RTL Coding and Optimization Guide for use with Design Compiler

Jack Marshall

Tera Systems Inc.

jackm@terasystems.com

ABSTRACT

My career at Synopsys spanned 9 years and was focused entirely on front-end design. With each new position within the company (FAE, Trainer, Consultant, Manager) I learned more about high level design, coding styles, synthesis, scripting, optimization techniques, timing closure, timing convergence and QoR. What I have noticed about the industry today, is that it is focused on the wrong end of the development cycle. Fixing timing problems with better placement technology is more of a last ditch effort than a state of the art approach. Ultimately, if the problem can't be fixed with placement – the designer has to go back to their RTL code and re-write it to fix the problem. Code it correctly from the beginning, anticipating implementation roadblocks and barriers, and you won't need the big fancy tools to solve your timing convergence/closure problems later in the design cycle.

Introduction

Your success directly depends on your RTL description. What I have tried to do with this paper is talk about what is involved to produce a good RTL description of a design. It's not just writing with good style. I have put together my lessons learned, favorite coding styles, rules of thumb and optimization techniques, into a sort of guidebook for successful RTL synthesis.

1. Successful Synthesis

Engineers are always in a hurry. They want answers, but they don't want to spend time to figure out the answers. Synthesis is the transformation of an idea from one format (RTL netlist) into another (gate-level netlist) with optimization for improving speed or reducing area. A design has the same function before and after synthesis. Treating synthesis/Design Compiler like a black box without understanding what goes on inside is wrong. Your success with Design Compiler is directly related to your RTL code and how it interacts with DC. Spend time up front to understand how it all works together and then take advantage of that information to write a better RTL description of your design.

1.1 Pre-RTL Preparation Checklist

There are many design issues that need to be resolved before you begin coding your design. They can significantly impact the speed and area of the design and consequently the RTL description. Don't accept a design implementation decision if it is just based on legacy. If the specification doesn't call out a specific implementation – then you are free to argue for and code up a design that is better (faster & smaller), and hopefully, "synthesis friendly" (I will explain "synthesis friendly" later).

1.1.0 Communicate design issues with your team

Start with communication. Naming conventions for hierarchical blocks or signal naming conventions (such as block_function or noun_verb or verb_noun), or designating active states (i.e. active high or active low) for signals have to be worked out as a team. Revision control, directory trees and other design organizational issues must also be communicated team wide. Good communication between your team members is key to a successful design.

1.1.1 Do you have a specification for your design?

The specification is necessary to define the functionality of what you are designing – as well as the I/O, packaging, timing and a host of other important details. I take it for granted that everyone has a specification BEFORE they start coding. I list it here as just a good starting point into the guideline.

1.1.2 Does the spec define how the design is to be partitioned?

Partitioning helps break down the design into manageable chunks. It also helps if you have a team working on the design. You can assign different partitions of the design to different members of the team. Follow the specification's recommendations for partitioning. If it doesn't specify how to partition – then break the design down into major functional blocks. If there is only one major function – degenerate it into its sub-components.

1.1.3 Work from the outside – in: what are the I/O requirements for your design?

At the chip level's I/O pads, as well as at the major functional block level, it is important to define the interfaces as soon as possible. What bus interface protocol will be used? PCI? AGP? Will you use off-the-shelf IP to manage the interface? Get the specification for each bus and interface to the design before you begin coding. Make sure the function and timing of each one is clear. This will also enable you to create high level models of your design before you start coding the RTL. The high level models (a.k.a. Black Boxes) can be used to do a preliminary floorplan, which could help discover issues with the top level interconnect as well as define top level first pass constraints/timing budgets.

1.1.4 How are the buses defined? Bi-directional or unidirectional?

If it is not required by the bus specification, try to use unidirectional buses wherever possible. This is part of a synthesis-friendly design. This can also cause trouble in the ranks if in the past it has always been a bi-directional bus due to routing or other legacy considerations. The benefits of unidirectional buses are 1. Better visual flow through the design: from left to right 2. No extraneous timing paths (i.e. no false paths associated with the bus shorting inputs to outputs 3. Simplified synthesis constraints (don't have to specify as many false paths) and 4. Simplified control logic design of the signals driving data onto the bus – you don't have to worry about overlapping directions – just don't have the enables for the three-state drivers overlapping while driving the bus. The downside to unidirectional buses is routing. However, with the amount of routing resources available with today's technology libraries – it should not be a problem.

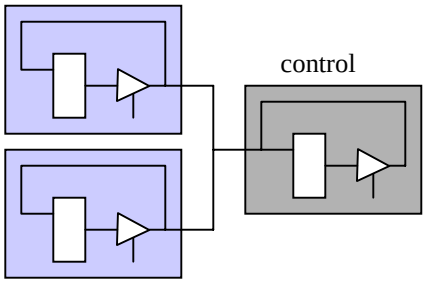

Fig 1: Bi-directional Bus - lots of false paths

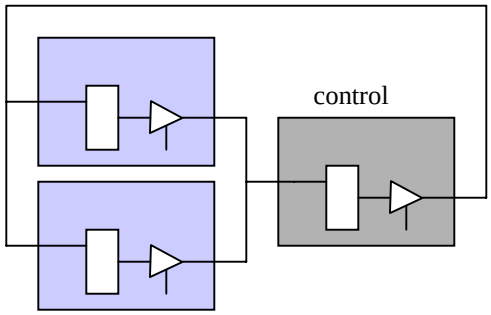

Fig 2: Unidirectional Bus - No more false paths!

If your design has bi-directional buses – try to keep out the bi-directional parts from your core and/or module level logic by promoting the bus interconnect to a level of hierarchy outside the particular design. This way the designers can have "clean" unidirectional internal hierarchies.

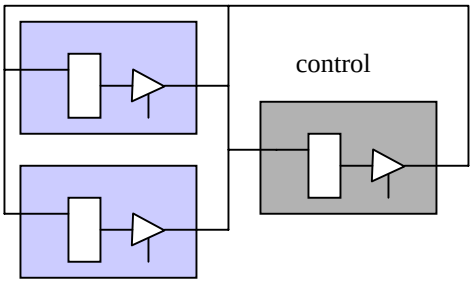

Fig 3: Bi-directional bus moved to outside module

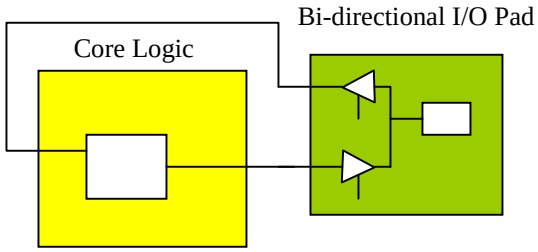

Fig 4: Bi-directional signals kept out of Core logic

1.1.5 Are there any latency/backward compatibility requirements?

Quite often, it is desired to have the new chip be 100% compatible with the previous version. That includes how it interacts with previous tester boards and software that were created to prototype the chip. Find out what legacy the chip has. If it is a brand new design – then you are free and clear – if not – then you need to get the tester's specification and understand how the chip is supposed to behave. It will also most likely affect the function of your I/O.

1.1.6 Testability: Full scan, no scan?

The ability to test for manufacturing defects (full scan) should be added to all designs. Full scan forces designs to be synchronous – which is very synthesis-friendly. How to check for the testability of a design's RTL description will be discussed in a later section.

1.1.7 What other IP are you using?

Does the design require any extra IP to be integrated into it? RAMs? Cores? Buses? FIFOs? Start with the interface to each IP block, define it and then move on to the function and what it is going to take to incorporate it into your design. At least you don't have to worry about synthesizing it (unless it is soft IP – RTL!).

1.1.8 Is it your expectation that you are pin-limited or gate limited?

Being pin-limited means that you don't have enough I/O pads in your package to do what you really want to do. You might be able to double up on the functions of each pin, which would require multiplexing signals and would prevent any ideas of a unidirectional bus interface at the I/O pad level. But if you need all the signals to be active simultaneously, you won't be able to do it. You'll have to split the design up. You should know before you begin your RTL coding what size package your chip is destined for. Check the specification.

Being gate-limited means that the design has too much functionality for the die size chosen. You might have to cut out functionality to fit on the die. Or you can try to optimize your design for area, which means speed objectives might be tough to meet. It is hard to estimate whether you will be gate limited at the beginning of a project unless you have been through this design before (with a different technology).

1.1.9 Is it your expectation that you will be pushing the speed envelope of the technology?

Just how much functionality are you putting into your design and at what speed will it be running at and with what technology are you going to use to implement it? Has it ever been done before? What changes to the design are you willing to make to achieve the speed goal for your design? Pipelining? Register retiming (partition_dp)? Duplicate logic cones? Just try to jam it all in? Knowing in advance any special timing issues would really help with your coding job. Single cycle multiplier? Serial chain of adders? Complex control logic? Large instruction decode section? It is the responsibility of the designer to know where the hot spots are in the design.

1.2 RTL Coding Style

My focus has always been on what's good for synthesis with little regard to the effect on simulation speed. Nowadays, with the advent of high-speed simulators, it is a moot point. These are my coding rules of thumb. Of course there are a lot more basic ones – I am just listing the

rules that tend to cause the most common errors. The next chapter discusses the structure implied by the coding styles and later in this paper, I will explain how different coding styles interact with the optimization process and how you can choose an optimization strategy based on your RTL coding style implied structure.

1.2.1 Create a block level drawing of your design before you begin coding.

Draw a block diagram of the functions and sub-functions of your design. Use it to guide you in coding up your design. It should define your inter-block partitioning as well as your intra-block constructs. Hey, it's useful for documentation too!

1.2.2 Always think of the poor guy who has to read your RTL code

I have seen too many disorganized RTL descriptions. Come on, let's make it easier for the RTL reviewer. Correlate "top to bottom" in the RTL description with "left to right" in the block diagram. Start with the inputs to your design (on the left side of block diagram) and describe the design's functionality from inputs to outputs. Don't try to be an ultra-efficient RTL coder. For example, you really might like for-loops, unfortunately, they only work best for certain types of logic (see below). And you don't get design awards for cramming the entire design into one file.

Please don't forget comments! Have a comment "header" for each module, comment the functionality of each I/O, and use comments generously throughout the design to explain the "tricky" parts. The reality to all this effort is that you might be the guy in the future who has to go back and reuse what you wrote 2 years ago!

1.2.3 Hierarchy

At the top level of your chip there should be 3, maybe 4 blocks: I/O pads, clock generator, JTAG logic (if you are using it) and the core design. They are in separate blocks because they might not be all synthesizable. Isolating them simplifies synthesis. Usually, the core design is hierarchical.

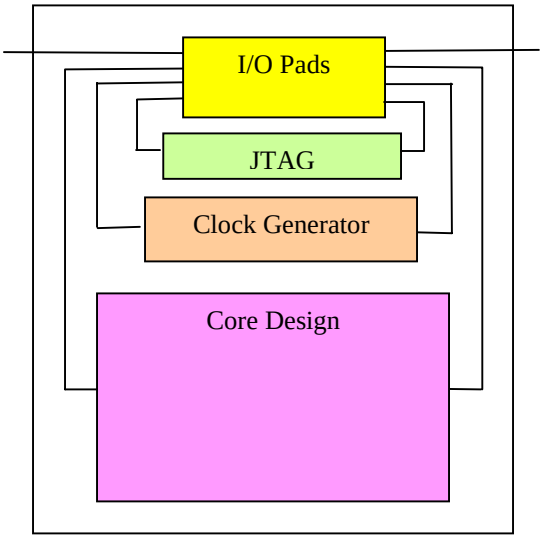

Fig 5: Chip-Level Hierarchy

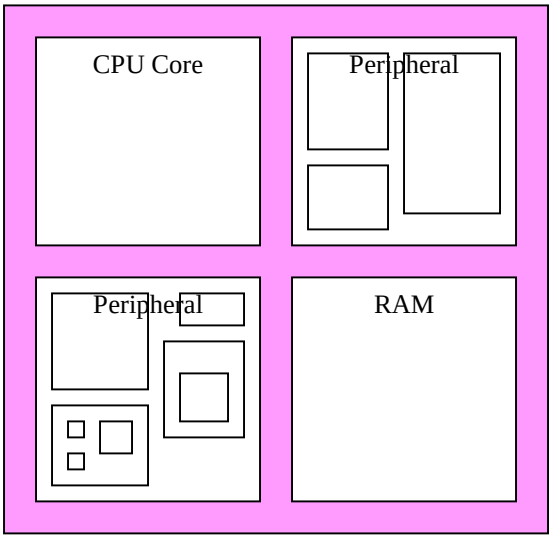

Fig 6: Example Core-Level Hierarchy

1.2.4 Use separate always@ processes for sequential logic and combinatorial logic

This used to be part of a religious war between synthesis and simulation users. It still might be – but I have always used it. It works. Three benefits – 1. It helps organize your RTL description (for example, you can easily describe a pipelined design). 2. There is a sequential optimization process in DC, which uses your coding style description of the sequential element to map it to the best sequential element in your technology library. When you combine sequential and combinatorial logic descriptions together, the tool can get confused and might not recognize the type of sequential element you are describing. 3. For synchronous-reset type designs – the synchronous reset signal is easily identified in case there are problems with it getting lost in a more complex (combined combo & sequential) description and the subsequent optimization.

1.2.5 Use blocking (=) for combinatorial logic and non-blocking (<=) for sequential logic

Another holdover from the synthesis vs. simulation wars. A compromise of sorts. Recognizes that combinatorial logic can be described with blocking statements without fear of losing implied structure. While sequential logic must be described by non-blocking assignments to prevent unanticipated loss of register stages due to the nature of a blocking statement (i.e. this is a "safe" coding style). The safest coding style is to use non-blocking assignments exclusively throughout your design – I will leave that decision up to you.

1.2.6 Know whether you have prioritized or parallel conditions

Designers like to use the "if " statement a lot. It is very popular. Unfortunately it can abused. Whether to use an "if" statement or not should depend on the select operator (the condition). If the conditions are mutually exclusive (i.e. parallel) then a case statement is better, because it is easier to read and it organizes the parallel states of the description. If multiple conditions can occur at the same time, use the "if" statement and prioritize the conditions (from highest to lowest or most often to least often) using "else if" for each subsequent condition. Priority encoded if statements infer a cascading multiplexer structure while parallel-encoded case statements infer a "tall and skinny" type of structure. Priority encoded muxes can impact timing due to their cascading structural nature. However, a priority mux can also be used to speed up a design by placing the late arriving operand and its condition at the top of the if –else-if statement – that way it travels through the least number of levels of logic. Know the type of conditions that exist so that you can use the proper construct.

1.2.7 Completely specify all branches of all conditional statements

A case statement, by default, in verilog, is prioritized from top to bottom. However, if you completely specify all possible combinations of 1's and 0's for the different cases (or use a default case) and you use the same select operator for all cases – DC will automatically recognize that case statement is fully specified and parallel (i.e. mutually exclusive). This is without any other user input (i.e. user doesn't need to use full_case or parallel_case directives). DC does this for if statements as well. As a rule, I try to completely specify all branches of all conditional statements. I challenge the designer not to use or rely on the "full_case" and "parallel_case" directives except when it is necessary. It forces the designer to completely think out his/her design (all possible cases). And it helps define all states of operation, so that the design never comes up from a power on sequence in an unknown (or unreachable) state (that would be bad). There will be designs where the directives have to be used. For example, one-hot muxes – they

use a different select operator for each state of the case statement and DC cannot determine whether the select operators are mutually exclusive (i.e. parallel) or completely specified (i.e. full case).

```
// no directives needed
case (a)
  2'b00 : c = x;
  2'b01 : c = y;
  2'b10 : c = z;
  2'b11 : c = w;
endcase
```

```
// directives needed
case (1'b1) // synopsys full_case parallel_case
  a : c = x;
  b : c = y;
  c : c = z;
  d : c = w;
endcase
```

Figure 7: Only use full_case & parallel_case directives when you have to.

When you elaborate a design into DC – you can check the log file to see if DC is properly interpreting the case statements. DC lists the line of code and whether it was able to automatically determine "full" or "parallel" case based on your description, or if you used directives it will list "user" for that particular case statement.

1.2.8 Initialize output of conditional statements prior to defining the statements

This goes along with fully specifying all branches of a conditional statement. Just in case you don't – initialize the output(s) of the conditional statement by assigning it (them) prior to the conditional statement. This is a safe coding style. Be careful selecting what value you initialize the output to. If there isn't a default state for that part of the design – then try to pick the "most popular" state to initialize the output to – that should help reduce extra switching (power) during operation.

1.2.9 Use high level constructs (case, if, always@) as much as possible.

Synthesis works best (i.e. it has the most freedom to try different optimizations) with high level RTL constructs. Low level gates or Boolean level constructs (verilog primitives) constrain DC. Later chapters will discuss how coding styles influence optimization. Also – don't spend a lot of time trying to force DC to implement a gate level solution by describing the logic with primitives. DC takes Boolean expressions and gate level instantiations and replaces them with a sum-of-products "pla"-like internal description that is fed to the gate level mapping optimization. You will never get exactly what you want and you will only frustrate yourself.

1.2.10 Don't instantiate gates unless you have to; make the code technology independent.

This goes along with the idea of getting exactly what you want. If it isn't a specification requirement or a design reuse situation – then don't instantiate standard cells in your design. You want portability – to be able to take the design description and synthesize it using any technology. If there are any instantiations in the code – they will have to be resolved (i.e. changed to match the new technology) before it can be synthesized into a new technology. Ugh. If it isn't necessary – then don't do it. A trick that works well is to put a "wrapper" around the gate level design. The wrapper enables a plug-n-play type swap out of different gate level implementations of the design while not interfering with the hierarchical description. The design

in the wrapper could also have a RTL description that could be swapped in if necessary (i.e. if no gate level description is available for a given library).

1.2.11 Use for-loops only for bit-wise operations that can only be described one bit at a time

For loops are really neat if you are a software designer. But for synthesis, they are very inefficient – they don't allow some high level optimizations to take place (resource sharing). For RTL synthesis, for-loops are "unrolled" during optimization – which means that the expressions within the for loop are duplicated for as many times as the index has iterations, with the index value inserted wherever it is used by the description. This can make it very difficult for the reviewer (remember him/her?) to understand what is going on – even though it can be a very efficient use of file space. I vote for ease of use.

However, for bit-wise operations that don't match up msb with msb, and lsb with lsb etc, use for-loops. Try not to put any resources like adders, incrementers or multipliers within the for-loops. Design Compiler doesn't perform resource level optimization within for-loops. Manipulate the operands without performing the operation - or else face multiple copies (one for each iteration of the for-loop) of the resource (adder, incrementer, etc.) in your synthesized design. Ouch!

1.3 RTL Structural Considerations

Using good coding style and writing "safe" RTL code is not enough! Understand what you are implying and figure out in advance where are the potential problems. You should be able to manually synthesize in your head what you have described in your RTL description – if you can't – then you are not done! KISS (Keep It Simple Stupid).

1.3.1 Break down the design elements into manageable chunks of RTL code.

It is really difficult to come up with a hard and fast rule about what is the proper size of a module.

Especially when you are coding up a new design for the first time and you don't know how big it is. At one time in its history, Synopsys said to set the max partition size to 5K gates. That caused all sorts of problems when Ambit came along and said it could do 30K gates at once. Soon it became a war of partition sizes (I think it topped out at 1 Million gates). Recently Kurt Kreutzer of UC Berkley [1] suggested physical partition sizes of 50K gates. But that doesn't mean that you can't have hierarchy within the 50K gate design – you can – it is just that you'll have to ungroup the synthesis hierarchy (a.k.a. the logical hierarchy) to satisfy the physical block size goal.

I look at the function being described. I also look at how to break the function down into component parts that I can understand and code individually. I know I can use the ungroup command when I bring my design into DC – so I don't worry about how many levels of hierarchy I use. Of course, too many and the design becomes unmanageable with too many files – and too few makes it unwieldy with too much information in a file. Rule of thumb: one module per file, and that logic description should describe the complete functionality of a design element in your block diagram (for example: I put the FSM in one module, the pipeline register in another and the interface logic in another for an instruction-pipe design).

1.3.2 Any Physical Requirements?

As mentioned above, do you have any physical issues with your design that could be manifested in the logical design? A block size target? Or the requirement to match your logical hierarchy with your physical hierarchy? Matching your logical hierarchy boundaries at some point in its hierarchy with your expected physical hierarchy boundaries will help with the proper selection of a wire load model and the wire load mode. However, for a lot of designs today, the state of the art in physical design is a completely flattened hierarchy – which would imply a flattened logical hierarchy if you were trying to match it.

A 1+ million-gate design with one level of hierarchy (i.e. flat) would have only one wire load model specified. Wire load models do not take into account localized clustering performed by the placement tools and the consequent reduction of the wire load at those spots. This discrepancy gets larger as you increase the design size. At 1+ million gates – you can't use a single wire load model. So keep your design with some hierarchy. Determine what wire load strategy you are going to use during synthesis – One wire load for the entire design? Or use the wire load mode "enclosed" to better approximate the clustering effects? Or use a custom wireload for each level of hierarchy (better idea) ? Or get rid of the wire load model (best idea) and use back-annotated parasitics from floorplanning or placement tools.

Other physical considerations could reflect specific physical locations of IP blocks such as RAMs or Cores in the layout of the chip. These blocks would not immediately impact the design process – they would be obstructions to the physical placement later in the cycle. It would be useful if they could be taken into account early in the design cycle to anticipate the problems they incur on the design. Floorplanning early in the design cycle would help anticipate their impact.

1.3.3 Registered outputs?

Where and at what level do you register outputs? Optimization is path based in DC. That means that only the cone of logic of a single path is considered during optimization (i.e. no simultaneous multi-path optimizations). Paths begin and end at registers (sequential cells, a.k.a. flip-flops) or at the ports of the design. The best optimization results are achieved when the entire cone of logic for a path is contained within the module undergoing optimization. Paths that "weave" in and out of several modules are called "snake" paths. They are more difficult to optimize since the boundaries of the different modules act as barriers to the optimization routines. By default, DC cannot change the polarity of port boundaries of a module or pass constants across port boundaries or change the number of ports that a module has (other than to add scan ports). Most of this can be overridden with the –boundary_optimization option to the compile command, which will allow optimization to transcend port boundaries. Or use the ungroup command, which removes the hierarchy and consequently the boundaries (no more barriers!).

Don't go crazy registering all outputs of every module. Rule of thumb: know at what level you are going to optimize at (compile the design) and make sure that at that level you have encompassed the entire cone of logic for each path and registered its output. Use "ungroup –flatten all" prior to optimization remove any lower levels of hierarchy (optimization barriers) and

you should get better results. If you have a choice of whether to include a path's "destination" register (endpoint), or a path's "source" register (startpoint), it is better to group a path's destination register with the cone of logic that describes the path. Two reasons 1. Sequential optimization has the ability to grab combinatorial logic in front of a flip-flop and map it along with the flip-flop into a more complex flip-flop from the library (for example: take a mux and a flip-flop and map it into a muxed flip-flop). 2. It is easier to describe the timing constraints of a sub-module with registered outputs vs. non-registered outputs.

1.3.4 Reset strategy: Synchronous vs. Asynchronous

Know what type of reset strategy you are going to use. Each has its pros and cons. This is another religious issue. I prefer asynchronous resets to synchronous ones. Early in DC's life there were problems with merging the synchronous reset signal in with the cloud of logic in front of a flip-flop. This would result in a simulation "lock-up" of unknowns as the Verilog simulator started each flip flop in an unknown state and they never came out of the unknown state. It was never really a DC problem – but DC got the blame. It is no longer an issue since DC has had the ability to recognize synchronous reset/set signals and put them as close to the inputs of the flip-flops as possible (no merging with regular functional logic). The ability to check for synchronous resets is activated by setting a variable to true: "hdlin_ff_always_sync_set_reset = true".

But I got used to describing asynchronous resets and the coding style stuck. The big thing you've got to watch out for is not when the asynchronous reset goes active, but rather when it goes inactive. As soon as the asynchronous reset goes inactive – the flip-flop is free to change state when the next clock edge occurs. If the reset signal is distributed throughout your design without care to how long the delay is on the reset network – you can have parts of the chip working in functional mode while other parts of the chip are still in reset mode. You've got to treat the asynchronous reset signal like a clock line and balance it so that the signal goes inactive within the same clock cycle throughout the entire chip. In case you don't plan to have a perfectly balanced reset signal – you can define the reset states for the flip-flops and the combinatorial logic sections to hold their values stable for "x" number of cycles or until the data inputs to the chip change (or some other equivalent "trigger" event occurs). What this means is that you design your default states or reset states such that they don't cause a change in the logic values downstream from their position in the logic cone when they are released (and free to change) from the reset state. This could also save power if the chip doesn't start switching as soon as the reset is released – but rather when there is work to do.

1.3.5 Design Reuse?

Design reuse can be the act of re-using a previous design from a previous project or it can be the act of creating a design to be used by the current project and/or future projects. Design reuse designs (IP) can occur in many different formats: Soft IP or soft macros describe the design using RTL. Firm IP or firm macros describe the design using a standard cell netlist. And hard IP or hard macros contain a finished placed & routed design described by a GDSII netlist.
If you are using a design reuse block – then instantiate it or the timing model given for it. If you are creating a design reuse block – be prepared to present the design in all the different formats including a timing model for the design.

1.3.6 Low Power design?

There are a couple of design styles that address low power design. The basic idea is to reduce the amount of switching (or logic transitions) going on in a design. One design method is to halve the clock speed while doubling the amount of concurrent operations in the design. The net effect is that the same amount of work is done, but with a slower clock – there are fewer clock transitions thus power is reduced.

Another design method, which is more popular, is to gate the clock and turn off sections of the design when they are not in use. It is called "gated-clock design". At Synopsys, we used to say, "Don't use clock gates", because Design Time couldn't measure the clock arrival time at a combinatorial gate (a.k.a. the clock gate) against a fixed setup or hold condition. It wasn't until the capability to check the clock at the clock gate was added that it was "ok" for the Synopsys DC users to gate clocks.

Decide whether you are going to use local clock gating (at the register) or global clock gating (at the module boundary or peripheral level). I would recommend putting global clock gating logic in a separate module outside of the functional design. By isolating it you can better constrain it and you can synthesize the core logic without the clock gating getting in the way. If you are going to use local clock gating – there is a DC friendly coding style that is pretty simple to write. See figure 8. DC creates the clock gates during elaboration, if the –clock_gate option is set. The clock gates are local to the registers they control, are already constrained to check setup and hold, and are fully testable due to a built-in override for the clock gating signal when under test (i.e. an active scan_enable causes the clock to bypass the clock gate and be available to shift data into the scan chain). Refer to the Power Compiler Reference Manual for more information.

```
module pipeline (…
...
always@(posedge clk or negedge reset)
begin
  if (reset == 1'b0)  // asynchronous reset
    q <= 32'd0;
  else
    if (clock_en)  // synchronous load enable
      q <= d;
end
…
endmodule
```

```
/* Initialization */
set_clock_gating_style -sequential latch \
                       - min  4 \
                       -control_point before \
                       - control_signal scan_enable
/* Run */
analyze pipeline.v
elaborate pipeline -gate_clock
include constraints.scr
propagate_constraints -gate_clock
set_clock_skew clk -propagate
compile
...
```

Fig. 8 Inferred clock gating coding style and corresponding dc_shell script modifications that go with it.

1.3.7 Muxes vs. Three-states for getting on a bus?

Full custom designers liked using tri-states whenever they came upon a bussed operation because they could split the three-states in half and distribute them throughout the design. With the advent of standard cell design (RTL based synthesis) this all changed. Three-states are quite big. For muxing multiple signals onto the same bus it is easier and smaller to use multiplexers. People will argue about the congestion this causes – but with greater number of metal layers available today – it is a lesser negative against muxes. I use muxes to get onto local buses within peripherals. But then I use three-states to drive data from one peripheral to another or to the CPU core.

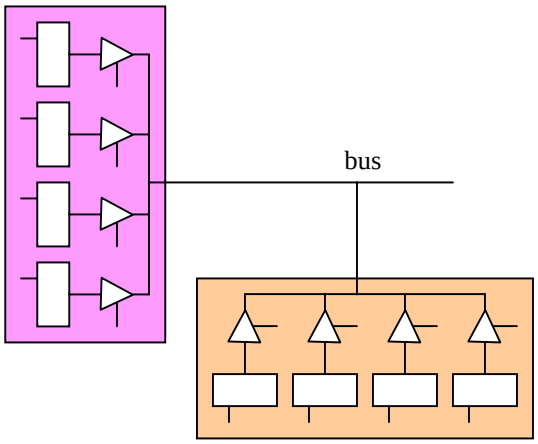

Fig 7: 3-State Muxing - that's a lot of 3-states!

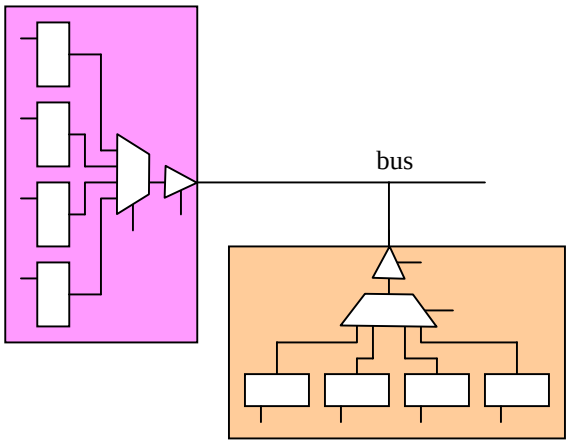

Fig 8: Using Muxes - Loading reduced

Testability becomes an issue with three-states: how do you test a bus for a stuck-at-1 or stuck-at-0 if it is in high impedance mode? Answer: You could put pull-up resistors on the bus, or you could control the enable pins on the three-states via the scan_enable. Or you could functionally test the bus. What about power? Is the bus transitioning a lot when it is not being used? A pull-up resistor could help there too.

1.3.8 How much functionality is being squeezed in between two registers? (Density)

"Physics dictates how much logic can be placed in between two registers and still meet timing"[2] Measuring or knowing the relative density of a path is going to help a lot in the synthesis of a design. As the responsible design engineer, you should have an idea of how much "work" you are describing for a given path. Watch out for single paths that "squirt" out of a module and go to other modules without being registered. The next guy in the path might assume that you already registered the path. This is how some snake paths form.

Balancing the distribution of functionality in a pipeline design is difficult but with some analysis it can be done. Know when to add pipeline stages: large single cycle multipliers, or complex designs that go back and forth between control and datapath. Use timing analysis to find out if you have positive slack on one side of a pipeline stage – then go back and rewrite the hdl to move the logic to the other side.

1.3.9 Where are the critical paths in your design?

Knowing the density, knowing the function and knowing the timing constraints should enable you to anticipate where the critical paths are going to be in the design. If not, explore. Do some design exploration (early synthesis & floorplanning) to find trouble spots in the design. You want to be confident that the design will operate at speed when you hand it off to the implementation folks (if it's not you who will be doing the synthesis).

1.3.10 Where are the multi-cycle / false paths in your design?

You are supposed to know your design. You should know where the false paths and multi-cycle paths are in your design. It is important to have them identified before you begin synthesis. If you don't know – then use design exploration to find out if you have any gross errors (candidates for multi-cycle paths), which can be allowed to remain multi-cycle in nature. Some designs have the requirement that all paths are single cycle. What are you going to do now? Add pipeline stages? Or clock enables? Gated clocks?

1.4 RTL Designed for Test

Full scan is based on the addition of a conditionally enabled path (scan chain), which sequentially connects each flip-flop in a design, effectively creating a huge shift register out of all the flip-flops. External scan ports are added to bring data into and out of the scan chain and control the operation of the scan chain during test from outside the chip. A design can have multiple scan chains.

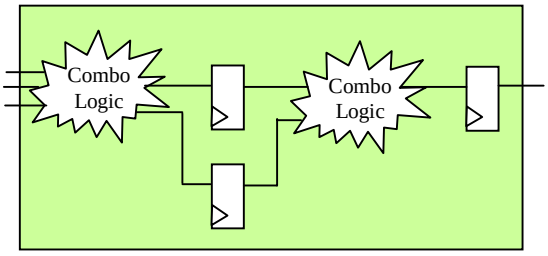

Fig 9: Example RTL design - no scan

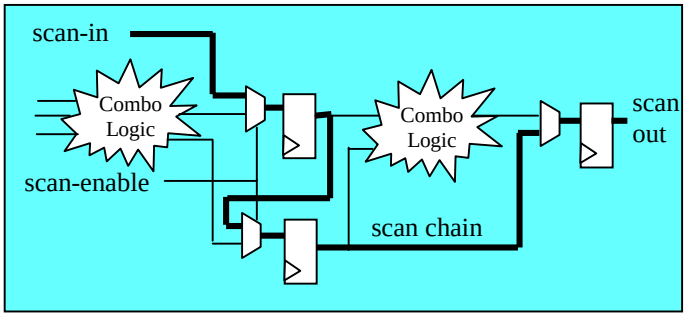

Fig 10: RTL design w/Full Scan

Full scan works by 1. Shifting a pattern of one's and zeroes into the scan chain - which initializes all the flip-flops in the design to a known value. 2. Next, enabling the functional clock for one cycle - which allows the known values to propagate through the functional cones of logic to the destination flip-flops where they are captured. 3. Finally, shifting out the captured result and checking the pattern against an expected result. Repeat until all possible combinations of results can be checked (100 % fault coverage).

Testability is all about the amount of control and observation of data in the scan chain. If there is any aspect of your design that interferes with the control and/or observation - it will decrease your ability to shift in data, capture it, and then shift it out. This means a decrease in fault coverage, which could mean that some manufacturing faults in your chip go undetected. The

goal of full scan is to get as close to 100 % fault coverage as possible. Therefore, we don't want anything to interfere with the test logic.
RTL doesn't have scan chains in it. How can we check it for testability? Answer: Look for anything in the design that would interfere with the control of the flip-flops or the observation of data from the flip-flops. You won't find all the testability errors in your RTL – but you can find some errors and more importantly, be aware of how to avoid coding testability problems into your RTL description.

1.4.1 Gating the reset signal? Internally generated only? Any outside the chip control?

The reset signal should not be controlled internally unless there is an external bypass signal available to override the internal controls during scan chain activation. (i.e. you don't want reset to go active during shifting data – it would erase or corrupt your scan pattern). Grep for the name of the reset signal and check its source.

1.4.2 Clocks: clock gates, muxing clocks with data or other clocks, internally generated?

Clocks need to be available (nothing stopping them) during scan chain operation. Even clock gates need to be activated during test. Check to make sure that all clock gates have an external control capability as well as no clocks are being used as data inputs to registers. If a design has multiple clock domains – what is the plan for test? The most common is to multiplex all the clocks to a single clock source during test. This has to be built into your RTL description. You also have to provide control to that mux so that the clocks can all be switched over during scan. Internally generated clocks need to be bypassed during test, so that an external clock can run the scan chain.

1.4.3 Three-State Drivers: How do you test for high impedance? Do they overlap?

Three-state drivers have to have known outputs during test. Full scan only checks for 1's and 0's. You can't test for high impedance. Use a pull-up resistor, which will drive the bus to a logic 1 when the three-state drivers are all disabled – which you can test for, but it does draw power. Or design the bus to be always driven by a logic value, which can use a lot of power if it is not often utilized. Or bite the bullet and functionally test the bus – removing the potential faults from the fault coverage list without adding risk to the design. If you have multiple three-state drivers on a bus, full-scan requires that they do not overlap while driving the bus. By design, this has to be correct. Best way to check is to make sure the control of any set three-states is from the same source. Independent sources can be difficult to guarantee that there is not overlap.

1.4.4 JTAG (Joint Test Action Group) (a.k.a. Boundary Scan)

JTAG is different from full scan. It wasn't created to test the internal design of the chip. Full scan works with all your internal registers. JTAG works with all your I/O pads. JTAG tests the interconnect between your chip I/O and the printed circuit board for faults (soldering mistakes, bent leads, open printed circuit board traces etc.) while the chip is "plugged in" on a working circuit board.

JTAG logic should be kept in a separate level of hierarchy. It usually consists of a TAP (test access port) controller (soft IP), a mux, an ID register and special I/O pads instantiated in the design. You will have to integrate the JTAG logic into your design description. Some designs call for the use of JTAG to test internal scan chains. This is not what it was meant for – but it is definitely possible to do. It requires a lot of JTAG and test knowledge to merge the test capabilities. I would recommend using the company's expert to help out. Good luck!

1.4.5 RAMs: Test plan for RAMs. BIST instantiated or created with RTL?

Does your design contain any RAMs? How are you going to test them? External ports? Ring of registers? Self contained BIST? Who is designing the BIST? RAMs and BIST might have to be integrated into your design. This is more of a heads up than a coding style issue that we can check for.

1.5 Pre-Design Compiler Optimization Preparations: The Pre-Compile Checklist

Early in my career at Synopsys, I was convinced that there was a DC "golden script" somewhere out there "in the field" (which means the AC staff), which would always produce the best-optimized design for any condition. I wanted to have it. So I got permission to hold a technical discussion with the AC's after hours during one of the sales conferences. The idea got a little attention and the discussion became a full blown, main event, "Speed Optimization Symposium" held right in the middle of the sales conference! (I guess other people wanted to know about the golden script too). I invited the most knowledgeable names from the field staff as well as corporate AE's, consultants and some R&D staffers to be panelists for a discussion of speed optimization techniques.

I asked each panelist to bring one slide describing their favorite optimization techniques for speed. What I got was a stack of slides per panelist with all their reasons and justifications used during synthesis. I was looking for a pattern to emerge from all the slides and ideas that we could possibly use as a baseline approach to optimization. But what came out of the discussion was totally unexpected. Instead of optimization techniques (and the golden script) the panel came up with a "Pre-Optimization Checklist"(a.k.a. The Pre-Compile Checklist). Each item on the list had to be satisfied before any customer was allowed to synthesize a design. Wow, the idea of keeping the customer away from DC was totally new. This symposium was successfully repeated and the discussions were expanded upon for the 1995 SNUG, as well as the 1995 Euro-SNUG meetings. The final results from the symposiums were used as a basis for the Chip Synthesis Workshop so that we could pass on the proper messages to our customers.

1.5.1 Good HDL Description of your RTL design

What you code is what you get. The optimization performed during synthesis is considered a NP-Hard (non-polynomial) type of problem: there is no clear one "best" solution; rather there are many possible "good" "so-so" and "not so good" solutions. Think of synthesis as refining (or polishing) what you started with (your RTL description). A good starting point yields good results while a poor starting point yields poor results. GIGO. This is the basis of my paper.

1.5.2 Good Partitioning within the RTL description of the design

Because module boundaries put restrictions on the optimization of paths that cross multiple hierarchical boundaries (snake paths) – it is best to have the complete cone of logic of a path to be present during optimization. If you have to partition a design, put your partition boundaries immediately following the registers (sequential elements).

1.5.3 Good Technology Library

A good technology library should have at least 4 power levels for each cell in the library – including registers. Library cells should include various "bubbled inputs" for logic cells (for example: a 2 input Nand gate with one bubbled input, a 3 input Nand gate with 1 & 2 bubbled input versions). The richer the library, the better the results during mapping. A library can affect QoR by as much as 20%.

1.5.4 Good Constraints defined

Completely constrain the I/O's of the design as well as the design itself. A minimum set of constraints: set_load, set_driving_cell, create_clock, set_clock_latency, set_clock_uncertainty, set_input_delay, set_output_delay, and set_wire_load_model. In addition, if the target library does not specify a default or per cell constraint for maximum capacitance and maximum transition use set_max_transition and set_max_capacitance. Likewise, if there are paths with exceptions to the single-cycle timing assumption, use set_false_path and set_multicycle_path. Check the timing budget defined by the constraints – don't over-constrain timing constraints by more than 20% of the clock period (you could be wasting time if you do).

1.5.5 Identify Multi-cycle and false paths

Know your design. Multi-cycle and false paths are decoys to synthesis. By default, DC assumes that all paths in a design are single cycle. If you don't identify the "timing exceptions", DC will spend all of its time trying to optimize the multi-cycle paths as single cycle paths. Ouch! By identifying the multi-cycle or false paths – you enable DC to focus on the real critical paths in the design.

1.5.6 Good Wire Load Model

Wire load models are part of a paradigm change in synthesis. It used to be the cell delays that were "way off" with regard to the actual cell delays reported by the P&R tool. Nowadays, it is the wire loads that have everyone concerned. As soon as possible, replace the foundry's "generic" wire load model with back-annotated parasitics from early floorplanning. During optimization, back annotated parasitics and delays are thrown out and the original wire load model is once again used to provide data for the wire's parasitics. Create a custom wire load model for each module from the back-annotated data so that you have a better timing convergence point. Otherwise – always try to do your timing analysis with back-annotated data.

1.5.7 Always start with default compile (i.e. compile –map_effort medium –scan)

Not really part of the checklist – but it was the only optimization step that everyone on the panel agreed with. The default compile is the same as "compile –map_effort medium -scan" with timing-driven-structuring activated. This sets a good baseline for future optimization. And if you can get it done (meet timing goals) with a default compile – good job!

1.6 Recommendations for Using Design Compiler

I spent most of my consulting career providing "design assistance" to customers. I was brought in whenever there was a synthesis problem and asked to find out what was wrong and clean up the problem. You would be surprised how many problems originated from simple little things not done or done incorrectly. I developed a series of rules for effectively using DC, which I affectionately call "How to be a Synopsys consultant". These are very simple – common sense rules that people just don't seem to want to follow. Please follow them.

1.6.1 Always read the log file.

For some engineers I don't think they know that there are log files. For others it seems they think: "If DC runs through the whole script and finishes – then there must be nothing important in the log file". A lot of my design assistance consisted of debugging and solving problems by simply reading the log file. You can grep for keywords "Unable" or "Error" in the log file. If there are none – that's good! Next look at and try to understand all of the "Warning" messages. If there are too many Warning messages of a particular type – and you know they don't represent a bad type of warning, use suppress_message to remove the warning from the display. Just don't forget that you removed it!

1.6.2 Always use analyze & elaborate – even for verilog files.

That is what the original designers of the front-end to Design Compiler intended. You can achieve the same goal (of bringing in your design to DC) with a read command but analyze & elaborate give you more capabilities and future enhancements will be toward elaboration. Analyze checks for syntax errors in your RTL descriptions and if there are none creates a pseudo code representation of your design. Elaboration takes the pseudo code representation of your design and builds a generic netlist in DC using generic technology cells (gtech) which represent the different functions in your RTL description. Elaboration allows you to pass/change parameters of parameterized designs. And Elaboration is where you invoke the clock gate generation option for localized clock gating. Another benefit is how the commands work. You analyze all your RTL files – but only elaborate the top-level module. Elaborate has a built-in link capability associated with the pseudo code representations. It goes out and grabs only the associated blocks that it needs.

There is a new command: "acs_read_hdl" which you pass the top level design name and the directory where all the RTL files are stored and the command builds everything automatically. Check it out. It could replace analyze & elaborate.

1.6.3 Always use link to make sure nothing is missing

Here is another command that is not required – so people tend not to use it. Yet, it solves a lot of missing files and incorrectly chosen file problems. Link pulls into DC any modules that are missing which have been saved by their module name into a db file. Link uses the search_path directories to find the missing parts (so make sure your search_path variable is well defined). Link is a utility command and should always be executed during an interactive session or within a batch script. Don't forget to grep for "Unable" – which lists any module link can't find.

1.6.4 Always use check_design and check_timing and understand the warning messages.

Two more commands that are optional and subsequently not used very often. Check_design looks for RTL netlist problems with your design. It reports unconnected ports, multiple drivers driving a signal, mismatched port sizes between blocks etc. It would behoove you to study all the messages from the check_design command to prepare in case it finds something wrong. Check_timing checks to make sure all timing paths in the design are constrained. Many mistakes with DC involve not having the constraints properly applied.

1.6.6 Always start with a default compile

Often designers waste a lot of time waiting for results from a compile that didn't have to go that long. Always starting with "compile -map_effort medium -scan" is the best way to go.

1.6.7 Always save the design with the –hierarchy option

Design Compiler will save empty blocks that represent lower level hierarchical branches of the current design if you write out the design without the –hierarchy option. It can be frustrating if you are trying to re-assemble a design later and the lower level blocks are missing – you will have to go find them and hope that the I/O definitions haven't changed. Think of the write command with the –hierarchy option as taking a "snapshot" of your entire design. Warning: it can use up a lot of disk space.

1.6.8 Always put constraints into a separate script file and call it from your main script

Design Compiler doesn't use the same control commands as Primetime – but it does use the same constraints. By putting the constraints into a separate script file, the user can easily isolate and convert the constraints into Tcl format later, when working with Primetime.

1.6.9 Use –syntax_check & –context_check prior to running a script for the first time

This is just common sense. How many times have you written a big fancy script and kicked it off as you left for the day – only to come back the next day and find that the script ran for 15 minutes after you left and then quit due to a syntax error? Or a misspelled object? Neither option will actually perform synthesis, but they do run through your scripts. By the way, these features – syntax_check and –context_check don't exist for the Tcl version of DC (yet).

"dc_shell –f run.scr –syntax_check" will check for syntax errors in the commands of your script
"dc_shell –f run.scr –context_check" will make sure that all named objects called out by "run.scr" exist and it will check for syntax errors in the commands too.

1.6.10 Fix all combinatorial feedback loops reported by DC.

Automatically, DC breaks timing loops for timing analysis and optimization. DC chooses what it considers to be the best spot to break the loop. You should decide where to break the timing loop. Use check_design to list the cells in the feedback loop so that you can fix it. Sometimes timing loops hide critical paths with large negative slacks. Other times, it is just bad coding style and no harm has been done.

1.7 Under the Hood of Compile

Optimization within DC can be broken down into three major processes: high-level optimization, logic-level optimization and gate-level optimization. All three levels of optimization contribute to the successful optimization of a design. Consequently, if any of the three levels "fail" (i.e. produce a less-than-optimal result) the chances for synthesis success (i.e. meeting timing goals) diminish.

1.7.1 High-Level Optimization (a.k.a. Resource-Level Optimization):

High-level optimization manipulates the structure (architecture) of the design at the DesignWare component level (resource level – big chunks of logic). The rest of the design is kept technology independent (mapped to generic technology cells) during HLO. DesignWare components represent adders, multipliers, and other resources inferred by operators (+, - , <, =) in your RTL code. DC performs three types of high-level optimizations: resource sharing, common sub-expression sharing and implementation selection.

Resource sharing: 1. determines whether operators listed in separate lines of code can be shared and 2. determines whether it is a good idea to share them – i.e. does it improve timing? If it doesn't cause the timing to get worse – sharing will happen because it also improves the area of a design. Some conditions of resource sharing: 1. The operators must be in expressions that are part of a conditional branch of a conditional RTL construct – like a case or if statement – the conditional branches must be mutually exclusive and must be within the same always@ process statement. 2. No sharing between different conditional constructs. (for example: no sharing between multiple case statements). 3. Timing constraints should be specified. If no timing constraints are specified, the design is optimized for area and resource sharing is used because it reduces the number of resources in a design thus improving area.

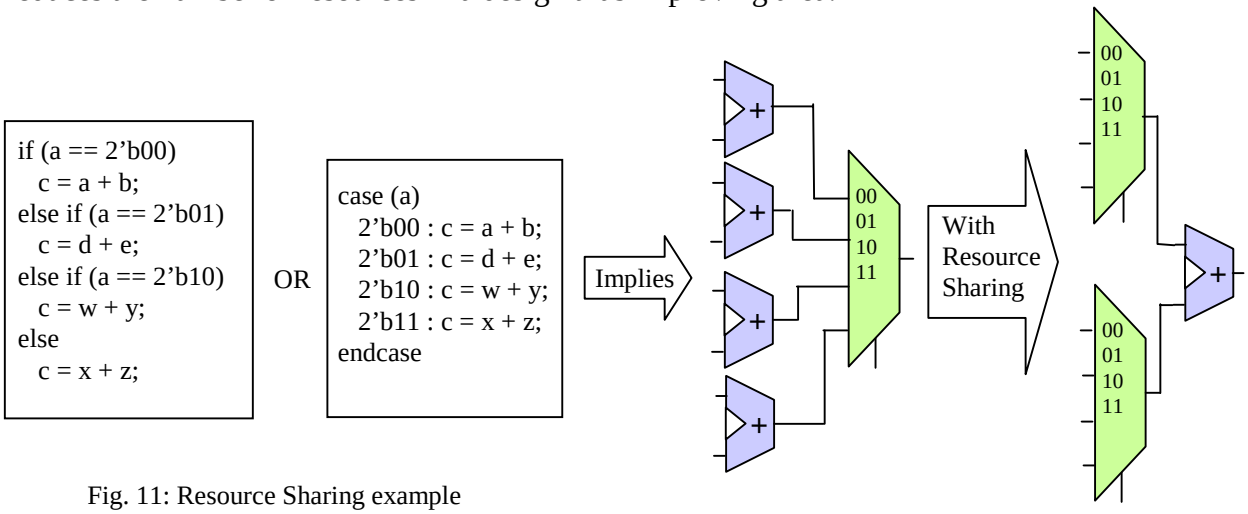

Fig. 11: Resource Sharing example

Common sub-expression sharing: 1. determines whether there is a possibility to share duplicate sub-expressions (consisting of a pair of operands and an DW operator) from multiple assignment statements. 2. determines whether sharing would improve the timing of the design. Conditions of common sub-expression sharing: 1. the operands must be in the same order for each of the expressions. For example: y = a + b + c; y = a + b + d; can be shared but y = a + b + c; y = b + a + d; can't. 2. Parentheses can be used to help the tool recognize common sub-expressions. 3. Timing constraints should be specified for the design. Various degrees of sharing (sharing amongst more than just 2 expressions) can be realized depending on the timing constraints. If no constraints are specified, sharing is used to reduce the number of resources thus improving area optimization.

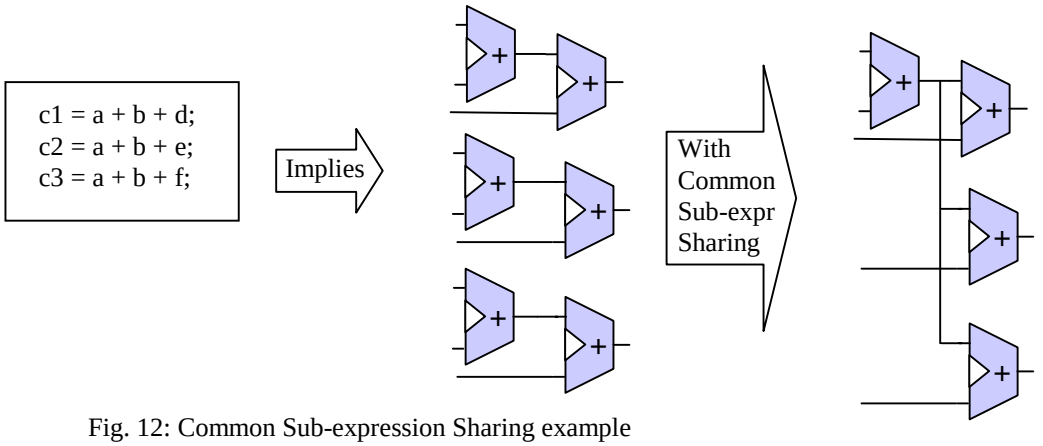

Fig. 12: Common Sub-expression Sharing example

Implementation selection decides what is the best architectural implementation for a given operator. There are several architectures in the DesignWare library for each operator type. Examples are Carry-Look-Ahead adders, Ripple-Carry adders, Booth multipliers, and Wallace Tree multipliers. Implementation selection builds a gate level netlist for each implementation of an operator, analyzes each netlist for area and speed, then chooses the best one based upon which implementation is the smallest – yet can still meet the timing goals. Conditions for implementation selection: 1. Timing constraints should be specified for the design. If no timing constraints are specified the tool defaults to an optimal area based implementation selection (i.e. chooses smallest architectural implementation).

1.7.2 Logic-Level Optimization (a.k.a. Boolean-Level Optimization):

Logic-level optimization manipulates the structure of the design at the Boolean level (gtech level – generic cell-sized chunks of logic). The design is kept technology independent as its generic technology (gtech) representation is shaped into various structures. Flattening and structuring are the two major processes that go on during logic-level optimization.

Flattening and structuring are the yin and yang of optimization. Structuring builds a network of intermediate terms which is optimized for speed or area. Flattening attempts to reduce all paths (i.e. eliminate all intermediate terms) to a sum-of-products representation regardless of the constraints. By default, flattening is disabled and structuring is enabled. Timing constraints should be specified otherwise structuring defaults to an area based optimization result.

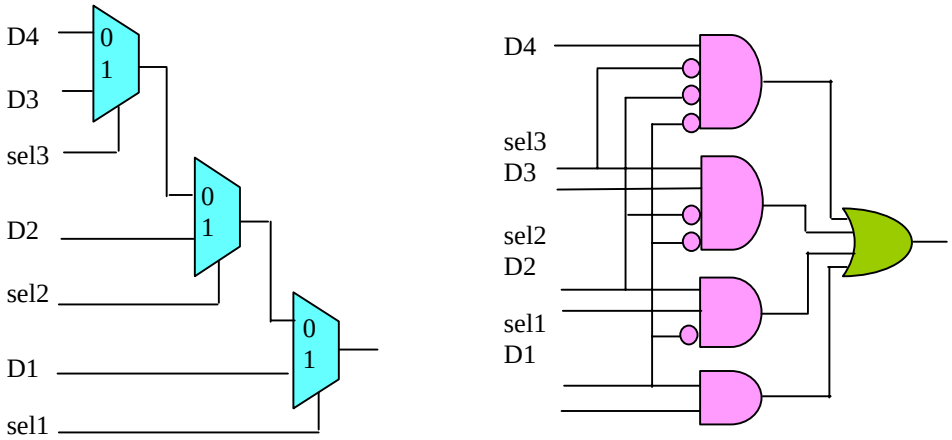

Fig. 13 A structured design vs. a flattened design.

1.7.3 Gate Level Optimization (a.k.a. Mapping):

Gate-level optimization starts with a structure (architecture) defined by generic technology cells handed down from logic-level optimization and replaces the gtech cells with standard cells from the target technology library. Mapping, sequential optimization, design rule fixing are the major processes of gate-level optimization.

Mapping degenerates (breaks down) all of the combinatorial logic paths in the design into a netlist of inverters and 2 input nand gates (any synthesizable function in a design can be represented with only inverters and 2 input nands – try it). Next, mapping pairs-up different combinations of the inverters and nand gates and attempts to find the identical functionality in a cell from the technology library – if it matches, the nand and inverters are replaced by the library cell. This process of mapping takes many iterations and there are many different algorithms that are used to pair up cells and find the best matching candidate from the library (based on area, speed and power).

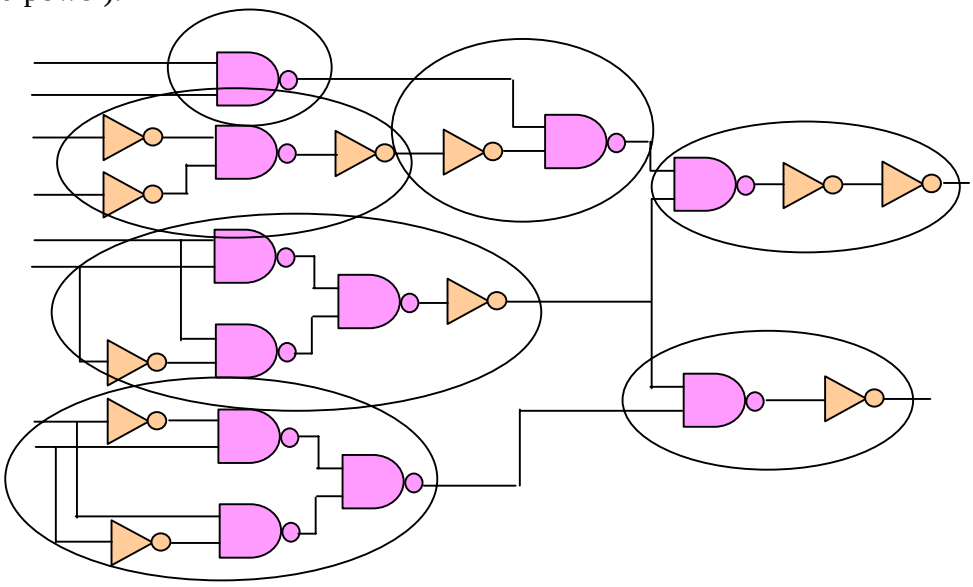

Fig. 14 Example: Possible gate-level grouping of nand gates and inverters for mapping.

Sequential optimization maps generic flip-flops and latches to technology specific cells from the target library. The functionality of the generic flip-flops is defined by the sequential logic description in your RTL. Flip-flops with synchronous or asynchronous resets and sets, multiplexed flip-flops, clock enabled flip-flops and latches are some of the different types of sequential devices that are inferable from your RTL description. Complex flip-flops can be instantiated in your RTL description if needed in your design, but they won't be sequentially optimized unless their function statement is understood by Library Compiler (check the attributes of a cell – if it is "black box" – then LC doesn't understand its function statement – or it is missing a function statement). Sequential optimization can take combinatorial cells in front of or behind a simple flip-flop and merge then into a more complex flip-flop. For example: a mux and a "D" flip-flop can be merged into a Muxed D-Flip-Flop. This assumes that such a cell exists in the target library – if it doesn't – the cells remain as is. Sequential optimization relies on clearly written RTL that infers the functionality of the flip-flop to be mapped to and constraints that help define drive and speed of the cell.

Design rule fixing – has the highest priority over all optimization. If the design doesn't meet a foundry's design rule – then the foundry doesn't have to accept the netlist from the customer. There are three rules that design rule fixing considers: maximum fanout, maximum capacitance and maximum transition. These rules are usually tied to a target library cell's description – but they can also be set by the user to override the values set by the library. They can never be set to a lesser value (easier to satisfy) but they can be set to a greater value (more difficult to satisfy) than what was defined in the target library. Sometimes libraries don't have design rules specified. With deep sub-micron designs – all libraries should have at least 2 out of 3 design rules specified.

Fixing design rule violations entails the use of buffers and cells with different drive strengths. Fanout violations are fixed by adding a buffer and splitting the signal into two or more paths. Transition violations are fixed by increasing the drive strength of a cell or adding a buffer with a large drive capability. Capacitance violations are fixed by adding a buffer and splitting the signal path into two or more paths.

1.7.4 Compile –map_effort low/med/high –incremental

The state of the design (mapped or gtech) and the -map_effort/ -incremental options of the compile command determine which of the different types of optimizations are activated by the compile command. See the chart on the next page to understand how they are related. The state of the design is considered "gtech" if it has not yet been synthesized to gates. Pre-"compiled" RTL descriptions are classified as gtech. Once a design is mapped to gates (as an outcome of the compile command) it is no longer eligible for high level optimization (with the exception of implementation selection), but can be re-compiled (experience logic-level and gate-level optimizations) as many times as the user wants. The -incremental option to the compile command is considered primarily a gate-level optimization process – but it does perform implementation selection (HLO) and CPR: critical path re-synthesis (logic-level optimization) as well. CPR is a localized re-structuring and re-mapping optimization of just the logic in the critical path.

Chart 1: Optimizations vs. compile –map_effort level (Y= Performed, N= Not Performed)

Optimization Technique	Low Effort Compile		Medium Effort Compile		High Effort Compile		Incremental Compile	
State of design	Gtech	Mapped	Gtech	Mapped	Gtech	Mapped	Gtech	Mapped
Common Sub-expr Sharing	N	N	Y	N	Y	N	N	N
ResourceSharing	N	N	Y	N	Y	N	N	N
Implementation Selection	N	N	Y	Y	Y	Y	Y	Y
Flattening	Y	Y	Y	Y	Y	Y	N	N
Structuring	Y	Y	Y	Y	Y	Y	N	N
Mapping	Y	Y	Y	Y	Y	Y	Y	Y
Sequential Optimization	N	N	Y	Y	Y	Y	Y	Y
Design Rule Fixing	Y	Y	Y	Y	Y	Y	Y	Y
Critical Path Re-synthesis	N	N	N	N	Y	Y	Y	Y

1.8 Design Compiler Optimization Strategies

"The random setting of optimization switches is NOT a valid optimization strategy"[2]. Why do some optimization commands work, while other times they don't? In the previous chapter we learned about the optimization process and how it is affected by RTL coding style. There is a direct relationship between coding style and successful optimization strategies. If you don't treat synthesis as a black box process – and you study how RTL architectures are affected by the different optimization commands, you can determine what is the best optimization strategy based on what type of architecture you are implying with your RTL. In this section I am going to match up different types of RTL structures with the optimization strategies that work best and explain why. The results I present are based on my own observations, from my personal optimization experiences, your results may vary. As I use to say at Synopsys, "It depends".

1.8.2 RTL structural classifications:

There is more work needed to create additional types of RTL architectural "structures" or shapes. For this paper I have identified only six types of RTL structures: instruction decode, random logic, datapath logic, control logic, finite state machine (FSM), and DesignWare (DW). It is possible to break down the six types further and add additional types.

1.8.3 set_flatten

The goal of set_flatten is to reduce the combinatorial logic in the paths of the design into a sum-of-products structure irregardless of the timing (or area) constraints. set_flatten can be very area inefficient. Any intermediate term or complex function will be reduced to a sum-of-product format (two levels of logic). set_flatten success depends on the technology library. When the sum or the product terms size exceeds the largest gate width in the library – DC will map to multiple levels of gates to implement the needed wide gate – losing the effectiveness of being able to represent the design in just two levels of logic. set_flatten has three options: -single_output, -multiple_output and –phase. set_flatten by default is disabled.

set_flatten has an effort level. By default, when the command is enabled it is set to low effort. Low effort should be good enough for random logic. Medium effort is for logic structures that might be a little more complicated and require more time to reduce them to a sop form. High effort should never be used unless you can guarantee that you don't have any complex structures in your design (adders, exclusive-ors, intentional structuring) because it doesn't stop flattening a design until all parts of the design are reduced to a sop form (or it runs out of memory). If you are using the command on the approved type of structure – you should never need to use the high effort.

set_flatten –single_output produces a sum-of-product structure which has separate cones of logic for each path. set_flatten –multiple_output produces a sum-of-product structure which has interwoven cones of logic – sharing product terms whenever possible. set_flatten –phase inverts the Karnaugh map of the logic in a path and tries to define the zero product terms (instead of the true or one product terms) followed by an inverter to restore the original function.

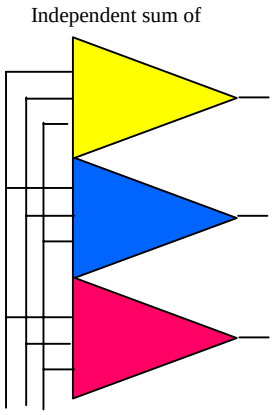

Fig. 15 set_flatten -single

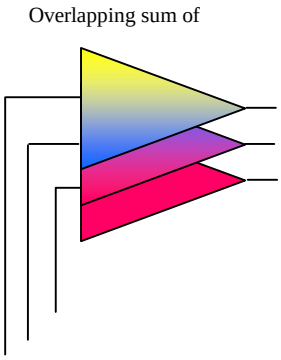

Fig. 16 set_flatten -multiple

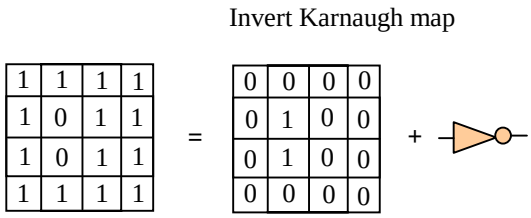

Fig 17 set_flatten -phase

set_flatten works best with small blocks of random logic, control logic and FSM's. If there are too many inputs – the product terms and the input fanout gets bulky (and slow), if there are too many outputs in the design – that is ok. set_flatten should never be used where there are complex structures in the design – like adders, comparators or exclusive-or trees. Instruction decode – while usually a random type of logic description – has too many input combinations to be "flattened" and it gets very area expensive – without a big pay out in speed.

1.8.4 set_structure

The set_structure command by default, is enabled with a -map_effort medium or higher compile command. It is the true workhorse of the compile command. set_structure works for both area and speed optimizations. set_structure has two options: -boolean and –timing_driven.

set_structure –boolean is disabled by default. It meant to be used for area based optimizations. It uses special area reducing algorithms (DeMorgan's law, minimize product terms). set_structure –timing_driven is enabled by default. It tries to create a structure that best fits the timing and area demands.

set_structure works best with all types of design. It is the one universal optimization command. It can be used with the set_flatten command but that would only be with random logic, control logic, FSM's and maybe with instruction decode logic. What happens is that the structure of the circuit is flattened first by set_flatten, then the set_structure command attempts to install intermediate terms and structure to meet the timing goals. Limit the set_flatten-set_structure strategy to small designs – otherwise it could take a long time to compile and use up a lot of area.

1.8.5 set_critical_range

During compile, after the design has gone through HLO, LLO, and one pass of gate level mapping, the worst case path (critical path) becomes the focus of the gate level optimization routines. If the design has no paths that violate timing goals – then optimization stops. If however, there are paths that violate the timing goal – the critical path (the worst path) is selected, and is optimized until it is no longer the worst path or the tool comes to a point of diminishing returns (i.e. can't fix the path) and then optimization stops. If the critical path is no longer the most critical path in the design, it is "set down" and a new critical path becomes the focus of the optimization routines until it is improved or the tool comes to a point of diminishing returns. This cycle continues until all paths meet timing or the tool gets "stuck" on a particular path and can longer improve it – optimization stops without considering any of the "sub-critical" paths. In summary, DC only works on the one critical path in a design (per clock domain) at a time.

The set_critical_range command allows the user to specify a range of time measured relative to the critical path's timing. All paths that fall within that range of time will be optimized. Even if the critical path cannot be improved – other paths in the design that are within that range of time will be optimized. Warning! This will increase compile times dramatically.

Critical range is very effective when optimizing logic designs that are densely packed and "intra-related". Intra-related means that the paths in the design share many intermediate terms. If you were to look at the critical path and the sub-critical paths – you would notice that they share common parts. This type of logic works best with critical_range because critical_range works on the worst case path (critical path) as well as the paths that fall within a range of time relative to the critical path. Because the logic paths share common parts with the critical path, often times you see a collapse (improvement) to the critical path and as well as the sub-critical paths. The types of RTL structures that work best with critical_range are instruction decode logic, random logic, FSM logic, and control logic.

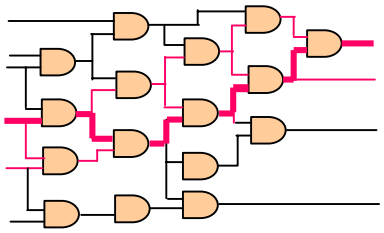

Fig. 18 Intra-related logic - candidate for critical_range

1.8.6 Simple Compile Mode

Simple compile mode started off as a scripted optimization strategy but was incorporated into DC as a feature that must be invoked prior to the compile command to work. It has changed since I left Synopsys (improved I'm told) so some I might be leaving off some features. But its purpose was to speed up compiles and improve quality of results as Synopsys battled Ambit in several benchmarks. Its features a bottom up compile strategy but it is activated at the top (i.e. you set your current design to the top-level design and it does the rest). Simple compile mode has a auto-uniquify ability (it will uniquify multiple instantiations in your design – you don't have to worry about them) and area-based HLO. It works. Its fast. And even though it was meant for slower designs – it has done a better than expected job with speed critical designs.

Simple compile mode can be used with ALL types of designs. Because it uses area based HLO – I would be leery of the implementations selected – check them – see if they are too slow. Simple compile mode has to be set before the compile and should be de-activated immediately after. The following command sequence can be used to activate/de-activate the mode.

```
set_simple_compile_mode true
compile –scan
set_simple_compile_mode false.
```

1.8.7 DC Ultra

DC Ultra is a license that you must purchase, which gives you more options when optimizing a design. DC Ultra enables pipeline retiming, special optimization algorithms and DesignWare Foundation components.

Pipeline retiming (partition_dp command) has the ability to break up and re-assemble functional logic across register boundaries – effectively "moving" the registers into the logic cloud. It ungroups DW parts that are in the design. Moving the flip-flops would enable you to balance each stage of a pipelined design – assuming that the different stages have uneven amounts of density (amount of functionality per stage). There are some drawbacks – formal verification has to be done since functional cones of logic are changed. And there is no way to back annotate your RTL with the changes made by pipeline retiming. Pipeline retiming is a neat way to create pipelined DesignWare components (a 5 stage multiplier) – and it could be very effective if all else fails to improve the speed through a datapath type design.

The special optimization algorithms include the ability to duplicate fan-in cones of logic for sub-critical paths to reduce the fanout of the critical path. And the Ultra algorithms enable better mapping to a wider variety of library cells (high fan-in gates). This could be very helpful in all types of designs.

1.8.8 DesignWare License

The DesignWare license offers more varieties of macros and faster architectural implementations of operators to the user. This enables the user to fix critical paths by selecting faster multi-gate components – which will have a greater effect than just tweaking individual standard cells. It does put a restriction on a design that it must always check out a DW license whenever DC manipulates it. But the benefits outweigh the inconvenience. If you have a DesignWare license – use it. To enable the DesignWare license you must list it in your synthetic_library variable as follows:

synthetic_library = dw_foundation.sldb (dc_shell-t: set synthetic_library dw_foundation.sldb)

1.8.9 Post Compile Checklist: Analyzing results

Start with a default (map_effort medium) compile. Analyze the results. This is your baseline. You want to look at the timing and the structure of the result. The following is a list of questions I came up with to use when analyzing a design. They are supposed to help you figure out what to do next. If I can't figure out what to do next – and yet my design is not meeting timing – I perform an incremental compile with map_effort high to see if that helps the critical path. By the way, whenever you re-compile a design always use map_effort high – whether you use –incremental_mapping or not depends on the problem. Whenever you make changes to the RTL code – re-start the synthesis process with a default compile. You always want to compare to your baseline to make sure you are making improvements to the design. You can take shortcuts if you have a large design, such as only re-starting those parts of the hierarchy that contain the changes. However, the result might not be the best since you are not giving DC the most latitude to change things with the new code.

FYI: when I say "fix" in the questions below, I mean to say, "can you re-code the RTL to fix the problem?" I never re-compile a design more than 3 times if it isn't getting any better. I try to dig into the design and localize the problem paths (and corresponding areas in the RTL description).

1. Where is the critical path in the design? Is this where you were expecting it to be? Is it obvious why it isn't meeting timing? Can you fix it easily? Can you isolate the path or is it part of a large cloud of logic? Use the group command to isolate the critical path, characterize the new level of hierarchy, compile it with –incremental –map_effort high, then ungroup it and repeat the analysis of the design at the top-level.

2. How big is the negative slack amount?
>100% of the clock period – big problem – continue analysis,
99% to 50% of the clock period - difficult – continue analysis,
49% to 25% of the clock period - might require something more than just an incremental compile – continue analysis,
24% to 1% - try compile –map_effort high –incremental.

3. How many paths in your design violate timing and by how much? This is a quality of result measurement and a "how much you got left to do" measurement. If there are a lot of paths that violating timing(>50%), with large negative slacks – there could be a fundamental architectural problem with the design. Are the critical timing paths related at all? (i.e. sharing intermediate terms?) If yes – then you might be able to focus and improve those paths with a critical_range. If no – then it is a larger problem - continue analysis. If there are not that many violating paths in the design – continue analysis but don't worry so much.

4. Do they share any startpoints or endpoints with each other? Could be a candidate for a critical_range. Do all the violating timing paths share the same startpoint and endpoint – that could make it easier to isolate the problem. Do none of the violating timing paths share any startpoints or endpoints ? Could be an over constrained design. Check constraints. Work on each path separately. Continue analysis.

5. Is there any large single item delays in the critical path?
Always look for the "biggest bang for the buck". Fix the big delay and you might not need to do anything else.

6. How many DW components are there in the critical path?
Are there just too many in one path? Should you consider changing the structure?

7. What are the DW implementations that were selected?
Easy way to speed up design – make sure the fastest component is being used in the critical path – no ripple carry adders please. You can use the set_implementation command to specify which implementation to use for a DW part. Or you can "set_dont_use" particular implementations in the DW library – but that will affect all levels of hierarchy – and there might be some parts of the design where you don't care if there is a small and slow adder. The other option is to instantiate a DW part in your RTL. See the DW manual in the Synopsys Online Docs to find out how.

8. Is the critical path a control signal or a datapath signal or both?
Recognizing what type of signal the critical path is can help you decide where and what in your code you can change to improve the speed of the design, or, if the design is a possible candidate for the Ultra optimization.

9. Does the critical path cross multiple levels of hierarchy?
Is it a snake path? Snake paths are difficult to optimization one piece at a time. You have to have a good timing budget – i.e. good constraints. Snake paths might prevent the optimal solution because of the hierarchical boundaries it crosses. You can get rid of the snake path by ungrouping the levels of hierarchy which contain it or if that yields too large a design – you can go back and re-write the RTL to include the snake path function in just one module.

10. Are there any design rule violations?
There shouldn't be. Design rule violations could indicate implied dont_touches have been applied to the design. Or some hierarchical boundary is causing a design rule violation to remain. Use report_attributes –all to see if any dont_touches exist. Use report_constraints –all_violators –verbose to get a detailed listing of where the design rule violations are in the design.

11. Are there any dont_touch attributes used in the design?
Did you forget to remove a dont_touch attribute? Or is it intentionally left there? Can you remove them? Remove them and execute a new compile.

12. Can re-arranging the datapath logic eliminate the critical path?
Lots of times if the critical path is in the ALU or other datapath portion of the design – you might see how the big blocks of logic (DesignWare) can be re-arranged to speed up the design.

13. Can re-structuring the design eliminate the critical path?
Is there something obvious that can be done to the critical path to reduce the amount of logic in the path? Priority encoders when there don't need to be. Muxes that could be re-arranged to change the depth of a critical path. Is there a flag that doesn't need to be a single cycle path?

14. Is the design "tall and skinny" or "long and stretched out"?
Is there room for further optimization (long and stretched out) or has everything been done that can be done? Could it be a candidate for DC Ultra? or compile –map_effort high –incr?

15. How many levels of logic does the critical path have?
Are you trying to squeeze 100 levels of logic in 3 ns? What is the density of the critical path? This sets expectations about how much work is left to do and is it possible with the current RTL description.

16. What does the fanout along the critical path look like? Is there a high degree of fanout throughout the path? Could consider using DC Ultra. Or does the logic in the critical path have only one fanout per cell? - That could indicate a highly optimized path. Which means that you will have to do something more dramatic (i.e. recode RTL) to improve timing.

17. What wireload model mode are you using? The mode relates to which wire load model is used for different levels of hierarchy that don't have a wire load model already specified. Many times the wire load model chosen for a level of hierarchy is too optimistic and gives a false positive. Check the wire load models and the mode in the timing report – it will show what they are set to. Verify that the wire load model chosen for a given module is appropriate and follows your physical implementation strategy. Otherwise – change the wire load model selection.

1.9 Timing Convergence, Timing closure and RTL

In the beginning, synthesis and place & route were separate, independent processes. Synthesized netlists were "thrown over the wall" to the place and route tools. Sometimes the place & route tools could NOT rout the synthesized netlists, or they could NOT achieve the same target speed that they could in Design Compiler. The designs would be "thrown back over the wall" to the responsible design engineer with instructions to try something else (i.e. recode the RTL). Later, it was found that the designs were unroutable, due to congestion, which was due to poor placement (no timing driven placement). And that the speed of the chips were slower, because the cell delays were different (larger) than the cell delays in Design Compiler, due to the type of timing model used in DC (linear delay model).

Synopsys made improvements to DC with the addition of the piece-wise linear timing model (and later with the 2 dimensional and 3 dimensional table lookup models) and in 1993 introduced "links-to-layout". Links-to-layout early abilities included back annotating timing values and net RC parasitics from P&R tools to Design Compiler, and forward annotating timing priorities from DC to the P&R tools. This is what timing convergence means: the exchange of timing information between tools. It enables DC to "see" the design's timing from the perspective of the P&R tool. And it enables the P&R tools to understand what are the critical paths in the design so that it can place them first. With the "new" information Design Compiler could spot and re-optimize the parts of the design that caused problems for the P&R tools.

But this was not enough. More designs were "handed back across the wall" to the designers with instructions to recode their RTL. Furthermore, cell sizes and cell delays kept getting smaller, net delays became the more dominant contributor to the total path delay, chips were getting bigger, wireload models used by Design Compiler were not accurate enough. Synopsys added more features to links-to-layout. PDEF (physical design exchange format) which describes the physical clustering and location of cells in the physical implementation (P&R) of the design. IPO (In Place Optimization) which sizes cells up or down depending on the back annotated timing data. LBO (Location Based Optimization) which uses the PDEF information and back annotated timing data to make changes AND suggest where to put the new cell in the physical implementation. And the create_wire_load command, which allowed the user to make a custom-sized wireload model for each module in their design.

Still, even with these improvements some designs that had met timing in DC could not meet timing in the physical design domain. Timing driven layout routines in the P&R tools were not 100% successful. That is what timing closure requires: satisfying all of your timing goals in all of your tools. In order to be successful with a design you need both timing convergence AND timing closure. Design Compiler usually achieved timing closure (zero negative slack), it was the placement by the P&R tools that prevented timing closure. And since the P&R tool provides the GDSII netlist for the chip's fabrication, not achieving timing closure in the P&R tool prevents the chip from running at speed.

Enter physical synthesis. Physical synthesis combines synthesis technology with placement technology. By combining both into one tool (Physical Compiler) you improve timing convergence by eliminating the external interfaces between synthesis and placement, and you improve timing closure by direct driving your physical placement from your synthesis results with as many internal iterations as necessary. Physical Compiler has been pretty successful so far and it will only get better.

But not all designs work well through PC. There are difficulties. And again, the solution is to go back to the source (the RTL code) and rework the design to meet timing goals or override the problem. The question I ask is: "is there something we can do better in the RTL code to prevent this in the first place?" or "are we all a victim of the implementation tools?" I feel that we should focus more attention on the RTL and the front end using what we know about the back end problems. This is where I am at today: investigating what can be done at the RTL level to prevent problems during the physical implementation. I still feel that we are treating the symptoms and not the root cause of the problem. And we need tools to help us prepare the RTL for the backend implementation.

2.0 Conclusions and Recommendations

RTL design is very holistic. It is not enough to just use good coding style – you have to be aware of what you are implying - logically and physically, the specifics of your design, what tools you are using and how they interact with your RTL. Don't rely on backend solutions to solve your timing problems – focus on doing it right at the beginning when you code your RTL. A little preparation and knowledge can really pay off. Good luck.

3.0 Acknowledgements

I would like to thank Stan Mazor for backing me on my original quest for the golden script. I would like to thank my wife Jeannine for putting up with all my talk of the importance of RTL coding styles. And I would like to thank Amin Shehata for proofreading this paper.

4.0 References

1 November 1999 EE Times Article "Rethinking Deep-Submicron Circuit Design" by Kurt Kreutzer and Dennis Sylvester
2 Speed Optimization Symposium held during the 1994 WorldWide Field conference.