Quartus II University Interface Program (QUIP) Tutorial

Altera Corporation

Version 3.1

April 6, 2005

Table of Contents:

1. Overview

The purpose of this document is to describe how to connect other CAD flows to the Altera® Quartus® II CAD design flow and Altera FPGAs. It is given in a tutorial format. We believe that by providing the ability to connect to industrial-class tools in easy ways, that the Quartus II design software will enable research in two ways:

1. By providing a complete infrastructure, researchers need only work on the part of a CAD flow they are interested in, leaving the remaining details to the existing flow. For example, those interested in technology mapping into lookup tables will be able to use the Quartus front-end language elaborators on HDL benchmarks, and also use the Quartus II software as a back-end to judge the ultimate post-routing quality of their technology mapper.

2. By providing access to a CAD tool suite that targets real devices with industrial strength delay annotation, timing-analysis, routing etc., the veracity of a researcher's results will be better. The data reported by the Quartus II design software is very accurate; it is far superior to approximate metrics for area and delay.

As another example, the Quartus II software could be used as an HDL front end that provides a technology-mapped netlist for an academic tool to perform placement. The Quartus II CAD software can then be used to do the routing and timing analysis of the placement.

This document gives an overview of the Quartus CAD flow without providing all the detail necessary to use every aspect of the CAD flow and Altera FPGA devices. Associated documents will provide the complete details of all aspects of the Quartus flow and several of the more recent FPGA devices that it targets.

2. What You Need Before Starting This Tutorial

For this tutorial to make sense, you will need:

1. To have installed Quartus II Version 5.0 on your computer.

2. To be familiar with the basic operation of the Quartus II 5.0 software. You can do this by doing the basic user's tutorial, "Tutorial – Using Quartus II CAD Software [1]." It is also available in this QUIP™ distribution.

3. Tutorial

In the Quartus II 5.0 software, it is possible to invoke various phases of the Quartus flow from a command-line interface. This version of the tutorial works exclusively with the command-line interface.

3.1 Command Windows, Shells and Conventions

You can run the command-line executables of the Quartus software under Windows using a DOS/NT/XP command window, or some other user shell, or a shell under Unix or Linux. Note that the exact shell you use may not conform to the command syntax we use below, which assumes DOS/Windows Command Prompt commands ("**copy**" for copy and "**chdir**" for change directory, and back slashes "\" to separate directory hierarchy). You must ensure that the Quartus **bin** directory is in the **path** for your command window or shell. You can find the Quartus bin directory as follows: If your main install directory is **quartus**, then the bin directory will be **quartus\\bin**. It will contain the executables quartus_map.exe, quartus_fit.exe, etc.

We will use the following conventions for this tutorial:

1. Explicit commands that you should type will be preceded by an angle bracket ("**>**") and typed in bold, for example

   > quartus_map file.v

   All such commands are submitted using the Enter key at the end of the line.

2. The output from the Quartus II design software will be shown in the courier font:

   ```
   This is the style of Quartus output.
   ```

3.2 Tutorial Circuit

We begin with a circuit described in Verilog code, and show how to synthesize, place, route and analyze the timing of the design. After each of these steps, we will show the textual output of the step, and modify it manually before passing it on to the next step.

In the directory that contains this tutorial, there is a subdirectory named **verilog_files**. In that directory there is a file named **fnf.v** which is a Verilog file that specifies a very simple circuit: a single NAND gate with registered inputs and output, fed from I/O pads of the chip. The clock is also fed from an I/O pad. The circuit is shown in schematic form in Figure 1.

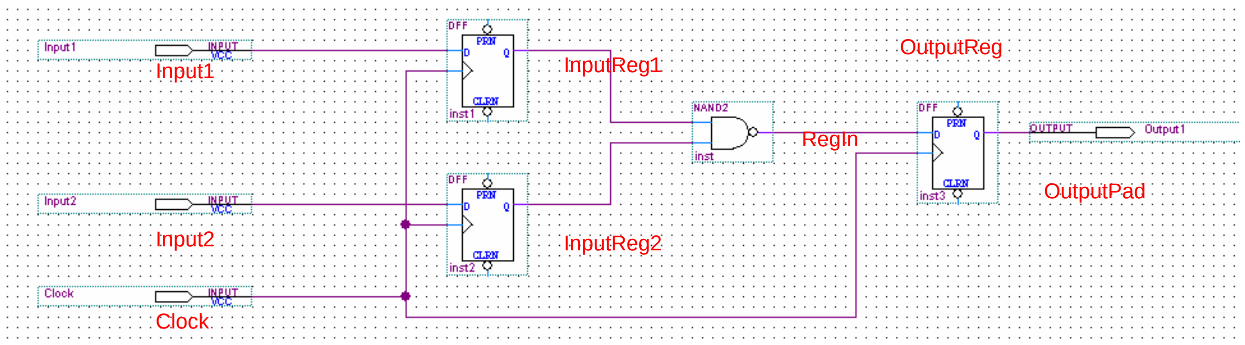


Figure 1 - NAND Gate with Registered I/Os Circuit used Through Tutorial

The Verilog code for the circuit is

```
// Simple registered NAND gate design for use in QUIP tutorial.
// "fnf" refers to Flip-flop -> NAND -> Flip-flop.

       module fnf (Input1, Input2, Clock, OutputPad);

// Declare two inputs and one output.
// Registers are clocked by signal Clock.
       input    Input1, Input2, Clock;
       output   OutputPad;
       reg       OutputReg;
       reg       InputReg1, InputReg2;
       wire     RegIn;

// Connect Registers
       always@(posedge Clock)
       begin
              InputReg1 <= Input1;
              InputReg2 <= Input2;
              OutputReg <= RegIn;
       end

//  Create NAND Function
       assign  RegIn = ~(InputReg1 & InputReg2);

//  Hook Output Register to the Output Pad

       assign  OutputPad = OutputReg;

       endmodule
```

We will use this circuit for the entire tutorial, beginning with synthesis.

3.3 Synthesis

The first step in the Quartus CAD flow is synthesis, which takes a hardware description language file as input and produces a mapped netlist output. We will use the Verilog file fnf.v as the input and the Quartus software will produce the netlist in a file format known as **VQM** (which stands for Verilog Quartus Mapping) as output. We give a brief description of the **VQM** format below.

Step 1: Create a directory named fnf, and copy the fnf.v file into it. In the commands below, replace {Tutorial Directory} with the name of the directory containing the

verilog_files from the QUIP distribution. (Note that we are using DOS/Windows commands here; if you are using Unix or some other type of shell, you will have to replace some of them with appropriate commands.)

```
> mkdir fnf

> chdir fnf

> copy  <Tutorial Directory>\verilog_files\fnf.v .
```

Step 2: Run the Quartus synthesis tool on this file. We will tell the mapping tool to target the Stratix® EP1S10F484C5 FPGA device. This is the smallest Stratix device, which has 10,000 logic cells and 484 pins on the package. Its speed grade is –5. Speed grade -5 is the fastest of three available speed grades, which are -5, -6 and -7.

```
> quartus_map fnf --part=EP1S10F484C5
```

(Note that the double dash, "--" is required, rather than a single dash. Also, in order to get the part name exactly correct, we recommend cutting and pasting from this document.) Upon running this command you will see several messages giving the Quartus software version and copyrights contained therein. At the end you should also see a message that looks like the following:

```
Info: Implemented 7 device resources after synthesis - the final
resource count might be different
    Info: Implemented 3 input pins
    Info: Implemented 1 output pins
    Info: Implemented 3 logic cells
Info: Quartus II Analysis & Synthesis was successful. 0 errors, 0
warnings
    Info: Processing ended: Fri Mar 18 10:58:04 2005
    Info: Elapsed time: 00:00:03
```

If you don't see these messages, check to make sure that the Quartus bin directory is in the search path of the command window or shell that your are using.

The quartus_map command performs HDL language elaboration, technology-independent logic optimization, technology mapping into lookup tables and then packs flip-flops and lookup tables together to form *logic cells*. (The logic cell for the Stratix family FPGA contains a 4-input lookup table, a flip-flop, and some support for arithmetic carry logic). You can see from the messages above that quartus_map required a total of three input pins, one output pin and three logic cells to implement the fnf circuit, the circuit for which is illustrated in Figure 2. The three inputs include the two registered inputs as well as the clock. Three logic cells are required because one contains the NAND gate and the flip-flop following it (labeled OutputReg in the figure), and the other two contain the two flip-flops for the inputs of the NAND gate (labeled InputReg1 and InputReg2).

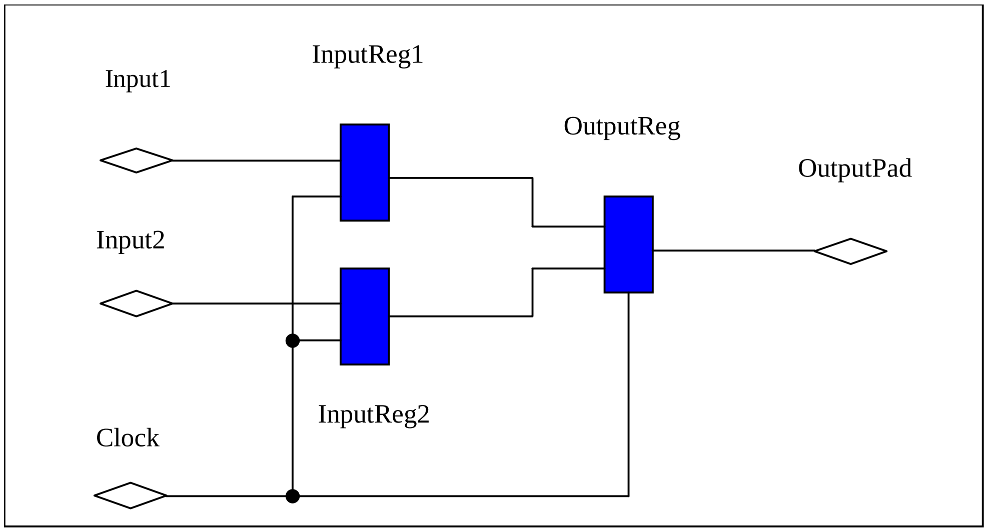


Figure 2 - Netlist of Logic Cells from Mapping

The quartus_map command creates a number of files and a directory in your fnf directory. We offer a brief description of each file here:

The file **fnf.qpf** is known as the quartus *project* file. If you wished to start the Quartus GUI on this project, you would simply double-click this file under Windows, or use the **File | Open Project** command from the GUI to open it. This file gives the Quartus II design software all of the information needed to know what has been done with the circuit so far.

The file **fnf.map.rpt** contains a report file from the Quartus synthesis tool, with more details on what device was targeted and how many on-chip resources were used.

The file **fnf.map.summary** contains a brief summary from the Quartus synthesis tool. It indicates the device that was targeted and the resource usage, but is not as detailed as fnf.map.rpt.

The file **fnf.map.eqn** contains the logic equations for the lookup tables that were created, as well as a description of the flip-flops and I/Os that were used.

The file **fnf.flow.rpt** contains a summary of the entire compilation thus far. Currently, we have only run the Quartus synthesis tool, so there will only be information about this step.

The file **fnf.abo** contains information about the individual atoms used in the design.

The file **fnf.qsf** contains settings for the project related to the Quartus compiler. For example, the line

```
set_global_assignment -name DEVICE EP1S10F484C5
```

shows that you selected the EP1S10F484C5 FPGA device.

Finally, the **db** directory contains the Quartus internal database files for describing the circuit, compiler outputs and user constraints.

Note that the **quartus_map** command did not actually create a readable file with the technology-mapped netlist, although this netlist was created in the internal database. The next step will create the readable netlist file, known as a **VQM** file.

Step 3: Create the output netlist in the VQM format.

The **quartus_map** command performed all of the synthesis described above and stored the result in the internal (unreadable) Quartus database files. To create a readable and usable file for other tools, or for reading VQM files back into the Quartus software, you need to use the **quartus_cdb** command. You can modify the resulting netlist to change the functionality of the circuit.

To create the netlist run the following command:

```
> quartus_cdb fnf --vqm=fnf.vqm
```

This command will create the file **fnf.vqm** that contains the netlist corresponding to the fnf.v Verilog description. The **VQM** format, which stands for "Verilog Quartus Mapping" is a netlist of logic cells, inputs and outputs (and other basic elements in Altera devices such as memory, multipliers, PLLs, etc.) in a Verilog-style format. (Note that Verilog can be used to describe circuits at the structural or register-transfer level (RTL) as in the original fnf.v file, or it can provide a rudimentary netlist format, as is the case with the VQM format.)

At first glance, the **fnf.vqm** file may seem somewhat long and complex but the basic structure is simple. If you wish to create a tool that generates the VQM format, it would be fairly easy to do so. You use the VQM format as input to the Quartus II design software if you wish to do technology mapping into lookup tables, but want Quartus' modules to do the subsequent placement, routing and timing analysis.

The VQM format is a restricted subset of the Verilog format, it instantiates primitives that the Quartus software will understand. To get a brief overview of the VQM format, open the fnf.vqm file in the fnf directory using a text viewer (such as WordPad or vi) or in the Quartus II GUI. This fnf.vqm file is reproduced in Figure 3. The initial module and signal declarations are quite similar to the original Verilog file fnf.v. The rest of the file has a simple format. There is a declaration of an element (either an input, an output, or a logic cell) and then a series of "defparam" statements that provide the details of the information needed to precisely specify the function of the element (i.e. Define the Parameters of the element, shortened to *defparam).*

For example, observe the lines numbered 62-64 in Figure 3:

```
stratix_io \Input2~I (
        .combout(\Input2~combout ),
        .padio(Input2));
```

These lines instantiate an input pad for Input2, which is labeled as **\Input2~I**. The backslash character is inserted by the software to ensure that any incoming name remains a legal Verilog name. The text in brackets on lines 62-64 indicates what signals are connected to the inputs and outputs of the IO cell. For example, the combout (\Input2~combout) text indicates that a signal called **\Input2~combout** is connected to the combinational output port (called .combout) of the IO cell.

The *defparam* statements that follow this line set various parameters relating to an I/O pad, indicating that the I/O pad is an input (the parameter **operation_mode** on line 65) and that there is no asynchronous reset (the parameter **input_async_reset** on line 70), for example.

As another example, lines 122 - 127 instantiate the logic cell that contains both the NAND gate and the flip-flop that registers its output:

```
stratix_lcell \OutputReg~I (
        .clk(\Clock~combout ),
        .dataa(InputReg2),
        .datab(InputReg1),
        .aclr(gnd),
        .regout(OutputReg));
```

This creates an instantiation of a Stratix FPGA logic cell named **\OutputReg~ I**. This cell has a 4-input lookup table whose inputs are named **dataa**, **datab**, **datac**, and **datad**. The cell also has a flip-flop that requires a clock (named **clk**), an asynchronous clear (named **aclr**) and the Q output (named **regout**).

In this logic cell instantiation, you can see that the signal **\Clock~combout** forms the clock of the register, and that the signal **InputReg1** is connected to **datab** and **InputReg2** is connected to **dataa**. Since the 2-input NAND gate only requires two inputs, there is no need to give connections to **datac** and **datad**. The output of the flip-flop is connected to the signal **OutputReg**, and the asynchronous clear is connected permanently to ground.

The parameters defined following this instantiation (in the set of defparams) further specify the functionality of the logic cell. For example its **operation_mode** in line 128 is “normal”, which is one of two choices, the other of which is “arithmetic”. The logic function of the 4-input lookup table (LUT) in the cell is specified by setting the truth table of the LUT. This is called the “LUT mask” and is provided in line 132:

```
defparam \OutputReg~I .lut_mask = "7777";
```

The LUT mask is a four-digit hexadecimal number that represents the truth table of the LUT, organized as shown in Table 1.

datad	datac	datab	dataa	LUT Output
0	0	0	0	O_0
0	0	0	1	O_1
0	0	1	0	O_2
0	0	1	1	O_3
0	1	0	0	O_4
0	1	0	1	O_5
0	1	1	0	O_6
0	1	1	1	O_7
…	…	…	…	…
1	1	1	1	O_{15}

Table 1 - Format for the Creation of the LUT Mask of a Logic Cell

The hexadecimal number is formed from the following binary number: $\mathbf{O_{15}\ O_{14}\ \ldots\ O_2\ O_1\ O_0}$. The complete truth table generated from the LUT mask 7777 is given in Table 2.

datad	**datac**	**datab**	**dataa**	**LUT Output**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Table 2 - LUT Mask for a 2-input NAND function of **dataa** and **datab**

By inspection, you can see that this truth table forms a NAND function of the inputs **dataa** and **datab**, regardless of the values of **datac** and **datad**.

The other logic cells instantiated in the **VQM** file are the two needed for the registers on the inputs of the NAND gate logic cell, in lines 80 and 110. The other I/O cells instantiated are for the signal **Input2**, the **Clock** and the **OutputPad**.

The above gives an abbreviated description of the **VQM** format. For a more complete description, see [2] for the basic **VQM** grammar. Also see [3], [4], and [5] for descriptions of the **VQM** format for the logic cells, memory blocks, and Stratix multiplier-accumulator (MAC) block respectively.

```
1   // Copyright (C) 1991-2005 Altera Corporation
2   // Your use of Altera Corporation's design tools, logic functions
3   // and other software and tools, and its AMPP partner logic
4   // functions, and any output files any of the foregoing
5   // (including device programming or simulation files), and any
6   // associated documentation or information are expressly subject
7   // to the terms and conditions of the Altera Program License
8   // Subscription Agreement, Altera MegaCore Function License
9   // Agreement, or other applicable license agreement, including,
10  // without limitation, that your use is for the sole purpose of
11  // programming logic devices manufactured by Altera and sold by
12  // Altera or its authorized distributors.  Please refer to the
13  // applicable agreement for further details.
14
15  // VENDOR "Altera"
16  // PROGRAM "Quartus II"
17  // VERSION "Version 5.0 Internal Build 129 03/10/2005 TO Full Version"
18
19  // DATE "03/18/2005 14:31:07"
20  module     fnf (
21    Clock,
22    Input2,
23    Input1,
24    OutputPad);
25  input      Clock;
26  input      Input2;
27  input      Input1;
```

```
output     OutputPad;
wire \Clock~combout ;
wire \Input2~combout ;
wire InputReg2;
wire \Input1~combout ;
wire InputReg1;
wire OutputReg;


wire gnd;
wire vcc;

assign gnd = 1'b0;
assign vcc = 1'b1;


stratix_io \Clock~I (
  .combout(\Clock~combout ),
  .padio(Clock));
defparam \Clock~I .operation_mode = "input";
defparam \Clock~I .ddio_mode = "none";
defparam \Clock~I .input_register_mode = "none";
defparam \Clock~I .output_register_mode = "none";
defparam \Clock~I .oe_register_mode = "none";
defparam \Clock~I .input_async_reset = "none";
defparam \Clock~I .output_async_reset = "none";
defparam \Clock~I .oe_async_reset = "none";
defparam \Clock~I .input_sync_reset = "none";
defparam \Clock~I .output_sync_reset = "none";
defparam \Clock~I .oe_sync_reset = "none";
defparam \Clock~I .input_power_up = "low";
defparam \Clock~I .output_power_up = "low";
defparam \Clock~I .oe_power_up = "low";

stratix_io \Input2~I (
  .combout(\Input2~combout ),
  .padio(Input2));
defparam \Input2~I .operation_mode = "input";
defparam \Input2~I .ddio_mode = "none";
defparam \Input2~I .input_register_mode = "none";
defparam \Input2~I .output_register_mode = "none";
defparam \Input2~I .oe_register_mode = "none";
defparam \Input2~I .input_async_reset = "none";
defparam \Input2~I .output_async_reset = "none";
defparam \Input2~I .oe_async_reset = "none";
defparam \Input2~I .input_sync_reset = "none";
defparam \Input2~I .output_sync_reset = "none";
defparam \Input2~I .oe_sync_reset = "none";
defparam \Input2~I .input_power_up = "low";
defparam \Input2~I .output_power_up = "low";
defparam \Input2~I .oe_power_up = "low";

stratix_lcell \InputReg2~I (
  .clk(\Clock~combout ),
  .dataa(\Input2~combout ),
  .aclr(gnd),
  .regout(InputReg2));
defparam \InputReg2~I .operation_mode = "normal";
defparam \InputReg2~I .synch_mode = "off";
defparam \InputReg2~I .register_cascade_mode = "off";
defparam \InputReg2~I .sum_lutc_input = "datac";
defparam \InputReg2~I .lut_mask = "AAAA";
defparam \InputReg2~I .output_mode = "reg_only";

stratix_io \Input1~I (
  .combout(\Input1~combout ),
  .padio(Input1));
defparam \Input1~I .operation_mode = "input";
defparam \Input1~I .ddio_mode = "none";
defparam \Input1~I .input_register_mode = "none";
defparam \Input1~I .output_register_mode = "none";
defparam \Input1~I .oe_register_mode = "none";
defparam \Input1~I .input_async_reset = "none";
defparam \Input1~I .output_async_reset = "none";
defparam \Input1~I .oe_async_reset = "none";
defparam \Input1~I .input_sync_reset = "none";
defparam \Input1~I .output_sync_reset = "none";
defparam \Input1~I .oe_sync_reset = "none";
defparam \Input1~I .input_power_up = "low";
defparam \Input1~I .output_power_up = "low";
defparam \Input1~I .oe_power_up = "low";

stratix_lcell \InputReg1~I (
  .clk(\Clock~combout ),
  .dataa(\Input1~combout ),
  .aclr(gnd),
  .regout(InputReg1));
defparam \InputReg1~I .operation_mode = "normal";
defparam \InputReg1~I .synch_mode = "off";
defparam \InputReg1~I .register_cascade_mode = "off";
defparam \InputReg1~I .sum_lutc_input = "datac";
defparam \InputReg1~I .lut_mask = "AAAA";
defparam \InputReg1~I .output_mode = "reg_only";

stratix_lcell \OutputReg~I (
  .clk(\Clock~combout ),
  .dataa(InputReg2),
  .datab(InputReg1),
  .aclr(gnd),
  .regout(OutputReg));
defparam \OutputReg~I .operation_mode = "normal";
defparam \OutputReg~I .synch_mode = "off";
defparam \OutputReg~I .register_cascade_mode = "off";
defparam \OutputReg~I .sum_lutc_input = "datac";
defparam \OutputReg~I .lut_mask = "7777";
defparam \OutputReg~I .output_mode = "reg_only";

stratix_io \OutputPad~I (
  .datain(OutputReg),
  .padio(OutputPad));
defparam \OutputPad~I .operation_mode = "output";
defparam \OutputPad~I .ddio_mode = "none";
defparam \OutputPad~I .input_register_mode = "none";
defparam \OutputPad~I .output_register_mode = "none";
defparam \OutputPad~I .oe_register_mode = "none";
defparam \OutputPad~I .input_async_reset = "none";
defparam \OutputPad~I .output_async_reset = "none";
defparam \OutputPad~I .oe_async_reset = "none";
defparam \OutputPad~I .input_sync_reset = "none";
defparam \OutputPad~I .output_sync_reset = "none";
defparam \OutputPad~I .oe_sync_reset = "none";
defparam \OutputPad~I .input_power_up = "low";
defparam \OutputPad~I .output_power_up = "low";
defparam \OutputPad~I .oe_power_up = "low";

endmodule
```

Figure 3 - The Entire fnf.vqm file generated in **quartus_cdb**

Step 4: Modify the logic function of the Logic cell and read it back into the Quartus II design software.

Now that we have a basic understanding of the **VQM** format, we will make a slight modification to the **fnf.vqm** file, and read this back into the Quartus II design software. We will modify the logic function of the NAND gate to become a NOR gate. By inspecting Table 2, you can see that a NOR function of the signals **dataa** and **datab** can be created with the LUT mask hexadecimal value 1111, as shown in Table 3.

datad	datac	datab	dataa	LUT Output	Mask Bit Number
0	0	0	0	1	O_0
0	0	0	1	0	O_1
0	0	1	0	0	O_2
0	0	1	1	0	O_3
0	1	0	0	1	O_4
0	1	0	1	0	O_5
0	1	1	0	0	O_6
0	1	1	1	0	O_7
1	0	0	0	1	O_8
1	0	0	1	0	O_9
1	0	1	0	0	O_{10}
1	0	1	1	0	O_{11}
1	1	0	0	1	O_{12}
1	1	0	1	0	O_{13}
1	1	1	0	0	O_{14}
1	1	1	1	0	O_{15}

Table 3 - LUT Mask for 2-input NOR function of **dataa** and **datab**

Create a new directory at the same level of the fnf directory, called fnfnor, and copy the file **fnf.vqm** file into it. (Note that if you changed the name of the file to something other than fnf, you would have to change the name of the Verilog module name at the beginning of the VQM file to the same name).

```
> mkdir ..\fnfnor

> copy fnf.vqm ..\fnfnor\fnf.vqm

> chdir ..\fnfnor
```

Edit the file **fnf.vqm** and change the LUT mask of the cell **stratix_lcell \OutputReg~I** to become a NOR gate. This is done by changing the line :

```
defparam \OutputReg~I. lut_mask = "7777";
```

to:

```
defparam \OutputReg~I. lut_mask = "1111";
```

Now, we will create a new Quartus project with the new netlist with the different function, and read it into Quartus. (We are creating a new project because this is now a circuit with a different function; we will return to the original project and continue to work with it after this exercise). Run the following command to do this:

```
> quartus_map fnf --part=EP1S10F484C5
```

This command will behave similarly to the **quartus_map** command used previously, but note that it is now reading in a VQM file, not the original Verilog HDL file. (This is because there is no "**.v**" file to read, just the "**.vqm**." and so the Quartus software selects this file to read in.) You can check that the logic function interpreted by Quartus did indeed change by inspecting the **fnf.map.eqn** produced in this run of **quartus_map**, and compare it to the same file produced in the original fnf directory. This should show that the logic equation of the lcell changed from something like:

OutputReg_lut_out = !InputReg1 # !InputReg2; // This is the NAND Function

to:

OutputReg_lut_out = !InputReg2 & !InputReg1; // This is the NOR Function

Where **#** is the OR function, **&** is the AND function, and **!** is the inversion function.

The table below gives four more examples of LUT masks and their associated logic function:

LUT Mask	Logic Function (a=dataa etc.)
0xF888	ab + cd
0x6666	a xor b
0x0001	a'b'c'd' (lowest minterm set)
0x8000	abcd (highest minterm set)

3.3.1 Aside #1: Ensuring That Your Technology Mapping Is Maintained Exactly

While the VQM input format gives an exact specification of the physical netlist (giving the interconnection of the actual on-chip logic resources), it is possible that the Quartus II design software, in its default mode, will modify the netlist in order to improve it. These optimizations include "dead code elimination" (removal of logic that doesn't drive an observable output) and constant propagation. An example of constant propagation is the removal of LUTs that have all constant inputs, or registers that are reset and clocked to the same value.

To prevent the Quartus II design software from performing these optimizations, ensure the following line appears in the **.qsf** file:

```
set_global_assignment –name TRUE_WYSIWYG_FLOW ON
```

The default setting, which is not explicitly set in the .qsf file, is to turn off TRUE_WYSIWYG_FLOW. Default settings are found in the file assignment_defaults.qdf in the **quartus/bin** directory (assuming your installation's main directory was **quartus**). Changing this to ON will cause the Quartus II CAD software to use the physical netlist exactly as given in a VQM file.

3.3.2 Aside #2: Entry into Quartus Prior to Logic Synthesis

The VQM format is useful for specifying a technology-mapped netlist into lookup-tables and flip-flops as well as other basic primitives. If you wish to enter into the Quartus flow with a circuit prior to the logic synthesis step (i.e. prior to technology-independent logic optimization and technology-dependent [LUT] mapping), but after basic HDL elaboration, you can provide the Quartus II design software with a netlist of basic logic primitives instead. You can do this by using standard logic gates provided in the Verilog language. Using this method you would simply instantiate a netlist of **not, and, or, nand, nor, xor** and **xnor** gates. For example, the following code is equivalent to the fnf circuit given above:

```
// registered NAND gate design implemented using gate primitives

module fnfvgates(
      Input1,
      Input2,
      Clock,
      OutputPad);

input  Input1,Input2,Clock;
output OutputPad;

wire  InputReg1, InputReg2,  RegIn;


// Register Input1
DFF Flop1 (.d(Input1), .clk(Clock), .q(InputReg1) );

// Register Input2
DFF Flop2 (.d(Input2), .clk(Clock), .q(InputReg2) );

//  Calculate the NAND of the registered inputs
nand gate1 (RegIn , InputReg1 , InputReg2);

//  Register the output of the NAND gate
DFF Flop3 (.d(RegIn), .clk(Clock), .q(OutputPad) );

endmodule
```

You can see that the **nand** gate is instantiated with its output listed first, and the inputs following. In general, any size gate can be specified simply by listing additional inputs. The D flip-flop primitive is specified as given, with the **.d(…)** giving the signal attached to the D-input, the **.clk(..)** giving the clock, and the **.q(..)** giving the Q output.

Other primitives (including flip-flops with more features) can be found from the Quartus help menu: to find these, bring up the Quartus GUI, select **Help|Index**. Type “primitives” into the keyword box, this will bring up another list. From this list select “**list of**” to display. This will provide you with the AHDL call convention for the primitives, which are the same as the Verilog convention, moved to lower case.

This style of Verilog (gate-level netlist) can be combined with other styles, if you wish, and you can instantiate Altera LPMs including RAMs, and for example, Stratix multiplier-accumulate blocks.

To read this file into the Quartus II design software, if it was called fnfvgates.v, you would use the **quartus_map** command as above, when reading in the behavioral Verilog:

Step 1a: Read the gate-level Verilog description of the fnf function into the Quartus II design software.

```
> quartus_map fnfvgates --part=EP1S10F484C5
```

Now that you have two methods to put logic functionality into the Quartus software, we will proceed with packing, placement, routing and timing analysis steps.

3.4 Packing, Placement, and Routing

In the next step we will return to the original directory, fnf, and perform the packing, placement and routing of the circuit, using the **quartus_fit** command. In Altera terminology, to "fit" an FPGA design means to pack logic cells into logic array blocks (LABs), placing the LABs, and routing the connections between LABs.

Step 5: Do Packing, Placement and Routing

```
> chdir ..\fnf

> quartus_fit fnf
```

This command will require about one minute of execution time.

If you wish to look at the placement of the circuit using the Quartus Graphical User Interface (GUI), do one of the following:

- For Windows machines, you can double-click on the file **fnf.qpf**.
- For Unix or Linux machines start the Quartus GUI and open the **fnf.qpf** file as an existing project, using **File|Open** Project.
- Alternatively, you can enter the following command on any platform:

  ```
  > quartus fnf.qpf
  ```

Next, you can view the placement in the Altera Chip Editor by following the procedure below. (Note that even if your Quartus II Web Edition software does not support the Altera Chip Editor feature, you can still read the following, and continue to work on the remaining of this tutorial.)

1. Once the GUI is active, you should see a sub-window on the left hand side that contains a folder with the title **Compilation Hierarchies**. Under that folder you should see the name **fnf**.
2. Right click on the **fnf** name and select **Locate → Locate in Chip Editor**. This will bring up a window that displays all of the blocks in the Stratix S10 FPGA device. If it isn't in full-screen mode already, double click the title bar of the chip editor to make it full-screen. In order to make sure you can see the entire chip, type <ctrl>W (the control key and the W key) or go to **View|Fit** to display all of it.
3. Now click on the zoom tool, which is the magnifying glass that is second from the top in the toolbar. The toolbar is a vertical column of icons located to the left of the chip editor. Click on the chip editor several times to zoom in until the display changes to show individual blocks. (If your display has very good resolution, you may already see the

individual blocks). You should see a picture that looks something like that in Figure 4 below:

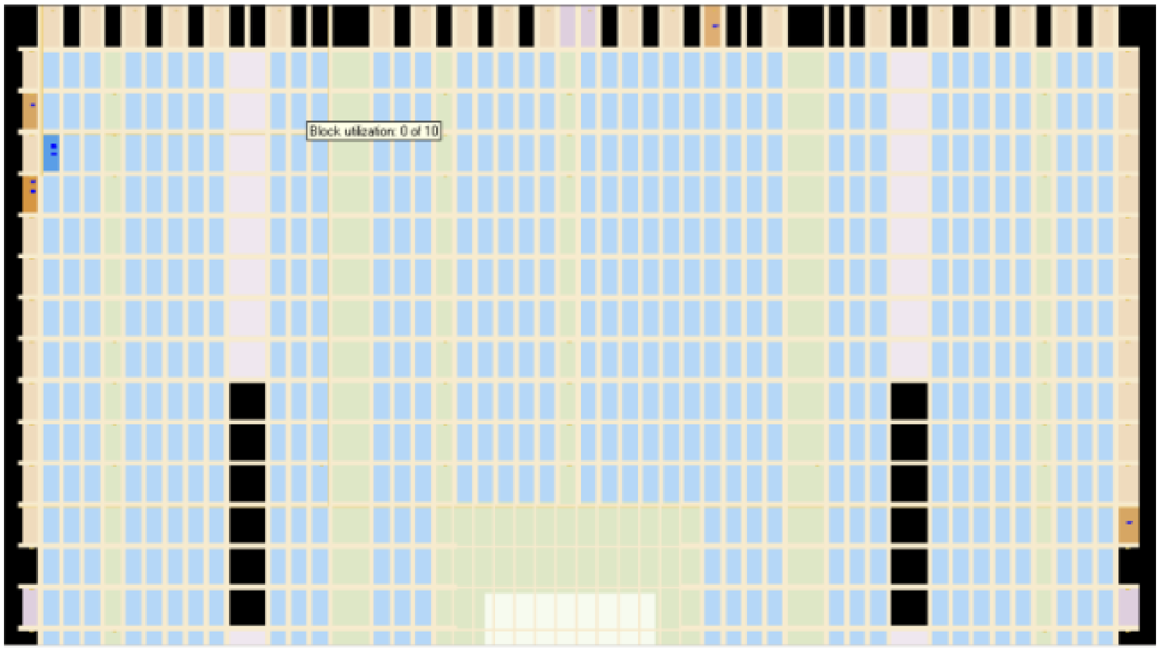


Figure 4 - Screen Shot of Chip Editor View of Placed & Routed fnf.v Circuit

4. The light orange blocks around the periphery contain the I/O pads. The light blue internal thin rectangles are Logic Array Blocks (LABs), each of which contains ten logic cells (LCELLs). Logic cells may also be known as logic elements (LE's). The columns of light green thin rectangles are small memory blocks, which contain 512 bits of memory. There are also squares that represent multiply-accumulate (DSP) blocks (light purple), medium sized memory (4K bit blocks) and one large (512K bit) memory block in the middle, shaded green. You should see several I/O pads and one LAB block shaded darker than the others, which is the mapping and placement of the fnf circuit. If you don't see these, use the scroll bars to find them.

5. Use the zoom tool to zoom in on the dark blue shaded LAB. If you zoom in far enough, you will be able to see the individual logic cells within the LAB, as shown below in Figure 5. You can see that all three logic cells have been packed into a single LAB. Holding the mouse over a logic cell will cause the editor to display the cell's X, Y location and logic cell number. Note the locations of the three cells for reference below.

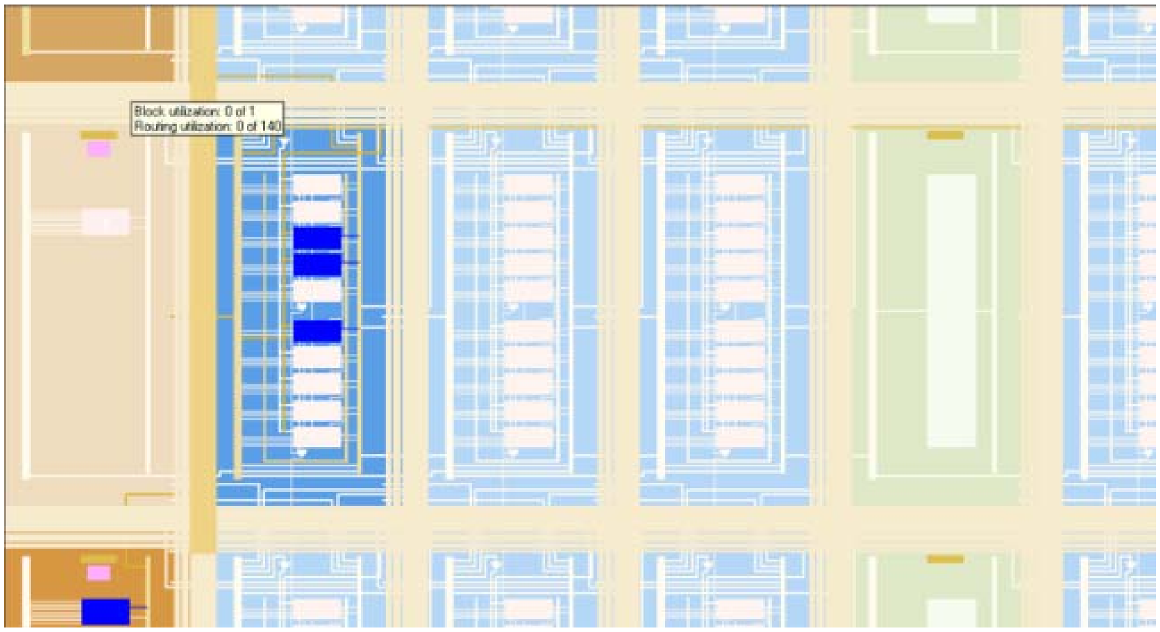


Figure 5 - Screen Shot Showing Internal Logic Elements within LAB in Chip Editor

6. Now select the pointer tool. The pointer tool should be above the zoom tool in the toolbar. Select one of the logic cells by left clicking on it, then right click on the selected logic cell and go to **Locate → Locate in Resource Property Editor**. This brings up an editor/viewer that shows the internal contents of the logic cell.

7. Return to the chip editor by closing the Resource Property Editor (or you can always use the **Locate in Chip Editor** command described earlier). You can experiment with the other features of the chip editor and read the online Quartus documentation to find out more about its features, or see [13].

8. You can also see a representation of the routing within each LAB and between the LABs. Click on the **Generate Fan-In Connections** and **Generate Fan-Out Connections** icons near the middle of the toolbar. You will see the incoming and outgoing routes of the selected cell.

Once you have finished viewing the placement and routing, you may exit the Quartus GUI by selecting **File|Exit**.

Next, we will make Quartus write out the placement into a human-readable file. This process is called **back-annotation**. We will make Quartus back-annotate the LAB positions of each logic cell (but not the specific logic cell within the LAB, although that can also be done).

Step 6: Write out the Placement of LABs to a .qsf file

```
> quartus_cdb fnf --back_annotate=lab
```

This command causes the logic cell LAB positions to be placed in the file **fnf.qsf**. Open this file in a text editor and you should find a section near the top that begins with the line:

```
# Pin & Location Assignments
```

Note that # represents a commented line. Here you will see something like this:

```
# Pin & Location Assignments

# ========================

set_location_assignment LAB_X52_Y30 -to InputReg2

set_location_assignment LAB_X52_Y30 -to OutputReg

set_location_assignment LAB_X52_Y30 -to InputReg1

set_location_assignment PIN_D1 -to Input1

set_location_assignment PIN_B4 -to Input2

set_location_assignment PIN_L2 -to Clock

set_location_assignment PIN_B2 -to OutputPad

set_location_assignment PIN_L8 -to ~DATA0~
```

The exact locations in your .qsf file may be different, since different versions of the Quartus software will generate different placements. However, the syntax and form will be the same as that shown above. Note that the coordinates are zero-based, which means that the bottom-left corner of the chip has the co-ordinates (0, 0).

This file shows that the location of the logic cell that contains the NAND gate and register (named OutputReg) is at X coordinate 52 and Y coordinate 30. You can also see that the logic cells containing just the two registers (InputReg1 and InputReg2) are located at the same position, since they were packed into the same LAB. It also shows the placement of the input, output and clock pads, using a numbering scheme based on the pin package-naming scheme. You need to know which device package you are using - the same silicon device is typically packaged in several different ways. This means that the same silicon pad will end up with different names in different packages. In this example, we're using the device EP1S10F484C5, which uses the F484 package, but there are several other options such as the B672, F672, and F780 packages, which have differing numbers of pins bonded out, and different package types.

You can find a more extensive description of the types of assignments that can be made in the QSF file in [7].

Edit the QSF file and change the location of the LAB containing the logic cell OutputReg from X position 52 to position 51. If the numbers in your QSF are different, set the location to an adjacent LAB and save your changes.

Step 7: Read modified placement back into Quartus and route it.

Rerun placement and routing by using the same **quartus_fit** command:

```
> quartus_fit fnf
```

Next, rerun the Quartus GUI as previously described, and open the chip editor. You will see that there are now two occupied LABs, one containing the single logic cell that was moved to a new LAB location.

Next, we will show how to constrain the logic elements into a fixed boundary area on the chip, using what is called a *custom region*. You can use this method to input a fixed floorplan of your circuit to the Quartus II design software.

Step 8: Compile the fnf.v Circuit in a new directory

We want to start with a clean slate so we will compile fnf.v in a new directory:

```
> mkdir ..\fnf_new

> copy fnf.v ..\fnf_new\fnf.v

> chdir ..\fnf_new

> quartus_map fnf --part=EP1S10F484C5

> quartus_fit fnf
```

Using a text editor, we will change the “Fitter Assignments” section of the file **fnf.qsf** which contains:

```
set_global_assignment -name DEVICE EP1S10F484C5
```

to the following:

```
set_global_assignment -name DEVICE EP1S10F484C5
set_location_assignment CUSTOM_REGION_X22_Y1_X25_Y6 –to OutputReg
set_location_assignment CUSTOM_REGION_X44_Y12_X48_Y16 –to InputReg1
set_location_assignment CUSTOM_REGION_X44_Y12_X48_Y16 –to InputReg2
```

The three new lines tell the Quartus software to place the named logic cells somewhere in a specific rectangular (Xlow, Ylow) (Xhigh, Yhigh) fixed region. For example, the logic cell InputReg1 will be placed in the rectangle (44,12) (48,16). If you look at the last compilation floorplan in the GUI and make the mouse hover over the LAB’s, you’ll see that this rectangle contains just LABs and no memory or multiply-accumulate blocks. In the previous example, we did not specify any location constraints on the IO cells in the circuit, so the Quartus placer was free to place them anywhere.

Save this file and then re-run the placement and routing using the **quartus_fit** command, which will read the custom region constraints automatically from the **fnf.qsf** file.

```
> quartus_fit fnf
```

Now view the placement by re-opening the GUI on the **fnf.qpf** file and view the new placement. You should see that the three logic cells are placed within the regions specified.

Using this method with the device floorplan information in [8] and the associated device file, you will be able to input your own floorplan regions into the Quartus II design software. One issue you will face is to make sure that you have the correct names for all of the elements of the circuit you wish to floorplan – these names will be available in the **.vqm** file. To refer to a function block, you can use either the name of any output signal of that block, or you can use the instance name of the block. For example, to set location constraints on the logic cell described in Figure 3 from lines 110 to 120, we could use the name **InputReg1** (an output of the logic cell), or we could use the “instance name” of **InputReg1~I**. Alternatively, you can set constraints on higher levels of a hierarchy. The associated constraint will apply to all of the logic synthesized under this level. See [7] for more details.

If you were interested in more sophisticated types of floorplanning, such as those that allow rectangular regions of logic to move throughout the chip (but still stay within the same sized region), you should use the **LogicLock** capability of the Quartus software, which is accessible through a TCL interface [14]. **LogicLock** can also automatically compute the size and location of a region in order to fit the region's associated design partition onto the chip.

3.4.1 Other Information You'll Need to Make a Complete Packing or Placement Tool

If you wish to write your own packing (or clustering) tool, you'll need to know all of the legality constraints on logic cells that can be packed into a LAB. These can be found in document [6].

In order to make a complete placement tool that places for the entire Stratix architecture, you need to know about all of the types of blocks and their positions in the Stratix chip. Altera architectures have been described in the XML format, and you can find documentation on this in the document [8]. In addition to this there are the XML files themselves, which are included with the QUIP distribution.

If you wish to make your own timing-driven placement tool, you will need information about the inter-lab delays. We have provided software that can be queried to determine the lab-to-lab delays within an Altera FPGA. See [11] for information on this interface. You will also need information on intra-lab delays, which is provided in XML format. See document [9] for information on retrieving these delays.

3.5 Constrained Routing: How to Input Routing to the Quartus Software

The Quartus software has a handy feature that allows users a broad range of control over the routing process. It is possible to precisely describe the routing you wish each connection to have (if you wished to write your own router) or to provide relatively high-level guidance as to the types of routing resources you want a particular connection to use. Another possibility is to give a *global routing* (the specification of which channel you wish a route to take, but not the specific track within a channel) and have the Quartus router complete the detailed routing. All of these things can be done by creating a *routing constraint file* (with an .rcf extension) as part of the input to the Quartus software.

We will first show how to generate an **rcf** file that fully describes the routing that the Quartus router has chosen for a circuit. This file will be used as the basis for describing the **rcf** format. We will modify the **rcf** file slightly, show how to read that back into the Quartus software, and display how the routing has changed.

To make the routing you obtain more closely match the example routing we'll discuss, it is best if we force the Quartus tools to place all the function blocks where we'd like. To do this, make sure you are in the original fnf directory. In the fnf.qsf file, delete any lines that contain the command, set_location_assignment, and insert the location assignment lines below.

```
set_location_assignment IOC_X53_Y12_N2 –to Clock
set_location_assignment LC_X52_Y30_N0 –to OutputReg
set_location_assignment IOC_X53_Y30_N0 –to OutputPad
set_location_assignment IOC_X52_Y31_N5 –to Input1
set_location_assignment IOC_X52_Y31_N2 –to Input2
set_location_assignment LC_X52_Y30_N2 –to InputReg1
set_location_assignment LC_X52_Y30_N3 –to InputReg2
```

The location constraints above highlight two new features. First, we have specified specific logic cell locations, rather than just LABs, at which the circuit logic cells should be placed. The syntax to do this is very similar to the syntax for constraining to LABs, except that a _N<number> is added after the x and y locations to specify which logic cell number within the LAB should be used. Second, we have specified the placement of the circuit IOs by saying which IO cell within the FPGA should be used to implement each of them, rather than which package pin (e.g. Pin_M6) should be used. Each package pin is connected to one IO cell in the FPGA, so either notation can be used. Often it is more convenient to use the IO cell locations, however, since they are (x,y) based and hence show which LABs are near which IO cells.

Next, run **quartus_fit**, as in Section 3.4.

Step 9: Write out a file that describes the routing produced for fnf.

To create the routing constraints file (RCF), run the following command:

```
> quartus_cdb fnf --back_annotate=routing
```

As the command suggests, this causes the routing to be “back annotated” into the file **fnf.rcf**. It also causes the placement to be back-annotated (written out to the fnf.qsf file), since a routing is meaningless if the placement of a circuit changes.

The complete fnf.rcf file produced is given below, with added line numbering (note that the file you see may have some numerical differences to this one, since the precise routing will vary from one version of the Quartus software to another):

```
1      ### Routing Constraints File: fnf.rcf
2      ### Written on:                     Mon Mar 21 12:33:52 2005
3      ### Written by:                     Version 5.0 Internal Build 129 03/10/2005 TO
    Full Version
4
5      section global_data {
6       rcf_written_by = "Quartus II 5.0 Internal Build 129";
7       device = EP1S10F484C5;
8      }
9
10     signal_name = Clock {  #IOC_X53_Y12_N2
11      CLK_BUFFER:X53Y12S2I0;
12      GLOBAL_CLK_H:X0Y19S0I9;
13      GLOBAL_CLK_V:X38Y20S0I9;
14      label = Label_LAB_CLK:X39Y30S0I5, LAB_CLK:X39Y30S0I5;
15      dest = ( InputReg2, CLK );    #LC_X52_Y30_N3
16
17      branch_point = Label_LAB_CLK:X39Y30S0I5;
18      dest = ( OutputReg, CLK );    #LC_X52_Y30_N0
19
20      branch_point = Label_LAB_CLK:X39Y30S0I5;
21      dest = ( InputReg1, CLK );    #LC_X52_Y30_N2
22     }
23     signal_name = Input1 { #IOC_X52_Y31_N5
24      IO_DATAIN:X52Y31S5I0;
25      C8:X51Y26S0I3;
26      LOCAL_INTERCONNECT:X52Y30S0I25;
27      dest = ( InputReg1, SYNCH_DATA ), route_port = DATAC;       #LC_X52_Y30_N2
28     }
29     signal_name = Input2 { #IOC_X52_Y31_N2
30      IO_DATAIN:X52Y31S2I0;
31      C4:X51Y27S0I14;
32      LOCAL_INTERCONNECT:X52Y30S0I13;
33      dest = ( InputReg2, DATAA ), route_port = DATAD;    #LC_X52_Y30_N3
34     }
35     signal_name = InputReg1 {      #LC_X52_Y30_N2
36      LOCAL_LINE:X52Y30S0I2;
37      dest = ( OutputReg, DATAB ), route_port = DATAD;     #LC_X52_Y30_N0
38     }
39     signal_name = InputReg2 {      #LC_X52_Y30_N3
40      LOCAL_LINE:X52Y30S0I3;
41      dest = ( OutputReg, DATAA ), route_port = DATAC;     #LC_X52_Y30_N0
42     }
43     signal_name = OutputReg {      #LC_X52_Y30_N0
44      LE_BUFFER:X52Y30S0I1;
45      LOCAL_INTERCONNECT:X53Y30S0I2;
46      IO_DATAOUT:X53Y30S0I0;
47      dest = ( OutputPad, DATAIN ); #IOC_X53_Y30_N0
48     }
```

Note that the pound sign (#) is used to indicate comment fields. The comments at the beginning indicate the file name, Quartus version and date that the file was written.

Line 5 is the start of the global_data section. It describes the Quartus version used to produce the **rcf** file, and the device target used in the compile.

Lines 23 - 28 describe a routing path created by the Quartus router. The identifier **signal_name** specifies the signal that is being routed. Here it is **Input1**, which is sourced by an I/O. The comment at the end of the line (#IOC_X52_Y31_N5) gives the location of the signal's source, which is an I/O block, located at position (52, 31), sub-location 5. (A sub-location is used to indicate one of a number of elements that can be present at the same X,Y location - in this case up to 6 I/Os can reside at a given sub-location.) See the document [8] for a detailed description of the coordinate system.

Lines 24 - 26 list the routing resources used to connect Input1 to its sink, or destination. The key word IO_DATAIN represents a connection between IO block and the chip's core. The line beginning with "C8" indicates that the next routing resource is a column wire of length 8. As indicated by the string **X51Y26S0I3** this C8 begins at X position 51, Y position 26, has no sublocation (most routing wires have a 0 sublocation field) and is a C8 with index number 3. (Note that there can be several C8 wires starting at the same X,Y location, and so they are distinguished by the index number).

Line 26 begins with LOCAL_INTERCONNECT, which is a wire that spans the entire internal height of the LAB, and is used to provide connectivity to the LUTs and flip-flops (i.e. the logic elements) in the LAB.

The fact that these three routing resources (IO_DATAIN, C8 and LOCAL_INTERCONNECT) are listed in order implies that they are connected in series.

Line 27 specifies the destination of the route. `(InputReg1, SYNCH_DATA)` identifies the block and a port attached to that block. This port is the sink of the connection. A (block, port) pair defines the destination of a signal. Here, InputReg1 refers to logic cell that drives signal named InputReg1, and `SYNCH_DATA` is an input port on the InputReg1 block (that is D input to the flip-flop). The section "`route_port = DATAC`" gives additional information on what the router actually connected the signal to. In some cases, the destinations (such as DATAA, B, C and D) are logically equivalent, and the router is free to choose among any of these destinations.

The comment at the end of line 27 (#LC_X52_Y30_N2) gives the X,Y and sublocation of the logic element that is the sink.

The overall coordinate system for routing resources is illustrated in Figure 6.

More details on the syntax and semantics of the routing constraint file format can be found in [10].

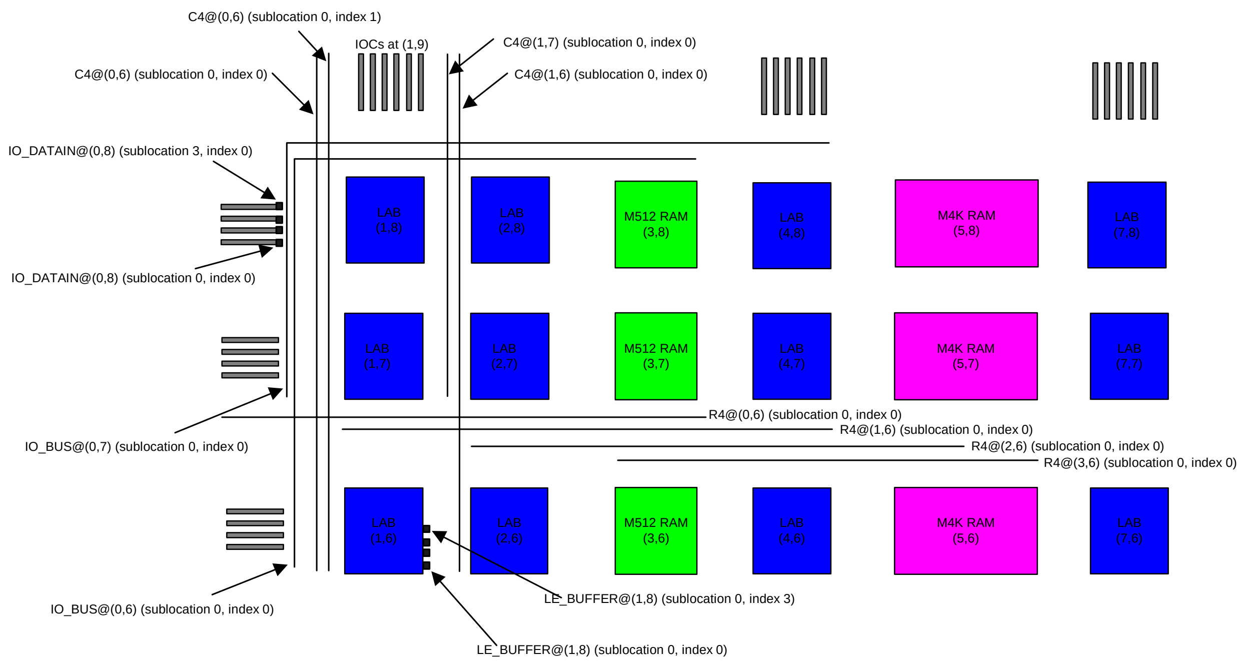

C4@(0,6) (sublocation 0, index 1)
IOCs at (1,9)
C4@(1,7) (sublocation 0, index 0)
C4@(0,6) (sublocation 0, index 0)
C4@(1,6) (sublocation 0, index 0)
IO_DATAIN@(0,8) (sublocation 3, index 0)
LAB (1,8)
LAB (2,8)
M512 RAM (3,8)
LAB (4,8)
M4K RAM (5,8)
LAB (7,8)
IO_DATAIN@(0,8) (sublocation 0, index 0)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

Next, we will modify the routing as given in the back-annotated **rcf** file, and view this change after the Quartus software reads this file back in.

Step 10: Modify the Constrained Routing File and Read it Into the Quartus Software

We will view the routing created by the Quartus II design software in the Chip Editor, and then we will change the rcf file, and view the change. As described in Section 3.4, start the Quartus GUI by double-clicking on the file **fnf.qpf**.

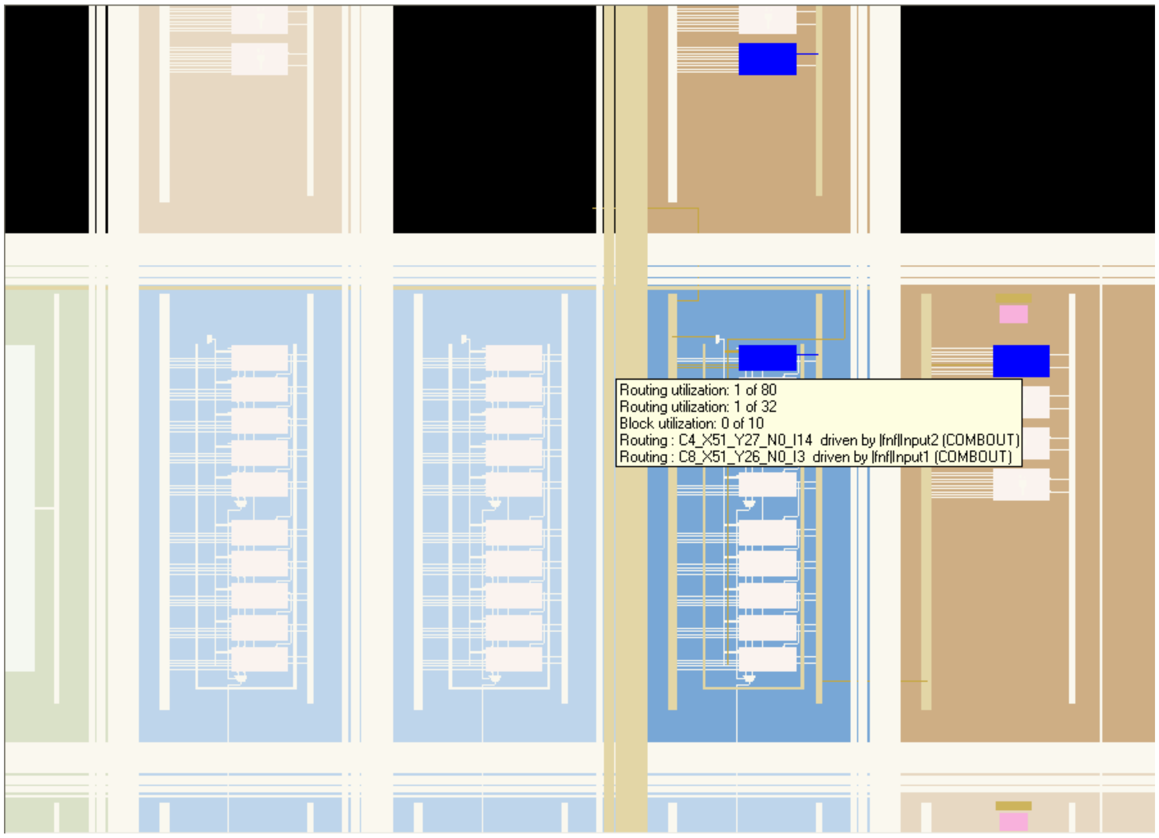
Select **Tools|Options** from the upper menu, and click on the **Chip Editor** category. Set the **Show names in tooltip** setting to **Unlimited** (this keeps the information visible for as long as the mouse hovers). Also, set the **Delay Showing Tooltip for** to **0** seconds, so that you can immediately see the routing information.

In the left hand side window, below the words **Compilation Hierarchies**, right click on **fnf** and select **Locate → Locate in Chip Editor**. This will bring up the chip editor again. Select **View|Detailed Tooltip** from the top menu row. This will cause the editor to show resource names similar to those in the RCF when the mouse hovers over the routing channel.

Now make the chip editor full-screen, if it isn't already, by double clicking the title bar.

Zoom into the upper right hand (or wherever the LAB ended up) region that contains the single LAB with three logic elements until you can see the actual logic elements themselves.

Place the mouse over the coloured vertical column (there should be only one). A tooltip text will show the set of vertical wires being used in this column, which are two wires driving from the two input pads, which are named in the **rcf** file: `C8:X51Y26S0I3` and `C4:X51Y27S0I4`, as shown below in Figure 7. In general, the chip editor will keep channel tracks visually grouped together, but the **Detailed Tooltip** feature will indicate all of the used wires in the group.



```
Routing utilization: 1 of 80
Routing utilization: 1 of 32
Block utilization: 0 of 10
Routing : C4_X51_Y27_N0_I14  driven by |fnf|Input2 (COMBOUT)
Routing : C8_X51_Y26_N0_I3  driven by |fnf|Input1 (COMBOUT)
```

Figure 7 - Screen Shot Showing Routing in Chip Editor

Next we will change the routing specified in the **rcf** file, read it back into the Quartus router, re-route, and view the change.

Open the file **fnf.rcf** in any text editor and change lines 25 and 26 from:

```
C8:X51Y26S0I3;
LOCAL_INTERCONNECT:X52Y30S0I25;
```

to

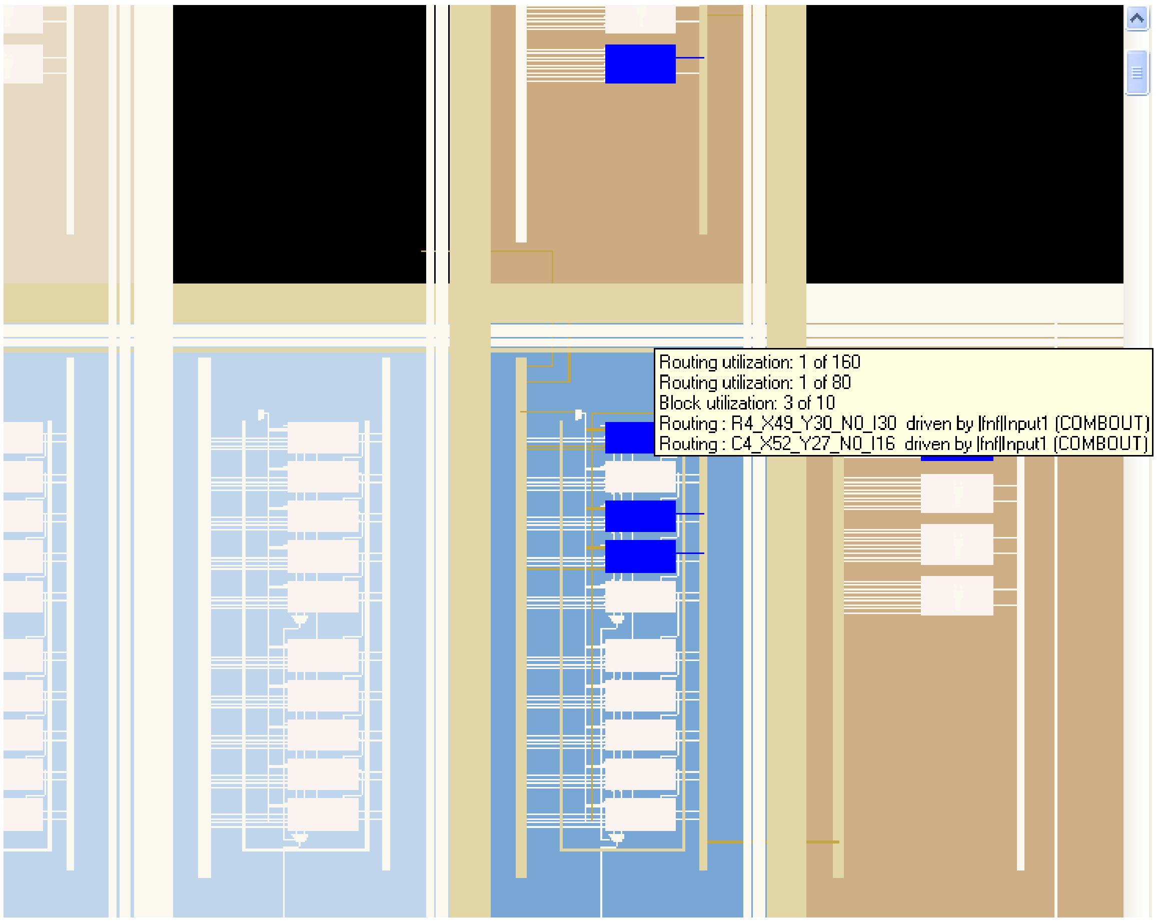
```
C4:X52Y27S0I16;
R4:X49Y30S0I30;
LOCAL_INTERCONNECT:X52Y30S0I22;
```

This will cause the connection from signal Input1 to be routed in a more circuitous path, using one C4 to the right of the current one, and then connecting to a row wire R4 (also known as horizontal R4), and then into the same LAB.

Re-run the placement and routing, as above by typing:

> quartus_fit fnf --set ROUTING_BACK_ANNOTATION_MODE=NORMAL

Now, re-invoke the chip editor as described above. (You should find that all of the settings you set above, such as **Detailed Tooltips**, remain set.) After zooming into the same area of the chip, you should now see that a new column of routing is occupied, as indicated by colouring. Place the mouse over the intersection between the new vertical wire and horizontal wire. In the tooltip text you should see the C4 specified above, `C4:X52Y27N0I16`, and the new row, `R4:X49Y30N0I30`, as shown below in Figure 8. Thus you can see that the routing constraint file indeed dictates the routing!



```
Routing utilization: 1 of 160
Routing utilization: 1 of 80
Block utilization: 3 of 10
Routing : R4_X49_Y30_N0_I30  driven by |fnf|Input1 (COMBOUT)
Routing : C4_X52_Y27_N0_I16  driven by |fnf|Input1 (COMBOUT)
```

Figure 8 - Screen Shot of Modified Routing using Routing Constraint File (RCF)

For more information on the details of the routing constraint file and architecture see [10]. You can write a global router using wild carding. Wild cards are described in detail in [10]. Briefly, you can replace the instance number (the number following the "`I`" at the end of the C4, R4, LOCAL_INTERCONNECT shown above) with an asterisk ("*"), telling the router to choose any available and connected resource. In this way you can specify a global route, with the X,Y indications, but not the details.

3.6 Timing Analysis

To determine the maximum rate at which this design can be clocked, we need to invoke the timing analyzer:

Step 10: Invoke the timing analyzer

> quartus_tan fnf

This produces a file called **fnf.tan.rpt** which reports on the various delay paths of the designed circuit, and gives its clock operating frequency.

Open this file using a text viewer. Go to Section 2, which is the Timing Analyzer Summary. The Timing Analyzer Summary summarizes the timing information for the design and should look something like this:

```
+------------------------------------------------------------------------------------------
; Timing Analyzer Summary
+-------------------------------+-------+---------------+----------------------------------
; Type                          ; Slack ; Required Time ; Actual Time
+-------------------------------+-------+---------------+----------------------------------
; Worst-case tsu                ; N/A   ; None          ; 2.291 ns
; Worst-case tco                ; N/A   ; None          ; 6.106 ns
; Worst-case th                 ; N/A   ; None          ; -2.102 ns
; Clock Setup: 'Clock'          ; N/A   ; None          ; 1131.22 MHz ( period = 0.884 ns )
; Total number of failed paths ;       ;               ;
+-------------------------------+-------+---------------+----------------------------------
```

This report indicates that the clock rate for this design is 1131.22 MHz. It also indicates that the clock setup time is 2.291 ns and the clock to output delay time is 6.106 ns.

3.7 Programming File Generation

Finally, to generate the programming file needed to actually program chips, type:

Step 11: Invoke the programming file generator.

> quartus_asm fnf

This will produce the file **fnf.sof**, which contains the programming information for the device. For details on how to download this file to an Altera FPGA consult the “Tutorial – Using Quartus II CAD Software” [1].

4. More Details

Now that you have the basics of how to use the various phases of the Quartus flow, you can learn more details by reading the documents in Section 5. Some important documents to look at are the following:

- When comparing different CAD flows, you should make sure to follow the recommendations given in the document “Benchmarking Using the Quartus University Interface Program (QUIP)” (quip_benchmarking.pdf) [15].

- QUIP includes a set of benchmark circuits that you can use to experiment with. The designs are described in the document “Benchmark Designs For The Quartus University Interface Program (QUIP)” (quip_benchmarks.pdf) [16].

If you have specific questions about the using the QUIP interfaces, you can either:

1. Send mail to QUIP@altera.com with your query, or

2. Post a question to the comp.arch.fpga newsgroup, available for example, at http://groups.google.ca/groups?hl=en&lr=&ie=UTF-8&group=comp.arch.fpga

Also, see [12] for more information about the options and operation of the command-line programs for the Quartus II software.

5. Referenced Documents

All documents are found in the quip/documents directory of this distribution.

1. "Tutorial – Using Quartus II CAD Software." (quartus_tutorial.pdf)

2. "VQM Extractor Functional Description." (vqmx_doc.pdf)

3. "WYSIWYG Device Primitives User Guide For Stratix." (stratix_wysuser_doc.pdf)

4. "Stratix RAM WYSIWYG User Guide." (stratix_ram_wys_user.pdf)

5. "Stratix MAC WYSIWYG Description." (stratix_mac_wys_user.pdf)

6. "Stratix EDA and Academic Developer Functional Description." (stratix_eda_academic_fd.pdf)

7. "QSF Assignment Descriptions Document." (qsf_assignment_descriptions.pdf)

8. "Altera XML Architecture Description File Detailed Design" (altera_xml_architecture_description_file_detailed_design.pdf)

9. "Altera XML Point To Point Delay File Detailed Design." (altera_xml_point_to_point_delay_file_detailed_design.pdf)

10. "Constrained Routing Tutorial and References." (quip_rcf.pdf)

11. "Altera EDA PLDM Specifications" (altera_eda_pldm_specifications.pdf)

12. "Command-Line Scripting in the Quartus II Software," Altera Application Note #309, available at http://www.altera.com/literature/an/an309.pdf

13. "Using the Quartus II Chip Editor," Altera Application Note #310, available at http://www.altera.com/literature/an/an310.pdf.

14. "Using the LogicLock Methodology in the Quartus II Design Software," Altera Application Note #161, http://www.altera.com/literature/an/an161.pdf.

15. "Benchmarking Using the Quartus University Interface Program (QUIP)" (quip_benchmarking.pdf)

16. "Benchmark Designs For The Quartus University Interface Program (QUIP)" (quip_benchmarks.pdf)