Module 16

Managing the RTL Coding Process

cadence

This module suggests some RTL coding “Best Practices” for code meant for synthesis.

Module Objective

In this module, you:

- Manage the RTL coding process

Topics

- Managing the project
- Partitioning for synthesis
- Coding RTL for synthesis

cadence

This module examines some rules of thumb for preparing a design to ease the synthesis process. It discusses:

- Project management – Conventions for organizing and naming things.
- Partitioning for synthesis – Setting up the design hierarchy.
- Coding RTL for synthesis – Mostly a summary of rules from previous modules.

Managing the Project

Adopt a convention for identifiers:

- The name reflects the meaning
  - Without being overly long
- The suffix reflects the category
  - _i(input), _n(active low)
- Agree on capitalization, separation
  - my_name, MyName, myName
- Agree on tense (action, description)
  - grant_bus, bus_grant
- Avoid look-alike characters
  - 0(zero)/O(oh), 1(one)/I(eye)/l(ell)
- Avoid names differing only in case
  - Myname, myName, MyName
  - Especially if they look like keywords

Adopt a convention for project file names and locations.

```
ASIC
|_ Sources          (.v)
|   |_ Archives
|_ Synthesis
|   |_ Scripts      (.tcl)
|   |_ Constraints  (.sdc)
|   |_ Logfiles     (.log)
|   |_ Reports      (.rpt)
|   |_ Netlists     (.v)
|   |_ Archives
|_ Simulation
|   |_ Scripts      (.tcl)
|   |_ Logfiles     (.log)
|   |_ Archives
|_ OtherTool...
```

302

cadence

To facilitate project management, it is imperative that you organize your project data. This illustration is a starting point for your team discussions. Your ultimate project organization will very likely be different than this illustration and probably also more complex.

To facilitate communication among your team members, it is imperative that you adopt a convention for file names. This illustration suggests file extensions that are commonly used. It is also suggested that each file describe a single module having the same name as the file.

As your project evolves, it is imperative to maintain project archives and records. This allows you to back out of changes that break the project and allows management to utilize project history to estimate future developments. For these purposes, several popular version control mechanisms are available to control project file creation, maintenance, replacement, and deletion.

It is equally important to adopt a convention for identifiers:

- The name should reflect the purpose of the net, variable, constant, module or subroutine, yet without being overly long.
- Suffixes should reflect the name category, for example _i for inputs, _o for outputs, and _n for active-low signals.
- Capitalization and word separation should be applied consistently.
- How you use names should also be consistent, for example, which do you mean the name to reflect – an action that the signal does, or a description of the signal?
- Avoid look-alike characters, for example in some fonts the uppercase I (eye) looks identical to the lowercase l (ell) and the number 1 (one).
- Avoid names that differ only in case, especially names that would be keywords if in lower case. As output netlists are almost invariably Verilog, VHDL users should also avoid names that are Verilog keywords.

Partitioning for Synthesis

This section examines partitioning the design for synthesis:

- Registering block outputs
- What to keep together
- What to keep apart
- Partitioning for design reuse

cadence

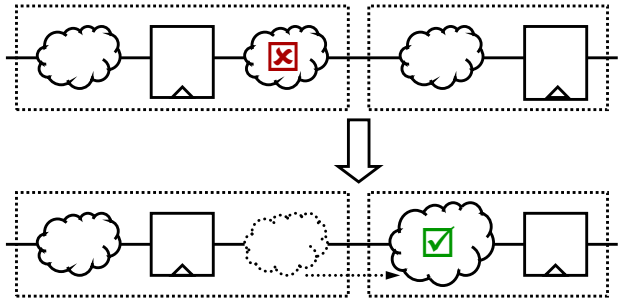
This section separately examines partitioning the design for synthesis. It discusses:

- Registering block outputs;
- What design parts to keep together;
- What design parts to keep apart; and
- Partitioning the design to facilitate its reuse.

Registering Block Outputs

Constraints are simple and identical for each module:

- Input drive strength is the drive strength of the preceding flip-flop.
- Input arrival time is the path delay through the preceding flip-flop.



304

cadence

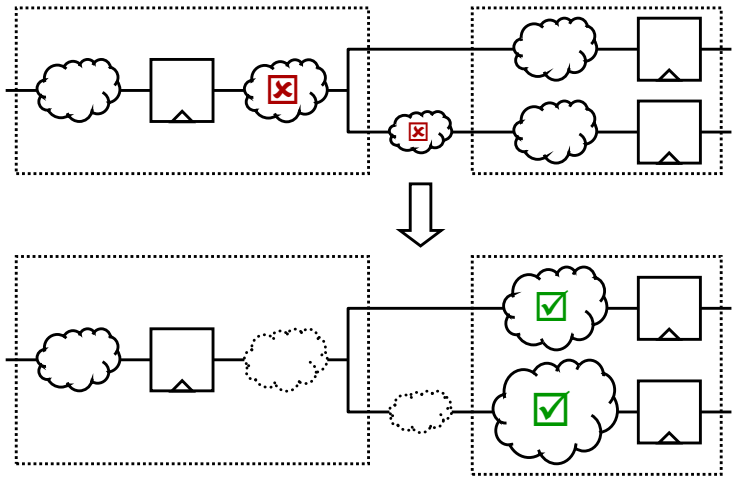
You will find the synthesis process is easier to manage if you have registers on the outputs of each of the hierarchical blocks in your design. This is because it is easier to set the constraints on each block, as the arrival time of signals at each input and output are well defined.

- With unregistered outputs, the designer must specify what proportion of the clock period is available to implement each combinational block. This is done by specifying input and output delays for each block. These delays must be realistic and reflect the comparative performance of the logic blocks.
- With registered outputs, the synthesis tool has almost a complete clock period to implement the combinational logic at the input of the downstream block.

Keeping Combinational Logic Together

Keep combinational logic in the block of its target register.

- This is especially true of “glue” logic.



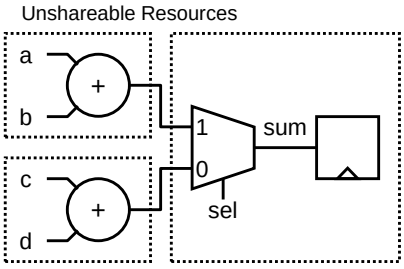
Synthesis tools do not generally, except perhaps for inverters, move logic across hierarchical boundaries. They thus cannot fully optimize combinational logic distributed among multiple blocks. This suggests two strategies you can utilize to facilitate optimization:

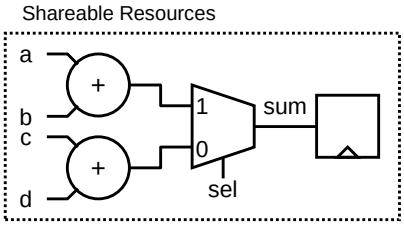
- 1st – As much as possible, group the combinational logic cone with the register that it feeds. This simplifies the timing constraints and permits the tool to fully optimize the combinational logic. Where combinational logic feeds registers in multiple blocks, consider merging portions of those blocks or duplicating the combinational logic.
- 2nd – Minimize or even eliminate the combinational “glue” logic at the parent level between instantiated blocks, as the tool can do little to optimize very small amounts of combinational logic.

You should as much as practical move minor connective (glue) logic into one or the other instances that it connects.

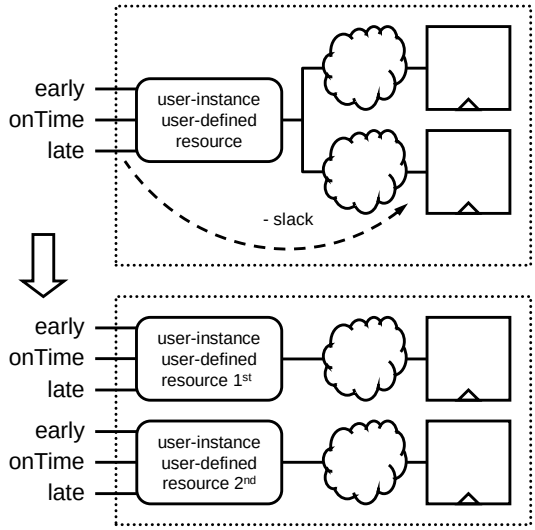
Especially Keep Resources Together

Keep resources with the register that they feed.





Keep sharable resources together.

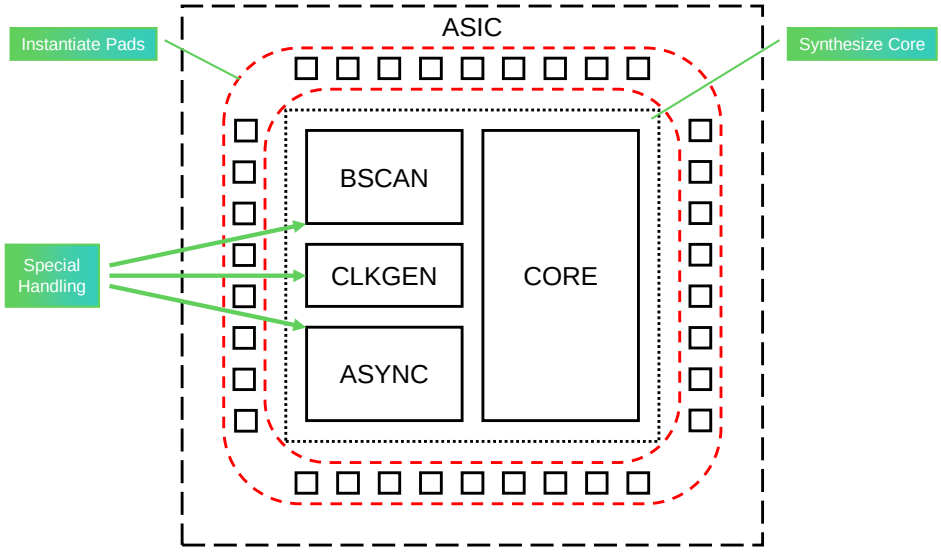


To share resources, the synthesis tool must perform a lifetime analysis of the operation results. The tool will likely be able to do this only if the operators are in the same procedural block. Keep the operators close together to facilitate this analysis.

Keeping the resource with the register that it feeds is an expansion of keeping combinational logic with the register that it feeds. To meet timing requirements, the tool can duplicate its own resources even if you manual instantiate them, like any combinational logic, as long as the resources are located in the scope of the registers that they feed. The tool cannot duplicate user-defined resources, but you will find it easier to duplicate an instance yourself if it is co-located with its register.

Separate Auxiliary Logic from Core Logic

Separate auxiliary logic (pad ring, clock generator, boundary scan, etc.) from core logic logically (and later) physically.



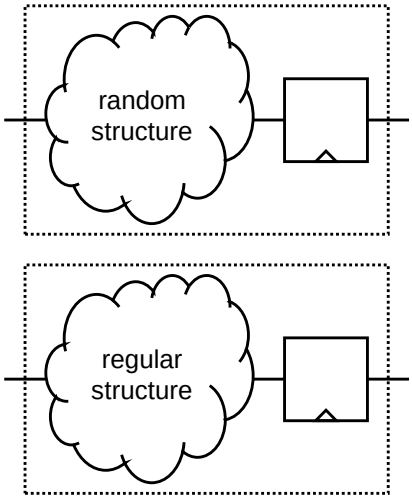
Separate auxiliary logic such as the pad ring, clock generator, and boundary scan from core logic both logically and later physically.

A middle level of hierarchy is recommended to clearly separate the instantiated pad ring from synthesized elements. If you utilize a clock backbone instead of a clock tree, you may want to place the backbone outside of the middle hierarchy of synthesized elements.

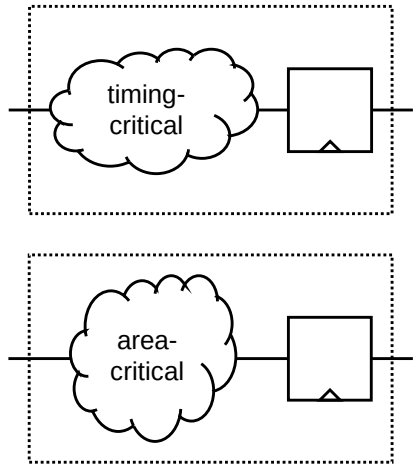
Within the middle hierarchy, you clearly separate the elements that require special handling from the bulk of sequential core logic for which synthesis is relatively straightforward.

Separate Blocks Needing Different Synthesis Techniques

Separate area-critical blocks from timing-critical blocks.



Separate regularly-structured blocks from randomly-structured blocks.



Place into separate modules those parts of your design that need special handling:

- The synthesis tool can perform 2-level minimization on random combinational logic that it can easily represent as a sum-of-products (SOP) or product-of-sums (POS). This technique may not by default be used. You may need to encourage the tool to use this technique, which encouragement you can most easily apply on a per-module basis. The tool likely cannot apply this technique to complex regular structures, especially those containing networks of exclusive-or operations.
- The synthesis tool will almost invariably favor timing over other cost factors as it optimizes the design. For a subdesign that you know has noncritical timing requirements, you can request that the tool favor some other cost factor, such area, which request you can most easily apply on a per-module basis.

Partitioning for Design Reuse

Keep reusability in mind throughout the design cycle:

- Adopt industry-standard interfaces and corporate-wide conventions.
- Partition in a manner that makes design pieces reusable.
- Parameterize your designs appropriately.

cadence

You will almost invariably during your career make frequent use of existing designs. You will probably at least occasionally think unhappy thoughts about the people who made some of those not-easily-reused designs. Keep reusability in mind throughout the design cycle – the next user could very easily be yourself:

- Adopt industry-standard interfaces and corporate-wide conventions;
- Partition in a manner that makes design pieces reusable; and
- Parameterize your designs appropriately. Appropriately means exactly as useful and no more. For example, a byte will always be 8 bits and a minute will always be 60 seconds, so those are constants you do not need to parameterize.

Coding RTL for Synthesis

Modules

- Avoid passing constants through ports.
  - Use parameters.
- Avoid port expressions.
  - Use port identifiers with/without a range.

Functions

- Avoid referencing values other than inputs.
- Synthesis generally make functions into combinational logic.
- Synthesis treats *static* functions as *automatic.*

Expressions

- Avoid unneeded parentheses in arithmetic expressions.
  - Prevents carry-save optimization.
- Use parentheses in non-arithmetic expressions to improve readability.

```
module my_mod (
 .i(a),
 .j1(b[1]),.j0(b[0]),
 .k({c,d}),
 y
);
 input         a,c,d;
 input  [1:0]  b;
 output        y;
 ...
```

```
y=(a+b)?(c*d+e*f):(c*e+d*f);
y=a&b^c&d|e&f^g&h;
```

```
y=a+b?c*d+e*f:c*e+d*f;
y=((a&b)^(c&d))|((e&f)^(g&h));
```


Passing constants through ports creates non-optimal code that synthesis must then optimize “away” and it may in some situation prevent that optimization. You should instead either declare the constant locally or pass it as a module parameter that the elaborator resolves.

A port can be just an identifier or it can be a port expression with or without an identifier. The port expression can be an identifier or a bit or part select of an identifier or a concatenation of these. Port expressions provide a way to have different external and internal names for ports and, and oh by the way, greatly complicate your debugging efforts.

The issues involved with functions are concerned mostly with simulation mismatches between the pre-synthesis RTL design and the post-synthesis gate-level design. These issues also apply to tasks, for synthesis tools handle tasks that do not contain event controls as if they were functions.

- The simulation tool most likely evaluates a function (used in a continuous assignment) only when one of its inputs transitions. If the function result is partly determined by a signal that is not an input to the function, synthesis of course includes that signal in the hardware representation of the function, thus potentially creating a mismatch between pre-synthesis simulation and post-synthesis simulation.
- The synthesis tool most likely infers combinational logic from a function assignment statement that in a procedural block would infer a latch due to incomplete assignment. The synthesis tool may choose a value for the missing assignment and might not even report that it is doing so. This can create a mismatch between pre-synthesis simulation and post-synthesis simulation.
- Synthesis expands the function at the point of the call, creating a new set of local variables for each call, thus treating the function definition as automatic. The standard requires that you declare functions automatic for synthesis but most tools do not enforce this requirement.

The issues involved with expressions are with the use of parentheses. You are encouraged to use parentheses to improve the expression readability, but they have the side affect of forcing term grouping that optimizations have to honor, thus significantly reducing optimization.

Module Summary

Adopting these suggestions will help you to manage the RTL coding process.

This module discussed:

- Managing the design project
  - Adopting a convention for project file names and locations
  - Adopt an identifier naming convention
- Partitioning for synthesis
  - What to keep together (what to keep apart)
  - Partitioning for design reuse
- Coding RTL for synthesis

cadence

This module examined some “rules of thumb” for preparing a design to ease the synthesis process.

It discussed:

- Project management – Conventions for organizing and naming things.
- Partitioning for synthesis – Setting up the design hierarchy.
- Coding RTL for synthesis – Mostly a summary of rules from previous modules.

Module Review

1. What can you do to promote reuse of your designs?
2. Suggest at least one preferred coding practice that you cannot always strictly adhere to.

cadence

This page does not contain notes.

Module Review Solutions

1. What can you do to promote reuse of your designs?
   - Adopt industry-standard interfaces and corporate-wide conventions
   - Partition in a manner that makes design pieces easy reused
   - Parameterize your designs appropriately
2. Suggest at least one preferred coding practice that you cannot always strictly adhere to.
   - You should in general partition your design to enable resource sharing, but in some situation, for example to decrease negative slack, you may need to prevent sharing of a specific resource.
   - You should in general parenthesize expressions to improve readability, but parenthesizing arithmetic expressions can prevent carrysave optimizations.

cadence

This page does not contain notes.

Labs

There are no labs in this module.

cadence

This page does not contain notes.