CERN PH-ESE-BE

The Gigabit Link Interface Board (GLIB) user manual

v1.1.0

2012.06.05

DRAFT

GLIB project homepage: https://espace.cern.ch/project-GBLIB/public

Contact: Paschalis.Vichoudis@cern.ch

Document History

- **V1.10, 2012.06.05:** Firmware architecture modified. Applicable from the release amc_glib_1.1.0 onwards.
- **V1.07, 2012.04.25:** Minor modifications. Applicable from the release amc_glib_1.0.6 onwards.
- **V1.06, 2012.03.26:** Documentation of the release amc_glib_1.0.6. Only the links have been updated.
- **V1.05, 2012.03.01:** Documentation of the release amc_glib_1.0.5.
- **V1.0, 2012.12.07:** First draft in the document history. Associated with the release amc_glib_1.0.0.

GLIB team

Manoel Barros Marin, Sophie Baron, Vincent Bobillier, Stefan Haas, Magnus Hansen, Markus Joos, Lorena Lobato Pardavila, Patrick Petit, Francois Vasey & Paschalis Vichoudis.

Table of Contents

1. INTRODUCTION

We have designed and built a platform for the evaluation of optical links in the laboratory as well as for triggering and/or data acquisition systems in beam- or irradiation tests of detector modules. The Gigabit Link Interface Board (GLIB) [1] is based on an FPGA with Multi-Gigabit Transceivers (MGT) operating at rates of up to 6.5 Gb/s. This performance matches comfortably the specifications of the Gigabit Transceiver (GBT) [2] and Versatile Link projects [3] with its targeted data rate of 4.8 Gb/s. Figure 1-1 highlights the baseline configuration of a GBT - Versatile Link - GLIB system. On the left side, front-end (FE) ASICs are electrically connected to the GBT ASIC through e-links [4] while the GBT high-speed serial data-streams are converted to/from the optical domain through the Versatile Transceiver (VTRx) [5]. At the other end, the GLIB converts data to/from the optical domain, implements the GBT protocol and codes/decodes the user payload at the link back-end.

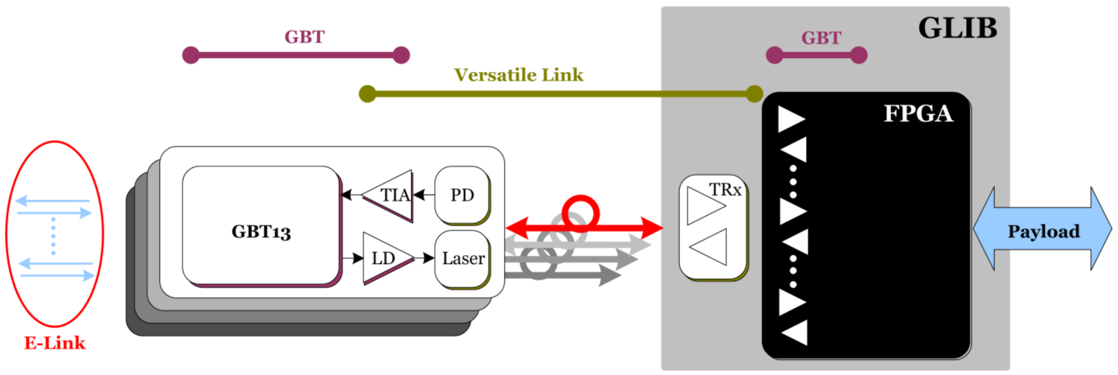


Figure 1-1: The GLIB board in a GBT-Versatile Link system

The GLIB I/O capability can be further enhanced with two FPGA Mezzanine Cards (FMCs). This gives users the flexibility to adapt the GLIB interface to their system, by for instance adding connectivity to the TTC network at the backend, or connecting to e-links at the frontend. Figure 1-2 illustrates a case where two GLIB boards are interconnected back-to-back, allowing implementing and experimenting with GBT-based systems well before full-fledged GBT ASICs become available.

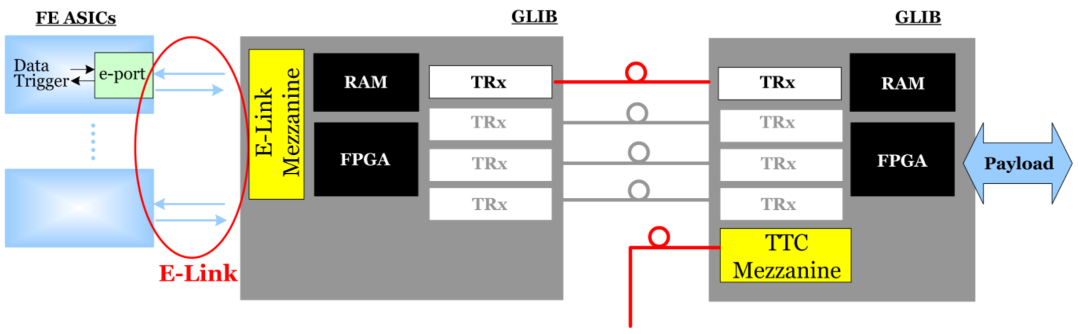


Figure 1-2: Back-to-back interconnected GLIB boards with customization mezzanines (TTC and E-Link) drawn in yellow.

The GLIB is conceived in the form of a double wide Advanced Mezzanine Card (AMC) that serves a small and simple system operating either stand-alone or residing inside a micro Telecommunications Computing Architecture (µTCA) crate. Figure 1-3 shows a picture of the GLIB prototype highlighting its major components i.e. the Virtex-6 FPGA, the AMC connector, the cage for 4 Small Form Factor Pluggable Plus (SFP+) optical modules, the FMC sockets, the SRAM devices, the socket for the

Module Management Controller (MMC) mezzanine card and the 1000Base-T interface (cabled Gigabit Ethernet). Figure 1-4 illustrates a block diagram of the GLIB architecture.

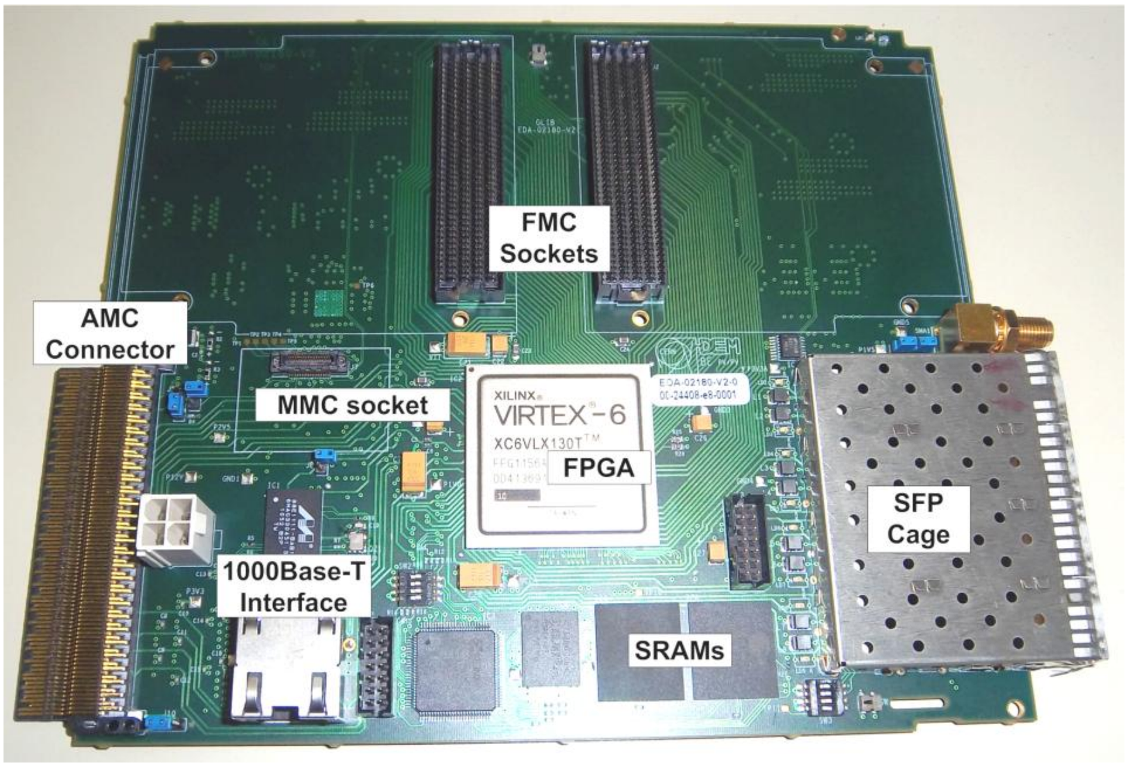


Figure 1-3: Picture of the GLIB prototype highlighting its major components.

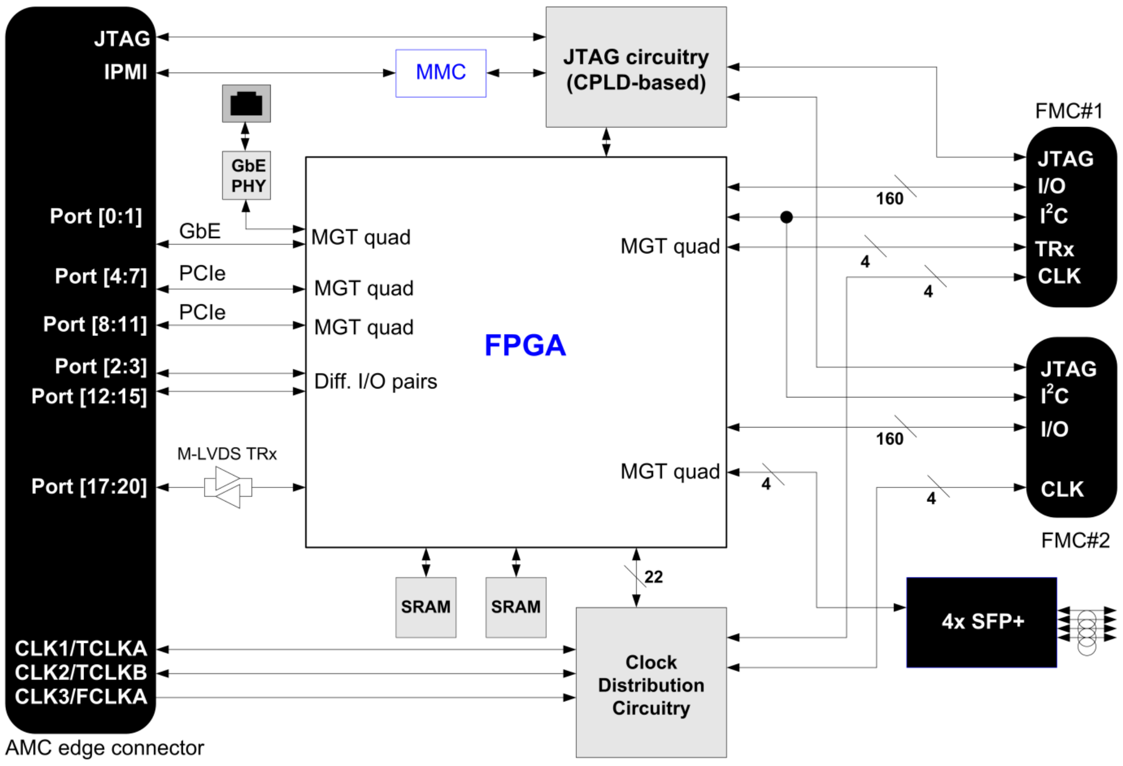


Figure 1-4: The Block diagram of the GLIB AMC card

Figure 1-5 shows a detailed diagram of the clock circuitry which is an essential part of the system. The MGT reference clocks, fabric clocks and control signals are shown on the FPGA side in red, black and blue, respectively.

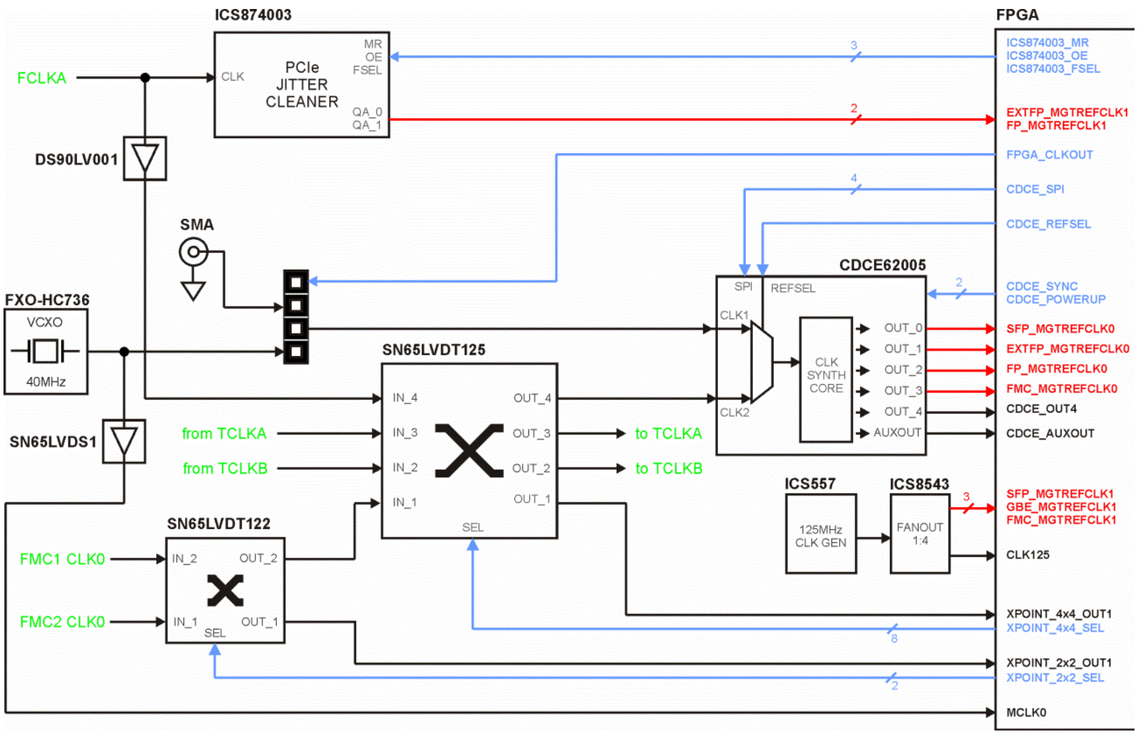


Figure 1-5: The clocking circuitry

It is important to mention that each MGT Reference clock (REFCLK) can be used to clock the MGT of its neighbouring MGT Quads (see Figure 1-6). For instance, the REFCLK0 of the MGT 114 can also clock the MGT quads 113 and 115. Details about the Virtex-6 clocking resources can be found in [6].

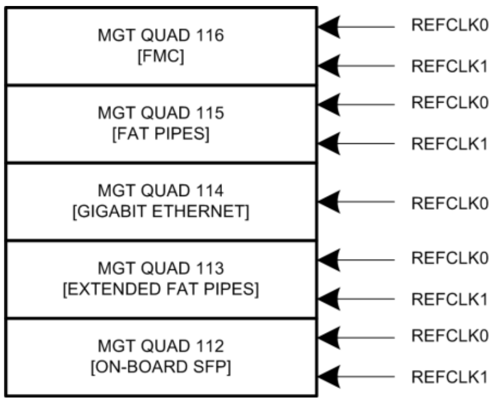


Figure 1-6: The MGT quads

2. ARCHITECTURE

Figure 2-1 illustrates the FPGA firmware architecture of the GLIB that is organized in two main parts, the *system_core* and the *user_logic*.

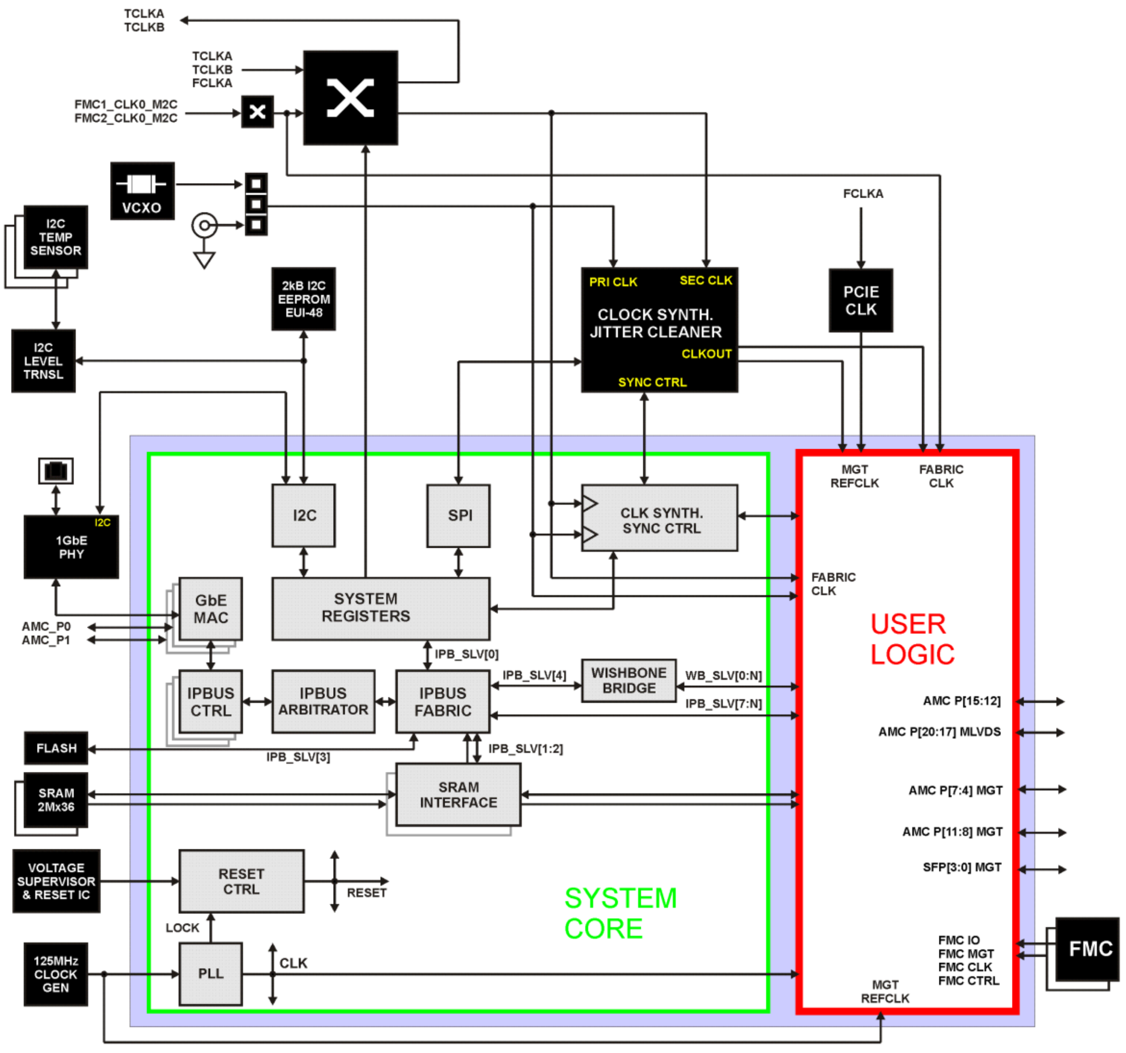


Figure 2-1: Firmware architecture.

The *system_core* firmware instantiates a simple IP-based control protocol (IPbus) designed for controlling xTCA-based hardware over Gigabit Ethernet that includes all basic transactions needed for this purpose (bitwise, single register and block transactions) [10]. The *system_core* also includes all interfaces to the external hardware e.g. I^{2}C communication with the on-board temperature sensors and the serial memory, SPI communication with clock synthesizer, SRAM interface with the two (operating at up to 160MHz) etc. In order to interface the GLIB with GBT protocol based systems, the GLIB firmware distribution provides GBT Hardware Description Language (HDL) modules [9] and associated optional Gigabit Transceiver (GTX) ready to be instantiated as is inside the *user_logic*.

2.1 System Core

PLL & Reset controller

The *system_core* contains a PLL which is clocked by the on-board 125MHz oscillator. The PLL provides a 62.5MHz clock which is used for the internal bus of the system (see §0) as well as a 125MHz clock for the Gigabit Ethernet MAC instantiations. The Reset controller generates a master reset pulse in the following cases:

- During power up (detected by the Voltage Supervisor)
- When the reset button is pressed (detected by the Voltage Supervisor).
- When the FPGA firmware is reloaded (internal logic).
- When the above mentioned PLL is not locked.

Both clocks (62.5MHz and 125MHz) as well as the reset pulse are forwarded to the *user_logic* block.

Gigabit Ethernet and IPbus

For the Gigabit Ethernet links, MAC cores are instantiated. Depending on the compilation settings, one or two Gigabit Ethernet Links are instantiated (for bench-top and crate operation respectively). In the case of bench-top operation, the MAC core is configured as 1000Base-T in order to communicate with the external PHY. In the case of crate operation, both MAC cores are configured as 1000Base-X for interfacing with the Gigabit Ethernet Switch carried on the crate's MCH.

For every MAC core, an IPbus endpoint is also instantiated. The IPBus system allows the control of hardware via a 'virtual bus', using a standard IP-over-gigabit-Ethernet network connection. The IPBus specifies a simple transaction protocol between the hardware and a software controller, which assumes an A32/D32 connection to slave devices connected to the hardware endpoint. The current IPbus firmware implementation is using a UDP/IP protocol and a simple synchronous SoC bus [8]. This protocol is based upon the Wishbone SoC protocol [9], and is compatible with Wishbone cores. However, there are two important differences:

- The master is *not* required to explicitly deassert strobe between cycles. However, it is *guaranteed* to deassert strobe or begin the new cycle on the clock cycle following ack.

- Slaves are *not* allowed to tie ack high, and *must* deassert ack on the same clock cycle that strobe is deasserted. However, it is allowed to tie ack to strobe, if a zero-wait-state response is always possible.

Timing diagrams of read and write transactions for a slave with and without wait states are given below. The first diagram illustrates a write cycle to a slave with one wait state; the bus idle for two clock cycles; then a read to a slave with zero wait states. The second diagram illustrates a Read-Modify-Write transaction with a slave with zero wait states, followed immediately by two reads from a slave with one wait state.

The reason of developing this custom SoC bus is to increase the bus efficiency by minimizing the dead-time.

Please note that although the Gigabit Ethernet module is part of the *system_core*, its IP address is set through the *user_ipaddr_package* (the default value is 192.168.0.111)

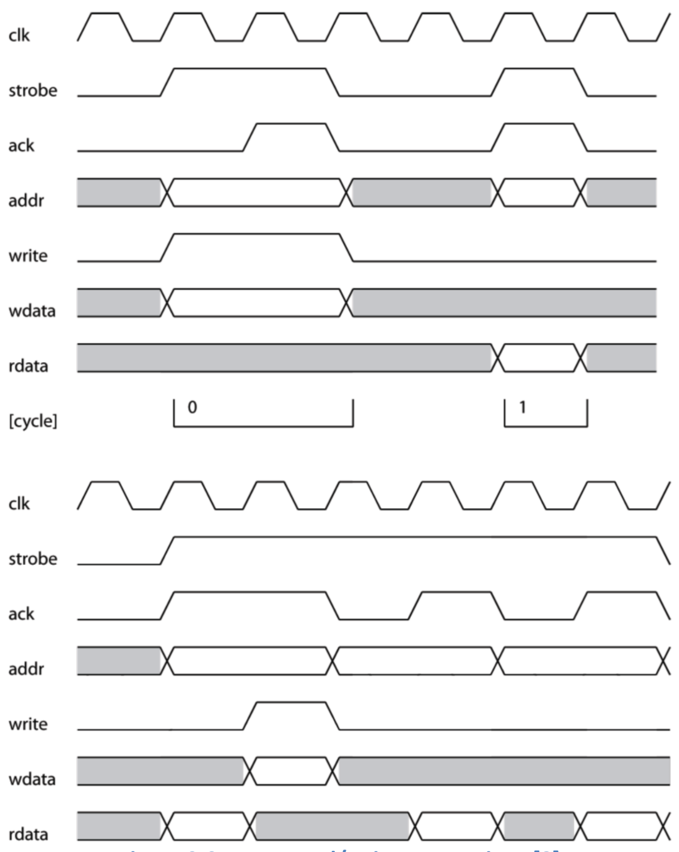


Figure 2-2: IPBus read/write transactions [8]

SoC bus

As mentioned in §0, there is a case where more than one MAC/IPbus cores are instantiated. For deciding which Gigabit Ethernet link will take over the bus, an Arbitration module is instantiated between the IPbus cores and the bus fabric. The bus fabric redirects the bus to only one of the slave devices instantiated by decoding the address (based on the memory map of Figure 2-3). The *system_core* instantiates various system slaves e.g. the system registers (base address = 0x00000000), SRAM1 (base address = 0x02000000) and SRAM2 (base address = 0x04000000). Additionally, the *system_core* allocates a large memory space for user slaves; IPbus slaves (base address = 0x40000000) and Wishbone slaves (base address = 0x80000000) that can be added to the *user_logic*.

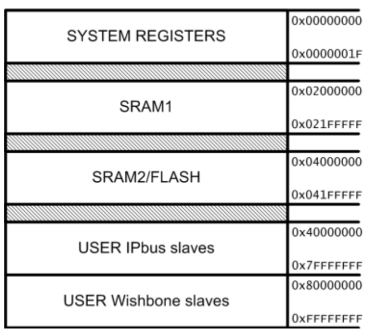


Figure 2-3: Memory map

System Registers

Table 2-1 shows the 16 registers that are currently implemented into the *System Registers* HDL block providing their address, a short description of their functionality and their type (Read-Write or Read-only). The 4-byte *Board_ID* register, as its name suggests, provides an identifier of the GLIB board (0x474C4942). When the 4 bytes of the identifier are represented in ASCII characters, the identifier corresponds to the word “GLIB”. Table 2-2 shows how the register is organized. The 4-byte *System_ID* register declares GLIB’s mode of operation (bench-top or crate). For bench-top mode, the register contains the value 0x4C414220 that corresponds to “LAB” when the 4 bytes are represented in ASCII format. When in crate mode, the register contains the value 0x75544341 that corresponds to “'uTCA”. Table 2-3 shows how the register is organized. The *Firmware_ID* register contains the date (*YY/MM/DD*) and the version number of the firmware (*major.minor.build*). The register *TestReg* is used only for testing the read/write transactions since it is not connected to the *system_core* logic. The *Ctrl* register (Table 2-5) is used to configure the clock circuitry (Figure 1-5). The register *Ctrl2* (Table 2-6) is used for loading firmware from the Platform Flash on demand.

Table 2-1: *System Registers*

Addr	Name	Description	Type
0x00	*Board_ID*	The board identifier code	RO
0x01	*System_ID*	The system identifier code	RO
0x02	*Firmware_ID*	The firmware date and version number	RO
0x03	*TestReg*	Register for test purposes only	RW
0x04	*Ctrl*	Controls the external clocking circuitry	RW
0x05	*Ctrl2*	Currently not used	RW
0x06	*Status*	Status from various external components	RO
0x07	*Status2*	Currently not used	RO
0x08	*Ctrl_SRAM*	SRAM interface: Control	RW
0x009	*Status_SRAM*	SRAM interface: Status	RO
0x0A	*SPI_txdata*	SPI interface: data from FPGA to clock synthesizer	RW
0x0B	*SPI_command*	SPI interface: configuration (polarity, phase, frequency etc)	RW
0x0C	*SPI_rxdata*	SPI interface: data from clock synthesizer to FPGA	RO
0x0D	*I2C_settings*	I2C interface: configuration (bus select, frequency etc)	RW
0x0E	*I2C_command*	I2C interface: transaction parameters (slave address, data to slave etc)	RW
0x0F	*I2C_reply*	I2C interface: transaction reply (transaction status, data from slave etc)	RO

Table 2-2: *Board_ID* Register

bit(s)	Name	Description
[7:0]	*board_id_char4*	Board ID 4th character (ASCII code)
[15:8]	*board_id_char3*	Board ID 3rd character (ASCII code)
[23:16]	*board_id_char2*	Board ID 2nd character (ASCII code)
[31:24]	*board_id_char1*	Board ID 1st character (ASCII code)

Table 2-3: *System_ID* Register

bit(s)	Name	Description
[7:0]	*system_id_char4*	System_ID 4th character (ASCII code)
[15:8]	*system_id_char3*	System_ID 3rd character (ASCII code)
[23:16]	*system_id_char2*	System_ID 2nd character (ASCII code)
[31:24]	*system_id_char1*	System_ID 1st character (ASCII code)

The *Status* register (Table 2-7) is providing status information from various external components. The register *Status2* is reserved for future use.

Table 2-4: *Firmware_ID* Register

bit(s)	Name	Description
[31:28]	*firmware_id_VER_MAJOR*	Firmware version (major)
[27:24]	*firmware_id_VER_MINOR*	Firmware version (minor)
[23:16]	*firmware_id_VER_BUILD*	Firmware version (build)
[15:9]	*firmware_id_YY*	Firmware year (0-99)
[8:5]	*firmware_id_MM*	Firmware month
[4:0]	*firmware_id_DD*	Firmware day

Table 2-5: *Ctrl* Register

bit(s)	Name	Description
[0]	*pcie_clk_fsel*	ICS874003 output multiplication factor. 0 -> OUT = 2.5 x IN. 1 -> OUT =1.25 x IN.
[1]	*pcie_clk_mr*	ICS874003 master reset. 1 -> reset. 0 -> normal operation.
[2]	*pcie_clk_oe*	ICS874003 output enable. 1 -> outputs enabled. 0 -> outputs disabled.
[4]	*cdce_powerup*	CDCE62005 Control: power up of. 0 -> power down. 1 -> power up.
[5]	*cdce_refsel*	CDCE62005 Control: Clock input selection. 1 -> CLK1. 0 -> CLK2.
[6]	*cdce_sync*	CDCE62005 Control: synchronization. A transition from 0 to 1 is needed to resync.
[7]	*cdce_ctrl_sel*	Select who drives the control of the CDCE62005. 0 -> *System_core.* 1 -> user logic.
[8]	*tclka_dr_en*	TCLKA direction. 0 -> from backplane to GLIB. 1 -> from GLIB to backplane
[9]	*tclkb_dr_en*	TCLKB direction. 0 -> from backplane to GLIB. 1 -> from GLIB to backplane
[10]	*xpoint_2x2_s1*	Input select for SN65LVDT122's OUT1. 0 -> IN1. 1 -> IN2
[11]	*xpoint_2x2_s2*	Input select for SN65LVDT122's OUT2. 0 -> IN1. 1 -> IN2
[13:12]	*xpoint_4x4_s1[1:0]*	Input select for SN65LVDT125's OUT1. 00 -> IN1. 10 -> IN2. 01 -> IN3. 11 -> IN4.
[15:14]	*xpoint_4x4_s2[1:0]*	Input select for SN65LVDT125's OUT2. 00 -> IN1. 10 -> IN2. 01 -> IN3. 11 -> IN4.
[17:16]	*xpoint_4x4_s3[1:0]*	Input select for SN65LVDT125's OUT3. 00 -> IN1. 10 -> IN2. 01 -> IN3. 11 -> IN4.
[19:18]	*xpoint_4x4_s4[1:0]*	Input select for SN65LVDT125's OUT4. 00 -> IN1. 10 -> IN2. 01 -> IN3. 11 -> IN4.

Table 2-6: *Ctrl2* Register

bit(s)	Name	Description
[1:0]	*flash_firmware_page*	Selects one of the 4 possible pages of the Platform Flash XL
[4]	*Load_flash_firmware*	Loads the firmware from the selected page

Table 2-7: *Status* Register

bit(s)	Name	Description
[2:0]	*glib_sfp1_status[2:0]*	on-board SFP1 status. bit[0] -> Mod_abs. bit [1] -> RxLOS. bit [2] -> TxFault.
[6:4]	*glib_sfp2_status[2:0]*	on-board SFP2 status. bit [0] -> Mod_abs. bit [1] -> RxLOS. bit [2] -> TxFault.
[10:8]	*glib_sfp3_status[2:0]*	on-board SFP3 status. bit [0] -> Mod_abs. bit [1] -> RxLOS. bit [2] -> TxFault.
[14:12]	*glib_sfp4_status[2:0]*	on-board SFP4 status. bit [0] -> Mod_abs. bit [1] -> RxLOS. bit [2] -> TxFault.
[16]	*gbe_int*	Interrupt request from GbE PHY. 0 -> no interrupt. 1 -> interrupt
[17]	*fmc1_presence*	Presence of FMC1. 1-> yes, 0-> no
[18]	*fmc2_presence*	Presence of FMC2. 1-> yes, 0-> no
[19]	*fpga_reset*	state of the fpga_reset line (driven by the CPLD)
[25:20]	*v6_cpld*	state of the 6-bit bus between the FPGA and the CPLD
[28]	*cdce_lock*	Status of CDCE62005. 1 -> locked. 0 -> unlocked

The registers *I2C_settings*, *I2C_command* and *I2C_reply* are presented in §0. The registers *SPI_txdata*, *SPI_command* and *SPI_rxdata* are presented in §0. The registers *Ctrl_SRAM* and *Status_SRAM* are presented in §**Error! Reference source not found.**.

I2C controller

The FPGA firmware instantiates a dual-bus I2C master controller that can be used to access the on-board I2C devices when in bench-top mode (in crate mode, the MMC is responsible for that task). Table 2-8 presents the I2C devices and their corresponding I2C 7-bit slave addresses. For the control of the I2C master, three registers are used, the following:

- The *I2C_settings* register used for configuring the controller (Table 2-9).
- The *I2C_command* for setting the I2C transaction parameters (Table 2-10).
- The *I2C_reply* provides the transaction status as well as data sent by the slave in case of read transactions (Table 2-11).

Table 2-12 gives an example list of transactions needed for reading the temperature of the FPGA with an I2C bus frequency of 62.5kHz.

Table 2-8: The I2C devices and their 7-bit slave addresses

I2C Device	Description	I2C Address
24AA025E48 EEPROM	256x8bit EEPROM with unique 48-bit serial number (EUI-48)	1010110
LM82 Temperature sensor #1	Provides the temperature of the FPGA's die at reg1 [addr:0x01]	0101010
LM82 Temperature sensor #2	Provides the pcb temperature at the front at reg0 [addr:0x00]	0011010
LM82 Temperature sensor #3	Measures the pcb temperature at the rear at reg0 [addr:0x00]	1001110
FMC#1	The I2C devices hosted on the FMC#1	XXXXX00
FMC#2	The I2C devices hosted on the FMC#2	XXXXX11

Table 2-9: "I2C_settings" register

bit(s)	Name	Description
[9:0]	*i2c_prescaler*	I2C clock prescaler. I2C clk (kHz)=62500/*i2c_prescaler*
[10]	*i2c_bus_select*	Select I2C bus. 1 -> I2C for PHY. 0 -> I2C for all other devices.
[11]	*i2c_enable*	Enable I2C controller. 1 -> enabled. 0 -> disabled.
[12]	*reserved*	Reserved. Keep it always at 0

Table 2-10: "I2C_command" register

bit(s)	Name	Description
[7:0]	*Wrdata*	Byte to write to the i2c slave.
[15:8]	*reserved*	Reserved. Keep always to 0.
[22:16]	*slv_addr*	The 7-bit slave address
[23]	*wr_en*	Write enable. 0 -> Read transaction. 1 -> Write transaction.
[24]	*reserved*	Reserved. Keep always to 0.
[25]	*mode16b*	16bit operation (for PHY only). 0 -> standard 8-bit mode. 1 -> 16bit mode
[31]	*i2c_strobe*	Execute Transaction strobe signal. It clears automatically. Keep it always to 1.

Table 2-11: "I2C_reply" register

bit(s)	Name	Description
[7:0]	*rddata_lo*	Byte read from the i2c slave. Low byte in case of 16-bit mode
[15:8]	*rddata_hi*	High byte read from the i2c slave in case of 16-bit mode
[27:26]	*i2c_status[1:0]*	Transaction status. 01 -> succeed. 11 -> failed. 00 -> pending. 10 -> pending.

Table 2-12: Example transaction list for reading the temperature of the FPGA's die

Register	Access	Description	Value (hex)
I2C_settings	WRITE	*i2c_enable -> 1,* *i2c_bus_sel -> 0,* *i2c_prescaler -> 1000*	0x00000BE8
I2C_command	WRITE	*wr -> 1, mode16b -> 0, slv_addr -> 0x2A* *wrdata -> 0x01* (select LM82's register#1)	0x80AA0001
I2C_reply	READ	Verify the transaction status	
I2C_command	WRITE	*wr -> 0, mode16b -> 0, slv_addr -> 0x2A,* *wrdata -> don't care*	0x802A0000
I2C_reply	READ	Verify the transaction status. If successful, the temperature will be available in *rddata_lo*	
I2C_settings	WRITE	*i2c_enable -> 0,* *i2c_bus_sel -> don't care,* *i2c_prescaler -> don't care*	0x00000000

SPI controller

The FPGA firmware instantiates an SPI master controller that is used for the configuration of the CDCE62005 clock synthesizer. For the control of the SPI master, three registers are used, the following:

- The *SPI_txdata* where the 32-bit word to be transmitted to the SPI slave is defined.
- The *SPI_rxdata* where the 32-bit word transmitted by the SPI slave is presented.
- The *SPI_command* where the SPI master's configuration is defined as well as the transaction parameters (Table 2-13).

Table 2-14 shows an example list of transactions for reading one of CDCE62005's registers. Table 2-15 shows an example of writing one of CDCE62005's registers.

Table 2-13: The *SPI_command* register

bit(s)	Name	Description
[11:0]	*spi_prescaler*	SPI clock prescaler. **SPI clk (MHz)=62.5/spi_prescaler. Suggested value: 0x014**
[27:12]	*reserved*	Reserved. **Keep always to 0xFA38.**
[31]	*spi_strobe*	Execute Transaction strobe signal. It clears automatically. **Keep it always to 1.**

Table 2-14: Transaction list for reading CDCE62005's Register 8

Register	Access	Description	Value (hex)
SPI_txdata	WRITE	Set the value that corresponds to the reading register 8 command of the CDCE62005	0x0000008E
SPI_command	WRITE	Execute the transaction	0x8FA38014
SPI_txdata	WRITE	Set a dummy value that does not corresponds to any CDCE62005 command	0xAAAAAAAA
SPI_command	WRITE	Execute the transaction	0x8FA38014
SPI_rxdata	READ	The contents of CDCE62005's Register 8 appear here	

Table 2-15: Example transaction list for writing CDCE62005's Register 0

Register	Access	Description	Value (hex)
SPI_txdata	WRITE	Set the value to write to Register 0. Note that bits[3:0] must be zero for addressing the register correctly	0xEB840720
SPI_command	WRITE	Execute the transaction	0x8FA38014

2.2 User Logic

From the *user_logic* point of view, there are two types of interfaces, direct connections with FPGA pins and interfaces with firmware blocks instantiated in the *system_core*. The use of the pins directly connected to the FPGA is straight forward (just if the *system_core* and the top level did not exist). This section focuses in the use of the SRAM, FMC I/O and Wishbone bus interfaces.

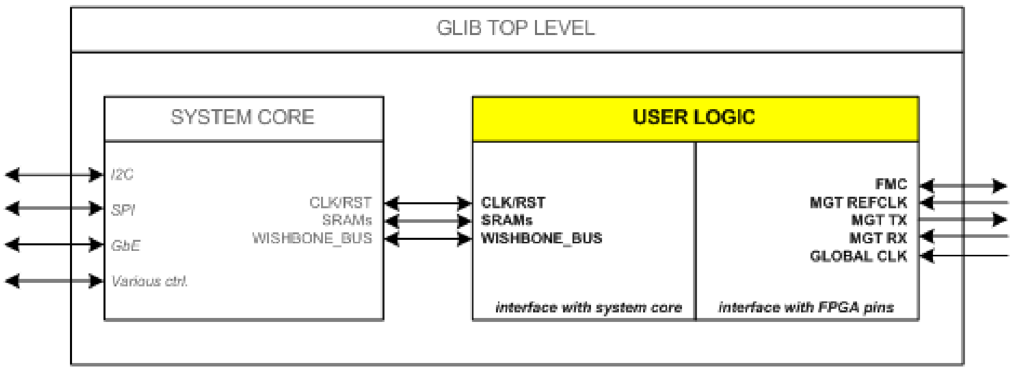


Figure 2-4: From the *User_logic* point of view

SRAM

The two single-port 2Mx36[1] SRAM devices that are available on board are accessible both by the *system_core* and the *user_logic*. The register *Ctrl_SRAM* (Table 2-16) is used to select which of the two blocks takes control over the SRAM, the mode of operation (normal or Built-In Self Test (BIST)). Additionally, since the SRAM2 and the FLASH memory share the same bus, it selects which of the two memories is addressed (please note that the access of the FLASH is not covered in this section). The register *Status_SRAM* (Table 2-17) is used only to report the results of the BIST. Logic Analyzer Waveforms captured with Chipscope for single write, single read, block write and block read SRAM transactions from the *user_logic* are shown in Figure 2-5, Figure 2-6, Figure 2-7 and Figure 2-8, respectively. Figure 2-9 shows an example use of an SRAM in the *user_logic* vhdl file.

Table 2-16: "Ctrl_SRAM" register

bit(s)	Name	Description
[0]	*user_ctrl_sram1*	Selects who is taking over the SRAM1 bus. 0 -> *system_core*. 1 -> *user_logic*.
[1]	*bist_sram1*	Built-in Self Test (BIST) Enable for SRAM1. 0 -> normal operation. 1 -> run the BIST.
[16]	*user_ctrl_sram2*	Selects who is taking over the SRAM2 bus. 0 -> *system_core*. 1 -> *user_logic*.
[17]b	*bist_sram2*	Built-in Self Test (BIST) Enable for SRAM2. 0 -> normal operation. 1 -> run the BIST.
[20]	*flash_select*	Selects between SRAM2 & FLASH. 0 -> SRAM2. 1 -> FLASH.

Table 2-17: "Status_SRAM" register

bit(s)	Name	Description
[0]	*bist_done_sram1*	SRAM1 BIST completed flag. 0 -> test pending. 1 -> test completed.
[1]	*bist_status_sram1*	SRAM1 BIST status flag. 0 -> test failed. 1 -> test passed.
[16]	*bist_done_sram2*	SRAM2 BIST completed flag. 0 -> test pending. 1 -> test completed.
[17]	*bist_status_sram2*	SRAM2 BIST status flag. 0 -> test failed. 1 -> test passed.

[1] Although the SRAMs have 2M addresses, therefore a 21-bit address bus is sufficient, the GLIB implements another four address lines connected to another 4 higher address pins of future SRAM devices of the same type, thus resulting a 25-bit address bus.

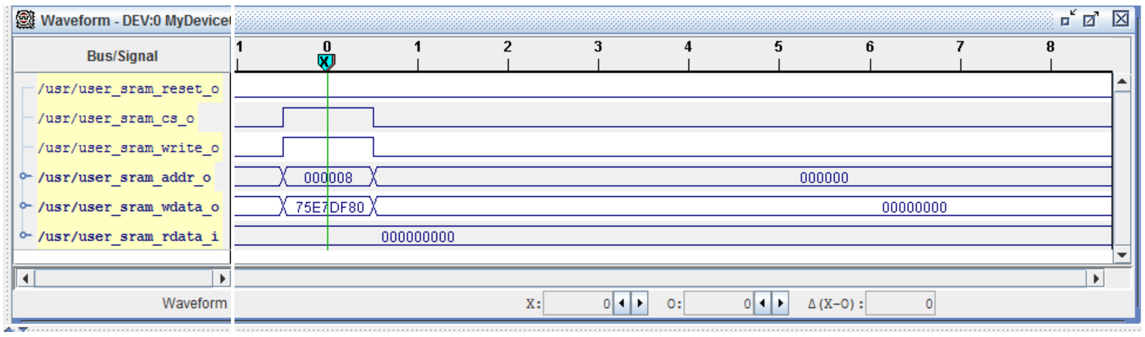


Figure 2-5: *User_logic* SRAM single write

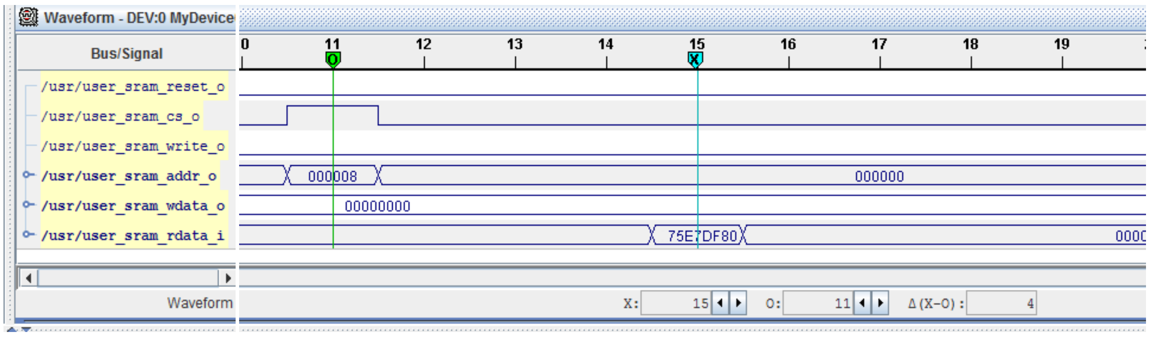


Figure 2-6: *User_logic* SRAM single read

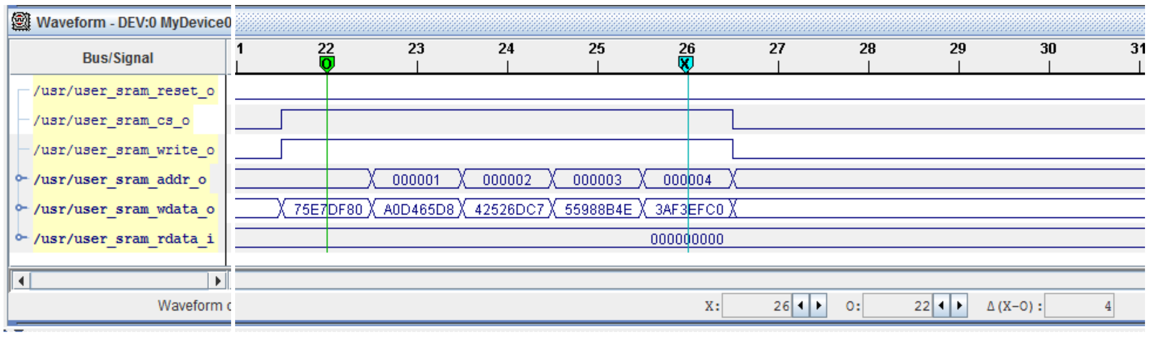


Figure 2-7: *User_logic* SRAM block write

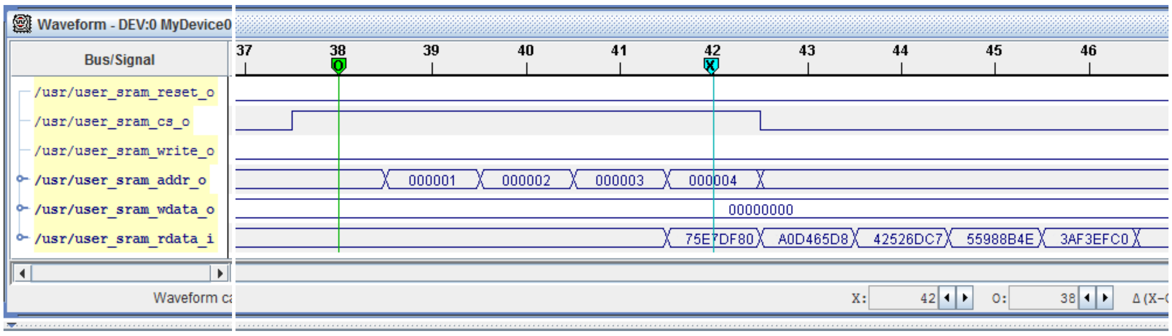
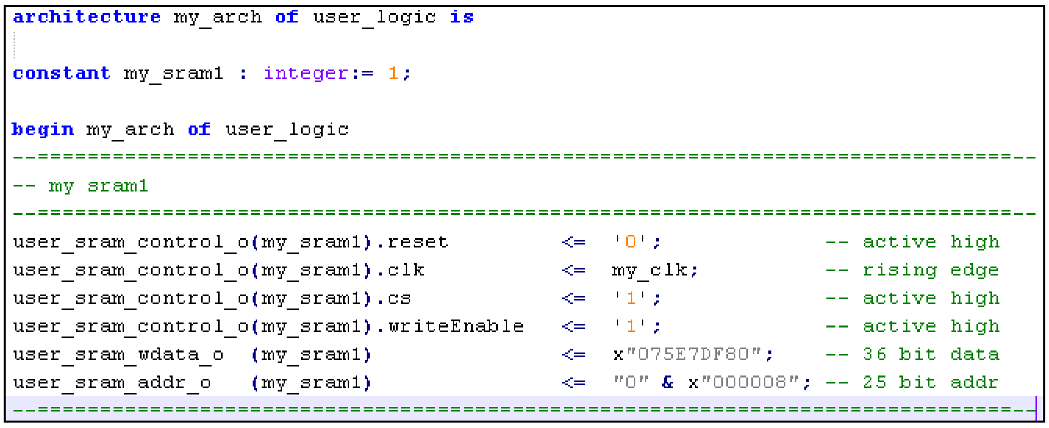


Figure 2-8: *User_logic* SRAM block read



```
architecture my_arch of user_logic is

constant my_sram1 : integer:= 1;

begin my_arch of user_logic
--===========================================================================--
-- my sram1
--===========================================================================--
user_sram_control_o(my_sram1).reset         <=  '0';              -- active high
user_sram_control_o(my_sram1).clk           <=  my_clk;           -- rising edge
user_sram_control_o(my_sram1).cs            <=  '1';              -- active high
user_sram_control_o(my_sram1).writeEnable   <=  '1';              -- active high
user_sram_wdata_o   (my_sram1)              <=  x"075E7DF80";     -- 36 bit data
user_sram_addr_o    (my_sram1)              <=  "0" & x"000008";  -- 25 bit addr
--===========================================================================--
```

Figure 2-9: Example use of the SRAM in the *user_logic* file

FMC I/O

In order to facilitate the interface with the 160 I/Os per FMC socket (that according to the FMC specification they can be freely configured as LVDS or single-ended as well as input, output or bidirectional) the GLIB firmware distribution provides a configurable buffer component (*fmc_io_buffers*) that needs to be instantiated in the *user_logic* and used in conjunction with the two special data types (records) developed (see Figure 2-10):

- “*fmc_from_fabric_to_pin_type*” (direction: from the *user_logic* to the FMC connector)
- “*fmc_from_pin_to_fabric_type*” (direction: from the FMC connector to the *user_logic*).

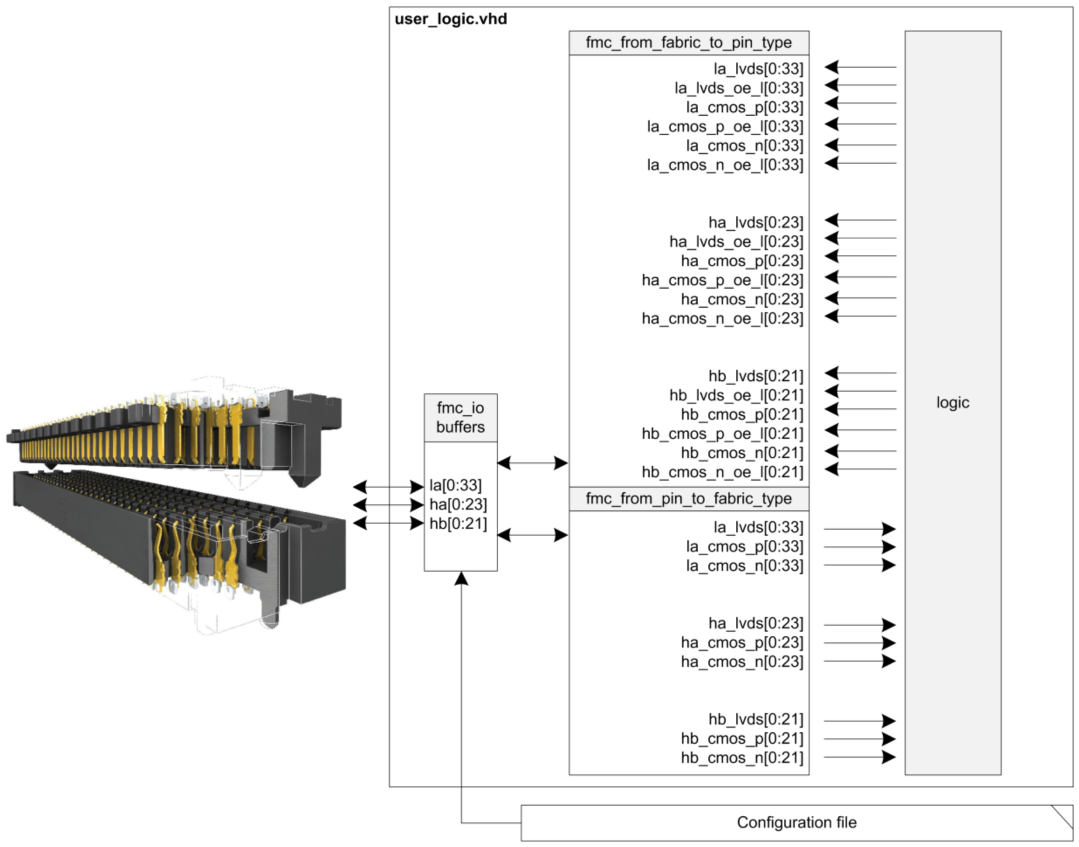


Figure 2-10: FMC I/O user interface diagram

The configuration of the FMC I/Os takes place into the following configuration packages:

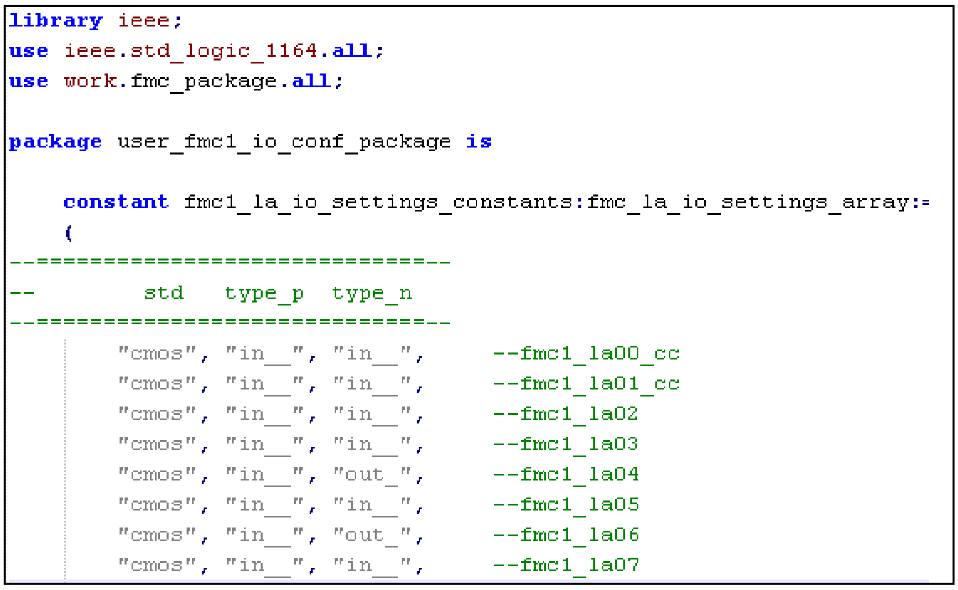
- *user_fmc1_io_conf_package*: for the FMC#1 (front)
- *user_fmc2_io_conf_package*: for the FMC#2 (rear)

An example of the above files is shown in Figure 2-11. In these files, an array of constants per FMC pin group (*LA[0:33], HA[0:23]* and *HB[0:21]*) is declared. Each array consists of three columns:

- 1st column: the logic standard. The legal values are "*cmos*" and "*lvds*".
- 2nd column: the buffer type for the lines *LA[*]_p / HA[*]_p / HB[*]_p* in case of "*cmos*" or the differential pair in case of "*lvds*". The legal values are "*in__*", "*out_*", "*i_o_*" and "*ckin*". The "*i_o_*" setting applies only to "*cmos*" configured pins. The "*ckin*" applies only to clock capable (CC) lines e.g. the line *FMC1_LA00_CC*.
- 3rd column: the buffer type of the lines *LA[*]_n / HA[*]_n / HB[*]_n* only in case of "*cmos*".

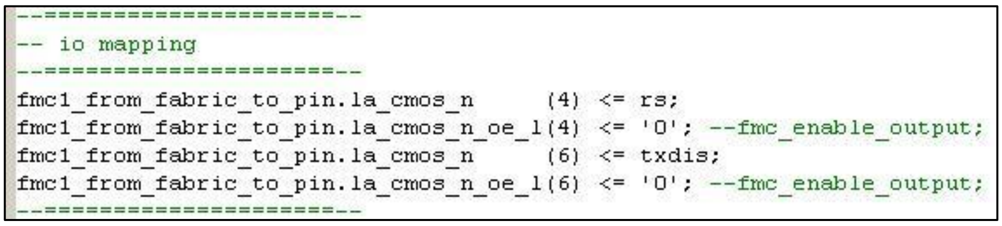
Please note that all above mentioned strings are case sensitive.

An example use of FMC I/Os in the *user_logic* is shown in Figure 2-12.



```
library ieee;
use ieee.std_logic_1164.all;
use work.fmc_package.all;

package user_fmc1_io_conf_package is

    constant fmc1_la_io_settings_constants:fmc_la_io_settings_array:=
    (
--================================--
--         std   type_p  type_n
--================================--
        "cmos", "in__", "in__",      --fmc1_la00_cc
        "cmos", "in__", "in__",      --fmc1_la01_cc
        "cmos", "in__", "in__",      --fmc1_la02
        "cmos", "in__", "in__",      --fmc1_la03
        "cmos", "in__", "out_",      --fmc1_la04
        "cmos", "in__", "in__",      --fmc1_la05
        "cmos", "in__", "out_",      --fmc1_la06
        "cmos", "in__", "in__",      --fmc1_la07
```

Figure 2-11: Example of an FMC I/O configuration file

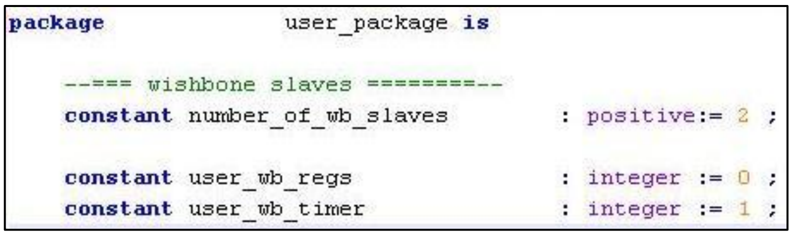


```
--========================--
-- io mapping
--========================--
fmc1_from_fabric_to_pin.la_cmos_n      (4) <= rs;
fmc1_from_fabric_to_pin.la_cmos_n_oe_l(4) <= '0'; --fmc_enable_output;
fmc1_from_fabric_to_pin.la_cmos_n      (6) <= txdis;
fmc1_from_fabric_to_pin.la_cmos_n_oe_l(6) <= '0'; --fmc_enable_output;
--========================--
```

Figure 2-12: Example use of FMC I/Os in the *user_logic* file

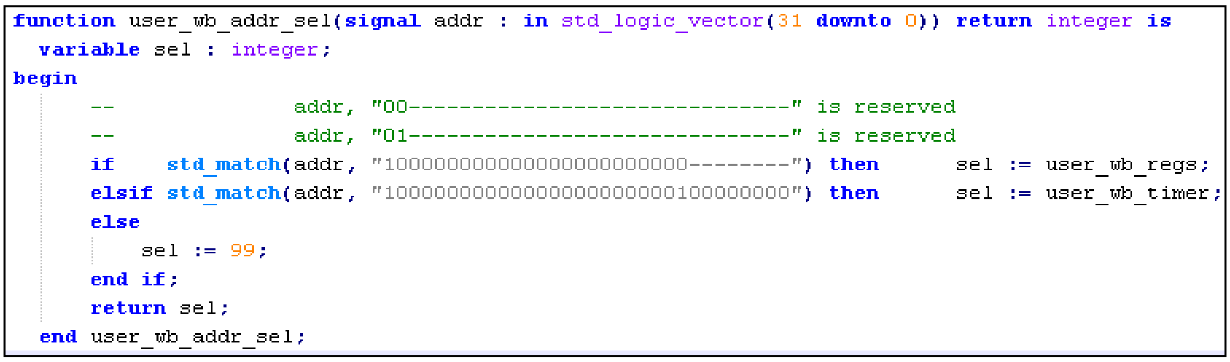
Wishbone bus

The user has the possibility to instantiate wishbone-compatible slaves inside the *user_logic* as well as to attach them on the wishbone bus available. Figure 2-13 shows an example of how to declare the total number of wishbone slaves (and their symbolic names). Figure 2-14 shows the address mapping of the wishbone slaves. Figure 2-15 illustrates an example of a wishbone slave instantiation inside the *user_logic.* Example wishbone slaves can be found in the example projects available.



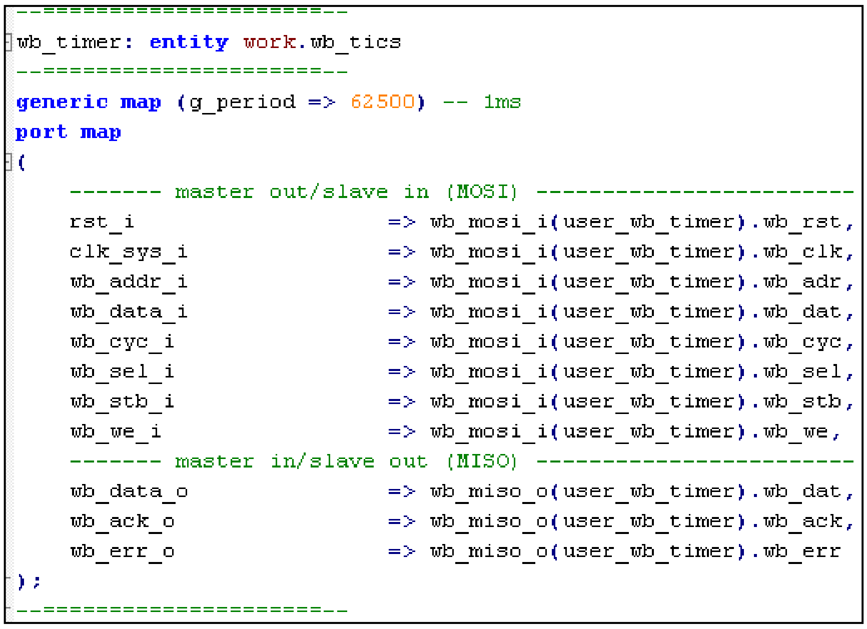
```
package                    user_package is

    --=== wishbone slaves =========--
    constant number_of_wb_slaves          : positive:= 2 ;

    constant user_wb_regs                 : integer := 0 ;
    constant user_wb_timer                : integer := 1 ;
```

Figure 2-13: Wishbone slave declaration (*user_package*)



```
function user_wb_addr_sel(signal addr : in std_logic_vector(31 downto 0)) return integer is
  variable sel : integer;
begin
      --              addr, "00------------------------------" is reserved
      --              addr, "01------------------------------" is reserved
      if     std_match(addr, "100000000000000000000000--------") then      sel := user_wb_regs;
      elsif std_match(addr, "10000000000000000000000100000000") then      sel := user_wb_timer;
      else
          sel := 99;
      end if;
      return sel;
  end user_wb_addr_sel;
```

Figure 2-14: Example wishbone slave address mapping (*user_addr_decode.vhd*)



```
--========================--
wb_timer: entity work.wb_tics
--========================--
generic map (g_period => 62500) -- 1ms
port map
(
    ------- master out/slave in (MOSI) -------------------------
    rst_i                   => wb_mosi_i(user_wb_timer).wb_rst,
    clk_sys_i               => wb_mosi_i(user_wb_timer).wb_clk,
    wb_addr_i               => wb_mosi_i(user_wb_timer).wb_adr,
    wb_data_i               => wb_mosi_i(user_wb_timer).wb_dat,
    wb_cyc_i                => wb_mosi_i(user_wb_timer).wb_cyc,
    wb_sel_i                => wb_mosi_i(user_wb_timer).wb_sel,
    wb_stb_i                => wb_mosi_i(user_wb_timer).wb_stb,
    wb_we_i                 => wb_mosi_i(user_wb_timer).wb_we,
    ------- master in/slave out (MISO) -------------------------
    wb_data_o               => wb_miso_o(user_wb_timer).wb_dat,
    wb_ack_o                => wb_miso_o(user_wb_timer).wb_ack,
    wb_err_o                => wb_miso_o(user_wb_timer).wb_err
);
--========================--
```

Figure 2-15: Example instantiation of a wishbone slave in the *user_logic* file

3. How to use the GLIB

3.1 Hardware

Jumpers

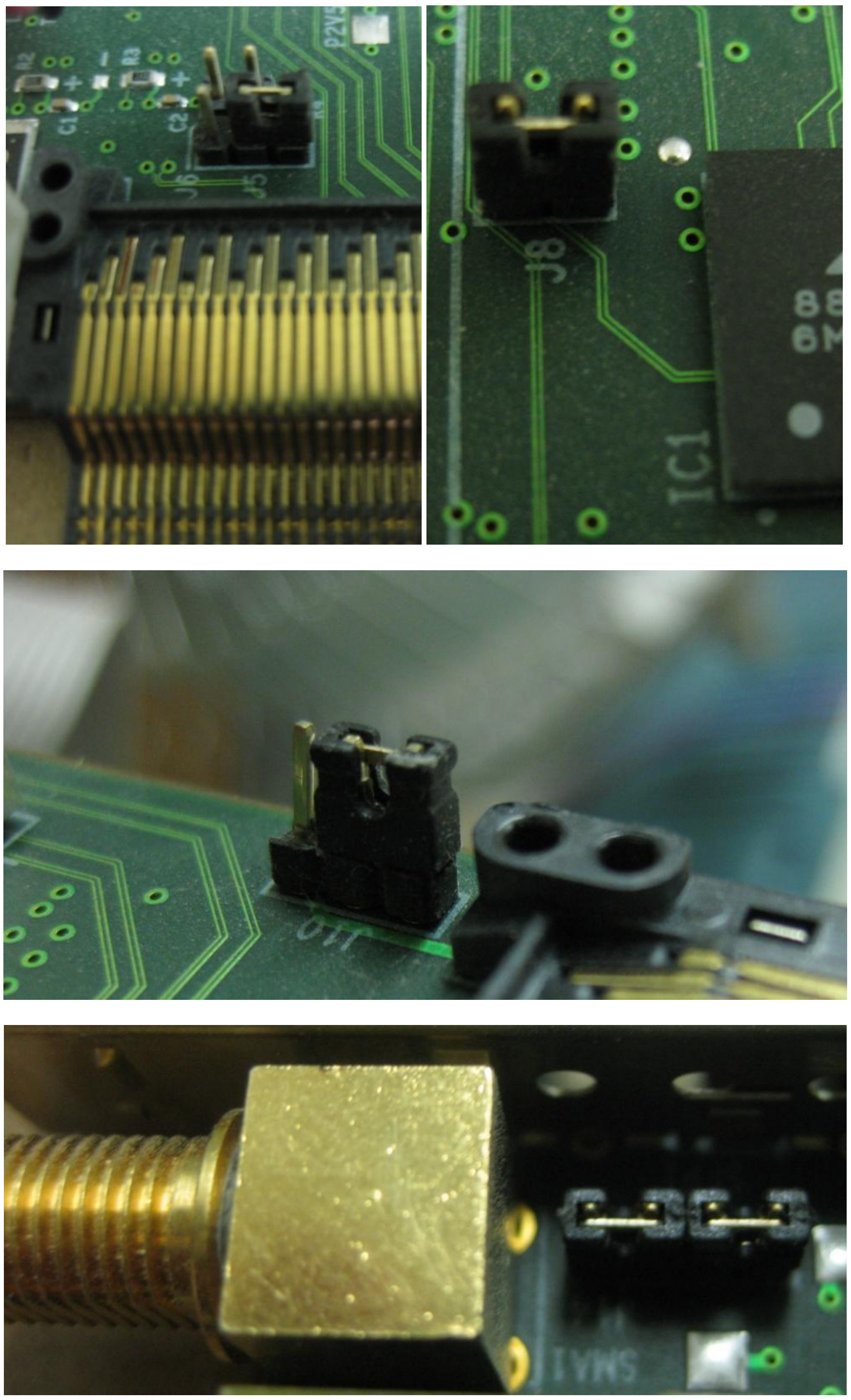


Figure 3-1: Default jumper positions

Switches

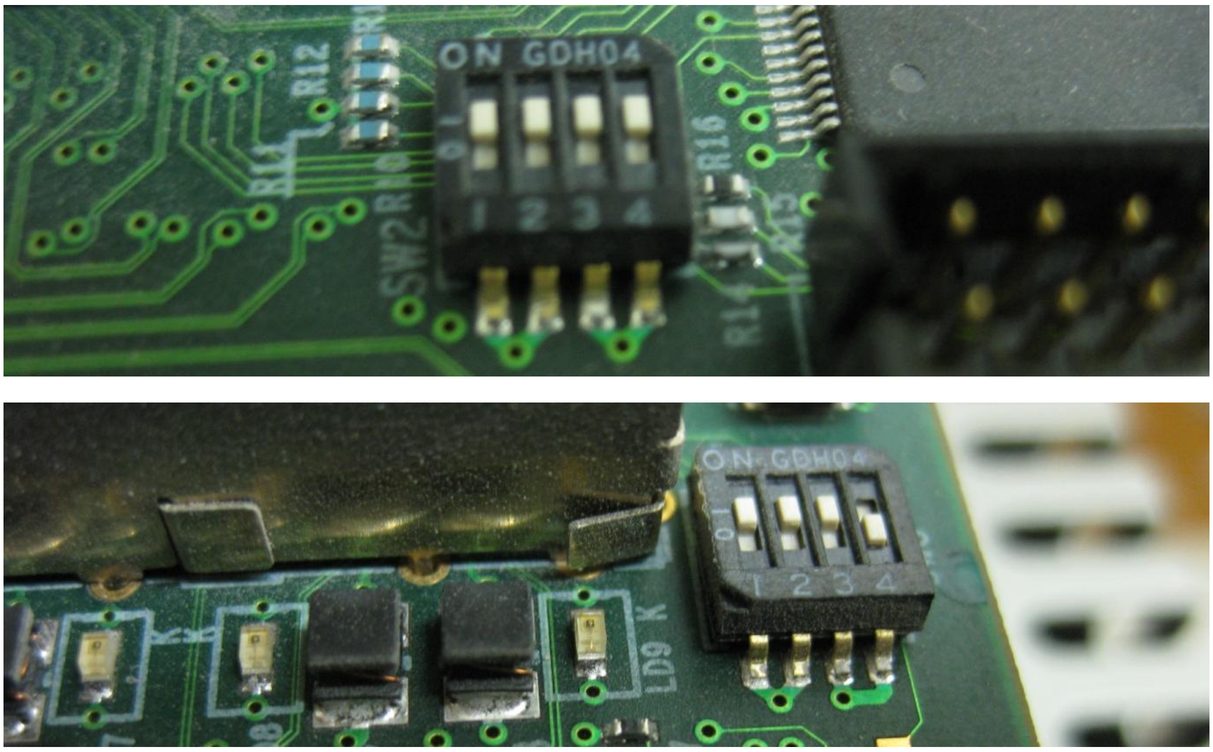


Figure 3-2: Default switch positions

Ethernet connection (bench-top)

Please plug the GLIB either directly to a PC with a Gigabit Ethernet card or to a Gigabit Ethernet switch.

Power (bench-top)

Please power the GLIB with 12V either by using the supplied cable or by a computer’s ATX power supply.

3.2 Firmware

Xilinx:

Please use Xilinx ISE v13.2 or above

How to download:

Windows: Do a checkout with an SVN client (e.g. TortoiseSVN) from the following path: https://svn.cern.ch/reps/ph-ese/be/amc_glib/tags and please use the latest release under this folder.

Linux: Type “svn co https://svn.cern.ch/reps/ph-ese/be/amc_glib/tags”. The firmware can be found under the directory “firmware”. The firmware can also be accessed using a web browser at https://svnweb.cern.ch/cern/wsvn/ph-ese/be/amc_glib/tags/ Please note only users with CERN account can download the firmware.

How to use it:

For using the existing example code as is, simply compile the *.xise project files that can be found under the firmware/projects. For developing your own code, please keep in mind that in order to receive support from the GLIB team, the files under firmware/src/system **MUST** remain unchanged. On the other hand, the files under the firmware/src/user can be freely modified according to the user needs.

3.3 Software

The SVN repository also contains several scripts in Python with self-explanatory names that are performing basic transactions. The scripts are using the PyChips library [11] developed for accessing IPbus based hardware in a simple way. In order to execute the scripts, please make that Python 2.x (minimum Python 2.3) is installed. PyChips is NOT compatible with Python 3.x. For the correct operation of the scripts, some variables have to be defined.

Under Windows, set the following environmental variables:

- PYTHONPATH with value *[installation_path]\amc_glib\software\[PyChips_distribution]\src* e.g. "E*:\amc_glib\software\Pychips_1_4_1\src"*
- PATH with value *[Python_2.x_installation_path]* e.g. "*C:\Python27*"

Under Linux, use the export command to define a variable with name PYTHONPATH and value "$HOME/*[PyChips_distribution]*/src:$PYTHONPATH" e.g
"export PYTHONPATH=$HOME/PyChips_1_4_1/src:$PYTHONPATH".

Additionally, in order to communicate successfully with the GLIB, the IP address of the Ethernet port has to be set in order to be in the same subnet e.g. for the default GLIB address (192.168.0.111) the configuration could be the following:

- IP Address: 192.168.0.100
- Subnet Mask: 255.255.255.0

In order to run the scripts, do the following:

Windows: go to the directory *installation_path]\amc_glib\software\[PyChips_distribution]\scripts, type the script's file name and press Enter.*

Linux: go to the directory $HOME/*amc_glib/software/[PyChips_distribution]/scripts, type the script's file name and press Enter.*

Brief description of the scripts:

glib_board_info.py: prints general board status information

glib_sysreg_test.py: example of r/w access of the system registers

glib_cdce_read.py: reads through SPI the contents of the CDCE62005 registers
glib_cdce_write.py: writes through SPI the GLIB default values to the CDCE62005 registers
glib_cdce_write_test_only.py: writes through SPI non-sense values to the CDCE62005 registers
glib_cdce_powerup_and_sync.py: power-cycles and synchronizes the CDCE62005

glib_sram1_read.py: reads the contents of the sram1, writes them to the tmp.txt and prints the first 32 values
glib_sram1_write.py: writes and incrementing by 1 value to all memory locations of sram1
glib_sram1_clear.py: writes “zeros” to all memory locations of sram1
glib_sram2_read.py: reads the contents of the sram1, writes them to the tmp.txt and prints the first 32 values
glib_sram2_write.py: writes and incrementing by 1 value to all memory locations of sram2
glib_sram2_clear.py: writes “zeros” to all memory locations of sram2

glib_i2c_eeprom_read_eui: reads the 48-bit serial number from the 24AA025E48 I2C EEPROM
glib_i2c_fmc_eeprom.py: example of r/w access of the I2C EEPROM carried by the FMC mezzanines
glib_i2c_temperature.py: reads the three on-board I2C temperature sensors
glib_phy_read.py: reads the registers of the M88E1111 PHY through I2C

glib_icap_interface_test.py:
glib_icap_trigg_fpga_conf_test.py:
glib_flash_program_test:

4. REFERENCES

[1] *P Vichoudis et al,* "The Gigabit Link Interface Board (GLIB), a flexible system for the evaluation and use of GBT-based optical links" *2010 JINST 5 C11007*

[2] *P. Moreira et al.,* "The GBT Project", in proceedings of the Topical Workshop on Electronics for Particle Physics TWEPP 2009, *CERN-2009-006*

[3] *L. Amaral et al.,"* The versatile link, a common project for super-LHC", *2009 JINST 4 P12003*

[4] *S. Bonacini et al.,* "e-link: A radiation-hard low-power electrical link for chip-to-chip communication", *in proceedings of* the Topical Workshop on Electronics for Particle Physics *TWEPP2009, CERN-2009-006*

[5] *J. Troska et al.,* "Versatile Transceiver developments", *2011 JINST 6 C01089*

[6] *Xilinx Inc. "Virtex-6 FPGA Clocking Resources" UG362*

[7] *Rob Frazier et al., "Software and firmware for controlling CMS trigger and readout hardware via gigabit Ethernet", in proceedings of the Technology and Instrumentation in Particle Physics TIPP 2011, to be published in Physics Procedia*

[8] *Dave Newbold, "Notes on Firmware Implementation of an IPBus SoC Bus", draft, 16/5/2011*

[9] *OpenCores "Wishbone B4 Specification" Spec*

[10] *S. Baron et al., "Implementing the GBT data transmission protocol in FPGAs", in proceedings of the Topical Workshop on Electronics for Particle Physics TWEPP-09, CERN-2009-006*

[11] *PyChips web page http://projects.hepforge.org/cactus/trac/wiki/PyChips*

APPENDIX A: User_logic-related custom data types

SRAM (from sram_package)

```
type userSramControlR is record
 reset              : std_logic;
 clk                : std_logic;
 cs                 : std_logic;
 writeEnable        : std_logic;
end record;

type userSramControlR_array is array(natural range <>) of userSramControlR;
```

FMC (from fmc_package)

```
type fmc_from_fabric_to_pin_type is record
 --- LA[0:33] ---
 la_lvds            : std_logic_vector(0 to 33);
 la_lvds_oe_l       : std_logic_vector(0 to 33);
 la_cmos_p          : std_logic_vector(0 to 33);
 la_cmos_p_oe_l : std_logic_vector(0 to 33);
 la_cmos_n          : std_logic_vector(0 to 33);
 la_cmos_n_oe_l : std_logic_vector(0 to 33);
 --- HA[0:23] ---
 ha_lvds            : std_logic_vector(0 to 23);
 ha_lvds_oe_l       : std_logic_vector(0 to 23);
 ha_cmos_p          : std_logic_vector(0 to 23);
 ha_cmos_p_oe_l: std_logic_vector(0 to 23);
 ha_cmos_n          : std_logic_vector(0 to 23);
 ha_cmos_n_oe_l: std_logic_vector(0 to 23);
 --- HB[0:21] ---
 hb_lvds            : std_logic_vector(0 to 21);
 hb_lvds_oe_l       : std_logic_vector(0 to 21);
 hb_cmos_p          : std_logic_vector(0 to 21);
 hb_cmos_p_oe_l : std_logic_vector(0 to 21);
 hb_cmos_n          : std_logic_vector(0 to 21);
 hb_cmos_n_oe_l: std_logic_vector(0 to 21);

end record;

type fmc_from_pin_to_fabric_type is record
 --- LA[0:33] ---
 la_lvds            : std_logic_vector(0 to 33);
 la_cmos_p          : std_logic_vector(0 to 33);
 la_cmos_n          : std_logic_vector(0 to 33);
 --- HA[0:23] ---
 ha_lvds            : std_logic_vector(0 to 23);
 ha_cmos_p          : std_logic_vector(0 to 23);
 ha_cmos_n          : std_logic_vector(0 to 23);
 --- HB[0:21] ---
 hb_lvds            : std_logic_vector(0 to 21);
 hb_cmos_p          : std_logic_vector(0 to 21);
 hb_cmos_n          : std_logic_vector(0 to 21);

end record;
```

WISHBONE (from wb_package)

```
type wb_mosi_bus is record -- The signals going from master to slave(s)
  wb_rst          : std_logic;
  wb_clk          : std_logic;
  wb_adr          : std_logic_vector(31 downto 0);
  wb_dat          : std_logic_vector(31 downto 0);
  wb_stb          : std_logic;
  wb_we           : std_logic;
  wb_sel          : std_logic_vector( 3 downto 0);
  wb_cyc          : std_logic;
end record;

 type wb_miso_bus is  record -- The signals going from slave(s) to master
  wb_dat          : std_logic_vector(31 downto 0);
  wb_ack          : std_logic;
  wb_err          : std_logic;
 end record;

type wb_mosi_bus_array is array(natural range <>) of wb_mosi_bus;
type wb_miso_bus_array is array(natural range <>) of wb_miso_bus;
```

GBT Tx (from gbt_package)

```
type gbt_enc_in is record
  reset           : std_logic;
  word_clk        : std_logic;
  frame_clk       : std_logic;
  data            : std_logic_vector ( 83 downto 0);
end record;

type gbt_enc_out is record
  word            : std_logic_vector ( 19 downto 0);
  frame           : std_logic_vector (119 downto 0);  -- debug only
  header          : std_logic;                        -- debug only
 end record;

type gbt_enc_in_array     is array(natural range <>) of gbt_enc_in;
type gbt_enc_out_array    is array(natural range <>) of gbt_enc_out;
```

GBT Rx (from gbt_package)

```
type gbt_dec_out is          record
  data            : std_logic_vector( 83 downto 0);
  data_dv         : std_logic;
  aligned         : std_logic;
  bit_slip_cmd    : std_logic;
  bit_slip_nbr    : std_logic_vector(4 downto 0);
  write_address   : std_logic_vector( 5 downto 0);    -- debug only
  word            : std_logic_vector(19 downto 0);    -- debug only
  frame           : std_logic_vector(119 downto 0);   -- debug only
  frame_dv        : std_logic;                        -- debug only
  header_flag     : std_logic;                        -- debug only
  shiftedword     : std_logic_vector(19 downto 0);    -- debug only
end record;
```

```
type gbt_dec_in is record
  reset            : std_logic;
  word_clk         : std_logic;
  frame_clk        : std_logic;
  word             : std_logic_vector(19 downto 0);
  gtx_aligned      : std_logic;
end record;

type gbt_dec_in_array     is array(natural range <>) of gbt_dec_in;
type gbt_dec_out_array    is array(natural range <>) of gbt_dec_out;
```

GTX_FOR_GBT (from gtx_package)

```
type gtx_in is record
  loopback         : std_logic_vector(2 downto 0);
  tx_powerdown     : std_logic_vector(1 downto 0);
  rx_powerdown     : std_logic_vector(1 downto 0);
  rxp              : std_logic;
  rxn              : std_logic;
  rx_sync_reset    : std_logic;
  rx_refclk        : std_logic;
  rx_reset         : std_logic;
  rx_slide         : std_logic;
  rx_slide_run     : std_logic;
  rx_slide_nbr     : std_logic_vector(4 downto 0);
  tx_refclk        : std_logic;
  tx_reset         : std_logic;
  tx_sync_reset    : std_logic;
  tx_data          : std_logic_vector(19 downto 0);
  drp_daddr        : std_logic_vector(7 downto 0);
  drp_dclk         : std_logic;
  drp_den          : std_logic;
  drp_di           : std_logic_vector(15 downto 0);
  drp_dwe          : std_logic;
  conf_diff        : std_logic_vector(3 downto 0);
  conf_pstemph     : std_logic_vector(4 downto 0);
  conf_preemph     : std_logic_vector(3 downto 0);
  conf_eqmix       : std_logic_vector(2 downto 0);
  conf_rxpol       : std_logic;
  conf_txpol       : std_logic;
end record;
type gtx_out is record
  rx_wordclk       : std_logic;
  rx_data          : std_logic_vector(19 downto 0);
  rx_slide         : std_logic;
  txp              : std_logic;
  txn              : std_logic;
  tx_wordclk       : std_logic;
  resetdone        : std_logic;
  phasealingdone : std_logic;
  drp_drdy          : std_logic;
  drp_drpdo        : std_logic_vector(15 downto 0);
end record;


type gtx_in_array         is array(natural range <>) of gtx_in;
type gtx_out_array        is array(natural range <>) of gtx_out;
```