CERN PH-ESE-BE

GLIBv3 user manual

version 1.61

2014.02.19

DRAFT

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

Contact: Paschalis.Vichoudis@cern.ch

Document History

- **v1.61, 2014.02.19:** Updated §3.2
- **v1.52, 2013.10.04:** Updated §3.2, added appendix B
- **v1.44, 2013.08.06:** GLIB with IPbus v2 and GBT-FPGA files from the associated SVN repositories.
- **v1.20, 2013.04.08:** Updated configuration section and added appendix B.
- **v1.09, 2012.02.19:** Updated documentation for release_v3.1.0 and above.
- **v1.00, 2012.12.20:** First draft in the document history.

GLIB team

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

Table of Contents

1.INTRODUCTION

The Gigabit Link Interface Board (GLIB) [1] is an FPGA-based system for users of high speed optical links in high energy physics experiments. The GLIB serves both as a platform for the evaluation of optical links in the laboratory as well as a triggering and/or data acquisition system in beam or irradiation tests of detector modules. The major hardware component of the platform is the GLIB Advanced Mezzanine Card (AMC) [2] that can be used either on a bench or in a µTCA [3] crate. The GLIB AMC is based on a Xilinx Virtex-6 FPGA with Multi-Gigabit Transceivers (MGT) operating at rates of up to 5 Gb/s. This performance matches the specifications of the Gigabit Transceiver (GBT) [4] and Versatile Link [5] [6] projects with targeted data rate of 4.8 Gb/s.

Figure 1-1 illustrates the baseline configuration of a GBT - Versatile Link - GLIB system is shown at the top. Front-end (FE) ASICs are electrically connected to the GBT ASIC through e-links [7] while the GBT high-speed serial data-streams are converted to/from the optical domain through the Versatile Transceiver [8]. At the other end, the GLIB system converts data to/from the optical domain, implements the GBT data transmission protocol [9] and codes/decodes the user payload at the link back-end. An alternative configuration, useful for intermediate prototyping, is shown in Figure 1-2 with one GLIB interfacing to FE ASICs and VTRx, thus emulating the GBT, and a second GLIB at the back-end.

Figure 1-3 shows a picture of the production version of the GLIB AMC, highlighting the two high-pin count (HPC) FMC Mezzanine Card (FMC) [10] sockets. The presence of the HPC FMC sockets is a big advantage since they provide additional user-specific I/O, high-speed transceivers and clock lines that can be used to extend the I/O connectivity of the GLIB AMC. For that reason, most of the auxiliary boards developed for the GLIB platform adopt the FMC format. The purpose of the auxiliary boards is to enhance the GLIB AMC compatibility with legacy and future triggering and/or data acquisition interfaces as well as its I/O bandwidth when in bench-top operation.

.

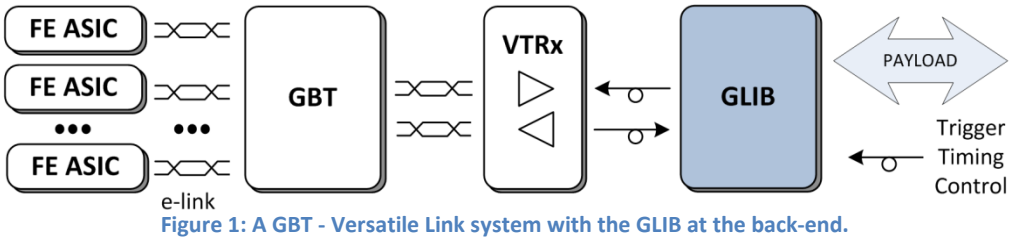

Figure 1: A GBT - Versatile Link system with the GLIB at the back-end.

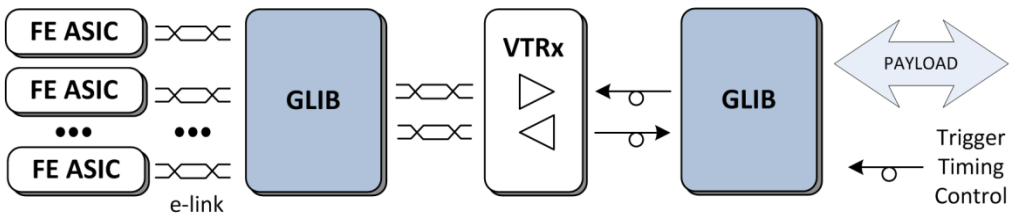

Figure 2: A GLIB interfacing to FE ASICs and VTRx with a second GLIB at the back-end.

Figure 3: Picture of the GLIB AMC, highlighting the two FMC sockets

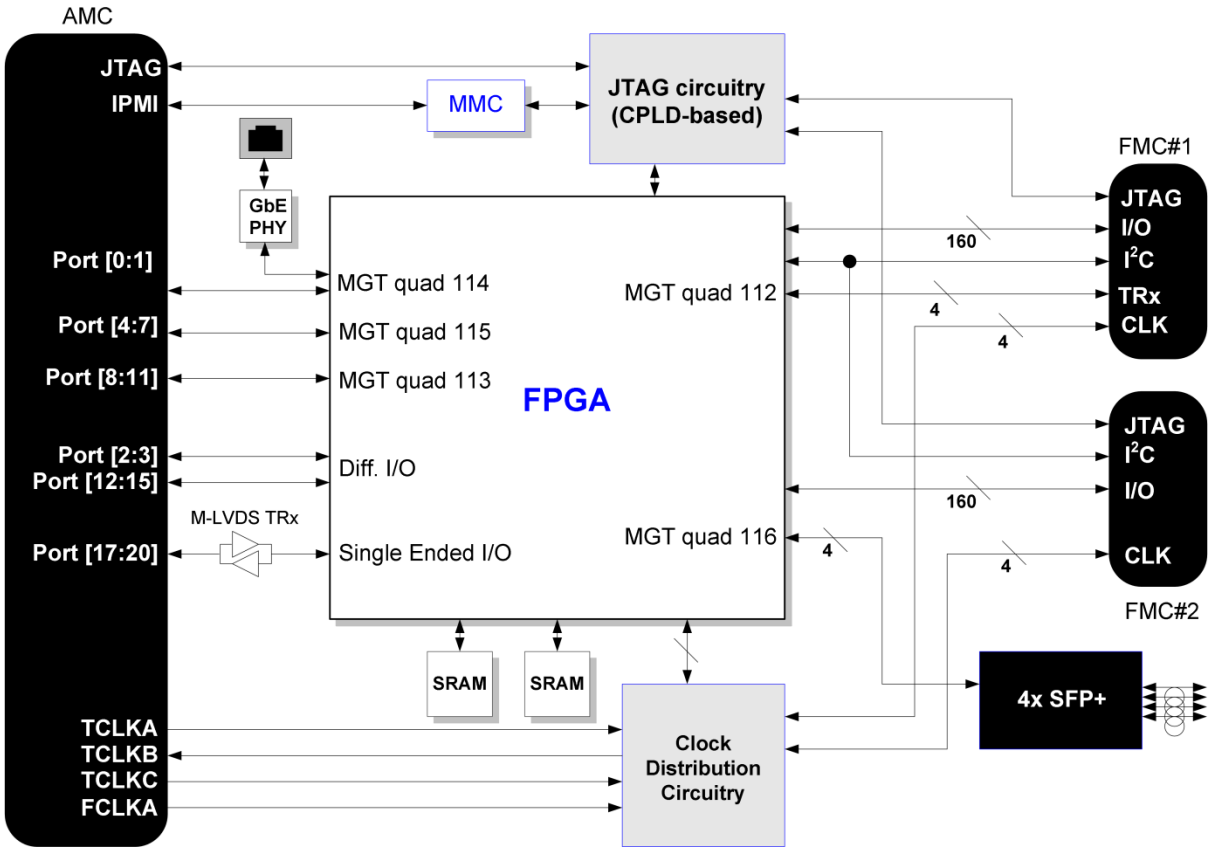

Figure 4: The block diagram of the GLIB card

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

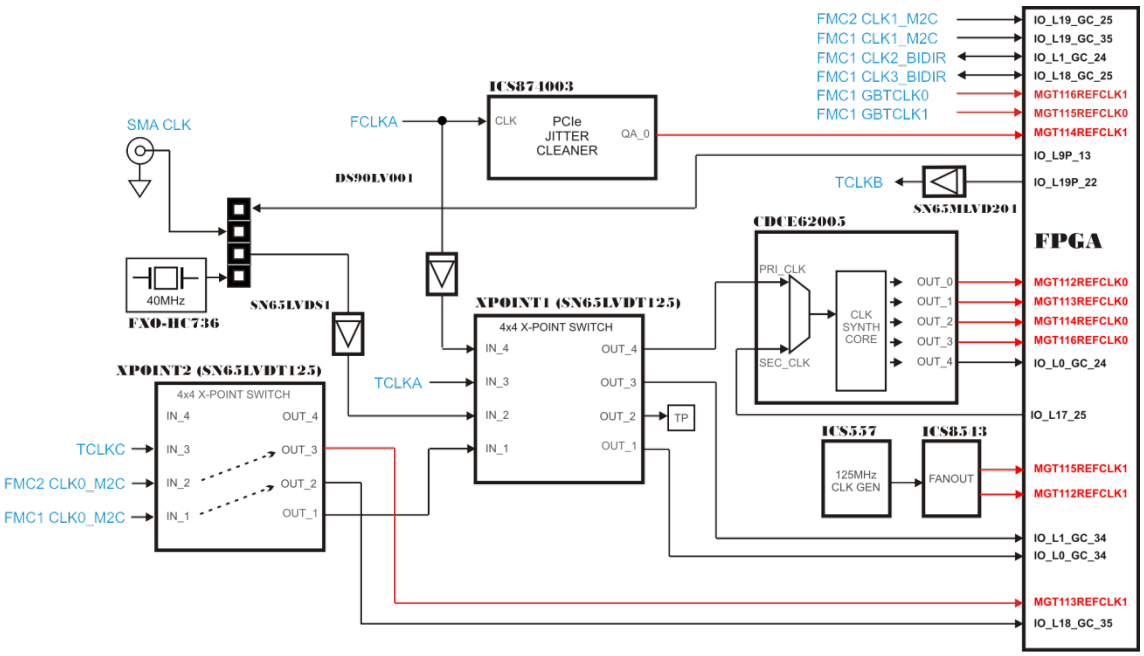

Figure 5: The clocking circuitry

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 Figure 1-6.

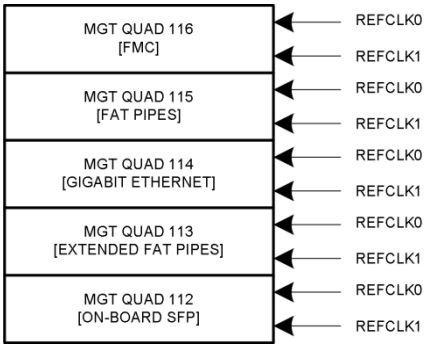

Figure 6: The MGT quads

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*.

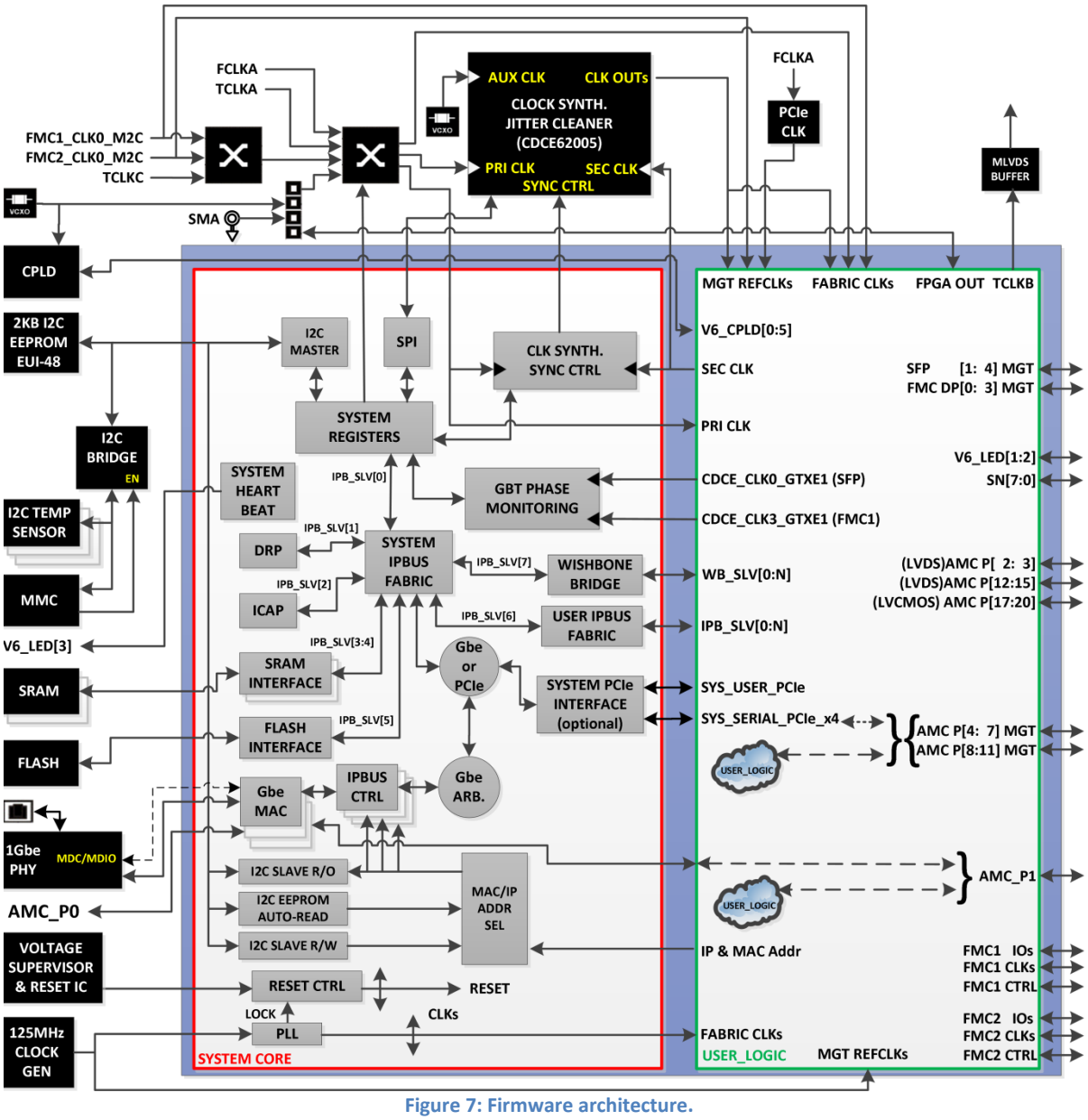

Figure 7: Firmware architecture.

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 on-board hardware e.g. I^2C 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.

2.1 System Core

2.1.1 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 §2.1.3) 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.

2.1.2 Gigabit Ethernet and IPbus

For the Gigabit Ethernet links, MAC cores are instantiated. 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, the two MAC cores (AMC P0 & P1[1]) 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 [12]. This protocol is based upon the Wishbone SoC protocol [13], 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.

In addition to IPbus's "DHCP-like" IP address assignment, IP & MAC addresses can also be assigned by either the *user_logic* block or the I2C EEPROM. Finally, there is also the possibility to be configured on-the-fly by the MMC through I2C. The default IP address of the GLIB when used on a bench is 192.168.0.175. The default IP address when used in a µTCA shelf ranges from 192.168.0.161 to 192.168.0.172 for AMC slots 1 to 12, respectively.

[1] The generation of AMC P1 MAC core when in crate mode is optional

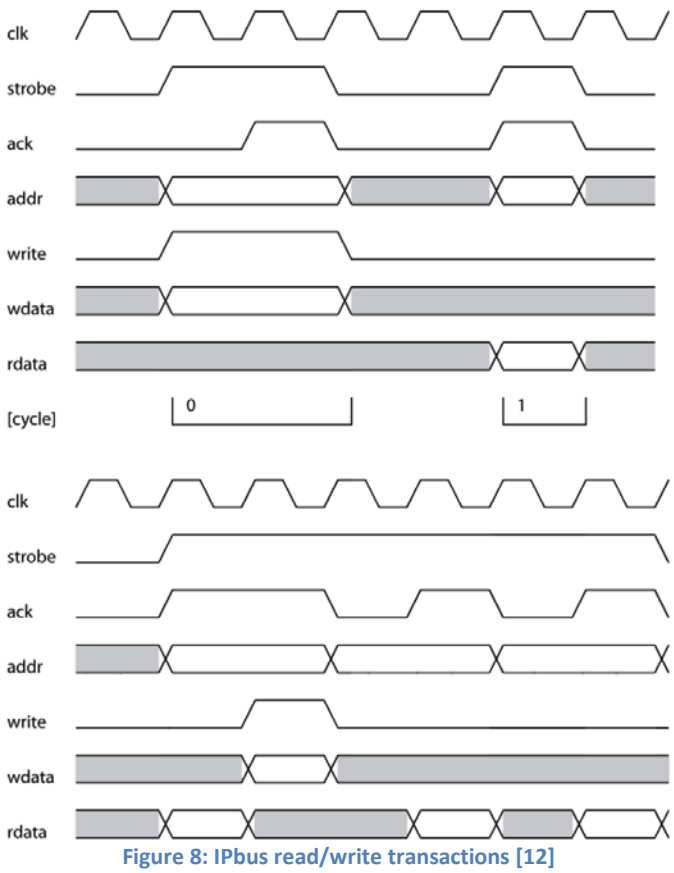

Figure 8: IPbus read/write transactions [12]

2.1.3 SoC bus

As mentioned in §2.1.2, 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*.

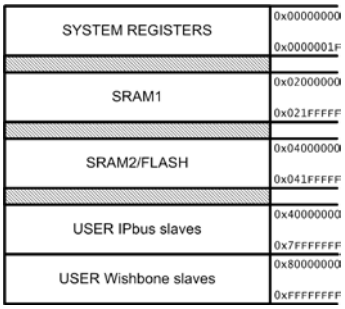

Figure 9: Memory map

2.1.4 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* when represented in ASCII characters corresponds to "ip2x" declaring that the second version of IPbus firmware is used. 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 2-7) 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*	Flash control	RW
0x06	*Status*	Status from various external components	RO
0x07	*Status2*	Currently not used	RO
0x08	*Ctrl_SRAM*	SRAM interface: Control	RW
0x09	*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 2-8) 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]	*tclkb_dr_en*	Enables the TCLKB output towards the backplane. 0 -> off. 1 -> on
[12]	*xpoint2_s10*	Input select for IC23 (SN65LVDT125) OUT1. See Table 2-6
[13]	*xpoint2_s11*	Input select for IC23 (SN65LVDT125) OUT1. See Table 2-6
[16]	*xpoint1_s10*	Input select for IC28 (SN65LVDT125) OUT1. See Table 2-6
[17]	*xpoint1_s11*	Input select for IC28 (SN65LVDT125) OUT1. See Table 2-6
[18]	*xpoint1_s20*	Input select for IC28 (SN65LVDT125) OUT2. See Table 2-6
[19]	*xpoint1_s21*	Input select for IC28 (SN65LVDT125) OUT2. See Table 2-6
[20]	*xpoint1_s30*	Input select for IC28 (SN65LVDT125) OUT3. See Table 2-6
[21]	*xpoint1_s31*	Input select for IC28 (SN65LVDT125) OUT3. See Table 2-6
[22]	*xpoint1_s40*	Input select for IC28 (SN65LVDT125) OUT4. See Table 2-6
[23]	*xpoint1_s41*	Input select for IC28 (SN65LVDT125) OUT4. See Table 2-6

Table 2-6: *xpoint1 input select*

Sx0	Sx1	Select Input	Example
0	*0*	IN_1 (see Figure 1-5)	When S40 = 0 & S41 = 0 then IN_1 -> OUT_4
0	*1*	IN_2 (see Figure 1-5)	When S40 = 0 & S41 = 1 then IN_2 -> OUT_4
1	*0*	IN_3 (see Figure 1-5)	When S40 = 1 & S41 = 0 then IN_3 -> OUT_4
1	*1*	IN_4 (see Figure 1-5)	When S40 = 1 & S41 = 1 then IN_4 -> OUT_4

Table 2-7: *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 2-8: *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 *Ctrl_SRAM* and *Status_SRAM* are presented in §2.2.1.

The registers *I2C_settings*, *I2C_command* and *I2C_reply* are controlling the operation of the 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 2-9 presents the I2C devices and their corresponding I2C 7-bit slave addresses. The operation of the above mentioned registers is the following:

- The *I2C_settings* register used for configuring the controller (Table 2-10).
- The *I2C_command* for setting the I2C transaction parameters (Table 2-11).
- The *I2C_reply* provides the transaction status as well as data sent by the slave in case of read transactions (Table 2-12).

Table 2-13 gives an example list of transactions needed for reading the temperature of the FPGA with an I2C bus frequency of 62.5kHz.

Table 2-9: 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
I2C Slave RW	Configures the IP and/or MAC address on-the-fly	1111010
I2C Slave RO	Reports the current IP & MAC address	1111110
FMC#1	The I2C devices hosted on the FMC#1	XXXXX00
FMC#2	The I2C devices hosted on the FMC#2	XXXXX11

Table 2-10: "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 2-11: "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 2-12: "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 2-13: 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

The registers *SPI_txdata*, *SPI_command* and *SPI_rxdata* are controlling the operation the SPI master controller that is used for the configuration of the CDCE62005 clock synthesizer. The operation of the above mentioned registers is 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 2-14).

Table 2-15 shows an example list of transactions for reading one of CDCE62005's registers. Table 2-16 shows an example of writing one of CDCE62005's registers.

Table 2-14: 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 2-15: 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 2-16: 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 as well as in the instantiation of GBT-based links.

2.2.1 SRAM

The two single-port 2Mx36 SRAM devices that are available on board are accessible both by the *system_core* and the *user_logic*. The register *Ctrl_SRAM* (Table 2-17) 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 2-18) 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 2-4, Figure 2-5, Figure 2-6 and Figure 2-7, respectively. Figure 2-8 shows an example use of an SRAM in the *user_logic* block.

Table 2-17: "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 2-18: "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.

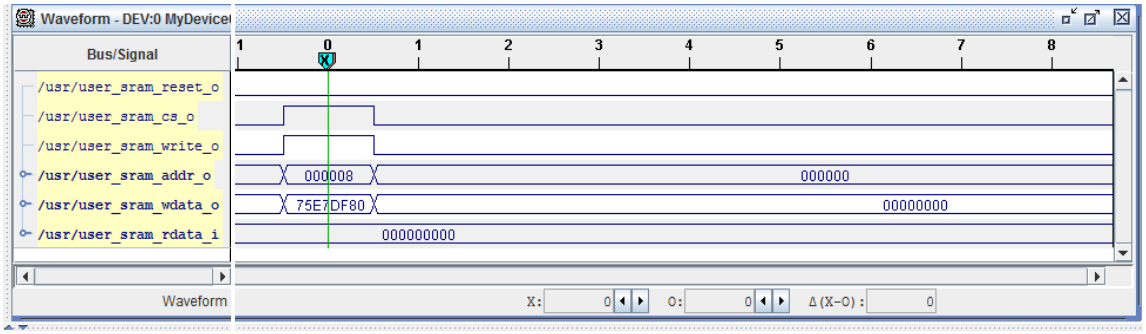

Figure 10: *User_logic* SRAM single write

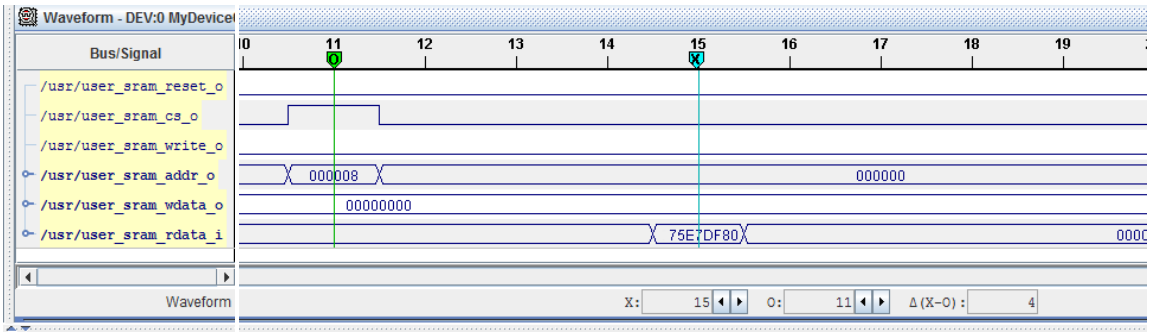

Figure 11: *User_logic* SRAM single read

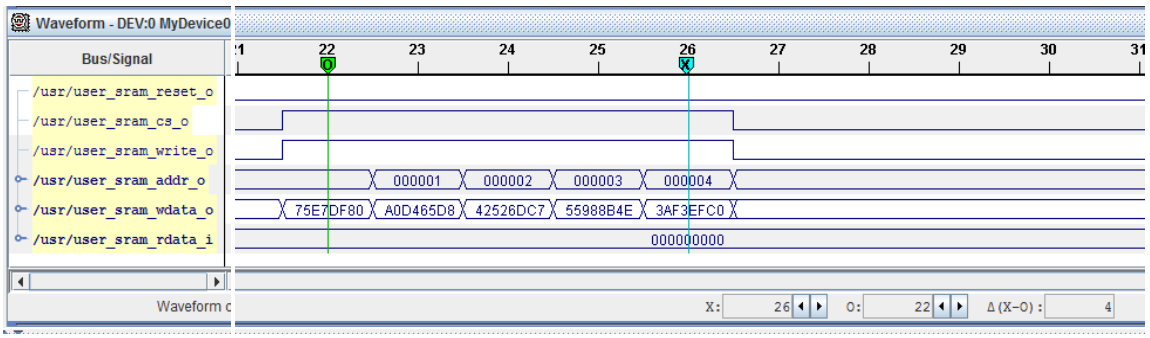

Figure 12: *User_logic* SRAM block write

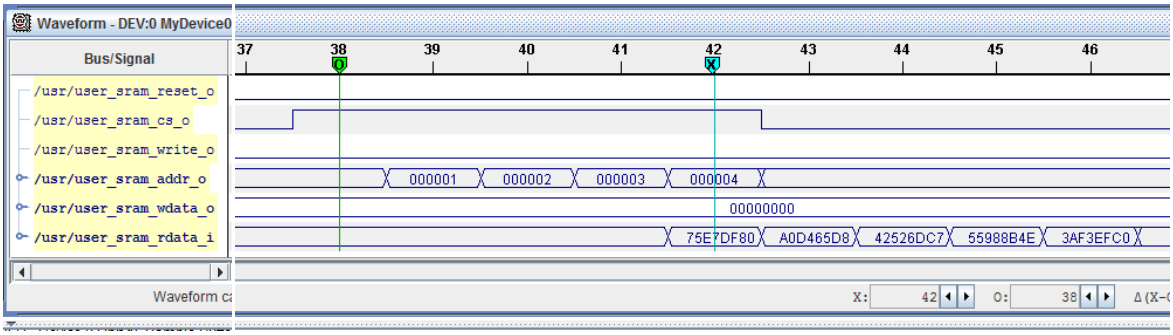

Figure 13: *User_logic* SRAM block read

```
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"0000008"; -- 25 bit addr
--------------------------------------------------------------------------------
```

Figure 14: Example use of the SRAM in the *user_logic* file

2.2.2 FMC I/O

All FMC I/Os are directly connected to the *user_logic* block and can be configured independently as LVCMOS 2.5V or LVDS inputs or outputs by simply instantiating the corresponding Xilinx primitive buffer such as IBUF, OBUF, IOBUF, IBUFDS, OBUFDS etc. (see Figure 2-9).

```
--================================================================================
fmc1_la_p_15_iobuf: iobuf
generic map  (iostandard => "lvcmos25")
port map     (io => fmc1_io_pin.la_p(15), i  => scl_o, o => scl_i, t => scl_oe_l);
--================================================================================
```

Figure 15: Example I/O buffer instantiation in the *user_logic* for the LA15_P line of FMC1 using Xilinx primitives

2.2.3 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 2-10 shows an example of how to declare the total number of wishbone slaves (and their symbolic names). Figure 2-11 shows the address mapping of the wishbone slaves. Figure 2-12 illustrates an example of a wishbone slave instantiation in 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 16: 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, "10000000000000000000000--------") 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 17: 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 18: Example instantiation of a wishbone slave in the *user_logic* file

2.2.4 GBT-based links

As mentioned in the introduction, one of the principle use cases of the GLIB is to be part of a GBT – Versatile Link system for evaluation purposes. For that reason, the GLIB firmware release includes the example project *glib_v3_gbt_fpga* that instantiates inside the *user_logic* block the GBT-FPGA reference design for Virtex-6 as delivered by the GBT-FPGA team. The source files and associated documentation can be found in [16].

3.How to use the GLIB

3.1 Hardware

3.1.1. Jumpers

The operation mode of the GLIB is set up through two jumpers (J5 and J10). The position of the jumpers for the two modes of operation is shown below.

Bench-top operation

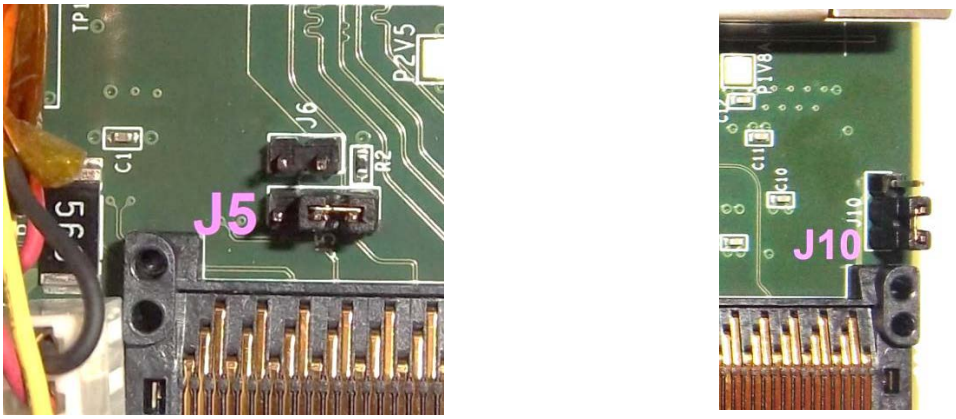

Figure 3-19: Jumpers J5 and J10 set up for bench-top operation.

Crate operation

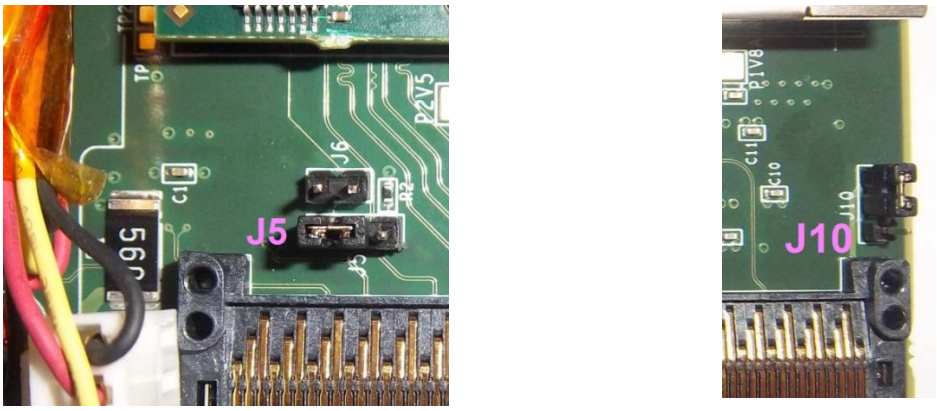

Figure 3-20: Jumpers J5 and J10 set up for crate operation.

External Clocking

As previously illustrated in Figure 1-5 and Figure 2-1, the GLIB features an SMA connector for external clocking. The connection scheme of the SMA connector (SMA1) and the 40MHz crystal oscillator (QZ2) of the GLIB is set up through the jumper J14. The default configuration sets the SMA connector as input/output of the FPGA and the 40Mhz crystal oscillator as input for the cross point switch 1 (IC28).

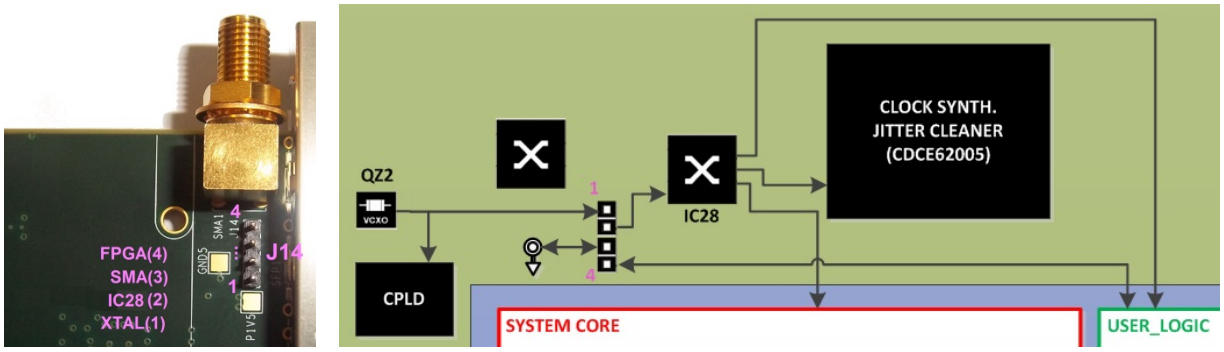

Figure 3-21: SMA/Jumper J14 picture (left). SMA and 40 MHz crystal oscillator (QZ2) setup block diagram (right)

3.1.2. Switches

The default position of the DIP switches is as illustrated below. All other combinations are reserved.

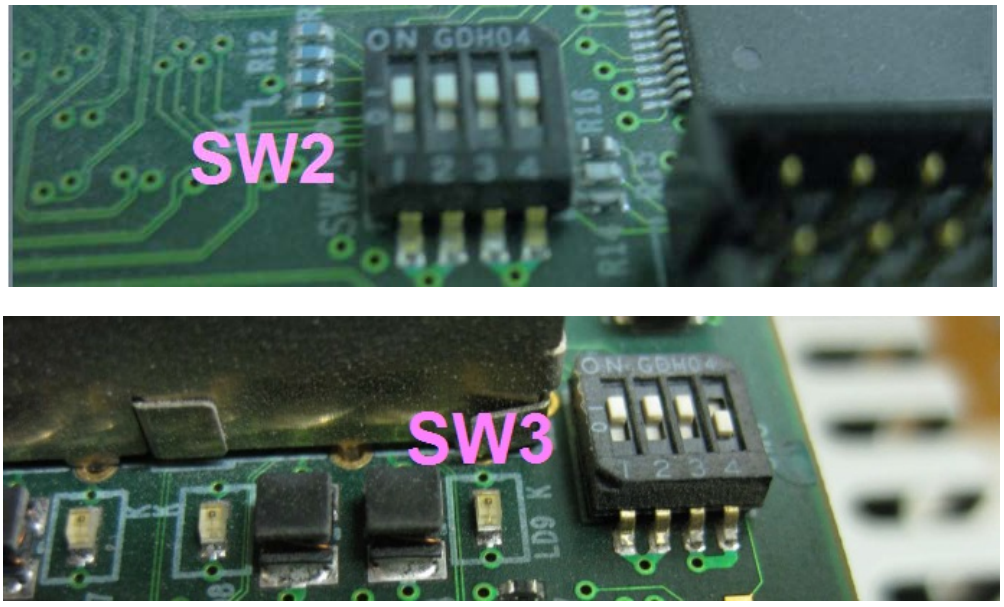

Figure 3-22: Default switches positions.

3.1.3. Powering

For bench-top operation, the GLIB can be powered through a socket compatible with the 12V 4-pin connector that is typically available in ATX compatible power supplies (Figure 3-5 left and right, respectively). In case of ATX power supply use, the PS-ON pin of the 20-pin ATX connector must be connected to ground as shown in Figure 3-6 (bottom) in order to start-up the power supply.

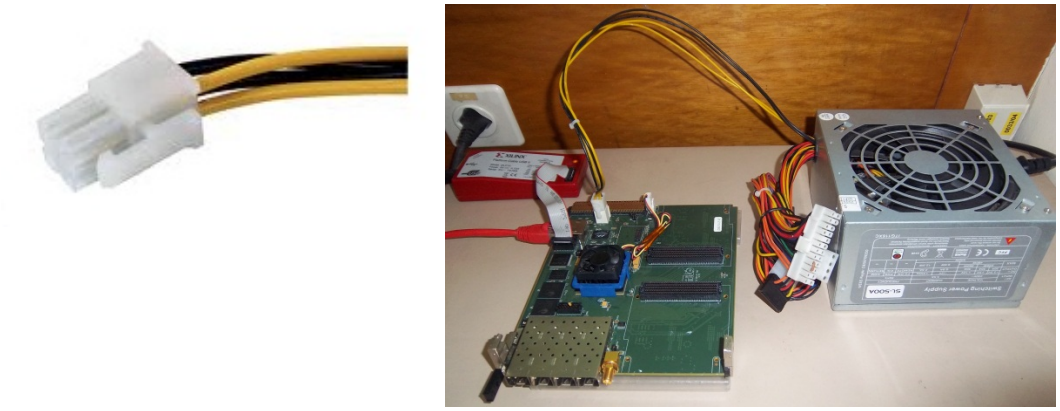

Figure 3-23: Powering a GLIB with an ATX power supply.

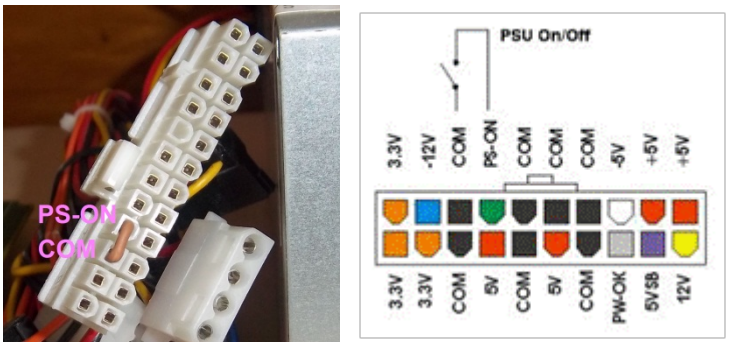

Figure 3-24: How to start-up an ATX Power supply outside a computer.

3.1.4. Configuration

JTAG connectors

The GLIB features two JTAG connectors. J13 configures the CPLD & J12 configures the FPGA.

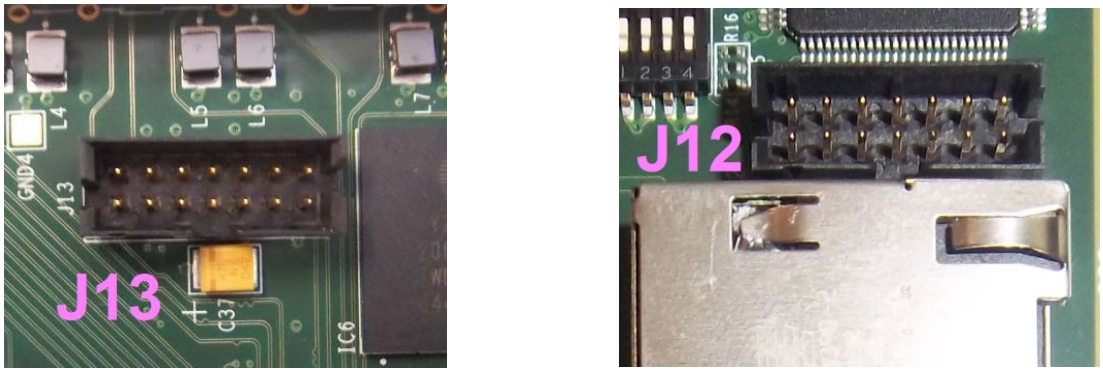

Figure 3-25: JTAG connectors for configuring the CPLD (left) and FPGA (right).

FPGA configuration scheme

For advanced FPGA configuration schemes please see Appendix B.

3.1.5. Reset

As mentioned in §2.1.1, the GLIB board features a button for resetting the FPGA logic. The button (SW1) is located in between the two FMC sockets and it is connected to a TLC7725 power supervisor (IC14). When the button is pressed, the power supervisor generates a pulse that is used to reset the *system_core*. Additionally, the reset signal is forward to the *user_logic* block where it can be freely used.

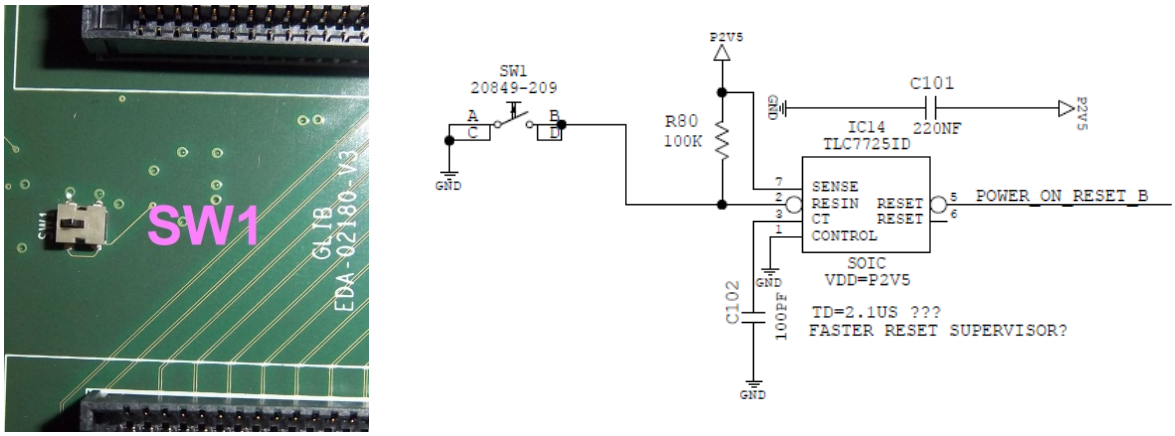

Figure 3-26: Reset button picture and reset scheme schematic.

3.1.6. RJ45 socket

The RJ45 socket (and associated circuitry) carried by the GLIB board provides a Gigabit Ethernet connection in case of bench-top operation. Please note that since the GLIB supports only Ethernet at 1Gbps (and not 10/100Mbps), please connect it using standard (not crossover) network cables either directly to a PC with a Gigabit Ethernet card or to a Gigabit Ethernet switch.

3.1.7. Serial Number

As previously mentioned in Table 2-9, the GLIB carries a serial memory with a 48-bit unique serial number that could be used to identify the GLIB boards. Furthermore, the GLIB offers the possibility to the user of an additional 8-bit serial number that is directly accessible from the *user_logic* block and it can be hard-coded by placing up to 8 resistors on the pads shown in Figure 3-9.

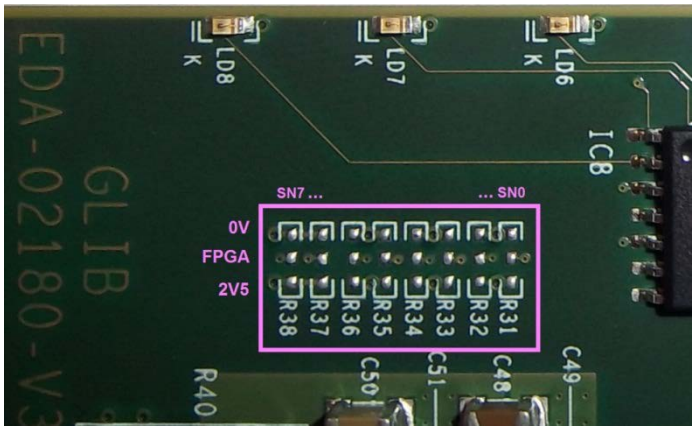

Figure 3-27: Resistors pads for optional hard-coding of 8-bit serial number.

3.1.8. LEDs

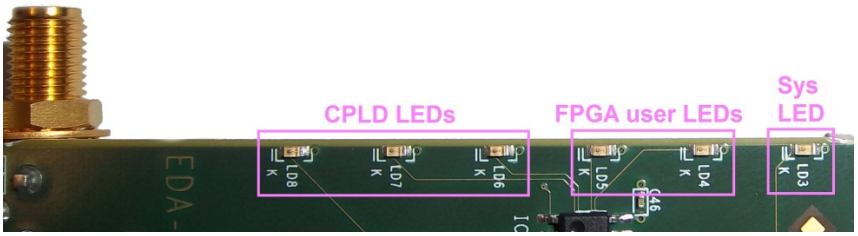

Figure 3-28: CPLD, FPGA user and system LEDs

System LED

The system core blinks the system LED (LD3) to indicate that the GLIB firmware has been loaded correctly.

FPGA user LEDs

The FPGA user LEDs (LD4 and LD5) can be directly controlled by the logic placed on the user core.

CPLD LEDs

The LEDs LD6, LD7, LD8 are connected to the CPLD. The meaning of these LED depends on the CPLD firmware version.

MMC LEDs

When in crate operation, the LEDs MMC_RED_LED (LD1), MMC_GREEN_LED (LD2) and MMC_BLUE_LED (LD9) are controlled by the MMC and indicate the state of the GLIB.

Table 19: MMC LEDs and GLIB state relationship

	GLIB state	
MMC LEDs	**Inactive**	**Active**
MMC_RED_LED	*ON*	OFF
MMC_BLUE_LED	*ON*	OFF
MMC_GREEN_LED	*OFF*	ON

Figure 3-29: Inactive GLIB (left) and active GLIB (right)

3.2 Firmware/Software

Firmware:

In order to access all files required, you will need to do the following:

- Checkout *amc_glib* repository [https://svn.cern.ch/reps/ph-ese/be/amc_glib] to a folder e.g. *dev*.
- Checkout *cactus* repository [https://svn.cern.ch/reps/cactus/] into *dev*.
- Checkout the *gbt-fpga* repository [https://svn.cern.ch/reps/ph-ese/be/gbt_fpga] into *dev*.
- Checkout the *mmc* repository [https://svn.cern.ch/reps/ph-ese/be/mmc] into *dev*.

The folder tree should be as shown in Figure 3-12

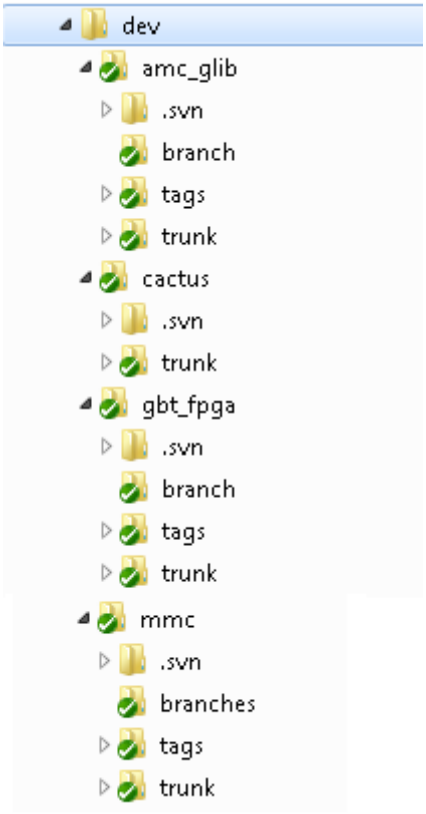

Figure 3-30: Folder tree

The GLIB team recommends using Xilinx ISE v14.5 for firmware development and implementation.

For using the existing example code as is, simply compile the *.xise project files that can be found under *[..]\glib_v3\fw\fpga\prj*. For developing your own code, please keep in mind that in order to receive support from the GLIB team, the files under *[..]\glib_v3\fw\src\system* **MUST** remain unchanged. On the other hand, the files under *[..]\glib_v3\fw\src\user* can be freely modified according to the user needs. Concerning the CPLD & MMC firmware, we provide the source files for your reference **only** and we strongly suggest not modifying them.

IMPORTANT NOTICE: The GLIB boards are delivered with the FLASH memory empty.

Software:

For software development, **it is strongly recommended to use the µHAL** [17] software distribution, also available in the CACTUS SVN repository. For hardware verification purposes **only**, we also provide a few Python scripts performing basic test functions that must be used *as is*. Please note that the scripts are using a Python library namely "Pychips" that is not supported any longer by its developers, therefore you should avoid in any cost to base your developments on that platform. In order to execute the hardware verification scripts, please make sure that Python 2.7.x 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, define the following environmental variables:

- PYTHONPATH with value the path pointing to the *[..]\glib_v3\sw\PyChips\src* folder.
- PATH with value the path pointing to the Python installation folder e.g. "*C:\Python27*"

Under Linux, use the export command to define the PYTHONPATH variable. All hardware verification scripts as well as the GLIB IP address and register address table declarations are located at *[..]\glib_v3\sw\PyChips\scripts*

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 addresses the configuration could be 192.168.0.10 as IP address and 255.255.255.0 as subnet mask.

Brief description of the hardware verification 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.py: reads the 48-bit serial number from the 24AA025E48 I2C EEPROM
glib_i2c_temperature.py: reads the three on-board I2C temperature sensors
glib_i2c_mac_ip_eeprom.py: programs the memory space of the I2C EEPROM allocated for the IP & MAC address assignment.
glib_i2c_mac_ip_ctrl.py: accesses the FPGA's "*I2C slave R/W*" in order to program on-the-fly the IP & MAC address (for test purposes only, this procedure is normally reserved for the MMC)
glib_i2c_mac_ip_status.py: accesses the FPGA's "*I2C slave R/O*" in order to read the actual IP & MAC address (for test purposes only, this procedure is normally reserved for the MMC)
glib_icap_interface_test.py: reads and writes several registers of the icap
glib_icap_jump_to_image.py: triggers the reconfiguration of the FPGA (used for Multiboot/Safe configuration)
glib_flash_golden_prom_rw.py: writes the Golden image PROM file to the FLASH or reads the Golden image on FLASH and dumps it to a file
glib_flash_user_prom_rw.py: writes the User image PROM file to the FLASH or reads the User image on FLASH and dumps it to a file
glib_flash_corrupt_user_image.py: corrupts the User image stored on FLASH, for testing purposes only

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] *PICMG, AMC.0 R2.0 (November 15, 2006), http://www.picmg.com/v2internal/specifications2.cfm?thetype=One&thebusid=1*

[3] *PICMG, MTCA.0 R1.0 (July 6, 2006), http://www.picmg.com/v2internal/specifications2.cfm?thetype=One&thebusid=5*

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

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

[6] *F. Vasey et al., The versatile link common project: feasibility report, 2012 JINST 7 C01075*

[7] *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*

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

[9] *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*

[10] *ANSI/VITA, 57.1-2008 (R2010), http://www.vita.com/fmc.html*

[11] *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*

[12] *Rob Frazier et al., "An IP-based control protocol for ATCA/µTCA", IPbus version 2.0 Draft 6, 22/3/2013 https://svnweb.cern.ch/cern/wsvn/cactus/trunk/doc/ipbus_protocol_v2_0.pdf*

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

[14] *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*

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

[16] *GBT-FPGA firmware releases https://svnweb.cern.ch/cern/wsvn/ph-ese/be/gbt_fpga*

[17] *CERN Cactus SVN/Wiki https://svnweb.cern.ch/trac/cactus/wiki*

APPENDIX A: GLIB v3 FPGA configuration scheme

Introduction

In most FPGA applications, one bitstream (FPGA image) is programmed onto the FLASH memory and the FPGA is configured when the system is powered on. This is the simplest FPGA configuration scheme possible when using a FLASH memory for storing the configuration file (from now on it will be referred as **single image FPGA configuration scheme**). However, there is an intrinsic safety problem when using this scheme because either a bitstream corruption or an incorrect firmware may lead to a system crash (see Figure A 1).

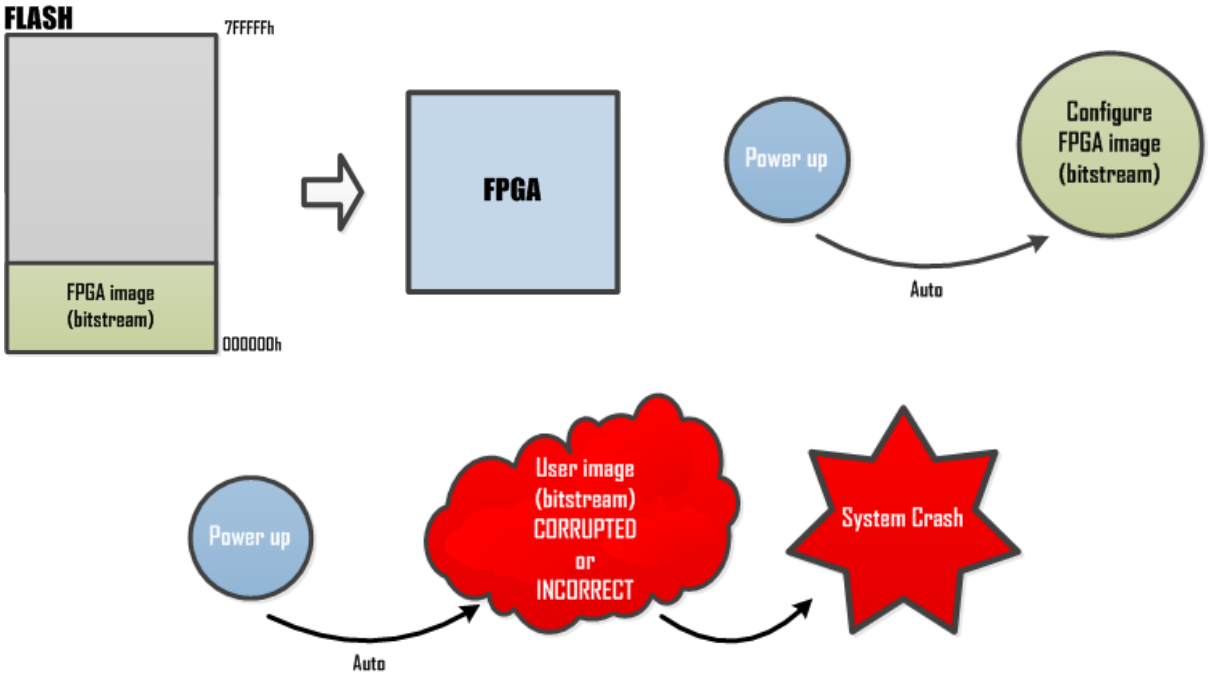

Figure A 1: Standard FPGA configuration scheme.

In order to provide a reliable FPGA configuration procedure, the GLIB board features the **Multiboot/Safe FPGA configuration scheme**. This scheme requires two different bitstreams stored on different memory positions of the FLASH (see Figure A 2). One of the bit streams, the "Golden image", is used as backup firmware enabling the possibility of remotely restoring/updating the system when needed. The other bitstream, the "User image", is the main firmware which comprises the actual system implemented by the user.

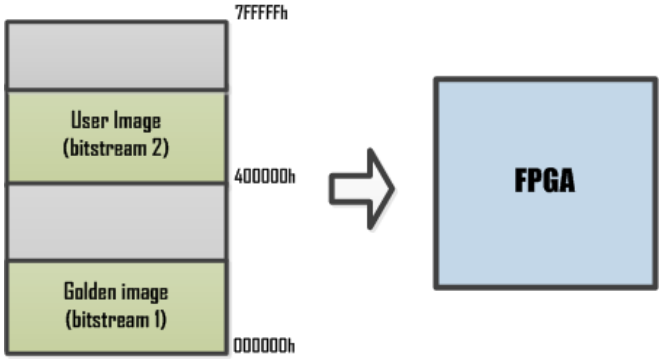

Figure A 2: Golden and User images on FLASH.

The handling of these bit streams by the GLIB system can be done through either the Internal Configuration Access Port (ICAP) or by pulling up the address 22 pin of the FLASH memory (A22).

Figure A 3 shows the **ICAP approach** (default setup), where the FPGA is initially loaded with the Golden image during power up. Once the FPGA is configured, the User image can be loaded on-demand through the ICAP interface module that is controlled via IPbus. If for any reason the User image is corrupted, the FPGA will automatically trigger a "Fallback configuration" after the failed configuration process, reverting to the Golden image. Moreover, if the User image is not corrupted but it does not behave as expected ("incorrect"), it is possible to revert to the Golden image by either another ICAP command or by power cycling (in case of a User image with non-functioning IPbus/ICAP interface). As mentioned before, once the Golden image is loaded, the system is again under control so a correct User image may be reprogrammed onto the FLASH and loaded to the FPGA on-demand (see Figure A 4).

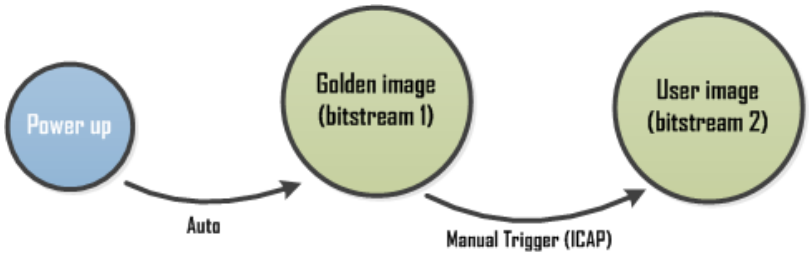

Figure A 3: Multiboot/Safe FPGA configuration scheme (ICAP approach).

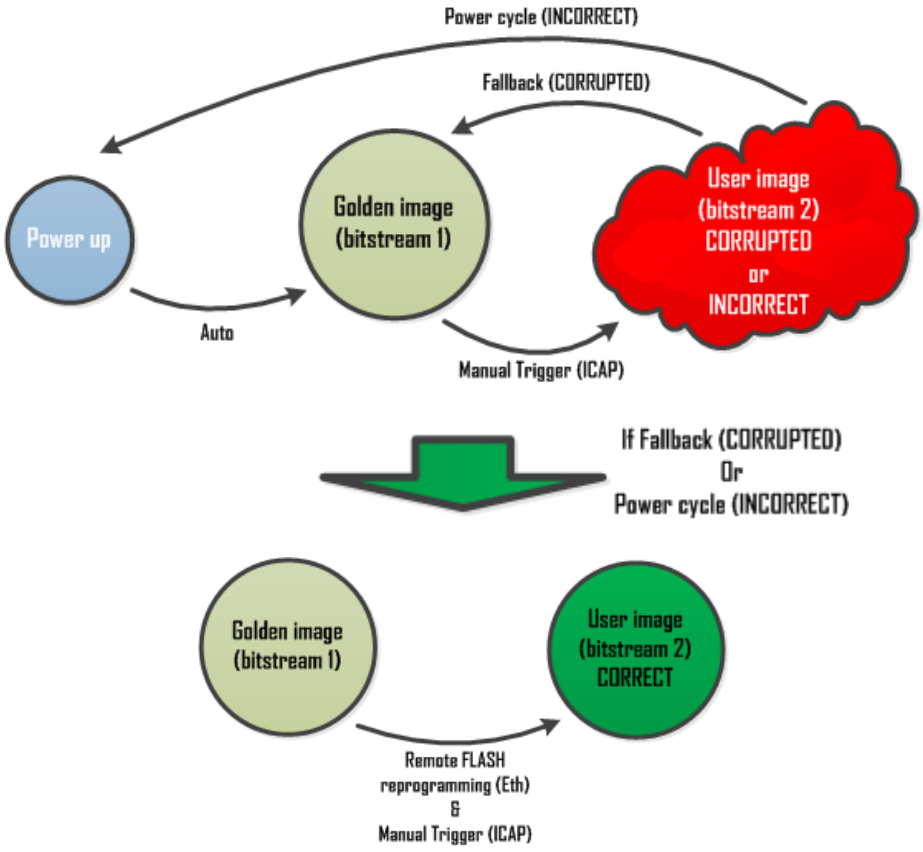

Figure A 4: Multiboot/Safe FPGA configuration system recovery (ICAP approach).

Although the ICAP is the recommended Multiboot/Safe FPGA configuration scheme approach due to the reliability provided by the Fallback and the Power cycle recovery procedures, there are some applications where the configuration time is a critical factor (such as PCIe) so the ICAP approach may not meet the requirements of the system. For this kind of applications, the user has the possibility of using the **A22 pin pull-up approach** (see Figure A 5) which allows the shortest configuration times, even if loosing reliability. In this case, the A22 pin of the FLASH memory is pulled up during power up so the FPGA is directly configured with the User image, avoiding the extra time of the Golden image configuration and the manual triggering of the User image. However, as aforementioned, this approach is not as good as the previous (ICAP) in terms of reliability because it only allows system recovery when the bitstream is corrupted (Fallback configuration). In case of an "incorrect" User image (e.g. just and LED blinking and no ICAP instead of the user's system), the power cycle procedure will no longer be a solution because in this approach, the Golden image is not the bitstream loaded after power up (see Figure A 6).

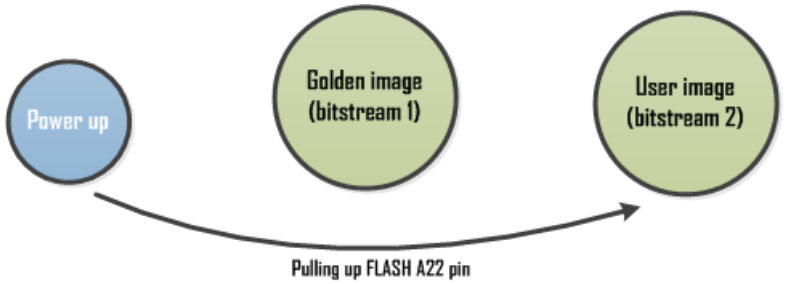

Figure A 5: Multiboot/Safe FPGA configuration scheme (A22 pin pull-up approach).

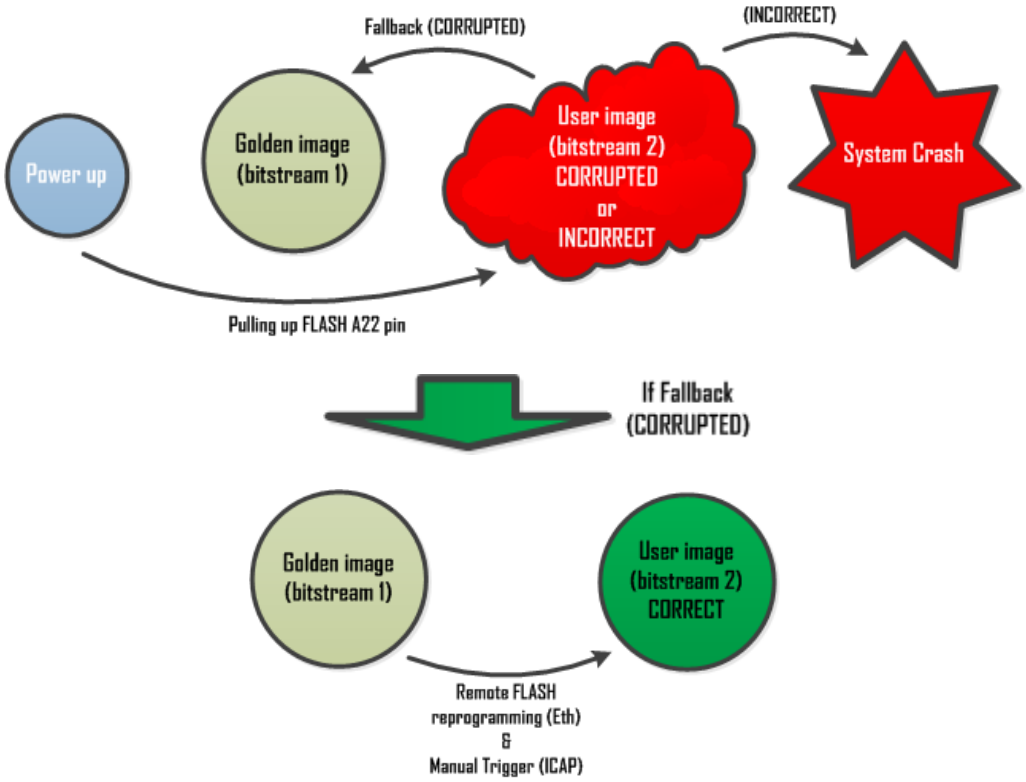

Figure A 6: Multiboot/Safe FPGA configuration system recovery (A22 pin pull-up approach).

Implementation on GLIB and utilisation

Figure A 7 shows the circuitry that implements the above mentioned configuration schemes where the CPLD serves as multiplexor between the FPGA and the FLASH memory. During the first configuration after power up, the A22 pin of the FLASH is pulled high or low, depending on the position of the switch #2 of the DIP switch SW2. If **pulled low** (SW2(2) = on) (default position), the first bitstream loaded by the FPGA is either the FPGA image (single image FPGA configuration scheme) or the Golden image (Multiboot/Safe FPGA configuration scheme with ICAP approach) depending on the FPGA configuration scheme chosen by the user. In case of the Golden image is loaded, the user may jump to the User image when needed by running the appropriate python script (see Table A-1). On the other hand, if the switch is **pulled high** (SW2(2) = off), the first bitstream loaded by the FPGA after power up is the User image (Multiboot/Safe FPGA configuration scheme with A22 pin pull-up approach).

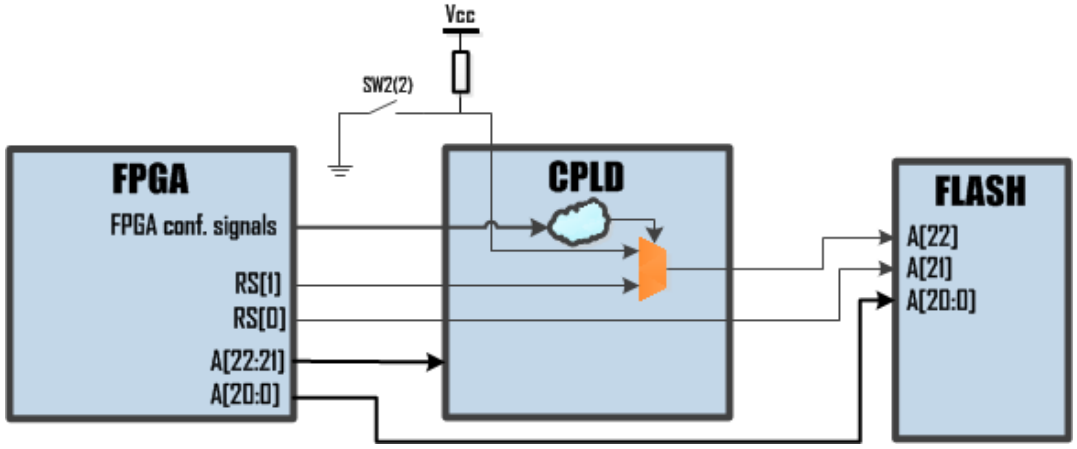

Figure A 7: GLIB configuration scheme.

Regarding to the programming of the FLASH memory, if the user decides to use the Multiboot/Safe FPGA configuration scheme, it is recommended to program the different images by utilising the python scripts provided for this purpose with the GLIB software. There is a programing script for the Golden image as well as other for the User image (see Table A-1). Both scripts use as PROM file a standard MSC file set up for BPI Flash/Single FPGA configuration (the same MCS file set up used for the single image FPGA configuration scheme) which can be generated using iMPACT (see Figure A 8). The user does not need to care about the memory position of the PROM file on the FLASH because it is automatically handled by the scripts.

Table A-1: Multiboot/Safe FPGA configuration main python scripts.

Description	Script name
Golden image programming script	glib_flash_golden_prom_rw.py
User image programming script	glib_flash_user_prom_rw.py
ICAP jump script	glib_icap_jump_to_image.py

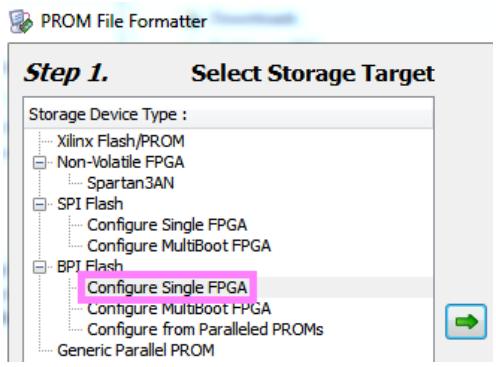

Figure A 8: iMPACT PROM File Formatter.

APPENDIX B: GLIB IP address assignment scheme in a shelf

As mentioned briefly in §2.1.2, the GLIB proposes a simplified scheme of IP address assignment in a shelf based on the slot number (firmware example available in the SVN repository) that is sufficient for most small-scale setups. However, in some complex system implementations, it is also required to have some additional options, such as:

- Arbitrary IP address assignment via IPMI
- IP address assignment via Reverse Address Resolution Protocol (RARP), a feature supported by IPbus.

Moreover, it is also required to be able to switch the IP assignment method dynamically between them without reprogramming. In order to satisfy the above requirements, special GLIB *system_core* firmware and MMC software have been developed.

Figure B 1 shows a diagram of the scheme which is based on the use of the two local I2C buses of the GLIB. Bus #1 (shown in green) was originally foreseen for the communication between the MMC and the three on-board temperature sensors while Bus #2 (shown in red) was intended to be used for the communication between the FPGA and the 24AA025E48 EEPROM for hardware identification reasons, since it contains both a unique 48-bit identifier hardcoded into the chip and the CERN assigned MAC address allocated for the GLIB (08:00:30:f1:00:xx).

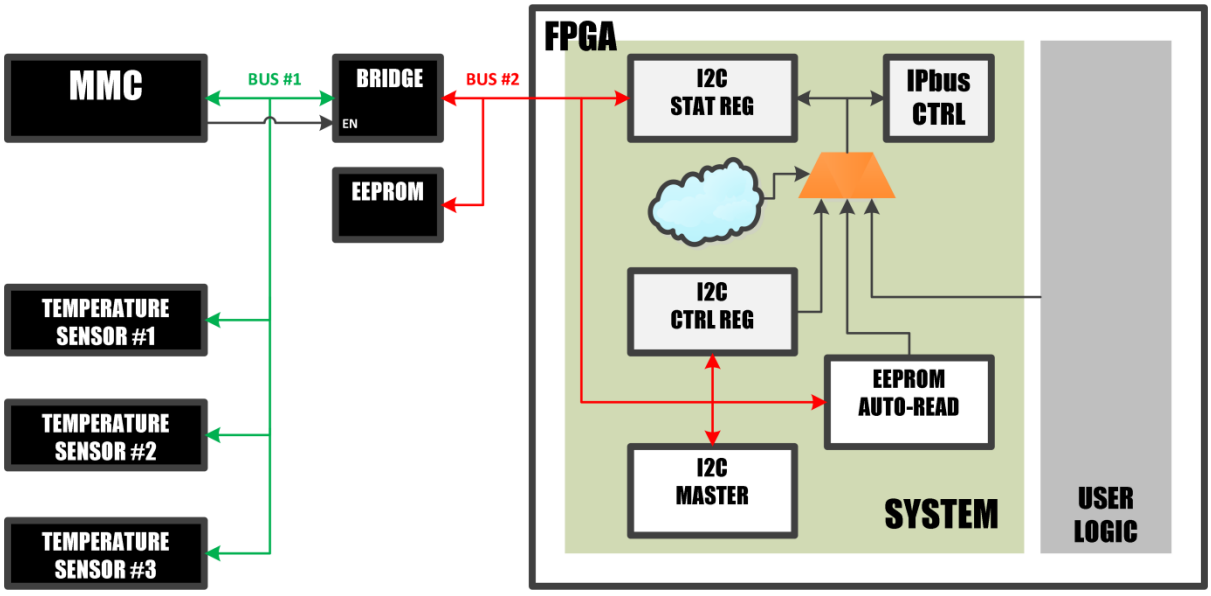

Figure B 1: The IP address assignment scheme

On the FPGA side, apart from the I2C master used for on-demand I2C transactions via IPbus, three additional I2C devices are implemented:

- The EEPROM auto-read: a secondary I2C master that takes over the bus only once after power-up/reset, reads the first 16 bytes of the memory and releases it automatically.
- An I2C slave that implements 16 control registers of 1 byte each.

- An I2C slave that implements 16 status registers of 1 byte each.

Finally, the last I2C component is the I2C bridge located between the two buses that allows the connection or separation of the two I2C buses. The *EN* pin of the bridge is controlled by the MMC.

The IP address assignment scheme functions as following: The MMC, after the handshake with the MCH and before enabling the payload power, ensures that the buses #1 & #2 are disconnected by forcing the *EN* line low, thus avoiding collisions from simultaneous transactions. When the FPGA is loaded, the EEPROM is read automatically and the bus #2 is released. If the contents are valid and the loading *mode* (2-bit setting) is set to "*load MAC & IP*"[2], the IP address is forwarded to the IPbus controller. In all other cases, the IPbus controller is using the IP address provided by the user logic. After the board initialization, a new IP address can be assigned via IPMI in two ways:

- By modifying the IP address stored in the EEPROM and powercycle the GLIB in order to load the new settings.

- By assigning an IP address on-the-fly by programming the I2C control registers (without altering the contents of the EEPROM). The settings in that case are volatile and are retained until the next powercycle.

Setting the IP address to 0.0.0.0 initiates the RARP mechanism that assigns IP addresses automatically.

In it important to mention that there is also the possibility for the user to read via IPMI the I2C status registers that report the current IP address loaded via this circuitry (this does not include any IP address loaded via RARP, since this is happening on a different level). Additionally, the circuitry also allows to identify the GLIB cards via IPMI by reading the unique 48-bit identifier hardcoded in the EEPROM as well as its CERN assigned MAC address.

[2] By default, the EEPROM is set in "*load MAC only*" mode, that serves only for loading the pre-programmed MAC address to the IPbus controller, while the IP address is taken by the *user_logic*