Active Partial Reconfiguration

Abhinav Golas Pawan Jain

30 July 2005

1 Aim

The aim of this project was to create a working model of an Active Partially reconfigurable system. An active partially reconfigurable system is one where we can reconfigure a part of the circuit with the fixed part unperturbed and working. For the implementation purposes we used a Xilinx Virtex IIP FPGA - 2VP30. The board has 4 dip=switches, 4 push buttons and 4 LEDs as on-board user interface. The board also contains support for a serial port, which we have used for communicating with the PC, which controls when the FPGA is reconfigured.

2 Circuit design

The design we have implemented is a very basic design, invloving taking inputs from dipswitches on the FPGA board The design consists of 2 main modules:

1. LatchIO : module to latch inputs, send to computation module, latch the result and return it, alongwith sending a status signal to the computer.

2. Reconfig : This is the computation module which can be configured as an adder or a subtractor(more options can be added), as per requirement.

2.1 The LatchIO module

This module is the interface of the design and thus is a fixed module.

Structurally, this module consists of:

1. Three 4-bit registers.

2. One rs232IO submodule which further consists of many submodules.

Behaviourally, this module performs the following functions:

1. It takes inputs from the dip switches, and latches them into 2 registers,each on pressing a push-button. These are sent to the module Reconfig for computation.

2. It gets the output from the module Reconfig, and latches it into another register, which is sent to the LED's on pressing a push-button.

3. The submodule rs232IO performs the function of communicating to the PC through the RS-232 serial port whenever the module Reconfig needs to be reconfigured. The importance of this additional module can be seen from the fact that it cannot be pre-determined when this reconfiguration may be required. So some means is required through which the hardware will demand reconfiguration. This is exactly what module rs232IO does. In our case just for the sake of demonstration, we send a reconfiguration request after every single computation. But this is absolutely modifiable. This request is sent in the form of a single byte of data which is sensed by a C program running on top of it all.

2.2 The Reconfig module

The Reconfig module is a very basic module which performs the computation on the two operands given by the user. In our design for the purpose of demonstration, we have two possible computation modules, an adder and a subtractor. Both designs have two 4-bit wide input ports to get the two input values, and one 4-bit wide output port to return the result.

3 Implementation details

This section contains information on the steps we took to implement the active partial reconfiguration.

3.1 Initial Budgeting phase

3.1.1 Basic requirements

The Initial Budgeting phase involves setting up of the top level design in which the modules will fit in. The basic requirements which we need to complete in this step are as follows:

1. Decide the modules and their names.
2. Decide the area on the FPGA, each module will take.
3. Create a basic top level design - implying that how each module will communicate (port maps) and any other top level logic which may be required.
4. Create a directory structure for the project. This is very essential as an incorrect structure can lead to very arbitrary errors.

The initial budgeting mode serves as a base for further operations and thus must be finalised before further development can take place. Once decided, the module constraints must be met, or it will result in errors. Keep in mind, that for each reconfigurable state you wish to achieve, you must create a separate top level design. Essentially it is like creating separate designs, except that they will have certain common modules.

3.1.2 Implementation considerations

While performing the Initial Budgeting, it is better to follow the following order:

Create the directory structure
It is essential that a proper directory structure be set up before design process begins. You can follow this structure which we followed.

- *HDL* folder for the HDL source files
- *src* folder for keeping the compiled netlists
- *ISE* folder for HDL compilation
- Top_i folders for the i^{th} top level design
- *Modules* folder for the files of each module
- *Assemble* and *Initial* folders in each Top_i folder for the Initial budgeting and Assemble phases
- Folders for each module and top level design in *ISE*
- Folders for each module in *Modules*

Creating the HDL files
The HDL files you create for the initial budgeting phase only need to contain the module component descriptions and the inter-module connections. This apart from the top-level logic that you wish to put into your design. However, try to keep the top level logic to a minimum. For the interconnections, you will need to route connections through bus macros for modules that will be reconfigured. This implies that any module that needs to be reconfigured, will have all its ports routed through a bus-macro. (For more details on bus macros, refer to Xilinx documents xapp290 and the Development guide on the Xilinx website. If you are using VHDL as your HDL, then you can use the VHDL files of our project as a guide.) However, if the modules which are

being connected are going to be fixed, then there is no need for a bus macro.

Compilation
For compiling the HDL files, you will need to create a separate project for each configuration layout. Each project will contain the HDL file, and its corresponding ucf. Each project must however, only contain the top level design for the project. All the modules whose design is not provided at this stage are assumed to be black boxes, whose design will be completed at a later stage. Also an effort must be made to keep top level logic to a minimum. While compiling it is up to you whether you wish to make use of the GUI or the command mode interface. However, it is our personal recommendation that the command line mode be used, as it tends to speed up the process. The commands below are given for command and GUI modes. Also these commands are given for the Xilinx XST VHDL compiler. The command mode is given in case you wish to run the compilation under batch mode from scripts. However, while using either, you will need to set the following options for :

- Setting Bus delimiters to () : This is required as bus macros are implemented using ()
- Setting Keep hierarchy to 'yes'
- Setting Add I/O buffers option to 'yes' : Note that this is only done in the Initial budgeting mode. For other phases, this option will be disabled
- Adding modular and initial commands to ensure compilation for Initial Budgeting mode : This is only required in the GUI mode if you plan to use Floorplanner or Pace to set area constraints, because in that case ISE will attempt to Translate the project on its own.

For GUI mode :
First select the HDL file which is your source file. Then you will see certain options for this file. These option ma be hidden by default. To display these options if not visible, go to Edit-¿Preferences-¿Processes tab and set the 'Property Display Level' to high.

- Select the properties for the Synthesize option. Under the Synthesis Options tab 'Bus Delimiter' must be set to '()'
- Under the same tab, set 'Keep Hierarchy' to 'yes'
- Under the Xilinx Specific Options tab, set 'Add I/O buffers' to 'yes'

- Select the properties for the Translate option. Under the 'Other Ngdbuild Command line Options' tab , add the text '-modular initial'

For the command mode :

- -bus_delimiter ()
- -keep_hierarchy YES
- -iobuf YES
- This is not required for compilation. This is required in the build step, where it will be given

For more details on writing the compilation scripts, please refer to our tips section.

Building
For building the design, we used the ngdbuild tool. Note, that for this step, you will need to have a compiled copy of the bus macro implementation. For using the ngdbuild tool, you need to specify the target FPGA, the command to specify Initial Budgeting Mode, i.e. '-modular initial' and the complete path of the compiled .ngc file and the .ucf file. The ngdbuild tool will build these files into a .ngd file which can be used for specifying area constraints usinf Floorplanner or Pace. However, if you do change the ucf file, it is recommended that you rerun the Initial budgeting mode. This completes the Initial Budgeting mode, and brngs us to the next step, i.e. the Active Module Implementation.

3.2 Active Module Implementation

3.3 Basic requirements

In the Active Module Implementation phase, we add each module to the design. At the end of this phase, you will have the partial bitfiles that are used to reconfigure the FPGA. Note, however, that these file can only change the configuration after a full bitfile, which is obtained after the Final Assembly phase , has been loaded. To complete this step, we will need to complete the following steps :

1. Create designs for each individual modules used in the design

2. Run certain commands to compile the module designs, and get partial bitfiles for each

This step is the final step in the design phase of the process. After this, the Final Assembly phase only involves putting together these modules based on the initial top level designs we made in the Initial Budgeting mode.

3.3.1 Implementation considerations

Note that this section onwards, we will only specify commands for command line execution. For GUI based working, the corresponding options can be easily figured out.

Creating the HDL files
In this phase, all the modules which were left as black boxes in the Initial budgeting phase need to be defined. Note, that the designs need to comply with the constraints you have set earlier in the ucf file. However, the designs made in this section, are no different from the usual designs you make. No extra code is required in the design files, other than that required for the design.

Compilation
For this phase, again you will need to setup different projects for each module. However, you will not need to include the ucf file in the project. During this phase, you will need to set similar options as in the Initial Budgeting phase. Bus delimiters will again be set to the same '()', however adding I/O buffers must be disabled. You may leave the 'Keep Hierarchy' option disabled as it defaults to a 'no' value, or you may explicitly define it to be so. After doing this for all modules, we are ready to build these modules.

Building
For the build phase you will need to run a series of commands for which you can refer to the active mode script given. At the end of this step, you will get partial bitfiles for each module, whether you can reconfigure with them or not. Also you will note that after this step, the pimcreate utility creates certain files for each module. These are the files which will be used to complete the top level designs.

3.4 Assembly phase

3.4.1 Basic requirements

To run this phase, you only need to ensure that the above phases have been completed successfully. Even if the Active Module Implementation did not complete successfully, you can still proceed with this step if the errors you received were in the partial bitstream generation. Why I am stating this, is that even while working, sometimes we did get segmentation faults in the bitgen tool. But since the pims were created successfully, we could still proceed and complete this step. Also, the partial bitstreams can be generated later as well. In this step, you do not need any new files, this step will complete you designs, and give you full bitstreams to configure the FPGA.

3.4.2 Implementation Considerations

For running this step, please refer to the scripts provided for your reference in this document. There is nothing else required to do for this.

4 Error log

5 Reference files

5.1 Scripts

5.1.1 Initial Budgeting phase toplevel script

```
echo ================================================================================
echo ==== Initial Stage ====
echo ================================================================================

cd ISE
#reconfig and reconfig1 are our 2 top level designs
#You may have as many as you like
mkdir reconfig
mkdir reconfig1
cd reconfig
#Compilation stage
#Calling compilation scripts for the 2 top level designs
xst -ifn ../../scripts/main.scr
cd ../reconfig1
xst -ifn ../../scripts/main1.scr
cd ../../
#Compiled sources are kept in this path for backup and use for further
stages
cp ISE/reconfig/main.ngc src
cp ISE/reconfig1/main.ngc src/main1.ngc
#Build stage begins
echo =====================
echo Running Initial Stage Top
echo =====================
cd Top/Initial
#Getting files for the build process
cp ../../src/main.ngc .
cp ../../ucf/main.ucf .
cp ../../bm_4b_v2p.nmc .
#Running the script for running build commands
#This script will be run separately for each top level design
./initial.cmd
cd ../..

echo =====================
```

```
echo Running Initial Stage Top1
echo ====================
cd Top1/Initial
cp ../../src/main1.ngc main.ngc
cp ../../ucf/main.ucf .
cp ../../bm_4b_v2p.nmc .
./initial.cmd
cd ../..
```

5.1.2 Command script for building each top level design (initial.cmd)

```
#rem ==== ngdbuild (Translate) ====
ngdbuild -p xc2vp30-ff896-5 -modular initial -uc main.ucf main.ngc
# Build Target Phase UCF Compiled design
```

5.1.3 Command script for compiling each top level design (main(1).scr)

The file main.prj contains the path of the vhdl file(s) to be compiled for the top level design `run`

```
-ifn ../../prj/main.prj
-ifmt VHDL
-opt_mode SPEED
-opt_level 1
-ofmt NGC
-ofn main.ngc
-p xc2vp30-ff896-5
-tristate2logic YES
-bus_delimiter ()
-iobuf YES
-keep_hierarchy YES
-ent main
```

For more details on what each option means, please refer to tool manuals and help.

5.1.4 Active Module Implementation phase top level script

```
echo ================================================================================
echo ==== Active Modules ====
```

```
echo ================================================================================
cd ISE
#Make directories for each module
mkdir latchio
mkdir adder
mkdir subt

cd latchio
#Compilation Stage
xst -ifn ../../scripts/latchio.scr
cd ../adder
xst -ifn ../../scripts/adder.scr
cd ../subt
xst -ifn ../../scripts/subt.scr
cd ../../

#Backing up compiled source
cp ISE/latchio/latchio.ngc src
cp ISE/adder/adder.ngc src
cp ISE/subt/subt.ngc src

#Building each module
echo ====================
echo Running Active module latchio
echo ====================
cd Modules/latchio
cp ../../src/latchio.ngc .
cp ../../ucf/main.ucf .
#Bitgen options are stored in this file
cp ../../bitgen_v2_jtag.ut .
#Build script for building each module
./active.cmd

echo ====================
echo Running Active module adder
echo ====================
cd ../adder
cp ../../src/adder.ngc .
cp ../../ucf/main.ucf .
cp ../../bitgen_v2_jtag.ut .
./active.cmd
```

```
echo ====================
echo Running Active module subt
echo ====================
cd ../subt
cp ../../src/subt.ngc .
cp ../../ucf/main.ucf .
cp ../../bitgen_v2_jtag.ut .
./active.cmd
```

5.1.5 Script for compiling each module

```
run
-ifn ../../prj/adder.prj
-ifmt VHDL
-opt_mode SPEED
-opt_level 1
-ofmt NGC
-ofn adder.ngc
-p xc2vp30-ff896-5
-iobuf NO
-ent adder
```

5.1.6 Module building script

```
#rem ==== ngdbuild (Translate) ====
ngdbuild -p xc2vp30-ff896-5 -modular module -active adder ../../Top/Initial/main.ng

#rem ==== map ====
map -pr b main.ngd -o main_map.ncd main.pcf

#rem ==== par (Placement & Routing) ====
par -w main_map.ncd main.ncd main.pcf

#rem ==== bitgen ====
bitgen -d -f bitgen_v2_jtag.ut -g ActiveReconfig:yes main.ncd
trce main.ncd main.pcf
pimcreate -ncd main.ncd -ngm main_map.ngm ../../Pims
```

5.1.7 Bitgen options file (bitgen_v2_jtag.ut)

```
-w
-l
-m
-g ReadBack
-g DebugBitstream:No
-g CRC:Enable
-g ConfigRate:4
-g CclkPin:PullUp
-g M0Pin:PullUp
-g M1Pin:PullUp
-g M2Pin:PullUp
-g ProgPin:PullUp
-g DonePin:PullUp
-g DriveDone:No
-g PowerdownPin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullNone
-g TmsPin:PullUp
-g UnusedPin:PullUp
-g UserID:0xFFFFFFFF
-g DCMShutDown:Disable
-g DisableBandgap:No
-g StartUpClk:JtagClk
-g DONE_cycle:4
-g GTS_cycle:5
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Match_cycle:NoWait
-g Security:None
-g Persist:No
-g DonePipe:No
-g Encrypt:No
```

5.1.8 Assemble phase top level script

```
echo ================================================================================
echo ==== Assemble Stage ====
```

```
echo ================================================================================

echo ======================
echo Running Assemble Stage Top
echo ======================
cd Top/Assemble
cp ../../src/main.ngc .
cp ../../ucf/main.ucf .
cp ../../bm_4b_v2p.nmc .
cp ../../bitgen_v2_jtag.ut .
./assemble.cmd

echo ======================
echo Running Assemble Stage Top1
echo ======================
cd ../../Top1/Assemble
cp ../../src/main1.ngc main.ngc
cp ../../ucf/main.ucf .
cp ../../bm_4b_v2p.nmc .
cp ../../bitgen_v2_jtag.ut .
./assemble.cmd
```

5.1.9 Assembly script for each top level design

```
ngdbuild -p xc2vp30-ff896-5 -modular assemble -pimpath ../../Pims
main.ngc
map -pr b main.ngd -o main_map.ncd main.pcf
par -w main_map.ncd main.ncd main.pcf -rl high
bitgen -f bitgen_v2_jtag.ut main.ncd
trce main.ncd main.pcf
```

5.2 The UCF file

```
#SWITCHES
NET "SW_0" LOC = "AC11";
NET "SW_1" LOC = "AD11";
NET "SW_2" LOC = "AF8";
NET "SW_3" LOC = "AF9";
```

```
NET "SW_0" IOSTANDARD = LVCMOS25;
NET "SW_1" IOSTANDARD = LVCMOS25;
NET "SW_2" IOSTANDARD = LVCMOS25;
NET "SW_3" IOSTANDARD = LVCMOS25;

#LEDs
NET "LED_0" LOC = "AC4";
NET "LED_1" LOC = "AC3";
NET "LED_2" LOC = "AA6";
NET "LED_3" LOC = "AA5";

NET "LED_0" IOSTANDARD = LVTTL;
NET "LED_1" IOSTANDARD = LVTTL;
NET "LED_2" IOSTANDARD = LVTTL;
NET "LED_3" IOSTANDARD = LVTTL;

NET "LED_0" DRIVE = 12;
NET "LED_1" DRIVE = 12;
NET "LED_2" DRIVE = 12;
NET "LED_3" DRIVE = 12;

NET "LED_0" SLEW = SLOW;
NET "LED_1" SLEW = SLOW;
NET "LED_2" SLEW = SLOW;
NET "LED_3" SLEW = SLOW;

#PUSH BUTTONS
NET "PB_UP" LOC = "AH4";
NET "PB_ENTER" LOC = "AG5";
NET "PB_RIGHT" LOC = "AH2";

NET "PB_UP" IOSTANDARD = LVTTL;
NET "PB_ENTER" IOSTANDARD = LVTTL;
NET "PB_RIGHT" IOSTANDARD = LVTTL;

# for RS232 :::
# master of system clock
NET "CLK" LOC = "AJ15";
NET "CLK" IOSTANDARD = LVCMOS25;
NET "CLK" TNM_NET = "CLK";
TIMESPEC "TS_SYSTEM_CLOCK" = PERIOD "CLK" 10.00 ns HIGH 50 %;
```

```
# RS232 connector
NET "RS232_TX_DATA" LOC = "AE7";
NET "RS232_TX_DATA" IOSTANDARD = LVCMOS25;
NET "RS232_TX_DATA" DRIVE = 8;
NET "RS232_TX_DATA" SLEW = SLOW;
NET "PB_DOWN" LOC = "AG3";
NET "PB_DOWN" IOSTANDARD = LVTTL;


# Beginning of reconfigurable constraints
# Bus macros
INST "busmacro1" LOC = "TBUF_X16Y4" ;
INST "busmacro2" LOC = "TBUF_X16Y12" ;
INST "busmacro3" LOC = "TBUF_X16Y20" ;

AREA_GROUP "AG_fix" RANGE = SLICE_X20Y159:SLICE_X91Y0 ;
AREA_GROUP "AG_fix" RANGE = TBUF_X20Y159:TBUF_X90Y0 ;
AREA_GROUP "AG_fix" MODE = RECONFIG ;
AREA_GROUP "AG_fix" GROUP = CLOSED ;
AREA_GROUP "AG_fix" PLACE = CLOSED ;
INST "take" AREA_GROUP = "AG_fix" ;

AREA_GROUP "AG_reco_module" RANGE = SLICE_X0Y159:SLICE_X19Y0 ;
AREA_GROUP "AG_reco_module" RANGE = TBUF_X0Y159:TBUF_X18Y0 ;
AREA_GROUP "AG_reco_module" MODE = RECONFIG ;
AREA_GROUP "AG_reco_module" GROUP = CLOSED ;
AREA_GROUP "AG_reco_module" PLACE = CLOSED ;
INST "reconfig" AREA_GROUP = "AG_reco_module" ;

INST "Internal_Gnd_Mux" AREA_GROUP = "AG_fix" ;
INST "Internal_Vcc_Mux" AREA_GROUP = "AG_fix" ;
INST "Internal_Gnd_Fix" AREA_GROUP = "AG_fix" ;
INST "Internal_Vcc_Fix" AREA_GROUP = "AG_fix" ;
INST "Internal_Gnd_Reco" AREA_GROUP = "AG_reco_module" ;
INST "Internal_Vcc_Reco" AREA_GROUP = "AG_reco_module" ;

INST Internal_Gnd_Mux LOCK_PINS;
INST Internal_Gnd_Fix LOCK_PINS;
INST Internal_Gnd_Reco LOCK_PINS;
```

```
INST Internal_Vcc_Mux LOCK_PINS;
INST Internal_Vcc_Fix LOCK_PINS;
INST Internal_Vcc_Reco LOCK_PINS;
```

5.3 VHDL source

5.3.1 Main top level design(main.vhd/main1.vhd

```
--
-- main.vhd - Simple Adder
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_misc.all;

library UNISIM;
use UNISIM.VComponents.all;

library work;

entity main is
port(
SW_0:  in std_logic;
SW_1:  in std_logic;
SW_2:  in std_logic;
SW_3:  in std_logic;

LED_0:  out std_logic;
LED_1:  out std_logic;
LED_2:  out std_logic;
LED_3:  out std_logic;

PB_UP: in std_logic ;
PB_RIGHT: in std_logic ;
PB_ENTER: in std_logic ;

CLK: in std_logic ;
RS232_RX_DATA: in std_logic ;
```

```
RS232_TX_DATA: out std_logic ;

PB_DOWN: in std_logic
);

end main;

architecture mixed of main is

signal clock :  std_logic;
signal RESET : std_logic ;

signal oper1 :  std_logic_vector(3 downto 0) ;
signal oper2 :  std_logic_vector(3 downto 0) ;
signal oper_rec1 :  std_logic_vector(3 downto 0) ;
signal oper_rec2 :  std_logic_vector(3 downto 0) ;
signal ans :  std_logic_vector(3 downto 0) ;
signal ans_rec :  std_logic_vector(3 downto 0) ;

-- signal count_oper_main :  std_logic_vector(3 downto 0) ;

-- signal load_new_bit_main :  std_logic ;
-- signal status_main :  std_logic_vector(7 downto 0) ;
-- signal status_main_tmp :  std_logic_vector(7 downto 0) ;
signal FFake_Gnd, FFake_Vcc, RFake_Vcc,RFake_Gnd, FMux_Gnd, FMux_Vcc
:  std_logic;

component latchio is
port(
SW_0:  in std_logic;
SW_1:  in std_logic;
SW_2:  in std_logic;
SW_3:  in std_logic;

out1:  out std_logic_vector(3 downto 0);
out2:  out std_logic_vector(3 downto 0);
res_out:  out std_logic_vector(3 downto 0);
res_in:  in std_logic_vector(3 downto 0);
RESET: in std_logic;

PB_UP: in std_logic ;
```

```
PB_RIGHT: in std_logic ;
PB_ENTER: in std_logic ;

-- Pins for rs232IO
td_pin:  out std_logic;
rd_pin:  in std_logic;
clk_pin:  in std_logic
);
end component ;

component adder is
port(
in1:  in std_logic_vector(3 downto 0) ;
in2:  in std_logic_vector(3 downto 0) ;
ans:  out std_logic_vector(3 downto 0)
);
end component ;
component bm_4b_v2p
port (LI : in std_logic_vector (3 downto 0);
LT : in std_logic_vector (3 downto 0);
RI : in std_logic_vector (3 downto 0);
RT : in std_logic_vector (3 downto 0);
O : out std_logic_vector (3 downto 0));
end component;

component LUT1
generic (INIT : bit_vector(1 downto 0) := b"01");
port (O : out std_logic;
I0 :  in std_logic);
end component;

begin

RESET <= PB_DOWN ;
clock <= CLK;


-------------------------
-- CLOCK DISTRIBUTION: --
-------------------------
```

```
-- Fake Gnd and Fake Vcc
Internal_Gnd_Mux:  LUT1
generic map (INIT => b"00")
port map (O => FMux_Gnd, I0 => FMux_Vcc);

Internal_Vcc_Mux:  LUT1
generic map (INIT => b"11")
port map (O => FMux_Vcc, I0 => FMux_Gnd);


--------------------
-- FIXED MODULE: --
--------------------

take :  latchio port map
(
SW_0 => SW_0,
SW_1 => SW_1,
SW_2 => SW_2,
SW_3 => SW_3,

out1 => oper1,
out2 => oper2,
res_out(3)=> LED_3,
res_out(2)=> LED_2,
res_out(1)=> LED_1,
res_out(0)=> LED_0,

res_in => ans,
RESET => RESET,

PB_UP => PB_UP,
PB_RIGHT => PB_RIGHT,
PB_ENTER => PB_ENTER,

td_pin => RS232_TX_DATA,
rd_pin => RS232_RX_DATA,
clk_pin => clock
) ;

-- Fake Gnd and Fake Vcc
```

```
Internal_Gnd_Fix:  LUT1
generic map (INIT => b"00")
port map (O => FFake_Gnd, I0 => FFake_Vcc);

Internal_Vcc_Fix:  LUT1
generic map (INIT => b"11")
port map (O => FFake_Vcc, I0 => FFake_Gnd);


----------------------------
-- RECONFIGURABLE MODULE: --
----------------------------

reconfig :  adder port map
(
in1 => oper_rec1,
in2 => oper_rec2,
ans => ans_rec
) ;


-- Fake Gnd and Fake Vcc
Internal_Gnd_Reco:  LUT1
generic map (INIT => b"00")
port map (O => RFake_Gnd, I0 => RFake_Vcc);

Internal_Vcc_Reco:  LUT1
generic map (INIT => b"11")
port map (O => RFake_Vcc, I0 => RFake_Gnd);

-----------------------------------------------
-- BUS MACRO FOR INTER-MODULE COMMUNICATION: --
-----------------------------------------------
--Bus macro for sending output from the adder module to latchio

busmacro1:  bm_4b_v2p -- LEFT side:  reconfigurable, right side:
fixed
port map (
LI(3) => ans_rec(3),
LI(2) => ans_rec(2),
LI(1) => ans_rec(1),
```

```
LI(0) => ans_rec(0),

-- LT(3) => RFake_Gnd, -- "enable" the left side...
LT(2) => RFake_Gnd, -- "enable" the left side...
LT(1) => RFake_Gnd, -- "enable" the left side...
LT(0) => RFake_Gnd, -- "enable" the left side...
-- ----------------------
RI(3) => FFake_Gnd, -- dummy data
RI(2) => FFake_Gnd, -- dummy data
RI(1) => FFake_Gnd, -- dummy data
RI(0) => FFake_Gnd, -- dummy data
--
RT(3) => FFake_Vcc, -- tri-state the right side...
RT(2) => FFake_Vcc, -- tri-state the right side...
RT(1) => FFake_Vcc, -- tri-state the right side...
RT(0) => FFake_Vcc, -- tri-state the right side...
-- ----------------------
O(3) => ans(3),
O(2) => ans(2),
O(1) => ans(1),
O(0) => ans(0)
);


-- Bus macro for sending inputs from latchio to adder module

busmacro2:  bm_4b_v2p -- LEFT side:  reconfigurable, right side:
fixed
port map (
LI(3) => RFake_Gnd,-- dummy data
LI(2) => RFake_Gnd,-- dummy data
LI(1) => RFake_Gnd,-- dummy data
LI(0) => RFake_Gnd,-- dummy data
--
LT(3) => RFake_Vcc, -- tri-state the left side...
LT(2) => RFake_Vcc, -- tri-state the left side...
LT(1) => RFake_Vcc, -- tri-state the left side...
LT(0) => RFake_Vcc, -- tri-state the left side...
-- ----------------------
RI(3) => oper1(3),
RI(2) => oper1(2),
```

```
RI(1) => oper1(1),
RI(0) => oper1(0),
--
RT(3) => FFake_Gnd, -- "enable" the right side...
RT(2) => FFake_Gnd, -- "enable" the right side...
RT(1) => FFake_Gnd, -- "enable" the right side...
RT(0) => FFake_Gnd, -- "enable" the right side...
-- ----------------------
O(3) => oper_rec1(3),
O(2) => oper_rec1(2),
O(1) => oper_rec1(1),
O(0) => oper_rec1(0)
);

-- Bus macro for sending input from latchio to adder module

busmacro3:  bm_4b_v2p -- LEFT side:  reconfigurable, right side:
fixed
port map (
LI(3) => RFake_Gnd,-- dummy data
LI(2) => RFake_Gnd,-- dummy data
LI(1) => RFake_Gnd,-- dummy data
LI(0) => RFake_Gnd,-- dummy data
--
LT(3) => RFake_Vcc, -- tri-state the left side...
LT(2) => RFake_Vcc, -- tri-state the left side...
LT(1) => RFake_Vcc, -- tri-state the left side...
LT(0) => RFake_Vcc, -- tri-state the left side...
-- ----------------------
RI(3) => oper2(3),
RI(2) => oper2(2),
RI(1) => oper2(1),
RI(0) => oper2(0),
--
RT(3) => FFake_Gnd, -- "enable" the right side...
RT(2) => FFake_Gnd, -- "enable" the right side...
RT(1) => FFake_Gnd, -- "enable" the right side...
RT(0) => FFake_Gnd, -- "enable" the right side...
-- ----------------------
O(3) => oper_rec2(3),
O(2) => oper_rec2(2),
```

```
O(1) => oper_rec2(1),
O(0) => oper_rec2(0)
);

end mixed ;
```

5.3.2 LatchIO module

```
--
-- latchio.vhd - Latch I/Os and send signals to serial port
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_misc.all;

library work;

entity latchio is
port(
SW_0:  in std_logic;
SW_1:  in std_logic;
SW_2:  in std_logic;
SW_3:  in std_logic;

out1:  out std_logic_vector(3 downto 0);
out2:  out std_logic_vector(3 downto 0);
res_out:  out std_logic_vector(3 downto 0);
res_in:  in std_logic_vector(3 downto 0);

PB_UP: in std_logic ;
PB_RIGHT: in std_logic ;
PB_ENTER: in std_logic ;
RESET: in std_logic ;

-- Pins for rs232IO
td_pin:  out std_logic;
rd_pin:  in std_logic;
```

```
clk_pin:  in std_logic
);
end latchio;

architecture behav of latchio is

component rs232IO is
port (
pin_sysclk:  in std_logic;
send_data :  in std_logic;
statusdata:  in std_logic_vector(7 downto 0);
reset_pushbtn:  in std_logic;
pin_rs232_rd:  in std_logic;
pin_rs232_td:  out std_logic
);
end component;

signal load_new_bit:  std_logic;
signal status:  std_logic_vector(7 downto 0);

begin

COMMUNICATOR: rs232IO port map
(
pin_sysclk => clk_pin,
send_data => load_new_bit,
statusdata => status,
reset_pushbtn => RESET,
pin_rs232_rd => rd_pin,
pin_rs232_td => td_pin
);

process(PB_UP)
begin
if(PB_UP'event and PB_UP='1')
then
res_out<=res_in;
end if ;
end process ;

process(PB_RIGHT)
```

```
begin
if(PB_RIGHT'event and PB_RIGHT='1')
then
out1(3) <= SW_3 ;
out1(2) <= SW_2 ;
out1(1) <= SW_1 ;
out1(0) <= SW_0 ;
end if ;
end process ;

load_new_bit <= '1' when PB_UP = '1' else '0' ;
status <= "01010101" when PB_UP = '1' else "00000000" ;

process(PB_ENTER)
begin

if(PB_ENTER'event and PB_ENTER='1')
then
out2(3) <= SW_3 ;
out2(2) <= SW_2 ;
out2(1) <= SW_1 ;
out2(0) <= SW_0 ;

end if ;
end process ;
end behav ;
```

5.3.3 Adder source

```
--
-- adder.vhd - 4 bit simple adder
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_misc.all;

library work;
```

```
entity adder is
port(
in1:  in std_logic_vector(3 downto 0);
in2:  in std_logic_vector(3 downto 0);
ans:  out std_logic_vector(3 downto 0)
);
end adder;

architecture behav of adder is
begin

ans <= in1 + in2 ;

end behav ;
```

6 Further Work

Further work may involve:

1. Enable taking inputs from the serial port itself so that the whole process can be made computerised and optimal performance can be extracted from this feature

2. Dynamic C++ Parsing and using partial reconfiguration to speed up computation by loading program specific computation units on demand

3. Attempt reconfiguration from the FPGA itself, using the PowerPC processors onboard, to enable development of Hardware based Learning systems