Quartus University

Interface Program

Constrained Routing Tutorial and Reference

Version 5.0

March 17, 2005

By

Altera Corporation

Table of Contents

What’s New in Quartus II 5.0?

Nothing. If you are familiar with constrained routing from Quartus II 4.0, there will be nothing new for you in this document.

The document’s version number was changed to match the version of Quartus II software that the document describes.

1. Overview

This document describes how to use the constrained routing feature of the Quartus® II development tools. Constrained routing is a way to control the Quartus router through a text interface with various levels of detail. Quartus can also write out constrained routing (.rcf) files to "back-annotate", or permanently store, the routing generated by Quartus for a circuit. This .rcf file can be edited to slightly modify the routing, or could be entirely generated by a separate routing algorithm, and used to feed a complete routing into Quartus.

Before proceeding with this tutorial, you should be familiar with the basic operation of Quartus (how to create and compile simple designs, using either the GUI or command-line) and understand the QUIP™ Tutorial. You should also be familiar with Stratix™ devices and have a basic understanding of the function blocks and routing architecture.

You should go through this whole tutorial to learn the basics of how to use constrained routing. There are several appendices which are meant to serve as a reference for using more advanced features of constrained routing.

2. Typographic Conventions

Words written in **bold sans-serif font** are defined in Appendix B.

Boxed text highlights facts key to the proper operation of constrained routing.

Commands to be run from a command line are preceded by an angle bracket and written in **bold.** For example:

```
>        quartus_fit design
```

3. Using Constrained Routing

In order to learn how to use constrained routing, it is useful to first use Quartus to back annotate the routing for a simple circuit and look at the routing constraints file (RCF) that is created. The RCF contains human-readable information about the routing of each connection for a given circuit.

For this example, we're going to use the FNF design from the QUIP tutorial. It can be found in *tutorials/quip_tutorial/TutorialFiles* subdirectory. Here's how to back-annotate the routing using the GUI:

1. Run the quartus GUI, open the FNF project and compile it.
2. Go to *Assignments* menu and select *Back-Annotate Assignments….* Choose *Pin, cell, routing & device assignments* and click *OK*.

After you've back-annoted the routing, there will be a new file called FNF.rcf in your project directory. You can open this RCF file and look at it -- exactly what it all means is described in the next section. Note that if you compile this design from the GUI again, the router will reproduce exactly the same routing as the first run, by obeying the routing constraints in FNF.rcf. If you decide that you don't want to use the back-annotated routing anymore, you can simply erase the FNF.rcf file. Alternatively, you can remove routing assignments through GUI by going to *Assignments|Remove Assignments...*, checking “Pin, Location & Routing Assignments,” and clicking OK.

If you prefer to use the command-line to run Quartus, you can back-annotate the routing by following these steps:

1. In the FNF project directory, compile the design
   > quartus_fit FNF
2. Back-annotate the routing
   > quartus_cdb FNF – –back_annotate=routing

These steps will create the FNF.rcf file in the project directory.

Unlike the GUI, simply running the fitter again won’t use the RCF file to control the router. If you want to use the RCF file, you should run the fitter as follows:

> quartus_fit FNF – –set ROUTING_BACK_ANNOTATION_MODE=NORMAL

The extra option tells Fitter to use the RCF. Alternatively, if you put

set_global_assignment -name ROUTING_BACK_ANNOTATION_MODE NORMAL

into FNF.qsf, constrained routing will be enabled for all subsequent compiles without the extra fitter option. Removing this option from the qsf file will turn off the use of constrained routing.

Whenever you are running with constrained routing turned on, it is generally a good idea to check the generated Quartus output (ie. FNF.fit.rpt file). You should see a message like this: “Info: Using constrained routing from file <path/FNF.rcf>.”

4. RCF Format

4.1 Basic example

We will use the FNF.rcf file, given below, as an example to describe the format of an RCF file. Line numbers have been added down the left-hand side to assist in describing the format. The precise FNF.rcf file you obtain by running Quartus may be different, since different versions of Quartus will produce a different placement and routing for the circuit. The syntax will be the same as the example below.

```
 1      ### Routing Constraints File: FNF.rcf
 2      ### Written on:                         Wed Feb 04 15:14:50 2004
 3      ### Written by:                         Version 4.0 Build 191 2/2/2004
 4
 5      section global_data {
 6          rcf_written_by = "Quartus II 4.0 Build 186";
 7          device = EP1S10F484C5;
 8      }
 9
10      signal_name = Input1 {     #IOC_X1_Y0_N1
11          IO_DATAIN:X1Y0S1I0;
12          C4:X1Y1S0I25;
13          LOCAL_INTERCONNECT:X1Y1S0I15;
14          dest = ( InputReg1, SYNCH_DATA ), route_port = DATAC;       #LC_X1_Y1_N3
15      }
16      signal_name = Clock {      #IOC_X53_Y19_N3
17          CLK_BUFFER:X53Y19S3I0;
18          GLOBAL_CLK_H:X0Y19S0I11;
19          GLOBAL_CLK_V:X14Y0S0I11;
20          label = Label_LAB_CLK:X1Y1S0I7, LAB_CLK:X1Y1S0I7;
21          dest = ( OutputReg, CLK );    #LC_X1_Y1_N5
22
23          branch_point = Label_LAB_CLK:X1Y1S0I7;
24          dest = ( InputReg2, CLK );    #LC_X1_Y1_N4
25
26          branch_point = Label_LAB_CLK:X1Y1S0I7;
27          dest = ( InputReg1, CLK );    #LC_X1_Y1_N3
28      }
29      signal_name = Input2 {    #IOC_X1_Y0_N5
30          IO_DATAIN:X1Y0S5I0;
31          C4:X0Y1S0I31;
32          LOCAL_INTERCONNECT:X1Y1S0I25;
33          dest = ( InputReg2, SYNCH_DATA ), route_port = DATAC;       #LC_X1_Y1_N4
34      }
35      signal_name = InputReg1 { #LC_X1_Y1_N3
36          LOCAL_LINE:X1Y1S0I3;
37          dest = ( OutputReg, DATAA ), route_port = DATAC;    #LC_X1_Y1_N5
38      }
39      signal_name = InputReg2 { #LC_X1_Y1_N4
40          LOCAL_LINE:X1Y1S0I4;
41          dest = ( OutputReg, DATAB ), route_port = DATAD;    #LC_X1_Y1_N5
42      }
43      signal_name = OutputReg { #LC_X1_Y1_N5
44          LE_BUFFER:X1Y1S0I10;
45          C4:X1Y0S0I20;
46          LOCAL_INTERCONNECT:X1Y0S0I7;
47          IO_DATAOUT:X1Y0S0I0;
48          dest = ( OutputPad, DATAIN ); #IOC_X1_Y0_N0
49      }
```

Comments

Anywhere the pound sign '#' appears denotes the start of a comment. Comments can start anywhere on a line and mean that the rest of the line is considered a comment. Examples are lines 1 and 10.

Global Data Section

This section is optional. It can contain a string identifying the Quartus version that generated the file and device used in the project. The device information is used to ensure that if the project is compiled again, that the device used for the project matches the RCF, since if the selected device is changed by the user, the routing constraints will no longer be useful. Lines 5-8 are an example of global data.

Routing Constraints

Line 10 begins the routing constraint for a signal. This signal consists of a single connection – from an IO block to an LE. Identifier `signal_name` specifies which signal we are constraining. In this case, it is Input1, which is coming from the output of the IO. Note that when a design is back-annotated, a comment is created at the end of the line to tell us the block type of the source of this signal and where the block was located (placed) at the time the RCF was written. In this case, the source originates from an IO block located at position x=1, y=0, sublocation=1. Refer to [1] a for description of the location format.

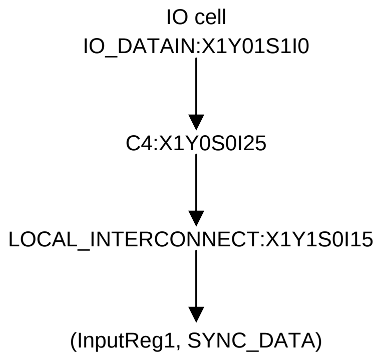
Lines 11-13 list the routing resources used to route the connection, in the order they are used. `IO_DATAIN` is a buffer that is the output of the IO block, connecting to the chip's core. Each routing resource is following by it's location. For example "IO_DATAIN:X1Y0S1I0", means the `IO_DATAIN` resource at x=1, y=0, sub-location=1, index=0. Index is used because there could be multiple resources of exactly the same type at the same (x, y, sub-location). The next resource is a `C4` wire, which is a vertical wire of length 4. The last used routing resource is a local interconnect at x=1, y=1.

For routing constraints to be valid, each routing resource must be able to drive the next listed resource. (i.e. the connections must be possible in the device)

Line 14 specifies the destination to which we are routing. For a signal with a single fanout, there is only one destination (sink) terminal. However, signals can fan out to several destinations and it is important that the RCF file clearly indicate which routing resources are to be used to route to which destination. Line 14 shows that a destination is indicated by the block and port on that block, to which this signal is connected in the vqm netlist [2, 3]. Here, `InputReg1` refers to a logic cell and `SYNCH_DATA` is an input port on the `InputReg1` block. When referring to a block by name, as in line 14, you can use either the name of any output signal of the block or the "instance name" given to the block [3].

In line 14, `route_port = DATAC` is an optional specification that can be used to force the router to use a specific input port to a block For example, the circuit netlist (visible if you generate a vqm file from Quartus) may specify that this signal should connect to port A on a logic cell. It may be possible to route the connect using port C, however, to improve circuit speed or routability. Specifying `route_port = DATAC` will force the signal to be routed to this destination using the DATAC input to the logic cell. If route_port is not specified, the router is free to use any input port that is legal for this connection to route it into the destination block.

Here is a pictorial representation of routing we just considered:



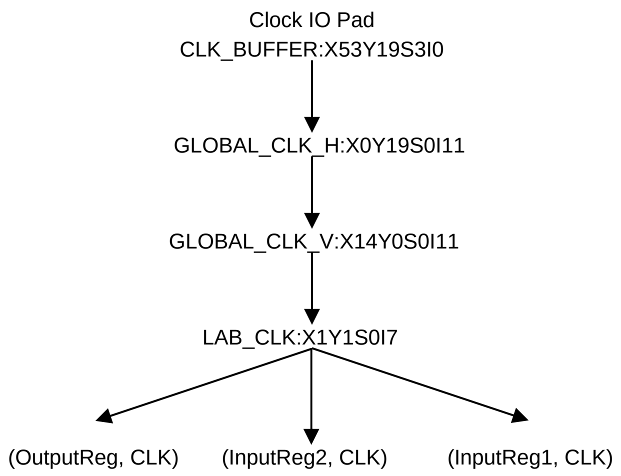
Lines 16-28 contain routing constraints for signal `Clock`. There are two major differences between the routing of `Clock` and `Input1`. The first difference has to do with the fact that `Clock` is a **global signal**. Global signals are routed on a special network that minimizes skew. Altera recommends leaving routing of global signals to Quartus.

The second difference that makes routing constraints for `Clock` look more complicated is that `Clock` is a fanout 3 signal in this circuit, and hence consists of three connections. This brings up the problem of representing a tree structure of routing resources in a flattened form. RCF solves the problem using `label` and `branch_point` identifiers.

On line 20 the RCF file preceded the routing resource `LAB_CLK:X1Y1S0I7` by statement `label = Label_LAB_CLK:X1Y1S0I7`, indicating that this routing resource will be used to start the routing for other connections. Command `branch_point = Label_LAB_CLK:X1Y1S0I7` on line 23 tells the router to start routing the next connection from routing resource labelled `Label_LAB_CLK:X1Y1S0I7`. If branch point is not used, routing starts at the source of the signal. This is the case for the first routed connection of any net. The label can be any text – we could have used the label `"my_label"` instead of `Label_LAB_CLK: X1Y1S0I7` in the example above.

Labelling a node and not using it later is OK. However, using a node twice without label/branch_point identifiers is illegal.

Here is a pictorial representation of the clock signal routing:



The rest of the RCF does not contain any new syntax features. The only other feature you may see in Quartus-generated RCF's is the `ram_sublocation` section. It will appear at the end of RCF for designs containing RAMs. This provides additional information concerning RAM port re-ordering needed to reconstruct a routing for circuits with RAM. This document will not describe this re-ordering information in detail, as it is quite complex. Do not modify this section of data; if you wish to work on RAM routing, contact quip@altera.com for information on this section of the RCF.

4.2 Modifying Routing Constraints

It is possible to modify an RCF created by Quartus to change the routing of a design, as long as the altered routing connectivity is possible in the device. We will modify FNF.rcf slightly, and then verify that the router obeyed the new routes we specified.

Open FNF.rcf in any text editor and replace lines 12 and 13:

```
        C4:X1Y1S0I25;
        LOCAL_INTERCONNECT:X1Y1S0I15;
```

with

```
        C4:X1Y1S0I37;
        R4:X0Y1S0I134;
        LOCAL_INTERCONNECT:X1Y1S0I27;
```

Re-run the fitter, either from the GUI or command-line, as shown in Section 3.

There are two ways to confirm that the new routing was accepted by the fitter. We recommend that you use both, at least for this tutorial. The first method is to write out the RCF again (doing so will overwrite current one!). After writing out the RCF file again, you should find the new routing in it.

The second method is to use the GUI. To use the GUI in order to confirm that the fitter obeyed the new routing, follow these steps:

1. If you do not have Quartus II GUI opened, do so now by running **quartus** from command prompt.
2. Open the FNF project using *File|Open Project*.
3. Click on *Processing|Compilation Report*
4. Expand *Fitter* folder and click on *Floorplan View*
5. Find the *Input1* node on the floorplan. The easiest way to do this is to press Ctrl+F to bring up the find box and type "Input1."
6. Find will pan to the appropriate section of the floorplan and highlight the node.
7. Move your mouse pointer over the node. A Tool tip will show you routing originating from this node.

In both cases you should see that new routing constraints were obeyed.

4.3 Coordinate notation

In order to use routing constraints properly, it is important to understand the coordinate system used for routing resources. Every routing resource name in RCF is followed by a string of the form `X_Y_S_I_`, where `"_"` stands for some integer. Location of signal sources and destinations (blocks) is of the same form, except the `I` field is absent.

`"X"` and `"Y"` fields refer to the coordinates of left end of horizontal wires, the bottom end of vertical wires, or the furthest counter-clockwise end of IO Bus wires. For example, `R8:X3Y1S_I_` is a length-8 horizontal wire that goes from (3, 1) to (10, 1). Notice that a vertical wire end point has the same (x, y) coordinate as the block to its left, and a horizontal wire end point has the same (x, y) coordinate as the block below it. Figure 1 illustrates the coordinate notation, and [4] has further details on the Stratix coordinate system.

The `"S"` field stores the sublocation. A given (x, y) location can have multiple sub blocks within it. For example, a LAB contains ten logic cells with sub locations ranging from 0 to 9. Think of sublocation as the third dimension of the chip. For wires, the sublocation field is not used – it is always set to 0.

The `"I"` field stores index information. It has no physical meaning and is used to distinguish wires or buffers of the same type that originate from the same (x, y) location.

A few resources use both index and sublocation as part of their coordinate. For example, `IO_DATAIN`, which represents the output pin from an IO cell can have non-zero values for both sublocation and index fields. The sublocation corresponds to the sublocation of the IO cell to which each IO_DATAIN belongs. Each IO has two IO_DATAIN resources, so they have indices 0 and 1 to distinguish them.

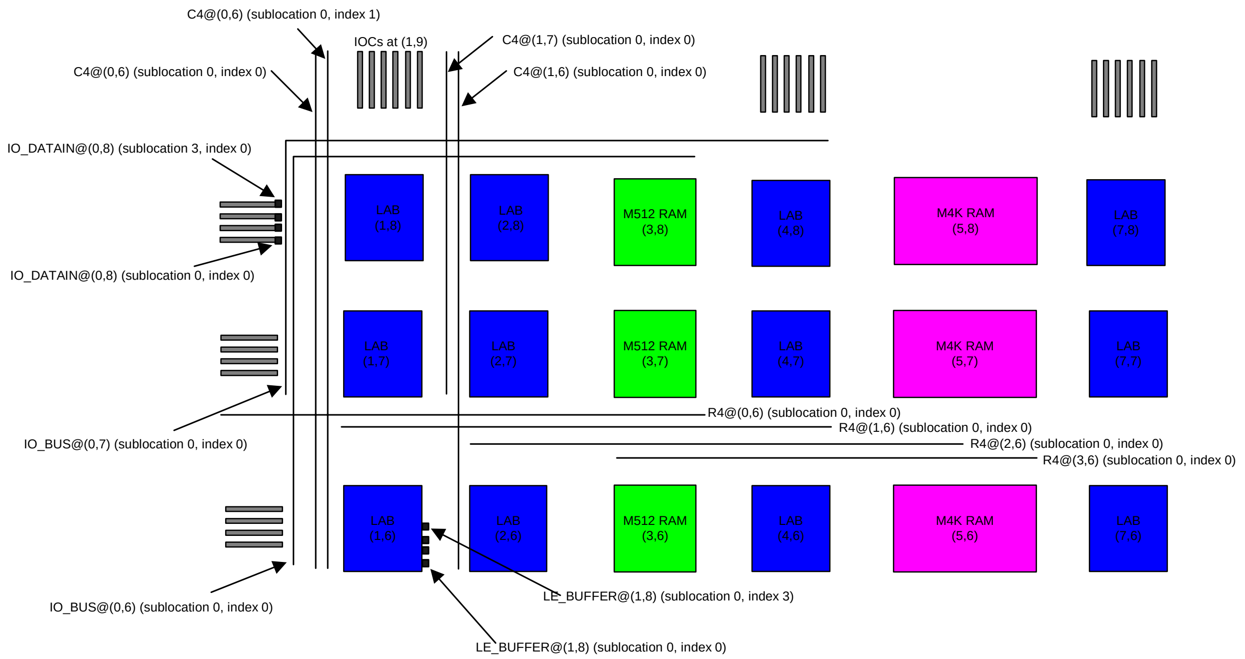

C4@(0,6) (sublocation 0, index 1)
IOCs at (1,9)
C4@(1,7) (sublocation 0, index 0)
C4@(0,6) (sublocation 0, index 0)
C4@(1,6) (sublocation 0, index 0)
IO_DATAIN@(0,8) (sublocation 3, index 0)
IO_DATAIN@(0,8) (sublocation 0, index 0)
LAB (1,8)
LAB (2,8)
M512 RAM (3,8)
LAB (4,8)
M4K RAM (5,8)
LAB (7,8)
LAB (1,7)
LAB (2,7)
M512 RAM (3,7)
LAB (4,7)
M4K RAM (5,7)
LAB (7,7)
R4@(0,6) (sublocation 0, index 0)
R4@(1,6) (sublocation 0, index 0)
R4@(2,6) (sublocation 0, index 0)
R4@(3,6) (sublocation 0, index 0)
IO_BUS@(0,7) (sublocation 0, index 0)
LAB (1,6)
LAB (2,6)
M512 RAM (3,6)
LAB (4,6)
M4K RAM (5,6)
LAB (7,6)
LE_BUFFER@(1,8) (sublocation 0, index 3)
IO_BUS@(0,6) (sublocation 0, index 0)
LE_BUFFER@(1,8) (sublocation 0, index 0)

Figure 1: Routing Coordinate System

4.4 Routing Elements

The tables below lists the common routing resources you will encounter on Stratix family chips, their uses, and special comments. The first column in each table gives the prefix to be used to name each wire. Specific wires are referred to in RCF files as <prefix>:<coordinate>. For example, a vertical length 4 wire at (x=51, y=27, sub-location = 0, index = 10) is:

```
C4:X51Y27S0I10;
```

Since constrained routing can also be used with the Cyclone device family, and uses the same syntax as Stratix, we also highlight which routing resource types are not available in Cyclone.

General Wires

C4	Vertical Wire Length 4
C8	Vertical Wire Length 8†
C16	Vertical Wire Length 16†
R4	Horizontal Wire Length 4
R8	Horizontal Wire Length 8†
R24	Horizontal Wire Length 24†
LOCAL_INTERCONNECT	Used to enter LAB from general routing.
LOCAL_LINE	Used to connect logic cells within one LAB and its neighbours.
IO_BUS	IO Bus Wire. It goes along chip periphery spanning 24 IO blocks (equals 24 rows or 48 columns) †

† Not available in Cyclone.

Function Block Output Buffers

Note that the function blocks are not directly represented in the FPGA routing resources. Only the output buffer nodes of the function blocks are represented in the routing resources.

IO_DATAIN	Output of an IO block
LE_BUFFER	Logic Element Output Buffer
DSP_BUFFER	MAC Output Buffer†
M512_BUFFER	Output Buffer of M512 RAM block†
M4K_BUFFER	Output Buffer of M4K RAM block
MRAM_BUFFER	Output Buffer of Mega-RAM block†
PLL_BUFFER	PLL Output Buffer

† Not available in Cyclone.

Global Clock Wires

GLOBAL_CLK_H	Horizontal Global Clock Wire
REGIONAL_CLK_H	Horizontal Local Clock Wire
GLOBAL_CLK_V	Vertical Global Clock Wire
REGIONAL_CLK_V	Vertical Local Clock Wire
LAB_CLK	Lab Clock
LR_IO_CLK	Left-Right IO Clock, used to drive IOs on left and right sides of the chip.
TB_IO_CLK	Top-Bottom IO Clock, drives IOs on top and bottom parts of the chip
CLK_BUFFER	Used to connect outputs of IOs to global routing network.

Miscellaneous

IO_DATAOUT	Input into an IO block
IO_BUS_BUFFER	IO Bus Buffer. Required to get off an IO Bus wire†
LUT_CHAIN	Fast Cascade out (LUT chain)
M512_CONTROL_INPUT	Input for global signals into M512 RAM block†
M4K_LEFT_CONTROL_INPUT	Input on left side of M4K RAM block, used by global signals
M4K_RIGHT_CONTROL_INPUT	Input on right side of M4K RAM block, used by global signals
LAB_CONTROL_MUX	LAB secondary MUX. Used by global signals to enter LAB

† Not available in Cyclone.

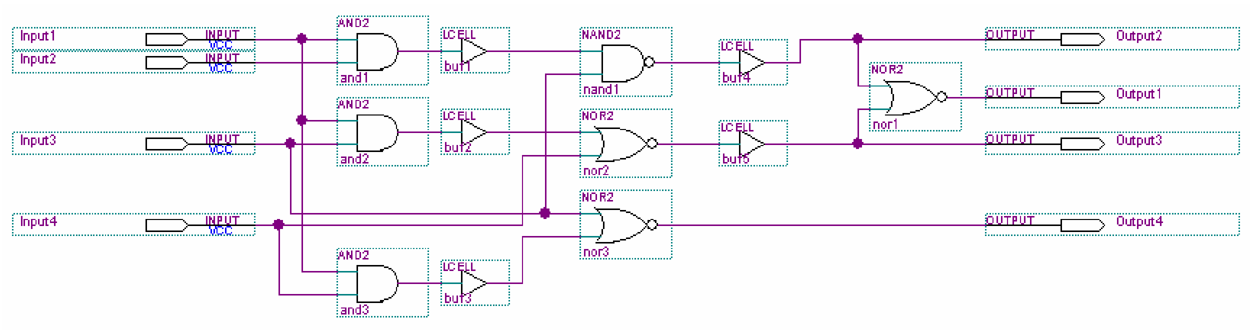
There are other, special-purpose, routing resources available that are not listed above.

4.5 More RCF Features: Wildcards, etc.

By now, you should be able to understand any routing constraints file generated by Quartus. However, there are many features of the constrained routing language which are not used by Quartus when it writes out a routing. To highlight these features, we will go through an example where we manually write constraints that use more advanced RCF features.

In this section we will be working with a slightly more complicated design. The project is called *elaborate_rcf* and is located in *constrained_routing/examples/elaborate_rcf* subdirectory. The design has one special feature – the logic is already locked down. Logic was spread all over the chip to create long routing paths that we can work with. You can view the location constraints by opening the *elaborate_rcf.qsf* file in a text editor, and searching for lines that begin with "set_location_assignment" keyword. The format of these location constraints is described in [1].

The circuit is illustrated in schematic form below:



Quartus will automatically read a routing constraints file named <project_name>.rcf, so the routing constraints file we will use in this example is called *elaborate_rcf.rcf*. The file's contents are:

```
section global_data {
    device = EP1S10B672C6;
}

signal_name = Input1 {     #IOC_X1_Y31_N2
    label = some_io_datain, IO_DATAIN:X1Y31S2I*;
    C16:*;
    *;
    R4:X0Y1S0I62;
    LOCAL_INTERCONNECT:X1Y1S0I6;
    dest = ( buf1, DATAA );       #LC_X1_Y1_N5

    branch_point = some_io_datain;
    IO_BUS:X1Y31S0I0;
    IO_BUS_BUFFER:X45-52Y31S0I0-100;
16          C8:X52Y15-25S0I*;
17          LOCAL_INTERCONNECT:X52Y30S0I19 || LOCAL_INTERCONNECT:X52Y30S0I23;
18          dest = ( buf3 );      #LC_X52_Y30_N9
19
20          branch_point = some_io_datain;
21          *;
22          *;
23          dest = ( buf2, DATAB );       #LC_X1_Y30_N2
24      }
25      signal_name = Input3 {    #IOC_X0_Y30_N2
26          IO_DATAIN:*;
27          zero_or_more, C8:* || R8:*;
28          LOCAL_INTERCONNECT:*;
29          dest = ( nor3, DATAA );       #LC_X33_Y7_N8
30      }
31      signal_name = Input2 {    #IOC_X0_Y30_N3
32          zero_or_more, *;
33          dest = ( buf1, DATAB ), route_port = DATAA;  #LC_X1_Y1_N5
34      }
35      signal_name = Input4 {    #IOC_X0_Y30_N0
36          IO_DATAIN:X0Y30S0I0;
37          C16:*;
38          dest = ( buf5, DATAA ), route_port = DATAB;  #LC_X1_Y30_N6
39      }
```

The first thing to notice is that not every net has a routing constraint. For nets that do not have constraints, the router will route them as it sees fit to obtain the best routability and/or circuit timing. Second, notice that net `Input4` drives two destinations (`buf3` and `buf5`) but routing is constrained to only one of them (`buf5`). This tells the router to route to `buf5` according to routing constraints, and then to the remaining connection however it pleases. Therefore, you can create routing constraints for specific nets and even just for particular connections on a net, without specifying the routing of the entire circuit.

This new RCF highlights some more features in the RCF format:

Line 6 shows that label names can be arbitrary strings (see Appendix on RCF Grammar for special characters that cannot appear in a string).

Next, notice there is a `*' character right before the semi-colon on Line 6. This is the wild-card character. It tells the router that you do not care about the value in that field. So the router is free to choose any IO_DATAIN resource among all those located at (1, 31), sub location 2 – any index is OK.

Line 7 is an example of wild-carding the whole coordinate section. This line tells the router to use any C16 wire. However, the router is not free to use absolutely *any* wire. It must pick a wire that is driven by the previous routing resource, IO_DATAIN, and it must drive the next routing resource, in order to form a legal route. Connectivity restrictions like this are always obeyed by the router.

Line 8 is an example of wild-carding the whole resource. This allows the router to use any *one* routing resource that was not used so far.

Line 11 does not contain a `route_port` setting. As we already mentioned, this piece of information is not required, the router will then just choose whatever input port to the logic cell seems best, as long as it is electrically legal.

Line 15 introduces the range specifier. Range can be given to more than one coordinate simultaneously. The constraint on this line tells the router to use an IO Bus Buffer with an x-coordinate between 45 and 52, a y-coordinate of exactly 31, and index between 0 and 100.

Line 16 shows how you can combine range and the wild-card. This constraint allows the router to use only C8's with x-coordinate 52, y-coordinate between 15 and 25, and any value of index.

An “or” symbol, `||`, is used on line 17. The constraint tells the router to use a local interconnect at (52, 30) with index being either 19 or 23. An arbitrary number of routing resources can be or’ed in a single line.

On line 18 we see that omitting the destination port name in a connection is allowed. If the signal connects only once to the block, the router can figure out which connection you are constraining without being given a destination port. However, it is a dangerous practice to omit port names in connection specifiers. Some Quartus optimizations can merge two destination blocks of one net together, potentially introducing an ambiguity about which connection you were attempting to constrain that will result in the whole set of constraints for this net being thrown away.

Lines 20 to 23 tell the router to branch from the resource labelled `some_io_datain` and to use *exactly* two previously unused routing resources to get to destination block `buf2`. The `some_io_datain` resource will be whatever routing resource was used by the router to satisfy the routing constraint of line 6. Given the constraint on line 6, it will be an IO_DATAIN type routing resource at (x=1, y = 31, sublocation = 2), but the index is not known until the router is run.

Lines 25 to 30, which are the constraints for signal `Input3` going to block `nor3`, tell the router to use only length-8 routing wires (besides IO_DATAIN and LOCAL_INTERCONNECT, which are required to get out of an IO cell and to get into a logic cell, respectively). This is achieved by the `zero_or_more` statement combined with `||`. `zero_or_more` tells the router to use as many resources following the keyword as needed.

Using `zero_or_more, *;` as on line 32, allows the router to use whatever it wants to route the connection. Does this mean that we could have just erased constraint for net `Input2` altogether? Almost. This constraint has an effect on the order in which nets are routed. The router routes nets in the following order: first all the nets listed in RCF are routed, in the order they appear, then unconstrained nets are routed. The same rule applies to connections of one net. Giving `Input2` a constraint before `Input4` constraint ensures that `Input2` will be routed first. If both nets contain wild-carded assignments, the net listed first will get its “choice” of routing resources.
You should run the design with this RCF and verify that the routing constraints are obeyed. For easier checking, you can use the procedure described in Section 3 to make Quartus generate an output RCF file. Make sure you save the *elaborate_rcf.rcf* file to a different name first so it is not overwritten.

If you check the routing carefully, you will notice that the constraint for signal `Input4` was not obeyed. To see if some routing constraint was not obeyed without checking the RCF output from Quartus manually, you can look at the messages generated by the Fitter. If you are using the GUI, look for messages starting with the words “Can’t route signal….” If you are working from the command line, search for the same string in the Fitter’s report file, *elaborate_rcf.fit.rpt*. Each routing constraint that could not be understood, was inconsistent with the circuit netlist, or could not be obeyed by the router will generate a message (up to 100 messages). Look in Quartus help for an explanation of the messages you get, their possible causes, and ways to resolve them. In the GUI, you can do this by right clicking on the message and choosing Help.

In this particular case, the router will point to line 37 of the RCF as the problem. This means that the router could not find a way to get from the routing resource on this line to the routing resource on the next line, a destination. This is because long routing wires, C16’s and R24’s, cannot drive logic directly. A signal needs to get onto an R4 or a C4, and then a LOCAL_INTERCONNECT before entering a logic cell.

4.6 Interfacing Various Types of Routers to Quartus

The first step in constructing an RCF file as the output of a separate tool is to use Quartus to generate a vqm file describing the circuit netlist, as described in [2]. Your tool will have to read in this vqm file so that it understands the circuit netlist, and knows (i) the name of signals, (ii) the names of function blocks and (iii) the names of the port(s) to which each signal connects on each function block. As described in Section 4.5, it is possible to write out a legal RCF file without using (iii), but there are cases in which such an RCF is ambiguous, so it is best to parse and use the names of the ports to which signals connect in RCF generation.

The second step in interfacing a router to Quartus is for your router to read the location assignments from the <project>.qsf file after a back-annotation of placement (described in Section 3), so your router knows the circuit placement. Alternatively, if you are using a placement program other than Quartus, you must read the placement generated by that tool into your router, and also input that placement to Quartus by writing appropriate location assignments to the <project.qsf> file.

The last step in interfacing a router to Quartus is to write a <project.rcf> file out from your router, and re-run the Quartus fitter with this new routing constraints file. It is valid to constrain the routing of only some nets – any other nets will be automatically routed by Quartus. This lets you focus on routing the common types of nets (such as those between logic cells and IOs) and avoid having to determine the rules for legally routing other types of function blocks like RAM and DSP blocks if you choose.

The exact form of the output RCF file will depend on the type of router you are writing. Some guidelines for common types of routing are given below.

- ### Global Routers

A global routing can be input to Quartus as an .rcf file where only the channels to be used to complete the router are specified. This is done by using wildcarding, choice and the *zero_or_more* keyword to allow Quartus to choose the type and number of wires in each channel. For example:

```
signal_name = Input3 {
        IO_DATAIN:*;
        zero_or_more, R4:X*Y30* || R8:X*Y30* || R24:X*Y30*;         #(1)
        zero_or_more, C4:X7* || C8:X7* || C16:X7*;                  #(2)
        LOCAL_INTERCONNECT:*;
        dest = ( nor3, DATAA );
}
```

The RCF fragment above indicates we desire a route that uses as many horizontal wires in the Y = 30 channel as necessary, and then as many vertical wires in the X = 7 channel as necessary. Any set of wires that matches these constraints is acceptable.

Note that lines (1) and (2) can be re-written in more compact but slightly more general form as follows:

```
        zero_or_more, *:X*Y30*;             #(1’)
        zero_or_more, *:X7*;                #(2’)
```

The only difference between (1) and (1’) is that other routing resources, besides R4, R8, and R24, will be allowed for that step.

- ### Wire Type Routers

A wire type router produces somewhat more detailed output than a global router. It not only says what routing channels to use, but also what type of wire to use, and potentially, how many of each type of wire. It does not specify exactly which wires should be used to implement route, however, it simply specifies their types. For example, the RCF fragment below specifies that this net should be routed using one R8 wire, followed by one C8 wire. Quartus is free to choose which R8 and which C8 wire to use.

```
signal_name = Input3 {
        IO_DATAIN:*;
        R8:*;
        C8:*;
        LOCAL_INTERCONNECT:*;
        dest = ( nor3, DATAA );
}
```

The example below specifies not only what types of wires to use, but also which channel each should be in.

```
signal_name = Input3 {
        IO_DATAIN:*;
        R8:X*Y30*;
        C8:X7*;
        LOCAL_INTERCONNECT:*;
        dest = ( nor3, DATAA );
}
```

The example below is even simpler as it allows Quartus to choose how to get in and out of the function blocks, and simply specifies the types and channels of the wires to be used between the function blocks.

```
signal_name = Input3 {
        zero_or_more, *;   # Quartus can start the route as it pleases
        R8:X*_Y30*;        # Must use one R8 wire in the Y=30 channel.
        C8:X7*;            # Must use one C8 wire in the X=7 channel.
        zero_or_more, *;   # Quartus can end the route as it pleases.
        dest = ( nor3, DATAA );
}
```

In order to generate legal wire-type routes, it is necessary to understand which routing resources connect to each other in Stratix devices. Table 1 provides this information for Stratix devices. See *MultiTrack Interconnect* section of Volume 1, Chapter 2 of *The Stratix Device Handbook* (electronic version available from www.altera.com) for a detailed description of the Stratix routing architecture [6].

Table 1: Connectivity between types of routing resources in the Stratix device family.

Resource Type	Keywords	Can Drive	Can be Driven by
Function block output buffers	IO_DATAIN, LE_BUFFER, DSP_BUFFER, M512_BUFFER, M4K_BUFFER, MRAM_BUFFER	R4, C4, R8, C8, IO_BUS_BUFFER	Function blocks
Length 4 wires	R4, C4	R4, C4, R24, C16, IO_BUS_BUFFER, LOCAL_INTERCONNECT	R4, C4, R24, C16, Function block output buffers
Length 8 wires	R8, C8	R8, C8, LOCAL_INTERCONNECT	R8, C8, Function block output buffers
Length 24/16 wires	R24, C16	R4, C4, R24, C16	R4, R24, C16
IO_BUS wires and buffers	IO_BUS, IO_BUS_BUFFER	IO_BUS_BUFFER, IO_BUS, LOCAL_INTERCONNECT, R4, C4	IO_DATAIN, IO_BUS_BUFFER, IO_BUS, R4, C4
Local interconnects	LOCAL_INTERCONNECT	Function block inputs	R4, C4, R8, C8, IO_BUS

Function block inputs	Most are not specified. The logic cell inputs DATA, DATAB, DATAC and DATAD can be specified after the route_port keyword.	Function block internals	LOCAL_INTERCONNECT

- **Detailed Routers**

To write a detailed router – a detailed router determines exactly which wires should be used to route each net – it is necessary to have the entire graph describing all the wires and switches that exist in a Stratix FPGA so that your router knows what are possible connections and what are not. This information is contained in a file called a Master Connectivity File (MCF). At this time, Altera has chosen not to generally release this file to university researchers. However, if you wish to write a detailed router for Stratix or Cyclone devices, contact quip@altera.com and we may be able to give you access to the MCF file under a non-disclosure agreement.

The output of a detailed router is fed into Quartus as an RCF file in which precise wires are specified for each step of the routing. An example would be:

```
signal_name = Input3 {
        IO_DATAIN:X0Y30S2I1;
        R8:X0Y30S0I22;
        C8:X8Y30S0I43;
        LOCAL_INTERCONNECT:X7Y33S0I18;
        dest = ( nor3, DATAA );
}
```

4.7 Errors and Illegal Routes

When the Quartus router does not understand or cannot obey a routing constraint, it will print a message to this effect. Quarts will then ignore the problematic signal constraints and route the signal whatever way seems best to it. Be sure to check for any messages related routing constraints.

There are two types of errors that you can have in RCF. Both will issue an information message warning you of the error.

Parse time errors: This type of error includes problems such as violation of the RCF grammar, and the routing constraints being inconsistent with the circuit netlist. The first error will cause immediate parser termination and all constrained routing being turned off for the compile. The second error will occur if a routing constraint is specified for a non-existent connection. The routing constraint for this connection will be disregarded.

Route time errors: These errors will be detected during routing of the connections. They occur when the router cannot satisfy a routing constraint, since it requires connectivity that does not electrically exist in the device. When this happens, the offending constraint will be removed, and the router will try to route the connection again, using whatever resources it needs to route the connection.

4.8 Comparing Routing Result Quality to Quartus

You may wish to compare the result quality of your router to that of the built-in Quartus router. To do this, simply run quartus_fit twice – once with the RCF file generated by your router, and once without, and compare the results. How best to compare depends on if you are writing a routability-driven or a timing-driven router.

- **Routability-driven router comparison:** Good metrics of the quality of a routability-driven router are the percentage of benchmark circuits it can successfully route, and the amount of routing wirelength it requires to achieve a successful route for a circuit. To count the number of wires used by Quartus, you can simply run wire counting scripts on the RCF generated by Quartus when you back-annotate the routing. An example script, *wire_usage_from_rcf.*pl, is located in *constrained_routing* subfolder. This script counts the total number of wires used in an RCF. Instead of simply counting how many wires are used, you may want to sum the total length of all the wires used to route a circuit.

  For a fair comparison of the Quartus router to a routability-driven router, you should turn off the timing-optimization features of the Quartus router, since they normally cost additional wire usage. This can be done from the GUI by selecting *Assignments***|***Settings***|***Fitting (under Compiler Settings)*. Uncheck *Optimize Timing* and *Optimize IO Cell Register Placement for Timing*. If you prefer to use the command line, open the file *<project_name>.qsf* in a text editor and set values of `OPTIMIZE_TIMING` and `OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING` to `OFF`.

- **Timing-driven router comparison:** To compare the circuit speed achieved by your router to that of the Quartus router, first you should ensure that you have set timing constraints on all the timing paths you will measure. If you wish to compare the speed of a certain clock, for example, set a high frequency (Fmax) requirement on that clock so Quartus will work hard to optimize it. In general, you should ensure that the constraint you set is at the limit or beyond the limit of what Quartus can achieve to ensure Quartus is working as hard as possible to get the best timing on that constraint. If you wish to compare IO timing, you should set Tsu, Tco and or Tpd timing constraints. The Quartus on-line help and on-line application notes [5] explain how to set timing constraints.

  Once the circuit routing is complete, you can look in the *<project>.tan.rpt* file to see the timing report, or look in the summary file *<project_name>.tao_summary* file for a brief summary of timing. If you prefer using the GUI, you can use the *Processing***|***Compilation Report* command and click on the timing report section to see the timing results of your circuit.

4.9 Exercises

1. Stratix chips have long routing wires that run around the perimeter of the chip, called the IO Bus. IO Bus wires are buffered, meaning that an IO Bus Buffer must be used right after using an IO Bus wire (even if you connecting to another IO Bus wire). The length of these wires varies but it is known that you can get on and off an IO Bus wire at any location adjacent to an IO block (not every location on the chip has an IO block, as you can check in Last Compilation Floorplan). For this exercise, you are to modify the routing constraint for the `Input4` signal in *elaborate_rcf* design. Create a routing constraint for this signal so that the router uses at least one IO Bus wire to get to destination `buf5`. IO Bus wires and buffers are called `IO_BUS` and `IO_BUS_BUFFER`, respectively, in RCF. Be sure to check that your routing constraint was obeyed.

2. The longest routing wires available on Stratix chips are C16 and R24 wires. To get on these wires, a signal must use an R4 wire, and to get off these wires, signals must use R4 or C4 wires. Constrain signal `buf3` to drive `nor3` using one LE_BUFFER (to get out of the `buf3` logic cell), two wires of length 4, as many R24 or C16 as needed, and one local interconnect (to get into the `nor3` logic cell).

Answers to the exercises are in Appendix E.

Referenced Documents

1. "QSF Assignment Descriptions Document."
2. "Quartus II University Interface Program (QUIP) Tutorial."
3. "VQM Extractor Functional Description.
4. "Stratix EDA and Academic Developer Functional Description."
5. Application Note 123: Using Timing Analysis in the Quartus II software. http://www.altera.com/literature/an/an123.pdf .
6. *The Stratix Device Handbook*, Volume 1, Chapter 2, M*ultiTrack Interconnect* section. http://www.altera.com/literature/hb/stx/stratix_section_1_vol_1.pdf .

Appendix A: Control Variables

There are two QSF settings to control constrained routing. `ROUTING_BACK_ANNOTATION_MODE` turns constrained routing on and off. Possible values are `OFF` and `NORMAL`. `OFF` turns off constrained routing. `NORMAL` turns it on.

`ROUTING_BACK_ANNOTATION_FILE` allows you to specify which file should be used as the routing constraints file. The default behaviour (if this variable is not set) is to look for <project_name>.rcf in the current directory. If no such file is found, *routing_constraints.rcf* is searched for next. If that file is also not found, constrained routing complains and turns itself off. Quartus always write routing constraints to <project_name>.rcf.

When using quartus_cdb to generate an RCF from command line, you can give it an extra argument: `write_rcf_for_vqm`. It is used to generate an RCF that will be compatible with VQM back-annotation – use this option if you plan on using the vqm file written out by quartus_cdb after quartus_fit has been run as your source netlist to re-run quartus_fit. You should not need to do this in order to write your own router. However, if you ever have to use VQM back-annotation, consult Quartus Help for more details.

There are also three quartus.ini settings that enable automatic testing of designs using constrained routing. Setting `VPR_STOP_ROUTING_IF_ROUTING_CONSTRAINTS_ARE_INCORRECT` to `ON` will cause Fitter to abandon routing and issue a no-fit as soon as unsatisfiable constraints are encountered. `VPR_CR_DIE_IF_PARSER_MESSAGES` will cause Fitter to thrown an internal error and die if there are any messages generated by RCF parser.

These settings can be added to your project in two ways: through quartus.ini or QSF. To use quartus.ini, simply add the following lines anywhere in the file:

```
VPR_STOP_ROUTING_IF_ROUTING_CONSTRAINTS_ARE_INCORRECT = ON
VPR_CR_DIE_IF_PARSER_MESSAGES = ON
```

This method is useful if you have a separate quartus.ini for you project, which is usually not the case. To set these variables through QSF, do the following:

1. Open <project_name>.qsf in any text editor.
2. On a seprate line add

```
set_global_assignment -name INI_VARS = "VPR_STOP_ROUTING_IF_ROUTING_CONSTRAINTS_
ARE_INCORRECT = ON; VPR_CR_DIE_IF_PARSER_MESSAGES = ON;";
```

3. Save and close the file.

See [1] for more information about QSF format.

When the quartus.ini settings take effect, Fitter will abandon compilation as soon as first problem with RCF is encountered. Therefore, use them only when needed and do not keep them for longer than necessary.

The last quartus.ini setting controls naming of routing resources in RCF file. Quartus II 4.0 was modified to generate user-friendly names in RCF file. If you prefer to see shorter Quartus II 3.0 names in Quartus-generated RCF, the this setting to quartus.ini file:

```
RCF_DONT_USE_USER_NAMES = ON
```

You can also add the setting to QSF file using procedure described for the previous two quartus.ini settings. Note that Quartus II 4.0 will understand both types of names without the setting above – the setting controls only name generation.

Appendix B: Glossary

Connection (or netlist connection) – A single source – sink pair that identifies one connection on a net, or signal. A net with fanout N consists of N distinct connections.

Destination – A sink terminal on a signal. In RCF files, this is identified by a pair of strings. The first string is the name of the driven block, while the second is the name of port on the driven block. The port name is needed to distinguish connections in cases when one signal is driving a block more than once.

Function Block – The various blocks in a Stratix device that perform functions, or processing. Examples are IO cells, logic cells, M512 RAM blocks, M4K RAM blocks, M-RAM blocks and DSP blocks.

Global Signal – Signal on special routing network designed to minimize signal skew. Clock and clear signals are usually made global.

LAB – Logic Array Block. Block of ten logic cells.

Logic Element, aka **Logic Cell** – A basic unit of logic on Stratix chip. It consists of 4-input binary function, a register, and some specialized arithmetic and other circuitry.

MCF – Master Connectivity File. Contains connectivity and delay information for all routing resources in the chip.

RCF – Routing Constraints File.

Signal – a net, or set of terminals that must be electrically connected. Consists of a source and one or more sinks, or destinations.

Appendix C: RCF Syntax

Below is a complete grammar of RCF in BNF. Names in angular brackets are rule names. `::=' shows how a rule should be expanded. `||' gives options in rule expansion. Strings in double quotes match themselves. ∅ denotes an empty string. RCF parser is case-sensitive.

```
<section_list>        ::=    ∅ ||
                             <section_list> <section>

<section>             ::=    "section" "ram_sublocations" <signal_open> <ram_subloc_list> "}" ||
                             "section" "global_data" "{" <global_data_list> "}" ||
                             "section" "extra_information" "{" <skipped_section> "}" ||
                             <signal_start> <steps> <signal_end>

<global_data_list>    ::=    ∅ ||
                             <global_data_list> "rcf_written_by" "=" <quote> <quoted_string> <quote>
<end_step> ||
                             <global_data_list> "device" "=" <resource_name> <end_step>

<ram_subloc_list>     ::=    ∅ ||
                             <ram_subloc_list> <ram_subloc>

<ram_subloc>          ::=    "(" <name> "," <number> <second_number> ")" <end_step>

<second_number>       ::=    ∅ ||
                             "," <number>

<signal_start>        ::=    "signal_name" "=" <name> "{"

<signal_end>          ::=    "}"

<steps>               ::=    ∅ ||
                             <steps> <step>

<step>                ::=    <branch_point> <end_step> ||
                             "branch_anywhere" <end_step> ||
                             <labeled_choice> <end_step> ||
                             <zero_or_more_step> <end_step> ||
                             <choices> <end_step> ||
                             <destination> <end_step>

<branch_point>        ::=    "branch_point" "=" <name>

<labeled_choice>      ::=    "label" "=" <name> "," <choices>

<zero_or_more_step>   ::=    "zero_or_more" "," <choices>

<choices>             ::=    <choice> ||
                             <choices> "||" <choice>

<choice>              ::=    <wildcard> ||
                             <resource> ":" <options>

<resource>            ::=    <wildcard> ||
                             <resource_name>

<options>             ::=    ∅ ||
                             <options> "X" <val> ||
                             <options> "Y" <val> ||
                             <options> "S" <val> ||
                             <options> "I" <val> ||
                             <options> <wildcard>

<val>                 ::=    <wildcard> ||
                             <number> "-" <number> ||
                             <number>
```

```
<destination>         ::=    "dest” "=“ "(" <name> <dest_port> ")” <route_port>

<dest_port>           ::=    ∅ ||
                             ",” <resource_name> <index>

<index>               ::=    ∅ ||
                             “[“ <number> “]“

<route_port>          ::=    ∅ ||
                             ",” <route_port> “=“ <resource_name> <index>

<skipped_section>     ::=    [^}]*
<quoted_string>       ::=    [^"]*
<resource_name>       ::=    [a-zA-Z][a-zA-Z0-9_]*
<name>                ::=    [^ \t,;{#\n()]+
<number>              ::=    [0-9]+
<wildcard>            ::=    [\*]
<end_step>            ::=    [;]
<comment>             ::=    [#]
<quote>               ::=    ["]
```

Comments

- Note that most parts of RCF are optional.
- `<extra_information>` section can contain arbitrary data. It can be used as one big comment.
- `<ram_subloc_list>` contains information necessary to reproduce RAM clustering. Numbers stored in this section have no physical meaning and should not be modified.
- The router prefers to use the minimum number of resource(s) with the `<zero_or_more>` modifier. If possible, it will generally complete a route using zero of these resources.
- `<dest_port>` can include bus index when the port to which they are connected is a bus port (i.e. a port that is more than 1 bit wide). For example, signals going to the address ports of RAM blocks will most likely have a bus index specification, e.g.: `dest=(ram_block, PORTAADDR[2])`.
- `<route_port>` represents the route_port specification. This field is only used for data inputs to logic cells and will be ignored for all other block or port types. The reason for this field’s existence is that the dataa to datad ports of logic cells are often logically equivalent, and Quartus will “rotate” the inputs used to improve the speed or routability of the design. `<route_port>` can be used to force a certain data input to be used to a logic cell.
- `<name>` defines strings that can be used for signal names and labels.

Appendix D: Advanced Comments

There are a few advanced issues we should mention.

1. When a connection cannot be routed, routing constraints for this connection are discarded. If a subsequent connection tries to branch from a label defined in the constraints of an unroutable connection, the branching connection also becomes unroutable. Consider the following routing constraints as an example:

```
signal_name = Input1 {
        label = some_io_datain, IO_DATAIN:X1Y31S2I2;
        zero_or_more, *;
        dest = ( buf1, DATAA );

        branch_point = some_io_datain;
        zero_or_more, *;
        dest = ( buf3, DATAB );
}
```

If `Input1` cannot be routed to `buf1`, the `IO_DATAIN` node will not be used. Hence, the branch point that starts routing for connection `buf3` will be illegal.

2. As previously stated, a `` `*' `` in the routing constraints means "any one unused routing resource node." The word "unused" is important here. It means that you cannot use `` `*' `` as a substitute for a branch point because the branch point node was already used (at the labelled location to complete the routing of a previous connection).

3. Quartus routes nets on connection-by-connection basis. Resources are picked to match wild cards to make the *current* connection routable. If you wild card a label, other connections branching from that resource will not affect the choice of resource for the branch point. This might lead to cases where the router cannot obey the constraints even though a valid routing exists.

Consider the following constraint as an example:

```
signal_name = input {
        label = my_label, IO_DATAIN:*;
        zero_or_more, *;
        dest = ( block1, DATAA );

        branch_point = my_label;
        LOCAL_INTERCONNECT:*;
        Dest = ( block2, DATAB);
}
```

`IO_DATAIN` will be picked to create a valid route to `block1`. However, the chosen `IO_DATAIN` might not drive any local interconnects near `block2`. Furthermore, there might exist another `IO_DATAIN` that can be used to satisfy both routing constraints.

4. `branch_anywhere` instructs the router to use any number of previously used routing resources at this step. This allows constraining parts of routing near destinations without having to specify precise net topology. The construct is primarily used for internal testing and will probably not be very useful in writing of custom routers.

Note that `branch_anywhere` is different from `zero_or_more, *`. The former tells the router to use any number of previously *used* resources, whereas the latter instructs usage of any number of previously *unused* resources. As an example, the code below constraints `clk` signal to enter two of its destinations using `LAB_CLK`s with index 3:

```
signal_name = clk {
        zero_or_more, *;        #(1)
        LAB_CLK:*I3;            #(2)
        Dest = (block1);

        branch_anywhere;        #(3)
        zero_or_more, *;        #(4)
        LAB_CLK:*I3;            #(5)
        dest = (block2);
}
```

Because `block1` is the first destination of `clk` signal, router does not have any used routing resources for the net. Therefore, we use `zero_or_more, *` on line (1) to specify an arbitrary path to the block. Line (3) says that path to `block2` can branch off at any point of the path to `block1`. Because we do not know which resource it is, we cannot be sure we can get to `LAB_CLK` from it. Therefore, we must allow the router to use as many routing resources as needed to get to the `LAB_CLK` on line (5).

Appendix E: Answers to Exercises

1. Replace lines 36 and 37 with the following text:

```
zero_or_more, *;
IO_BUS:*;
IO_BUS_BUFFER:*;
zero_or_more, *;
```

First `zero_or_more,*` is used to get to IO Bus. Then we use one IO_BUS wire with IO Bus Buffer. Afterwards, another `zero_or_more,*` is used to get to the destination.

Note that Input4 is an IO block. This means that you only need to go through IO_DATAIN to get to IO Bus wire. Therefore, the following is also a valid answer:

```
IO_DATAIN: *;
IO_BUS:*;
IO_BUS_BUFFER:*;
zero_or_more, *;
```

2.

```
signal_name = buf3 {
       LE_BUFFER:*;
       R4:*;
       zero_or_more, R24:* || C16:*;
       R4:* || C4:*;
       LOCAL_INTERCONNECT:*;
       dest = ( nor3 );
}
```