The Human ECO Compiler

Steve Golson

Trilobyte Systems
388 Stearns Street
Carlisle MA 01741
Phone: +1.978.369.9669
Fax: +1.978.371.9964
Email: sgolson@trilobyte.com
http://www.trilobyte.com

ABSTRACT

Engineering Change Orders or ECOs are all too prevalent in ASIC design. Unfortunately there are few tools that directly support these “last minute” changes to a design. So it’s left to us humans to figure out a solution.

This paper will cover the ad hoc solutions that have been developed for implementing ECOs. Topics to be covered:

- basic definitions: the types of ECOs
- where in the flow to make the change
- advanced netlist dissection techniques
- implementing large ECOs using thousands of gates
- equivalence checking: your best friend
- back-end issues
- why ECOs are really a management problem, and how to deal with it

Revised: March 21, 2004

1.0 Introduction—What is an ECO?

We define an *engineering change order* or *ECO* as follows:

> An ECO is a modification made to an automatically-derived representation of a design. This change is made outside of the normal tool flow.

ECOs are sometimes called an *engineering change note* (ECN) or just *engineering change* (EC).

The following are examples of ECOs:

- A schematic capture program generates a netlist of TTL components. The netlist is used to lay out and manufacture a printed circuit board (PCB). The schematic and resulting netlist is then changed. Rather than building an entirely new PCB, the ECO is applied by cutting traces on the existing PCB and adding jumper wires (sometimes called “yellow wires,” “white wires,” or fly wires) [1].
- The same TTL netlist is used by an automatic wire-wrap machine to wire up a circuit board. The ECO is applied by using manual wire-wrap tools to change the connections [2]. Typically the automatic machine used one color wire (e.g., red) and the manual changes are made with a different color (e.g., blue) which led to the term “blue wire” for any type of change to a design.
- An RTL representation of a design is synthesized to a gate-level netlist. The ECO is applied by hand-editing the netlist.
- An IC gate-level netlist is placed and routed using an automatic tool. The ECO is applied by hand-editing the placement and/or routing.

We can broadly split ECOs into two categories: *functional* and *non-functional*. For example a *functional* ECO for an ASIC would require a change to the original RTL. This could be a bug or an added feature. In contrast, a *non-functional* ECO for an ASIC would *not* require a change to the original RTL design[1]. Examples include timing fixes, hold fixes, max capacitance violations, max transition violations, and crosstalk problems.

In this paper we are primarily concerned with functional ECOs for cell-based ASIC design.

1. It is possible to make non-functional changes to the RTL. This can wreak havoc with flows based on `make`, because the RTL file timestamp will change and cause an unnecessary resynthesis.

1.1 Complexity

We can further classify these functional ECOs by how difficult they are to implement. This “complexity” can be measured on several orthogonal scales. In each case a larger number indicates higher complexity.

Mask complexity

How many masks are required to make the change? A full set of masks is very expensive, so requiring a smaller number can save considerable NRE costs. Furthermore partially-processed wafers can be stored awaiting the new metal masks, which saves fab time for the ECO part.

1. Top-layer metal only. Allows changes to existing die using focused-ion-beam (FIB) milling.
2. Metal-only. No base layers are changed.
3. All layers are changed.

Floorplan complexity

In a hierarchical layout, if only one hierarchical block is changed, then back-end testing and verification may be skipped for the unchanged blocks. This can save time and money.

1. One hierarchical block has internal changes.Top-level wiring and all other blocks are not affected.
2. Top-level interconnect is changed, but no ports are changed on the blocks.
3. Multiple blocks are changed.

Design complexity

1. Only a single RTL module is changed.
2. Multiple RTL modules are changed.

Combinational logic complexity

1. Existing logic gates are rewired. No new gates are needed.
2. Additional gates (spare cells) are used to implement the change.

Sequential logic complexity

1. No flops are changed.
2. Redundant or spare flops are used, but the clock tree and scan chain are unaffected.
3. New flops are required. The clock tree and/or scan chain must be changed.

Size complexity

1. Only one new gate is needed.
2. Between one and 10 new gates are needed.
3. Between 10 and 100 new gates are needed.
4. More than 100 new gates are needed.

2.0 Purpose—Why do an ECO?

Consider a typical ASIC flow (Figure 1).

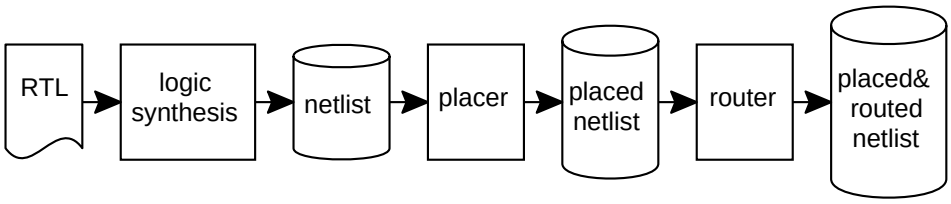

Figure 1. Typical ASIC flow

Normally when a change is made to the RTL, the entire flow is run again using the new RTL to generate a completely new placed and routed netlist (Figure 2).

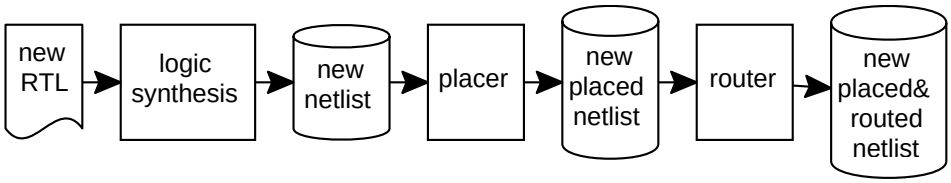

Figure 2. Typical flow when RTL changes

In contrast to implement an ECO we skip over one (or more) of the automated steps (Figure 3).

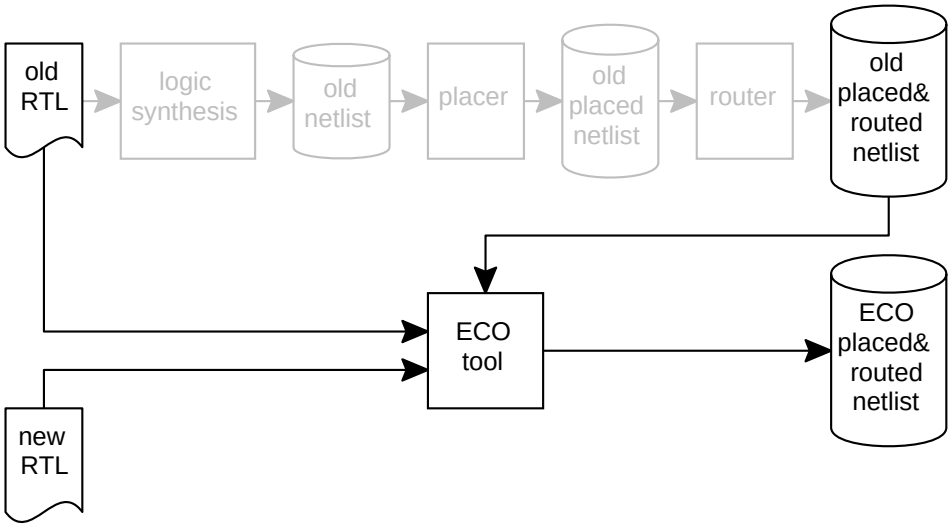

Figure 3. ECO flow skips over some automated steps

Why do an ECO at all? Why not just run through the entire flow again? The ECO flow has several advantages:

- Shorter fab time. Typically some wafers are saved prior to metallization, which enables a metal-only fix to be quickly implemented. If the chip has already been fabricated, a low-complexity ECO could be performed on existing silicon using FIB techniques.
- Lower fab cost. Even if fab time wasn't an issue, having a metal-only fix keeps the mask costs down. Using previously fabricated base layer wafers saves the cost of new wafer starts.
- Shorter design time. Even if fab time and cost were not an issue, running the complete flow can take an enormous amount of time. A low-complexity ECO allows many steps in the flow to be skipped.
- Lower design cost. There may be additional NRE costs to your vendor to pay for a complete respin. A simple ECO could be much cheaper.
- Less political risk. Regardless of the engineering issues, if you've passed a milestone (synthesis completed, placement done, timing closure) it may be politically easier to tweak the design with an ECO than admit to "starting over."
- Predictable results. This is the reason that underlies all the others. The normal flow is *unpredictable* and *chaotic* [3]. Small changes to the input cause large differences in the output. In contrast, the ECO flow is *incremental*—the existing design is kept intact as much as possible [4]–[6].

3.0 Flow—Who will implement the ECO?

Most modern place and route tools have support for incremental changes (Figure 4). However they require an input netlist that is very similar to the original netlist. These tools compare the new netlist to the old netlist, remove just the modified cells and nets, then place and route the new cells. The netlist comparison is name-based, consequently the ECO netlist needs to be syntactically almost identical to the original netlist.

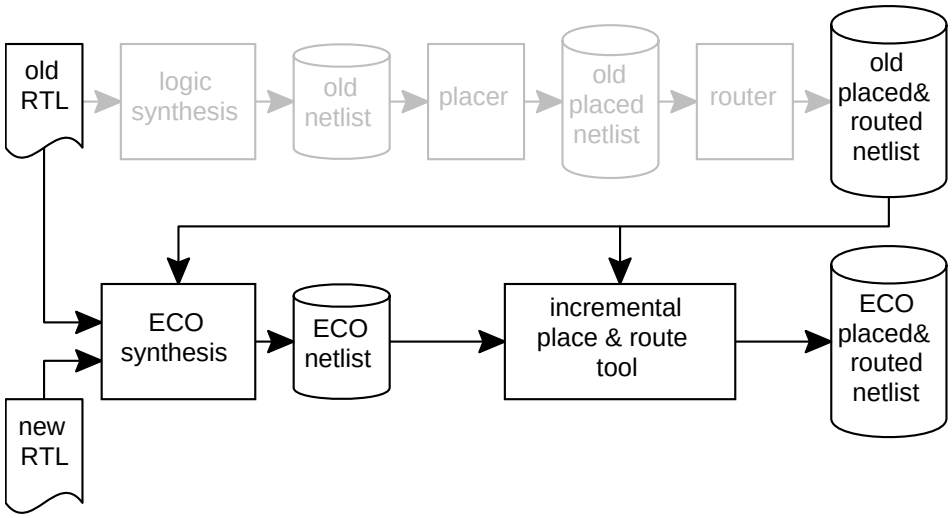

Figure 4. ECO flow using incremental place and route

Thus the big problem is logic synthesis—is there an incremental synthesis methodology that will maintain as much of the original netlist as possible?

Here is one such methodology [4] for implementing ECOs in a synthesized netlist:

1. Modify the HDL specification and reverify the HDL
2. Hand-edit the logic change into the gate-level netlist
3. Check for equivalence between the gate-level and HDL specifications
4. Iterate until the gate netlist matches the HDL specification
5. Perform incremental place and route

Unfortunately this methodology

> …requires designers to search for a gate-level solution. In an HDLA methodology, the logic designer has difficulty matching the HDL specification to the automatically generated synthesis schematic. The resulting change is not "correct by construction." Often the designer must iterate, searching for solutions and using formal equivalence checking before converging on a logically correct solution. [4]

Synopsys ECO Compiler was introduced in April 1997 [7][8] as a way to overcome these obstacles. This tool was developed to enable a correct-by-construction methodology for safe and reliable implementation of ECOs [4]. The basic ECO Compiler flow is shown in Figure 5.

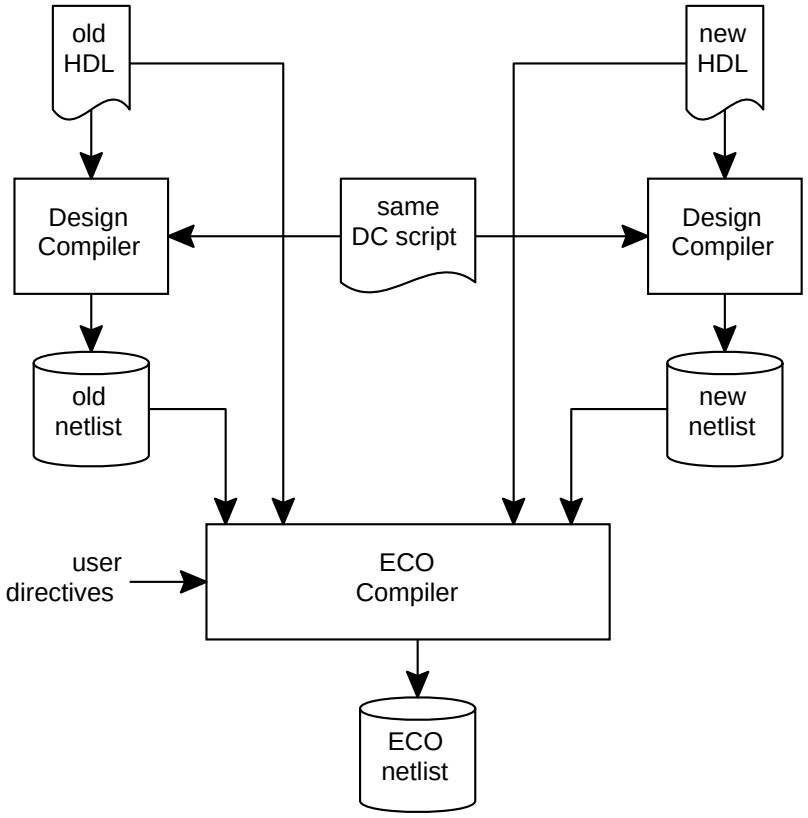

Figure 5. Synopsys ECO Compiler flow

The “ECO Compiler User Guide” [9] summarizes the tool as follows:

> ECO Compiler can be used any time in the design cycle to implement logic changes but generally is used late in the design cycle. Usually you use ECO Compiler in the post-layout phase near the end of the design cycle.
>
> ECO Compiler incorporates in a mapped design a functional change made to its HDL design description. It does this by incrementally combining the functionally unchanged portions of the mapped design and implementing the new portions of the modified (new) HDL. Using ECO Compiler guarantees functional equivalence between the modified HDL and the netlist created by ECO Compiler that implements the modification.
>
> Use ECO Compiler when you need
>
> - Incremental functional change to the RTL specification
> - Stability of synthesis results through incremental functional changes
> - Faster place and route after incremental functional changes to the HDL when the changes are small and localized
> - Guarantee of functional equivalence between the modified HDL and the netlist that implements the modification

The further development of ECO Compiler has lagged [10] and as of version 2001.08 it is no longer supported [11].

Why did ECO Compiler fail? I suspect there were several reasons[2]:

- Incremental synthesis is a *very* hard problem. As changes are made to the mainstream Design Compiler synthesis algorithms, similar changes need to be made to ECO Compiler. This requires scarce R&D resources.
- No support for physical synthesis. After Physical Compiler was introduced, there were many requests that incremental synthesis capabilities be added to it [12]–[15]. Again, this would require a tremendous R&D investment.
- Lackluster sales. It takes a very enlightened manager to authorize the purchase of a tool that is predicated on the design team making mistakes. Furthermore a large company would probably only need a single license of ECO Compiler, in contrast to the many copies of Design Compiler, PrimeTime, and VCS that are typically needed. Finally some potential customers may have already developed their own internal ECO methodology and had no use for another tool.

As for other EDA vendors, although some in-house ECO tools have been developed (e.g., LSI Logic [16], IBM [17], and Mitsubishi [18]) none are currently available on the open market.

If there are no logic synthesis tools to help, we must implement our ECOs using the manual methodology outlined above—which makes us *The Human ECO Compiler.*

2. This is sheer speculation! I have not talked to any Synopsys folks about the fate of ECO Compiler.

4.0 Tools—What you need to implement an ECO

4.1 Things to read

Reviewing the “ECO Compiler User Guide” [9] is a good place to start. This gives a great discussion about ECO concepts, flows, and commands. ECO Compiler is an unsupported product so the documentation is no longer on the SOLD CD or SolvNet. You’ll have to scrounge an old SOLD CD (version 2000.11 or earlier) to get it.

There are several excellent SNUG papers that discuss ECOs. Fox [19] presents a methodology for using ECO Compiler. Rao [20] shows a novel way to implement and verify gate-level ECOs. Horgans et al. [21] discusses a formal verification flow for ECOs.

4.2 Software

It is critical that you have a robust revision control or configuration management system in place [22]. You need a way to track modifications to RTL sources, synthesis scripts, formal equivalence scripts, and netlists. Note that a normal (non-ECO) flow might not keep netlists under revision control, because they are “derived files.” However as part of an ECO flow we will be editing and changing netlists, so they become “primary files.” Furthermore we may need to keep binary files (such as `.db` files and place-and-route databases) under revision control. Thus you must have a revision control tool that can reliably handle very large text and binary files.

Most widely used place and route tools (Cadence, Synopsys/Avant!, and in-house tools such as IBM) support incremental changes to the placed and routed netlist [11]. Study the documentation for your tool. Review the complexity scales in Section 1.1 on page 3. What is the most complex ECO ever implemented by your tool? Ask your EDA application engineer for guidance. If the back-end is done by your silicon vendor, ask them to describe their ECO flow.

It is imperative that you have a formal verification tool such as Formality from Synopsys. Only the most trivial ECOs should be attempted unless you have some way to prove equality with the source RTL [23]. However for very small designs you can use the `compare_design` command in Design Compiler [20].

Bug tracking tools are very useful [24]. Keeping track of all the pending changes is a challenge that is greatly eased when you have a tool for the entire team to use. Also every ECO needs a name, and the bug tracking number is the obvious name to use.

Your favorite text editor is necessary for editing netlists. Very large netlists can be a challenge for some editors.

Perl. You must have Perl. Find it, get it, love it, use it. Having Perl counterbalances the limitations of many other tools.

4.3 Transistors

Very simple ECOs may be implemented by rewiring existing cells. However for any reasonable complexity, extra gates will be needed.

If this is an ECO on a chip that has not yet taped out, then all layers are available, and any needed gates can be placed in unused areas of the die.

On the other hand, if the chip has already taped out, then a metal-only ECO is needed. To provide for these sorts of ECOs, *spare cells* may be included in the design [25][26]. These can be at any level of the logical hierarchy and are spread physically throughout the die [19]. Typically a mix of logic cells and flip-flops make up the spare cell package (also called spare gates, bonus gates, sewing kits, and tool kits). Furthermore some designers may add spare nets [27], spare wiring channels, spare pads (EC pads), spare PLA product terms [28], etc. to ease the implementation of future ECOs and to enable FIB modifications on existing silicon.

A problem with the spare cells methodology is what happens if you pick the wrong type of cell? What happens if you don't have enough spares, or they are too far away? A superior approach is called *gate-array backfill* [29]–[31]. This technique adds gate-array cells in *all* of the space on the die that is unoccupied by active circuitry. These gate-array cells require only metal layers to be characterized into any needed logic gate. Vendors using this technique include IBM [29], Texas Instruments [32], and LSI Logic [33].

Gate-array backfill is a powerful methodology that supports very large metal-only ECOs. Consider that a typical 100k-gate block on a cell-based design might have 80% utilization. The remainder of the active area is backfilled with gate-array cells. Assume after tapeout we need to implement an ECO that requires 1,000 gates—all of which must be created from the backfilled gate-array cells. This is a large ECO, however it only represents a 1% increase to the original gates in the block. Adding our ECO gates to the initial 100k gates marginally increases the utilization from 80% to 81%. Routability and timing closure are still major issues, but placement should not be a problem.

4.4 Vocabulary

We use the following terms to describe netlists and how they differ. They are summarized below in Table 6.

Textually identical

> UNIX `diff` says the netlist files are the same, character for character.

Syntactically identical

> Cellnames are the same, connectivity is the same, netnames are the same, but the netlist files are textually different. Perhaps statements are in a different order, or line breaks are different. (This could be called *lexically identical*, although computer science types would quibble.)

Structurally identical

Cellnames are the same, connectivity is the same, netnames are different.

Structurally different

Cellnames are different and/or connectivity is different.

Functionally identical

Same behavior. An equivalence checker (e.g., Formality) says the netlists are equal. A structurally different netlist can still be functionally identical.

Table 6: Types of netlist equivalence

Vocabulary term	text the same?	same cellnames?	same connectivity?	same netnames?	functionally identical?
textually identical	yes	yes	yes	yes	yes
syntactically identical	no	yes	yes	yes	yes
structurally identical	no	yes	yes	no	yes
structurally different	no	maybe	maybe	maybe	maybe

These next terms are slightly modified from similar definitions introduced by ECO Compiler.

old RTL

The RTL design description from which the old netlist was synthesized.

old netlist

The synthesized and optimized old RTL.

new RTL

The old RTL modified to include the engineering change order.

new netlist

The netlist synthesized from the new RTL.

ECO netlist

The gate-level mapped design generated by the ECO flow. This netlist is functionally identical to the new netlist, but is as syntactically identical as possible to the old netlist.

5.0 Basic flow to apply an ECO to a leaf module

Here is a simple example of a leaf module that is compiled once and then linked. The hierarchy is not changed during the synthesis process, so this module exists in the final netlist. (This could happen with bottom-up compile, or with a top-down compile that preserves hierarchy.)

The basic flow is as follows:

Step 1. Use Formality to confirm equivalence of old RTL and old netlist

Step 2. `diff` the old RTL and new RTL

Step 3. Use Formality to confirm differences between old netlist and new RTL

Step 4. Explore the old netlist with Design Vision and Formality

Step 5. Design a logic fix that implements the ECO in the old netlist

Step 6. Modify the old netlist to make the ECO netlist

Step 7. Use Formality to confirm the new RTL agrees with the ECO netlist

Using the following simple example (Figure 7) we will discuss each step in detail.[3]

```
// $Id: flops.v,v 1.1 2003/10/31 17:06:05 sgolson Exp $

module flops (clk, ina, inb, outc);
input clk, ina, inb;
output outc;

reg flopa, flopb, flopc;

always @ (posedge clk) begin
     flopa <= ina;
     flopb <= inb;
     end

wire mysig = flopa && flopb;
wire mysig_bar = !mysig;

always @ (posedge clk)
     flopc <= mysig_bar;

assign outc = flopc;

endmodule

// $Log: flops.v,v $
//
// Revision 1.1 2003/10/31 17:06:05 sgolson
// new checkin
```

Figure 7. flops.v RTL version 1.1

After synthesis (using the Synopsys class.db library) the resulting netlist[4] is

```
module flops ( clk, ina, inb, outc );
input  clk, ina, inb;
output outc;
    wire n52, n53, n54, n55;
    AN2 U12 ( .A(n53), .B(n52), .Z(n54) );
    IV U13 ( .A(n54), .Z(n55) );
    FD1 flopa_reg ( .D(ina), .CP(clk), .Q(n52), .QN() );
    FD1 flopb_reg ( .D(inb), .CP(clk), .Q(n53), .QN() );
    FD1 flopc_reg ( .D(n55), .CP(clk), .Q(outc), .QN() );
endmodule
```

Figure 8. flops.psv synthesized netlist version 1.1

3. All these examples are in Verilog. Similar techniques can be used in VHDL.
4. For gate-level netlists we use the file suffix `psv` meaning *post-synthesis Verilog*.

Now let's assume we are given the following update. According to our bug tracking tool the new RTL fixes gnat 1234.

```
// $Id: flops.v,v 1.2 2003/12/25 06:15:23 sgolson Exp $

module flops (clk, ina, inb, outc);
input clk, ina, inb;
output outc;

reg flopa, flopb, flopc;

always @ (posedge clk) begin
     flopa <= ina;
     flopb <= inb;
     end

wire mysig = flopa || flopb;
wire mysig_bar = !mysig;

always @ (posedge clk)
     flopc <= mysig_bar;

assign outc = flopc;

endmodule

// $Log: flops.v,v $
//
// Revision 1.2 2003/12/25 06:15:23 sgolson
// bugfix for gnat 1234
//
// Revision 1.1 2003/10/31 17:06:05 sgolson
// new checkin
```

Figure 9. flops.v RTL version 1.2

Figure 7 is the *old RTL*. Figure 8 is the *old netlist*. Figure 9 is the *new RTL*.

Step 1. Use Formality to confirm equivalence of old RTL and old netlist

This is a very important step. Do *not* skip this! Eventually you will use Formality to confirm that the new RTL and the ECO netlist are equivalent. If you have a problem getting that to work, you will naturally assume that your ECO is wrong. Instead, perhaps the Formality setup isn't quite right! The only way to tell is to prove the Formality setup first, by comparing the old RTL and the old netlist. Also this ensures that the old netlist actually was derived from the old RTL.

Step 2. `diff` the old RTL and new RTL

This gives us a feel for what is going on—what has changed? Inputs, outputs, flops, logic? I recommend you use `diff -c` *`old_RTL new_RTL`*. The `-c` argument gives some context around the differences (see Figure 10).

```
*** flops.v.1  Thu Dec 25 22:49:11 2003
--- flops.v.2  Thu Dec 25 22:50:03 2003
***************
*** 1,4 ****
! // $Id: flops.v,v 1.1 2003/10/31 17:06:05 sgolson Exp $

  module flops (clk, ina, inb, outc);
  input clk, ina, inb;
--- 1,4 ----
! // $Id: flops.v,v 1.2 2003/12/25 06:15:23 sgolson Exp $

  module flops (clk, ina, inb, outc);
  input clk, ina, inb;
***************
*** 11,17 ****
               flopb <= inb;
               end

! wire mysig = flopa && flopb;
  wire mysig_bar = !mysig;

  always @ (posedge clk)
--- 11,17 ----
               flopb <= inb;
               end

! wire mysig = flopa || flopb;
  wire mysig_bar = !mysig;

  always @ (posedge clk)
***************
*** 23,27 ****
--- 23,30 ----

  // $Log: flops.v,v $
  //
+ // Revision 1.2 2003/12/25 06:15:23 sgolson
+ // bugfix for gnat 1234
+ //
  // Revision 1.1 2003/10/31 17:06:05 sgolson
  // new checkin
```

Figure 10. Differences between old RTL and new RTL

The revision id has changed, and the revision history has been updated. The new comment confirms that we are working on gnat 1234. Only one functional change has been made; net `mysig` has changed from AND to OR.

Step 3. Use Formality to confirm differences between old netlist and new RTL

At first this seems like an odd thing to do—of course they will be different! Not necessarily. Modifications to the RTL may not actually cause a functional change. Consider the following change:

```
always @ (posedge clk) addr[31:0] <= data_reg2[31:0] // old RTL
always @ (posedge clk) addr[31:0] <= data_reg3[31:0] // new RTL
```

At first glance it appears that all 32 flops need to have their input values changed. However it might turn out that some (or all) of the `data_reg3` bits are always equal to the `data_reg2` bits, therefore no ECO needs to be performed on those bits. Formality will tell you which flops and outputs are affected by the RTL change and actually need an ECO.

Synthesis can do deep and mysterious things to your RTL. Trust your equivalence checker.

Step 4. Explore the old netlist with Design Vision and Formality

Netnames in the RTL are typically not preserved in the gate-level netlist. This is unfortunate; we need to find internal netnames in order to modify them or because we need that signal to make a change elsewhere. In our example earlier (Figure 10 on page 14) we need to find `mysig` in order to modify it, and we need to find signals `flopa` and `flopb` to make the modification.

If the signal is an input or output of a sequential element then it is fairly easy to find the signal. Refer back to our example old RTL and old netlist (Figure 7 and Figure 8 on page 12). Net `flopa` is the output of a flip-flop which takes the name `flopa_reg` in the netlist, thus the equivalent netlist net is `n52`. Likewise RTL net `flopb` is equivalent to netlist net `n53`.

Internal nets are trickier. Some equivalence checkers will automatically map these as part of their normal flow. (This is another reason to run Step 1.) For example, DesignVERIFYer from Chrysalis[5] will print the equivalence report shown in Figure 11.

```
Equivalence #7  - The two groups are equivalent.
->   NSB        +1 17258     A.$371
                +1 17258     A.mysig

                +1 5324      B.n54
                -1 13346     B.n55
                -1 13346     B.U13.Z
->   NSB        +1 5324      B.U12.Z
```

Figure 11. DesignVERIFYer equivalence report

5. Bought by Avant!, bought by Synopsys, now dead.

This tells us that `mysig` (from the RTL design `A.`) is equivalent to netlist net `n54` (design `B.`) which is the `Z` pin of cell U12. Also there is an inverted version of `mysig` on net `n55` which is the `Z` pin of cell `U13`. Such a report is invaluable when planning ECOs to a netlist.

Formality does not have such a direct report. During verification these internal equivalences are found and used, but sadly that information is not retained. However you can run the graphical logic cone viewer and query individual nets that you suspect are equivalent. In the above example you would say

```
verify r:/WORK/flops/mysig i:/WORK/flops/n54
```

and Formality will confirm their equivalence.

The Formality GUI can be used to interactively examine your netlist. Check the documentation for examples.

Design Vision can be used to interactively explore a netlist. Even a very large netlist can be effectively and easily investigated. The important thing to remember is that you do not want to draw the whole schematic, just a little bit of it. Here is the basic procedure:

1. Start up Design Vision.
2. `read_db your_design.db ; link ; reset_design ; update_timing`
   Alternatively you could read in a `.pdb` from Physical Compiler, or read the Verilog netlist. For very large designs this may take several minutes.
3. Select the proper design in the hierarchy, then dismiss the hierarchy browser. This leaves more room for the schematic windows.
4. From the menu bar select Schematic > Add Paths From/Through/To…
5. In the dialog box select Delay type: min
   This is very important. It makes Design Vision draw the shortest logic path, which is what you want when you are exploring a netlist (especially a large netlist).
6. In the dialog box use the From: / Through: / To: boxes and type in the pin/port you want to investigate. You can also give a space-separated list of pins or ports (be sure and change the Max paths: value to be greater than 1).
7. Click OK (or Apply to keep the dialog box around for future use)

Now you can expand the schematic by left-click selecting pins/nets/cells then right-click to pop up a menu and select Add Next Fanin/Fanout Level or Add Logic > Fanin/Fanout…

If you draw the wrong thing, right-click to pop up a menu and select Back to undo. There are multiple levels of undo.

If you select objects in the schematic window, then you can automatically import the selected objects into the From/Through/To text boxes of the Add Paths From/To/Through to Path Schematic dialog box by clicking the Selection button. Make sure you have the proper object type.

The trick is knowing where to start the schematic. If you know that the logic you are investigating begins with a particular flop, or ends on a particular port, then use those as a starting point.

It is best to add one fanout/fanin level at a time. Otherwise you run the risk of the tool attempting to draw lots of logic. If you ask it to draw something and the tool goes away for a long time, using up CPU but not drawing anything, chances are you asked it to do too much. If you think a pin may have a large fanout (or fanin), before you draw that net try

```
report_net [all_connected cell_name/pin_name]
```

Once you get the schematic you want, for future reference print it out and/or save the PostScript into a file. For our small example we get the schematic of Figure 12 (which corresponds to the old netlist of Figure 8 on page 12).

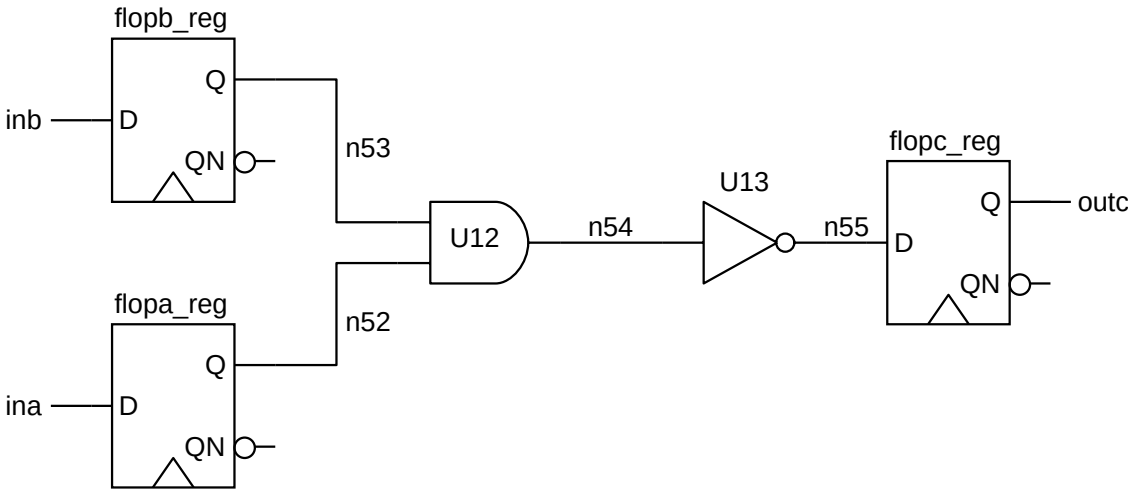

Figure 12. Schematic for flops.psv synthesized netlist version 1.1

Step 5. Design a logic fix that implements the ECO in the old netlist

Recall in Step 4. we discovered that RTL signal `flopa` is equivalent to netlist net `n52`. Likewise `flopb` is equivalent to `n53`. There is only one logical AND of these two signals, the 2-input gate `U12`. Furthermore we can see in the schematic that `U12` eventually drives `flopc_reg`. So the fix is simple enough—cut out the `U12` gate, and replace it with a 2-input OR gate.

A good way to document the fix is to take your printed schematic and manually mark up the desired changes. Use a distinctive color such as red (see Figure 13).

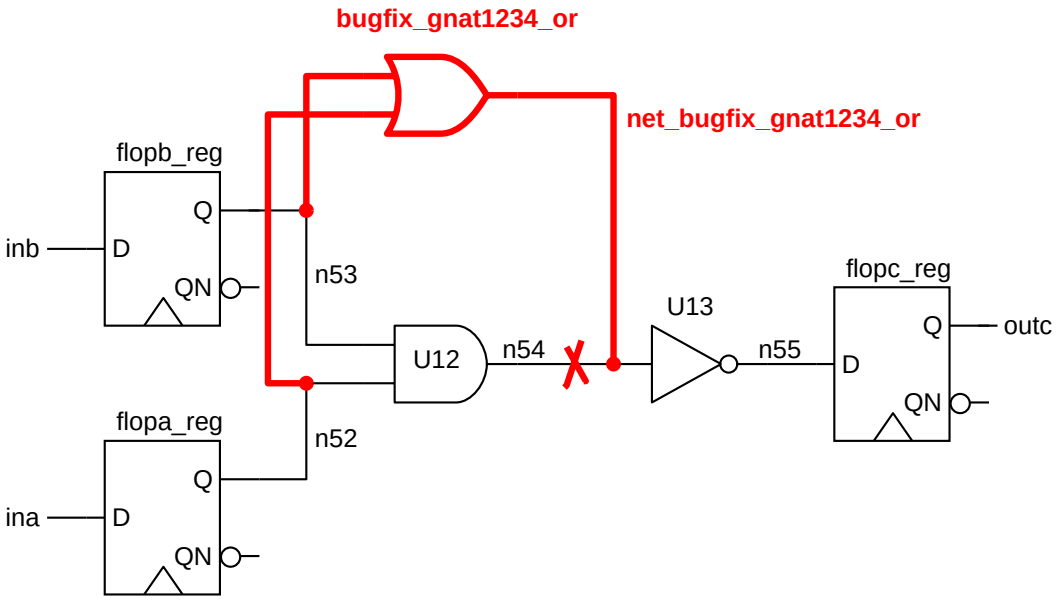

Figure 13. Old schematic with ECO added, corresponding to flops.v version 1.2

Old net `n54` is left with no loads. Your design rules may require that a load cell be placed on this net.

Step 6. Modify the old netlist to make the ECO netlist

For such a simple fix, the easiest way to implement it is to edit the netlist directly (see Figure 14).

```
module flops ( clk, ina, inb, outc );
input  clk, ina, inb;
output outc;
    wire n52, n53, n54, n55;
    AN2 U12 ( .A(n53), .B(n52), .Z(n54) );
// bugfix gnat 1234
    wire net_bugfix_gnat1234_or;
//  IV U13 ( .A(n54), .Z(n55) );
    IV U13 ( .A(net_bugfix_gnat1234_or), .Z(n55) );
    OR2 bugfix_gnat1234_or ( .A(n53), .B(n52), .Z(net_bugfix_gnat1234_or) );
// end of bugfix gnat 1234
    FD1 flopa_reg ( .D(ina), .CP(clk), .Q(n52), .QN() );
    FD1 flopb_reg ( .D(inb), .CP(clk), .Q(n53), .QN() );
    FD1 flopc_reg ( .D(n55), .CP(clk), .Q(outc), .QN() );
endmodule
```

Figure 14. ECO netlist corresponding to flops.v version 1.2

I recommend the following editing techniques:

- The edit is surrounded by comment lines giving the unique ECO name “gnat 1234.”
- When modifying a line, the original line is left in as a comment.
- New cellnames and netnames include the ECO name to avoid possible conflicts.
- The output net from an ECO cell is the name of the cell with prefix `net_`.
- Wire declarations for your ECO nets are recommended but not strictly necessary for structural netlists.

Note that even though the `U12` cell is the one being replaced, the edit is done to its load cell `U13`. This leaves the original net untouched, in case it is used elsewhere.

Step 7. Use Formality to confirm the new RTL agrees with the ECO netlist

Rerun the scripts you created for Step 1. but now compare the new RTL with the ECO netlist. When the comparison is clean, your ECO is done.

6.0 Advanced topics

6.1 Use Design Compiler netlist editing commands to implement the fix

In addition to using a text editor on the netlist (see Step 6. on page 19) there are more ways to implement an ECO. Often we will use a combination of techniques, or use different procedures for different ECOs.

For example, you can implement your change by using the netlist editing commands in Design Compiler. This technique enables you to make intricate changes using sophisticated Tcl procedures. Plus staying within Design Compiler might make more sense for your particular flow.

Figure 15 is a DC Tcl script that implements the gnat1234 ECO described earlier.

```
create_cell bugfix_gnat1234_or class/OR2
create_net bugfix_gnat1234_or_net

connect_net [all_connected U12/A] bugfix_gnat1234_or/A
connect_net [all_connected U12/B] bugfix_gnat1234_or/B
connect_net bugfix_gnat1234_or_net bugfix_gnat1234_or/Z

disconnect_net [all_connected U13/A] U13/A
connect_net bugfix_gnat1234_or_net U13/A
```

Figure 15. Design Compiler Tcl script to implement ECO gnat 1234

6.2 Use Design Compiler to synthesize the ECO logic

For very complex fixes you can use Design Compiler to synthesize the new logic, then paste the gates into your netlist. This is an extremely powerful technique. Starting with the simple flow we outlined earlier in Section 5.0 on page 11, the first steps are unchanged:

Step 1. Use Formality to confirm equivalence of old RTL and old netlist

Step 2. `diff` the old RTL and new RTL

Step 3. Use Formality to confirm differences between old netlist and new RTL

Step 4. Explore the old netlist with Design Vision and Formality

Step 5. Design a logic fix that implements the ECO in the old netlist

Now we get to the interesting part:

Step 6A. Write a delta RTL module that describes the ECO logic

Step 7A. Edit this delta RTL into the old netlist to create a hybrid gates+RTL ECO netlist

Step 8A. Use Formality to confirm the new RTL agrees with this hybrid ECO netlist

Step 9A. Synthesize the delta RTL module

Step 10A. Edit the resulting delta gates into the old netlist to create an ECO netlist

and the final step remains the same:

Step 11A. Use Formality to confirm the new RTL agrees with the ECO netlist

We'll discuss each of the new steps in detail.

Step 6A. Write a delta RTL module that describes the ECO logic

Figure 16 is a Verilog module that implements the new logic for our `flops.v` example. We will call this a *delta RTL* module.

```
// fixes gnat 1234 in flops.v version 1.2

module gnat1234 (

// port declarations
input n53,
input n52,
output mysig
);

//-----------------------------------------------
// fix the port names to the correct RTL names

wire flopa = n52;
wire flopb = n53;

//-----------------------------------------------
// the logic

assign mysig = flopa || flopb;

endmodule
```

Figure 16. Delta RTL implementing ECO logic for gnat 1234

The `input` declarations correspond to netnames from the old gate-level netlist. The output port `mysig` is the fixed version that needs to be edited into the netlist.

The `wire` statements are used to map the old netlist names to the RTL names. This allows the actual ECO logic statements to be directly copied from the new RTL (see Figure 9 on page 13).

The module name is taken from whatever name we are giving this particular ECO.

This module is never simulated—it is only used to generate the proper ECO gates.

Step 7A. Edit this delta RTL into the old netlist to create a hybrid gates+RTL ECO netlist

```
module flops ( clk, ina, inb, outc );
input  clk, ina, inb;
output outc;
    wire n52, n53, n54, n55;
//================================================ delta RTL starts here
// fixes gnat 1234 in flops.v version 1.2

//module gnat1234 (

// port declarations
//input n53,
//input n52,
//output mysig
wire mysig;
//);

//-----------------------------------------------
// fix the port names to the correct RTL names

wire flopa = n52;
wire flopb = n53;

//-----------------------------------------------
// the logic

assign mysig = flopa || flopb;

//endmodule
//================================================ delta RTL ends here
    AN2 U12 ( .A(n53), .B(n52), .Z(n54) );
// bugfix gnat 1234
//  IV U13 ( .A(n54), .Z(n55) );
    IV U13 ( .A(mysig), .Z(n55) );
// end of bugfix gnat 1234
    FD1 flopa_reg ( .D(ina), .CP(clk), .Q(n52), .QN() );
    FD1 flopb_reg ( .D(inb), .CP(clk), .Q(n53), .QN() );
    FD1 flopc_reg ( .D(n55), .CP(clk), .Q(outc), .QN() );
endmodule
```

Figure 17. Hybrid ECO netlist comprising old netlist + delta RTL

Starting with the old netlist (Figure 8 on page 12), the delta RTL is pasted in at the top right after the `wire` declarations. This allows the RTL statements to see all the old wires, and any new wires declared in the RTL can be used elsewhere in the netlist. The `module`, `endmodule`, and `input` statements are commented out. (Normally these would be deleted; they have been left in to aid discussion.) The `output` statements have been changed to `wire` declarations.

Furthermore, old inverter `U13` has been edited to graft the ECO logic into the existing gates (compare with Figure 14 on page 19).

Step 8A. Use Formality to confirm the new RTL agrees with this hybrid ECO netlist

This is the key step. Rerun the scripts you created for Step 1. but now compare the new RTL with the hybrid netlist. If Formality says these two descriptions are equivalent then we continue. If not, then modify the delta RTL until equivalence is reached.

You can edit the delta RTL right in the hybrid netlist. This makes turnaround very fast—there is no need to synthesize new gates every time you need to make a change. Furthermore it is much easier to play “what if” games when testing possible mappings of old netlist netnames to actual RTL netnames. The Formality GUI can be very helpful when exploring any differences.

Step 9A. Synthesize the delta RTL module

This is a very simple synthesis run. There are no timing or design rule constraints; all we want is correct functionality and the smallest possible area. First the delta RTL source is read in, then all input ports are given infinite drive, all output ports have zero load, and a wireload model is selected that has zero resistance and capacitance for all interconnect. Max area is likewise set to zero:

```
read_verilog gnat1234.v

set_wire_load_model -name NONE
set_wire_load_mode top
set_drive 0 [all_inputs]
set_load 0 [all_outputs]
set_max_area 0
```

All new cells must be given unique cellnames that do not conflict with any existing cells in the old netlist. This is accomplished by incorporating the ECO name into the instance name prefix:

```
set compile_instance_name_prefix “bugfix_gnat1234_U”
set compile_instance_name_suffix “”
```

The compile command uses the `-ungroup_all` switch to ensure that any inferred DesignWare components do not create a new level of hierarchy. Also the `-no_design_rule` option is selected, and high effort is used for both mapping and area:

```
compile -ungroup_all -map_effort high -area_effort high -no_design_rule
```

Finally we use `change_names` to force all new nets to have unique legal netnames:

```
define_name_rules netnames -type net -map { { \
    {“[“,”_”}, \
    {“/”,”_”}, \
    {“]”,””}, \
    {“^n”,     “bugfix_gnat1234_”} \
} }
define_name_rules add_net -type net -map { {{“^bugfix”,”net_bugfix”}} }
define_name_rules fix_ports -type net -equal_ports_nets

change_names -rules netnames
change_names -rules add_net
change_names -rules fix_ports
```

Figure 18 shows the entire script.

```
##################################################################
# define some useful name rules

define_name_rules netnames -type net -map { { \
    {“[“,”_”}, \
    {“/”,”_”}, \
    {“]”,””}, \
    {“^n”,     “bugfix_gnat1234_”} \
} }

define_name_rules add_net -type net -map { {{“^bugfix”,”net_bugfix”}} }

define_name_rules fix_ports -type net -equal_ports_nets

##################################################################

read_verilog gnat1234.v

set_wire_load_model -name NONE
set_wire_load_mode top
set_drive 0 [all_inputs]
set_load 0 [all_outputs]

set_max_area 0

set compile_instance_name_prefix “bugfix_gnat1234_U”
set compile_instance_name_suffix ““

compile -ungroup_all -map_effort high -area_effort high -no_design_rule

change_names -rules netnames
change_names -rules add_net
change_names -rules fix_ports

write -f verilog -out gnat1234.psv
write

report_area -nosplit
report_reference -nosplit

redirect gnat1234.report_net  {report_net -nosplit}
redirect gnat1234.report_cell {report_cell -nosplit}

exit
```

Figure 18. Design Compiler script to synthesize delta RTL into delta gate-level netlist

The reports at the end can be reviewed to verify that all new cells and nets have unique names. New cellnames must start with `bugfix_gnat1234` and new netnames must start with `net_bugfix_gnat1234`.

Figure 19 shows the resulting delta netlist.

```
module gnat1234 ( n53, n52, mysig );
input n53;
input n52;
output mysig;
    wire   net_bugfix_gnat1234_1;
    IV bugfix_gnat1234_U1 ( .A(net_bugfix_gnat1234_1), .Z(mysig) );
    NR2 bugfix_gnat1234_U2 ( .A(n52), .B(n53), .Z(net_bugfix_gnat1234_1) );
endmodule
```

Figure 19. Delta netlist corresponding to delta RTL from Figure 16

Step 10A. Edit the resulting delta gates into the old netlist to create an ECO netlist

Similar to how the hybrid netlist was created, we start with the old netlist (Figure 8 on page 12) and paste the delta netlist in right after the `wire` declarations (see Figure 20).

```
module flops ( clk, ina, inb, outc );
input  clk, ina, inb;
output outc;
    wire n52, n53, n54, n55;
// bugfix gnat 1234
    wire mysig;
    wire   net_bugfix_gnat1234_1;
    IV bugfix_gnat1234_U1 ( .A(net_bugfix_gnat1234_1), .Z(mysig) );
    NR2 bugfix_gnat1234_U2 ( .A(n52), .B(n53), .Z(net_bugfix_gnat1234_1) );
// end of bugfix gnat 1234
    AN2 U12 ( .A(n53), .B(n52), .Z(n54) );
// bugfix gnat 1234
//  IV U13 ( .A(n54), .Z(n55) );
    IV U13 ( .A(mysig), .Z(n55) );
// end of bugfix gnat 1234
    FD1 flopa_reg ( .D(ina), .CP(clk), .Q(n52), .QN() );
    FD1 flopb_reg ( .D(inb), .CP(clk), .Q(n53), .QN() );
    FD1 flopc_reg ( .D(n55), .CP(clk), .Q(outc), .QN() );
endmodule
```

Figure 20. ECO netlist comprising old netlist + delta gates

The `module`, `endmodule`, and `input` port declarations from the delta netlist are deleted. The `output` port declarations are converted to `wire` statements.

Step 11A. Use Formality to confirm the new RTL agrees with the ECO netlist

Rerun the scripts you created for Step 1. (and used again in Step 8A.) but now compare the new RTL with the ECO netlist. When the comparison is clean, your ECO is done.

6.3 Adding ports to a subdesign

Sometimes you would like to alter the ports of a subdesign. Perhaps your spare gates are at the wrong level of hierarchy, or you need a signal that only exists in a subdesign. In these cases you can text edit both designs, or you can add a port using Design Compiler editing commands.

Consider the following hierarchical design (Figure 21).

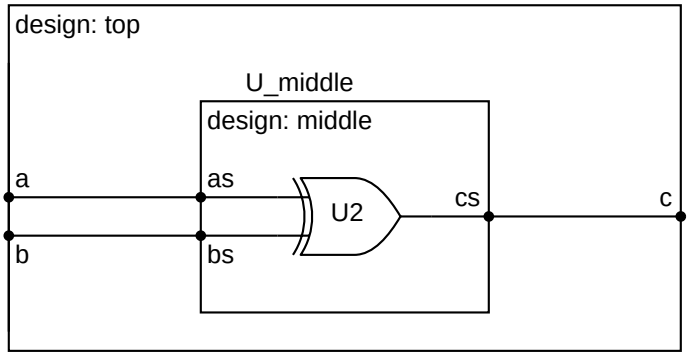

Figure 21. Example hierarchical design

We would like to add a port `ds` to the subdesign `middle` (Figure 22).

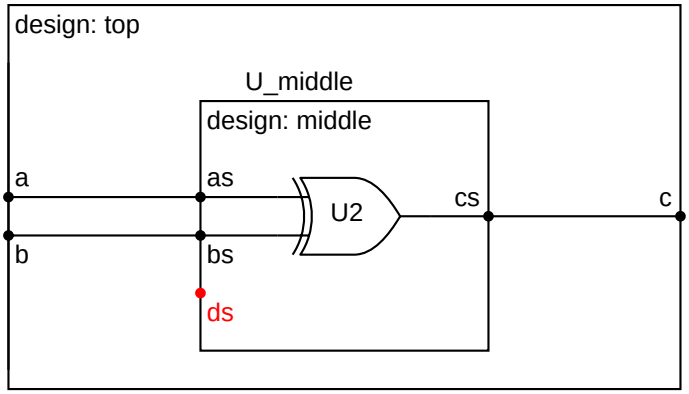

Figure 22. Example hierarchical design with new port added to subdesign

Your first attempt will be something like

```
current_design top
link

current_design middle
create_port -direction in {ds}

current_design top
link
```

But now `top` will fail to link, because the ports now mismatch! `top` does not know about new port `ds`.

One solution is to create a copy of design `middle` and add it in your design as a temporary placeholder to keep track of the port connectivity. Then add the port to `middle`, put the instance of `middle` back in your design, and wire it up. Delete the temporary design and you are done. Figure 23 is a script that shows how it works.

```
current_design top

# create a temporary copy of the target design
copy_design middle TEMP_DESIGN

# instantiate the temp design
create_cell TEMP_CELL TEMP_DESIGN

# wire up the temp design ports in parallel with the target instance ports
foreach_in_collection _pin [get_pins U_middle/*] {
  set _name [get_attribute $_pin pin_name]
  connect_net [get_nets -of_objects $_pin] [get_pins TEMP_CELL/$_name]
}

# remove the target instance
remove_cell U_middle

# add the port to the target design
current_design middle
create_port -direction in {ds}

# put the target instance back, and wire up its ports
current_design top
create_cell U_middle middle
foreach_in_collection _pin [get_pins TEMP_CELL/*] {
  set _name [get_attribute $_pin pin_name]
  connect_net [get_nets -of_objects $_pin] [get_pins U_middle/$_name]
}

# delete the temp design
remove_cell TEMP_CELL
remove_design TEMP_DESIGN
```

Figure 23. Design Compiler Tcl script to add a port to a subdesign

6.4 More on netlist dissection

When tracing signals in Design Vision, be careful to find all the copies of the signal you are looking for. Consider this RTL fragment:

```
assign enable = a || b;
always @ (posedge clk) rdy <= enable ;
```

Suppose our ECO requires us to change signal `enable`. We need to find it in the netlist. Using Design Vision we trace backwards from flop `rdy_reg` and find something like Figure 24.

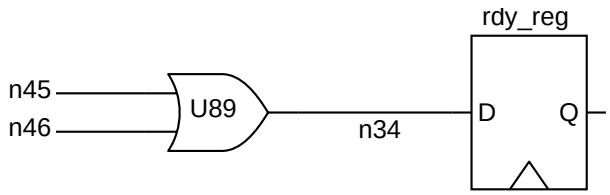

Figure 24. Design Vision schematic

It appears that `n34=enable` and a quick Formality check confirms that `a=n45` and `b=n46`.

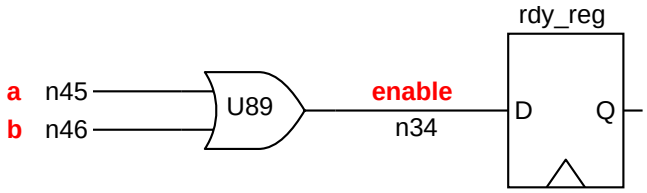

Figure 25. Annotated schematic

So we are all set, right? Not so fast! We need to look at the fanout of `a` (net `n45`) and `b` (net `n46`). When you use the Design Vision menu Schematic > Add Paths From/Through/To…, the resulting schematic window has annotated nets. Just roll the cursor over a net, and a pop-up appears showing the netname and its fanout. This is extremely useful in determining if there are additional loads that are not in the current schematic view. In this case we find a fanout of three (Figure 26).

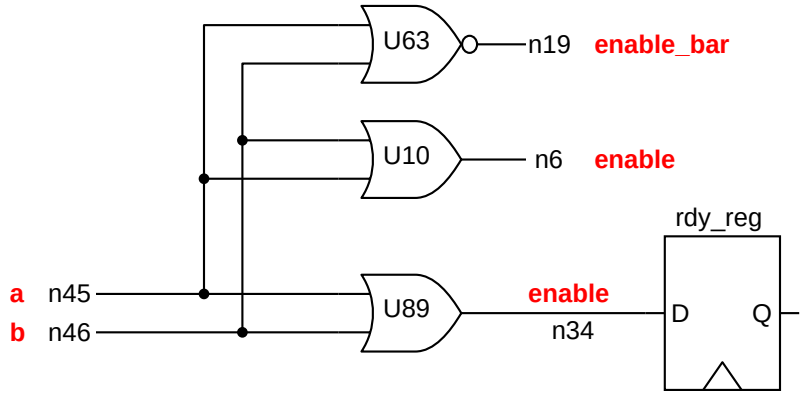

Figure 26. Schematic showing more logic

We've found another net where `enable` exists, plus an inverted version. Are we done yet? No, we really need to trace `a` and `b` back through any inverter/buffer trees to their original source, and then forward again (Figure 27).

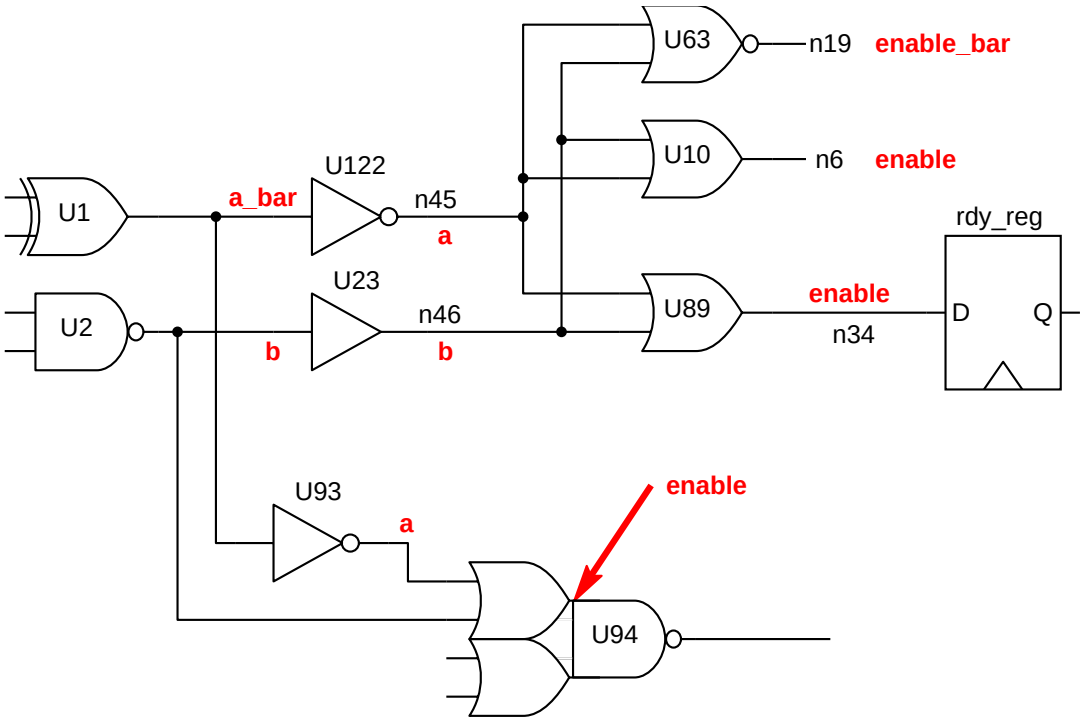

Figure 27. Schematic showing still more logic

Yet another version of `enable`, and it's *inside* complex OAI gate `U94`. But we're still not done; you need to trace back the input nets driving `U1` to see if there is another XOR gate in parallel with `U1`… You get the idea.

By careful inspection of the RTL we could have anticipated this—how many places is `enable` used in the RTL? Is there anywhere else that the expression `a||b` is used? Where do `a` and `b` go?

Assume that elsewhere in our RTL we find this statement:

```
assign detonate = c && (b || a || ready);
```

It appears that some of the nets we thought were `enable` signals may instead be part of this `detonate` signal (because both signals include the term `a||b`). If our ECO requires a change to `enable` but not `detonate`, how can we distinguish between the two?

One method is to use timing analysis to find *all* paths through signal `a`. Do the same with signal `b`. Some endpoints will correspond to `enable` and others to `detonate`. Careful examination of the timing reports will allow you to determine which of the `a||b` nets we found in the netlist are used to create `enable` (and thus need the ECO) and which should remain unchanged. When in doubt, make the fix and ask Formality to confirm.

6.5 Laws of Boolean algebra

When you are decrypting a netlist and attempting to correlate it to your RTL, often you will need to apply DeMorgan's laws and the principal of duality [34]. Every logic symbol has a dual representation which is arrived at by turning all ANDs into ORs, all ORs into ANDs, and inverting all inputs and outputs. Note this works for complex gates as well as simple ones. See Figure 28 for examples.

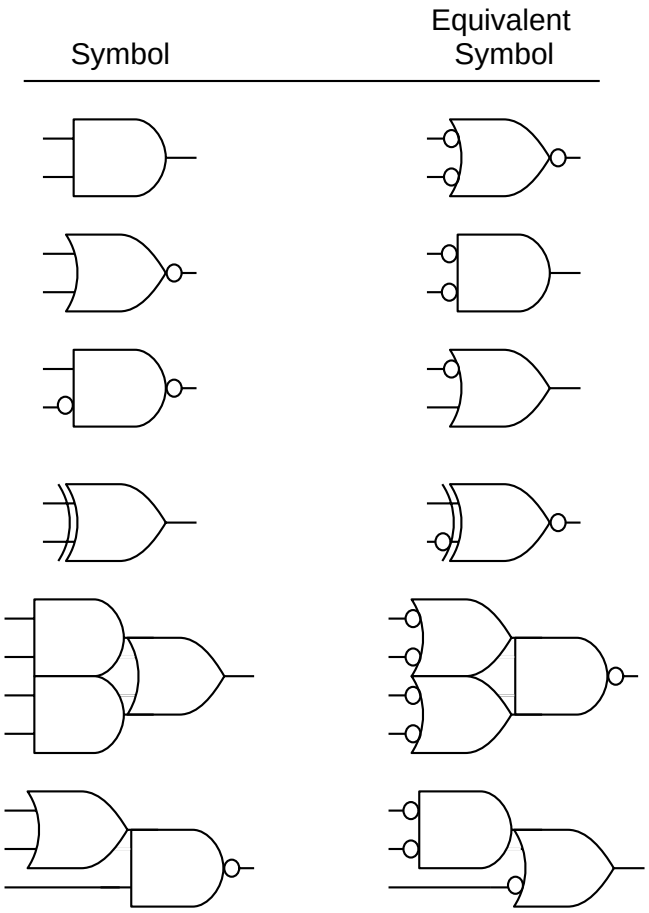

Figure 28. Dual representations of example logic gates

Design Vision can only draw one representation of a logic gate. Sometimes it is useful to convert the symbol to its dual by hand on your printed schematic, to make the logic easier to follow.

Synthesis can do profound and mysterious things to your RTL. Observe, marvel, and learn.

6.6 Related logic sometimes has related names

Sometimes related logic will end up with similar cell names and netnames. This can be helpful when dissecting a netlist. Consider the example netlist fragment of Figure 29.

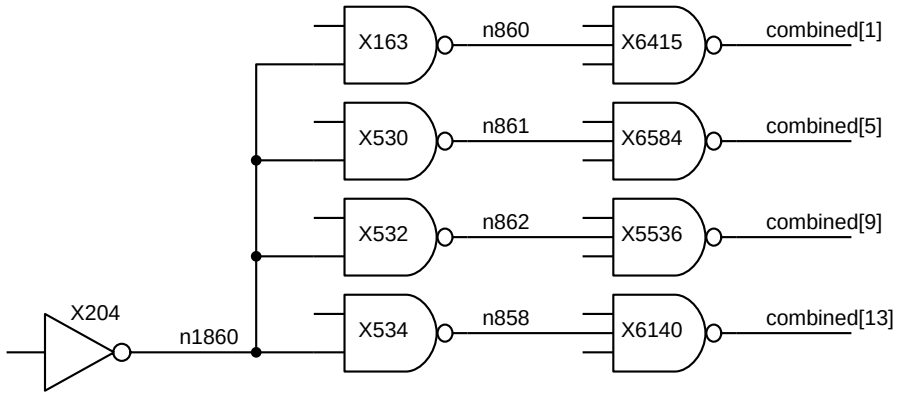

Figure 29. Schematic showing related logic having similar names

Notice that the netnames on the outputs of the 2-input NANDs are all related. Also, three of the 2-input NANDs have similar cellnames.

6.7 More ways to design the fix

The difficult part of designing an ECO is typically not the sequential logic (flip-flops) but rather the combinational logic.

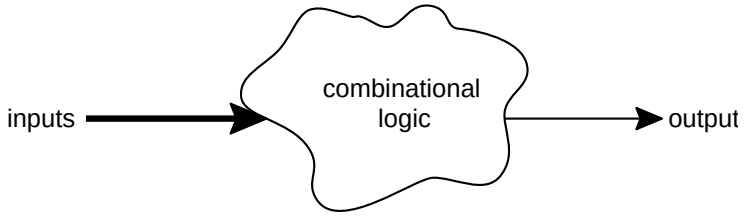

Figure 30. Logic in need of an ECO

Consider Figure 30. Naturally this “output” could be driving the D-input of a flop; however that does not affect the implementation of the ECO in the combinational logic. Likewise the “inputs” could be actual inputs to the module, or perhaps Q-outputs from flops in the module, or some of each. In any case we can consider them inputs to the combinational logic, and they are easy to find in the netlist.

There are several ways to design the actual fix to the combinational logic. If you can figure out the details of the old netlist (like we did in Step 5. of the simple example back on page 18) then you have an *incremental fix*. You’ve actually implemented the fix pretty much the way an incremental synthesis tool like ECO Compiler would.

What if the logic is too complicated to figure out? One approach is to completely replace the existing logic with the correct logic (Figure 31). This is a *brute force fix*. This can be very expensive because you have doubled the gates used for this output.

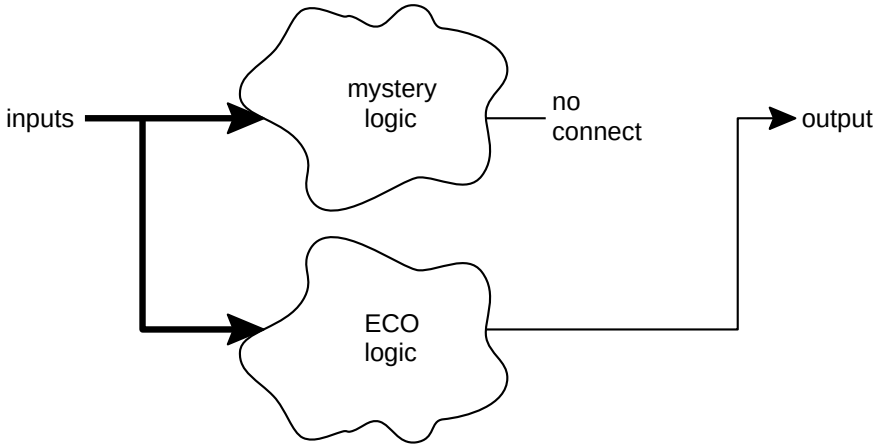

Figure 31. Inscrutable mystery logic requiring brute force ECO fix

Another technique assumes that the existing logic is correct most of the time, thus all we need is a signal to tell us when the existing logic is wrong. Use this bypass signal to control a mux that selects between the original logic and the corrected ECO logic [20]. This is a *muxed output fix*.

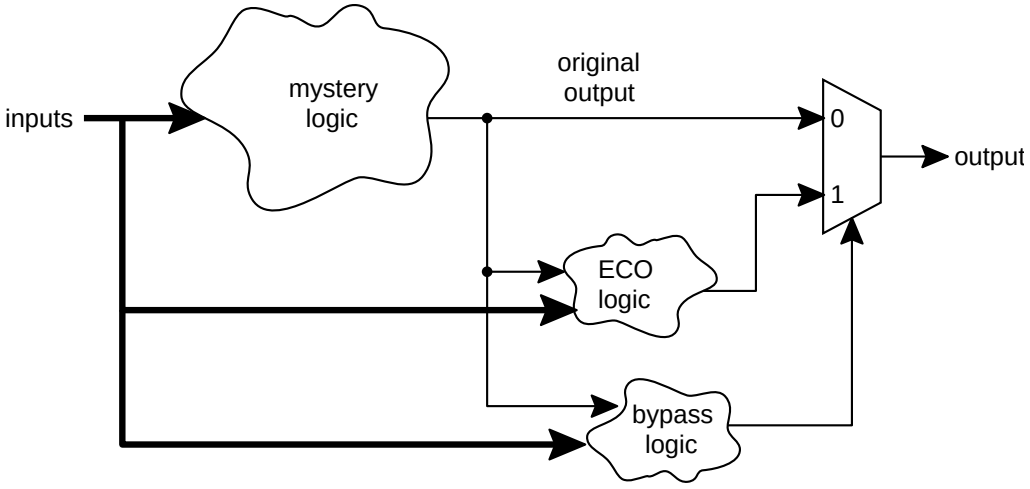

Figure 32. Muxed output ECO fix

Inputs to the bypass and ECO logic can include the primary inputs to the module, flop outputs, any internal signals you are able to find, and the original output from the mystery logic. Often the ECO logic is simply a constant value that is forced onto the output.

The mux, ECO logic, and bypass logic are all implemented in a single delta RTL module (see Step 6A. on page 22). The original output must be renamed in the netlist; for example if the output signal being fixed is `wrdata[63:0]` then rename it to `wrdata_orig[63:0]` and feed it as an input to the delta RTL. Don't forget to add a `wire` declaration for these `_orig` nets.

Rather than muxing the outputs, it might be easier to design a *muxed inputs fix* that forces the inputs to a value that will give us the output we want. This is particularly useful in state machines—force the current state vector and FSM inputs to values that cause the next state vector and FSM outputs to be what we want.

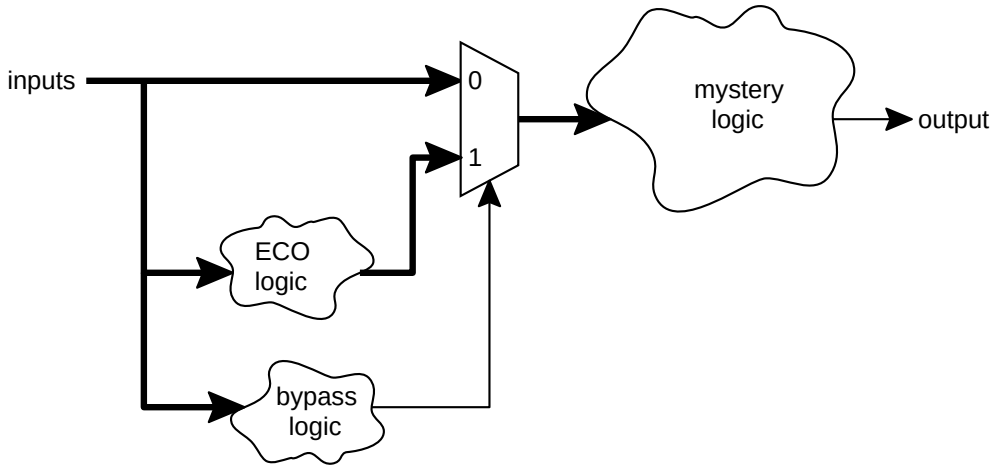

Figure 33. Muxed inputs ECO fix

Combinations of these techniques can be used. You may be able to design an incremental fix for some signals, while others require muxing or brute force.

6.8 Flopwhacking: How to use a spare or redundant flop

Let's return to our simple example from before. Assume we have been given yet another update to our module.

```
// $Id: flops.v,v 1.3 2003/12/31 23:59:58 sgolson Exp $

module flops (clk, ina, inb, outc);
input clk, ina, inb;
output outc;

reg flopa, flopb, flopc, new_flop;

always @ (posedge clk) begin
     flopa <= ina;
     flopb <= inb;
     end

wire mysig = flopa && flopb;
wire mysig_bar = !mysig;

always @ (posedge clk)
     new_flop <= mysig_bar ^ ina;

always @ (posedge clk)
     flopc <= new_flop;

assign outc = flopc;

endmodule

// $Log: flops.v,v $
//
// Revision 1.3 2003/12/31 23:59:58 sgolson
// bugfix for gnat 1240, update to gnat 1234
//
// Revision 1.2 2003/12/25 06:15:23 sgolson
// bugfix for gnat 1234
//
// Revision 1.1 2003/10/31 17:06:05 sgolson
// new checkin
```

Figure 34. flops.v RTL version 1.3

As before we'll do a `diff -c` on the RTL to see what's changed. We will compare the original version 1.1 (Figure 7 on page 12) with this new version 1.3 (Figure 34), skipping over the previous ECO version 1.2. Sometimes it is easiest to start fresh using the original netlist from the last all-up synthesis run, rather than add an ECO onto a previous ECO. This is particularly attractive when using the delta RTL implementation technique, because you can often easily add

the new ECO statements to your delta RTL module. When deciding which path to take, consider how much back-end work has been done to implement the previous ECO—all those back-end tasks will need to be redone.

```
*** flops.v.1  Thu Dec 25 22:49:11 2003
--- flops.v.3  Wed Dec 31 23:59:58 2003
***************
*** 1,2 ****
! // $Id: flops.v,v 1.1 2003/10/31 17:06:05 sgolson Exp $

--- 1,2 ----
! // $Id: flops.v,v 1.3 2003/12/31 23:57:32 sgolson Exp $

***************
*** 6,8 ****

! reg flopa, flopb, flopc;

--- 6,8 ----

! reg flopa, flopb, flopc, new_flop;

***************
*** 17,20 ****
  always @ (posedge clk)
!               flopc <= mysig_bar;

  assign outc = flopc;
--- 17,23 ----
  always @ (posedge clk)
!               new_flop <= mysig_bar ^ ina;

+ always @ (posedge clk)
+               flopc <= new_flop;
+
  assign outc = flopc;
***************
*** 25,26 ****
--- 28,35 ----
  //
+ // Revision 1.3 2003/12/31 23:57:32 sgolson
+ // bugfix for gnat 1240, update to gnat 1234
+ //
+ // Revision 1.2 2003/12/25 06:15:23 sgolson
+ // bugfix for gnat 1234
+ //
  // Revision 1.1 2003/10/31 17:06:05 sgolson
```

Figure 35. Differences between old RTL and new RTL

The update for gnat 1240 has added a flop `new_flop` and an XOR gate.

Figure 36 shows our fix.

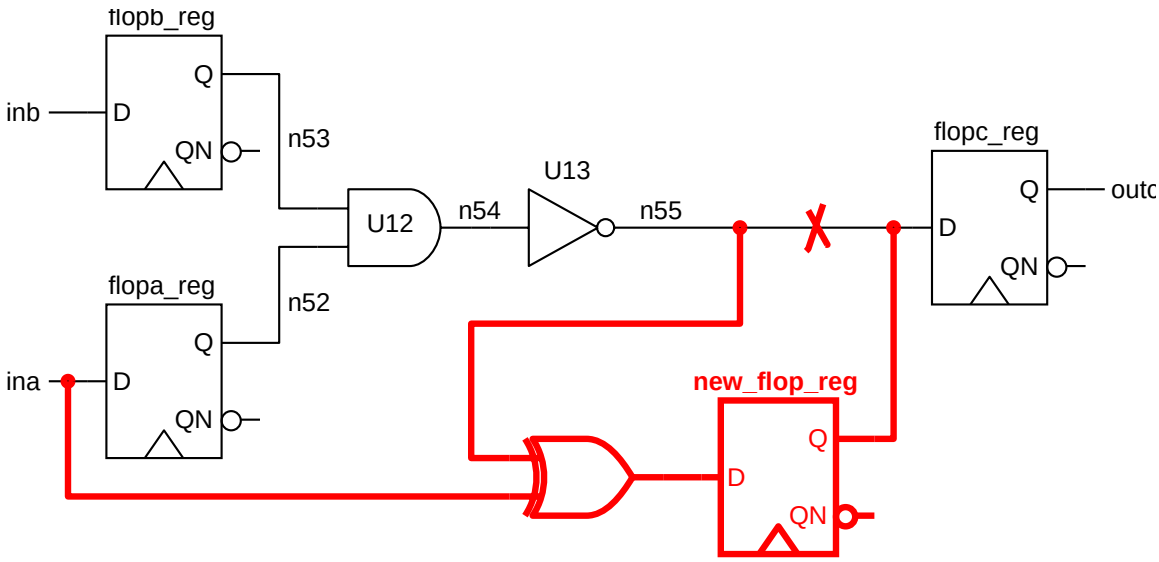

Figure 36. Old schematic with ECO added, corresponding to flops.v version 1.3

After a quick delta RTL and synthesis, the resulting ECO netlist looks like Figure 37.

```
module flops ( clk, ina, inb, outc );
input  clk, ina, inb;
output outc;
    wire n52, n53, n54, n55;
// bugfix for gnat 1240
  wire new_flop;
  EO bugfix_gnat1240_U9 ( .A(n55), .B(ina), .Z(net_bugfix1240_n28) );
  FD1 new_flop_reg ( .D(net_bugfix1240_n28), .CP(clk), .Q(new_flop), .QN() );
// end of bugfix for gnat 1240
    IV U13 ( .A(n54), .Z(n55) );
    AN2 U12 ( .A(n53), .B(n52), .Z(n54) );
    FD1 flopa_reg ( .D(ina), .CP(clk), .Q(n52), .QN() );
    FD1 flopb_reg ( .D(inb), .CP(clk), .Q(n53), .QN() );
// bugfix for gnat 1240
//  FD1 flopc_reg ( .D(n55), .CP(clk), .Q(outc), .QN() );
    FD1 flopc_reg ( .D(new_flop), .CP(clk), .Q(outc), .QN() );
// end of bugfix for gnat 1240
endmodule
```

Figure 37. ECO netlist corresponding to flops.v version 1.3

We have increased the number of flip-flops in the design. Now you must add the new flop to the clock tree, check the clock tree for skew problems, and modify the scan chain. This can add a significant amount of work to the back-end flow. Is there any way to avoid this problem?

One possibility is to take an existing flop in the netlist and rewire it to perform the function you want. We'll call this *flopwhacking*. There are several types of flops you can enlist to be whacked:

- *Spare flop*—One of the flops that was inserted as part of your spare gates methodology. If you have multiple clocks, make sure the clock net is connected to the proper one. Otherwise you are back to fixing the clock buffer tree.
- *Redundant flop*—A redundant flop has the same functionality as some other flop in the design; since they are identical you can tie their outputs together and use one of them for your ECO. One useful place to find redundant flops is in a synchronous reset distribution tree [35].
- *Unused flop*—This is a flop that has some logic driving it, but never changes state (at least not during normal operation). Perhaps a writable control register bit can be tied off as a read-only bit, freeing up its flop for this use. Sometimes counter registers are given excess upper bits that are never incremented. Upper address bits (or the lowest bits) of a pipeline may be static. See if your formal verification tool will report any flops that can never change state. If you change the functionality of your design (e.g., by tying off a control register bit) be sure and modify the RTL as well.

In our example let's assume that top-level Formality analysis tells us that `ina` and `inb` are equivalent. This means that `flopa_reg` and `flopb_reg` are redundant flops. The netlist only needs one of these flops; we can use the other to implement our ECO. Figure 38 shows the revised ECO design.

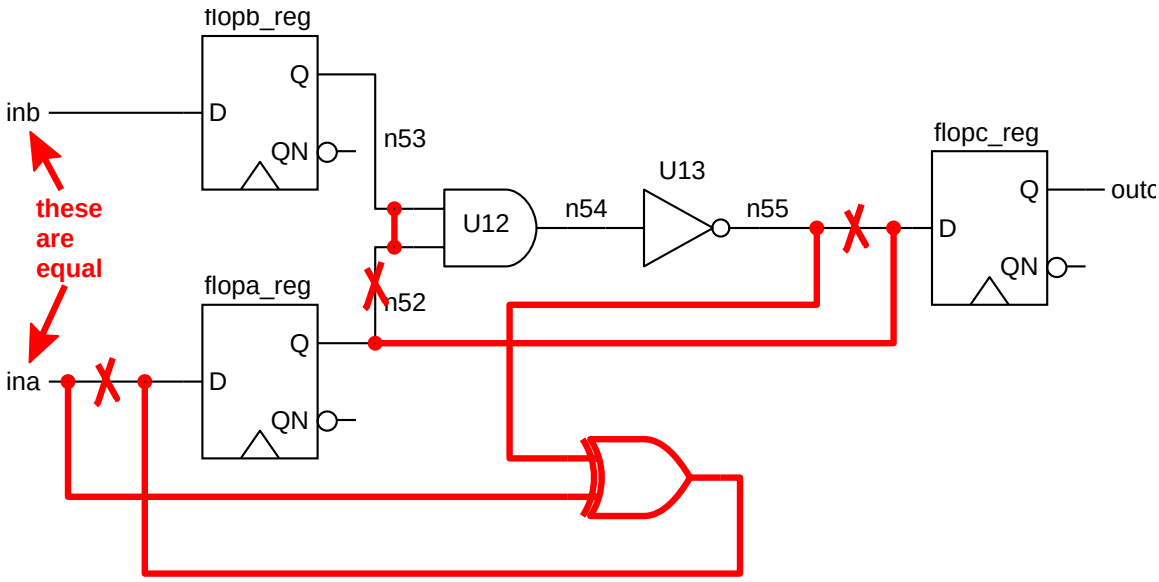

Figure 38. ECO schematic modified to use redundant flopa_reg

We've disconnected `flopa_reg` from its driver and loads, and tied its loads to the output of `flopb_reg`. Now we can rewire `flopa_reg` to perform the function of `new_flop_reg`.

Now, how shall we actually perform the flopwhack? How do we transform the ECO netlist of Figure 37 into the netlist of Figure 38? We could edit the text file directly, or we can use a Design Compiler Tcl script. By using a script we can ensure that these changes are done correctly; also in combination with other Design Compiler commands (e.g., `ungroup`, or the port manipulation tricks from Section 6.3 on page 27) it allows the flops to be at different levels of the hierarchy.

An example script is given in Figure 40. To use the script, first define a few Tcl variables that specify the flop cellnames:

```
set target_flop    {new_flop_reg}
set donor_flop     {flopa_reg}
set redundant_flop {flopb_reg}
```

`$target_flop` will be replaced by `$donor_flop`. Thus they must use the same cell from the cell library. Prior to `compile`, your delta RTL synthesis script should have a statement like

```
set_register_type -exact -flip_flop FD1 [get_cell {new_flop_reg}]
```

The previous loads on the outputs of `$donor_flop` will be tied to the same outputs on `$redundant_flop`, so these cells must have functionally equivalent outputs. They don't have to be the same type of cell, but they must have the same output pins.

Starting with the ECO netlist of Figure 37, running the flopwhacking script shown in Figure 40 gives us the netlist of Figure 39.

```
module flops ( clk, ina, inb, outc );
  input clk;
  input ina;
  input inb;
  output outc;
  wire   n53, n54, n55, new_flop, net_bugfix1240_n28;

  EO bugfix_gnat1240_U9 ( .A(n55), .B(ina), .Z(net_bugfix1240_n28) );
  IV U13 ( .A(n54), .Z(n55) );
  AN2 U12 ( .A(n53), .B(n53), .Z(n54) );
  FD1 flopa_reg ( .D(net_bugfix1240_n28), .CP(clk), .Q(new_flop), .QN() );
  FD1 flopb_reg ( .D(inb), .CP(clk), .Q(n53), .QN() );
  FD1 flopc_reg ( .D(new_flop), .CP(clk), .Q(outc), .QN() );
endmodule
```

Figure 39. ECO netlist after flopwhacking

```
# foreach output pin on donor flop,
# tie load pins to same output pin on redundant flop

foreach_in_collection _pin [get_pins -filter “@pin_direction==out” $donor_flop/*] {
  set _name [get_attribute $_pin pin_name]

  # get the net on this pin
  set _donor_net [get_nets -of_objects $_pin]
  if {$_donor_net == ““} { continue ; }

  # get the loads on this net
  set _donor_loads [get_pins -of_objects $_donor_net]
  set _donor_loads [remove_from_collection $_donor_loads $_pin]
  if {$_donor_loads == ““} { continue ; }

  # disconnect everything from this net, and remove it
  disconnect_net -all $_donor_net $_donor_loads
  remove_net $_donor_net

  # now drive the loads from the redundant flop
  set _redundant_net [get_nets -of_objects $redundant_flop/$_name]
  if {$_redundant_net == ““} {
    set _redundant_net flopwhacknet_${redundant_flop}_${_name}
    echo #FLOPWHACK# Creating net $_redundant_net on pin $redundant_flop/$_name
    create_net $_redundant_net
    connect_net $_redundant_net [get_pins $redundant_flop/$_name]
    }
  connect_net $_redundant_net $_donor_loads
}

# disconnect all pins on donor flop (you might want to leave the clock pin alone)

foreach_in_collection _pin [get_pins $donor_flop/*] {
  set _donor_net [get_nets -of_objects $_pin]
  if {$_donor_net == ““} { continue ; }
  disconnect_net $_donor_net $_pin
}

# connect all donor pins in parallel with target pins (except clock pins?)

foreach_in_collection _pin [get_pins $target_flop/*] {
  set _name [get_attribute $_pin pin_name]

  # get the net on this pin
  set _target_net [get_nets -of_objects $_pin]
  if {$_target_net == ““} { continue ; }

  # connect the donor flop to the net
  connect_net $_target_net [get_pins $donor_flop/$_name]
}

# remove target flop

remove_cell $target_flop
```

Figure 40. Design Compiler Tcl script for flopwhacking

When running the module-level formal equivalence check you must tell the tool that `ina` and `inb` are assumed equal. This should not be necessary during top-level equivalence checking,

Also for all equivalence checking you must manually map the affected flops, because name-based mapping will try to map signal `flopa` in the RTL with `flopa_reg` in the netlist, and they are not equivalent. Table 41 lists the signal mappings that may be required. Note there are two RTL reg bits that map onto the single flop `flopb_reg`.

Table 41: Manual mappings required for proper equivalence checking after flopwhacking

RTL reg	netlist flop pin
`flopa`	`flopb_reg/Q`
`flopb`	`flopb_reg/Q`
`new_flop`	`flopa_reg/Q`

6.9 ECO considerations for hierarchical designs

So far our examples have been a single module. Our synthesis flows are rarely so simple! Consider a design `top` which has three subdesigns `A`, `B`, and `C`. Figure 42 shows an outline of a more representative flow for such a chip.

```
read_verilog all_the_RTL
...
current_design A
compile
...
current_design B
compile
...
current_design C
compile
...

current_design top
link
ungroup -all -flatten
...
write -hier -format verilog -out top.psv
```

Figure 42. Example flow for hierarchical design with three subdesigns

Now assume we have an ECO for design `B`. Where in the flow should we implement the ECO? One solution is to just edit the final ungrouped netlist `top.psv`. This can be very tedious and error-prone. An alternative approach is to perform the ECO at the module level, and then run the rest of the flow as usual such that the unaffected parts of the chip are synthesized normally. Figure 43 shows how the ECO netlist is linked in.

```
read_verilog all_the_RTL
...
current_design A
compile
...
remove_design B
read_verilog B_ECO.psv           ;# read the ECO netlist for design B
...
current_design C
compile
...

current_design top
link
ungroup -all -flatten
...
write -hier -format verilog -out top_ECO.psv
```

Figure 43. Modified flow reading ECO netlist for a subdesign

This can work even if you do top-down hierarchical compile, because the module boundaries are preserved. Figure 44 shows the original flow. All the RTL is read in, and a single top-down compile is done.

```
read_verilog all_the_RTL
...
current_design top
link
...
compile                        ;# top-down hierarchical compile of top,A,B,C
...
ungroup -all -flatten
...
write -hier -format verilog -out top.psv
```

Figure 44. Top-down hierarchical compile flow

Figure 45 shows the modified flow, where the ECO netlist for design B is read in to replace the synthesized B netlist.

```
read_verilog all_the_RTL
...
current_design top
link
...
compile                        ;# top-down hierarchical compile of top,A,B,C
...
remove_design B                ;#
read_verilog B_ECO.psv         ;# read the ECO netlist for design B
current_design top
link                           ;# link in the ECO netlist
...
ungroup -all -flatten
...
write -hier -format verilog -out top_ECO.psv
```

Figure 45. Modified top-down compile flow reading ECO netlist for a subdesign

If design B has subdesigns of its own, be careful when reading in the ECO netlist. Are these subdesigns included in the ECO netlist? You might wish to use the ones from the main netlist instead. Do the port names match up? To avoid nasty `link` problems, make sure you only have one copy of any given design in memory.

These techniques can even be used if you do a `compile -incremental` or `compile -top` at the end of your script. Sometimes these compiles make only slight changes to your netlist, and having the ECO already inserted will not perturb the results. The only way to know is to try.

Remember, if these techniques don't work you can always fall back and make the ECO edits directly to the final top.psv netlist. It's just harder to navigate around and dissect a big netlist.

6.10 Repeatability

The techniques in the previous section only work if the scripts and flow are *repeatable*. This means that every time we run our script we should get the same results—the netlist is textually identical each and every time. The following conditions must be met [36]:

- Identical data inputs (RTL, constraints, scripts, etc.)
- Same version of operating system and Synopsys tools
- DesignWare cache has identical state for each run

The last item is the tricky one—we don't think about the cache changing, however it can cause subtle differences in the resulting netlist. (One way around this is to delete the cache before every run. Also consider setting the `cache_write_info` variable to `true` so you can tell when the cache is modified.)

First try running your flow using the original flow and old RTL. Do you get the identical netlist? Is this repeatable? If not, why not?

Next it's a good idea to try modifying the flow *without* performing the ECO. That is, instead of reading in the ECO netlist for `B`, we read in the *old* netlist for `B`. Now all we have done is modify the flow a bit. The resulting netlist should be the same as the old netlist. For example, before attempting to use the flow in Figure 45, try the flow of Figure 46.

```
read_verilog all_the_RTL
...
current_design top
link
...
compile                       ;# top-down hierarchical compile of top,A,B,C
...
remove_design B               ;#
read_verilog B.psv            ;# read the old netlist for design B
current_design top
link                          ;# link
...
ungroup -all -flatten
...
write -hier -format verilog -out top_test.psv
```

Figure 46. Modified top-down compile flow

This `top_test.psv` netlist should be textually identical to the old `top.psv` netlist resulting from the original flow of Figure 44. If they are identical, then the ECO flow of Figure 45 should correctly implement the ECO to design `B` and leave the other designs unchanged.

Small changes to your flow can cause unexpected differences in the netlist. Consider the four synthesis flows described in Figure 47.

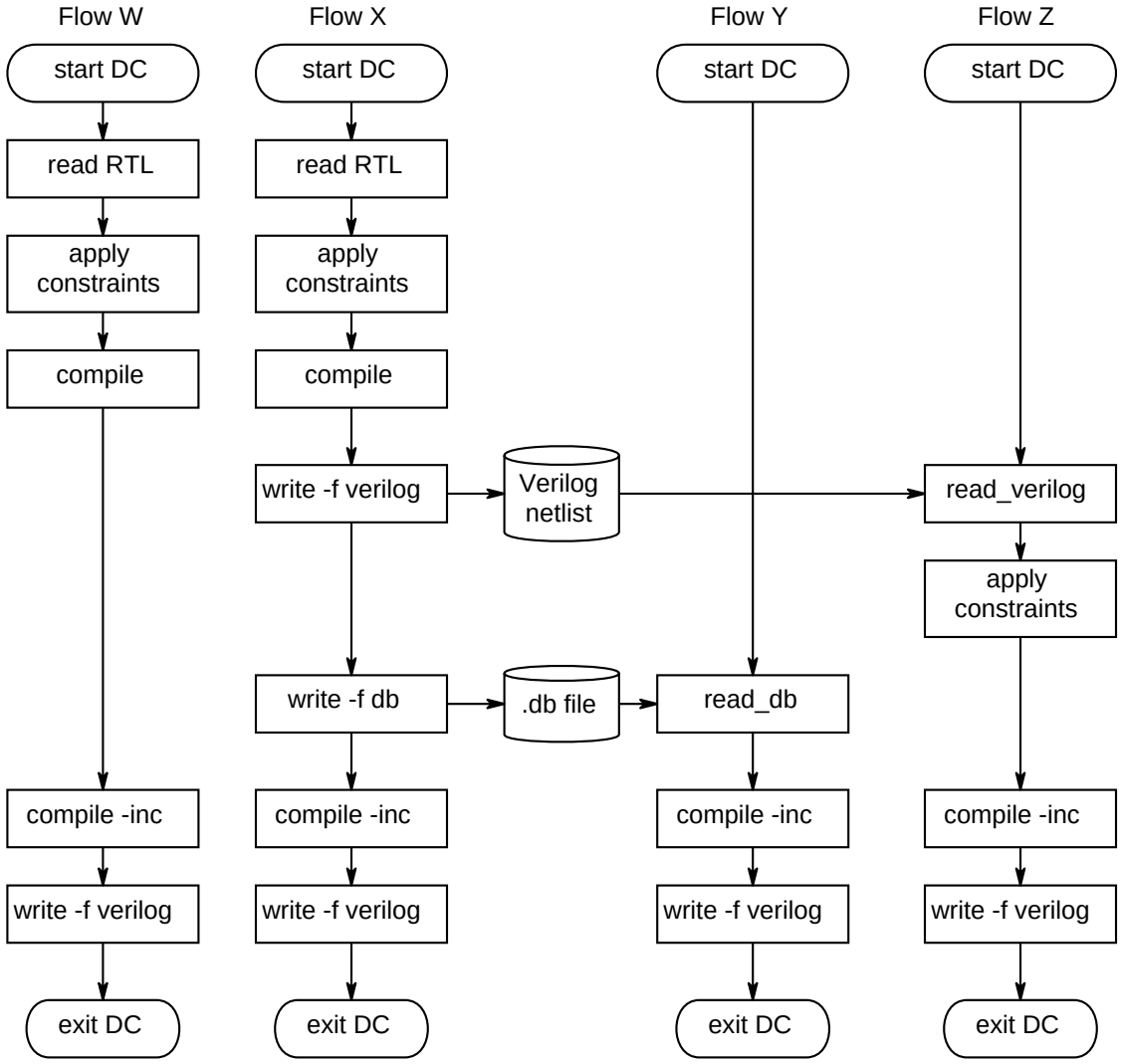

Figure 47. Four flows that should give identical results

You would expect these four netlists to be textually identical, but they may not. Often you will see different netnames, and sometimes cellnames will be changed. This is important because when we are implementing an ECO, we might modify the flow to allow reading an ECO netlist at some point. Be careful and always check your results. You may have unexpected changes.

This is why the netlist editing commands described in Section 6.1 on page 20 are so useful. Rather than reading a netlist you can just perform the ECO edits as part of the flow.

You can cause scan insertion to be repeatable by forcing the scan order to agree with the original scan chain [37][38].

6.11 Netlist equivalence and how to fix it

We want our ECO netlist to be syntactically identical as much as possible with our old netlist (i.e., same cellnames, same connectivity, same netnames). This minimizes the changes that the incremental place-and-route tools will see.

What can we do if our ECO flow mixes up the netnames? or maybe even a few cellnames? Given two netlists that are mostly structurally identical (same cellnames, same connectivity, different netnames) there are several ways you can force the netnames to agree:

- Run a custom dc_shell script to massage the netnames [39].
- Use the dc_shell `change_names` command [40].
- Use Wilson Snyder's excellent Verilog-Perl module [41] to write your own Perl script. The script should parse the old netlist and build up a hash of cellname output pins and the netnames they are connected to. Now parse the ECO netlist, and if you find a cellname output pin that matches one from the old netlist, force its attached netname to be the name saved in the hash. Check out the excellent `vrename` script that comes with Verilog-Perl.

Anecdotal evidence says `change_names` can take a long time, and sometimes doesn't do what you expect.

There are several ways to check that two netlists are structurally identical:

- Use Formality to compare the two netlists with all gates treated as black-boxes [42]
- Write a Perl script. Parse the two netlists and for each cellname output pin, build up a list of pins attached to the net. Compare the lists for each matching cellname output pin.

A quick way to roughly compare netlists is to `grep` out the cellnames from each netlist, sort the list, and `diff` the results. This is necessary but not sufficient for complete equivalence.

7.0 Random useful thoughts and suggestions about ECOs

Tools

The old dc_shell dcsh command language is dead, dead, dead. Use Tcl.

Emacs cannot handle files larger than several hundred megabytes. If you need to edit such a file, and your editor cannot handle it, use the UNIX `split` command to break up the file into manageable sections. Use `cat` to glue them back together. Or use a different editor—large files have been successfully edited in Vim and with Solaris textedit. Alternatively use `sed` or Perl to perform your edits.

In your synthesis scripts, if you are saving a Verilog netlist and a binary `.db` file, always write the Verilog netlist first. Sometimes writing the Verilog will cause implicit name changes, and you want the `.db` to agree with the Verilog netlist.

```
write -hier -format verilog -out my_design.v
write -hier                 -out my_design.db
```

ECO Compiler has some wonderful reporting commands. The `eco_netlist_diff` command produces information about the differences (changes) between the old netlist and the ECO netlist and reports the differences. The `eco_report_cell` command generates a report that lists the number and area of old cells, recycled cells, new cells, spare cells, and obsolete cells in the ECO netlist. Don’t you wish you had something like this? Can you write a Perl script that does it?

Use a graphical utility such as TkDiff when comparing RTL source files.

In Design Vision, you can put lists of pin and port names in the From/Through/To text boxes in the Add Paths From/To/Through to Path Schematic dialog box. The list of names should be space-separated, no commas, no brackets around the list. You will need to change the Max paths: value to be greater than 1 (and perhaps the Nworst paths: value as well).

When using Design Vision, be sure you have the symbol library set properly, otherwise cells may be drawn incorrectly. I have seen AND-OR complex gates drawn with OR-AND symbols. Very confusing…

Get Wilson Snyder’s wonderful Verilog-Perl module [41]. It includes a script called `vrename` which creates Verilog cross-references and makes it easy to rename signal and module names across multiple files. (Early versions of this module had problems parsing Verilog concatenations {} so run some test cases to make sure you understand what is going on.)

You can convince management to buy Formality, Design Compiler, PrimeTime, but not a tool like ECO Compiler! So make do with the tools you have. Figure out a flow that works for you.

Design

When you tape out an ECO design, be sure and change the software-visible revision number! Also make sure the package markings are changed.

When designing bypass logic for delta RTL fixes, you need to build a signal that indicates the ECO logic needs to be activated (the *bypass* signal). Look for flops (or combinations of flops) that are about to change to a unique value. Grab the input pins to these flops. Look for internal signals in unique states.

If you cannot find a particular signal in the netlist, go ahead and build it from scratch in your delta RTL. Copy the pertinent RTL statements from your old source RTL. Remember, you can easily find all module inputs and flop outputs and use them as inputs to your delta RTL. If you have more time you can go back and explore the netlist more to find the signals you have duplicated thus saving this redundant logic.

A really smart place-and-route tool will optimize away logic that has no loads. This might happen if a module output is unconnected at the top level, and your synthesis flow does no top-level optimization. Now assume you need to use that logic to perform an ECO. In the synthesized netlist the logic is there, however it's not in the routed netlist.

One-hot state machines are *much* easier to ECO than highly-encoded machines [43].

A synthesis flow using DC Ultra can perform aggressive sequential optimization on your netlist (e.g., register retiming and FSM optimization) [44]. This makes it difficult to correlate RTL signals with the resulting netlist, but it can be done [6].

When implementing an ECO using gate-array backfill, be sure to use the proper synthesis library. It will be different from the normal standard cell library [45].

Much work has been done on designing spare cells to support focused-ion-beam (FIB) repairs [25][26]. It is even possible to FIB a chip in a cavity-down package [46]–[48].

When you need to find a bus for a delta RTL fix, sometimes you can only find inverted versions of the bits. This is fine because once the signal is inside your delta RTL module, you can change the name to its proper form:

```
wire [2:0] tclo_offset = {
                !n6546,          // tclo_offset[2]
                \tos_offset[1] , // tclo_offset[1]
                n4368};          // tclo_offset[0]
```

In your delta RTL modules feel free to use Verilog-2001 (or SystemVerilog) features. Remember, these modules are only synthesized, never simulated.

If you have a subdesign that is multiply-instantiated, and you need to ECO only one instance, your RTL cannot match your netlist. Change your RTL to agree, or else ECO all the instances.

When searching for signals in a netlist, or writing a parser, be aware that Verilog allows whitespace where you least expect it. In particular a bus name can be separated from its index. Note that netname `bugfix_gnat6362_cell_data_in_mangled[17]` is wrapped across two lines:

```
NAND3 bugfix_gnat6579_U44 ( .Z(bugfix_gnat6362_cell_data_in_mangled
    [17]), .A(net_bugfix_gnat6579_274), .B(net_bugfix_gnat6579_275), .C(
    net_bugfix_gnat6579_276) );
```

Figure 18 on page 25 shows an example synthesis script for a delta RTL module. There are three `change_names` commands, and they are in careful order:

```
change_names -rules netnames
change_names -rules add_net
change_names -rules fix_ports
```

The first changes all netnames to have `bugfix_gnat1234_` prepended. The second then prepends `net_` onto that, so now all nets start with `net_bugfix_gnat1234_`. Finally the last makes sure that all nets attached to outputs will take the output name. (You can't combine the first two into one step, because it would require a recursive mapping rule which won't work.)

You might need to add more mapping rules to the `define_name_rules`. In particular if you synthesize DesignWare components you may get some screwy netnames and cellnames. Check the `report_net` and `report_cell` outputs to make sure all the names are as expected. Here is an example from a very complex delta RTL module:

```
define_name_rules fix_DW_cells -type cell -map { { \
    {"/","_"}, \
    {"^C",     "bugfix_gnat6579_C"} \
} }
define_name_rules netnames -type net -map { { \
    {"[","_"}, \
    {"/","_"}, \
    {"]",""}, \
    {"^n",     "bugfix_gnat6579_"} \
    {"^N",     "bugfix_gnat6579_"} \
    {"^C",     "bugfix_gnat6579_C"} \
} }
define_name_rules add_net -type net -map { {{"^bugfix","net_bugfix"}} }
define_name_rules fix_bussed_nets -type net -map { { \
    {"^dhad","net_bugfix_gnat6579_dhad"} \
    {"^sub_","net_bugfix_gnat6579_sub_"} \
} }

define_name_rules fix_ports -type net -equal_ports_nets

change_names -rules fix_DW_cells
change_names -rules netnames
change_names -rules add_net
change_names -rules fix_bussed_nets
change_names -rules fix_ports
```

Flow

Given the new RTL that incorporates the ECO fix, in addition to running your ECO flow, you should always run through your normal synthesis flow (Figure 48). This gives you a known good new top-level netlist to use for formal verification with the ECO top-level netlist. Because this comparison is gates-to-gates it can run much faster than comparing the new RTL with the ECO gates. Furthermore if we cannot derive an ECO fix, we already have the new netlist we need to start fresh…

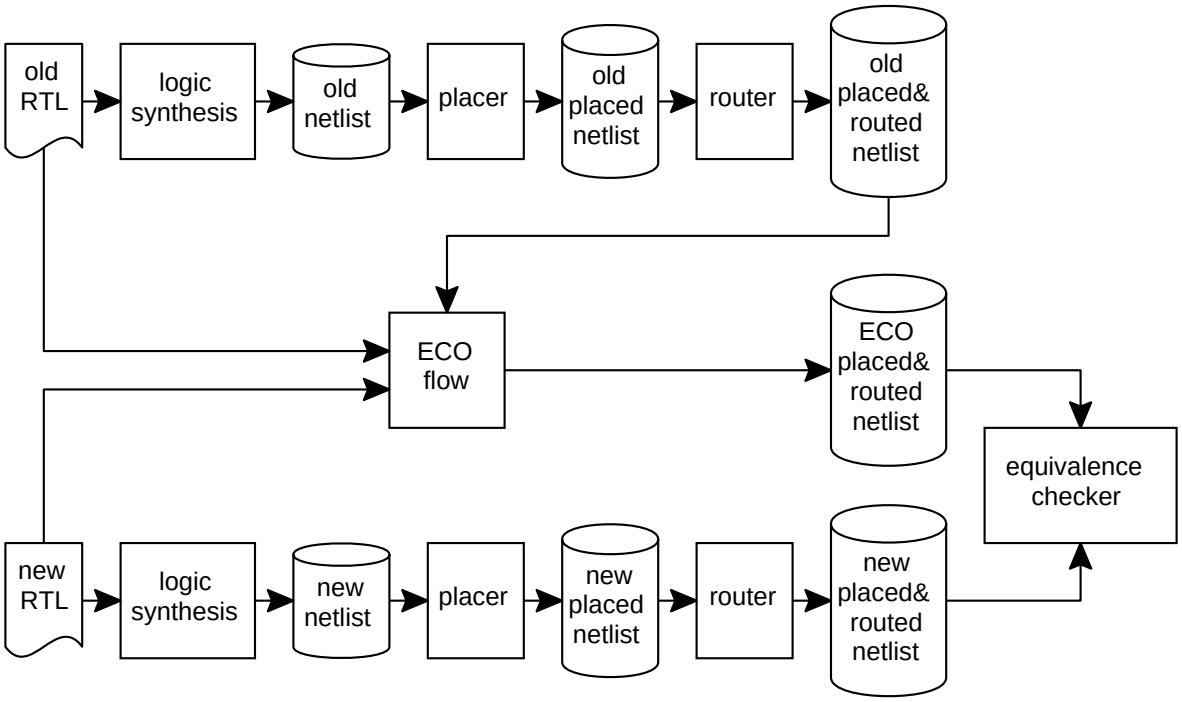

Figure 48. Run your ECO through the normal flow to enable gates-to-gates equivalence checking

With this ECO methodology we don't care about timing, just functionality. Your incremental place-and-route tool will have to make timing closure for you. Do you have block-level timing constraints that work with your place-and-route tool? Do they cover both setup and hold?

Sometimes your ECO fix will not agree with your RTL fix, because the fixes are at different levels of the design hierarchy. For example, consider a control signal needs to be inverted. The easiest place to invert it in the RTL might be the module that receives the signal. However in the gate-level netlist the easiest place might be to change the gate that creates the signal. This may cause the block-level formal verification to fail, however the top-level verification will work.

Always figure out your ECO fix at the module level first. Even if you `ungroup` and `compile -incremental` later you can still find many of the gates in common. If an `ungroup` is followed by a full `compile` then you have problems—there will be no agreement between gate names in the module-level netlist and the newly-compiled top netlist. (Nevertheless it still might be useful to do the module ECO fix first, because you will learn what to expect when dissecting the top-level netlist.)

In your normal synthesis flow, change the `compile_instance_name_prefix` and `compile_instance_name_suffix` variables to indicate which synthesis step you are performing. This makes the gates easier to trace in the netlist—you can tell which ones are from the initial compile, which are from top-level incremental compile, etc.

```
set compile_instance_name_suffix ""
set compile_instance_name_prefix "U"
compile                              ;# normal compile gates have Uxxx names
...
set compile_instance_name_prefix "I"
compile -incremental                 ;# incremental compile gates named Ixxx
...
set compile_instance_name_prefix "T"
compile -top                         ;# top compile gates named Txxx
...
ungroup -all -flatten
set compile_instance_name_prefix "Z"
compile -incremental                ;# flattened incremental gates names Zxxx
...
set_fix_hold clocks
set compile_instance_name_prefix "HOLD"
compile -only_design_rule            ;# hold fix gates named HOLDxxx
```

If your flow writes out a hierarchical netlist, it is helpful to split this file into individual modules. That way each module can be compared and analyzed individually—which modules have really been changed by your ECO? Many will be textually identical. Figure 49 is a Perl script that splits up a hierarchical netlist into separate files, one per module. You can use `cat` to glue them back together (be careful to concatenate them in the right order).

```
#!/usr/local/bin/perl
# splits a Verilog file into multiple files, one per module
while (<>) {
  if (/^module /) {
    split;
    ($module_name = $_[1]) =~ s/(.*$// ;
    close(FILE) if ($filename);
    $filename = $module_name.".v";
    open (FILE, "> $filename") or die "Could not open file $filename : $!\n";
    select FILE;
  }
  print if ($filename) ;
}
close(FILE) if ($filename);
```

Figure 49. split_verilog Perl script

You need to be aware of your *entire* flow. Make sure you are using the *final* netlist. Was the hierarchy modified? Was the netlist `group`ed or `ungroup`ed? Scan inserted? Clock tree inserted? `change_names`, explicit or implicit? What sort of optimizations were done during place-and-route? What netlist does your incremental place-and-route tool really need to see, to correctly perform your ECO?

If you are using a spare cells methodology, are the spares instanced in your RTL? If so, where in the logical hierarchy will you put them? If not, how will you run functional verification of your RTL vs. gates? How will you place the spares [49], and where on the die? What flavors of cells will you use, and how many spare cell kits should your design have?

Although this paper focused on functional ECOs, many more non-functional ECOs are routinely done for hold fixing, design rule fixing, etc. Similar implementation techniques can be used for both types of ECOs. Review the flow you may already have for these non-functional ECOs.

Fab

Consider holding back some wafers prior to metallization. You can then quickly turn around metal-only fixes [31]. What NRE fee will your vendor require for such a respin?

If you hold back some wafers, at what fab step do you need to stop? Be careful with the first contact layer—it might require a change if you modify your metal-1.

Bookkeeping

Every ECO must have a unique name. This is why bug tracking tools are so useful. If the bug hasn't been logged, don't work on the ECO.

When you `diff -c` the old and new RTL, print out the `diff` report. Use two different highlighter pens to mark up the differences. I use pink for the old RTL and yellow for the new. Only highlight areas that are actually different.

After tracing your schematic in Design Vision, print it out. Better yet, save the PostScript so you can print out copies at will. Now you can sketch potential ECO fixes right on the printed schematic.

Get a manila folder for each ECO. Write the ECO name on the folder tab. Keep the RTL `diff` reports, schematics, delta RTL sources, bug tracking reports, and other relevant documentation in it.

There are many many files used to implement ECOs—you need a naming convention. Unfortunately you won't realize all the different types of files you need until you've done several ECOs…

Stupid Management Tricks

One of the stupidest management tricks is to use revision tags like *bronze*, *silver*, and *gold*. What happens when you perform an ECO to the "golden" netlist? What will you call the netlist now, gold-prime? even more golden? polished gold? 18-karat gold? platinum? What about the next ECO? Your revision naming scheme is constraining your flow. Use numbers (or datestamps) to refer to your design versions, and *always* increment the version when the design changes. Always. Otherwise you'll be in for a world of hurt:

> Engineer One: "Here's the golden design."
> Engineer Two: "Do you mean this morning's golden design, or last week's golden design?"
> Engineer One: "Nope, it's the golden design that has Frank's fix, but not the restart bug fix."
> Engineer Two: "Oh, *that* golden design."

Other descriptive words to avoid include *last* and *final*. A sure-fire way to find a bug requiring an ECO is to conclusively state "This is the *final* netlist" and copy it into a directory named `final`.

If you are truly enamored of metallic names like silver and gold, then why not use the entire periodic table? Start with hydrogen and work your way up. Then you'll get to say things like "We completed the arsenic (As) netlist, and we expect to have the selenium (Se) netlist out by next week." I still say you should stick with integers; there's no upper bound.

ECOs allow you to appear to meet milestones when you really haven't. This can be useful when dealing with customers or management. Table 50 shows some examples.

Table 50: Management Translation Table

What we say	What we mean
RTL is frozen	…but there are a few pesky bugs, don't worry, after it's fixed we'll still call it the *golden RTL*
Synthesis is done	We're not about to run synthesis again, but we're still making changes to the netlist
Placement is done	We're still running incremental placement on a few ECO cells
Timing closure is complete	…and these three ECOs won't affect it at all
We've taped out the chip	We've taped out the base layers, and we're still adding metal-only ECOs
This respin is just for timing fixes to improve yield	…plus 3,600 gates repairing 8 functional bugs

When your boss says these things they are Stupid Management Tricks. When you say them they are Clever Engineering Techniques. Don't allow yourself to be fooled—remember you can't fool physics with a milestone chart—you aren't done until you are done.

8.0 Final comments

Many trends in the IC design world are conspiring to make this ECO flow an attractive solution for last-minute design changes:

- The back-end flow is becoming longer, more complex, and less predictable.
- NRE costs are increasing. Masks are expensive.
- Chips are getting bigger. Adding a few thousand ECO gates is no big deal.

We want to minimize the cost required for the change, where

$$\text{cost} = f(\text{time, money, licenses, people})$$

When you don't have enough time, just spend more of something else (i.e., money, licenses, people). This is what engineers do—we figure out how to solve a problem with the resources at hand.

Always expect more ECOs.

9.0 Acknowledgements

Thanks to the following people for their helpful discussions:

Pete Churchill, EdgeRate Consulting
Larry Dennison, Avici
Juergen Froessl, Synopsys
Nicholas Howorth, Intel
Steve Lamb, Synopsys
Rick Posch, Change Research Associates
Greg Squires, Blue Chip Design
Jon Stahl, Avici
Michael Stein, Paradigm Works
Erich Whitney, Axiowave Networks
Tim Wilson, Intel

Anonymous thanks to those design teams that unwittingly provided examples for this paper. You know who you are.

As always, a huge thank-you to Joanne Wegener of Synopsys for all her hard work.

10.0 References

[1] Charles H. Crockett, Jr. et al., “Method and apparatus for reworking printed circuit boards using surface coating and selective removal of an electrically conductive material,” U.S. Patent 5 615 387, March 25, 1997.

[2] Tracy Kidder, *The Soul of a New Machine*. Boston: Atlantic Monthly Press, 1981.

[3] Howard A. Landman, “Visualizing the Behavior of Logic Synthesis Algorithms,” *SNUG San Jose 1998*.

[4] “ECO Compiler Synthesis,” Synopsys, Technology Backgrounder, April 1997.

[5] Gitanjali Meher Swamy, “Incremental Methods for Formal Verification and Logic Synthesis,” PhD thesis, University of California at Berkeley, 1996. UMI publication 9723211. [Online]. Available: http://citeseer.nj.nec.com/swamy96incremental.html

[6] Laurent Arditi, Gerard Berry, and Michael Kishinevsky, “Late Design Changes (ECOs) for Sequentially Optimized High-Level Esterel Designs,” to be presented at *Designing Correct Circuits 2004*, Barcelona, Spain, 27–28 March 2004.

[7] “Tools and Technologies,” *integrated system design*, July 1997. [Online]. Available: http://www.eedesign.com/editorial/1997/toolsandtech9707.html

[8] “Synopsys Introduces First-Ever Synthesis Tool for ECOs,” *RSVP*, vol. 4 no. 1, pp. 36–37, Spring 1997. Synopsys. [Online]. Available: http://www.synopsys.com/news/pubs/rsvp/spr97/rsvp_spr97_10.html

[9] “ECO Compiler User Guide,” Synopsys, Version 2000.11, November 2000.

[10] John Cooley, *ESNUG*, SNUG 00 Item 21, April 5, 2000. [Online]. Available: http://www.deepchip.com/posts/snug00.html#21

[11] “Tips on ECO within the Physical Synthesis flow,” Synopsys, SolvNet Doc Id 000653, April 18, 2002. [Online]. Available: http://solvnet.synopsys.com/retrieve/000653.html

[12] John Cooley, *ESNUG*, ESNUG 358 Item 3, August 23, 2000. [Online]. Available: http://www.deepchip.com/posts/0358.html#3

[13] John Cooley, *ESNUG*, DAC 01 Item 27, July 31, 2001. [Online]. Available: http://www.deepchip.com/items/dac01-27.html

[14] John Cooley, *ESNUG*, ESNUG 404 Item 11, January 8, 2003. [Online]. Available: http://www.deepchip.com/posts/0404.html#11

[15] John Cooley, *ESNUG*, SNUG 03 Item 10, May 14, 2003. [Online]. Available: http://www.deepchip.com/items/snug03-10.html

[16] Andrey A. Nikitin, et al., “Direct transformation of engineering change orders to synthesized IC chip designs,” U.S. Patent 6 651 239, November 18, 2003.

[17] Anthony D. Drumm, “Incremental logic synthesis system for efficient revision of logic circuit designs,” U.S. Patent 5 436 849, July 25, 1995.

[18] Genichi Tanaka, “Engineering-change method of semiconductor circuit,” U.S. Patent 6 581 199, June 17, 2003.

[19] Mark Thomas Fox, “Methodology for ECO Compiler,” *SNUG San Jose 1999*.

[20] Kandimalla Babu Rao, “Verification of Gate level ECOs Using Compare_design,” *SNUG San Jose 1997*.

[21] John Horgans, John Pedicone, Greg Guiher, James P. Flynn, “Formal Verification: Verification of an ECO-Intensive Hierarchical Design,” *SNUG Europe 2003*.

[22] Shlomi Fish, ed. (2004, January). Computers: Software: Configuration Management: Tools. [Online]. Available: http://dmoz.org/Computers/Software/Configuration_Management/Tools/

[23] Shawn Clayton, John Sweeney, Mark Tetreault, and Scott Sandler, “A Set of Formal Applications,” *integrated system design*, November 1996. [Online]. Available: http://www.eedesign.com/editorial/1996/systemdesign9611.html

[24] Linas Vepstas. (2002, February). Call Center, Bug Tracking and Project Management Tools for Linux. [Online]. Available: http://linas.org/linux/pm.html

[25] Jacques Wong et al., “Efficient use of spare gates for post-silicon debug and enhancements,” U.S. Patent 6 255 845, July 3, 2001.

[26] Dennis Lee, “Method and apparatus for quick and reliable design modification on silicon,” U.S. Patent 5 696 943, December 9, 1997.

[27] Mark E. Masters et al., “Integrated circuit wiring,” U.S. Patent 6 307 162, October 23, 2001.

[28] Anthony Correale, “Method for providing engineering changes to LSI PLAs,” U.S. Patent 4 880 754, November 14, 1989.

[29] Doug Dreibelbis and Sheila Franz, “The Hidden Benefits of IBM ASICs:Part 1,” *MicroNews*, vol. 6 no. 3, pp. 1–6, August 2000. IBM Microelectronics. [Online]. Available: http://www.chips.ibm.com/micronews

[30] Elliot L. Gould, et al., “Method of combining gate array and standard cell circuits on a common semiconductor chip,” U.S. Patent 4 786 613, November 22, 1988.

[31] Robert L. Payne, “Cell-based integrated circuit design repair using gate array repair cells,” U.S. Patent 5 959 905, September 28, 1999.

[32] “SR40 0.095-µm High-Speed Copper Standard Cell/Gate Array ASIC,” Texas Instruments, Literature Number SRST142, March 4, 2002.

[33] Jon Stahl, personal communication.

[34] John Gregg, *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets*. New York: IEEE Press, 1998.

[35] Clifford E. Cummings, Don Mills, Steve Golson, “Asynchronous & Synchronous Reset Design Techniques - Part Deux,” *SNUG Boston 2003*. [Online]. Available: http://www.trilobyte.com/papers/#snug03_boston

[36] “Achieving Deterministic Results From Design Compiler,” Synopsys, SolvNet Doc Id 006062, June 2, 2003. [Online]. Available: http://solvnet.synopsys.com/retrieve/006062.html

[37] “Converting a report_test -scan_path to a scan,” Synopsys, SolvNet Doc Id 900774, June 26, 1998. [Online]. Available: http://solvnet.synopsys.com/retrieve/900774.html

[38] “Design with Test Compiler Test Logic,” Synopsys, SolvNet Doc Id 903425, August 11, 1998. [Online]. Available: http://solvnet.synopsys.com/retrieve/903425.html

[39] “Decrease Number of Net Names Changed by ECO_Compiler,” Synopsys, SolvNet Doc Id 901659, September 25, 1998. [Online]. Available: http://solvnet.synopsys.com/retrieve/901659.html

[40] “Control Port Name With eco_implement,” Synopsys, SolvNet Doc Id 903427, August 18, 1998. [Online]. Available: http://solvnet.synopsys.com/retrieve/903427.html

[41] Wilson Snyder. (2004). Verilog-Perl. [Online]. Available: http://www.veripool.com/verilog-perl.html

[42] “Using Formality To Perform Exact Structural Comparison of Gate-level Netlists,” Synopsys, SolvNet Doc Id 004400, June 16, 2003. [Online]. Available: http://solvnet.synopsys.com/retrieve/004400.html

[43] Steve Golson, “State machine design techniques for Verilog and VHDL,” *SNUG San Jose 1994*. [Online]. Available: http://www.trilobyte.com/papers/#snug94

[44] “What is DC Ultra?” Synopsys, SolvNet Doc Id 901707, August 19, 2002. [Online]. Available: http://solvnet.synopsys.com/retrieve/901707.html

[45] “Using gate array backfill as spare cells,” Synopsys, SolvNet Doc Id 901522, December 11, 1997. [Online]. Available: http://solvnet.synopsys.com/retrieve/901522.html

[46] Jian Li et al., “Circuit edit interconnect structure through the backside of an integrated circuit die,” U.S. Patent 6 376 919, April 23, 2002.

[47] Richard H. Livengood, “Method and apparatus providing a circuit edit structure through the back side of an integrated circuit die,” U.S. Patent 6 309 897, October 30, 2001.

[48] Richard H. Livengood et al., “Method for performing a circuit edit through the back side of an integrated circuit die,” U.S. Patent 5 904 486, May 18, 1999.

[49] “Methods for Handling Spare Cell Placement in Physical Compiler,” Synopsys, SolvNet Doc Id 900142, September 17, 2003. [Online]. Available: http://solvnet.synopsys.com/retrieve/900142.html

Additional Bibliography

David A. Morgan, “RTL annotation tool for layout induced netlist changes,” U.S. Patent 6 530 073, March 4, 2003.

Kuochun Lee, Tsung-Yen Chen, “Automatic engineering change order methodology,” U.S. Patent 6 453 454, September 17, 2002.

Chenmin Zhang, “Method and system for improving the performance of a circuit design verification tool,” U.S. Patent 6 226 777, May 1, 2001.