Creating a reconfigurable FPGA system

Author: Vegard Endresen

1 Introduction

This tutorial will explain the steps to create a reconfigurable FPGA system following the framework for self reconfigurable systems developed by Sverre Hamren. The framework has only been proven to work with the Suzaku-sz410 so it is assumed that the reader following this tutorial is using it as target device. Readers should also have completed [1] and [2]. Xilinx Design Suite 10.1 is used in this tutorial.

2 Design steps

2.1 Platform Studio design

The first step is to create a Platform Studio project for the design. Download a base project from http://suzaku-en.atmark-techno.com/filebrowser/fpga_proj. Sz410-20090427.zip is used in this tutorial. Extract the project and open it in Platform Studio.

First an ICAP module needs to be added to the design. Go to the IP window, right click the *FPGA Internal Configuration* IP and click *Add IP* as shown in figure 2.1. If the module does not appear in the IP list go to *edit->preferences* , select *IP Catalog and IP Config Dialog* in the category list in the window that opens and check the *Display Available IP cores (including legacy PLB/OPB cores) in IP catalog* checkbox.

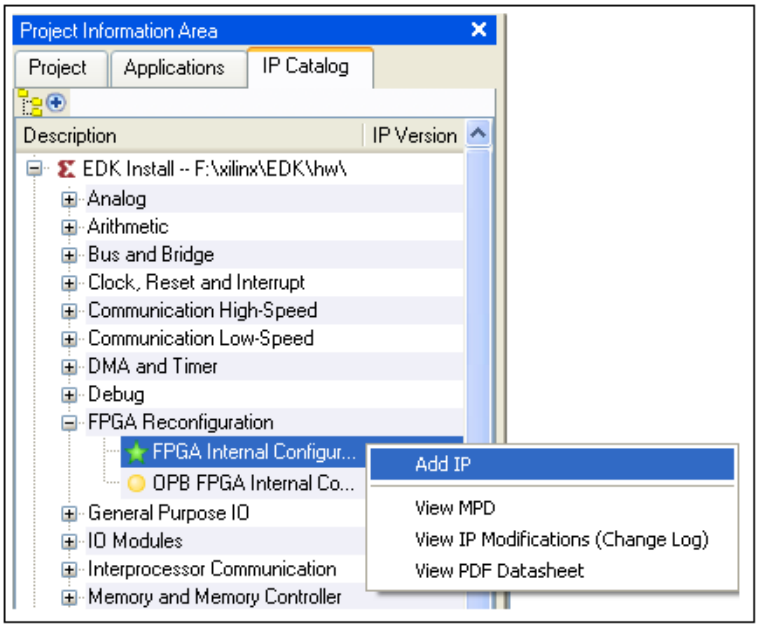

Figure 2.1 Add ICAP

A module called xps_hwicap0 should appear in the system assembly window. Select base address 0xF0F000000 and connect it PLB.

In order to enable ICAP the values of pins M2-M0 in the project's bitgen.ut needs to be changed. Go to project information area and open the bitgen.ut file.

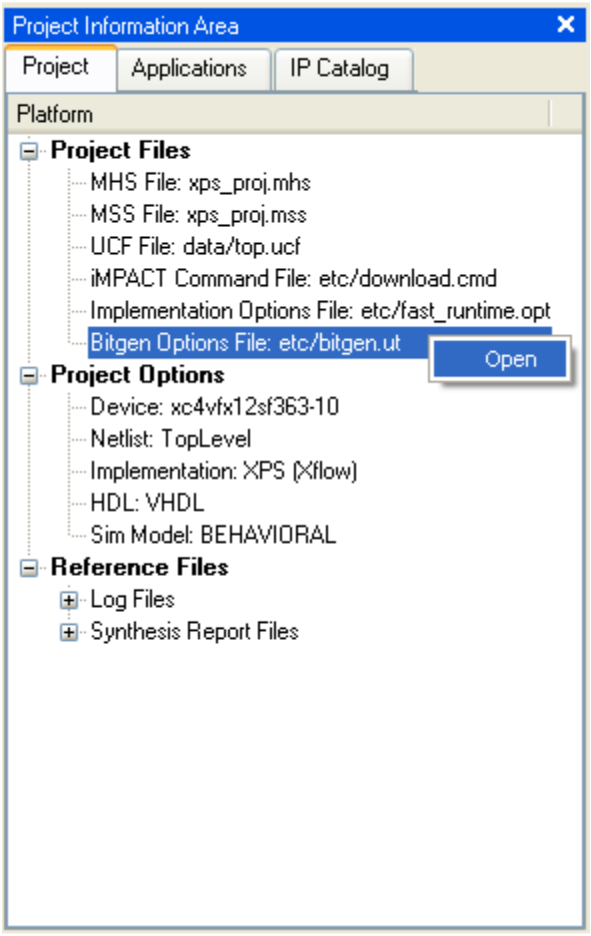

Figure 2.1.2 Open bitgen.ut

Change the values of pins M2-M1 according to source 2.6.1.

```
-g M0Pin:PULLDOWN
-g M1Pin:PULLUP
-g M2Pin:PULLUP
```

Source 2.1.1

These pin values are necessary to enable the ICAP interface. The default values of are enabling JTAG which blocks the ICAP interface.

Next create a new peripheral by *Hardware -> Create or I*mport *Peripheral*. Name the peripheral *reconf*. Use PLB as communication bus and specify one SW accessible register with a size of size 32 bits. After creating the peripheral, add it to the design from IP-Catalog->User, set base address to 0x81000000 and set bus connection to PLB in the *System Assembly View*.

Open the peripheral's user_logic.vhd file by going to *System Assembly Window*, right click *reconf*, select *Browse HDL Sources* and select user_logic.vhd in the file browser. Add two ports in the user port region as shown in source 2.1.2.

```
-- ADD USER PORTS BELOW THIS LINE ------------------
    in_16    : in std_logic_vector(0 to  C_SLV_DWIDTH/2-1);
    out_16 : out std_logic_vector(0 to  C_SLV_DWIDTH/2-1);
-- ADD USER PORTS ABOVE THIS LINE ------------------
```

Source 2.1.2

Add source 2.1.3 to the main process.

```
out_16 <= slv_reg0(0 to 15);
```

Source 2.1.3

Modify SLAVE_REG_WRITE_PROC according to source 2.1.4.

```
SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
Begin

if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        if Bus2IP_Reset = '1' then
                slv_reg0 <= (others => '0');
        else
                slv_reg0(16 to 31)   <= in_16;
                case slv_reg_write_sel is
                        when "1" =>
                                for byte_index in 0 to 1 loop
                                if ( Bus2IP_BE(byte_index) = '1' ) then
                                        slv_reg0(byte_index*8 to byte_index*8+7) <=
                                Bus2IP_Data(byte_index*8 to byte_index*8+7);
                                end if;
                                end loop;
                        when others => null;
                end case;
        end if;
 end if;

end process SLAVE_REG_WRITE_PROC;
```

Source 2.1.4

Close user_logic.vhd and open reconf.vhd. Define two components in the architecture declaration region as shown in source 2.1.5. Notice that the names of the components are the same as the names of the two NMC files included in the tutorial files.

```
Architecture IMP of reconf is
	component busmacro_xc4v_l2r_async_narrow is
		port (
			input0 : in std_logic;
			input1 : in std_logic;
			input2 : in std_logic;
			input3 : in std_logic;
			input4 : in std_logic;
			input5 : in std_logic;
			input6 : in std_logic;
			input7 : in std_logic;
			output0 : out std_logic;
			output1 : out std_logic;
			output2 : out std_logic;
			output3 : out std_logic;
			output4 : out std_logic;
			output5 : out std_logic;
			output6 : out std_logic;
			output7 : out std_logic
		);
	end component;

	component busmacro_xc4v_r2l_async_narrow is
		port (
			input0 : in std_logic;
			input1 : in std_logic;
			input2 : in std_logic;
			input3 : in std_logic;
			input4 : in std_logic;
			input5 : in std_logic;
			input6 : in std_logic;
			input7 : in std_logic;
			output0 : out std_logic;
			output1 : out std_logic;
			output2 : out std_logic;
			output3 : out std_logic;
			output4 : out std_logic;
			output5 : out std_logic;
			output6 : out std_logic;
			output7 : out std_logic
		);
	end component;
	…
```

Source 2.1.5

Also declare four signals in the same region as shown in source 2.1.6.

```
Architecture IMP of reconf is
…
signal bm_l2r_in : std_logic_vector(0 to 15);
signal bm_l2r_out : std_logic_vector(0 to 15);

signal bm_r2l_in : std_logic_vector(0 to 15);
signal bm_r2l_out : std_logic_vector(0 to 15);
…
begin
…
```

Source 2.1.6

Now create two instances of each of the components declared in source 2.1.5, according to source 2.1.7. Port map for the components are given in table 2.1.1.

```
…
begin
bm_l2r_0 : component busmacro_xc4v_l2r_async_narrow
        port map(
                …
        );

bm_l2r_1 : component busmacro_xc4v_l2r_async_narrow
        port map(
                …
        );

bm_r2l_0 : component busmacro_xc4v_l2r_async_narrow
        port map(
                …
        );

bm_r2l_1 : component busmacro_xc4v_l2r_async_narrow
        port map(
                …
        );
```

Source 2.1.7

Instance	Input	Output
bm_l2r_0	signal bm_l2r_in(0 to 7)	signal bm_l2r_out(0 to 7)
bm_l2r_1	signal bm_l2r_in(8 to 15)	signal bm_l2r_out(8 to 15)
bm_r2l_0	signal bm_r2l_in(0 to 7)	signal bm_r2l_in(0 to 7)
bm_l2r_0	signal bm_r2l_in(8 to 15)	signal bm_r2l_in(8 to 15)

Table 2.1.1 Component port maps

I am not writing out the full port map here, look in reconf.vhd among the tutorial files if portmap is unclear.

Next create a new file called reconfigurable_module.vhd in /pcores/reconf_v1_00_a/hdl/vhdl. Open this file in Platform Studio and enter source 2.1.8.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reconfigurable_module is
	port (
		in_16    : in   std_logic_vector(0 to 15);
		out_16   : out  std_logic_vector(0 to 15)
	);
end entity reconfigurable_module;

architecture arch of reconfigurable_module is
begin
	out_16(0 to 7) <= in_16(0 to 7) -  in_16(7 to 15);
	out_16(8 to 15) <= (others => '0');
end arch;
```

Source 2.1.8

Go back to reconf.vhd and create an instance of the newly defined entity as shown in source 2.1.9.

```
R : entity reconfigurable_module
port map (
	in_16 => bm_l2r_out,
	 out_16 => bm_r2l_in
  );
```

Source 2.1.9

Last find the user_logic instance in reconf.vhd and add the two lines to the port map according to source 2.1.10.

```
USER_LOGIC_I : entity reconf_v1_00_a.user_logic
…
 -- MAP USER PORTS BELOW THIS LINE ------------------
	in_16    => bm_r2l_out,
```

```
        out_16 => bm_l2r_in,
-- MAP USER PORTS ABOVE THIS LINE ------------------
```

Source 2.1.10

The resulting design peripheral design is shown in figure 2.1.3. One SW accessible register is used as input and output for a reconfigurable module, performing addition as defined in source 2.1.8, through the ports added to user_logic in source 2.1.2. The special feature in this design is that the input and output signals for the reconfigurable module are routed through instances of the components declared in source 2.1.5. The two component types are hard bus macros where the component type referred to the two attached NMC files by name.

The idea is to use the reconf peripheral as a backend module for the reconfigurable module. By creating proper placement constraints in Planahead it is possible to reconfigure only the reconfigurable module and keeping the connection to processor through reconf and bus macros.

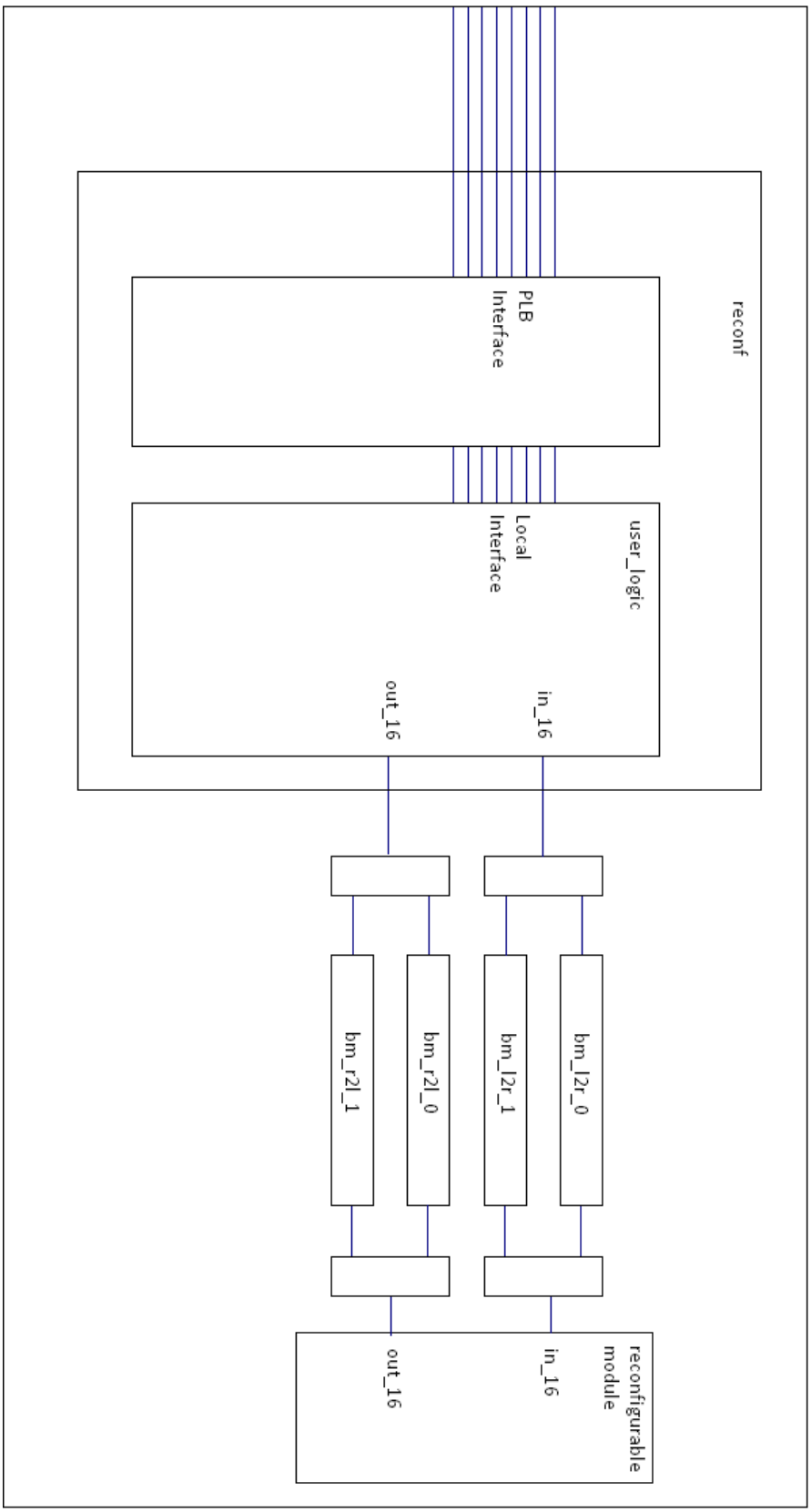

Figure 2.1.3 Reconfigurable design

Before synthesis, any external VHDL files that are used as sources for the peripheral need to be added in the peripheral's PAO file. For this design pcores/reconf_v1_00_a/data/reconf.pao is edited according to source 2.1.11. The sequence of the file listing indicates order of synthesis, thus reconfigurable_module.vhdl needs to be added above reconf.vhdl.

```
…
lib reconf_v1_00_a reconfigurable_module vhdl
lib reconf_v1_00_a user_logic vhdl
lib reconf_v1_00_a reconf vhdl
```

Source 2.1.11

The design can now be synthesized by *Hardware -> Generate Netlist* in Platform studio.

2.2 Placement constraints

After synthesis is complete create a new Planahead project according to [2], but instead of importing the UCF file from the Platform Studio project import *planahead.ucf* from the UCF folder attached to this tutorial. This should produce the floorplan in figure 2.2.1 where all modules in the design have been placed into Pblocks. To separate the reconfigurable design from the static design, the bus macros and the reconfigurable module need to be unassigned from the Pblock holding reconf. In the netlist window expand reconf, expand primitives and scroll down. The reconfigurable module can be found by its name R, changed to R0 by synthesizing tools. Mark all the primitives of R0 and unassign them as indicated in figure 2.2.1

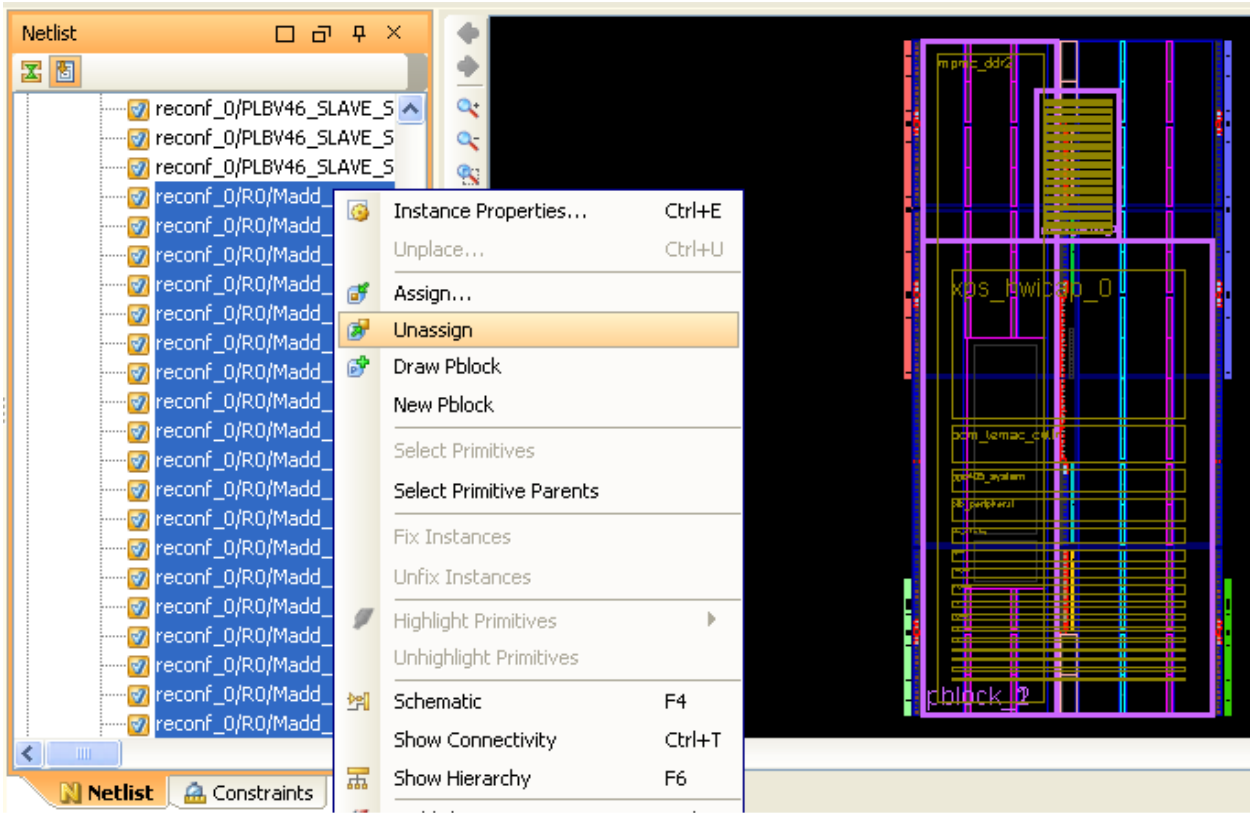

Figure 2.2.1 Unassign reconfigurable module

Unassign the bus macros in the same way as shown in figure 2.2.2. Next export the floorplan to the file top.ucf and import the constraints into the Platform Studio project as described in [2].

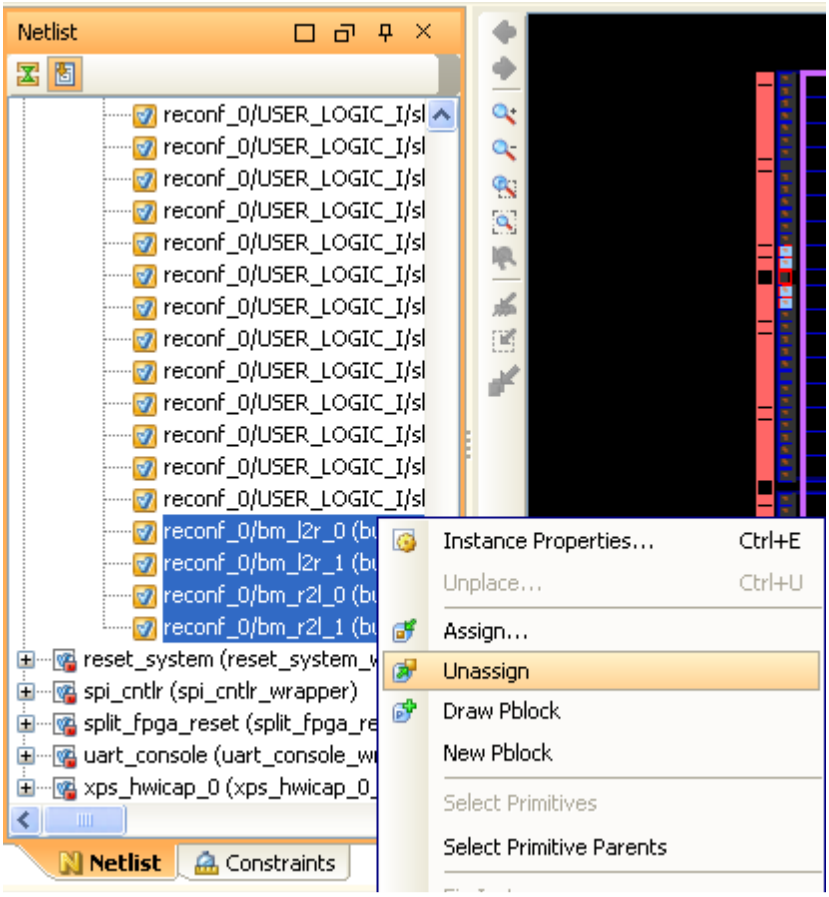

Figure 2.2.2 Unassign bus macros

I have had no luck using Planahead to place bus macros, so this is done manually. To place the bus macros add source 2.2.1 to the Platform Studio's procject UCF file.

```
INST "reconf_0/reconf_0/bm_l2r_0" LOC = "SLICE_X42Y126";
INST "reconf_0/reconf_0/bm_l2r_1" LOC = "SLICE_X42Y124";
INST "reconf_0/reconf_0/bm_r2l_0" LOC = "SLICE_X42Y122";
INST "reconf_0/reconf_0/bm_r2l_1" LOC = "SLICE_X42Y120";
```

Source 2.2.1

This specifies the absolute position of the bus macros in the design in terms of slice coordinates. Slice coordinates can be inspected in Planahead by zooming in the device window. Slice X42Y126 is highlighted in figure 2.2.3 as an example. The reconfigurable module is small in this case and will be placed next to the bus macros anyway, so no placement constraints are specified for it.

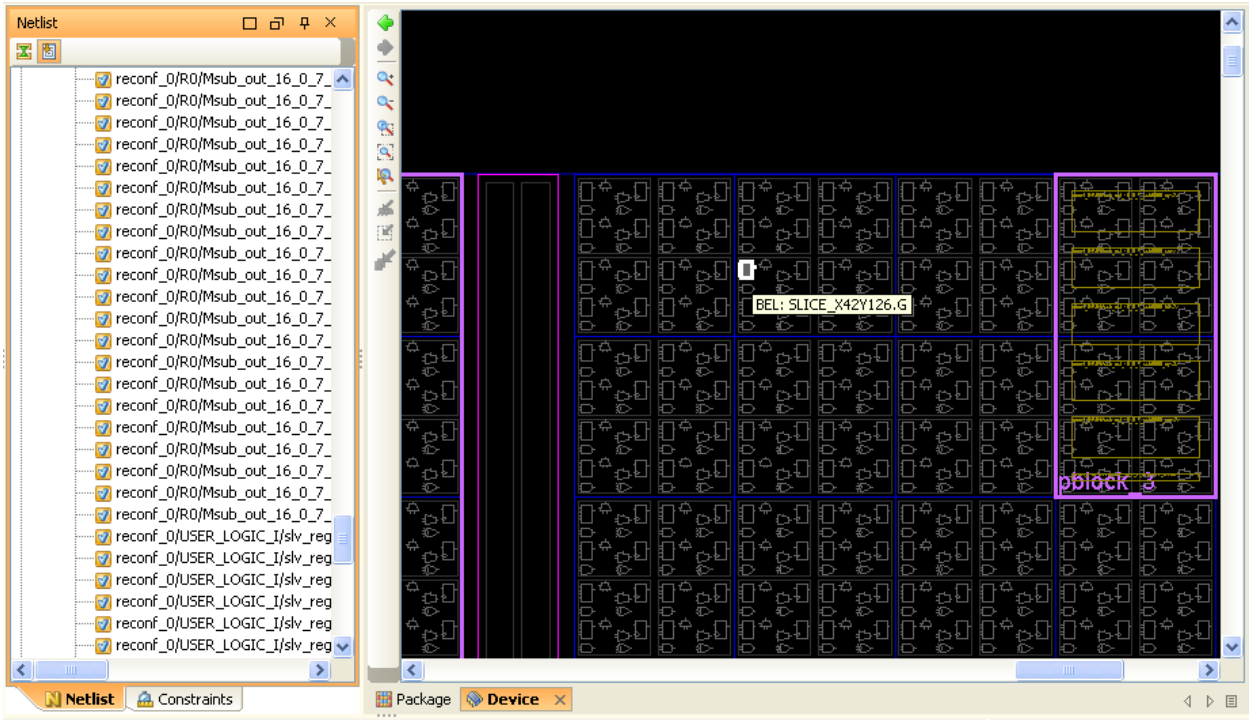

Figure 2.2.3 Slice coordinates

Before running PAR, the two NMC files need to be copied into the /implementation folder. Next run PAR by *Hardware -> Generate Bitstream* in Platform Studio

2.3 Rerouting in FPGA editor

After bitstream generation is complete, open *xps_proj.ncd* in the */implementation* folder with FPGA editor. This file contains the routed design created by PAR. Inspect the top right corner, it should look like in figure 2.3.1 but different versions and patches of Xilinx tools might give different results. The bus macros are highlighted in white and the logic of the reconfigurable module is contained in the rightmost CLB column. Routing in and out of a reconfigurable module must be equal before and after reconfiguration to avoid wire corruption. To achieve this, the bus macros ends connected to the reconfigurable logic are regarded as a part of the reconfigurable module itself and the bus macros are the only routing allowed in and out of the reconfigurable module. According to this the nets highlighted in red in figure 2.3.1 must be rerouted.

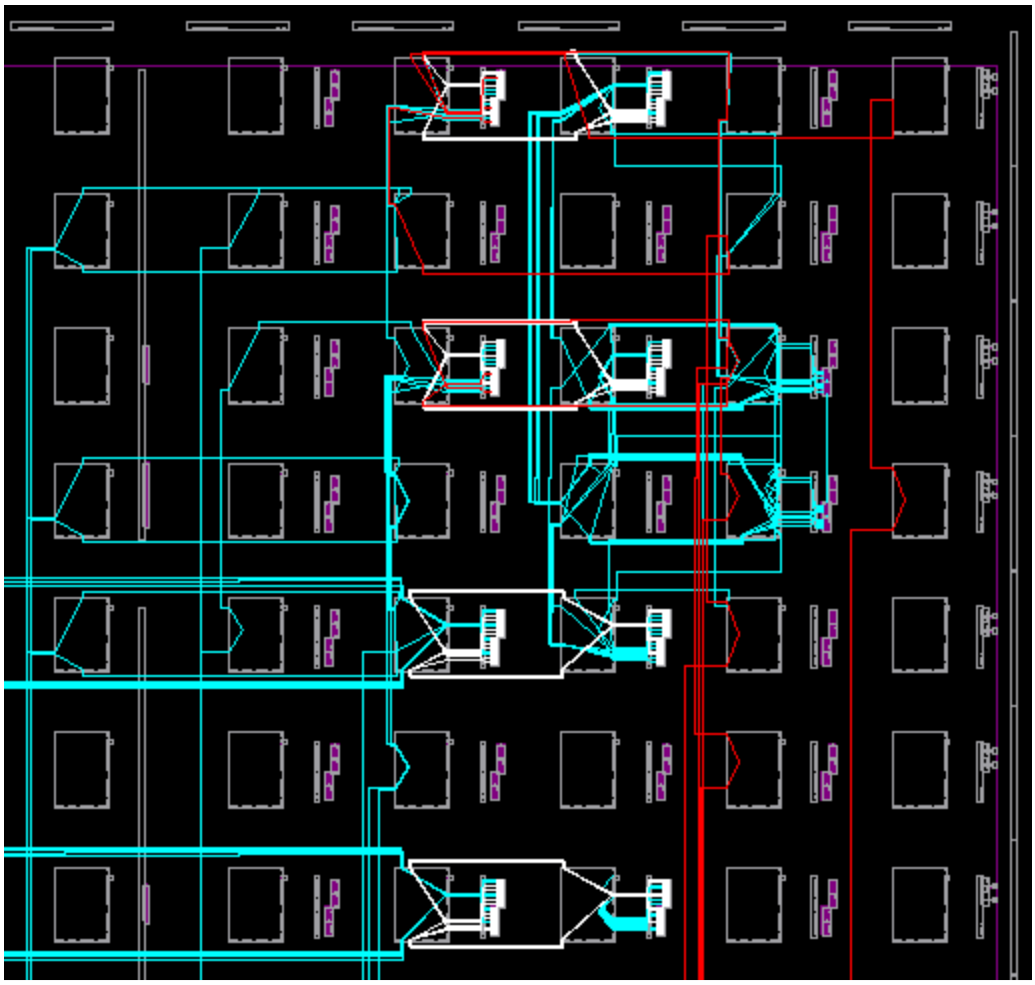

Figure 2.3.1 FPGA editor – Routed design

The process of rerouting all the nets in figure 2.3.1 is not demonstrated here as it requires many steps. Instead I will demonstrate one way of rerouting a single net.

First create a copy of the NCD file that is to be rerouted and open the copy in FPGA editor. If some irreversible error occurs during editing, it is possible to start from the original file without running PAR over again. First select a net that that must be rerouted, like in figure 2.3.2, and click the unroute-button in the right side menu. The purpose is to route the net so that it does not cross CLB columns 22 and 23. Scroll to the left in FPGA editor and find an unused slice to route the net through. Add the slice to the design as shown in figure 2.3.3. Name the slice *slice0*. If it is not possible to add a slice, toggle the editmode button to get into write-mode.

In the list box, select unrouted nets as filter and find the net that was unrouted earlier as shown in figure 2.3.4. Add an input pin on *slice0* to the unrouted net by selecting an input pin on *slice0* and the unrouted net from the list and press add. Select the net again and press unroute. Next select only the pin added to *slice0* and the bus macro destination pin and press autoroute. The result is showed in figure 2.3.5 where the net now has been forced to route left due to *slice0*.

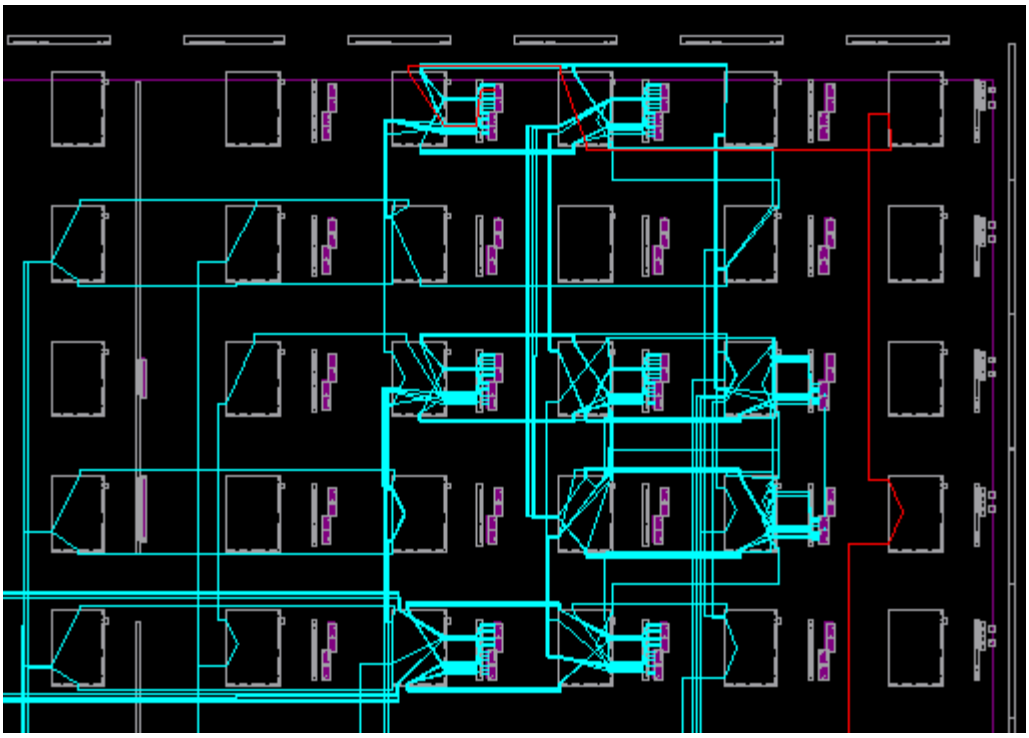

Figure 2.3.2 Select and unroute

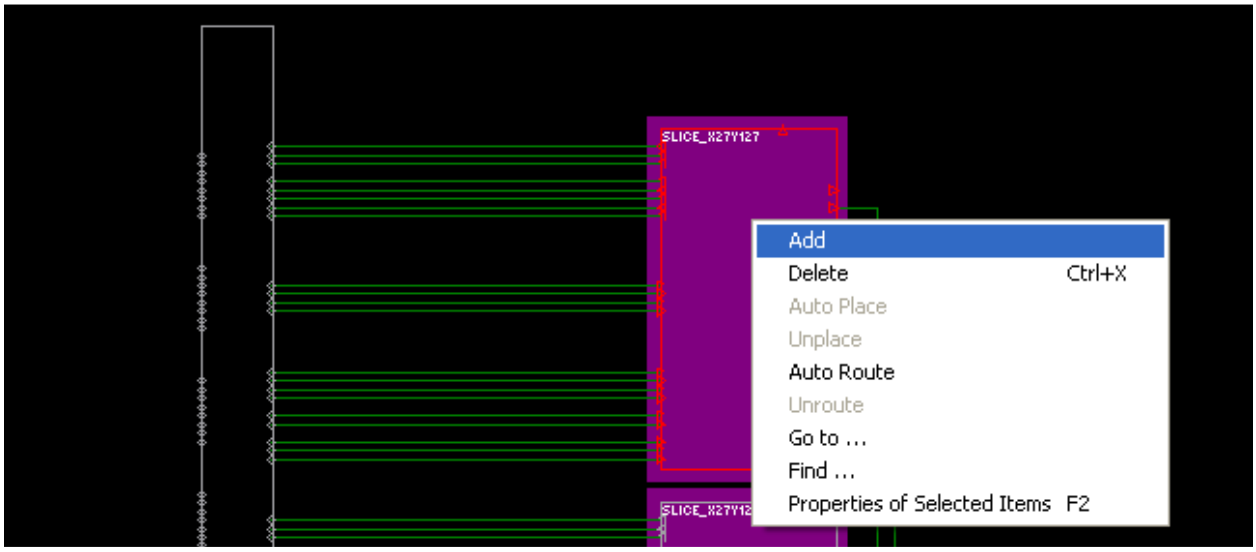

Figure 2.3.3 Add a slice

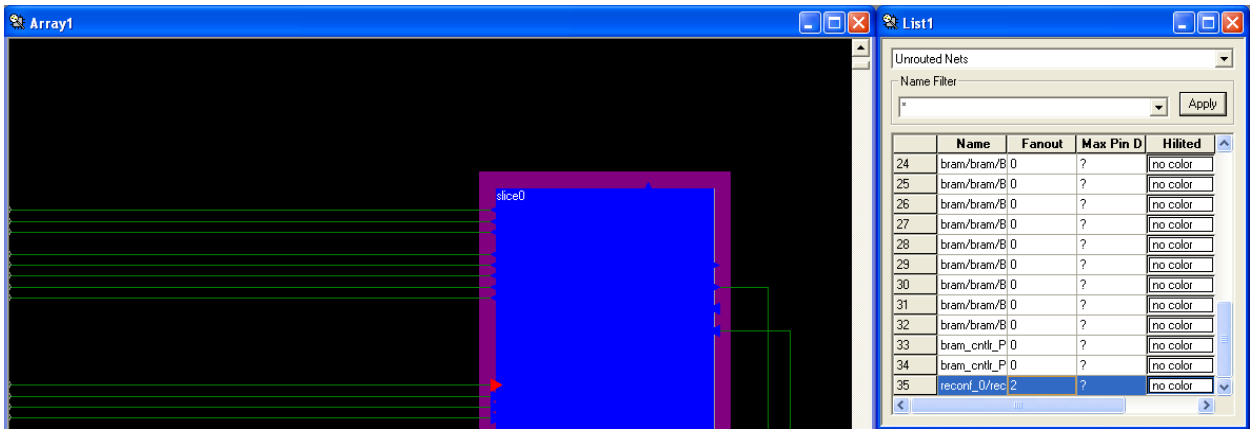

Figure 2.3.4 Add pin to slice

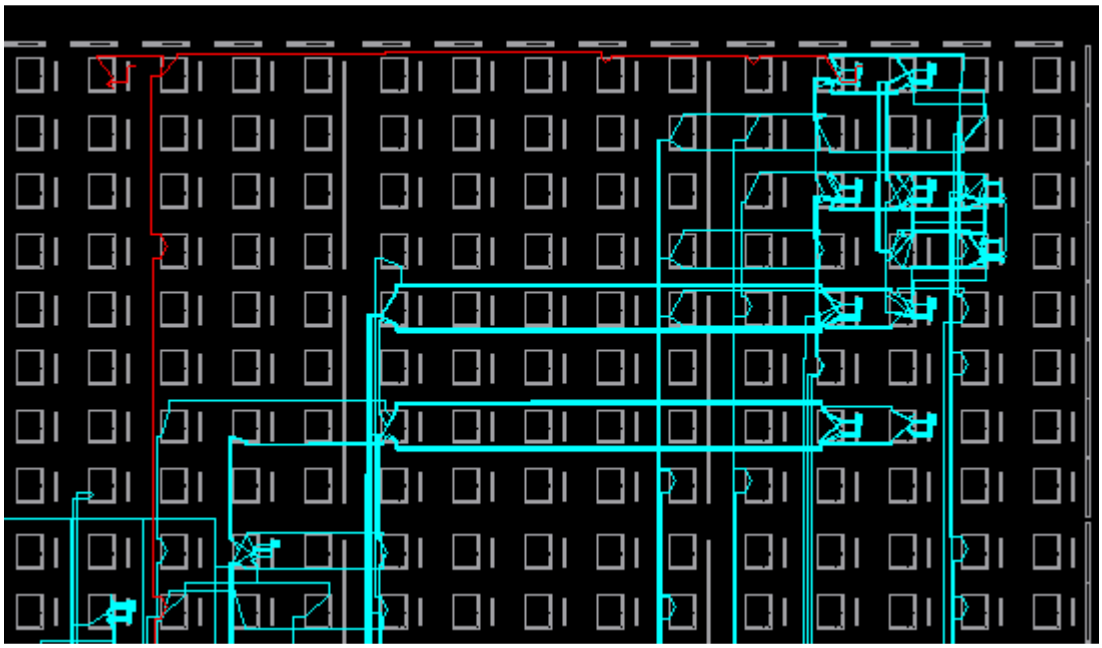
Figure 2.3.5 Forced autorouting

Last the branch to *slice0* has to be removed from the net as Xilinx tools regard this as invalid routing. Select only the added pin on *slice0* and the branch point as showed in figure 2.3.6 and press unroute. Delete the pin on *slice0* by right clicking it and press delete. If the net is still in the unrouted nets list , select the net and press autoroute. This result is showed in figure 2.3.7 where the net does not collide with the reconfigurable region.

.

A fully re-routed design, where all the wires red marked wires from figure 2.3.1 has been rerouted, is shown in figure 2.3.8.

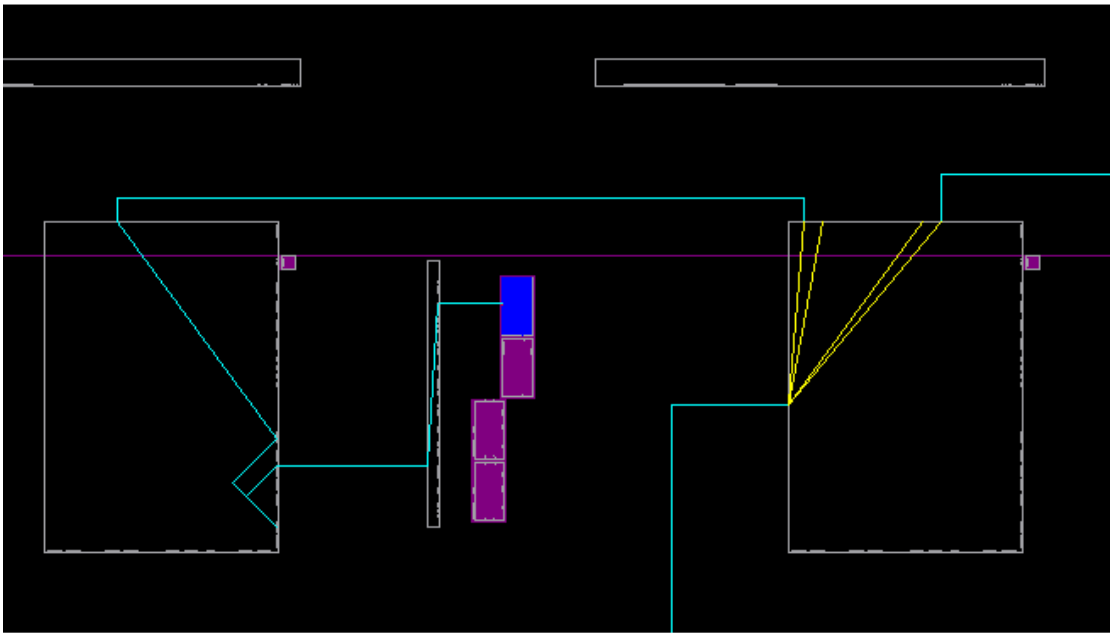

Figure 2.3.6 Unroute branch

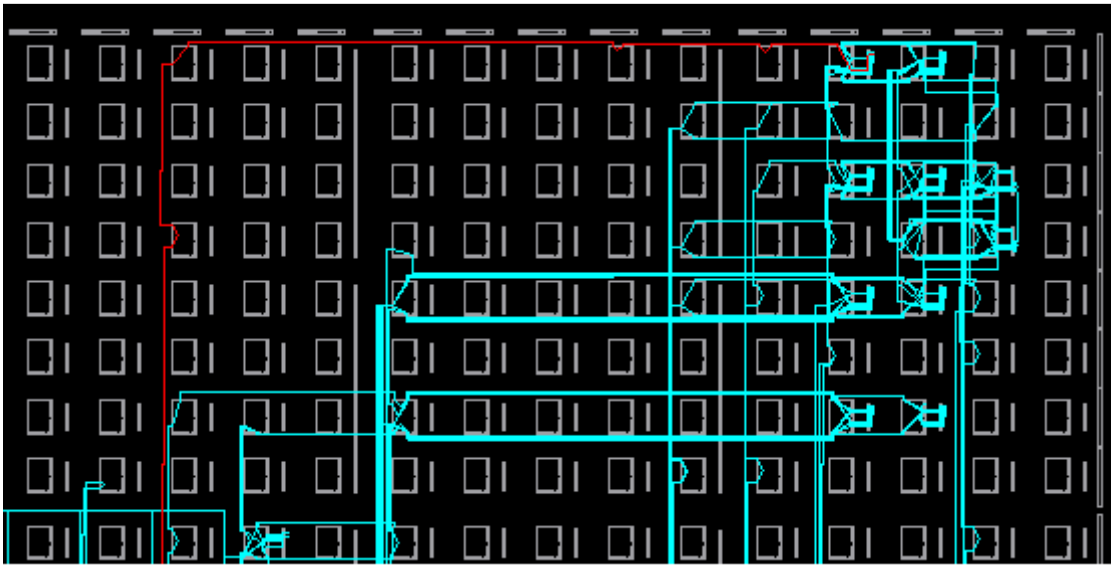

Figure 2.3.7 Complete rerouting

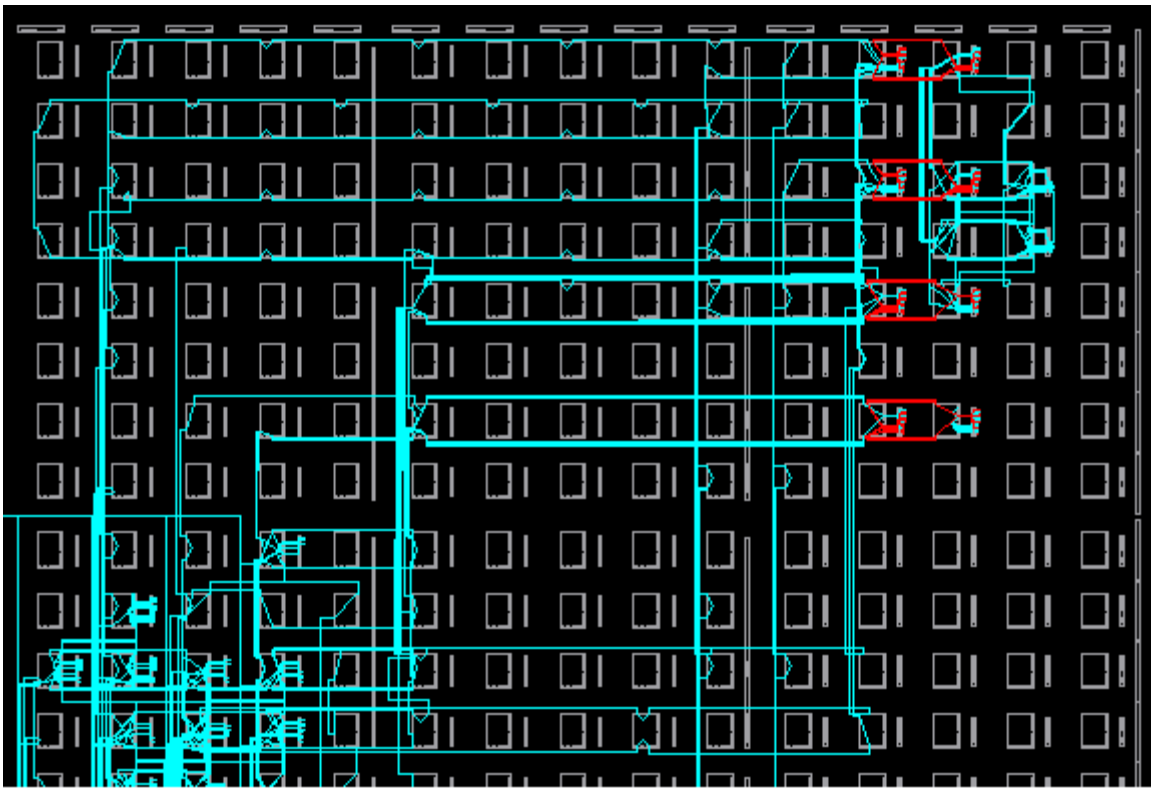

Figure 2.3.8 Fully rerouted design

2.4 Export the design to bitfile

After rerouting, create a bitfile from the NCD file by using bitgen. This can be done from command line/shell. An example is shown in source 2.4.1 where current working directory is the /implementation folder of the Platform Studio project. B*itgen.ut* , edited earlier in this tutorial, specifies bitfile creation parameters and *test.ncd* is the path to the source NCD file. This will produce a file called test.bit in the current working directory.

```
bitgen -w -f bitgen.ut  test.ncd
```

Source 2.4.1

After bitfile creation, the bitfile must be processed through bitinit. Bitinit incorporates initialization of processor instruction memory into a bitfile. If a bitfile that has not processed through bitinit is uploaded to a Suzaku board, hermit and Linux will not boot and the board will have to be re-flashed.

An example is shown in source 2.4.2 where the working directory is the Platform Studio project's root folder. This produces the file test_bitgen.bit based on test.bit.

```
bitinit ../xps_proj.mhs  -pe ppc405_system ppc405_system/code/executable.elf
-bt implementation/test.bit –o test_bitgen.bit
```

Source 2.4.2

2.5 Creating compatible reconfigurable modules

Reconfigurable modules for the design developed above can be created by using the same Platform Studio project as for the first design, but changing functionality of the reconfigurable module. For instance the reconfigurable module could perform subtraction as shown in source 2.5.1.

```
…
architecture arch of reconfigurable_module is
begin
        out_16(0 to 7) <= in_16(0 to 7) -  in_16(7 to 15);
        out_16(8 to 15) <= (others => '0');
end arch;
…
```

Source 2.5.1

After creating bitfiles from designs where the reconfigurable module has different functionalities, programs from the framework for self reconfigurable systems can be used to extract the partial configuration defining the reconfigurable modules and use them for run time reconfiguration of a Xilinx FPGA.

3 Final Words

This tutorial demonstrates the steps for creating a reconfigurable design for the framework for self reconfigurable systems and how to create compatible reconfigurable modules. By changing the types of bus macros used and Placement constraints in Planahead a wide variety of reconfigurable systems and modules can be created. Any comments or questions to the tutorial can be sent to vegarend@gmail.com.

4 References

[1] Sverre Hamren, Tutorial for adding an adder to the suzaku hardware, 2008
[2] Vegard Endresen, Using Xilinx Planahead to create physical constraints, 2009
[3] Sverre Hamren, Tutorial for creating a hard/bus macro to the spartan3 fpga, 2008
[4] Sverre Hamren, Framework for self reconfigurable system on a Xilinx FPGA, 2009