizers.
D. For passing data between clock domains, a true dual-ported block RAM can be very convenient, along with a couple semaphores, or handshaking synchronizer.

Timing Analysis

A. Modern design tools by default use preliminary timing information to determine how to synthesize and fit your design. In Quartus II, this is called "Timing-driven synthesis". From version 10 onwards, timing-driven synthesis is enabled by default, and you must at minimum constrain your external clock oscillator.
B. Quartus has a tool called "TimeQuest" that analyzes the final compiled design to see that all logic meets timing.
C. At some point your code will be of sufficient complexity that even having specified no constraints, you'll see "Timing requirements not met" as a critical warning. It's time to tell Quartus about the external clocks you're using and any specific constraints you want it to use when synthesizing your logic.
D. SDC files (short for Synopsys Design Constraints) are used to describe clocking and any I/O that needs explicit setup/hold timing. These files are written in the Tcl language.
E. Adding proper timing constraints is a process in itself, here's a stripped down file that has the minimum:

```
create_clock -name clk_50 -period 20.0 [get_ports clk_50]
derive_pll_clocks
derive_clock_uncertainty
set_clock_groups -asynchronous \
-group { clk_50 } \
-group { clk_usb } \
-group { \
      mibp|altpll_component|auto_generated|pll1|clk[0] \
} \
-group { \
      mp0|altpll_component|auto_generated|pll1|clk[0] \
}
```

First, the external 50mhz clock with a 20ns period is created, and it is used for deriving the PLL clocks that may be based off this source clock.

Then, explicit clock groups are declared and false paths are cut between the domains. The long verbose names describe a particular PLL output.

For high speed I/O like an external DDR data bus where specific setup and hold times must be observed, these should be detailed in your constraints file as well. Pay attention to the synthesizer warnings – if there are any errors or missing/extraneous nodes in the SDC, the entire constraint may be ignored.

Compiler Directives

A. Besides plain includes (`` `include 'header.v' ``) it's best to leave these alone.
B. Depending on your Analysis/Synthesis settings (Quartus default is 'Safe State Machine') your FSM registers (**`reg [4:0] state`** for a 32-state FSM) may be converted to Gray coding or one-hot
C. These directives are a holdover from Synopsys tools which were/are industry standard. Synplify is the backend behind several vendors' tools, and also used standalone.
D. **`/* synthesis preserve */`** is applicable to a register which prevents state machine inferring; the FSM still works as intended but you will be able to tap the register with SignalTap. Logic usage will decrease once you remove the directive and allow full optimization.
E. **`/* synthesis noprune */`** prevents Quartus from optimizing away zero-fanout registers that aren't directly connected to outputs or combinational logic.
F. **`/* synthesis keep */`** prevents combinational nets from being optimized out. The result is less efficient logic utilization.
G. Example: **`reg [7:0] state /* synthesis preserve */ ;`**
H. Once a module is debugged you should remove these directives.
I. Additional instructions to the synthesis engine are possible via vendor-specific configuration. In Quartus, you can set behavior of the synthesizer with respect to all manner of things such as register duplication, area optimization and so on via the Assignments > Settings > More Settings dialog box.

General

A. Do not mix blocking and nonblocking assignments (**=**, **<=**) within the same always block. I use nonblocking only.
B. Continuous assignments: Expression on the left of the assignment (**=**) must always be a net or structural element.

C. Caveat: Inside clockless always blocks such as **`always @(*)`**, all assignments (**`<=`**) are continuous, and those "registers" you are assigning turn into wires. This is confusing as the same code but inside a clocked block becomes synchronous with proper registers.
D. Procedural assignments: Expression on the left of the assignment (**`<=`**, **`=`**) must always be a register. These are placed inside **`always`** blocks.
E. Assignment precedence: You can have multiple assignments to the same register within one **`always`** block. The last occurring assignment in the same clocked **`always`** block will take precedence over the others. You can use/abuse this for cleaner code, exercise caution.
F. Don't have two **`always`** blocks assigning values to the same register. If you're lucky your tool will catch it, but in any case you should re-think your design. Designs may have a giant always block, or contain many blocks with separate, non-overlapping functionality. Either one is correct.
G. Case statements inside clockless (asynchronous) **`always`** blocks must have a default case. Put another way, every signal must have an assignment in all possible control flows, or latches are inferred.
H. Mixing logical and bitwise operators: Don't. Within the same **`if()`** condition, stick with one or the other.
I. When you have an arithmetic operation (addition, subtraction, multiplication, etc) done to two registers or structural nets, each must have been explicitly declared 'signed' or by default the operation will be unsigned. This extends to explicitly instantiated DSP or multiplier blocks. Each operand may be casted **`$signed()`** for the operation if it wasn't explicitly defined as such.
J. Numbers without a specified bit width (e.g. **`12'h6F`**) will be implicitly treated as 32 bits.
K. Bit sequence repetition: **`{2{16'hBEEF}};`** yields **`32'hBEEFBEEF;`**
L. You can put spacers in number definitions for clarity – **`8'b100_10001;`**
M. If you want sign extension to work in your arithmetic when using constants, always specify the full number of bits in the constant.

This guide isn't at all intended to be comprehensive or the best out there, it's simply what's worked for me and if you have suggestions or would like to point something out, email me:

mail@retroactive.be