Hardware Design Guidelines

Rajesh Bawankule (rajesh52@hotmail.com)

Coding Style Guidelines

Editor Guidelines

- A single line should not contain more than 80 characters.
- Avoid tabs, since they will not indent the same in all editors. If you must use tabs, re-format your source before checking it in using `expand -4' on Unix.
- Use 4 hard spaces for indentation.

Source and Revision Control:

- Use a source control header at the top of all source files (i.e. verilog source, synthesis scripts, makefiles, include files, run scripts).
- Keep all these files under revision/version control.
- At every check-in add comments which state the purpose for the change(s).

Hierarchy Guidelines.

Each ASIC / FPGA should have a top level module, a pad module and a core module

- the top level module should be named `<ASIC>.v`
- the pad module should be named `<ASIC>_pad.v`
- the core module should be named `<ASIC>_core.v`

where <ASIC> is the top level name of your ASIC / FPGA.

This describes the directory hierarchy standards we want the designers to follow in order to ease system integration/FPGA-ASIC conversion.

Module and File Naming Conventions:

- Write only one module per file.
- The filename should be `<module_name>.v`
- The top level module names need to be ALL CAPS.
- A bus functional model should have name `<model_bfm.v>`. The module name should be the same as the model to be replaced.

example: The bus model of the pentium processor should be called `pentium_bfm.v`

- Verilog control files use .vc suffix.
- Synchronizer module names should use "sync_" prefix.

Module Organization

- Suggested order for parts of a module. This should ease debugging. Port names should be one per line to ease automatic parsing.

  - o Module header (includes copyright, RCS information, description, modification history)
  - o Input ports, place clocks at the beginning.
  - o Output ports
  - o InOut ports
  - o Declaration of internal signals
  - o Local parameters, including state parameters (state parameters can also be placed just before the state machine)
  - o Continuous assignment statements
  - o Control/Next state logic (use function or always blocks)
  - o Control/State registers

- Short description of requirements / specifications / IO description should be provided inside the module.
- Simple timing diagrams should be provided wherever possible.

Parameters and `defines:

- Avoid using long explicit path identifiers.
- Place all system or chip-wide macros (`define statements) and parameters in an include file named `<chip>.h or <system>.h.`

Inter-module Connections

- Always use explicit signal names for inter-module connections. DO NOT use positional or implied association. For example: Use

```
ModA modA_inst (.sys_clk(clock), .in1(start),...);
```

- Do not concatenate pin lists to vectors in instantiations:

```
Don't:  modA modA_inst(.bus({ad[0], ad[2], ad[4]}), ...);
Do:     assign evenadd = {ad[0], ad[2], ad[4]};
          modA modA_inst(.bus(evenadd), ...);
```

Signal Naming Conventions:

- Always maintain clock signal names across levels of hierarchy.
- Whenever possible, maintain non-clock signal names the same at all levels of hierarchy.
- Top-level pin signal names must not be vectored (i.e. use `BUS_0,BUS_1...,not BUS[0:n]).`
- Top-level pin signal names need to be ALL CAPS.
- Top-level pin signal names should contain suffix _I, _O, or _BI to indicate direction of pins.
- Do not use "_" as a prefix or suffix. This may cause problems when going through vendor-specific tools.
- Use "_" between words in a signal name.

example: `byte_sel, addr_bus_en`

- Use "_" and not capital letters to separate items in a port name :

example: `my_clever_bus` instead of `myCleverBus.`

- Signal names should be case insensitive.

example : Do not use `mybus` and `MyBus` to designate 2 different object.

- Use underscores to separate bit numbers from port name .

example: `my_bus_12` instead of `my_bus12.`

- Use "_L" or "_l" for active-low signals.

example: `byte_sel_l, addr_bus_en_l`

- Use common suffixes (_en, _set, _rst, _rdy, etc.) for control signals like

enable, set, reset, ready, etc.

- Use "_clk" suffix for flip flop clocks. This will aid in finding these clocks for test purposes.
- If using latches, use "_g" suffix for latch clocks.
- When synchronizing asynchronous signals, use "u_" prefix on the unsynchronized signal.
- Use ALL CAPS for parameter's and `defines.
- Be consistent when you use abbreviations. The following show the preferred Abbreviations

preferred	instead of	meaning
DATA	DA	Data Bus
ADDR	ADD, AD	Address Bus
CLK	CLOCK,CK,CLOK, KLOK	All top level clocks
RESET	RST	reset
VALID	VLD	valid pin

State Machines

- Use parameter declarations for the value of all states.
- Meaningful state names should be used in parameter declarations instead of s0, s1, s2.
- State variable names should have an "_st" suffix.
- Place "nx" prefix on all next state variable names.
- Every state should preferably have a small description section in the beginning to indicate the action in that state.

Design For Test Guidelines

Clock Issues

- Use negative edge clocks and flip-flops only where absolutely necessary (e.g. on the write enable of SRAMs).
- Do not use a clock in the data path to a flip flop which it controls.
- Drive clocks directly from primary input pins whenever possible.
- For internally generated clock signals, the clock generation flip-flops must not be in the scan path, and there must be a path making the clock controllable from primary inputs.
- Generally each clock domain will require a separate scan chain. It is possible, but not preferable, to make a single chain from multiple clock domains.
- Internally-generated clocks need to be controllable from primary inputs.
- All frequency dividers must be resetable from primary inputs.
- In multi-clock designs, each module consists of a single clock, except synchronizer and clock generator modules.

Latches

- Use latches only when absolutely necessary.
- Watch for feedback loops involving latches that may exist outside of functional operation.
- Latches will require special logic to enable the gate pin during test.

Set and Reset

- Do not internally generate asynchronous set/reset. If you use asynchronous set/reset, these must come from a primary input.
- If you bring an asynchronous set/reset line in, and you must synchronize it before sending it to some flip-flops, then include the synchronized set/reset in your data path to the flip flop, do not use it to drive asynchronous set/reset lines.

RAMs

- Use BIST. It requires a lot of gates, but can be very effective.
- Multiplex all RAM I/O signals to primary I/O's. This introduces a delay in all RAM signal timing, but allows very good control of the RAM for further testing and diagnostic purposes.

Internal Tristates

- If at all possible, avoid internal tristate busses.
- Use either busholders or pull-up resistors on all internal tristate buses.
- All internal tristate buffers must be disabled by the test enable (TE) pin to avoid bus contention during scan.
- Fully decode tristate enables to only allow one driver at a time.

Pin Requirements

- The following pins are required for boundary scan:
  - `TDI, TCK, TMS, TDO, TRST_L`
- The following pins are required for internal scan, for each scan chain:
  - `SCAN_IN, SCAN_OUT, TE`
- It is possible to multiplex scan data pins with normal primary inputs, or with BSCAN pins, if pin count is extremely tight.

Combinational Feedback

- Combinational feedback is strictly forbidden

Bidirectional I/Os

- Disable all bidirectional I/O's with the `TRST_L` pin.

Flip-flops

- Use only flip-flops with scannable equivalents.
- Synthesize with timing margins to ease replacement of non-scan with scan flip-flops.

Design for Synthesis Guidelines

Latch Inference

- Reset should be used in all synchronous blocks to drive all output signals.
- "case" statements should be used instead of "if-else" for proper mux instantiation. Every case statement should have all the inputs in the sensitivity list that are driving the output in the case block.
- Avoid latches and use registers instead. Registers are more testable.

Tristate Buses

- Do not use internal tristate buses. Tristate logic should be used only for IOs of the top level design. Divide the top level design into two parts. top_core . top_with_pads

top_core contains all the logic and sub-modules. Its IOs are either inputs or outputs, not inout. For tristate signals say data[31:0] you should have a data_out[31:0], data_in[31:0] and data_out_oe. top_with_pads instantiates top_core and the IO pad cells.

- Implement the shared bus using a mux.
- Do not mix hierarchical logic and behavioral logic. Top level design should be all pure hierarchial, leaf modules will be behavioral. Intermediate modules should either be pure hierarchial or pure behavioral.
- Limit the hierarchy to a maximum of 5 levels; 3 levels however, are the preferred number.

Synchronous Logic

- Design synchronous logic instead of asynchronous logic. If there is asynchronous logic then there should be clear justification for it.
- Each module should have only one clock.
- Synchronize the asynchronous input using dual flip-flop logic or synchronizers from the target library.
- If the signal crosses one clock domain to another, the signals should be synchronized.

Finite State Machines

- Keep the state machines in separate files and do not mix any other logic within the state machine module.

Multiple Drivers

- A signal in a module should be driven by a single process/always statement. Multiple drivers are not allowed.

Design Verification Guidelines

Code Coverage

- Plan your tests to cover all features of design.

Node Coverage

- Plan and write your tests to toggle all input pins with different combinations. For example tests should exercise at least apply all zeros, all ones and alternate zero and ones to an address input.

- Tests should toggle all output nodes. Use Comit's node coverage tool to check this requirement.

Pattern Generation

- Write tests to generate exhaustive patterns in the debugging stage. Modify your tests to generate random patterns for stable environment testing.

Ease of debugging

* Avoid conditions derived from multiple input conditions. For example, avoid

```
   if(condition_1 & condition_2 & condition_3 & condition_4 &
signal_1 &
                signal_2 & signal_3 & signal_4) .........
         if(condition_1 & condition_2 & condition_3 & condition_4 |
                signal_1)....
```

Use

```
   assign mode_1 = (condition_1 & condition_2 & condition_3 &
condition_4);
   assign mode_2 = (signal_1 & signal_2 & signal_3 & signal_4);
   if (mode_1 & mode_2) ...........
   if (mode_1 | signal_1) .........
```

This page is created and maintained by Rajesh Bawankule
(rajesh52@hotmail.com) (Last Modified 12/13/98)
