FP7- Grant Agreement no. 283393 – *RadioNet3*

Project name: Advanced Radio Astronomy in Europe

Funding scheme: Combination of CP & CSA

Start date: 01 January 2012 Duration: 48 month

Deliverable 8.1

Document on definition of coding interfaces and conventions

Due date of deliverable: 2012-12

Actual submission date: 2012-12-17

Deliverable Leading Partner: JOINT INSTITUTE FOR V.L.B.I. IN EUROPE (J.I.V.E.)

An European project supported within the 7th framework programme

Document information

Document name:	Document on definition of coding interfaces and conventions
Type	Report
WP	8 (UniBoard^2)
Authors	Jonathan Hargreaves (JIVE)
	Eric Kooistra (ASTRON)
	Arpad Szomoru (JIVE)

Dissemination Level

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Content:

Terminology:

ASIC	Application Specific Integrated Circuit
CX4	Standard four lane copper interface for XAUI connections
DDR3	Double Data Rate memory interface standard
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
Gsps	Giga Samples Per Second
PPS	Pulse Per Second
SDC	Synopsys Design Constraints
SFP+	Small Form factor Pluggable interface
SKA	Square Kilometer Array
Sps	Samples Per Second
SVN	Subversion (database for source code version control)
TCL	Tool Command Language
UDP	User Datagram Protocol (packet layer used for data transport via Ethernet)
UNB	UniBoard
VHDL	Very high speed integrated circuit Hardware Description Language

References:

1. “Specification for module interfaces using VHDL records”, ASTRON-RP-380, E. Kooistra
2. $UNB/doc/howto/how_to_write_VHDL.txt, E. Kooistra
3. http://www.radionet-eu.org/fp7wiki/doku.php?id=jra:uniboard:documents – UniBoard Wiki
4. http://www.radionet-eu.org/uniboard2 - UniBoard^2 Wiki
5. https://svn.astron.nl/UniBoard_FP7/ - SVN repository for UniBoard source code
6. https://support.astron.nl/astron_issuetracker/projects/uniboard - UniBoard bug tracking

1 Introduction

In distributed projects, like RadioNet3, the re-use and sharing of code could potentially save a tremendous amount of effort. Throughout the UniBoard project however, even through firmware was stored in a common repository accessible to all partners, exchange of code hardly occurred. Part of the effort in UniBoard² will deal with formalizing the exchange mechanism through the definition of coding conventions and common interfaces, in order to optimize the re-use and the combination of available blocks of firmware among developers.

The aim of this document is to propose guidelines to make code reuse amongst the partners easier. The emphasis is on providing the documentation and files necessary for another engineer to understand what the module does and how to connect it up in another application. These steps are listed in *Section 3, Requirements*. In many cases modules can be reused without reading the source code, however this will be necessary if the module needs to be modified and maintained. The steps listed in *Section 4, Recommendations,* are intended to help an engineer understand how the module works. First, in *Section 2*, a brief summary of both UniBoard and the current UniBoard^2 project is given.

2 UniBoard and UniBoard^2

2.1. UniBoard Research Activity

UniBoard was a research activity of RadioNet FP7 (grant number: 227290) [3], which ended in June 2012.

The aim of the UniBoard project was to create a generic platform for radio astronomy signal processing with as much processing power and IO as could reasonably fit on one board. Standard 10 gigabit Ethernet interfaces would be used for IO and front-to-back symmetry would allow data flow in both directions. All processing nodes would be identical and as generic as possible so that the same hardware could be used for many applications.

2.2. UniBoard Hardware

A block diagram of the UniBoard hardware is shown in Figure 1. The large squares denoted FN (front node) 0-3 and BN (back node) 0-3 represent eight Altera EP4SGX230 field programmable gate arrays (FPGAs). An FPGA is a programmable integrated circuit which contains an array of logic elements which can be wired together to perform a function similar to that performed by a fully customized application specific integrated circuit (ASIC). Unlike an ASIC however, an FPGA can be re-programmed as often as needed. Its function is determined by firmware, usually written in a hardware description language such as VHDL (Very high speed integrated circuit Hardware Definition Language).

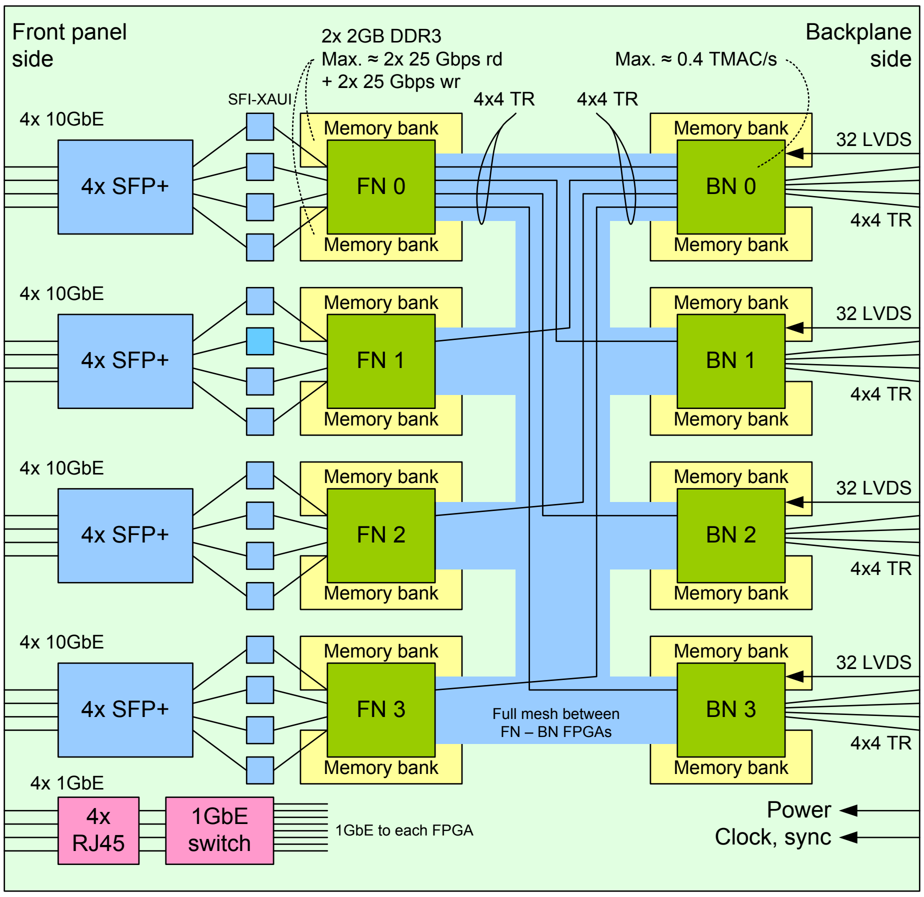

Figure 1: Block Diagram of UniBoard Hardware

All FPGAs contain an array of logic cells that can be programmed to form such structures as AND/OR gates, multiplexers and small memories. The FPGAs on the UniBoard also contain 1288 eighteen bit multipliers, 182400 flipflops and 14.6Mbits internal memory. Four multipliers can be linked to form one eighteen bit complex multiplier. The multipliers offer high performance with the result available within 1.2ns, however the overall performance of a system depends on the speed with which data can be moved in and out of the multiplier cells. This is highly application dependent because routing delays between elements inside the FPGA increase with the complexity of the design. At a realistic system clock rate of 250MHz the whole board can perform 644E9 complex multiply-accumulate operations per second.

Two 64bit wide, 1066MT/s SODIMM (laptop memory) slots are connected to each FPGA. These allow the whole board to be configured with up to 64GB DDR3 storage. Each FN is connected to each BN by a 24Gbps fully duplex transceiver mesh. Only the connections to FN0 and BN0 are shown in Figure 1 for clarity.

External connections differ slightly between the front and back nodes. All nodes have four external ten-gigabit Ethernet ports, although currently only three can be used simultaneously due to internal routing restrictions in the FPGAs. The FNs are provided with SFP+ cages, whilst the BNs use CX4 copper connectors. Additionally each BN has 32 low voltage differential signal (LVDS) input pairs suitable for connecting an analog to digital converter board. Two differential pair input signals are distributed to all eight FPGAs: one for use as an external system clock and the other as a global sync pulse such as a PPS.

The power requirements are a -48V supply at 10A, with typical consumption of 2.5 to 7A depending on the application.

A single chip gigabit Ethernet switch, shown at the bottom left of Figure 1, provides a 1Gbps control connection to each FPGA. A JTAG boundary scan bus permits continuity testing between major components and downloading the FPGA firmware, either directly to the FPGA or to a non-volatile EEPROM placed beside each FPGA. The EEPROMs can hold up to three compressed images:

typically a safe power-up configuration and one or two application configurations. It is possible to reprogram the EEPROM via the Ethernet control port, so that the JTAG cable can be removed in production installations.

2.3. UniBoard Applications

The UniBoard architecture is especially suited for processing one or multiple, high-bandwidth input signals in parallel. The LVDS interfaces at the back nodes can directly connect to an external board with one or multiple ADCs. For examples the 4 back nodes on one UniBoard together allow connecting 16 digitized signals from 16 eight-bit ADCs each sampling at up to 1 Gsps. For a very wideband digital receiver one UniBoard may even use the LVDS interfaces to connect 1 digitized signal from 1 eight-bit ADC that is sampling at 16 Gsps. The input signals may also be digitized a distant location that can be meters or even thousands of kilometers away. The digitized signals are then send to the UniBoard via the gigabit links, in which case they can enter the UniBoard at the front nodes as packetized data, e.g. in the form of UDP/IP packets via 10G Ethernet. On the UniBoard typically the input signals are first separated into frequency subbands by a set of filterbanks. The frequency subbands are independent and therefore they can be processed further at different nodes. The subbands are distributed to different nodes via the full duplex giga bit links of the mesh-interconnect between the front nodes and the back nodes on the UniBoard. Typical examples of astronomical applications that use processing of multiple input signals at subband level are correlation and beamforming. Furthermore the large DDR3 memories on the UniBoard support temporary storage of raw sampled input signal data or subband data e.g. for delay compensation and event capturing.

2.4. Firmware Development

As noted previously, FPGA applications consist of firmware written in VHDL or Verilog. It is also possible to enter a design graphically but the UniBoard applications have been written in text as this allows better control over the implementation and improves portability. During development, ModelSim simulation software is used to check the design for functional correctness. Synthesis software provided by the FPGA manufacturer maps the design onto the logic elements available on the chip, and then performs a place-and-route operation to connect everything up. Finally a bit file is generated which can be downloaded to the FPGA.

Generally the term ‘design’ refers to a top-level entity whose ports are connections to the external pins of the FPGA. Typical designs contain a hierarchy of modules. The term ‘module’ covers a wide range of complexity, from a single entity performing a simple logic operation, to many thousands of lines of code spread over several files. A module can also refer to a library of several related entities.

UniBoard designs make extensive use of Altera MegaWizard IP blocks for infrastructure functions such as the 10 Gigabit Ethernet ports, DDR3 controllers, DSP functions and FIFOs. The remaining code is custom written by engineers at the participating institutions, however there are many functional blocks which are common across different applications – from simple counters to FFTs and filter banks. Clearly there is an advantage if code written for one application can be reused in another. To an extent this was already done in the UniBoard project since finished designs and modules are shared between the project partners using an SVN repository, however relatively few of the shared modules were actually re-used. The purpose of this document is to recommend ways of making code re-use work better in UniBoard^2.

2.5. Introduction to UniBoard2

UniBoard² (grant number 283393) started on July 1st 2012. UniBoard² will create an FPGA-based, generic, scalable, high-performance computing platform for radio-astronomical applications [4]. This WP consolidates and builds upon the experience obtained through the UniBoard project to create a completely re-designed platform with several innovative features, that will be ready for the next generation of astronomical instruments (notably the SKA), at the end of 2015.

Power efficiency is going to be a crucial issue for future instrumentation. For this platform the newest technology available on the timescale of the project will be used, which means replacing the current 40 nm with 28 nm or even 20 nm FPGAs. The use of a technique offered by FPGA manufacturers under names such as HardCopy or EasyPath will be investigated. This enables one to develop on standard FPGAs and then to freeze the design into ASICs with the same footprint. While a full-blown hard-copy production run is not feasible due to the high initial cost involved, UniBoard² will design the applications with hard copy in mind, and run extensive simulations to determine its effect on power consumption. Further "green" measures will include the use of non-leaded components, the careful balancing of system parameters and performance and the optimization of firmware designs and algorithms.

To be able to freeze an application design into a hard-copy FPGA it is essential that the firmware code is robust and well-tested. Modular design and code-reuse greatly help to make the code more robust. Modular design allows a piece of code to be reused in different applications. Reusing modules increases the reliability of the module and the application because if a module already works fine in many applications it is likely to work fine in the next application as well. Should a module still contain a bug then fixing this bug will benefit all applications that use the module. Initially modular design requires extra development effort, however the increase in quality and the potential of reusing the module will reduce the application development effort.

3 Requirements

It is a requirement that documentation be provided with every module. This section describes the information that the documentation must include, and the other files needed, in addition to the source code, to simulate and implement the module.

3.1. Description of the Module

A paragraph in the documentation must state the purpose of the module and provide an overview of its function. Say what it does but not how it works, for example state the algorithm used but don’t explain it in detail here. This text can also be in the top level VHDL file.

3.2. List of Files

The documentation is expected to include a list (and brief description) of all source files (VHDL source code, SDC, TCL scripts and data files) needed to synthesize the module. Make it clear which file contains the top level entity. A script file to automatically add the source files to the project is strongly recommended. For an Altera project this would be a QIP file.

3.3. Hardware Interface

The documentation must explain how to connect the module in the design. A table of the top level ports and generics should include:

- a description of the ports (signals)
- a description of the generics (parameters)

State the range of generics over which the design can be synthesized and has been verified. A timing diagram can illustrate signal behavior if this is not clear from the text. In most cases the waveforms created by running the test bench (see below) serve this purpose.

3.4. Software Interface

If the module contains an interface to a microprocessor or controller, the documentation is expected to describe the module as seen by the software. The minimum requirement is a memory map and bit level descriptions of the software registers in the module.

3.5. Test Bench

A test bench must be provided. This illustrates the behavior of the external signals and gives an example of how to instantiate the module. Include a ModelSim .do or .mpf file to automatically compile the test bench and its dependencies, and run it. Also include a wave.do file to display the relevant waveforms. Optional, but highly recommended, is to make the test bench self checking.

For modules that include a physical interface, include a reference design to show that it works.

3.6. SVN Conventions

The SVN repository for the UniBoard project is available via [3]. Follow the established directory structure for modules and designs. The documentation can be placed in the SVN under `<modulename>/doc` or on a wiki. If it is on a wiki, make a `<modulename>/doc/readme.txt` file to say where it can be found. Both UniBoard and UniBoard^2 have a wiki [3, 4].

The SVN `trunk/` contains the latest version of the code. Modules on the trunk can be (and are) re-used, but may occasionally break when the source is updated. Creating occasional tagged versions of stable code would ease reuse however the disadvantage is that it requires extra documenting to identify the tagged versions and could mean a larger effort in case one needs to upgrade to the latest version at once.

4 Recommendations

4.1. Extended Documentation

The designer is free to provide additional explanation, both for his own convenience and that of others, as to how the module works. While many modules can be re-used without reading the VHDL, the code still needs to be maintained and sometimes modified either by the original designer or someone else. Additional text and diagrams to explain the VHDL can help with this, as can a clear coding style.

4.2. Coding Style

For re-use the code can be treated as a black box, however for development and maintenance of code it is essential to use a proper coding style. There are several ways of writing correct VHDL. Some engineers use a low level style where every wire is defined, whereas others prefer a higher level style where it is left to the synthesis tools to infer structures such as state machines and memory blocks. The latter style can be more portable between different vendor's FPGAs, but gives less control over the implementation. Engineers working at different locations will use different conventions for signal names, capitalization and so on. Thus it was decided not to impose a strict coding style across the project. An example of a VHDL coding style that helps to write proper code is described in [1] and [2].

4.3. Bug Reporting

The Redmine issue tracking software was used during the UniBoard project to keep track of hardware and firmware problems and their solutions. The Redmine issue tracking for the UniBoard project is available via [6]. It is recommended to continue using this, or similar software, in UniBoard^2. When a designer fixes a bug or modifies a module, the issue tracker should be used to disseminate the information to others using the module.

Copyright