PandABlocks		diamond

PandABlocks
FPGA Development Framework

Author: Isa Uzun | Senior FPGA Development Engineer
Diamond Light Source

Revision History		
Author	**Date**	**Description**
Isa Uzun	27/03/2017	Initial release

Contents

1. INTRODUCTION

This document describes the PandABlocks FPGA development framework targeting the Zynq-based PandABox hardware platform. It provides steps to follow for 3rd-parties to develop and integrate custom functionalities into the framework and run on target hardware.

2. SYSTEM ARCHITECTURE

FPGA firmware and TCP-Server comprise the core of the PandABlocks system architecture implemented on the Xilinx Zynq FPGA device as shown in Figure 1. These two layers are tightly coupled through a common set of configuration files as described in Section 5. Successful operation of PandABlocks on target hardware requires proper management of these fileset.

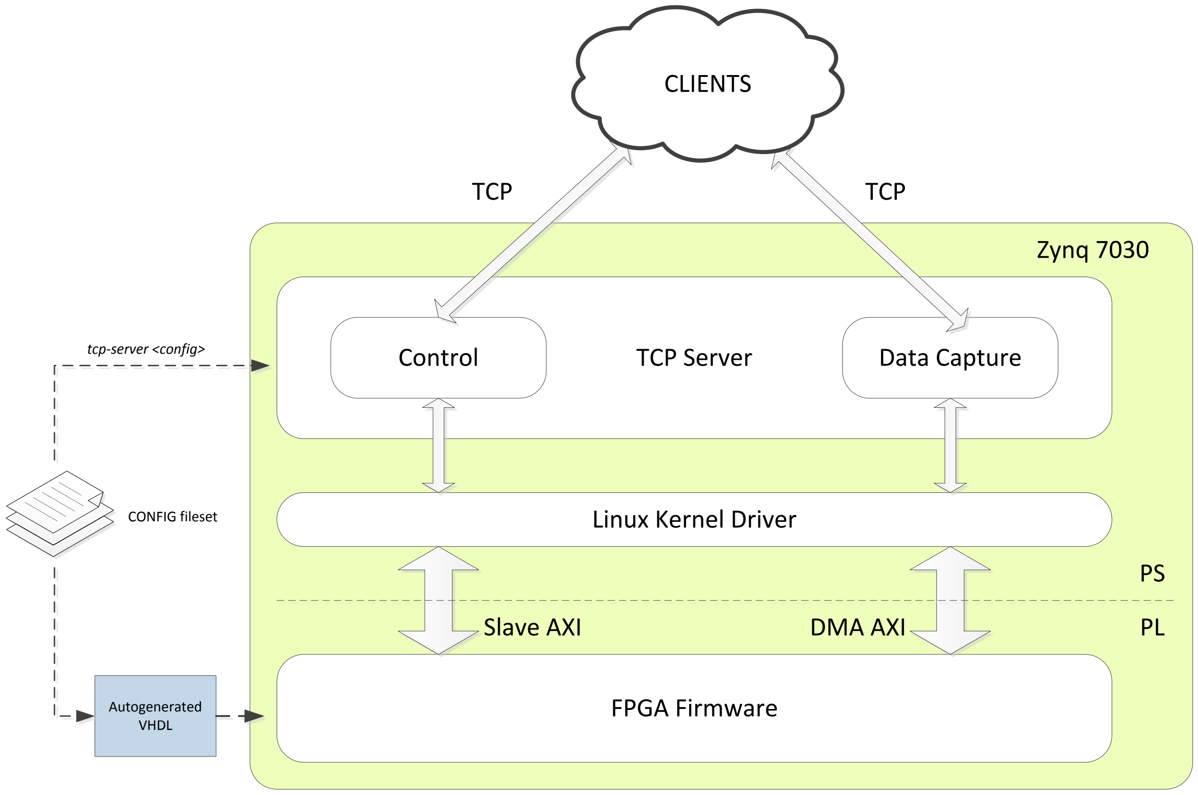


Figure 1 Firmware and TCP-Server on Zynq 7030 FPGA.

3. PANDABOX HARDWARE ARCHITECTURE

Figure 2 shows a high-level block diagram of the PandaBox hardware [1]. There are two FPGA devices on its carrier board:

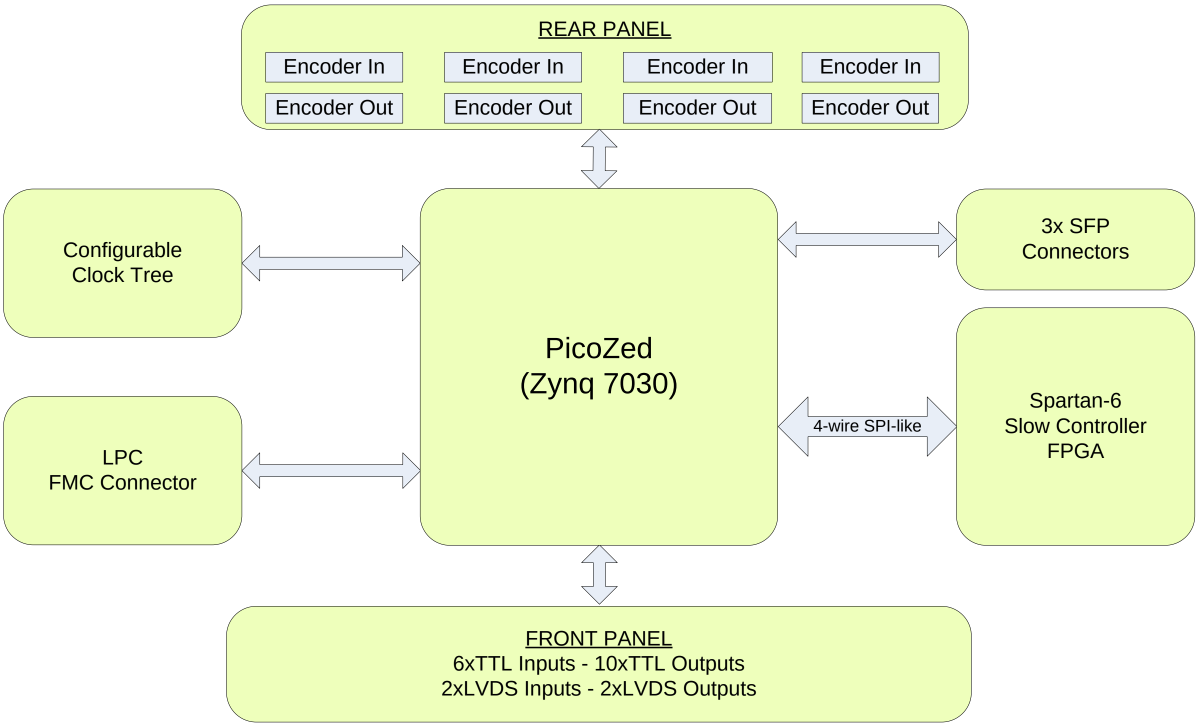


Figure 2 PandA hardware peripherals

Picozed Zynq 7030: implements all the main capabilities at the firmware level. It interfaces to main I/O ports, implements logical blocks and data capture functionality.

Spantan-6: serves for housekeeping purposes and interfaces to slow I/O ports such as encoder control, status LEDs and temperature monitoring.

These 2 FPGAs are connected via 4-wire SPI-like interface using a custom protocol where data flows in both directions.

NOTE: SlowFPGA firmware is out-of-scope of this document.

4. PANDABLOCK FIRMWARE FRAMEWORK

PandABlocks is the firmware framework that can target any Zynq-based hardware platform. The framework is composed of Functional Blocks (FB). A Functional Block is a VHDL module performing a specific task such as Look-Up Table, Pulse Generation, Divider and Counter. A FB is controlled and interrogated through a number of registers.

User-defined number of instances of functional blocks are compiled and built together for an application-specific FPGA firmware to run Zynq device.

A functional block has [2]:

- Discrete bit-type input and output ports,
- 32-bits wide position-type input and output ports,
- Configuration and status registers.

A simple block diagram of a generic FB is shown in Figure 3.

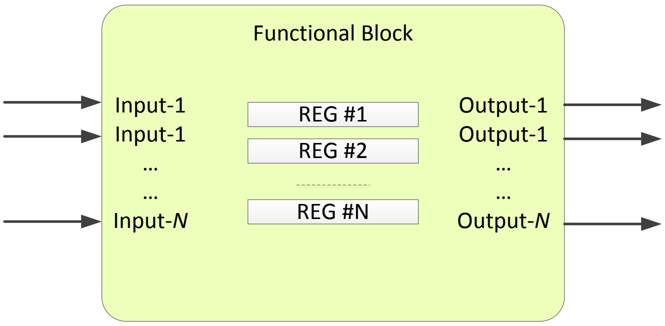


Figure 3 FB block diagram.

Wiring of FBs is achieved via a common system bus and position bus routed internally across the design. System bus feeds functional block’s bit-type inputs while position bus feeds the position-type inputs. Configuration and monitoring of FBs are achieved via a registers mapped to processor memory address space.

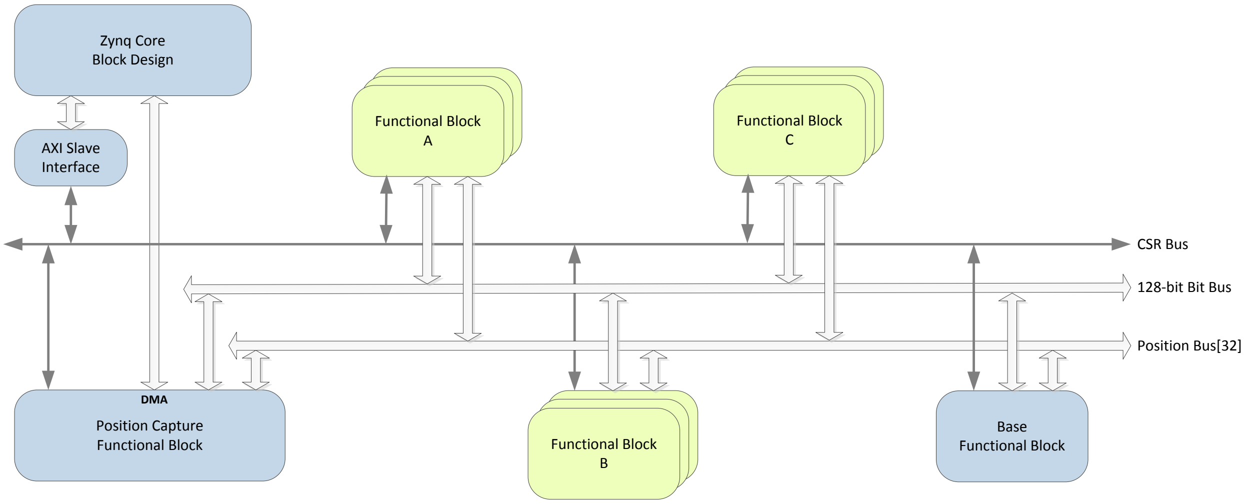


Figure 4 PandABlocks framework architecture

Figure-4 shows the overall architecture of PandABlocks framework with following components.

- Zynq Core Block Design: is hardware specific implementation of Zynq ARM core and AXI interconnect logic.

- Base Functional Block: is hardware specific block in order to provide physical IO interfacing available on the carrier board such as TTL, LVDS and Encoders.
- Position Capture Functional Block: implements dedicated DMA-based data capture logic for all bit and position type data fields in the system. This block is tightly coupled with Linux kernel driver.
- Generic Functional Blocks: are VHDL modules implementing user-defined configurable functionalities. There can be multiple instances of these FBs.

4.1 Functional Block VHDL Module Structure

PandABlocks employs a standard module structure for the design of its functional blocks following bottom-to-top hierarchy as shown in Figure-5. 3rd party developers wishing to add own FBs into the framework should obey this structure for easy integration. A FB is composed of 3 VHDL modules as below.

- <FB>.vhd: This file implements the "function core" such as pulse generation, counter etc. The entity has direct port declarations for physical input and outputs, configuration parameters and status information. It doesn't implement a memory bus interface.
- <FB>_block.vhd: The block file wraps the function core (*<FB>.vhd*) and auto-generated memory interface module (*<FB>_ctrl.vhd*) for parameter and status register access. It also implements bus multiplexers for input ports.
- <FB>_top.vhd: The top level file instantiates required number of functional blocks and implements a thin layer of address decoding logic to target each block individually on the address space.

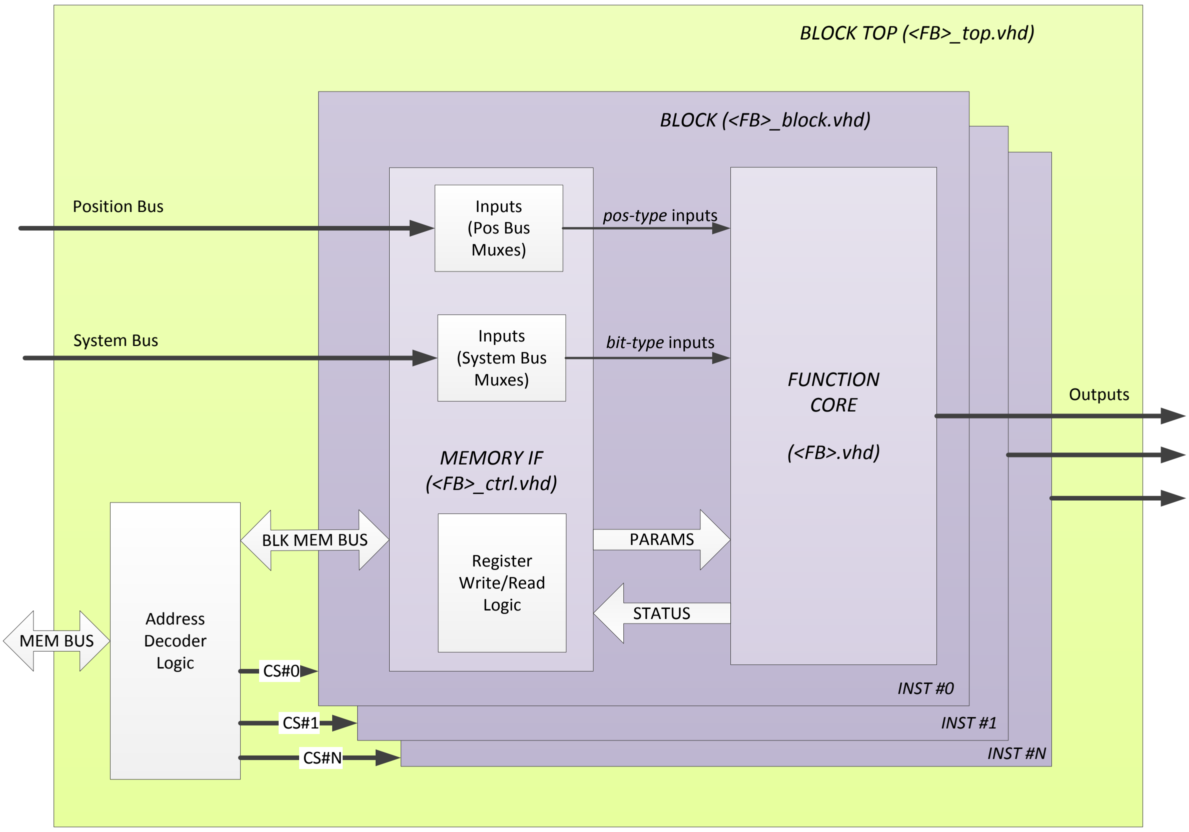


Figure 5 PandABlocks Functional Blocks implementation structure.

The development framework provides VHDL templates for the *<FB>_top.vhd* and *<FB>_block.vhd* modules as shown in the Appendix-A so that developers can focus their efforts to implementing the core FB.

Finally, top-level VHDL design module (*top.vhd*) is composed by instantiating *<FB>_top.vhd* components for target application in-mind.

5. COMMON CONFIG FILESET

CONFIG filesets are used to capture Functional Block’s structure. As an example; SRGATE block has 2 bit-type inputs, 1 bit-type output and 4 configuration registers as depicted in Figure-6.

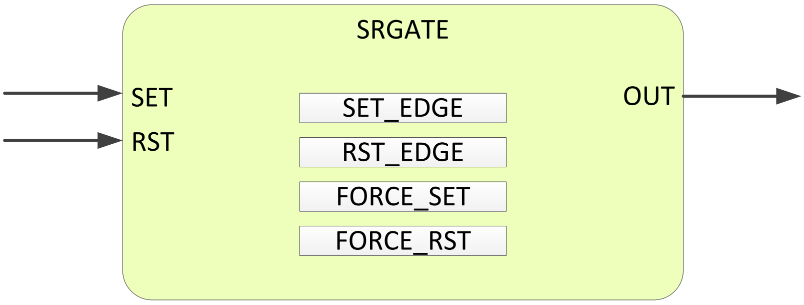


Figure 6 SRGATE block diagram

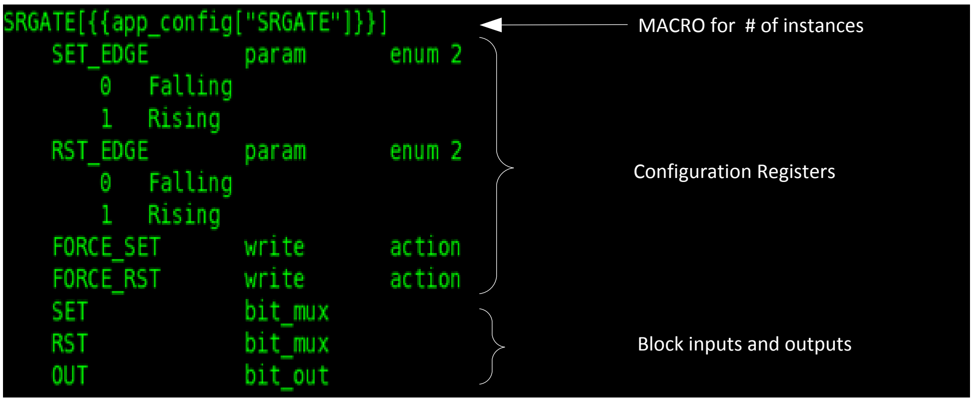
The CONFIG fileset for a functional block consists of 4 individual files as demonstrated in the following subsections for the SRGATE.

5.1 config

This file describes a functional block’s fields including:

- Inputs and Outputs.
- Configuration and status registers.
- Number of instances in the top-level design.

Example below shows *config* file for SRGATE functional block.

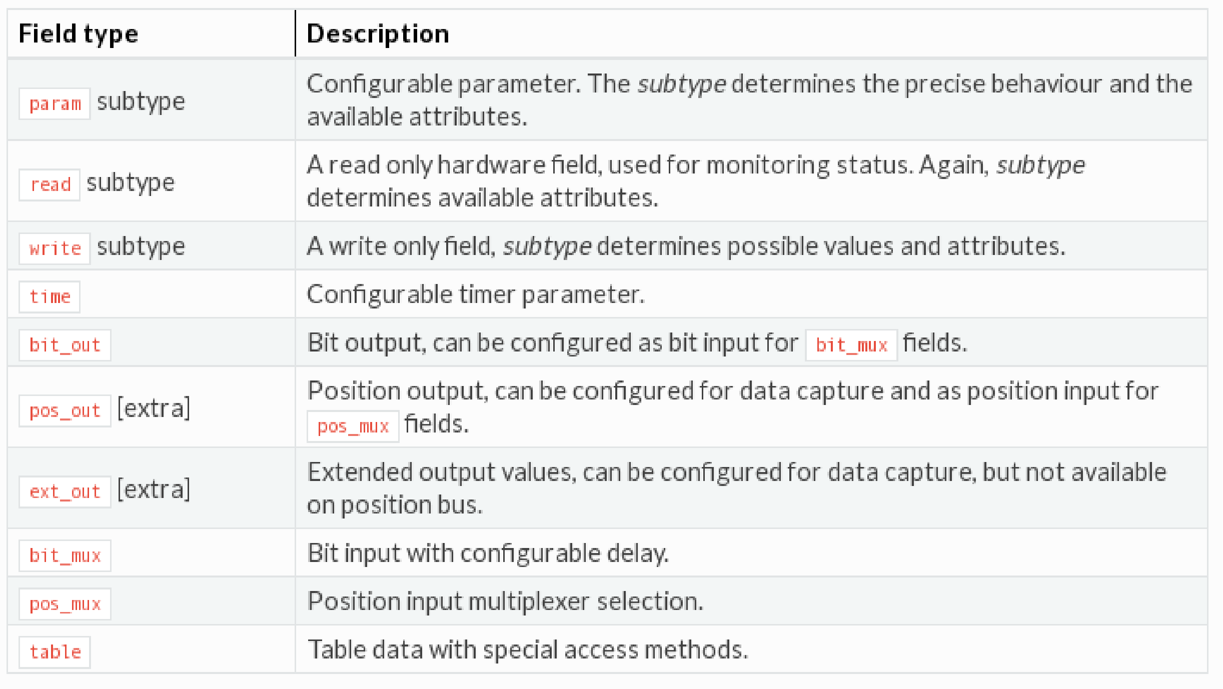


```
SRGATE[{{app_config["SRGATE"]}}]
    SET_EDGE        param       enum 2
        0   Falling
        1   Rising
    RST_EDGE        param       enum 2
        0   Falling
        1   Rising
    FORCE_SET       write       action
    FORCE_RST       write       action
    SET             bit_mux
    RST             bit_mux
    OUT             bit_out
```

Figure 7 config _file for SRGATE block_

The second column in the file above defines the field type for each field. The types of fields PandABlocks framework supports are listed in Table 1. For detailed information regarding field types and their attributes please see reference [2].

Table 1 Supported Field types for Functional Blocks



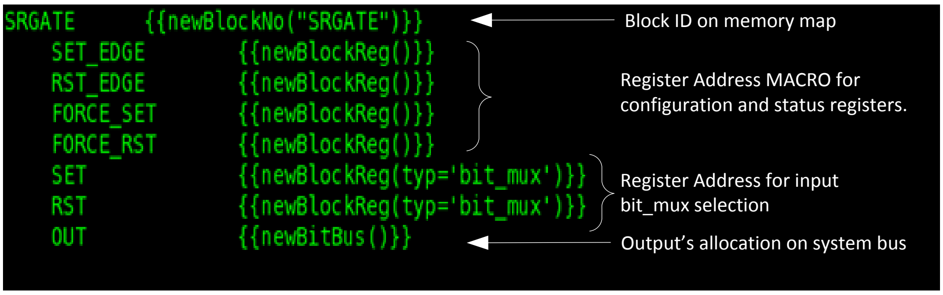
Field type	Description
`param` subtype	Configurable parameter. The *subtype* determines the precise behaviour and the available attributes.
`read` subtype	A read only hardware field, used for monitoring status. Again, *subtype* determines available attributes.
`write` subtype	A write only field, *subtype* determines possible values and attributes.
`time`	Configurable timer parameter.
`bit_out`	Bit output, can be configured as bit input for `bit_mux` fields.
`pos_out` [extra]	Position output, can be configured for data capture and as position input for `pos_mux` fields.
`ext_out` [extra]	Extended output values, can be configured for data capture, but not available on position bus.
`bit_mux`	Bit input with configurable delay.
`pos_mux`	Position input multiplexer selection.
`table`	Table data with special access methods.

5.2 registers

This file defines following information regarding a functional block;

- Register addresses for:
  - Configuration registers.
  - Status registers.
  - Control registers for *bit_mux* type inputs.
- System bus bit index for each output port.

Figure-8 shows an example registers file with MACRO definitions for SRGATE.



```
SRGATE      {{newBlockNo("SRGATE")}}
    SET_EDGE        {{newBlockReg()}}
    RST_EDGE        {{newBlockReg()}}
    FORCE_SET       {{newBlockReg()}}
    FORCE_RST       {{newBlockReg()}}
    SET             {{newBlockReg(typ='bit_mux')}}
    RST             {{newBlockReg(typ='bit_mux')}}
    OUT             {{newBitBus()}}
```

Block ID on memory map

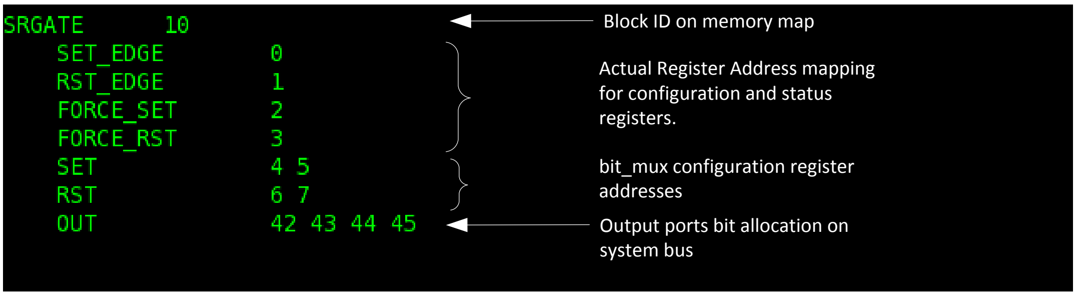
Register Address MACRO for configuration and status registers.

Register Address for input bit_mux selection

Output’s allocation on system bus

Figure 8* registers *file for SRGATE block

This file is then processed within the project build flow and populated with the actual register mapping assignments. The processed registers file is shown in Figure-9.



```
SRGATE      10
    SET_EDGE        0
    RST_EDGE        1
    FORCE_SET       2
    FORCE_RST       3
    SET             4 5
    RST             6 7
    OUT             42 43 44 45
```

Block ID on memory map

Actual Register Address mapping for configuration and status registers.

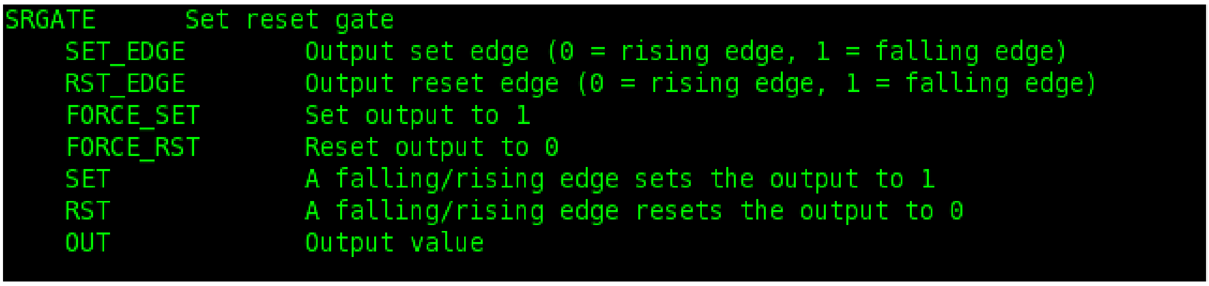
bit_mux configuration register addresses

Output ports bit allocation on system bus

Figure 9 registers file after processing for SRGATE

5.3 description

This file contains short description of each field in the config file.

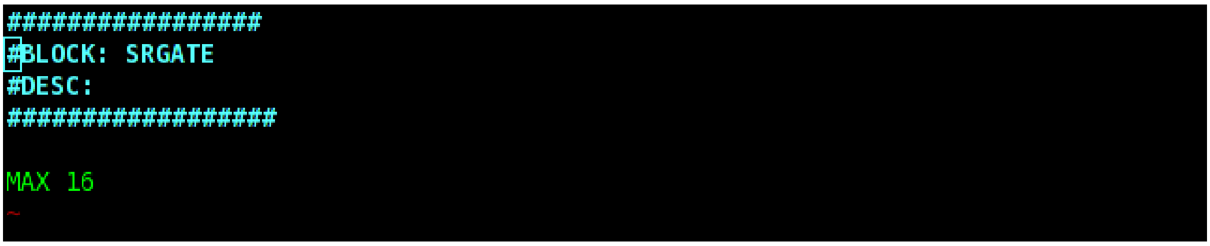


```
SRGATE      Set reset gate
    SET_EDGE        Output set edge (0 = rising edge, 1 = falling edge)
    RST_EDGE        Output reset edge (0 = rising edge, 1 = falling edge)
    FORCE_SET       Set output to 1
    FORCE_RST       Reset output to 0
    SET             A falling/rising edge sets the output to 1
    RST             A falling/rising edge resets the output to 0
    OUT             Output value
```

Figure 10* description *file for SRGATE block

5.4 meta

This file defines the maximum number of instances for the block in the firmware



```
#################
#BLOCK: SRGATE
#DESC:
##################

MAX 16
~
```

Figure 11* meta *file for SRGATE which states that maximum 16 SRGATE block can be instantiated in the top-level design

These set of configuration files for each block are used to auto-generate:

- A global VHDL package files to define register names and addresses for all FBs.
- VHDL Memory Interface modules for individual FBs.
- A VHDL module to combine all bit-type and pos-type FB outputs into two global busses namely system bus and position bus.

The same config fileset is also used as an argument to tcp-server. This way tcp-server is informed about the functional blocks and their attributes available on the FPGA firmware.

6. PROJECT DIRECTORY STRUCTURE

PandABlocks repository contains following sub-directories.

<table>
<tr><th>Sub-directory Name</th><th>Description</th></tr>
<tr><td>common/
|---python/
|---vhdl/</td><td>Common source and build files
• Python scripts used in the build process
• VHDL source files shared between FBs</td></tr>
<tr><td>modules/
|--- <FunctionalBlock>
| |---config/
| |---sim/
| |---vhdl/
|
|--- <FunctionalBlock>
| |---config/
| |---sim/
| |---vhdl/
...</td><td>It contains all sources per Functional Block
• Functional Block sub-directory
○ Config fileset per FB (config, registers and description)
○ Python and FPGA simulation files
○ VHDL source files</td></tr>
<tr><td>apps/</td><td>Firmware application configuration files</td></tr>
<tr><td>targets/
|---PandABox/
|---bd/
|---config/
|---constr/
|---scripts/</td><td>Target hardware platform specific files
• Target hardware
○ Zynq ARM core block design source
○ Vivado BSP
○ FPGA constraints
○ Build scripts</td></tr>
<tr><td>tools/</td><td>Utilities</td></tr>
<tr><td>doc/</td><td>HTML documentation</td></tr>
<tr><td>SlowFPGA/*</td><td>FPGA project files for the Xilinx Spartan-6 Slow Control</td></tr>
<tr><td>build/
|---PandABox/
| |---config_d/
| |---ip_repo/
| |---panda_ps/
| |---panda_top/
|---SlowFPGA/</td><td>Build directory
• Zynq-based target hardware
○ Assembled application specific config fileset
○ Xilinx IP netlists
○ Zynq ARM core netlist
○ Top-level Vivado project output files
• SlowFPGA output files</td></tr>
</table>

* This directory is temporarily kept in this repository and will be moved to PandABox repository on OHW.

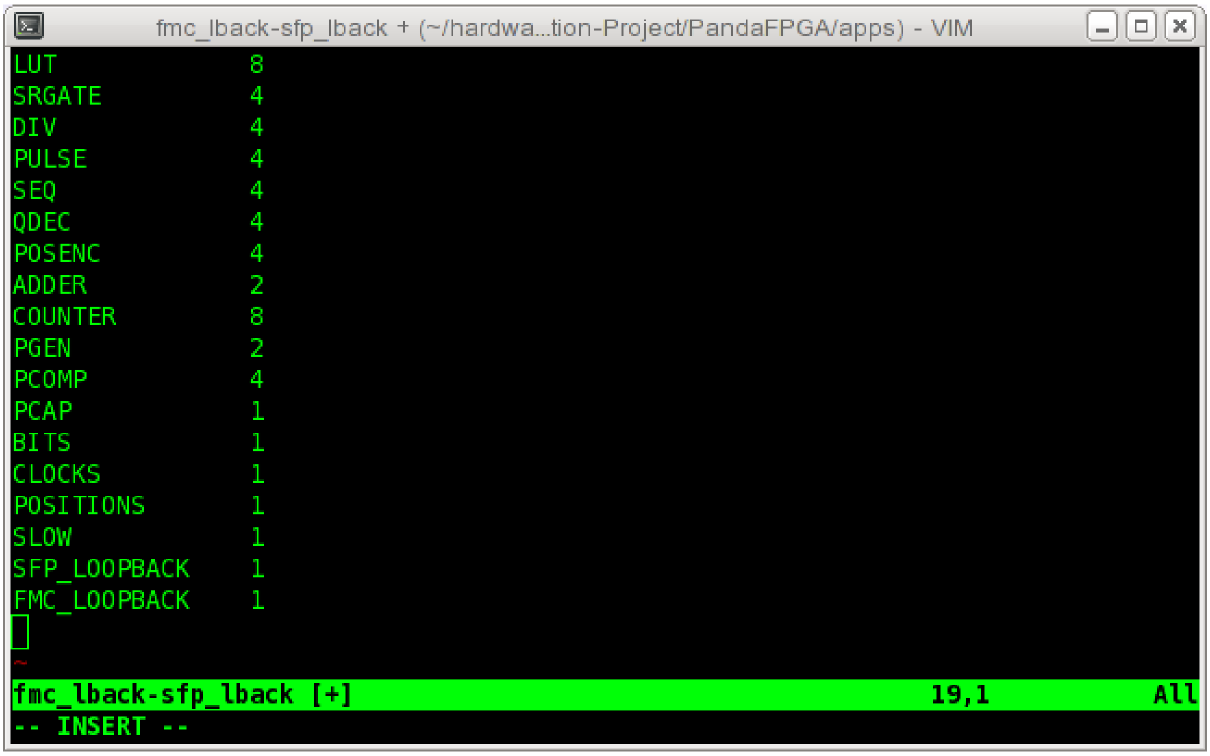
7. BUILDING PANDABLOCKS FIRMWARE

FPGA firmware built process is Makefile driven. It uses multiple python and tcl scripts as explained in following sections. PandaBlocks build flow requires Vivado 2015.1.

Output projects and built files are stored in <*BUILD_DIR*>/<*TARGET*> directory. <*BUILD_DIR*> and <*TARGET*> variables are defined in the CONFIG file. By default TARGET is set to PandABox.

Step-1: Generate Application CONFIG Fileset

Application files, stored in <*apps*> directory, control which functional blocks are included in the firmware built, targeting a particular application. The base PandABox application file is shown in Figure-12 as an example.



fmc_lback-sfp_lback + (~/hardwa...tion-Project/PandaFPGA/apps) - VIM

```
LUT             8
SRGATE          4
DIV             4
PULSE           4
SEQ             4
QDEC            4
POSENC          4
ADDER           2
COUNTER         8
PGEN            2
PCOMP           4
PCAP            1
BITS            1
CLOCKS          1
POSITIONS       1
SLOW            1
SFP_LOOPBACK    1
FMC_LOOPBACK    1

~
fmc_lback-sfp_lback [+]                                  19,1         All
-- INSERT --
```

Figure 12 Application file example. The first column declares the required functional blocks in the firmware and the second column defines the number of instantiations per FB.

The first step in the build process pulls all module-based CONFIG fileset from <*modules*/<*FB*>/*config*> directories and creates a global application-specific config fileset which includes config, registers and description files. The generated config files are stored in <*BUILD_DIR*>/<*TARGET*>/*config_d* directory.

<*TOP_DIR*>*/common/python/config_generator.py* script is called in the Makefile to carry out this step.

Step-2: Auto-Generate Memory Interface VHDL Modules

<TOP_DIR>/common/python/vhdl_generator.py script is used in the Makefile to auto-generate following VHDL files from the information available in the config fileset.

1. *address_defines.vhd* : VHDL package file which defines register names and address for all FBs in the design,
2. *<FB_Name>_ctrl.vhd* : VHDL modules implementing memory interface logic for each FB.
3. *panda_busses.vhd* : VHDL module to combine all bit-type FB outputs into “system bus” and pos-type outputs into “position bus”.

All resulting files are stored in *<BUILD_DIR>/<TARGET>/autogen* directory.

Step-3: Generate Xilinx IPs

All Xilinx IPs used in the design are generated in out-of-context mode using *<scripts/build_ips.tcl>* script. The script includes configuration of all IPs in the design. An example for configuring the “pulse_queue” FIFO IP is shown in Table-2. Any new user IP configuration must be added in this file as required.

This step places all generated IP netlists in the *<BUILD_DIR>/PandABox /ip_repo* directory.

Table 2 Snippet from build_ips.tcl script showing configuration of fifo_generator IP.

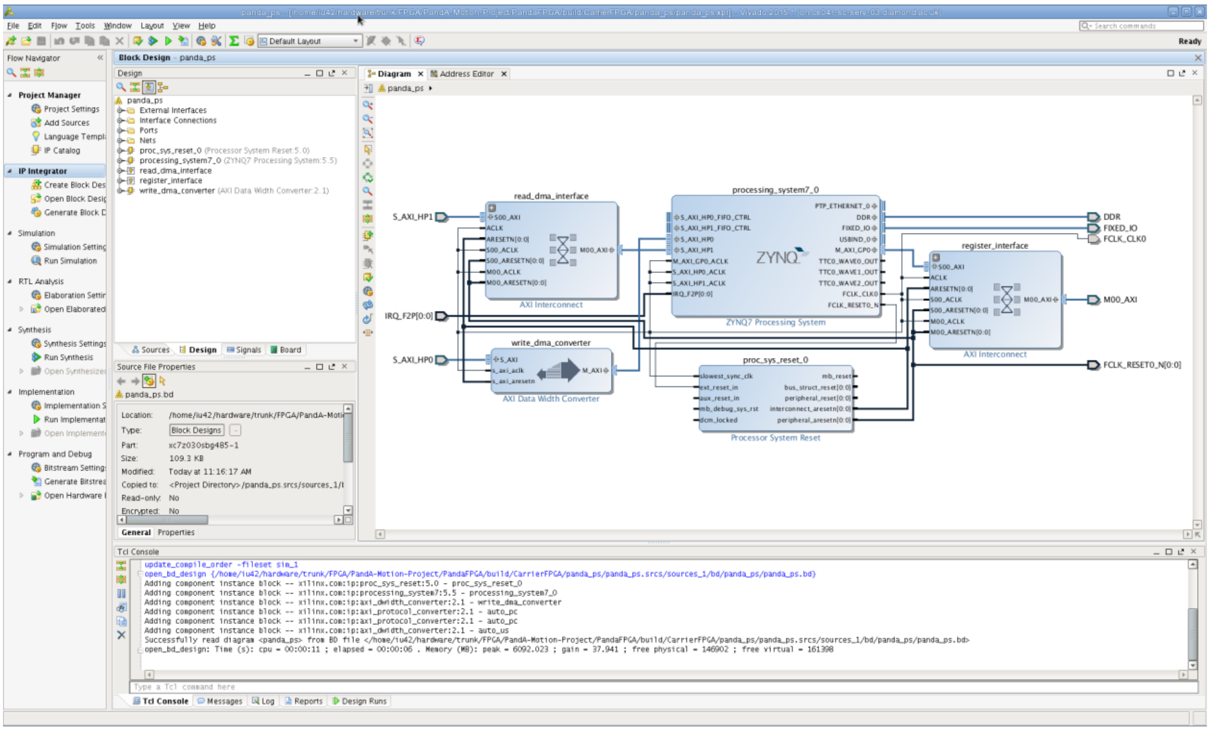
```
#
# Create PULSE_QUEUE IP
#
create_ip -name fifo_generator -vendor xilinx.com -library ip -version 12.0 \
-module_name pulse_queue -dir $origin_dir/

set_property -dict [list \
   CONFIG.Performance_Options {First_Word_Fall_Through} \
   CONFIG.Input_Data_Width {49}    \
   CONFIG.Data_Count {true}        \
   CONFIG.Output_Data_Width {49}   \
   CONFIG.Reset_Type {Synchronous_Reset} \
] [get_ips pulse_queue]

generate_target all [get_files $origin_dir/pulse_queue/pulse_queue.xci]
synth_ip [get_ips pulse_queue]
```

Step-4: Generate Zynq Arm Core

At the heart of the design lies the Zyqn ARM core which is captured as a Vivado Block Design. Block design source file (*panda_ps.tcl*) is located in *PandABox/src/bd/* directory.



```
panda_ps.bd
Location:     /home/iu42/hardware/trunk/FPGA/PandA-Moti
Type:         Block Designs
Part:         xc7z030sbg485-1
Size:         109.3 KB
Modified:     Today at 11:16:17 AM
Copied to:    <Project Directory>/panda_ps.srcs/sources_1/
Read-only:    No
Encrypted:    No
```

```
update_compile_order -fileset sim_1
open_bd_design {/home/iu42/hardware/trunk/FPGA/PandA-Motion-Project/PandaFPGA/build/CarrierFPGA/panda_ps/panda_ps.srcs/sources_1/bd/panda_ps/panda_ps.bd}
Adding component instance block -- xilinx.com:ip:proc_sys_reset:5.0 - proc_sys_reset_0
Adding component instance block -- xilinx.com:ip:processing_system7:5.5 - processing_system7_0
Adding component instance block -- xilinx.com:ip:axi_dwidth_converter:2.1 - write_dma_converter
Adding component instance block -- xilinx.com:ip:axi_protocol_converter:2.1 - auto_pc
Adding component instance block -- xilinx.com:ip:axi_protocol_converter:2.1 - auto_pc
Adding component instance block -- xilinx.com:ip:axi_dwidth_converter:2.1 - auto_us
Successfully read diagram <panda_ps> from BD file </home/iu42/hardware/trunk/FPGA/PandA-Motion-Project/PandaFPGA/build/CarrierFPGA/panda_ps/panda_ps.srcs/sources_1/bd/panda_ps/panda_ps.bd>
open_bd_design: Time (s): cpu = 00:00:11 ; elapsed = 00:00:06 . Memory (MB): peak = 6092.023 ; gain = 37.941 ; free physical = 146902 ; free virtual = 161398
Type a Tcl command here
```

Figure 13 PandABox Zyqn ARM core block design

<scripts/build_ps.tcl> script creates *<BUILD_DIR>/PandABox /panda_ps* project and builds Zynq ARM core netlist.

NOTE: This project can be opened using Vivado in order to modify the Zynq block design. Modified block design must be exported back into the *PandABox/src/bd/* directory.

Step-5: Build Top-Level Design

In the final step, *<scripts/build_top.tcl>* script read all IP netlists, Zynq ARM core netlist and all VHDL design sources in order to create the firmware image. At the end of this step, output firmware image *panda_top.bit* is stored in *<BUILD_DIR>/PandABox*.

Step-6: Generate panda-fpga_<APP>.zpg

Firmware installation onto the target PandABox platform is managed through zpkg files. In this last step, firmware image and configuration files are packaged into *panda-fpga_<APP>.zpg* file. The directory structure of the package file is defined in *<TOP_DIR>/etc/ panda-fpga.list* file.

The *.zpkg file must be transferred onto the target platform and installed as defined in the PandABox Setup and Installation Manual [5].

8. FMC OR SFP BLOCKS

FMC and SFP-based functional blocks are special blocks because:

- They have direct connectivity to physical I/O pins, and
- Their instantiation is limited to 1 FMC FB and 3 SFP FBs on PandABox hardware platform.

PandABlocks already supports three different FMC applications through functional blocks below:

- fmc_loopback: This FB implements loopback on FMC connector for unit testing.
- fmc_24vio: A FB to provide configurable 24V I/O capability through FMC.
- fmc_acq420: 3-rd party FB to interface D-Tacq ACQ420 FMC product.

Each FMC (and SFP) application is managed under its own subdirectory in modules/. For FMC and SFP functional blocks, there is in additional folder (*const*) to manage FMC/SFP application specific constraints.

Table 3 FMC and SFP Functional Blocks Directory Structure

modules/ |--- fmc_*<app#1>* | |---config/ | |---sim/ | |---vhdl/ | |---const/ | |--- fmc_*<app#2>* | |---config/ | |---sim/ | |---vhdl/ | |---const/	It contains all sources per Functional Block • Functional Block sub-directory ○ Config fileset per FB (config, registers and description) ○ Python and FPGA simulation files ○ VHDL source files ○ FMC/SFP specific constraint files

To facilitate the integration of FMC and SFP Functional Blocks into the top-level design, VHDL templates for FMC and SFP components (*sfp_top.vhd* and *fmc_top.vhd*) are provided by the framework as shown in Appendix-B and C.

9. ADDING CUSTOM FUNCTIONAL BLOCKS

PandABlock FPGA Development Kit provides following VHDL templates to help developing and integrating custom FBs. All VHDL template files are located in *<TOP_DIR>/common/vhdl/templates* directory.

- *FB_top.vhd* : Top-level Functional Block entity template,
- *FB_block.vhd* : FB block entity template
- *fmc_top.vhd* : FMC-based FB entity template
- *sfp_top,vhd* : SFP-based FB entity template

The flow diagram in Table-4 shows the steps to follow in order to add a new functional block into the development framework.

10. REFERENCES

[1] PandABox Hardware User Manual

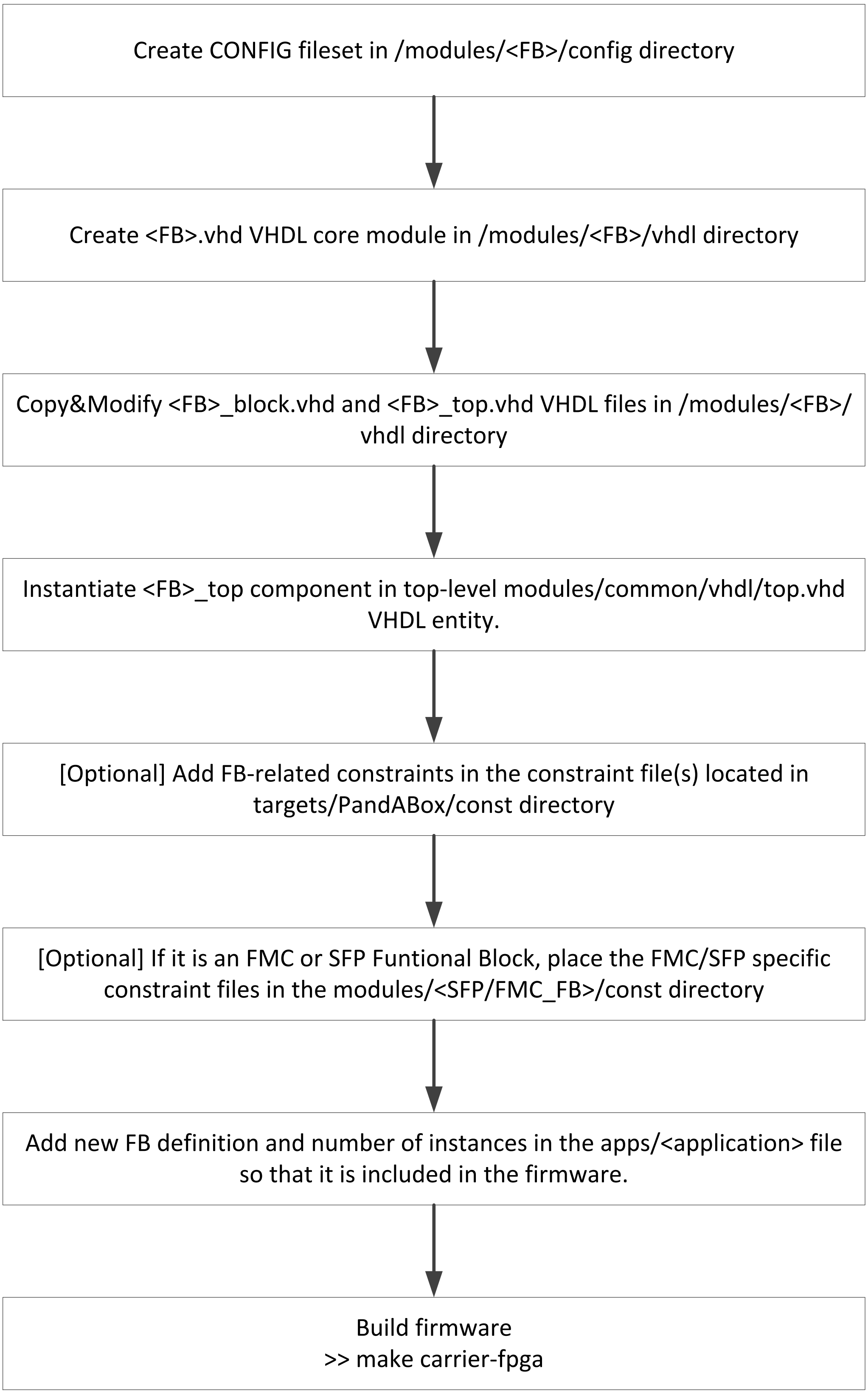
[2] Panda-Server Documentation

[3] PandABox Firmware Design Documentation

[4] PandaBox Register Address Map Documentation

[5] PandABox Setup and Installation Manual

Table 4 Flow for adding new Functional Block into the framework



Create CONFIG fileset in /modules/<FB>/config directory

Create <FB>.vhd VHDL core module in /modules/<FB>/vhdl directory

Copy&Modify <FB>_block.vhd and <FB>_top.vhd VHDL files in /modules/<FB>/ vhdl directory

Instantiate <FB>_top component in top-level modules/common/vhdl/top.vhd VHDL entity.

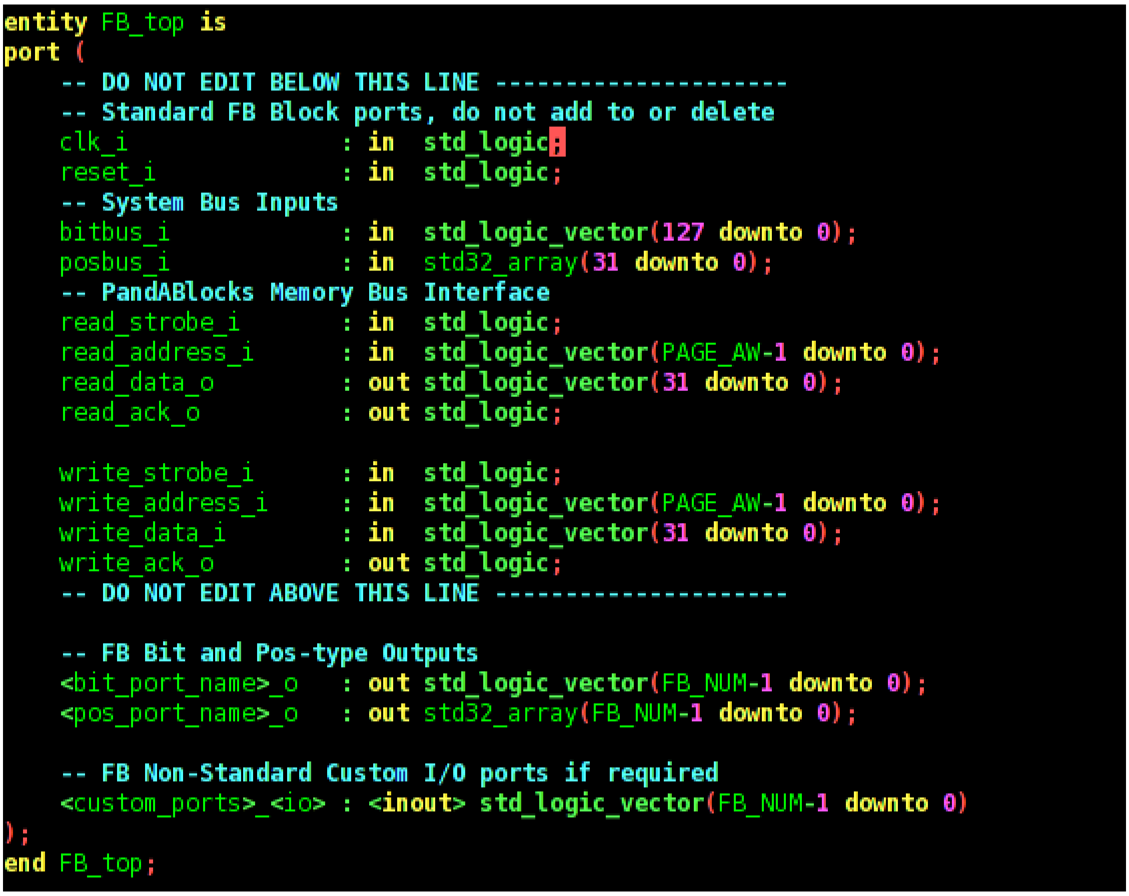
[Optional] Add FB-related constraints in the constraint file(s) located in targets/PandABox/const directory

[Optional] If it is an FMC or SFP Funtional Block, place the FMC/SFP specific constraint files in the modules/<SFP/FMC_FB>/const directory

Add new FB definition and number of instances in the apps/<application> file so that it is included in the firmware.

Build firmware
>> make carrier-fpga

APPENDIX-A: FB_TOP VHDL TEMPLATE



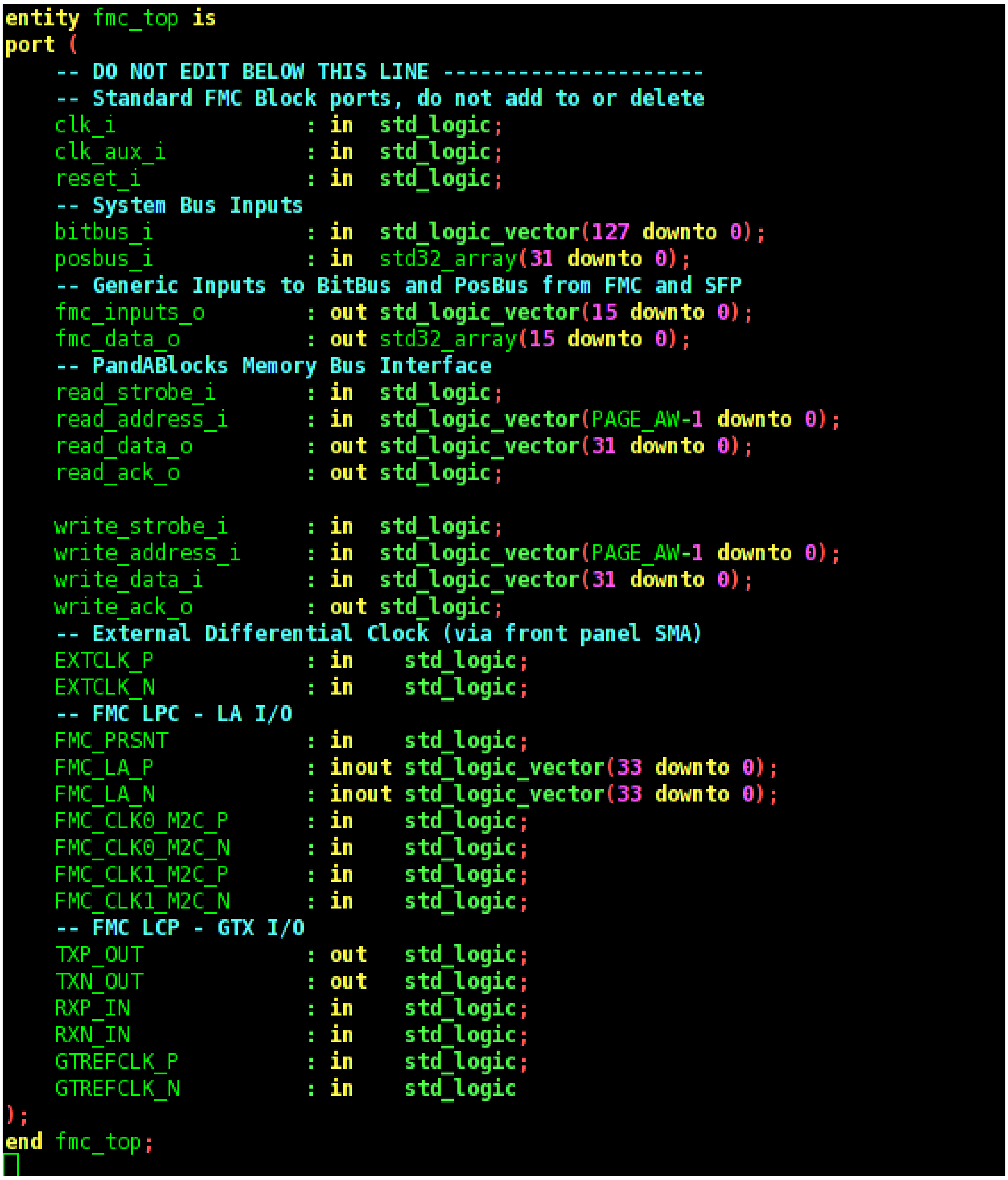
```
entity FB_top is
port (
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Standard FB Block ports, do not add to or delete
    clk_i               : in  std_logic;
    reset_i             : in  std_logic;
    -- System Bus Inputs
    bitbus_i            : in  std_logic_vector(127 downto 0);
    posbus_i            : in  std32_array(31 downto 0);
    -- PandABlocks Memory Bus Interface
    read_strobe_i       : in  std_logic;
    read_address_i      : in  std_logic_vector(PAGE_AW-1 downto 0);
    read_data_o         : out std_logic_vector(31 downto 0);
    read_ack_o          : out std_logic;

    write_strobe_i      : in  std_logic;
    write_address_i     : in  std_logic_vector(PAGE_AW-1 downto 0);
    write_data_i        : in  std_logic_vector(31 downto 0);
    write_ack_o         : out std_logic;
    -- DO NOT EDIT ABOVE THIS LINE ---------------------

    -- FB Bit and Pos-type Outputs
    <bit_port_name>_o   : out std_logic_vector(FB_NUM-1 downto 0);
    <pos_port_name>_o   : out std32_array(FB_NUM-1 downto 0);

    -- FB Non-Standard Custom I/O ports if required
    <custom_ports>_<io> : <inout> std_logic_vector(FB_NUM-1 downto 0)
);
end FB_top;
```

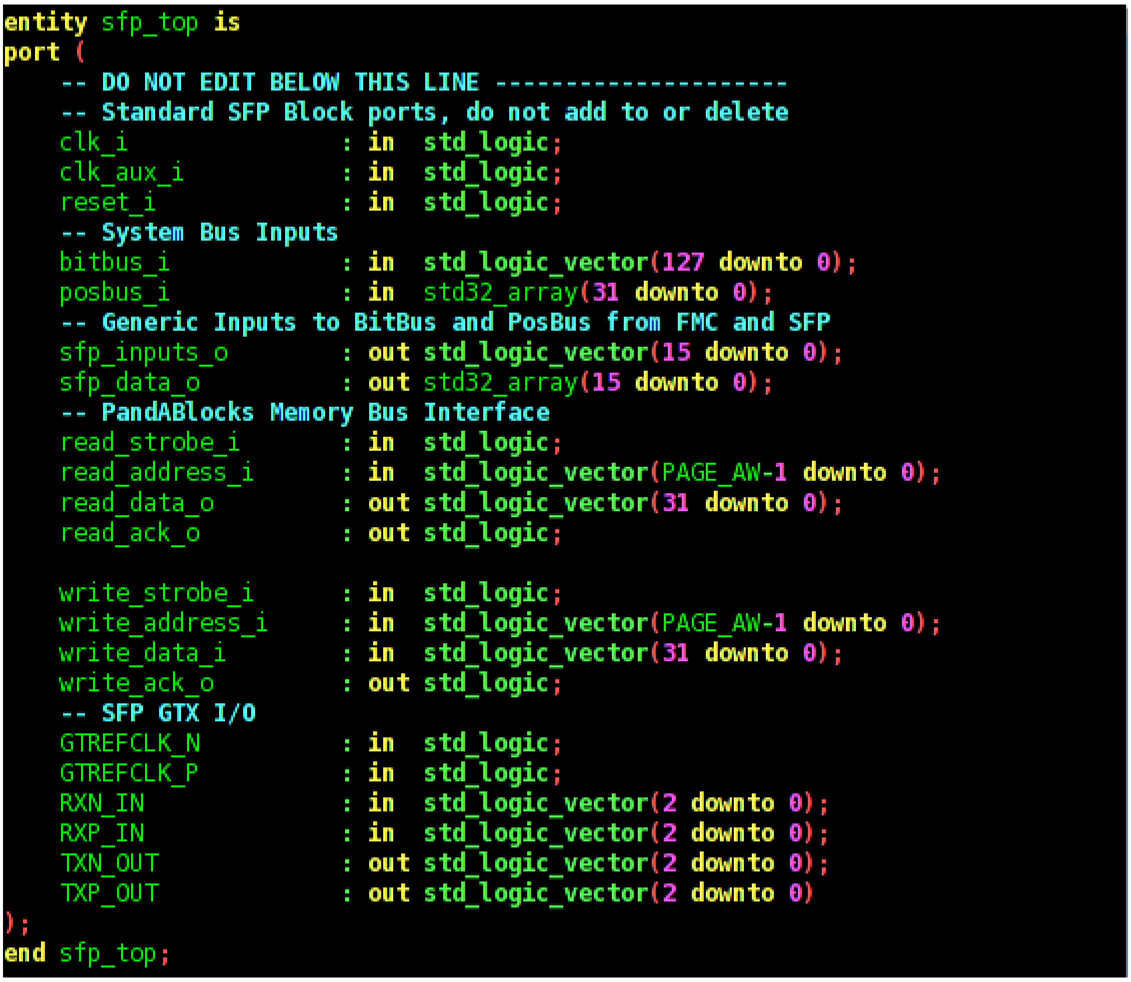
APPENDIX-B: FMC_TOP VHDL TEMPLATE



```vhdl
entity fmc_top is
port (
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Standard FMC Block ports, do not add to or delete
    clk_i               : in  std_logic;
    clk_aux_i           : in  std_logic;
    reset_i             : in  std_logic;
    -- System Bus Inputs
    bitbus_i            : in  std_logic_vector(127 downto 0);
    posbus_i            : in  std32_array(31 downto 0);
    -- Generic Inputs to BitBus and PosBus from FMC and SFP
    fmc_inputs_o        : out std_logic_vector(15 downto 0);
    fmc_data_o          : out std32_array(15 downto 0);
    -- PandABlocks Memory Bus Interface
    read_strobe_i       : in  std_logic;
    read_address_i      : in  std_logic_vector(PAGE_AW-1 downto 0);
    read_data_o         : out std_logic_vector(31 downto 0);
    read_ack_o          : out std_logic;

    write_strobe_i      : in  std_logic;
    write_address_i     : in  std_logic_vector(PAGE_AW-1 downto 0);
    write_data_i        : in  std_logic_vector(31 downto 0);
    write_ack_o         : out std_logic;
    -- External Differential Clock (via front panel SMA)
    EXTCLK_P            : in    std_logic;
    EXTCLK_N            : in    std_logic;
    -- FMC LPC - LA I/O
    FMC_PRSNT           : in    std_logic;
    FMC_LA_P            : inout std_logic_vector(33 downto 0);
    FMC_LA_N            : inout std_logic_vector(33 downto 0);
    FMC_CLK0_M2C_P      : in    std_logic;
    FMC_CLK0_M2C_N      : in    std_logic;
    FMC_CLK1_M2C_P      : in    std_logic;
    FMC_CLK1_M2C_N      : in    std_logic;
    -- FMC LCP - GTX I/O
    TXP_OUT             : out   std_logic;
    TXN_OUT             : out   std_logic;
    RXP_IN              : in    std_logic;
    RXN_IN              : in    std_logic;
    GTREFCLK_P          : in    std_logic;
    GTREFCLK_N          : in    std_logic
);
end fmc_top;
```

APPENDIX-C: SFP_TOP VHDL TEMPLATE



```
entity sfp_top is
port (
    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Standard SFP Block ports, do not add to or delete
    clk_i                 : in  std_logic;
    clk_aux_i             : in  std_logic;
    reset_i               : in  std_logic;
    -- System Bus Inputs
    bitbus_i              : in  std_logic_vector(127 downto 0);
    posbus_i              : in  std32_array(31 downto 0);
    -- Generic Inputs to BitBus and PosBus from FMC and SFP
    sfp_inputs_o          : out std_logic_vector(15 downto 0);
    sfp_data_o            : out std32_array(15 downto 0);
    -- PandABlocks Memory Bus Interface
    read_strobe_i         : in  std_logic;
    read_address_i        : in  std_logic_vector(PAGE_AW-1 downto 0);
    read_data_o           : out std_logic_vector(31 downto 0);
    read_ack_o            : out std_logic;

    write_strobe_i        : in  std_logic;
    write_address_i       : in  std_logic_vector(PAGE_AW-1 downto 0);
    write_data_i          : in  std_logic_vector(31 downto 0);
    write_ack_o           : out std_logic;
    -- SFP GTX I/O
    GTREFCLK_N            : in  std_logic;
    GTREFCLK_P            : in  std_logic;
    RXN_IN                : in  std_logic_vector(2 downto 0);
    RXP_IN                : in  std_logic_vector(2 downto 0);
    TXN_OUT               : out std_logic_vector(2 downto 0);
    TXP_OUT               : out std_logic_vector(2 downto 0)
);
end sfp_top;
```