HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.co.uk
www.hunteng.co.uk
www.hunt-dsp.com

Getting Started with HERON modules that have FPGAs

v2.4 P.Warnes 09-05-05

The HERON-FPGA and HERON-IO families are ranges of HERON modules with FPGAs, often combined with some interface capability. They may be used as processing elements, as interface elements, or as some combination of the two. They can implement simple, high-speed tasks very efficiently, and so are ideal for fast DSP processing – such as in wireless, RADAR or imaging; but they can also implement interface elements such as UARTS (RS232 etc), PWM generators or digital camera interfaces.

HUNT ENGINEERING provides a Hardware Interface Layer written in VHDL, allowing developers to focus on the application-specific parts of the system. All of the module hardware should be accessed using parts from the library.

This tutorial describes an easy approach to starting with the HERON FPGA modules. It is intended to be a generic tutorial, so does not address any I/O facilities.

History

Rev 1.0	first written
Rev 1.1	changed to use zip files for examples
Rev 2.0	modified to reflect the VHDL/ISE 4.1 support modified to include HEART carriers and HEPC8
Rev 2.1	modified to reflect 6 FIFO support
Rev 2.2	some spelling mistakes, clarify the steps to draw attention to the purpose of the tutorial
Rev 2.3	added section for users of ISE 6
Rev 2.4	updated to reference new document 'Using Different Versions of ISE'

This Tutorial

The purpose of this tutorial is to familiarise you with the tools you will use for loading your FPGA module and building a new FPGA design.

There is a program provided to work on your Host computer with this tutorial. This allows you to test your FPGA design. Changing this program is not discussed in this tutorial. To learn about Host programs please follow the HOST-API examples. When you have done that you can use this host program as an example of the interaction between an FPGA module design and a Host program.

There is also a DSP program that demonstrates the same thing, but using a DSP. You can use this to learn about the interaction between an FPGA design and a C6000 program. To be able to do this you must first have worked through the tutorials about making a C6000 program.

So you should follow this tutorial to:

1. Load a bitstream from the HUNT ENGINEERING into your FPGA.
2. Run the Host example program to see the example1 working.
3. Copy the Example1 FPGA project for your module and re-build it
4. See the re-built bitstream still performs the same as the one supplied on the HUNT ENGINEERING CD.
5. Change the FPGA project to invert the data
6. Re-run the example with the new bitstream and see that the inversion is taking place.

Now you will be ready to move on to one of the other examples for your module. Choose one that is closest to what you are trying to do in your own system, and after following the standard example, modify it to meet your needs.

Concepts

FPGA devices are programmable hardware, which can be configured to meet the needs of your system. As with a microprocessor, the function of the FPGA is controlled entirely by the program – and by the peripherals the FPGA is connected to.

A HERON module has access to FIFOs for communicating with other modules in the system – there may be up to 6 input FIFOs and 6 output FIFOs. It may have additional interfaces, such as ADCs, or level-shifting buffers, allowing it to interface to the real world.

The Heron-FPGA family use Xilinx reconfigurable logic devices. These can be programmed at low level, but can also be programmed using high level blocks, such as FIR filters, FFT transforms and so on. This process is covered in a separate paper.

Getting Started

Every HERON-FPGA or HERON-IO module is provided with a Hardware Interface Layer that makes it simple for a users FPGA design to access the hardware. Every module has an “example1” which is supplied as a project for ISE. It contains the top level design, the user constraints that define the pinning of the FPGA, and the timing of the clocks, and the simple example is implemented in a user_ap file – the one you will modify to make your own FPGA design.

The first part of our tutorial is to use example1 to check your understanding of the system. You can also use it as a “sanity check” later, to provide confidence in your hardware.

The example is very simple – reading data from a single input FIFO and writing it back to an output FIFO. This loop-back test (if successful) proves that a) we can program the FPGA and b) the FPGA module is basically functional. It also demonstrates the use of HSB to configure hardware within the FPGA.

This tutorial assumes that you are using the latest version of ISE from Xilinx. For users who have different versions of ISE please follow the application note ‘Using Different Versions of ISE’ while working through the Getting Started example. Alternatively, if you are using a different synthesis tool to ISE please refer to the application note “Using non ISE development tools".

Running the Tutorial

Follow this sequence:

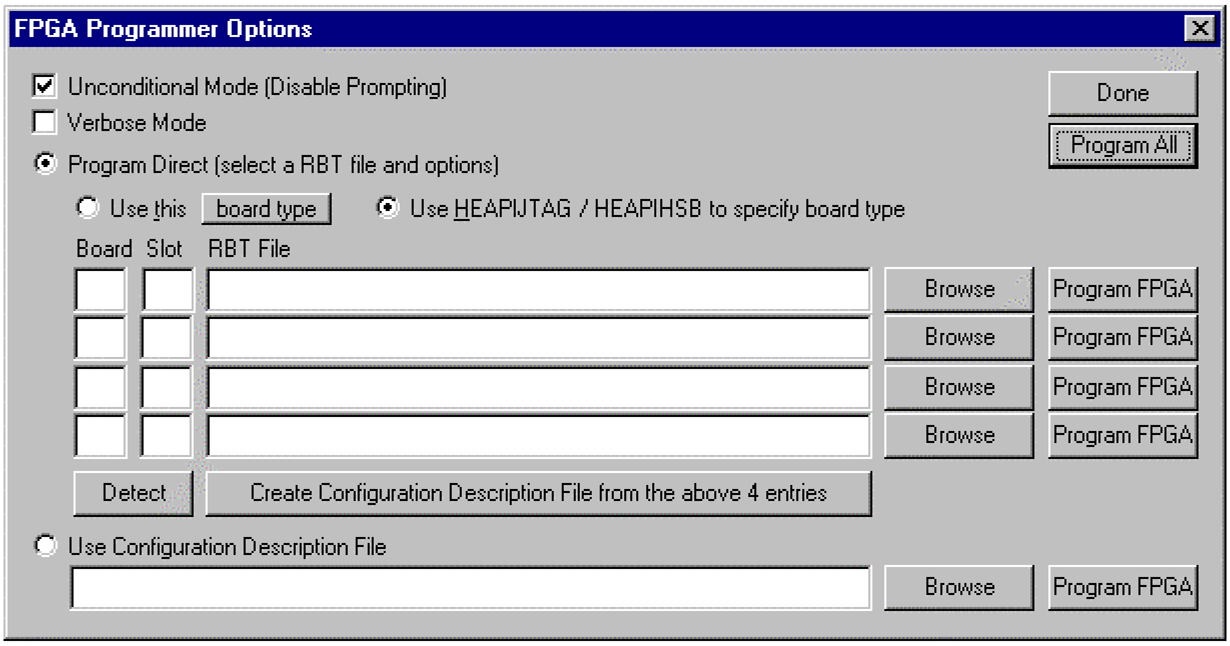
1) Select "Getting Started" from the Hunt Engineering CD menu, then "to start using FPGA modules and tools …." and then "Starting your FPGA development". This should be how you reached this document.

   From the list of examples at the bottom of the screen click "General FPGA Examples". Windows Explorer will open in \fpga\generic_examples directory of the CD. At the same level there are directories for each module type. Select the correct one for your module (e.g. for HERON-FPGA1 version1 it is fpga1v1), here you will see a directory tree that has the directories \example1 and \common. These are the files for your example1 project.

   There is also a zip file, which is a zip of the entire tree for this module type. This is provided because restoring files from this zip file will restore files that are not set to read only. If you simply copy the files from the CD you will need to set the files in the example1 directory so that they do not have the read only attribute set. In fact the files in common should have this attribute set.

2) The next step is to program the FPGA by downloading the example bitstream. You will need to do this whenever the fpga program has been changed, or when the PC has been powered off.

   Select START → PROGRAMS → HUNT ENGINEERING → Program HERON-FPGA you should see:-


FPGA Programmer Options

[✓] Unconditional Mode (Disable Prompting)
[ ] Verbose Mode
● Program Direct (select a RBT file and options)
○ Use this board type ● Use HEAPIJTAG / HEAPIHSB to specify board type
Board Slot RBT File
○ Use Configuration Description File
Done
Program All
Browse
Program FPGA
Detect
Create Configuration Description File from the above 4 entries

Use the Detect button to find your FPGA module, and the Board and Slot should get filled in for you. Then use the browse button to select the bitstream to use. You need to go to the directory of the CD that we were just in, and enter the "example1" directory. Here there may be several bitstream files. The name shows the number of gates and the package, e.g. 2v1000fg456.hcb is for a Virtex II 1Mgate FPGA in an FG456 package. This means that you need to know the type of FPGA on your module and select the right bitstream for your module type. The bitstream files might be a .rbt file which is the direct output from the Xilinx tools, or may be a .hcb file which is a HUNT Compressed Bitstream file. You can choose either file type to load with this tool.

If you have an HEPC8 module carrier you must choose the file with _pc8 on the end. For all other module carrier boards you must use the one without the _pc8.

Load the standard bit stream

Now select “Program FPGA” or “Program All” to download the bitstream. The configuration you have set up will be remembered for the next time so you do not have to go through the browsing process every time.

Actually the program that does the downloading is an executable that will have been installed in your %HEAPI_DIR%\utils directory and is called hrn_fpga.exe. This can be called directly from your own program, or perhaps from your autoexec.bat file. The program takes all of the options as command line parameters so that you will not have to answer prompts. Use hrn_fpga -h to see the options.

A useful indicator is the “DONE” LED fitted to HERON modules that have an FPGA. It will be on after power-up or during a reconfiguration, and will only go out if the FPGA is programmed correctly. Check this to ensure the FPGA is programming correctly.

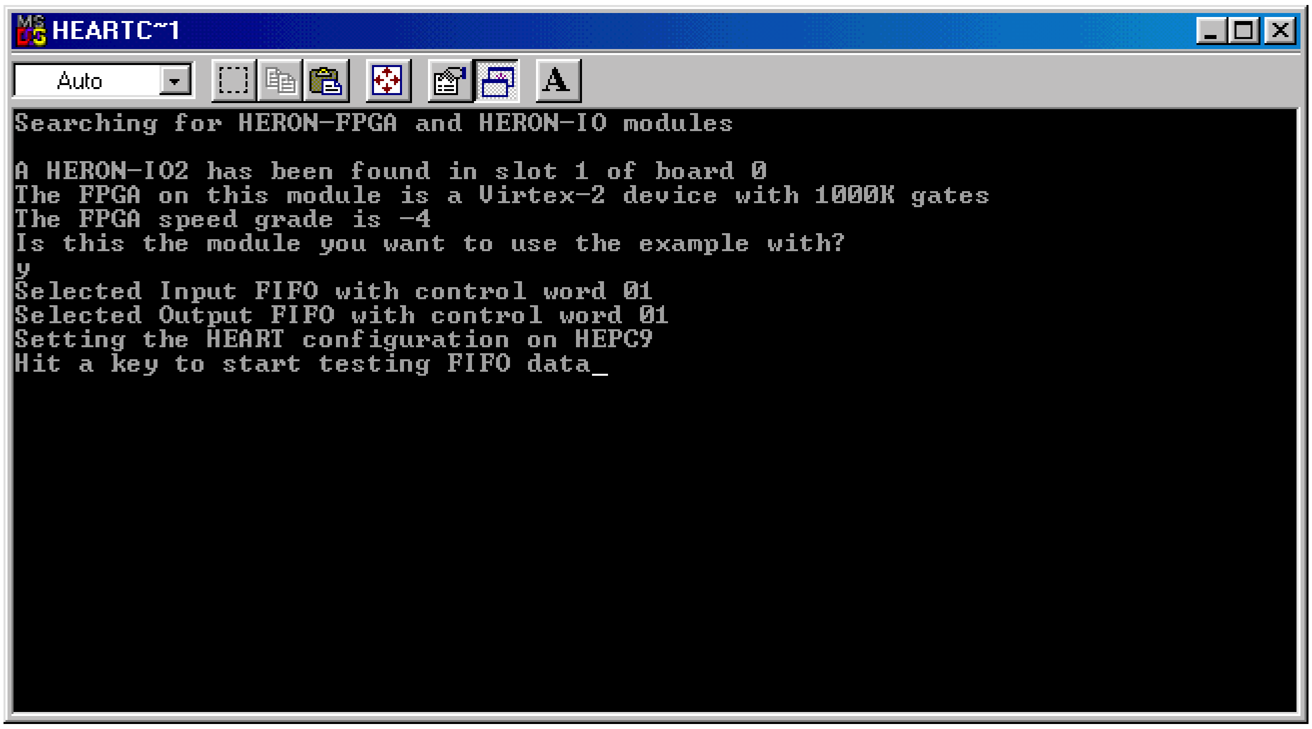
3) Now you need to understand how your system is set up. Run the Host example to verify that your FPGA module is loaded and working. If you have a DSP module in your system, you can then move onto using the C6000 example code to send and receive data to and from the FPGA module. This requires the FPGA module to be in slot 1 if you have an HEPC8, but it can be any slot of an HEPC9.

a) <u>Host example</u>

This can be run directly from the CD. In your explorer window (from stage1) change to the \fpga\getting started\hostex1 directory. Double click on the .exe file there.

HEPC9

For an HEPC9 you can use HEART to connect to any module slot from the Host PC. In this case when you run the hostex1.exe it will search for the FPGA modules in your system. If you have more than one, you can choose the correct one by answering no to the questions “is this the module you want to use for this example?” until the module you want to use is listed. i.e. you should see:-



```
Searching for HERON-FPGA and HERON-IO modules

A HERON-IO2 has been found in slot 1 of board 0
The FPGA on this module is a Virtex-2 device with 1000K gates
The FPGA speed grade is -4
Is this the module you want to use the example with?
y
Selected Input FIFO with control word 01
Selected Output FIFO with control word 01
Setting the HEART configuration on HEPC9
Hit a key to start testing FIFO data_
```

Once it has identified the module to use, the HEART will be set up to connect one Host FIFO to each slot. Then it uses the correct Host FIFO device to access the FPGA module you have chosen.

In the case of our example we have connected FIFO#0 of the module back to the Host PC, but you can change that in the network file if you want. Then the program will need to be re-built to set the correct FIFO number for the FPGA to use (via HSB).

HEPC8

If you have an HEPC8 the FPGA module must be in slot 1 in order to run this example. Then the module will use FIFO #1 to communicate with the PC .

The program will send some messages over the HSB to set registers in the FPGA example1 that "connect" to the FIFOs you have selected. Then it will send and receive data through the HERON FIFOs and test that the FPGA module returns the same value as was sent.

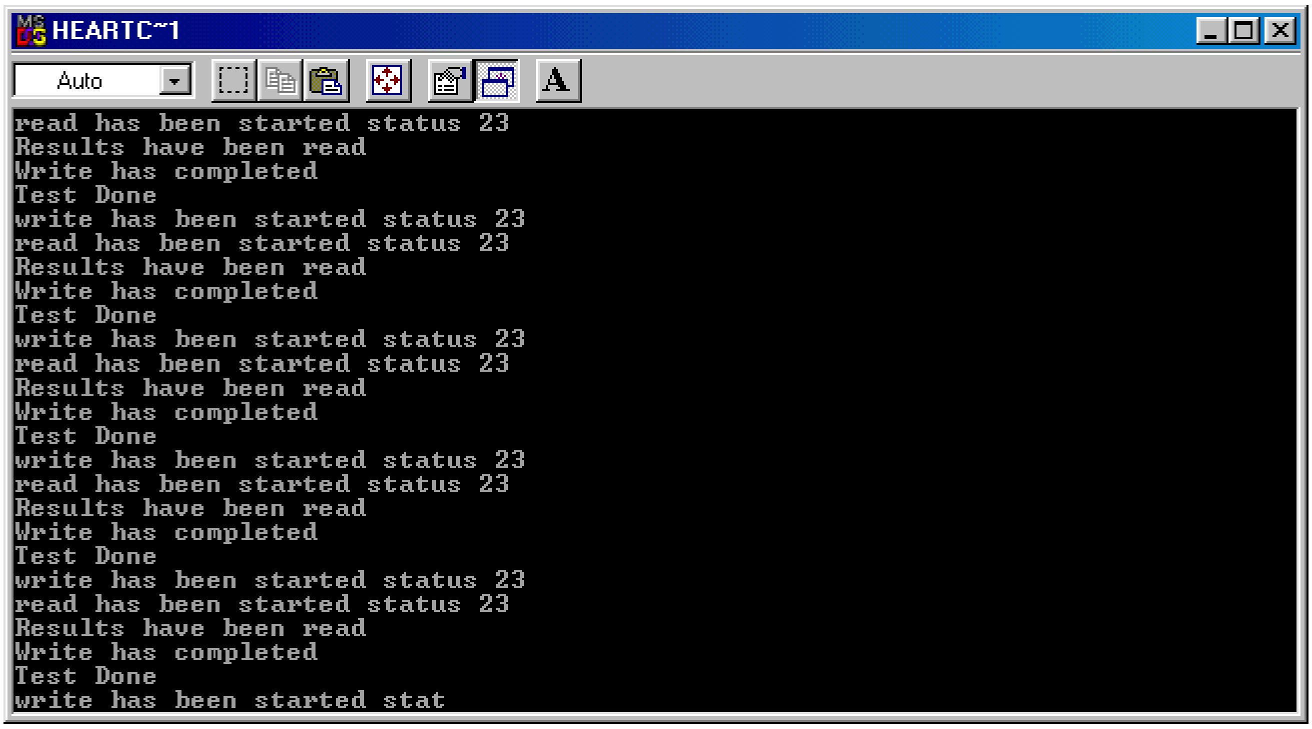
The source file for this program is also included so you can rebuild it with any changes you want.

This Host example serves as a confidence check and an example of how to send and receive data to and from the FPGA module using both the HSB and the FIFOs.

The program should start to loop quickly sending, receiving and checking blocks of data. If no print messages can be seen then the example is not running correctly.

See the FPGA working

i.e you should see



```
read has been started status 23
Results have been read
Write has completed
Test Done
write has been started status 23
read has been started status 23
Results have been read
Write has completed
Test Done
write has been started status 23
read has been started status 23
Results have been read
Write has completed
Test Done
write has been started status 23
read has been started status 23
Results have been read
Write has completed
Test Done
write has been started status 23
read has been started status 23
Results have been read
Write has completed
Test Done
write has been started stat
```

Assuming that the test completed successfully, you are now ready to proceed to the next stage. We have validated that we can access the FPGA, and that the FPGA is functioning correctly. It is worth looking at the DSP or host code to understand how the test works before proceeding to the section "Modifying the FPGA Program".

b) <u>C6000 example</u>

Run this only if you want to see the interaction between a C6000 module and your FPGA module.

Make a directory for the example on your hard drive. In your explorer window (from stage1) change to the \fpga\getting started\dspex1 directory. Copy the C file from there into the directory that you just made. Now open Code Composer Studio, and use the "create new HERON project" plug in to generate a project for your DSP module that uses the C program you have copied. If you are not sure how to do this, then refer to the Getting started with C6000 section of the CD and the tutorials there.

The project should build without any changes.

Now you need to understand which FIFO number the DSP will use to access the FPGA, and which FIFO number the FPGA will use to access the DSP.

HEPC9

For an HEPC9, you can use the heartconf utility to make the connections how you want, or you can use the default routing jumpers to make a simple connection.

For full instructions on how to use the heartconf utility, or the jumpers you should efer to the documentation for the HEPC9, but lets look at a simple example.

If you have a DSP module in Slot1, and the FPGA module in slot2, you can use a network file to connect whichever FIFOs you choose, i.e.

```
# Using API
BD API HEP9A 0 0
# Nodes description
# ND  BD_nb  ND_NAME  ND_Type  CC-id HERON-ID  filename(s)
#------------------------------------------------------------
  ibc  0      ibc1    normal          0x06
  pcif 0      host1   normal          0x05
  c6   0      dspmod  normal          0x01  none.out
  fpga 0      fpgamod normal          0x02  none
#--------------------------------------------------------
#       from:slot  fifo  to:slot   fifo   timeslot
#--------------------------------------------------------
heart      dspmod  2      fpgamod  5      1
heart      fpgamod 5      dspmod   2      1
```

Would connect FIFO #2 of the DSP module to FIFO5 of the FPGA module. You can use this “network” file from the heartconf entry under programs→HUNT ENGINEERING group, but as you will be using Code Composer Studio to build and load the DSP example code it is probably better to use the settings on the Reset plug in to configure the network for you after each reset. (The connections get cleared by the reset)

Using the same module set up, you could use the Default Routing jumpers for such a simple, set up. Fit the input and output jumpers to 0 on both modules and you will have a connection between the FIFO#0 of the DSP module and FIFO#0 of the FPGA module in both directions.

HEPC8

For an HEPC8 the connections are determined by which slots your modules are fitted to. There is probably a document shipped with your system that explains how your system is configured, but if you have lost that or changed your configuration you will need to refer to the user manual for the module carrier you are using to work it out.

For example if the DSP is in slot 1 of an HEPC8, and the FPGA is in slot 2, the DSP will use FIFO #2, and the FPGA will use FIFO #3.

When you run the DSP example, you will be prompted to enter the FIFO number that the DSP should use, then which slot number the FPGA module is in. Lastly you will be asked which input and output FIFO numbers you want the FPGA to use.

The DSP program will send some messages over the HSB to set registers in the FPGA example1 that “connect” to the FIFOs you have selected. Then the DSP will send and receive data through the HERON FIFOs and test that the FPGA module returns the same value as was sent.

A common error here is not resetting the system. The test writes data to the FPGA and reads it back. If there is any old data in the FIFOs, this will be read instead. A system reset will flush the stale data out of the FIFOs – make sure you use the “Reset System” plug-in!

This DSP example serves as a confidence check and an example of how to send and receive data to and from the FPGA module using both the HSB and the FIFOs.

The code will loop quite quickly printing out what it is doing. If no prints are seen then the example is not running correctly.

Rebuild the FPGA program

The FPGA Program

At this stage you should have been able to test the FPGA using either the DSP or host. However, before the FPGA is useful, we will want to develop our own FPGA code. The following assumes that you are using the ISE toolset – If you have another synthesizer tool then please refer to the “Using non ISE development tools” application note.

This part of the tutorial will show you how to make some very simple modifications to the FPGA program – another tutorial shows you how to implement more complex programs.

The example projects for ISE are shipped on the HUNT ENGINEERING CD.

Using these projects will allow you to run the complete design flow, from RTL-VHDL source files to the proper bitstream, ready to download on your Heron FPGA board.

No special skills are required to do this.

However, if you want to write your own code and start designing your own application, you must make sure that you have acquired the proper level of expertise in:

*VHDL language

*Digital Design

*Xilinx FPGAs

*ISE environment and design flow.

Proper training courses exist which can help you acquire quickly the required skills and techniques. Search locally for courses in your local language.

You may also visit ALSE's Web site where you will find design tips, useful links, design examples, etc… at : http://www.alse-fr.com (then click on the "English version" banner).

Preparing ISE

Before opening the getting started example you will first need to make sure ISE is properly installed. In addition, you should ensure that you have downloaded the latest service pack from the Xilinx website for the version of ISE you are using.

Important: If you are using a VirtexII - ES FPGA, make sure that the environment variable : XIL_BITGEN_VIRTEX2ES is properly set to 1.

Using Different Versions of ISE

The getting started example 'Example1' provided on the HUNT ENGINEERING CD is written to use the latest version of ISE. It may be that you are using a different version of ISE, in which case you must continue the getting started example by working through the application note 'Using Different Versions of ISE'.

If you are using a different version of ISE, first work through the application note 'Using Different Versions of ISE' and then continue through this document starting from the section 'Project's Functional Parameters'.

Copying the examples from the HUNT ENGINEERING CD

On the HUNT ENGINEERING CD, under the directory "fpga" you can find directories for each module type. In the case of the HERON-FPGA3 the correct directory is "fpga3v1".

There are two ways that you can copy the files from the CD.

The directory tree with the VHDL sources, bit streams etc can be copied directly from the CD to the directory of your choice. In this case there is no need to copy the .zip file, but the files will be copied to your hard drive with the same read only attribute that they have on the CD. In this case all files in the "examplex" directories need to be changed to have read/write permissions. It is a good idea to leave the permissions of the common directory set to read only to prevent the accidental modification of these files.

To make the process more convenient we have provided the zip file, which is a zipped image of the same tree you can see on the CD. If you "unzip" this archive to a directory of your choice, you will have the file permissions already set correctly.

Opening the Example1 Project in ISE

In the tree that you have just copied from the CD, open the `Example1` sub-directory. You should see some further sub-directories there:

* `ISE` holds the ISE project files.
* `Src` holds the application-specific Source files.
* `Sim` holds the simulation scripts for ModelSim.
* `Leo_Syn` holds the synthesis scripts for Leonardo Spectrum and Synplify users.
  (You may ignore this directory in this tutorial.)

Open the Xilinx ISE Project Navigator. If a project pops up (from a previous run), then close it. Use `File→Open project`.

You need to select and open the correct ISE project file (with the `.ise` or `.npl` extension) from the `ISE` directory under the example1 directory. There may be several project files for different versions of your module.

The project should now open in ISE without error. If you are encountering errors at this stage, you should verify that:

The example files have been correctly copied onto your hard disk, and especially the `\Common` and `Example1\Src` directories.

The correct version of ISE has been successfully installed. Be sure to have installed XST VHDL synthesis and the support for the correct FPGA families.

Project's functional parameters

Double click on "`USER_Ap`" in the Sources window. This opens the VHDL colour-coded text editor so that you can see the part of the project where you can enter your own design.

The first code that you will see at the beginning of this file is a VHDL Package named "`config`" which is used to configure the design files according to the application's requirements. See the next section of this manual for a description of these items

Below the package section, you will see the User_Ap1's VHDL code. This is where you will insert your own code when you make your own design.

We provide a system which is built in such a way that the user should not edit any other file than User_Ap (and the entities that this module instantiates).

In particular, the user should NOT modify the HE_* files,
even when creating new designs for the FPGA.

Setting up the Configuration Package.

At the top of the user_ap.vhd file there are the settings that you can use to affect your design (in this case the example). The idea is that settings that are often changed are found here.

1. Virtex II Engineering Samples

Set VIRTEX2_ES to "True" if your board is equipped with an "ES" version of the Virtex-II.
Set it to "False" if you are using a Virtex II "Production" part, or any other FPGA type. This allows the example to work around a problem with the tools, which applies only to the ES silicon, where the Clock DLL can be placed in a location where it cannot be connected.

2. Divide External Clock by 2

The example uses the 100Mhz oscillator that is fitted to the module. It generates the FIFO clock either directly from this 100Mhz, or divides it by 2 to generate a 50Mhz FIFO clock. Unfortunately the HEPC8 module carrier cannot support a clock as high as 100Mhz, and the HEPC9 carrier cannot support a clock as low as 50Mhz.

Set this parameter to "True" if you want to divide the external clock by two and use this as your main Clock.

If you are using an HEPC8 carrier board, set DIV2_FCLK to "True".

If you are using an HEPC9 carrier board, set DIV2_FCLK to "False".

3. Fifo Clocks

You must decide whether you will have a single common clock for driving the input and output Fifos. Normally a design is simpler if the same clock is used for input and output FIFOs, but the module design allows you to use different frequencies or phases if that is more convenient for the design of your system. Whether you use a common clock or separate clocks will affect your design, but it also affects the use of clocks in the Hardware Interface Layer.

Set FCLK_G_DOMAIN to True if you have the same clock driving both Fifos.
This is the default option for the Examples. If you are unsure, select this choice.

Then, you must know whether your clocks are running slower than 60 MHz or not.

Set HIGH_FCLK_G to True if your global clock is running at 60 MHz or above. In this case an HF DLL will be used in the FIFO clocks to ensure the proper timing.

Set HIGH_FCLK_G to False otherwise. In this case the HF DLL does not work, and indeed no DLL is necessary.

Set FCLK_G_DOMAIN to False is you have a different clock driving each Fifo.
This option should be reserved to advanced users familiar with the management of multiple clock domains systems.

Then, you must know whether your clocks are running slower than 60 MHz or not.

Set HIGH_FCLK_RD to True if your Input Fifo clock is running at 60 MHz or above, so that the HF DLL will be used for the input FIFO clock.

Set HIGH_FCLK_RD to False otherwise, so no DLL will be used for the input FIFO clock.

Set HIGH_FCLK_WR to True if your Output Fifo clock is running at 60 MHz or above, so that the HF DLL will be used for the output FIFO clock.

Set HIGH_FCLK_WR to False otherwise, so that no DLL will be used for the output FIFO clock.

The Table below summarises the available choices:

FCLK_G_DOMAIN	HIGH_FCLK_G	HIGH_FCLK_RD	HIGH_FCLK_WR
True	True / False	n.a.	n.a.
False	n.a.	True / False	True / False

In the case of example1, the correct choices are:-

For the HEPC8

DIV2_FCLK	FCLK_G_DOMAIN	HIGH_FCLK_G	HIGH_FCLK_RD	HIGH_FCLK_WR
True	True	False	n.a.	n.a.

For the HEPC9

DIV2_FCLK	FCLK_G_DOMAIN	HIGH_FCLK_G	HIGH_FCLK_RD	HIGH_FCLK_WR
False	True	True	n.a.	n.a.

You also need to consider the timing constraints that are defined in the .ucf file for your design. Actually if you use a time specification that is more strict than needed there is no problem, so the standard .ucf file have the clocks specified at 100Mhz. If the project builds (as example1 does) with this specification it is still guaranteed to work at lower clock speeds. If you add new clock nets into your design then you need to add new timing constraints into your design.

Creating the Bitstream for Example1

Once the project has been opened as described above :

1. In the "Sources in project" window (Project Navigator), highlight (single-click on) "top".

   This is extremely important! Otherwise, nothing will work!

2. Double-click on the "Generate Programming File" item located in the "Processes for Current Source".

   This will trigger the following activity :

   Complete synthesis, using all the project's source files.
   Since warnings are generated at this stage, you should see a yellow exclamation mark appear besides the "Synthesize" item in the Processes window.

   Complete Implementation :
   - "Translation"
   - Mapping
   - Placing
   - Routing

3. Creation of the bitstream.
   Note that this stage does run a DRC check. Which can potentially detect anomalies created by the Place and route phase (especially with Virtex II ES parts).

   When the processing ends, the proper bitstream file, with extension `".rbt"` can be found in the project directory.

4. In the "Pin Report" verify a few pins from the busses, like :
   LED(0) = H3, LED(1) = J1, LED(2) = L5, LED(3) = J5, etc...
   If you see different assignments, STOP HERE, and verify the UCF file selected for the project.

   You can download this file on your FPGA board and see how it works.

Note that the user_ap level includes a very large counter which divides the main system clock and drives the LED #4. It is then obvious to see if the part has been properly programmed and downloaded: the LED should flash.

If the LED does not flash, try a hardware reset. The DLL used in the clock generation often does not start until a hardware reset. If the LED still does not flash we recommend that you shut down the PC or reprogram the device using a "safe" bitstream. Otherwise, some electrical conflicts may happen (see below).

Test the rebuilt FPGA

Load the bitstream that you have just generated (top.rbt) and re-run the DSP or Host example that you did before – It should work in exactly the same way as we have not changed anything. If it does not then there is a problem with the way you are building the project. You must resolve that before continuing. If you cannot make it work by re-reading the instructions here - then you will need to contact support.

Possible causes for the device failure to operate are :

1. Wrong (or no) UCF file.
This happens (for example) if you select the XST-version of the UCF with a Leonardo Spectrum (or Synplify) synthesis. The pin assignment for all the vectors (busses) will be ignored, and these pins will be distributed in a quasi-random fashion!

2. Wrong parameters in the CONFIG package.

3. Design Error.

If nothing seems obvious, rerun the confidence tests, then return to the original example 1.

Simulating the complete design

To generate the bitstream as above, you did not need to do any simulation. However, if you start modifying the provided examples and add your own code, verification will very soon became a central issue.
You will then need to install a VHDL simulator, like ModelSim (Xilinx Edition, Personal Edition, or Special Edition). If you do not have Modelsim then you can ignore this section.

We provide behavioural simulation models for use at this stage. (`SIM_*.vhd under /common directory, + test benches`).

We will explain very briefly how to do a complete functional simulation of the design with ModelSim (any version except the demo version, a.k.a. "Starter" which is too crippled for that purpose).

ALSE has designed behavioural models for the FPGA's environment (FIFOs, HSB interface,…) that, coupled with the FPGA design, enable the user to simulate the complete application in a very realistic way. This simulation is usually purely functional, meaning that no timing is taken into account.
This is part of the Synchronous Design Methodology and makes sense when specific design rules have been respected, and when timing issues are verified by other means (like static timing analysis). Note that advanced users may perform timing simulations (post-layout, Vital).

1. Start ModelSim.

2. Change the current directory to Example1/Sim

   `cd x:/xxx…xxx/Example1/SIM` (replace the X'es with your path)

3. Verify that the Xilinx-specific primitives libraries `UNISIM` and `XilinxCoreLib` are properly set up and accessible. you may use the "`Design > Browse library`" command for that purpose. If these libraries are not properly compiled and/or mapped, you will not be able to simulate the

design. In this case, refer to the Xilinx documentation (the steps to follow differ with the ModelSim versions). Just a hint : do *NOT* recompile the libraries if you are using the XE version…

4. From the transcript window, type :
   `do simu.do`
   or use the GUI command "`Macro > Execute Macro`" and select `simu.do`.

ModelSim should then compile all the design units, load the test bench, display all the top-level signals in the Waveform viewer, run the simulation, issue messages in the transcript, and stop the simulator. Note that vector Files are read and created during the simulation.

The simulation has created Fifo_out.txt that you may compare with Fifo_In.txt !
You may use the DOS command:

```
fc Fifo_In.txt Fifo_out.txt.
```

This should prove that the data flows correctly :

Input file → External Fifo_In model → FPGA's Fifo interface
→ User_Ap1's Internal control logic → FPGA's Fifo out Interface
→ External Fifo_Out model → Output file.

You could now investigate and see how we have built the simulation environment, and specifically the behavioural models (which read and write from/to text files).

If you look at the waveforms, you will notice that our External Fifo behavioural models do simulate some irregular data flow (with busy times) to ensure that the Full/Empty logic works fine.

See the separate application note on simulating HUNT ENGINEERING FPGA examples if you want more details.

Contact ALSE if you are interested in specific models and/or training courses.

Looking at the Example

Now you can open the example by double clicking on the file User_Ap1.vhd in the Project manager.

Inside the file User_Ap1.vhd you will see some components of the design. One of these is HSB1.

Lets look at what these components do.

Double click on hsb1.vhd in the design manager to open it.

In the middle of the file you can see the "guts":

```
INFIFO_SEL  <= REG_0(5 downto 0);
OUTFIFO_SEL <= REG_1(5 downto 0);

-- Decode address & output mux
process(ADDR, REG_0, REG_1)
begin
  EN <= (others=>'0');
  case ADDR is
    when "00000000" => EN(0)<='1'; DOUT <= REG_0;
    when "00000001" => EN(1)<='1'; DOUT <= REG_1;
    when others     => DOUT <= (others=>'-');
  end case;
end process;

-- DATA_REGs
process (CLK)
begin
  if rising_edge(CLK) then
    if WEN='1' then
      if EN(0)='1' then
        REG_0 <= DIN;
      end if;
      if EN(1)='1' then
        REG_1 <= DIN;
      end if;
    end if;
  end if;
end process;

-- READY
process (RST, CLK)
begin
  if RST='1' then
    READY <= '1';
  elsif rising_edge(CLK) then
    READY <= WEN OR REN;
  end if;
end process;
```

The INFIFO_SEL and OUTFIFO_SEL buses are driven by the registers REG_0 and REG_1.

There is a multiplexor used to read back either of the registers according to the address bit ADDR(0).

```
INFIFO_READ_REQ <= NOT FIFO_AF;

 iFIFO0 : fifo15x32
      port map (
         din               => INFIFO0_D,
         wr_en             => INFIFO_DVALID(0),
         wr_clk            => FCLK_G,
         rd_en             => FIFO_RD(0),
         rd_clk            => FCLK_G,
         ainit             => RESET,
         dout              => FIFO_A,
         full              => open,
         empty             => FIFO_EF(0),
         almost_full       => FIFO_AF(0),
         almost_empty => open );
```

The registers REG_A and REG_B are written with the data in, according to the address value when WEN is set high.

The READY signal is driven by a registered version of WEN or REN, so that all accesses whether read or write are single cycle.

```
process(FCLK_G)
begin
  if rising_edge(FCLK_G) then
    MUX_SEL(0) <= INFIFO_SEL(1) OR INFIFO_SEL(3) OR INFIFO_SEL(5);
    MUX_SEL(1) <= INFIFO_SEL(2) OR INFIFO_SEL(3);
    MUX_SEL(2) <= INFIFO_SEL(4) OR INFIFO_SEL(5);
  end if;
end process;

process (MUX_SEL, FIFO_A, FIFO_B, FIFO_C, FIFO_D, FIFO_E, FIFO_F)
begin
  case MUX_SEL is
    when "000"  => FIFO_G <= FIFO_A;
    when "001"  => FIFO_G <= FIFO_B;
    when "010"  => FIFO_G <= FIFO_C;
    when "011"  => FIFO_G <= FIFO_D;
    when "100"  => FIFO_G <= FIFO_E;
    when "101"  => FIFO_G <= FIFO_F;
    when others => FIFO_G <= (others=>'0');
  end case;
end process;

AF_v <= AF & AF & AF & AF & AF & AF;

FIFO_RD <= (NOT FIFO_EF) AND (NOT AF_v) AND INFIFO_SEL;

NOT_EMPTY_A <= ((NOT FIFO_EF(0)) AND INFIFO_SEL(0)) OR
              ((NOT FIFO_EF(1)) AND INFIFO_SEL(1));

NOT_EMPTY_B <= ((NOT FIFO_EF(2)) AND INFIFO_SEL(2)) OR
              ((NOT FIFO_EF(3)) AND INFIFO_SEL(3));

NOT_EMPTY_C <= ((NOT FIFO_EF(4)) AND INFIFO_SEL(4)) OR
              ((NOT FIFO_EF(5)) AND INFIFO_SEL(5));

process(RESET, FCLK_G)
begin
  if RESET='1' then
    WR_EN  <= '0';
  elsif rising_edge(FCLK_G) then
    WR_EN <= '0';
    if (NOT_EMPTY_A='1' or
         NOT_EMPTY_B='1' or
         NOT_EMPTY_C='1') and AF='0' then
      WR_EN <= '1';
    end if;
  end if;
end process;
```

Now look at the VHDL in the file User_Ap1.vhd which is already open.

In this source module are six small Core-Generator FIFOs. Each FIFO is connected to the Six FIFO Read Interface component of the Hardware Interface Layer.

The INFIFO_READ_REQ signal is asserted whenever these FIFOs show there is space to write.

Data is written into these FIFOs on each clock cycle where the INFIFO_DVALID signal is asserted to indicate valid data.

Further on in this source module, there is some code that transfers data from any of the six small Core-Gen FIFOs to a seventh FIFO, FIFO-G.

In this logic, the INFIFO_SEL bus, written over HSB, is used to select which of the six input FIFOs is used to provide data.

Data is read from one of the first six FIFOs and transferred to FIFO G, when the first FIFO is not EMPTY, and when FIFO G

```
 iFIFO6 : fifo15x32
  port map (
       din              => FIFO_G,
       wr_en             => WR_EN,
       wr_clk          => FCLK_G,
       rd_en            => RD_EN,
       rd_clk          => FCLK_G,
       ainit             => RESET,
       dout              => OUTFIFO_D,
       full               => open,
       empty            => EMPTY,
       almost_full     => AF,
       almost_empty => open );

READY_A <= (OUTFIFO_READY(0) AND OUTFIFO_SEL(0)) OR
           (OUTFIFO_READY(1) AND OUTFIFO_SEL(1));

READY_B <= (OUTFIFO_READY(2) AND OUTFIFO_SEL(2)) OR
           (OUTFIFO_READY(3) AND OUTFIFO_SEL(3));

READY_C <= (OUTFIFO_READY(4) AND OUTFIFO_SEL(4)) OR
           (OUTFIFO_READY(5) AND OUTFIFO_SEL(5));

RD_EN <= (READY_A OR READY_B OR READY_C) AND (NOT EMPTY);

EMPTY_v <= EMPTY & EMPTY & EMPTY & EMPTY & EMPTY & EMPTY;

WRITE <= (NOT EMPTY_v) AND OUTFIFO_READY AND OUTFIFO_SEL;

process(RESET, FCLK_G)
begin
  if RESET='1' then
    OUTFIFO_WRITE <= (others=>'0');
  elsif rising_edge(FCLK_G) then
    OUTFIFO_WRITE <= WRITE;
  end if;
end process;
```

is not ALMOST FULL.

To transfer the data from one of the six FIFOs, there is a 6-to-1 multiplexor that is selected according to the state of the INFIFO_SEL bus.

Data placed in FIFO-G is then transferred to the Heron Six FIFO Write Interface of the Hardware Interface Layer.

When the six FIFO write interface OUTFIFO_READY signal is asserted, this indicates data can be transferred. The output FIFO that data is transferred to is selected via the OUTFIFO_SEL bus that is programmed over HSB.

The other requirement that must be met before data is transferred is that FIFO-G is not EMPTY. This is done by only transferring data when the signal EMPTY is set low.

Changing the Example

Modify the FPGA program

Now change the example as follows, and build the new bitstream. This will prove that you are properly set up to generate your own FPGA programs. If you experience any problems here, you will not be able to continue with your development until they are resolved. Work through the instructions again, and if you still have problems you will need to contact support.

In the file User_Ap1.vhd you can find the line:

```
dout => OUTFIFO_D,
```
(this line is part of the port map of FIFO-G)

This connects the output bus of FIFO-G to the FIFO data input bus of the HERON Six FIFO Write Interface of the Hardware Interface Layer.

As an exercise add a new signal (before the 'begin' statement), as follows:

```
signal NEW_DATA_BUS : std_logic_vector(31 downto 0);
```

Then change the port map of FIFO G:

```
dout => NEW_DATA_BUS,
```

And add the following line:

```
OUTFIFO_D <= NOT NEW_DATA_BUS;
```

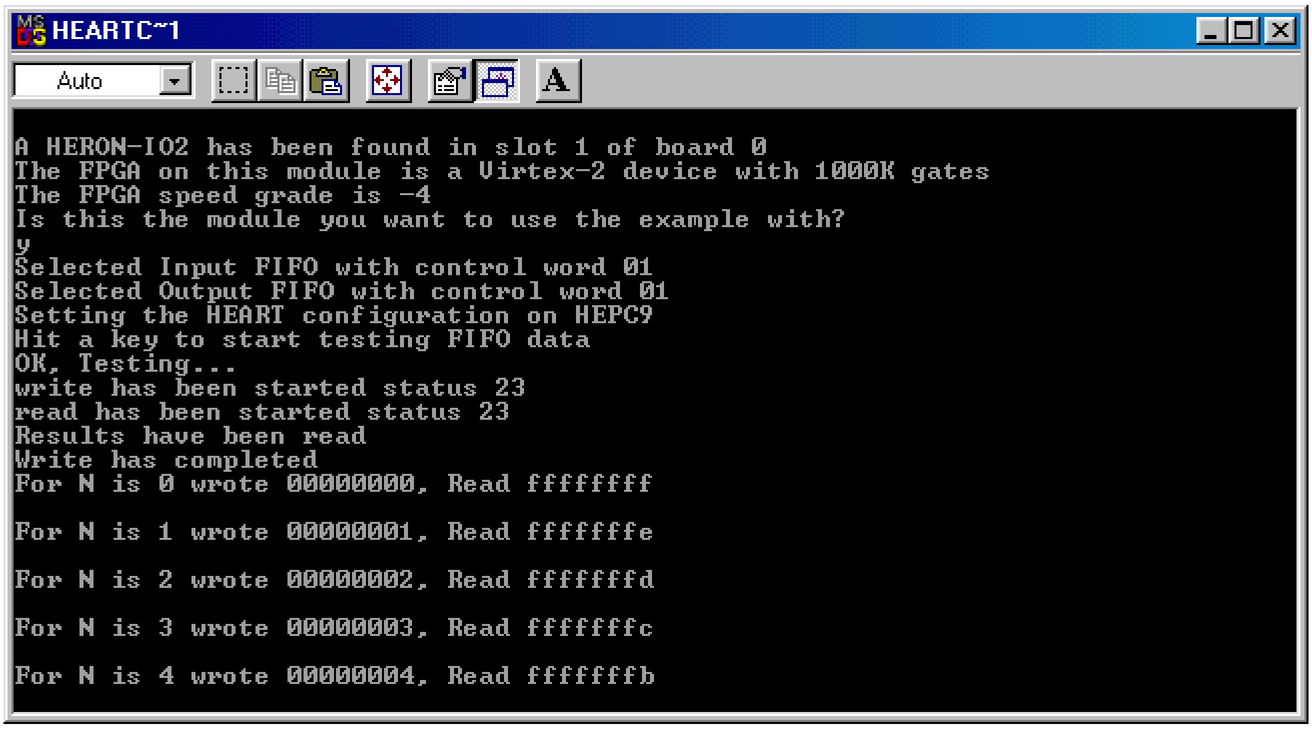
Doing so should connect the inverse of the FIFO-G data to the HERON FIFO data out. That is, you have just inserted a 32 bit inverter.

Rebuild the project as detailed in the section above.

Check that the timing constraints are still met. What you have done is inserted some logic between the register stages that are part of the Hardware Interface Layer. The simple inverter should not be hard for the FPGA tools to route efficiently, but if you inserted more and more logic here without introducing more registers, you would eventually cause the timing constraints to fail. This shows the importance of designing synchronous logic that pipelines the data processing to keep the processing clock rate high.

Test the modified FPGA program

Now load your new bitstream (top.rbt) and re-run the Host or DSP example you used before. You should see that errors are given, printing out that the data has all bits inverted. I.e.



HEARTC~1

```
A HERON-IO2 has been found in slot 1 of board 0
The FPGA on this module is a Virtex-2 device with 1000K gates
The FPGA speed grade is -4
Is this the module you want to use the example with?
y
Selected Input FIFO with control word 01
Selected Output FIFO with control word 01
Setting the HEART configuration on HEPC9
Hit a key to start testing FIFO data
OK, Testing...
write has been started status 23
read has been started status 23
Results have been read
Write has completed
For N is 0 wrote 00000000, Read ffffffff

For N is 1 wrote 00000001, Read fffffffe

For N is 2 wrote 00000002, Read fffffffd

For N is 3 wrote 00000003, Read fffffffc

For N is 4 wrote 00000004, Read fffffffb
```

You can also re-run the simulation and see the results of your inversion in the text file generated.