BEAM

Dune NVMe Storage Project

Directory layout and Coding Styles

Project	DuneNvme
Date	2020-04-29
Reference	DuneNvmeStorageProject
Version	2
Author	Dr Terry Barnaby

Table of Contents

1. Introduction

This document describes the project directory and files layout for Beam FPGA projects along with information on the programming styles we use.

We typically use VHDL or Verilog and the FPGA hardware description language and ‘C++’ and Python for the software programming languages for various vendors FPGA’s. We have used the following tools for the Dune NvmeStorage project.

Operating System	Fedora 31	
VHDL simulation	ghdl and Xilinx Vivado 2019.2.1	
VHDL synthesis	Xilinx Vivado 2019.2.1	
C++ compiler	Gcc 9.3.1	
Python	Python 3.7.6	

2. Directory Layout

The following shows a typical FPGA core's directory tree. It includes all of the source code, test code and build directories to build the FPGA bit file and test software for the generated core. We use the Make system to manage building both the software and FPGA bitfiles.

DuneNvmeStorageTest

├── Makefile	Overall build control makefile
├── doc	Documents
│ ├── Readme.pdf	Readme source for subtree
│ └── *	General document files
├── docsrc	Document source files
│ ├── Makefile	Makefile to build and install documents including doxygen
│ └── *	General document source files
├── src	VHDL/Verilog HDL source code
│ ├── *.vhd	VHDL source files
│ ├── *.xdc	FPGA Constraint files
│ ├── ip	FPGA IP core definition source files
│ │ └── *.xci	FPGA IP core definition source files
├── test	Test software
│ ├── Makefile	Test software overall Makefile for build
│ └── *.[h,c++]	Test software source files
│ ├── bfpga_driver	The Beam bfpga FPGA Linux driver for Xilinx XDMA PCIe core
│ │ ├── Makefile	The bfpga software build Makefile
│ │ ├── *.[hc]	Bfpga 'C' source files
├── sim	Simulation of FPGA hardware
│ ├── Makefile	Build and run test simulation
│ ├── testbench	Simulation testbench sources and output files
│ │ ├── test*.vhd	
│ │ ├── test*.sav	
│ ├── bin	Tool program directory
│ ├── isim	Xilinx isim working directory
│ ├── simu	Working directory
│ └── work	ghdl working directory

├── tools	Tools directory
│ ├── pciRescan	Rescan PCIe bus for PCIe devices
│ └── pciReset	Reset Bfpga FPGA device
└── vivado	Vivado synthesis build
├── Makefile	FPGA synthesis build makefile
├── vivado.mk	FPGA build makefile rules
├── rev	Directory to hold final build bit files
└── *	Vivado working build directories and files

3. Makefiles

We use the GNU “make” system to manage the building of both software and FPGA bitfiles from the source code. All of the “Makefiles” have the following major targets along with other specific targets:

- all: Builds the software/FPGA bit file
- clean: Cleans all of the build files
- distclean: A more fuller clean of the directory tree
- install: Install to appropriate location
- program: Program an FPGA with a bitfile

4. Vivado Builds

We also use the make system to build FPGA bit files. In the vivado directory there is a master Makefile that describes the various source file used for the build and uses the vivado.mk makefile rules file to generate the FPGA bitfile from the sources. The build tree is compatible with the vivado GUI. So once the build system, has generated the Vivado projects <project>.xpr file the vivado GUI can be run using that project definition file.

If new Xilinx IP cores are generated, the make target sync_ip can be used to copy the *.xci files into the main source tree location.

The “rev” directory contains previous built bitfiles.

Apart from the makefile and vivado.mk all other files in the vivado directory are temporary files created by the vivado system.

5. Software Style

Beam uses an object oriented style for all of its software designs. We have build an extensive portfolio of tools and software libraries to help us with this. Some of our key policies include:

5.1. Object orientated Modules

We strongly design the system using a modular structure. Where possible we design these modules/classes for re-use and where appropriate add them to our standard libraries so they can be further developed and used in other projects.

We use a consumer/producer style when designing modules. When we design a module we design it for the consumer as first priority. That is we consider what the user of the module would want from it and provide an API suited to the users needs. The API will use function and data names orientated to the users domain. The modules internals will use function and data names as suited for internal use only.

Where possible we abstract functional design to higher levels and hide the details of operation in lower level modules.

All primary user functionality functions would be listed first and the more in-depth, internal functions listed last so the core functionality is more obvious to see.

5.2. Descriptions and Comments

At the top of each file there should be a description of the file and its contents. This should at a minimum describe:

- The files name.
- Short Title describing its use.
- The company and author.
- The date the file was originally created.
- Any short copyright notices possibly pointing to an overall code copyright/license description file.
- A short description on what the code does, should be used for and special notes on its operation.

An example for ‘C++’:

```
/*****************************************************************************
 *    test_nvme.cpp     Test of FPGA NVME access over PCIe DMA channels
 *    Fred.Jones, Beam Ltd, 2020-03-01
 *    Copyright (c) 2020 All Right Reserved, Beam Ltd, http://www.beam.ltd.uk
 *****************************************************************************
 *
 * This is a simple test program that uses the Xilinx xdma Linux driver to
 * access an Nvme device on a KCU105 with the test009-nvme bit file running.
 *
 */
```

In code comments should be placed where needed to provide notes on the operation. However the code and the function/variable names used should be as self documenting as possible.

Code comment start markers should match the doxygen comment tag style so that automatically generated documentation can be used.

```
/*****************************************************************
 *    test_nvme.cpp Test of FPGA NVME access over PCIe DMA channels
 *    Fred.Jones, Beam Ltd, 2020-03-01
 *****************************************************************
 */
/**
 * @file          test_nvme.cpp
 * @class         Control
 * @author        Fred Jones (fred.jones@beam.ltd.uk)
 * @date          2020-03-13
 * @version       0.0.1
 *
 * @brief
 * This is a simple test program.
```

```
 *
 * @details
 * Some details on this.
 */
```

For class definitions use the following syntax above the class name:

```
/// Text
```

and for in-line comments:

```
///< Text
```

5.3. Naming Convention

We generally use CamelCase for the names of items (capital letters separate words) while sometimes using lower case with an underscore separator when we are producing code that will work with existing code that uses this naming style.

- All type names and constants start with a capital letter.
- All object/variable instances and function names start with a lower case character.
- All pre-processor macros are in upper case.
- All names should be meaningful, but not to long, to help with documenting the operation of the code.
- All object instance variables should start with the letter “o” to differentiate them from function arguments which often would have the same basic name.

5.4. Types

All user type names begin with a capital letter. Our standard beam-lib libraries provide fixed bit width types suitable for micro-controllers as well as large desktop/server processing systems. These include types such as:

- BUInt32 32 bit unsigned integer
- BInt32 32 bit signed integer
- BUInt8 8 bit unsigned byte

5.5. Indents and Spacing

We use the tab character for intents with editors normally set for 8 spaces per tab, but the user can control this using their editors controls based on their viewing needs. We try and keep the level of indents relatively low to aid code readability, moving code to functions where the depth would become excessive.

We try and keep vertical spacing the the minimum, so more code can be seen on a screen. An example C++ member function would look like:

```
BUInt32 MyClass::subtractValues(BUInt32 a, BUInt32 b){
	if(a > b){
		return a – b;
	}
	else {
```

```
                return 0;
        }
}
```

Blocks start with the ‘{‘ character on the end of the line starting the block. Then end with a ‘}’ lining up with the indent level of the whole block.

Spaces are used to separate operator characters and comma characters, but no spaces are used around brackets to reduce horizontal spacing and to make code more readable (in our eyes!).

5.6. File names

All classes have a “*.h” interface description file and a “*.c++” implementation file with the same name as the class. Generally only one class is define in a file, but if the class is part of a larger module set, there may be dependent classes with the same file set.

5.7. Classes

All classes should have a constructor to initialise all member data items. They may also have an init() function that is used to initialise the runtime state of the objects created at a later stage than object creation.

Generally all data members and all internal functions should be set to private or protected.

6. VHDL Style

As we program in many software languages as well as in HDL’s we try and keep similar “programming” styles for both. Unfortunately VHDL is an old and limited language which makes this difficult to achieve. It is also a case insensitive language, but that does not discount the use of character case to enhance readability.

6.1. Modules

We strongly design the system using a modular structure. Where possible we design these modules for re-use and where appropriate add them to our standard libraries so they can be further developed and used in other projects.

We use a consumer/producer style when designing modules. When we design a module we design it for the consumer as first priority. That is we consider what the user of the module would want from it and provide an API suited to the users needs. The API will use generic constants and signal port names orientated to the users domain. The modules internals will use constants, variables and signa names as suited for internal use only.

Where possible we abstract functional design to higher levels and hide the details of operation in lower level modules.

All primary user ports would be listed first and in order of functionality. Where possible signal sets will be abstracted into records.

6.2. Naming Convention

We generally use CamelCase for the names of items (capital letters separate words) while sometimes using lower case with an underscore separator when we are producing code that will work with existing code that uses this naming style.

- All type, component names and constants start with a capital letter.
- All signal/variable instances and function names start with a lower case character.
- All global constants are in upper case.
- All names should be meaningful, but not to long, to help with documenting the operation of the code.
- All ports should have names that reflect their status such as input and output ports.
- Names can end with: “_n” for inverted logic signal, “_i” for input, “_o” for output, “_l” for a local signal that will be output.
- All record and subtype names should end with “Type” to differentiate them as well as start with a capital letter.

6.3. State Machines

All state machines should use a StateType to define the state machines state. The state variable/signal should include the text “state”.

6.4. Indents and Spacing

We use the tab character for intents with editors normally set for 8 spaces per tab, but the user can control this using their editors controls based on their viewing needs. We try and keep the level of indents relatively low to aid code readability, moving code to functions where the depth would become excessive.

We try and keep vertical spacing the the minimum, so more code can be seen on a screen. An example VHDL Process would look like:

```
-- Process register access
process(clk)
begin
      if(rising_edge(clk)) then
            if(reset = '1') then
                  reg_control <= (others => '0');
                  reg_test1   <= (others => '0');
                  reg_test2   <= (others => '0');
                  reg_test3   <= (others => '0');
                  reg_test4   <= (others => '0');
                  reg_test5   <= (others => '0');
                  state       <= STATE_START;
            else
```

```
                    case(state) is
                    when STATE_START =>
                          axil1Out.arready  <= '0';
                          axil1Out.rvalid   <= '0';
                          axil1Out.awready  <= '0';
                          axil1Out.wready   <= '0';
                          state             <= STATE_IDLE;
                    end case;
               end if;
          end if;
     end process;
```

Blocks start with the ‘begin‘ text on the next line starting the block. Then end with a ‘end <type>;’ lining up with the indent level of the whole block.

Spaces are used to separate operator characters and comma characters, but no spaces are used around brackets to reduce horizontal spacing and to make code more readable (in our eyes!).

6.5. File names

All modules have a “*.vhd” file with the same name as the VHDL module/component.

6.6. Packages

Where possible we use higher level abstraction for signal sets etc. We normally define records and types in packages for this purpose. There are normally two packages in use:

- External definitions package: This defines the records, types, constants and functions for a top level module/component that will be used by external systems. This may be defined by the external system itself.
- Internal definitions package: This defines the records, types, constants and functions for the internal modules/components of the tope level module/component. External systems do not see or use these.

6.7. Descriptions and Comments

At the top of each file there should be a description of the file and its contents. This should at a minimum describe:

- The files name.
- Short Title describing its use.
- The company and author.
- The date the file was originally created.
- Any short copyright notices possibly pointing to an overall code copyright/license description file.
- A short description on what the code does, should be used for and special notes on its operation.

An example:

```
-------------------------------------------------------------------------------
--    NvmeStorage.vhd   Nvme storage access module
--    Fred.Jones, Beam Ltd.   2020-02-28
```

```
--------------------------------------------------------------------------------
--!
--! @class        NvmeStorage
--! @author       Fred Jones (fred.jones@beam.ltd.uk)
--! @date         2020-03-13
--! @version      0.0.1
--!
--! @brief
--! This is a very basic module.
--!
--! @details
--! This is some more detailed description.
--!
```

In code comments should be placed where needed to provide notes on the operation. However the code and the function/variable names used should be as self documenting as possible.

Code comment start markers should match the doxygen comment tag style so that automatically generated documentation can be used.

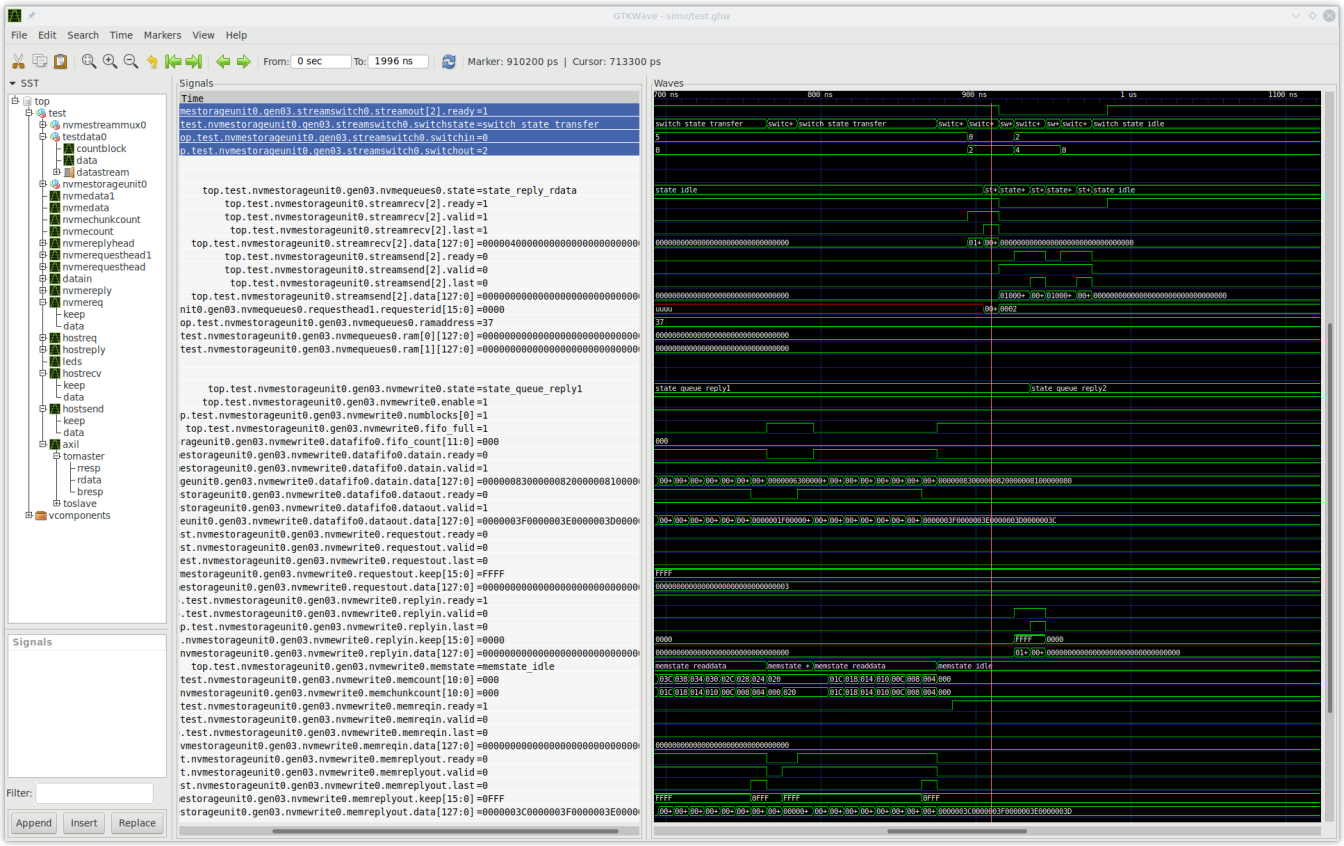
```
--! Mux first input
```

7. VHDL Simulation with ghdl and gtkview

We use ghdl and gtkview for VHDL simulation during development. This provide an easy to use Opensource simulation environment. The directory sim contains the simulation system that uses the actual source files in src.

A “make” based system us used to build, run and view the simulation results.

BEAM


GTKWave - simu/test.ghw
File Edit Search Time Markers View Help
From: 0 sec
To: 1996 ns
Marker: 910200 ps | Cursor: 713300 ps
SST
top
test
nvmestreammux0
testdata0
countblock
data
datastream
nvmestorageunit0
nvmedata1
nvmedata
nvmechunkcount
nvmecount
nvmereplyhead
nvmerequesthead1
nvmerequesthead
datain
nvmereply
nvmereq
keep
data
hostreq
hostreply
leds
hostrecv
keep
data
hostsend
keep
data
axil
tomaster
rresp
rdata
bresp
toslave
vcomponents
Signals
Filter:
Append
Insert
Replace
Signals
Time
Waves