Wilga Symposium

Automatic management of local bus address space in complex FPGA-implemented hierarchical systems

Wojciech M. Zabołotny[1], Marek Gumiński[1],Michał Kruszewski[1],

[1]Institute of Electronic Systems, Warsaw University of Technology

XLIV-th IEEE-SPIE Joint Symposium Wilga 2019

Introduction

- The data processing systems are often implemented in FPGA as parameterized, complex, multilevel hierarchical systems
- Its configuration and diagnostics requires convenient access to the internal blocks via *control interface*
- Scalable and flexible implementation of the control interface is particularly important in systems developed by many independent teams, or in subsystems designed for reuse. An example may be firmware components and subsystems developed for CBM experiment.

Introduction

- The data processing systems are often implemented in FPGA as parameterized, complex, multilevel hierarchical systems
- Its configuration and diagnostics requires convenient access to the internal blocks via *control interface*
- Scalable and flexible implementation of the control interface is particularly important in systems developed by many independent teams, or in subsystems designed for reuse. An example may be firmware components and subsystems developed for CBM experiment.
- The key questions in such interfaces are:
  - routing of control interface between blocks
  - creating of address maps (assignment of addresses both for HW and SW)

Vendor tools - Xilinx - block design

clk_100MHz
reset_rtl_0

jtag_axi_0
aclk
aresetn
M_AXI
JTAG to AXI Master

jtag_axi_0_axi_periph
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M01_ACLK
M01_ARESETN
M02_ACLK
M02_ARESETN
M03_ACLK
M03_ARESETN
M00_AXI
M01_AXI
M02_AXI
M03_AXI
AXI Interconnect

axi_gpio_0
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
AXI GPIO
gpio_rtl_0

axi_iic_0
S_AXI
s_axi_aclk
s_axi_aresetn
IIC
iic2intc_irpt
gpo[0:0]
AXI IIC
iic_rtl_0

axi_uart16550_0
S_AXI
s_axi_aclk
s_axi_aresetn
freeze
UART
ip2intc_irpt
AXI UART16550
uart_rtl_0

myslave_0
S00_AXI
s00_axi_aclk
s00_axi_aresetn
REG1[31:0]
myslave_v1.0 (Pre-Production)
REG1_0[31:0]

Vendor tools - Xilinx - address allocation

Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
jtag_axi_0					
Data (32 address bits : 4G)					
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_iic_0	S_AXI	Reg	0x4080_0000	64K	0x4080_FFFF
axi_uart16550_0	S_AXI	Reg	0x44A0_0000	64K	0x44A0_FFFF
myslave_0	S00_AXI	S00_AXI_reg	0x44A1_0000	64K	0x44A1_FFFF

Vendor tools - summary

- Well integrated with GUI
- Intuitive user interface
- Automatic assignment of addresses

Vendor tools - summary

- Well integrated with GUI
- Intuitive user interface
- Automatic assignment of addresses
- Poor support for parametrized number of blocks

Internal Interface and Component Internal Interface

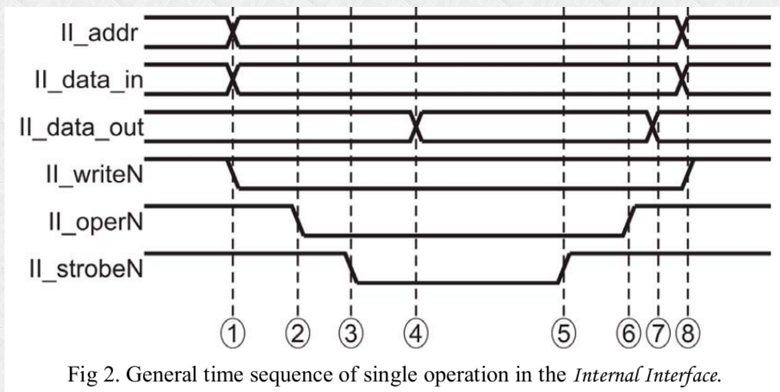


Fig 2. General time sequence of single operation in the *Internal Interface*.

Source: [1]

- VME-like interface inside FPGA

Internal Interface and Component Internal Interface

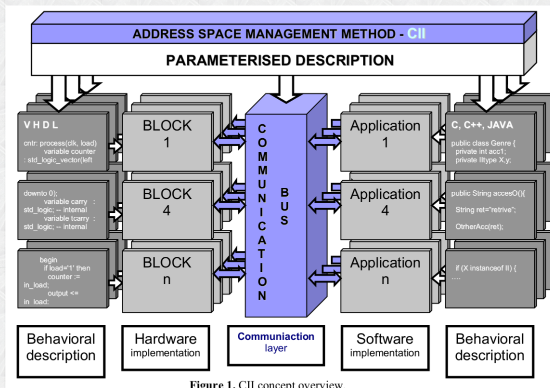


Figure 1. CII concept overview.

Source: [2]

- Sophisticated solution resulting in complex FPGA logic, not Open Source

IPbus addressing scheme

- IPbus is well established standard for Ethernet communication with FPGA-based systems
- It is fully Open Sourced
- It uses nice XML tables for defining addresses of slaves
- There are software libraries for Python and C++

```
<node id="crob">
  <node id="crob_addr_ver" address="0x0" permission="r"/>
  <node id="sys_ctrl" address="0x200" permission="rw">
    <node id="febs_mode" mask="0x0000000E"/>
  </node>
  <node id="link_mask[0]" address="0x201" permission="rw"/>
  <node id="link_mask[1]" address="0x202" permission="rw"/>
  <node id="ic">
    <node id="ctrl" address="0x203" permission="rw">
      <node id="reset" mask="0x00000001"/>
      <node id="start_write" mask="0x00000002"/>
      <node id="start_read" mask="0x00000004"/>
      <node id="addr" mask="0x0000FF00"/>
    </node>
    <node id="tx_rega_nbtr" address="0x204" permission="rw">
      <node id="reg_addr" mask="0x0000FFFF"/>
      <node id="bytes_to_read" mask="0xFFFF0000"/>
    </node>
    <node id="tx_data" address="0x205" permission="rw"/>
    <node id="status" address="0x1" permission="r">
      <node id="ready" mask="0x00000001"/>
      <node id="empty" mask="0x00000002"/>
      <node id="addr" mask="0x0000FF00"/>
    </node>
    <node id="rx_mptr_nbw" address="0x2" permission="r">
      <node id="mem_ptr" mask="0x0000FFFF"/>
      <node id="words_read" mask="0xFFFF0000"/>
    </node>
    <node id="rx_data" address="0x3" permission="r"/>
  </node>
  [...]
</node>
```

IPbus addressing scheme

- IPbus is well established standard for Ethernet communication with FPGA-based systems
- It is fully Open Sourced
- It uses nice XML tables for defining addresses of slaves
- There are software libraries for Python and C++
- Unfortunately, it provides very limited support for automatic generation of address tables

```
<node id="crob">
  <node id="crob_addr_ver" address="0x0" permission="r"/>
  <node id="sys_ctrl" address="0x200" permission="rw">
    <node id="febs_mode" mask="0x0000000E"/>
  </node>
  <node id="link_mask[0]" address="0x201" permission="rw"/>
  <node id="link_mask[1]" address="0x202" permission="rw"/>
  <node id="ic">
    <node id="ctrl" address="0x203" permission="rw">
      <node id="reset" mask="0x00000001"/>
      <node id="start_write" mask="0x00000002"/>
      <node id="start_read" mask="0x00000004"/>
      <node id="addr" mask="0x0000FF00"/>
    </node>
    <node id="tx_rega_nbtr" address="0x204" permission="rw">
      <node id="reg_addr" mask="0x0000FFFF"/>
      <node id="bytes_to_read" mask="0xFFFF0000"/>
    </node>
    <node id="tx_data" address="0x205" permission="rw"/>
    <node id="status" address="0x1" permission="r">
      <node id="ready" mask="0x00000001"/>
      <node id="empty" mask="0x00000002"/>
      <node id="addr" mask="0x0000FF00"/>
    </node>
    <node id="rx_mptr_nbw" address="0x2" permission="r">
      <node id="mem_ptr" mask="0x0000FFFF"/>
      <node id="words_read" mask="0xFFFF0000"/>
    </node>
    <node id="rx_data" address="0x3" permission="r"/>
  </node>
  [...]
</node>
```

IPbus extension - adr_gen

- To support automatic generation of address maps for IPbus, the “adr_gen” [3] system was created
- It uses the standard IPbus *ipbus_ctrlreg_v* block which provides vector of control registers and vector of status registers
- Similar block may be also generated for AXI bus instead of IPbus
- The hierarchy of connected blocks and registers is described in Python
- The registers in connected hierarchy of blocks are assigned consecutive addresses (that may result in inefficient decoding)
- The addresses are generated in VHDL package, in IPbus XML and in Python module

```
TOP
|
|
+-SREG: top_status
+-CREG: sys_control
+-CREG: resets
|
+-N_OF_A x ABlocks
| |
| +-SREG: A_status
| +-CREG: A_control
| +-N_OF_I2C_SLAVES x I2CBlock
| | |
| | +-CREG: I2C_Config
| | +-SREG: I2C_Status
| | +-CREG: I2C_Command
| |
| +-N_OF_SPI_SLAVES x SPIBlock
|   |
|   +-CREG: SPI_Config
|   +-SREG: SPI_Status
|   +-CREG: SPI_Tx
|   +-SREG: SPI_Rx
|
+-N_OF_B x BBlocks
  |
  +- N_OF_CELLS x CREG: Out_data
  +- N_OF_CELLS x SREG: In_data
  +- SREG: B_status
  +- CREG: B_config
```

IPbus extension - adr_gen

```
#!/usr/bin/python3
from addr_gen import *

#Definitions of constants used in the package
c.ADDR_VERSION=int(time.time())
c.N_OF_A = 13
c.N_OF_I2C_SLAVES = 6
c.N_OF_SPI_SLAVES = 8
c.N_OF_B = 5
c.N_OF_CELLS = 12

#Define registers in the BBlock
bbl_def=aobj("BBLOCK",[
  ("out_data",sreg_def,c.N_OF_CELLS),
  ("in_data",sreg_def,c.N_OF_CELLS),
])

#Define registers in SPI block
spi_def=aobj("SPI",[
  ("spi_config",creg_def),
  ("spi_status",sreg_def),
  ("spi_tx",creg_def),
  ("spi_rx",sreg_def),
])

#Define registers in I2C block
i2c_def=aobj("I2C",[
  ("i2c_config",creg_def),
  ("i2c_status",sreg_def),
  ("i2c_command",creg_def),
])

#Define registers and subblocks in the ABlock
abl_def=aobj("ABLOCK",[
  ("a_status",creg_def),
  ("a_control",creg_def,2),
  ("spi",spi_def,c.N_OF_SPI_SLAVES),
  ("i2c",spi_def,c.N_OF_I2C_SLAVES),
])

#Define registers and subblocks in the TOP block
top_def=aobj("TOP",[
  ("addr_ver",sreg_def),
  ("top_st",sreg_def),
  ("sys_ctrl",sreg_def),
  ("resets",creg_def),
  ("ab",abl_def,c.N_OF_A),
  ("bb",bbl_def,c.N_OF_B),
])

#Generate package with constants
gen_vhdl_const_package("top_const_pkg")

#Generate package with types and addresses
gen_vhdl_addr_package("top_adr_pkg","",crob_def,0,0)

#Generate Python module with addresses
gen_python_addr_module("top_adr",crob_def,0,0)
```

IPbus extension - adr_gen

Wilga Symposium

VHDL code with type definitions

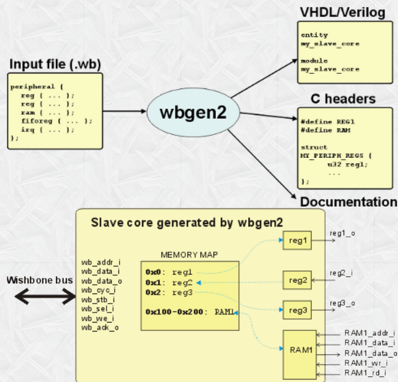
```
type T_I2C_CTRL is record
  i2c_config : std_logic_vector(31 downto 0);
  i2c_command : std_logic_vector(31 downto 0);
end record T_I2C_CTRL;
type T_I2C_CTRL_ARR is array(natural range<>)
   of T_I2C_CTRL;

type T_SPI_CTRL is record
  spi_config : std_logic_vector(31 downto 0);
  spi_command : std_logic_vector(31 downto 0);
end record T_SPI_CTRL;
type T_SPI_CTRL_ARR is array(natural range<>)
   of T_SPI_CTRL;

type T_ABL_CTRL is record
  a_control: std_logic_vector(31 downto 0);
  spi : T_SPI_CTRL_ARR(0 to N_OF_SPI_SLAVES-1);
  i2c : T_I2C_CTRL_ARR(0 to N_OF_I2C_SLAVES-1);
end record T_ABL_CTRL;
```

VHDL code for connecting registers

```
-- Process for connecting the signals
process (all) is
begin  -- process
  stat_reg(tad_addr.addr_ver) <=
      std_logic_vector(to_unsigned(32,ADDR_VERSION));
  stat_reg(tad_addr.top_st) <= s_top_status;
  s_top_control <= ctrl_reg(tad_addr.sys_ctrl);
  s_resets <= ctrl_reg(tad_addr.resets);
  for an in 0 to N_OF_A-1 loop
     stat_reg(tad_addr.ab(an).a_status)<=s_a_stat(an).a_status;
     s_a_ctrl(an)<=ctrl_reg(tad_addr.ab(an).a_control);
     for spin in 0 to N_OF_SPI_SLAVES loop
        s_a_ctrl(an).spi(spin).spi_config <=
            ctrl_reg(tad_addr.ab(an).spi(spin).spi_config;
        stat_reg(tad_addr.ab(an).spi(spin).spi_status) <=
            s_a_stat(an).spi(spin).spi_status;
        s_a_ctrl(an).spi(spin).spi_command <=
            ctrl_reg(tad_addr.ab(an).spi(spin).spi_command;
     end loop; -- spin
     -- Similar loop for I2C slaves
  end loop;  -- an
  -- Similar loop for B Blocks
end process;
```

Wishbone slave generator (wbgen2)

- It is fully Open Source (written in lua)
- Uses C-like description of the peripheral
- Generates the HDL (VHDL/Verilog) code for FPGA
- Generates the C headers to access registers
- Generates very nice documentation in LaTeX, texinfo or HTML

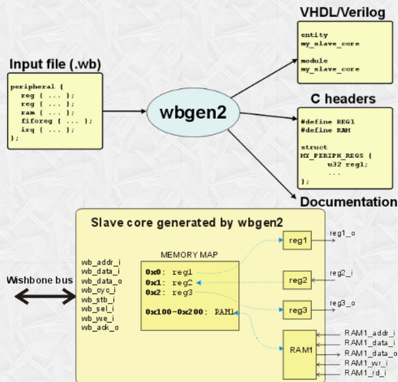


Source: [4]

Wishbone slave generator (wbgen2)

Wilga Symposium

- It is fully Open Source (written in lua)
- Uses C-like description of the peripheral
- Generates the HDL (VHDL/Verilog) code for FPGA
- Generates the C headers to access registers
- Generates very nice documentation in LaTeX, texinfo or HTML
- Unfortunately, it does not handle vectors of registers and hierarchy of blocks



Source: [4]

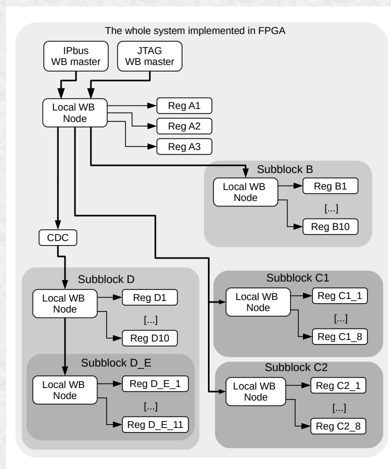
- All existing solutions had certain limitations
- It appeared that none of them may be easily extended for our needs
- It was necessary to create a new system
- The aim was to combine the best features of the existing solutions

addr_gen_wb selection of the bus

- The solution was inspired by wbgen2
- Wishbone bus is a standard Open Source internal bus for FPGA-based system
- IPbus slaves may be controlled by Wishbone bus
- IPbus masters may control Wishbone bus in classic single mode
- There are bridges enabling control of Wishbone bus from AXI masters
- Therefore Wishbone bus (WB) was selected as an FPGA internel bus

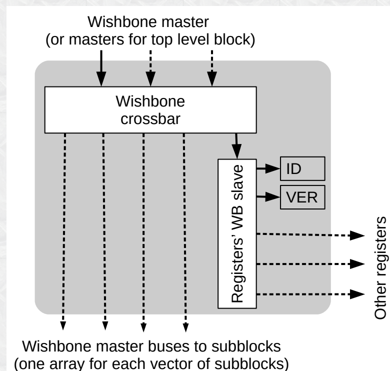
addr_gen_wb structure of created system

- It is possible to describe complex multilevel hierarchical system
- It is possible to create vectors of registers and blocks
- The external control connections in each block are limited to two records
- The internal control interface (Local WB node) for each node is automatically generated



addr_gen_wb local WB node

- The local WB node includes standard WB crossbar
- It is possible to implement crossbar in registered mode to shorten critical path (at cost of increased latency)



addr_gen_wb system description

- The system is described with XML files

```
<sysdef top="MAIN">

<block name="SYS1">
  <creg name="CTRL" desc="Control register" stb="1">
    <field name="START" width="1"/>
    <field name="STOP" width="1"/>
  </creg>
  <sreg name="STATUS" desc="Status register" ack="1" />
  <creg name="ENABLEs" desc="Link enable registers" reps="10" default="0x0"/>
</block>

<block name="MAIN">
  <subblock name="LINKS" type="SYS1" reps="5"/>
  <blackbox name="EXTERN" type="EXTTEST" addrbits="10" reps="3" />
  <sreg name="INS" desc="Input registers" reps="2" ack="1" />
  <creg name="CTRL" desc="Control register in the main block" default="0x11" stb="1">
    <field name="CLK_ENABLE" width="1"/>
    <field name="CLK_FREQ" width="4"/>
    <field name="PLL_RESET" width="1"/>
  </creg>
</block>

</sysdef>
```

addr_gen_wb VHDL package

- For registers with bitfields, the VHDL package with complex data types is generated

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.wishbone_pkg.all;

package MAIN_wb_pkg is

subtype t_INS is std_logic_vector(31 downto 0);
type t_INS_array is array(0 to 1) of t_INS;

type t_CTRL is record
  CLK_ENABLE:std_logic_vector(0 downto 0);
  CLK_FREQ:std_logic_vector(3 downto 0);
  PLL_RESET:std_logic_vector(0 downto 0);
end record;

function stlv2t_CTRL(x : std_logic_vector) return t_CTRL;
function t_CTRL2stlv(x : t_CTRL) return std_logic_vector;
end MAIN_wb_pkg;
```

```
package body MAIN_wb_pkg is
function stlv2t_CTRL(x : std_logic_vector) return t_CTRL is
variable res : t_CTRL;
begin
  res.CLK_ENABLE := std_logic_vector(x(0 downto 0));
  res.CLK_FREQ := std_logic_vector(x(4 downto 1));
  res.PLL_RESET := std_logic_vector(x(5 downto 5));
  return res;
end stlv2t_CTRL;

function t_CTRL2stlv(x : t_CTRL) return std_logic_vector is
variable res : std_logic_vector(31 downto 0);
begin
  res := (others => '0');
  res(0 downto 0) := std_logic_vector(x.CLK_ENABLE);
  res(4 downto 1) := std_logic_vector(x.CLK_FREQ);
  res(5 downto 5) := std_logic_vector(x.PLL_RESET);
  return res;
end t_CTRL2stlv;
end MAIN_wb_pkg;
```

addr_gen_wb VHDL for local WB node

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.wishbone_pkg.all;
use work.MAIN_wb_pkg.all;

entity MAIN_wb is
  port (
    slave_i : in t_wishbone_slave_in;
    slave_o : out t_wishbone_slave_out;
    EXTERN_wb_m_o : out t_wishbone_master_out_array(0 to 2);
    EXTERN_wb_m_i : in t_wishbone_master_in_array(0 to 2);
    LINKS_wb_m_o : out t_wishbone_master_out_array(0 to 4);
    LINKS_wb_m_i : in t_wishbone_master_in_array(0 to 4);

    INS_i : in  t_INS_array;
    INS_i_ack : out std_logic;
    CTRL_o : out  t_CTRL;
    CTRL_o_stb : out std_logic;

    rst_n_i : in std_logic;
    clk_sys_i : in std_logic
    );

end MAIN_wb;
-- [...]
-- (Implementation of the entity is omitted)
```

addr_gen_wb algorithm for address allocation

Wilga Symposium

- The algorithm starts from the most nested blocks
- For each block required number of addresses N is calculated from the number of its registers
- The power of 2 $2^K \geq N$ is found. K is the number of address bits required by the block
- For the parent block the required number of addresses is calculated basing on the number of its registers and requirements (2^K) of the child blocks.
- The blocks are sorted in the order of decreasing number of addresses.
- The based addresses are assigned starting from address 0. Each 2^M group of addresses is aligned to the 2^M boundary. That simplifies address decoders.
- In the parent blocks the addresses for child blocks are allocated in the same way, starting from the base address of the parent.

addr_gen_wb IPbus compatible address table

```xml
<node id="MAIN">
  <node id="EXTERN[0]" address="0x00000000" module="file://EXTERN_address.xml"/>
  <node id="EXTERN[1]" address="0x00000400" module="file://EXTERN_address.xml"/>
  <node id="EXTERN[2]" address="0x00000800" module="file://EXTERN_address.xml"/>
  <node id="LINKS[0]" address="0x00001000" module="file://SYS1_address.xml"/>
  <node id="LINKS[1]" address="0x00001010" module="file://SYS1_address.xml"/>
  <node id="LINKS[2]" address="0x00001020" module="file://SYS1_address.xml"/>
  <node id="LINKS[3]" address="0x00001030" module="file://SYS1_address.xml"/>
  <node id="LINKS[4]" address="0x00001040" module="file://SYS1_address.xml"/>
  <node id="ID" address="0x00001080" permission="r"/>
  <node id="VER" address="0x00001081" permission="r"/>
  <node id="INS[0]" address="0x00001082" permission="r"/>
  <node id="INS[1]" address="0x00001083" permission="r"/>
  <node id="CTRL" address="0x00001084" permission="rw">
    <node id="CLK_ENABLE" mask="0x00000001"/>
    <node id="CLK_FREQ" mask="0x0000001e"/>
    <node id="PLL_RESET" mask="0x00000020"/>
  </node>
</node>
```

addr_gen_wb Forth compatible address table

```
: %/ $0 ;
: %/#EXTERN %/ $0 + swap $400 * + ;
: %/#LINKS %/ $1000 + swap $10 * + ;
: %/#LINKS_ID %/#LINKS $0 + ;
: %/#LINKS_VER %/#LINKS $1 + ;
: %/#LINKS_CTRL %/#LINKS $2 + ;
: %/#LINKS_CTRL.START %/#LINKS_CTRL $1 $0 ;
: %/#LINKS_CTRL.STOP %/#LINKS_CTRL $2 $1 ;
: %/#LINKS_STATUS %/#LINKS $3 + ;
: %/#LINKS#ENABLEs %/#LINKS + $4 + ;
: %/_ID %/ $1080 + ;
: %/_VER %/ $1081 + ;
: %/#INS %/ + $1082 + ;
: %/_CTRL %/ $1084 + ;
: %/_CTRL.CLK_ENABLE %/_CTRL $1 $0 ;
: %/_CTRL.CLK_FREQ %/_CTRL $1e $1 ;
: %/_CTRL.PLL_RESET %/_CTRL $20 $5 ;
```

addr_gen_wb - latest extensions

The system is intensively developed and certain extensions were added after the preparation of the SPIE paper:

- Possibility to include XML files (with special *<!– include path/to/file.xml –>* metacomment)
- Possibility to define constants and use expressions inside of the system definition

```
<sysdef top="MAIN">
<constant name="NEXTERNS" val="4" />
<constant name="NSEL_BITS" val="3" />
<constant name="NSEL_MAX" val="(1 &lt;&lt; NSEL_BITS)-1" />
<!-- include block1.xml -->
<block name="MAIN">
  <subblock name="LINKS" type="SYS1" reps="NSEL_MAX+1"/>
  <blackbox name="EXTERN" type="EXTTEST" addrbits="10" reps="NEXTERNS" />
  <creg name="CTRL" desc="Control register in the main block" default="0x11">
    <field name="CLK_ENABLE" width="NSEL_BITS"/>
    <field name="CLK_FREQ" width="4"/>
    <field name="PLL_RESET" width="1"/>
  </creg>
</block>
</sysdef>
```

Results & Conclusions

- addr_gen_wb supports automated allocation of the addresses for registers in the complex, hierarchical data processing systems implemented in the FPGA.
- addr_gen_wb automatically generates VHDL code needed to provide Wishbone bus connectivity for nested blocks.
- It is possible to create vectors of registers or blocks.
- For registers with bitfields the VHDL record types and conversion functions are automatically created.
- addr_gen_wb supports parameterized definition of the system.

Results & Conclusions

- addr_gen_wb supports automated allocation of the addresses for registers in the complex, hierarchical data processing systems implemented in the FPGA.
- addr_gen_wb automatically generates VHDL code needed to provide Wishbone bus connectivity for nested blocks.
- It is possible to create vectors of registers or blocks.
- For registers with bitfields the VHDL record types and conversion functions are automatically created.
- addr_gen_wb supports parameterized definition of the system.
- The blocks comprising the system are well isolated regarding their interconnection with the control bus. That facilitates development and maintaining of systems assembled from blocks developed different teams independently. That's an essential feature in electronics created e.g., for High Energy Physics experiments.

Results & Conclusions

- addr_gen_wb supports automated allocation of the addresses for registers in the complex, hierarchical data processing systems implemented in the FPGA.
- addr_gen_wb automatically generates VHDL code needed to provide Wishbone bus connectivity for nested blocks.
- It is possible to create vectors of registers or blocks.
- For registers with bitfields the VHDL record types and conversion functions are automatically created.
- addr_gen_wb supports parameterized definition of the system.
- The blocks comprising the system are well isolated regarding their interconnection with the control bus. That facilitates development and maintaining of systems assembled from blocks developed different teams independently. That's an essential feature in electronics created e.g., for High Energy Physics experiments.
- addr_gen_wb has been successfully used in the development of FPGA firmware for the GBTX emulator for CBM experiment. It is also planned as a tool to integrate various blocks in the future CRI firmware for the CBM experiment.

Results & Conclusions

- addr_gen_wb supports automated allocation of the addresses for registers in the complex, hierarchical data processing systems implemented in the FPGA.
- addr_gen_wb automatically generates VHDL code needed to provide Wishbone bus connectivity for nested blocks.
- It is possible to create vectors of registers or blocks.
- For registers with bitfields the VHDL record types and conversion functions are automatically created.
- addr_gen_wb supports parameterized definition of the system.
- The blocks comprising the system are well isolated regarding their interconnection with the control bus. That facilitates development and maintaining of systems assembled from blocks developed different teams independently. That's an essential feature in electronics created e.g., for High Energy Physics experiments.
- addr_gen_wb has been successfully used in the development of FPGA firmware for the GBTX emulator for CBM experiment. It is also planned as a tool to integrate various blocks in the future CRI firmware for the CBM experiment.
- Sources of addr_gen_wb are available in the Github repository `https://github.com/wzab/addr_gen_wb` [5].

Bibliography

[1] Krzysztof T. Poźniak.
Internal interface i/o communication with fpga circuits and hardware description standard for applications in hep and fel electronics ver. 1.0, 2005.
Available from the website
https://flash.desy.de/reports_publications/tesla_reports/tesla_reports_2005/ .

[2] Pawel Drabik and Krzysztof T. Pozniak.
Maintaining complex and distributed measurement systems with component internal interface framework.
In Ryszard S. Romaniuk and Krzysztof S. Kulpa, editors, *Proc. SPIE*, volume 7502, page 75022C, Wilga, Poland, June 2009.

[3] Adr_gen - automatic address generator.
https://github.com/wzab/wzab-hdl-library/tree/master/addr_gen .

[4] Wishbone slave generator.
https://www.ohwr.org/project/wishbone-gen .

[5] Wojciech M. Zabołotny.
adr_gen_wb.py - register access for hierarchical wishbone connected systems, 2017.
https://github.com/wzab/addr_gen_wb .