Register Duplication

for

Timing Closure

Version 1.2 - June 21st, 2011 - Quartus II 11.0 - by Ryan Scoville

Introduction:

This document addresses how duplicating registers can improve performance, as well as different methodologies for achieving this duplication.

Table of Contents

When to Duplicate

The basic scenario where register duplication might help timing is when analyzing a critical path in the design and seeing that the source register has a high fan-out and large IC delay. Here is an example:

Path #1: Setup slack is -0.198 (VIOLATED)

Path Summary | Statistics | Data Path | Waveform

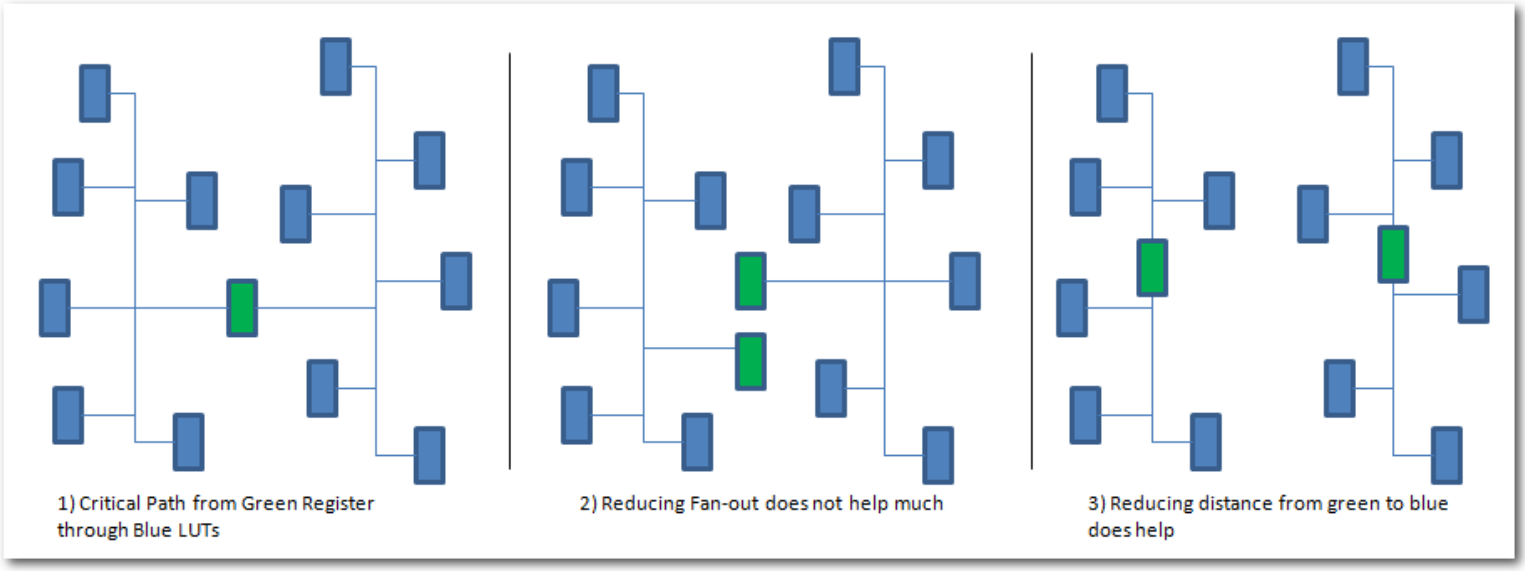
Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⊟ 3.281	3.281					clock path
3	3.281	3.281	R				clock network delay
4	⊟ 6.218	2.937					data path
5	3.366	0.085		uTco	1	FF_X29_Y23_N11	handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]
6	3.366	0.000	RR	CELL	2417	FF_X29_Y23_N11	G1.inst[3].input_lfsr|shiftreg_vector[0]|q
7	4.842	1.476	RR	IC	1	LABCELL_X56_Y27_N6	G1.inst[3].oc_stamp|desinst|u0|u4|X[24]|datac
8	5.097	0.255	RR	CELL	4	LABCELL_X56_Y27_N6	G1.inst[3].oc_stamp|desinst|u0|u4|X[24]|combout
9	5.280	0.183	RR	IC	1	LABCELL_X56_Y27_N26	G1.inst[3].oc_stamp|desinst|u0|u4|u3|Ram0~2|datab
10	5.741	0.461	RR	CELL	1	LABCELL_X56_Y27_N26	G1.inst[3].oc_stamp|desinst|u0|u4|u3|Ram0~2|combout
11	5.877	0.136	RR	IC	1	LABCELL_X56_Y27_N2	G1.inst[3].oc_stamp|desinst|u0|R4~27|datad
12	6.115	0.238	RF	CELL	1	LABCELL_X56_Y27_N2	G1.inst[3].oc_stamp|desinst|u0|R4~27|combout
13	6.115	0.000	FF	IC	1	FF_X56_Y27_N3	G1.inst[3].oc_stamp|desinst|u0|R4[10]|d
14	6.218	0.103	FF	CELL	1	FF_X56_Y27_N3	oc_des_des3perf:G1.inst[3].oc_stamp|des3:desinst|des:u0|R4[10]

As can be seen the IC(InterConnect) delay from the first FF is 1.476ns, and the Fanout of this register is 2,417. This is an excellent candidate for register duplication.

How Register Duplication Improves Timing

A common misunderstanding is that high fan-out nets are slow because of the large load put onto a single driver, and that by duplicating the source, each duplicate will have a smaller load and timing will improve. Although true for ASICs, this is generally not true for FPGAs. Note that FPGA architectures are highly re-buffered, and hence each independent route has a low fan-out. Just reducing the fan-out does not help timing very much at all. A large component of FPGA delays come from the actual routes themselves, independent of its load, and so duplicating a register is useful only if it reduces the routing distance. In the example below(which is a course drawing of FPGA placement and routing) there is a critical path from the green register through the blue LUTs.

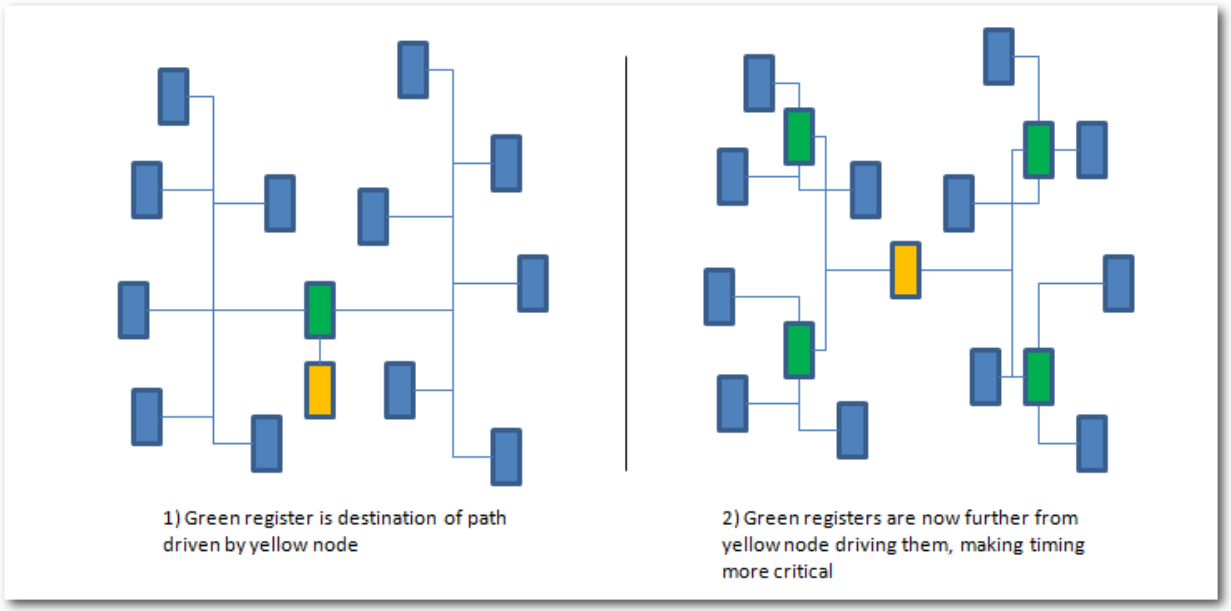


The middle picture has the source register duplicated, but the duplicate is placed next to the original. The load on each green register is reduced, but they still drive the same distance as before, so there will be minimal improvement in timing. In the third picture, the registers are moved closer to the nodes they are driving, reducing total interconnect and making the path faster.

Why is this important? The main reason is to realize that duplication works best when the source duplicates can be placed closer to the destinations. Let's pretend the blue nodes above are locked down, or must be placed in their respective left/right locations due to LogicLock constraints, I/O constraints, or just because the logic is pulled away to drive other logic. If the source register was duplicated in a way that drove some logic on the left and some on the right, there would be no way to move it closer to its destination, and register duplication would not provide much benefit.

Another scenario is when a register's fan-out is quite small, but they are to hard IP. For example, a register might feed 10 M20K blocks. A fan-out of 10 is not large, but because M20K blocks are dedicated silicon, the register cannot be placed near all of them, and there will be a large route delay to the furthest. Duplicating the register allows each duplicate to be placed closer to the M20Ks it drives, and that reduces the delay.

A secondary issue is to realize that we are "stealing slack" from the previous path. For example, let's look at a case where we duplicate the source register 4 times, allowing the placer to put it much closer to each destination. The green register is the source of the path we're interested in, but it's also the destination of other paths. In picture 1) on the left, it is driven by the yellow node:



Picture 2) on the right shows what happens after we have duplicated the green register four times. The green registers can now be placed much closer to their destinations and the slack on those paths are reduced. Conversely, the paths from the yellow node to the green registers have gotten longer, making the slack increase on those paths.

More often than not, the path leading up to the high fan-out register has lots of slack on it, and so stealing slack is not a problem. Before duplicating a register I will often run:

report_timing –setup –to [get_keepers {high_fanout_reg}] –panel_name "Slack we can steal"

This returns the slack on paths feeding the high fan-out register. If there is significant positive slack, than I feel more comfortable that the fitter can steal slack. If I forget to do this, then there's a chance those paths will show up as critical on the next pass, or that the gains I hoped for won't be as large as expected. This also leads into another issue that often comes up…

Duplicating Combinatorial Nodes for Performance

Sometimes the node with a high fan-out is not a register but a combinatorial node in the middle of a path. The user wants to address the problem the same way as a high fan-out register and tries to duplicate it. The problem, as described above, is that duplicating a node improves timing by stealing slack from the connections before it. If the node being duplicated is not a register but a combinatorial node, then the slack is stolen from the same path, and the net sum is that there isn't any improvement to the total path. In the picture above, let's say the green node is a combinatorial LUT with a large fan-out and part of a critical path in the design. If the user were to duplicate the green node four times so that it can be placed closer to its destination nodes as in 2), the paths to the destinations will get

shorter, but the path from the yellow node to the green ones grows by the same amount, and overall the critical path did not improve.

There is one method that can work with combinatorial logic, which requires duplicating not only the high fan-out combinatorial node, but all of the logic and registers before it. This can only be done in the RTL by duplicating the original registers in source code as well as all the logic after them, as shown in the Duplicating in HDL vs Assignment Editor section, specifically Duplicating Combinatorial Logic.

Three Methods for Duplicating Registers

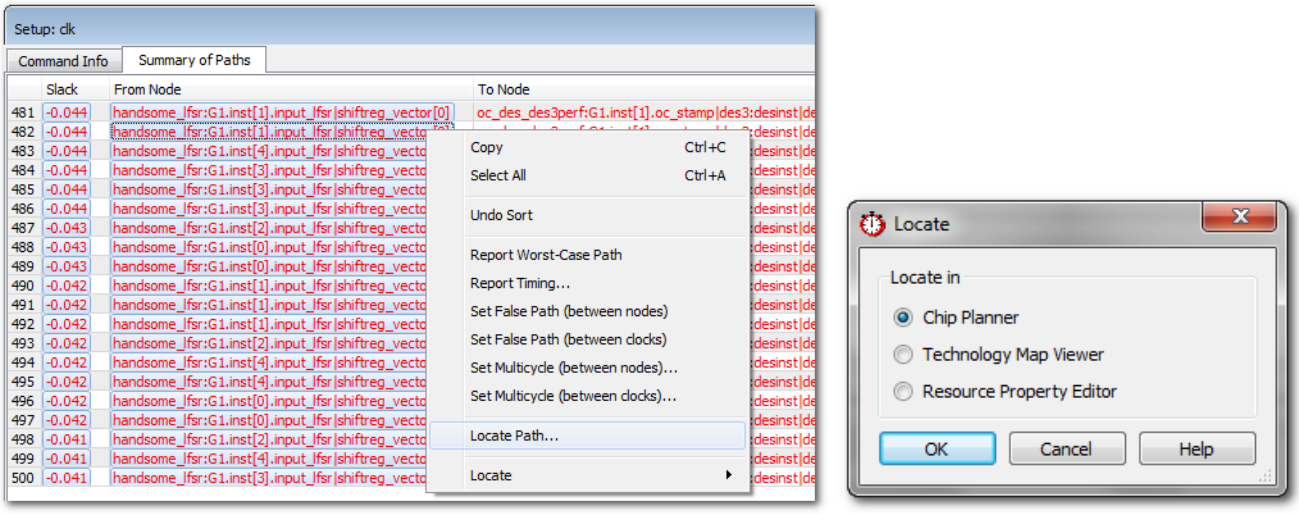
There are three basic methods for duplicating register:

1) Physical Synthesis Register Duplication
2) Maximum Fan-out
3) Manual Logic Duplication

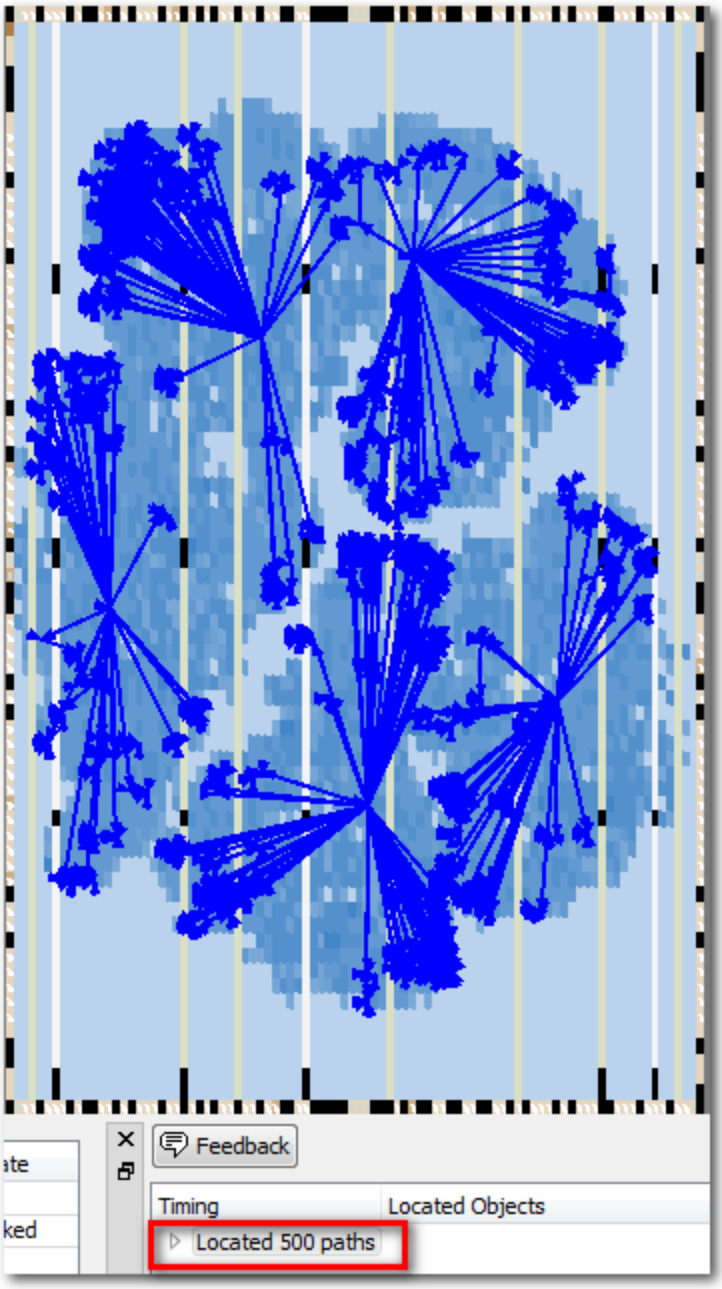
This order is based on difficulty, where the first option is nothing more than a checkbox, while the third option requires careful consideration on how to duplicate a register and what destinations it will drive. Let's start by analyzing a the critical path in a design:

report_timing -to_clock { clk } -setup -npaths 500 -detail full_path -panel_name {Setup: clk}

After listing the first 500 paths, I highlight them all in the Summary of Paths and right-click -> Locate Path -> Chip Planner.



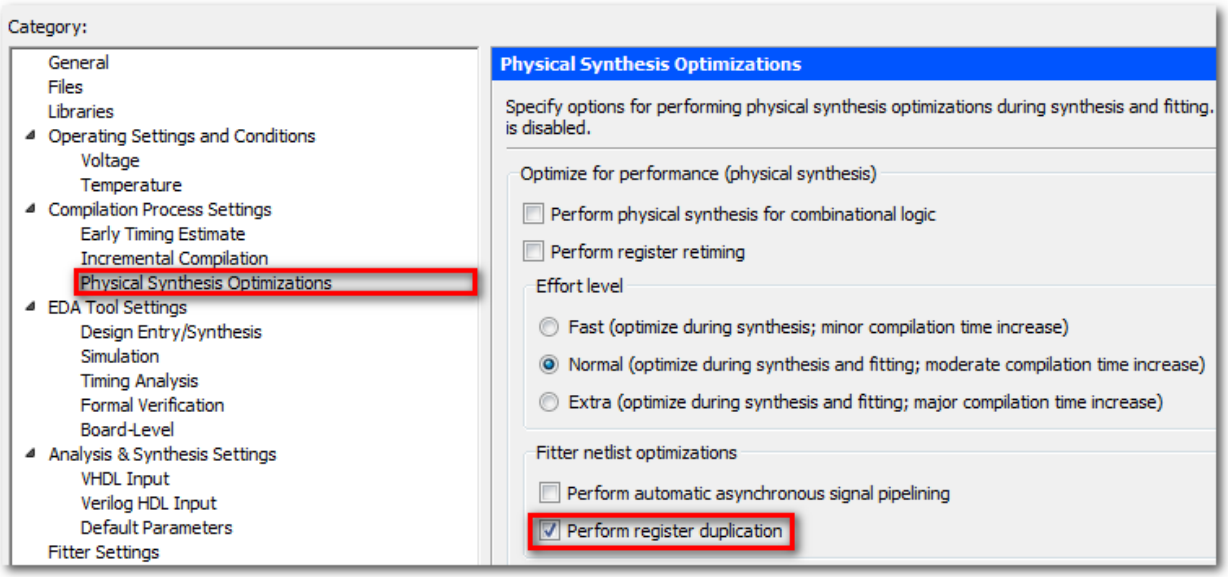
Note that all 500 paths will move to the Chip Planner's Located Objects window(new in Quartus II 10.0), but only the worst path will be shown in the Chip Planner. The user could select on individual paths to show them, or double-click on Located 500 Paths to show them all:



After double-clicking, we see all 500 paths. The design being analyzed has a core stamped out 5 times, and each instance has a single register with a fan-out of 2,417 where the worst interconnect delay is 1.476ns. This is a perfect case for analyzing logic duplication. (Most designs I have duplicated registers on have a much smaller fan-out, but it is still critical).

Register Duplication – Physical Synthesis

The simplest method is a check box found under Assignments -> Settings -> Physical Synthesis -> Perform Register Duplication:



For designs not meeting timing, this is the first place to start since it is easy. I turn on all three options "Perform physical synthesis for combinational logic", "Perform register retiming" and "Perform register duplication" to try and improve timing, without really differentiation between what the settings do. The only major downside to these options is that they increase run-time, although the increase is usually not significant with Normal Effort Level.

A secondary issue is that the fitter will determine what to replicate mid-fit. This means the fitter works for a while without knowing about the duplication, and will be pulling logic together that may not need to be once the duplication has occurred.

Finally, it may not duplicate what the user thinks should be duplicated. It may not realize the duplication is critical. It also has various rules on logic it won't duplicate. Examples include logic that drives an asynchronous reset, logic that is driven by an input port without timing constraints, etc. There are good reasons physical synthesis won't optimize these types of registers, in that they may cause a design to behave differently than the user's intent, and these broad algorithms are extra careful not to modify behavior.

I am listing potential issues only because the upside is pretty obvious, whereby the user clicks a checkbox and gets the improvements they want. I generally always start with this option.

Maximum Fan-out

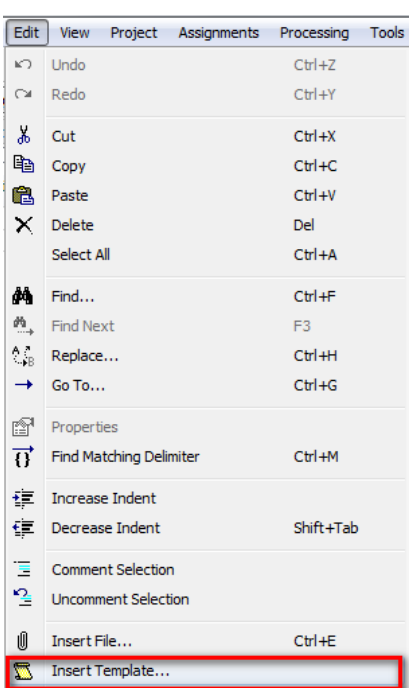
The second option for duplicating registers is the maximum fan-out constraint. This one is pretty commonly known, as it is often used in third-party synthesis tools. The user tells synthesis what the maximum fan-out of the register can be, and as soon as it meets that limit synthesis will duplicate the register. For the design being analyzed, there are 5 registers we want to apply the max fan-out

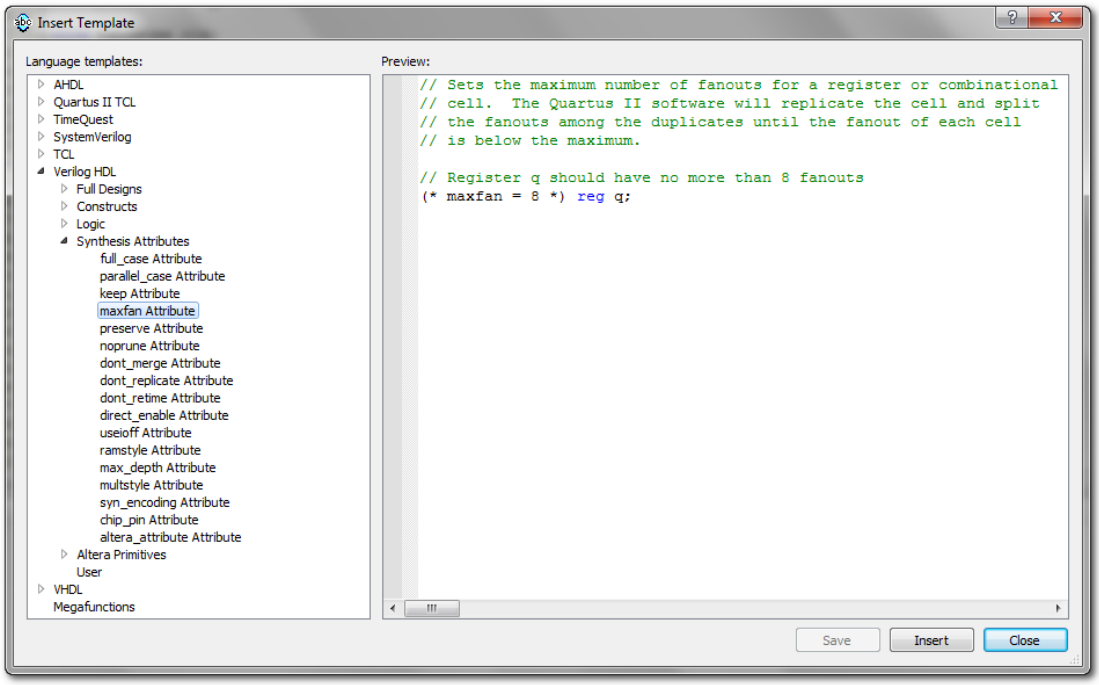
constraint to. I applied a max fan-out of 200. Since they have a fan-out of 2,417, there will be 13 total registers, twelve with a fan-out of 200 and one with a fan-out of 17.

An important point about maximum fan-out is that its duplication is "dumb", i.e. as soon as it counts a fan-out of 200, it duplicates the registers and starts counting again. If a register's fan-outs are in three different hierarchies, for example, a single duplicate may fan-out to all three hierarchies. There is nothing to prevent this. Now, if a register's fan-outs are all similar, this won't be a problem, but if a register fans out to clearly different destinations, max fan-out may not work as well as the user would like. We will see this when analyzing the maximum fan-out results.

The constraint can be entered directly in the RTL or as an assignment in the Assignment Editor, of which there are pros and cons of both methods, discussed here.

To enter through RTL, open the VHDL or Verilog file in Quartus and go to Edit -> Insert Template and browse to Verilog or VHDL Synthesis Attributes to find Max Fanout:





So for Verilog, the syntax looks like so:

(maxfan = 50*) reg highfanout_register;*

For VHDL, the syntax looks like so:

signal highfanout_register : std_logic;
attribute maxfan : natural;
attribute maxfan of highfanout_register :signal is 50;

These are from the templates, but Quartus understands other methods. For example, Synplify uses the *syn_maxfan* attribute, and Quartus synthesis will accept this too.

Another option besides modifying the source code is to enter the constraint through the Assignment Editor, so that it is stored in the .qsf. With the example project, I entered the following constraints:

	From	To	Assignment Name	Value	Enabled
1		handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]	Maximum Fan-Out	200	Yes
2		handsome_lfsr:G1.inst[1].input_lfsr|shiftreg_vector[0]	Maximum Fan-Out	200	Yes
3		handsome_lfsr:G1.inst[2].input_lfsr|shiftreg_vector[0]	Maximum Fan-Out	200	Yes
4		handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]	Maximum Fan-Out	200	Yes
5		handsome_lfsr:G1.inst[4].input_lfsr|shiftreg_vector[0]	Maximum Fan-Out	200	Yes
6		<<new>>	<<new>>		

Which get stored in the .qsf like so:

```
set_instance_assignment -name MAX_FANOUT 200 -to "handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]"
set_instance_assignment -name MAX_FANOUT 200 -to "handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]"
set_instance_assignment -name MAX_FANOUT 200 -to "handsome_lfsr:G1.inst[1].input_lfsr|shiftreg_vector[0]"
set_instance_assignment -name MAX_FANOUT 200 -to "handsome_lfsr:G1.inst[2].input_lfsr|shiftreg_vector[0]"
set_instance_assignment -name MAX_FANOUT 200 -to "handsome_lfsr:G1.inst[4].input_lfsr|shiftreg_vector[0]"
```

After compiling, one of the registers becomes thirteen individual registers with the following names:

handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_1
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_2
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_3
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_4
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_5
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_6
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_7
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_8
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_9
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_10
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]~SynDup_11

The first one has a fan-out of 17 and the rest have a fan-out of 200.

Manual Logic Duplication

The final option is manual logic duplication. This is generally the most difficult option, but gives the user more control. Not only do they control what register gets duplicated, but exactly what destination each duplicate fans out to. This makes the most sense when a register has distinct fan-outs

that would be placed in different locations, and hence the duplicates can be placed near them. The most common scenario is a register that fans out to separate hierarchies, which we will see in our example design's results.

Once again, there are two ways to make this assignment. It can be done in the RTL or through the Assignment Editor, which stores the constraint in the .qsf. Let's look at the Assignment Editor example first. The assignment is called Manual Logic Duplication. The From column has the name of the original register to be duplicated. The To column contains a wildcard of what destinations the new duplicate will feed, and the Value contains the new register name:

From = High fan-out register name
To = Wildcard of nodes the new duplicate register will feed
Assignment Name = Manual Logic Duplication
Value = Name of new register

In the example design, looking at the timing report, I noticed the high fan-out register feeds three distinct hierarchies. Because of that I want to create two duplicates for two of the hierarchies, leaving the original register to feed the third hierarchy. Since this high fan-out register exists five times in the design, that means 10 assignments need to be made:

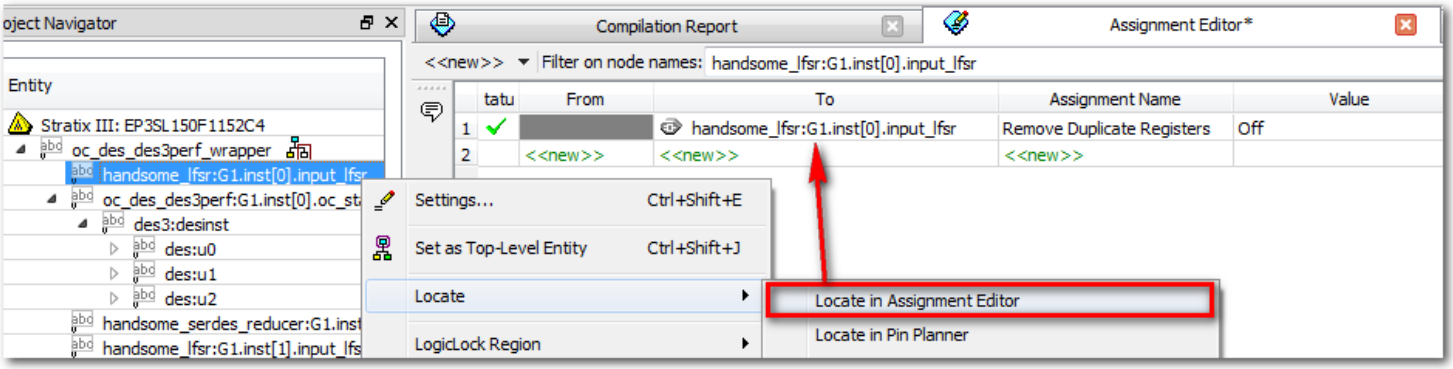
From	To	Assignment Name	Value
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]	*|des:u1|*	Manual Logic Duplication	shiftreg0_vector0_to_u1
handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]	*|des:u2|*	Manual Logic Duplication	shiftreg0_vector0_to_u2
handsome_lfsr:G1.inst[1].input_lfsr|shiftreg_vector[0]	*|des:u1|*	Manual Logic Duplication	shiftreg1_vector0_to_u1
handsome_lfsr:G1.inst[1].input_lfsr|shiftreg_vector[0]	*|des:u2|*	Manual Logic Duplication	shiftreg1_vector0_to_u2
handsome_lfsr:G1.inst[2].input_lfsr|shiftreg_vector[0]	*|des:u1|*	Manual Logic Duplication	shiftreg2_vector0_to_u1
handsome_lfsr:G1.inst[2].input_lfsr|shiftreg_vector[0]	*|des:u2|*	Manual Logic Duplication	shiftreg2_vector0_to_u2
handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]	*|des:u1|*	Manual Logic Duplication	shiftreg3_vector0_to_u1
handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]	*|des:u2|*	Manual Logic Duplication	shiftreg3_vector0_to_u2
handsome_lfsr:G1.inst[4].input_lfsr|shiftreg_vector[0]	*|des:u1|*	Manual Logic Duplication	shiftreg4_vector0_to_u1
handsome_lfsr:G1.inst[4].input_lfsr|shiftreg_vector[0]	*|des:u2|*	Manual Logic Duplication	shiftreg4_vector0_to_u2

These get stored in the .qsf like so:

```
set_instance_assignment -name DUPLICATE_ATOM shiftreg0_vector0_to_u1 -from "handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]" -to "*|des:u1|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg0_vector0_to_u2 -from "handsome_lfsr:G1.inst[0].input_lfsr|shiftreg_vector[0]" -to "*|des:u2|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg1_vector0_to_u1 -from "handsome_lfsr:G1.inst[1].input_lfsr|shiftreg_vector[0]" -to "*|des:u1|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg1_vector0_to_u2 -from "handsome_lfsr:G1.inst[1].input_lfsr|shiftreg_vector[0]" -to "*|des:u2|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg2_vector0_to_u1 -from "handsome_lfsr:G1.inst[2].input_lfsr|shiftreg_vector[0]" -to "*|des:u1|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg2_vector0_to_u2 -from "handsome_lfsr:G1.inst[2].input_lfsr|shiftreg_vector[0]" -to "*|des:u2|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg3_vector0_to_u1 -from "handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]" -to "*|des:u1|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg3_vector0_to_u2 -from "handsome_lfsr:G1.inst[3].input_lfsr|shiftreg_vector[0]" -to "*|des:u2|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg4_vector0_to_u1 -from "handsome_lfsr:G1.inst[4].input_lfsr|shiftreg_vector[0]" -to "*|des:u1|*"
set_instance_assignment -name DUPLICATE_ATOM shiftreg4_vector0_to_u2 -from "handsome_lfsr:G1.inst[4].input_lfsr|shiftreg_vector[0]" -to "*|des:u2|*"
```

The other option is to duplicate the registers in the source code. This just means creating registers in the RTL with the same behavior as the original and having them drive the intended destinations. Many a user has manually done this, only to be thwarted by Quartus, which defaults to merging all duplicate registers. There are a few ways around this:

- Stop Quartus synthesis from merging any duplicate registers in the design. This can be done under Assignments -> Settings -> Analysis & Synthesis -> More Settings -> Remove Duplicate Registers and setting it to Off. I generally don't recommend this because most designs have a surprisingly large number of unintended duplicate registers, and removing them tends to give better results. That is why this option defaults to being on.
- Prevent Quartus synthesis from removing duplicate registers in a specific hierarchy. This can be done by selecting the hierarchy where the registers have been duplicated(or one above it) and right-clicking Locate in Assignment Editor. Then copy the hierarchy name from the top Filter and paste it into the To column. Finally, select the Assignment Name called Remove Duplicate Registers, and set it to Off.



- Rather than assigning to a hierarchy, the Remove Duplicate Registers = Off can also be assigned directly to the register
- The final option is to apply a *dont_merge* attribute in the RTL. This can be added in a manner similar to the Max Fanout constraint, and is often helpful to use the Edit -> Insert Template from Quartus's text editor.

So for Verilog, the syntax looks like so:

(dont_merge *) reg highfanout_register;*

For VHDL, the syntax looks like so:

signal highfanout_register : std_logic;
attribute dont_merge : boolean;
attribute dont_merge of highfanout_register :signal is true;

The benefit of adding don't_merge into the code is that it keeps the duplication inside the RTL. The other methods use a combination of duplicating the logic in the RTL, while assignments to prevent

synthesis from merging those duplicates are made with project assignments. Of course, there are pros and cons to either method…

Duplicating in HDL versus Assignment Editor

As just shown max fan-out and manual logic duplication strategies can be done in the RTL or through assignments in the Assignment Editor, which get stored in the .qsf. There are pros and cons to both, which I will briefly discuss. Here are some of the issues:

Porting

When duplication assignments are in the HDL, it is very portable. The design hierarchy can change, the HDL can be moved to a new design, project, etc. and the assignments will follow. When the assignments are stored in the .qsf, the user must remember to carry them over to any new projects. If the hierarchy to the register changes, they must remember to modify the assignments(wildcards can help get around this). I have seen user make design changes that affect the hierarchy and find timing gets worse, since they forgot to modify the assignment paths in the .qsf. Making the assignments in the RTL gets around this problem.

Vendor Neutrality

Many users want their HDL to be as vendor neutral as possible, and hence avoid adding synthesis attributes like max fan-out values. Although I agree with this goal, I find assignments to be very unobtrusive, and when using a synthesis tool that does not understand the attribute, they usually ignore the assignment but they don't error out.

Reading

When user's are modifying RTL, it is much easier to see assignments they were not looking for. For example, if they want to change a register name, they will see the assignment right next to it and will be sure to keep it. If it were in the .qsf, the user probably would not know the register was being duplicated, and by changing the name they would stop that duplication. For user who wants to see "all the assignments", it can be difficult to search through hundreds of HDL files, while opening the .qsf and searching on the assignment is very straightforward. In summary, putting assignments in the HDL or .qsf are readable in different ways, and the user must decide what is important to them.

Duplicating Combinatorial Logic

The Manual Logic Duplication assignment in the Assignment Editor will only duplicate registers. By duplicating registers in the RTL, the user can also duplicate downstream combinatorial logic, which may be necessary for timing closure. For example, let's say a state-machine creates a combinatorial control signal that feeds two DDR3 interfaces, one on the top of the device and one on the bottom, and these paths consistently fail timing because of the distance they must travel. The user can duplicate the state-machine and all the control logic for this signal in the HDL, allowing one instance to be placed near the top DDR3 core and one near the bottom. This duplication of registers and logic can only be done in the RTL.

Accessing

More and more logic is now part of IP. In many cases the HDL can't be edited because it is encrypted. In other cases, the HDL is generated by a high-level tool like QSYS or Advanced DSP Builder, and any changes the user makes will be lost the next time the tool is run. These cases require assignments to be made through the Assignment Editor. I just worked on a QSYS design that had a bus feeding multiple pipeline bridges. I created a Tcl script to create assignments that manually duplicate a 16 bit bus for four of their destinations:

```
for {set i 0} {$i <= 15} {incr i} {
        set_instance_assignment -name DUPLICATE_ATOM dup_result$i\_2_bridgeA -from top:top|qsys:qsys|result\[$i\] -to
                top:top|pipeline_bridge:the_pipeline_bridge_A|*
        set_instance_assignment -name DUPLICATE_ATOM dup_result$i\_2_bridgeB -from top:top|qsys:qsys|result\[$i\] -to
                top:top|pipeline_bridge:the_pipeline_bridge_B|*
        set_instance_assignment -name DUPLICATE_ATOM dup_result$i\_2_bridgeC -from top:top|qsys:qsys|result\[$i\] -to
                top:top|pipeline_bridge:the_pipeline_bridge_C|*
        set_instance_assignment -name DUPLICATE_ATOM dup_result$i\_2_bridgeD -from top:top|qsys:qsys|result\[$i\] -to
                top:top|pipeline_bridge:the_pipeline_bridge_D|*
}
```

The names were modified from a real design. I put the above in a file called dup_regs.tcl, opened the project in Quartus, went to View -> Utility Windows -> Tcl Console and typed "source dup_regs.tcl" 64 assignments were made to make 64 duplicates, and the now the QSYS system can be regenerated without losing the duplication(assuming the user doesn't change hierarchy names).

Note that in the above script, the –from node matches the source register of a critical path, but the –to wildcard is not necessarily the hierarch of the destination register, because the source register did not directly feed the destination. Instead it goes through logic, and the –to value specifies which combinatorial LUT after the register should be fed by the duplicate. To find this, I listed out a lot of paths from the source register and individual clicked on them to find their second combinatorial node. If I mistakenly used the hierarchy of the destination register, and the source register didn't directly feed anything in that hierarchy, then the assignment would not do anything.

Results

The example design had a core stamped out 5 times. The critical path in each core began with a single register that had a fan-out of 2,417. The critical path looked like the TimeQuest screenshot shown in the beginning. This is the base compile, from which I tried three different approaches, Physical Synthesis Register Duplication, a Max Fan-Out of 200 on each register, and Manual Logic Duplication.

Base Compile:

Slack: -198ps

TNS: 14.439ns

Failing Paths: 413

Note: TNS = Total Negative Slack, which is the sum of negative slacks for each failing destination node. If a node has multiple failing paths leading to it, only the worst one is used.

Physical Synthesis – Register Duplication:

Slack: -198ps

TNS: 14.439ns

Failing Paths: 413

Analysis: These results are identical to the Base Compile, so that Register Duplication did not do anything. I submitted this to software to find out why, and the answer was that the register that needs to be duplicated is fed by an input port and there is no timing assignment on it. Because of this, if the register were duplicated, each duplicate would sample the input at different times and could potentially send different values to the downstream logic. To be safe, physical synthesis will not duplicate the register. This is a very good thing, as it could break the design, but points to the issue that physical synthesis has no input from the user, and hence the user has less control on what occurs.

As a test, I added a register before the one that needs to be duplicated. This allowed physical synthesis to duplicate the register and it got results very similar to the following Max Fan-out results.

Max Fan-out of 200(Assignment Editor)

Slack: -118ps

TNS: 0.637ns

Failing Paths: 20

Analysis: Although the slack didn't improve too much, the number of failing paths and TNS got considerably better. Note that each register had 12 duplicates, so there were a total of 5 instances X 1 = 65 registers to get this improvement.

Manual Logic Duplication(Assignment Editor)

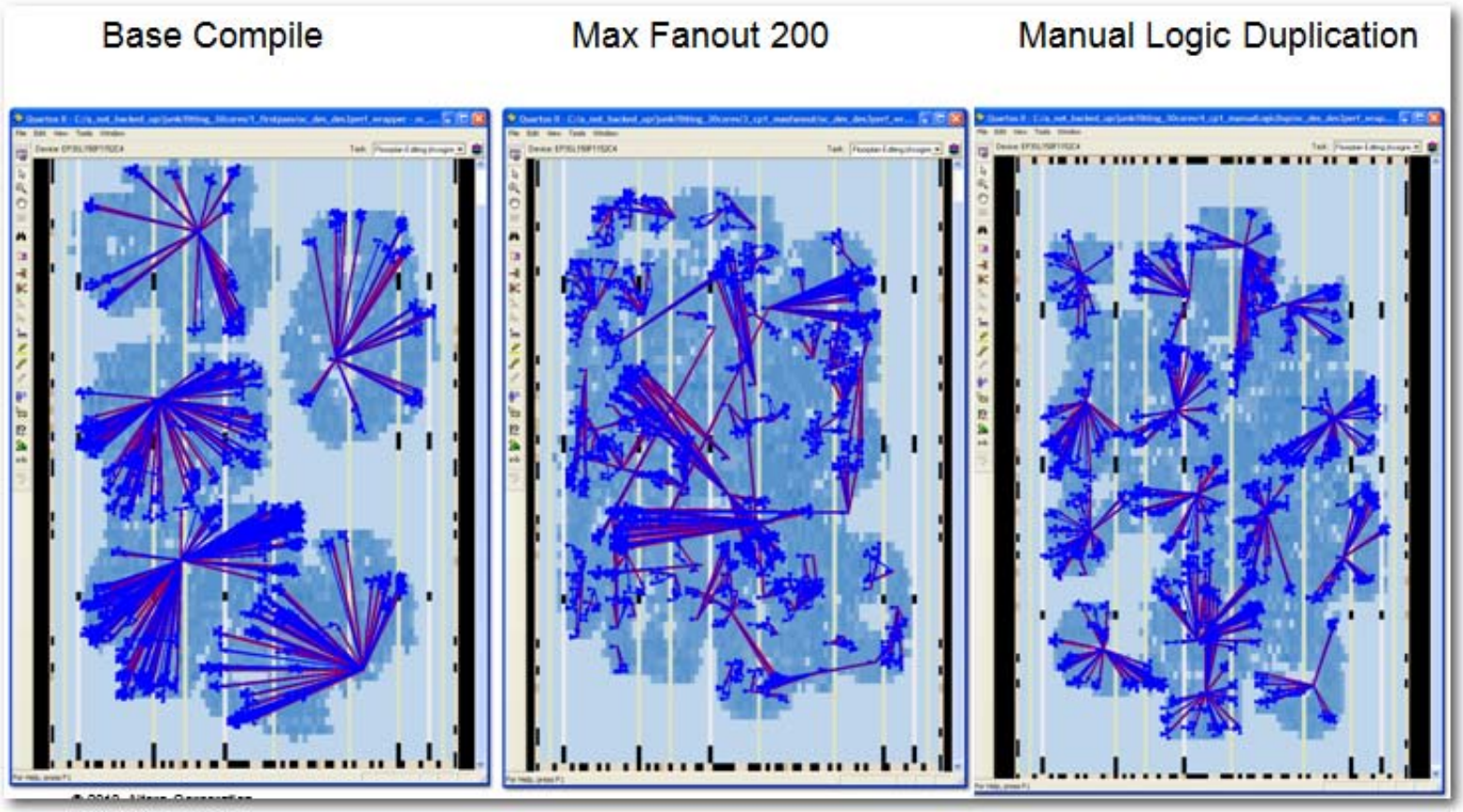
Slack: -49ps

TNS: 0.126ns

Failing Paths: 4

Analysis: This flow only made two duplicates of each register, or 5 instances X 3 = 15 total, and yet gave the best results. It was the most difficult, not just in making the assignment but in detecting that the signal fanned out to three distinct hierarchies.

Finally, let's look at the critical paths from the flows:



Once again, the base compile shows the 5 high fan-out registers, and their longest paths to the destinations furthest away. This creates 5 unique clusters. The max fan-out constraint shows many smaller fan-outs, but they are also much more random in nature. Finally the manual logic duplication created 15 distinct groups, which are much smaller and more controlled, mimicking the natural design hierarchy.

Final analysis:

In this design, although register duplication helped, the total improvement was only about 150ps. I have seen bigger improvements than this, but register duplication is not going to make huge improvements. This design is also somewhat unique in that all the critical paths are from the high fan-out registers. In most designs there are other paths having difficulty meeting timing. I have seen that duplicating registers allows the fitter to concentrate on other paths, and allows destinations of the high fan-out registers to move to locations that ease timing for other logic, making timing in the overall design improve. Register duplication is seldom the perfect trick for timing closure, but it is often a useful tool for chipping away at negative slack, and hopefully with this guide, allows more users to be successful.