Reusable IP Coding Guidelines

Basic Principles of Reusable RTL Coding Guidelines

- **Readability**
- **Simplicity**
- **Locality**
- **Portability**
- **Reusability**
- **Reconfigurability**
- General recommendations
  - **Simple** constructs, simple clocking scheme
  - Consistent coding style, naming conventions and structure
  - Regular partitioning with registered output
  - Make RTL code easy to understand by comments, meaningful names

Naming conventions

- **Consistent naming convention** for the design
- Lowercase for signal names
  - e.g. `ram_addr`
- Upper case for constants
  - e.g. `WIDTH`
- *clk* prefix for clocks, e.g. `clk1, clk2`
- *rst* prefix for resets, e.g. `rst_n`
- Suffix
  - _n: active low, _z: tristate, _`nxt`: data before being registered, _a: asynchronous
- `_cs` for current state, `_ns` for next state
- Same or similar names for connected ports and signals
- Consistent ordering for multibit signals, recommended `[x:0]`

File Header

```
// +FHDR------------------------------------------------------------------------
// Copyright (c) 2003, ABC Corporation.
// ABC's Proprietary/Confidential
//
// -----------------------------------------------------------------------------
// FILE   :
// TYPE   : Verilog Module
// AUTHOR :
// -----------------------------------------------------------------------------
// Revision History
// VERSION DATE       AUTHOR DESCRIPTION
// 1.0     6 Jan 2003 Name   First release
// -----------------------------------------------------------------------------
// KEYWORDS : for file searching
// -----------------------------------------------------------------------------
// PURPOSE : Short description of functionality
// -----------------------------------------------------------------------------
// PARAMETERS
// PARAM_NAME RANGE : DESCRIPTION : DEFAULT
// DATA_WIDTH [32:16]: width of data : 32
// -----------------------------------------------------------------------------
// REUSE ISSUES
// Reset Strategy :  rst_n
// Clock Domains :   clk
// Critical Timing :
// Test Features :
// Asynchronous I/F :
// Scan Methodology :
// Instantiations :
// Other :
// +FHDR------------------------------------------------------------------------
```

- Included for all source files
- Corporation-wide standard template

Comments and Formats

- Appropriate comments
  - For processes, functions, ports, signals …
  - Describe the intent behind the section of code
  - Insert comments before a process for readability
- Keep commands on separate lines
- Line length <= 72 characters
- Coding in a tabular manner
- Indentation
  - 2 spaces
  - Avoid using tabs

Ports

- Port ordering
  - One port per line with a comment
  - Declare in a logical order
    - Inputs: clocks, resets, enables, other control signals, data and address signals
    - Outputs: clocks, resets, enables, other control signals, data
  - Comments for group of ports
- Port mapping
  - Explicit **name mapping** instead of positional mapping
    - BAD: `bad u_bad(4’h2, a, b, c);`
    - Good: `good u_good(.x(4’h2), .a(a), .b(b), .c(c));`

Coding Practices (1/2)

- Little-endian for multi-bit bus
  - `[31:0]` instead of `[0:31]`
- Operand size should match
  - BAD: `reg [32:0] a; reg [31:0] b; a = b;`
- Expression in condition should be an 1-bit value
  - `if(abc != 16’h0)` instead of `if(abc)`
- Use ( ) in complex statements
- No X assignment
  - Avoid X-state propagation
- Reset all storage elements
  - Avoid X-state propagation

Coding Practices (2/2)

- Use function for common combo logic
  - Avoid repeat the same code
- Use local variables
- Use `for` loop judiciously
  - Improve readability
  - Increase simulation and synthesis compilation time
  - Use arrays whenever possible
- Use meaningful labels for debug

```
always @ (a or b)
begin: p_test
end
```

```
foo u_foo1(..);
foo u_foo2(..);
```

Coding for Portability

- Do not use HDL reserved words for naming
  - Designs should be bilingual for automatic translation
- Avoid embedded synthesis commands
- Use constant definition files
- Do not use hard-coded numeric values
- Use technology independent libraries
  - Avoid instantiating logic gates, isolate them if needed
  - use DesignWare components

Poor coding style

```
wire    [7:0]   my_in_bus;
reg     [7:0]   my_out_bus;
```

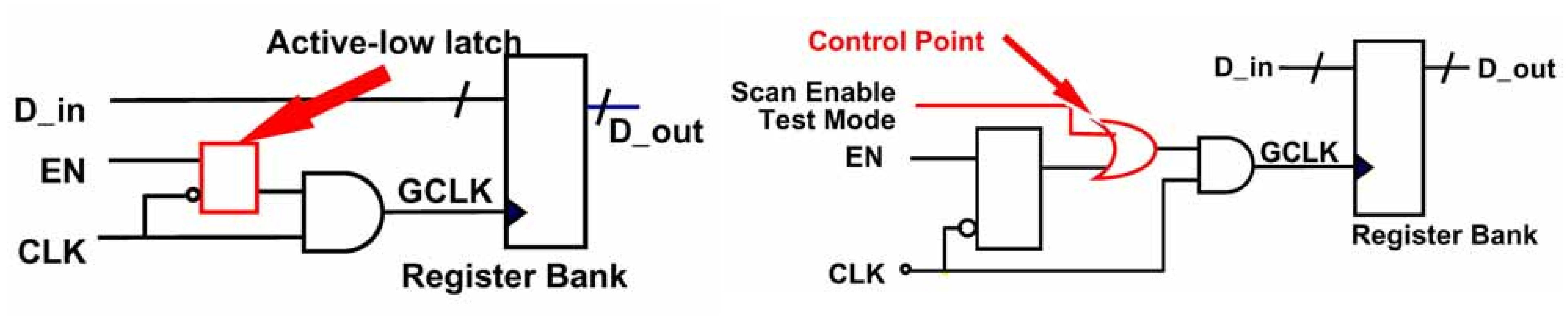
Recommended coding style

```
`define MY_BUS_SIZE 8
wire    [MY_BUS_SIZE-1 :0]     my_in_bus;
reg     [MY_BUS_SIZE-1 :0]     my_out_bus;
```

Clocks and Resets

- **Simple clocking** is easier to understand, analyze, and maintain
- Avoid using both edges of the clock
  - Duty-cycle sensitive
  - Difficult DFT process
- Do not buffer clock and reset networks
- Avoid gated clock except for low power design
- Avoid internally generated clocks and resets
  - Limited testability
  - Isolate clock/reset control module if needed
- Use single-bit synchronizers instead of multiple-bit synchronizers for transfer between clock domains
  - Possible skew in bits results in error sampling value

DFT for Gated Clock



- Use scan enable or test mode as control point for better fault coverage
- Latch GN stuck-at-0 fault is untestable

Coding for Synchronous Design

- Infer technology independent registers
  - No `initial` statement to initialize the signal to avoid mismatch
- Avoid latches intentionally or unintentionally
  - Exception: low power design
  - Latch infer
    - Incomplete assignment in case statements
    - Incomplete if-then-else
  - Isolate them if needed
- Avoid combinational feedback
  - STA and ATPG problem

Combinational and Sequential Blocks

- Combinational blocks
  - Use block assignments (=)
  - Complete but not redundant sensitivity lists
  - In topological order

```
always @(a or b or c)
  begin
   x = a & b;
   y = x | c;
end
```

- Sequential blocks
  - Use non-block assignments (<=)
  - Avoid race in simulation
  - Separate combinational and Sequential blocks

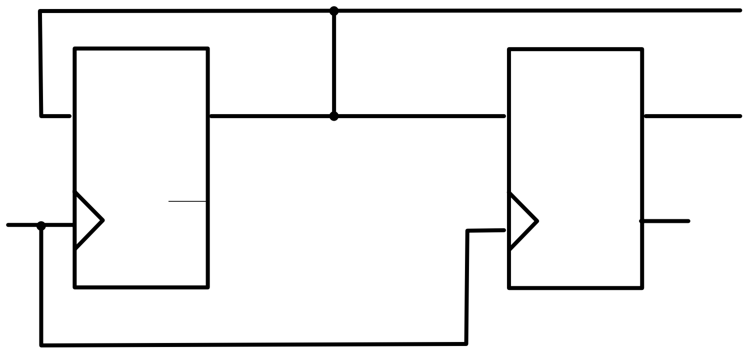
```
always @(posedge clk)
  begin
       x_r <= x;
       y_r <= y;
   end
```

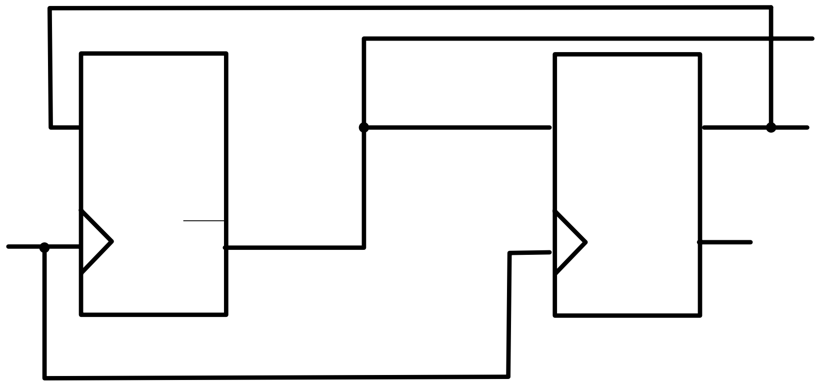
Blocking v.s. Nonblocking

- Blocking & nonblocking assignments
  - Always use nonblocking assignments in `always @ (posedge clk)` blocks

```
always @ (posedge clk)
begin
    b = a;
    a = b;
end
```

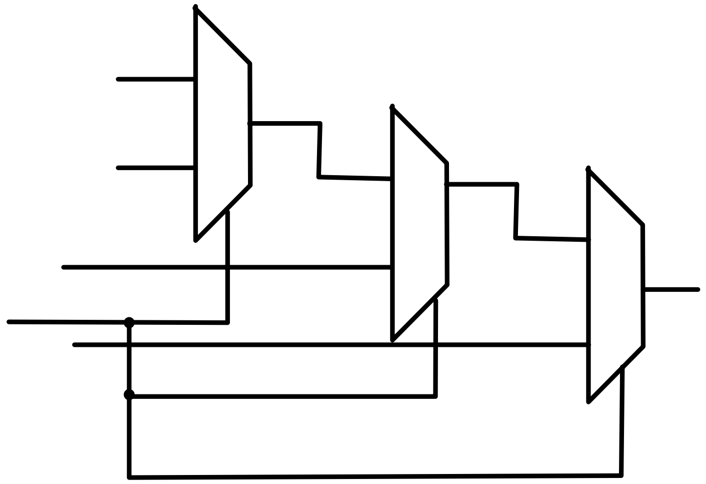
```
always @ (posedge clk)
begin
    b <= a;
    a <= b;
end
```

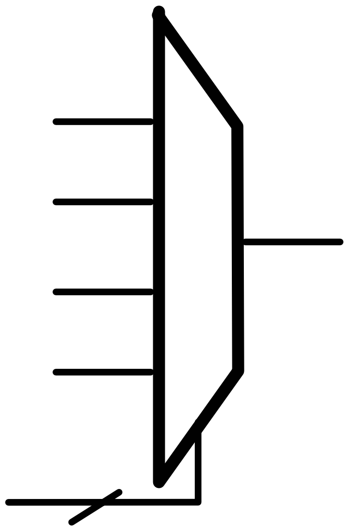




if-then-else v.s. case

- `if-then-else` often infers a cascaded encoder
  - Suitable for signals with different arrival time
- `case` infers a single-level MUX
  - `case` is better if priority is not required
  - `case` is generally simulated faster then `if-then-else`
  - MUX is a faster circuit
- Conditional assignment (`?:`)
  - Infers a MUX or priority encoder
  - Slower simulation performance
  - Better avoided





Coding for FSM

- Keep FSM and non-FSM separate
  - Ease synthesis
- Partition into combinational and sequential part
  - Two always style (Mealy style)
  - Three always style (Moore style)
- Use parameters to define state vector
  - Readability
- Use default (reset) state

Coding for Synthesis

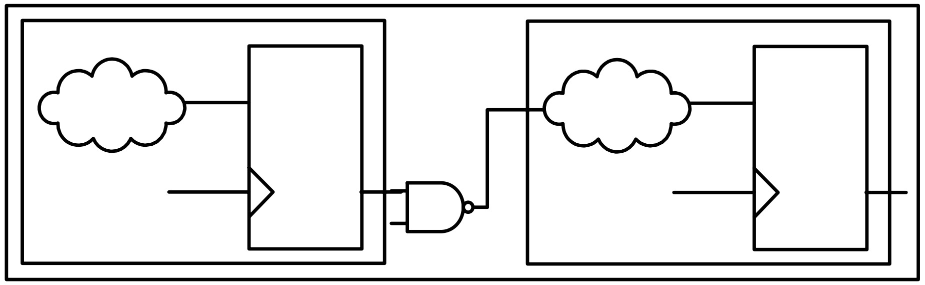
- No #delay statements
  - Mismatch between pre- and post-layout
  - Delay only for
    - Mixed RTL and gate-level simulation
- Avoid full_case and parallel_case
  - Mismatch between pre- and post- simulation
- Avoid expressions in port connections
  - Bad for debug
  - e.g. `test u_test (.a (x & y), …);`
- Coding critical signals
  - Late arriving signals closest to the output

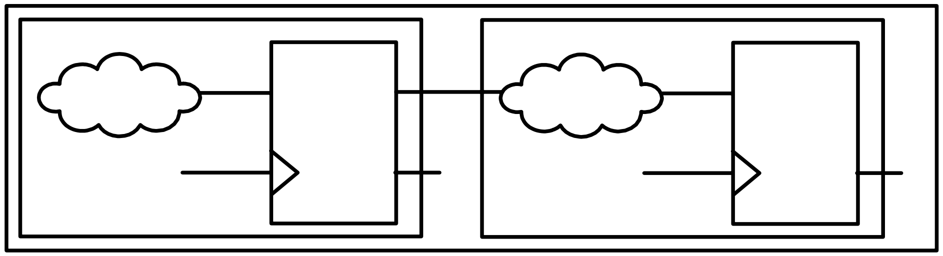
Partition for Synthesis (1/3)

- Register all outputs of subblocks
  - Predictable output drive strengths and input delay
  - Ease timing budget
- Locate related combinational logic in a module
  - Improve synthesis quality
- Separate modules that have different design goals
- Avoid asynchronous logic
  - Technology dependent
  - Hard to ensure correct functionality and timing
  - Isolate if needed and keep it as small as possible

Partition for Synthesis (2/3)

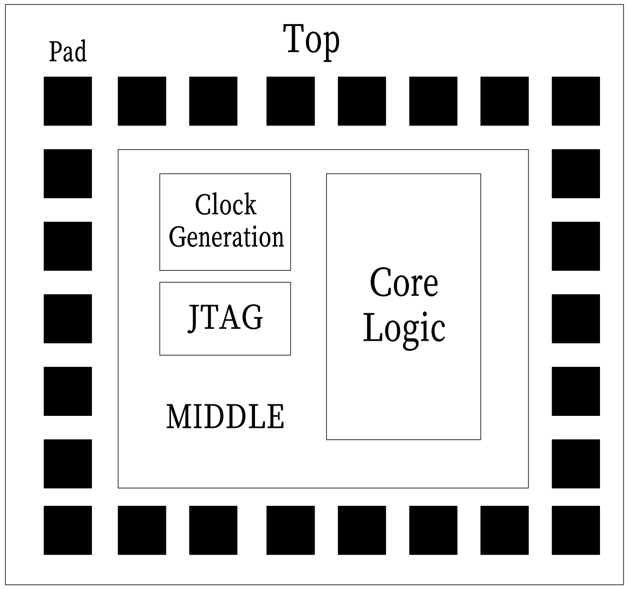
- Resource sharing
  - Keep sharable resource in one always block
- Partition for synthesis runtime
  - Avoid over constraints
- Avoid timing exceptions
  - Hard to analyze, slow down design tools
  - Isolate point-to-point exception in one module
- Eliminate glue logic at the top level





Partition for Synthesis (3/3)

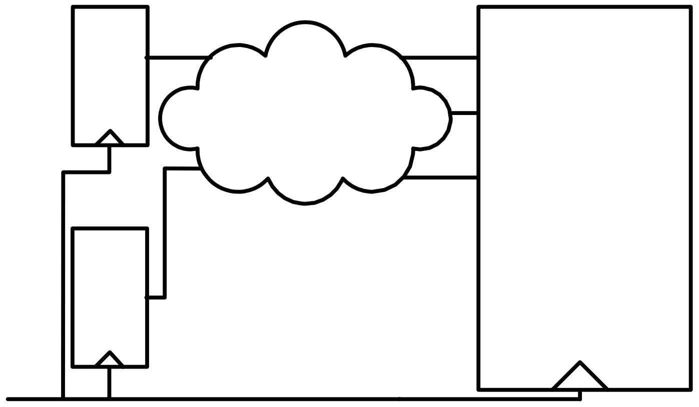
- Chip level partitioning
  - Level 1: I/O pade ring only
  - Level 2: clock generator, analog, memory, JTAG
  - Level 3: digital core



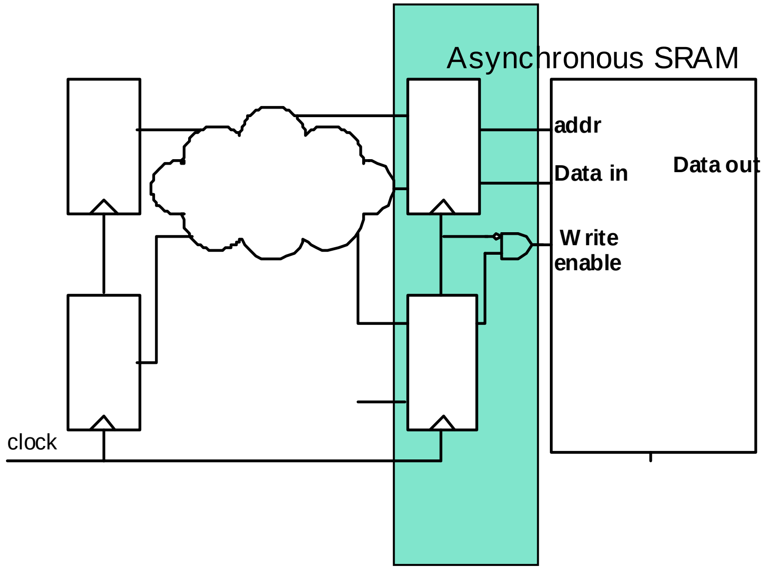
Design with Memories

- Synchronous memory is preferred
  - Asynchronous RAM suffers write enable pulse problem

Synchronous memory interface



Asynchronous memory
with synchronous interface



Coding for DFT

- Avoid tri-state buses
  - bus contention, bus floating
- Avoid internally generated clocks and resets
- Scan support logic for gated clocks
- Clock and set/reset should be fully externally controllable under the test mode

Code Profiling

- Indicate how much time each module takes during simulation
  - 20-80 rule
  - Profiler looks only at the line execution frequency instead of machine cycles
- Help optimize simulation performance
  - Necessary for large designs

Linter

- Static RTL code checker
  - Fast, without simulation
- Category
  - RTL purification
    - Syntax, semantics, simulation
  - Testability checks
  - Reusability checks
  - Timing checks
- Shorten design cycle by avoiding lengthy iterations

More Guidelines

- Verilog HDL Coding – Motorola's SRS
- Design Style Guide – Japan STARC
- FPGA Reuse Field Guide – Xilinx