Core Design and Implementation Guidelines for ISAAC

Jason Zheng, Yutao He

January 28, 2009

Contents

1 Introduction

The objective of the ISAAC project is to develop and deliver a highly-capable, highly-reusable, modular, and integrated FPGA-based common instrument controller and computing platform, ISAAC (Instrument ShAred Artifact for Computing) that targets a wide range of digital electronics needs for the National Research Council (NRC) Decadal Survey missions.

As part of the main objective, ISAAC aims to deliver a catalogue of high-quality synthesizable cores (iCores) that can be customized and integrated into digital designs for flight missions. To ensure the overall quality and consistency of the iCores designs, a guideline is needed in the design and implementation of the cores delivered by ISAAC; this document serves precisely for that purpose.

At the current revision, this document is mostly Verilog centric. However certain sections, such as design organization is RTL-transparent.

This document is organized as follows. Section 2 makes recommendations for the hierarchical organization of the cores. Section 3 talks about issues related to clock domains. Section 4 discusses the coding guidelines in designing the core logic. Section 5 makes recommendations on code formatting. Finally, a summary list of the important guidelines is shown in Section 6.

1.1 Acronyms

- ISAAC - Instrument ShAred Artifact for Computing
- RTL - Register Transfer Level
- HDL - Hardware Descriptions Language
- SEU - Single Event Upset
- SET - Single Event Transient
- FSM - Finite State Machine
- TMR - Triple Modular Redundancy

2 Hierarchical Planning/Organization

3 Clock Domains

Multiple clock domains are much more difficult to manage than a single clock domain, due to the potential need to synchronize data across clock domains. However, often the power requirements and mixed usage of different bus technologies demand multiple clock domains in designs. In such cases, the number of clock domains shall be kept at a minimum.

When multiple clock domains are used, assign only one clock as the primary clock for each functional block, i.e. most of the logic in the functional block shall

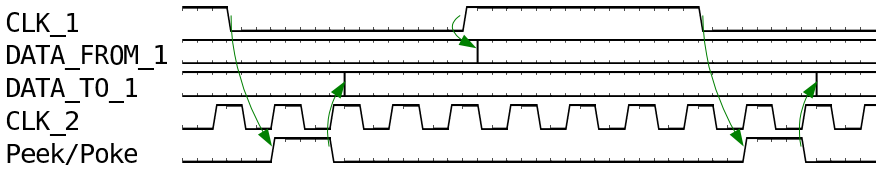

Figure 1: Peek-n-Poke Illustrated

use only one clock except for synchronization logic. Minimize the need to send signals across clock domains by carefully consider alternative design methods.

3.1 Crossing Clock Domains

When data must be communicated between logic units in different clock domains, the probability of meta-stable values must be carefully considered. Meta-stability issues arise when an input signal does not stabilize before the setup time of a flip-flop. As a result, the holding value of the flip-flop during the next clock cycle can either be interpreted as zero or one. Although both rising and falling transitions will settle in the next clock cycle, meta-stable conditions are harmful because different logic units may interpret the value differently. To prevent meta-stable conditions from emerging, the following methods are recommended:

- "Peek-n-Poke"
- "Anticipated Peek-n-Poke"
- "Meta-stability Filter"
- "Asynchronous FIFO"

3.1.1 Peek-n-Poke

The Peek-n-Poke method applies to two unrelated/asynchronous clock domains, where one clock is much faster (at least 4 times) than the other, e.g. data exchange between a 33MHz domain and a 1MHz domain. Due to the large speed difference, the faster logic can sample and detect the active and inactive edge of the slower clock signal, and have full control over when data is received from the slower region (peek), and when data is sent to the slower region (poke). Data to the slower region must be stabilized immediately after inactive edge, and data from the slower region can be sampled after the inactive edge as well. This is illustrated in Figure 1.

3.1.2 Anticipated Peek-n-Poke

When the two clocks are generated from the same source, they have a fixed cycle ratio and phase-shift amount. The Peek-n-Poke can then be implemented

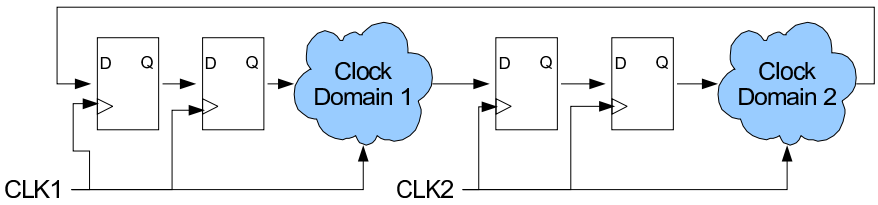

Figure 2: Double FF Meta-stability Filter

without sampling the slower clock. With the help of a counter logic, the faster clock domain can anticipate the arrival time of the slower clock's active edge, and make sure that data going to the slower clock domain maintain stable well before the required setup time. In the case of the slower logic sending data to the faster logic, the faster logic makes sure that data is sampled from the slower region well before the arrival of the slower clock's active edge.

3.1.3 Meta-stability Filters

If the two unrelated clocks have relatively close frequencies, metastability filters can be used as a last resort. A meta-stability filter is two (or more) flip-flops daisy-chained together, such that if the first flip-flop becomes meta-stable, the second flip-flop acts as a buffer to prevent different logic units integrating the meta-stable value differently. However, since meta-stability filters require two flip-flops per signal wire, they should only be used when no other methods can be applied. See Figure 2 for a demonstration of meta-stability filter.

3.1.4 Asynchronous FIFO

Finally, when shuttling wide data bus between two unrelated clock domains with fairly close frequencies, a carefully designed asynchronous FIFO can be used. Such FIFO allow the data read and write to operate with different clocks, and are tricky to design. The actual implementation is beyond the scope of this article, and a detailed discussion of asynchronous FIFO is referenced in the recommended readings (Section 7).

3.2 Clock Signal Generation

When generating clock signals, avoid gated clocks, i.e. clocks "ANDed" with enable signals. If gated clocks must be used, they shall be implemented such that the enable signal maintains stable during active cycle of the clock. For example, if clock CLK_IN is gated by enable signal CLK_EN to produce CLK_OUT, and CLK_OUT a positive-edge clock, CLK_EN is not allowed to change value when CLK_IN is high; doing so will result in a glitched cycle, as illustrated in Figure 3.

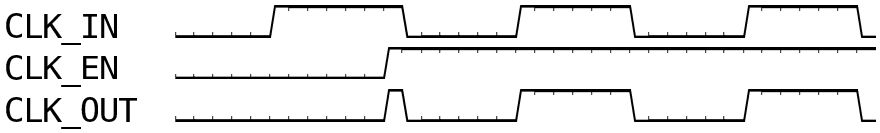

Figure 3: Gated Clock with Glitch

4 RTL Coding Guidelines

4.1 Blocking vs. Non-blocking

Non-blocking assignments evaluate the right-hand-side expression, but hold off the actually assignment until the end of the current time step or the specified delay time. Blocking assignments evaluate the right-hand-side expression, and assign results to the left-hand-side variables before executing anything else. Although the Verilog syntax permits it, mixing blocking and non-blocking assignments can often result in race conditions in simulation. Therefore it is generally considered poor coding to mix the two types of assignments.

The delayed behavior of non-blocking assignments closely models sequential digital circuits, and therefore should be used in always blocks for sequential logic. For example, a flip-flop should use non-blocking assignments:

```
always @ (posedge clk)
  if (reset)
     dout <= 1'b0;
  else
     dout <= 1'b1;
```

In combinational always blocks where multiple variables are touched in sequence, blocking assignments should be used for their C-like behavior:

```
always @ (code[1:0] or a or b or c or d)
  begin
    out[3:0] = 0;
    case (code[1:0])
      2'd0: if (a) out[1] = 1'b1;
            else out[0] = 1'b1;
      2'd1: if (b) out[2] = 1'b1;
            else out[1] = 1'b1;
      2'd2: if (c) out[3] = 1'b1;
            else out[2] = 1'b1;
      2'd3: if (d) out[0] = 1'b1;
            else out[3] = 1'b1;
    endcase
  end
```

In general:

- Use blocking assignments in always blocks for combinational logic
- Use non-blocking assignments in always blocks for sequential logic
- Never mix blocking and non-blocking assignments in one always block
- Never assign to one register in more than one always block

4.2 Full Cases and Parallel Cases

4.2.1 Full Case Statements

From a synthesis tool perspective, a full case refers to a case statement, where all possible bit patterns are either covered by explicit case items or by a default case. The following two examples are both full case:

```
reg [2:0] code;
always @ (code[2:0])
case (code[2:0])
  3'd0: dout = 8'b0000_0001;
  3'd1: dout = 8'b0000_0010;
  3'd2: dout = 8'b0000_0100;
  3'd3: dout = 8'b0000_1000;
  3'd4: dout = 8'b0001_0000;
  3'd5: dout = 8'b0010_0000;
  3'd6: dout = 8'b0100_0000;
  3'd7: dout = 8'b1000_0000;
endcase
```

```
reg [2:0] code;
always @ (code[2:0])
case (code[2:0])
  3'd0:    dout = 8'b0000_0001;
  3'd1:    dout = 8'b0000_0010;
  3'd2:    dout = 8'b0000_0100;
  3'd3:    dout = 8'b0000_1000;
  default: dout = 8'b0000_0000;
endcase
```

A non-full case is a case statement where some bit patterns are not covered. Here is an example of non-full case:

```
reg [2:0] code;
always @ (code[2:0])
case (code[2:0])
  3'd0: dout = 5'b0_0001;
  3'd1: dout = 5'b0_0010;
  3'd2: dout = 5'b0_0100;
  3'd3: dout = 5'b0_1000;
  3'd4: dout = 5'b1_0000;
endcase
```

Perhaps the designer's intent here is that code would never take on values higher than 4. But without additional hints, the synthesis tool must stay consistent with simulation results in all cases: if code takes on a value of 5, the simulator will ignore all the listed conditions, and not change dout's value. The synthesis tool, therefore, will generate latches for dout to match the simulation results.

4.2.2 full_case Synthesis Directive

In the previous example, if the designer is absolutely sure that code can never take on values higher than 5, then synthesis tool can be forced to ignore the bit patterns that are not stated by inserting a *full_case* directive:

```
case (code[2:0]) // synthesis full_case
```

This directive tells the synthesis tool to treat the following non-full case statement as if it were a full case. As a result, no latch will be generated, and dout can take on any value if code is ever higher than 4. Notice that this causes a synthesis/simulation mismatch, since simulators simply treat these directives as normal comments.

The synthesis/simulation mismatch and the possibility of undefined output values are the main reasons why designers should stay away from *full_case* directives. Unless compelling reasons are given, default cases must be used instead of *full_case* synthesis directives.

4.2.3 Parallel and Non-parallel Cases

A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item. If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not "parallel." A parallel case lends itself to generating parallel decoding logic constructs.

A non-parallel case statement is a case statement that contains overlapping case items, and implies a priority order among overlapping items. An example is:

```
always @(code[2:0])
  begin
    {out2, out1, out0} = 3'b0;
    casez (code[2:0])
      3'b1??: out2 = 1'b1;
      3'b?1?: out1 = 1'b1;
      3'b??1: out0 = 1'b1;
    endcase
  end
```

In the above example, since it is possible for more than one bits in code to be set, the case statement is not parallel. Most synthesis tools can distinguish between parallel and non-parallel cases, and generate priority logic constructs for the latter. Another way to write this priority logic is:

```
always @(code[2:0])
  begin
    {out2, out1, out0} = 3'b0;
```

```
    if (code[2])
      out2 = 1'b1;
    else if (code[1])
      out1 = 1'b1;
    else if (code[0])
      out0 = 1'b1;
  end
```

The above example is more clear about the designer's intention than the non-parallel example, and is the preferred method to write priority logic.

4.2.4 parallel_case Synthesis Directives

Synthesis tools can be forced to generate parallel logic constructs for non-parallel case statements by adding a *parallel_case* synthesis directive:

```
always @(code[2:0])
  begin
    {out2, out1, out0} = 3'b0;
    casez (code[2:0]) // synthesis parallel_case
      3'b1??: out2 = 1'b1;
      3'b?1?: out1 = 1'b1;
      3'b??1: out0 = 1'b1;
    endcase
  end
```

In the above example, if more than one bit in code is set, the synthesized logic may generate unexpected results due to the parallel structure. *parallel_case* directives have even more short-comings than *full_case* directives:

Synthesis/simulation mismatch

Possible unexpected output values

Complete different interpretation of original RTL code (priority vs. parallel encoding)

Hence, *parallel_case* directives should never be used unless compelling reasons (e.g. enforcing an intended parallel case in optimized one-hot state machines) are given. Priority logic should always be coded with cascaded if statements.

4.2.5 Combining full_case and parallel_case Directives

Although the use of *full_case* and *parallel_case* directives are generally not good ideas, the two can be used jointly for a special case: one-hot decoding. Here's an example:

```
always @ (state[3:0] or a or b or c or d)
  begin
    nextState[3:0] = 0;
    case (1'b1) /*synthesis full_case, parallel_case*/
      state[0]: if (a) nextState[1] = 1'b1;
                else nextState[0] = 1'b1;
      state[1]: if (b) nextState[2] = 1'b1;
                else nextState[1] = 1'b1;
      state[2]: if (c) nextState[3] = 1'b1;
                else nextState[2] = 1'b1;
      state[3]: if (d) nextState[0] = 1'b1;
                else nextState[3] = 1'b1;
    endcase
  end
```

The one-hot decoding is the only recommended usage for *parallel_case* or *full_case* directives.

4.3 Non-intentional Latches

Incomplete variable assignments in combinational always blocks cause synthesis tools to generate non-intentional latches. Generally such latches are hard to detect by simulation, but can be easily avoided by following a few simple rules. The following example shows a combinational logic block with inferred latches:

```
always @ (a or b or c)
 if (a ^ b)
   dout = 1'b0;
 else if (c & b)
   dout = 1'b1;
```

When the two conditions ($a\hat{b}$) and (c&b) both fail, dout has no other value to take on, therefore the synthesizer has to infer a latch to hold dout's value. To avoid the latch, dout's value must be assigned at all times:

```
always @ (a or b or c)
 if (a ^ b)
   dout = 1'b0;
 else if (c & b)
   dout = 1'b1;
 else
   dout = 1'b0;
```

A more generic way to avoid the latch is to specify a fall-back value at the beginning of an *always* block:

```
always @ (a or b or c)
```

```
begin
  dout = 0;
  if (a ^ b)
    dout = 1'b0;
  else if (c & b)
    dout = 1'b1;
end
```

In general, follow these rules to avoid non-intentional latches:

Always look for warning about any inferred latches in synthesis run logs.

In if statements, all left-hand-side variables must either have fallback values specified before the if statement, or be assigned values in all if/else-if/else cases.

If no fall-back values are given, a case statement should be written as a full case (see Section 4.2.1), where all left-hand-side variables are assigned in all case items.

4.4 Resets

All flip-flops must reset to a known state following a power-on reset event. The reset trigger can either be asynchronous, synchronous, or both. It is recommended that asynchronous resets use active-low signaling, and that synchronous resets use active-high. If other reset conventions are used, they must be followed consistently throughout the design.

4.4.1 Synchronous Resets

Use of synchronous resets generally create an additional layer of logic, and if a large number of flip-flops share the same synchronous reset, the sheer number of fan-outs will force the synthesis tool to insert buffers. Hence there are several short-comings of using synchronous resets in FPGA designs:

The synchronous reset distribution network use up precious routing and logic resources.

The additional buffers and routing delays have negative impacts on the worst-case static timing.

The additional FPGA realty taken up by a large synchronous reset network could push the other logic cells further away from each other, further worsening the static timing.

However, compared to asynchronous resets, synchronous resets are less vulnerable to Single-Event Transients (SET). For each flip-flop, there is only a short window in a clock period where SET on the synchronous reset line could erroneously reset a flip-flop, whereas SET on an asynchronous line could reset flip-flop anytime as long as the SET is wide enough.

4.4.2 Asynchronous Resets

In FPGA design implementation, large asynchronous reset networks are more manageable than the synchronous reset networks. Most FPGA platforms have several built-in clock-distribution networks that can be used for either clock or asynchronous reset distribution. These built-in networks do not have the fan-out issue that plagues synchronous reset networks, and do not take up any logic or routing resources.

However, due to the asynchronous nature of the resets, care must be taken on the release time of the resets. First, skew in an asynchronous reset network must be minimized to ensure all flip-flops leaving reset mode in the same clock cycle. This is generally not an issue unless the asynchronous reset signal is not distributed using a clock network. Secondly, if asynchronous resets are released too close to the active edge of a clock, some flip-flops may become meta-stable. The release time of any asynchronous reset must be locked to occur after the active edge of the clock. The following is an example of an asynchronous reset release logic:

```
module RST_SYNC (ARSTn, PORn, CLK);
  output ARSTn; // This is the signal that resets all flip-flops
  input  PORn;  // This is the input Power-On Reset
  input  CLK;   // Assuming rising edge is the active edge

  reg [1:0] rst;

  always @ (posedge CLK or negedge PORn)
    if (~PORn)
      rst[1:0] <= 2'b00;
    else
      rst[1:0] <= {1'b1, rst[1]};

  assign ARSTn = rst[0];
endmodule
```

The above circuit implementation guarantees that ARSTn rises shortly after the clock rises, provided that the total delay in the ARSTn distribution network is small compared to the clock period.

4.5 FSM Guidelines

4.5.1 General FSM Design

Finite state machines (FSM) are used in all control logic designs. As mentioned before, FSM should generally be placed in dedicated modules to separate control and data path. It is recommended that all FSM are partitioned to the following structure:

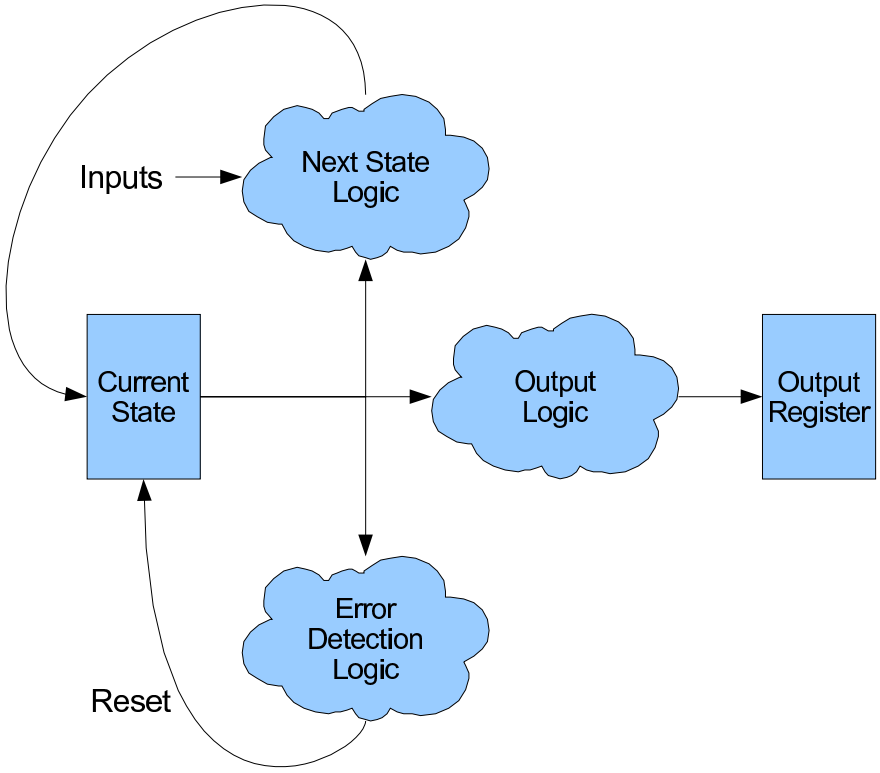

Figure 4: FSM Conceptual Diagram

Declaration of state register, next-state variable, and parameters that define each state's symbolic names.

A sequential block that sets and resets state register

A combinational block that generates next-state based on current state and inputs

A combinational/sequential block that generates output

An optional state error detection/correction logic block that resets the state machine or generates an interrupt output when the state register is corrupted by Single-Event Upsets.

Figure 4 shows a conceptual view of a FSM implementation. When designing FSM, the following general rules shall be followed:

All states should have symbolic names defined as module parameters.

Registered outputs are preferred over combinational outputs to help improve static timing.

Next-state logic should reside in a separate combinational logic block. If using binary encoding, small FSM (less than 8 states) may omit the combinational next-state block, and determine the next state inside the state register's sequential block.

4.5.2 State Encoding

4.5.3 Special Notes on One-hot FSM

State machines designed with one-hot encoding should **NOT** have the following:

Decode state with case statements containing a default case. This will result in a large and slow decoding circuit. Use a hand-crafted XNOR tree if SEU is a concern for the state register.

Decoding all bits of a state register. This practice results in a slow state machine. Instead, compare 1 bit only when decoding one-hot states.

An example of one-hot state machine is given below:

```
‘timescale 1ns/100ps
module fsm_count4 (state, rst_an, rst, clk);
  input        rst_an;
  input        rst;
  input        clk;
  output [3:0] state;

  parameter st0 = 2’h0;
  parameter st1 = 2’h1;
  parameter st2 = 2’h2;
  parameter st3 = 2’h3;

  reg [3:0] state /* synthesis syn_preserve=1 syn_keep=1 */;
  reg [3:0] nextState;

  always @ (posedge clk or negedge rst_an)
    if (~rst_an) state <= 4’h1;
    else if (rst) state <= 4’h1;
    else state <= nextState;

  // next-state logic (combinational)
  always @ (state) begin
    nextState = 4’h0;
    case (1’b1) // synthesis full_case parallel_case
      state[st0]:   nextState[st1] = 1’b1;
      state[st1]:   nextState[st2] = 1’b1;
      state[st2]:   nextState[st3] = 1’b1;
      state[st3]:   nextState[st0] = 1’b1;
```

```
    endcase
  end

endmodule
```

4.6 Miscellaneous Items

Synthesis tools usually cannot optimize well across modular boundaries, therefore output ports driven by registers are preferred over those driven by combinational logic. For the same reason, kludge logic in high-level modules shall be avoided by all means.

Avoid cascading conditional expressions like this:

```
assign a = b ? c : (d ? e: (f ? g : h));
```

Such logical constructs can be alternatively written using cascaded if-else-if statements:

```
  reg a;

  always @ (b or c or d or e or f or g or h)
    if (b)        a = c;
    else if (d)   a = e;
    else if (f)   a = g;
    else          a = h;
```

Incomplete sensitivity lists can cause synthesis/simulation mismatches, and usually do not reflect the designer's intention. Make sure all sensitivity lists are complete in all combinational always blocks.

Early anticipation of the design critical path(s) can make the synthesis and place-n-route process significantly easier. If a part of the design has been identified as a potential critical path, try simplifying or pipelining the logic if possible. If no additional latencies can be added, re-timing (moving the pipeline registers) might help. A good practice is to estimate logic levels by synthesizing part of the design.

5 Code Formatting

5.1 File Header Section

All HDL files shall carry a descriptive header that includes the following items:

- Main module's name
- Designer/owner's names
- Brief description of the module and key features

- Revision History
- Table of Contents to help readers understand the design
- (Optional) Variable naming conventions

5.2 Code Comments

Always write accurate, meaningful comments, and avoid excessive comments such as:

```
assign a = b ? c : d; // if b is set, a=c, otherwise a=d
```

When maintaining/updating code, make sure that the associated comments are kept consistent with the changes. Code/comment inconsistency are always confusing; a reader cannot determine whether the comments are wrong, or if there is a bug. For example, when updating original code like this:

```
assign alarm = sun_rise|sun_set; // sun rise/set trigger alarm
```

An engineer updates the trigger condition, but forgets to update the comment:

```
assign alarm = sun_rise|sun_set|bat_low; // sun rise/set trigger alarm
```

The result is a confusing design by another reader. Is bat_low intended as an alarm trigger? Or is the comment line wrong? Without a design specification, it is hard to determine whether this comment mismatch indicates a design flaw.

5.3 Modules

Keep the module port names independent from other module port names. Try to keep port names consistent from top-level to sub-modules for easier integration. Whenever possible, use multi-bit buses instead of single-bit ports.

All module instantiations must use explicit port names to avoid port connection errors, and ports should be grouped in a logic order, preferably in the order of: outputs, inputs, and inouts. Keep one port connection per line, and line up the port connections vertically.

5.4 Naming Conventions

The following naming conventions are recommended:

- Module names should start with a upper-case letter
- Variable names should start with a lower-case letter
- Use n or _n suffix to identify negative logic variables or wires

- Use _d or _z suffix to denote delay pipelines
- When naming module instantiations, prefix the instance names with ModuleName_, and always use a "_" suffix. For example, an instantiation of module ICC could be named ICC_0_. The suffix is used to help preserve hierarchy in the post-synthesis netlist.

5.5 Miscellaneous Formatting

Keep the line length to 80 characters or less to prevent line-wrapping in text editors. Use consistent indentation.

Width of bus or registers shall be declared in [MSB:LSB] format, do not use [LSB:MSB]. When using ranged bus or registers, always explicitly write the range. For example:

```
output [31:0] PCI_AD;
input  [32:1] DATA_1553;
assign PCI_AD = DATA_1553;
```

Should be replaced by:

```
output [31:0] PCI_AD;
input  [32:1] DATA_1553
assign PCI_AD[31:0] = DATA_1553[32:1];
```

Use proper indentations to make long logic assignments more clear:

```
assign #g_dly reset_timer = (go_idle_hi_sync |
                             go_hi_sync_lo_sync |
                             go_hi_sync_error);
```

6 Summary

- Break up functional blocks into Datapath, Control/Status, RAM/FIFO, and FSM sub-modules.
- Avoid kludge logic in high-level modules.
- Avoid modular outputs driven by combinational logic, always use registered outputs if possible.
- Keep the number of clock domains at minimum
- One primary clock domain per functional block/module
- Avoid unnecessary cross-domain signals.
- Use meta-stability filters only when other synchronization methods can't be used

- Avoid gated clocks; if must use, change the clock-enable in the inactive cycle.
- Use non-blocking assignments for sequential always block.
- Use blocking assignments for combinational always block.
- Never mix blocking and non-blocking in one always block.
- Never use more than one always blocks to assign to one variable
- Stay away from both full_case and parallel_case synthesis directives except for one-hot decoding.
- In case or if statements, if no fall-back values are given, variables must have assigned value in all conditions, including default/else cases.
- All flip-flops must have resets; be consistent with reset conventions.
- Partition FSM designs into declaration, state register, next-state, output, and error detection/correction sections.
- Never use default case with one-hot FSM, never use more than one bit to decode one-hot states.
- Avoid cascading conditional expressions; use if/else if/else instead.
- Complete all sensitivity lists for combinational always blocks.
- Each source file must carry a header
- Comments should be accurate, consistent, meaningful, and not excessive.
- Keep module port names independent from other module's port names.
- Keep port names consistent from top to bottom.
- Use multi-bit buses instead of single-bit ports
- Use explicit port names in module instantiation.
- Group ports in the order of outputs, inputs, and inouts.
- Prefix module instance's names with module's name.
- Keep lines short enough to avoid line-wrapping.
- Use explicit bit-ranges for all multi-bit nets or registers at all times.
- Bit-ranges should be [MSB:LSB].
- Use proper indentations, especially for long logic assignments.

7 Recommended Readings

- Cliff Cummings, Simulation and Synthesis Techniques for Asynchronous FIFO Design, SNUG 2002.
- Cliff Cummings, Nonblocking Assignments in Verilog Synthesis; Coding Styles That Kill!, SNUG 2000.
- Cliff Cummings, "full_case parallel_case", the Evil Twins of Verilog Synthesis, SNUG 1999.
- Cliff Cummings, State Machine Coding Styles for Synthesis, SNUG 1998.

All the papers above are available at Cliff Cumming's website: http://www.sunburst-design.com/papers/