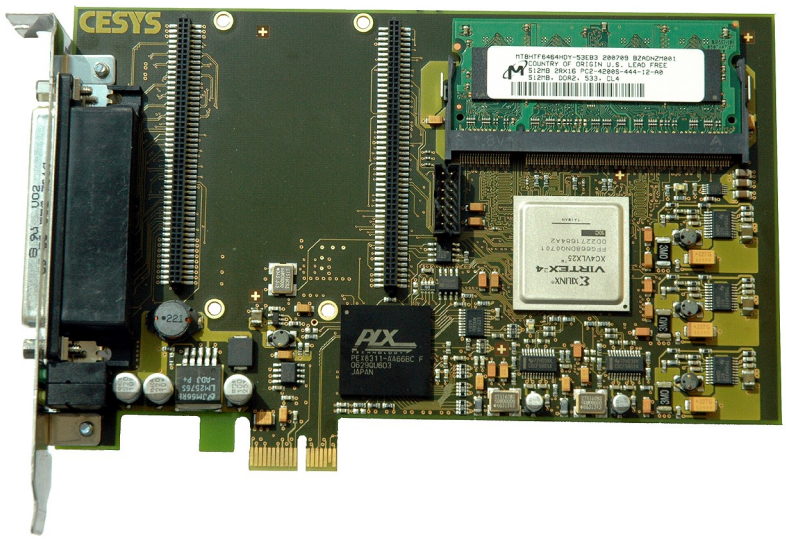
VIRTEX-4™ FPGA board with PCIe interface

Order number: C1080-3807



Copyright information

⇒ Please check www.cesys.com to get the latest version of this document.

CESYS Gesellschaft für angewandte Mikroelektronik mbH

Zeppelinstrasse 6a

D – 91074 Herzogenaurach

Germany

Overview

Summary of PCIeV4Base

The PCIeV4Base board is designed to meet today's demands on development speed and flexibility. It was developed to be used as a OEM component in users systems1, but can also be utilized for learning purposes or prototype development. Its heart is a 24,192 logic cells Virtex-4™ FPGA. 93 FPGA I/O balls and the local-bus clock are routed to the Plug-In-board (PIB) slot, which is wired to a 78-pin HD-SUB I/O connector as well. Plug-In boards can be standard boards from CESYS, a board carrying the functionality defined by you, or your own board. The standard Plug-In board that comes with 0, provides 64 signals with 5 Volt tolerant buffers. Plug-In boards can carry various interfaces like ADC, DAC, TTL Level I/O, RS232, RS485, LVDS, Camera Link or user-defined interface standards. In addition to the Virtex-4™ FPGA, there is a 512 MByte SODIMM DDR2 memory module and a bus-master capable PCI Express bridge-chip on board. Using a PCIe bridge chip instead of implementing the PCIe interface inside the FPGA makes the board much easier to use and allows FPGA-configuration at any time without the need to reboot the PC or re-enumerate the PCIe board. VHDL and C++ sample code that demonstrates how to exchange data between the FPGA and the PC comes with the 0. Users who wish to develop their own PCIe-boards based on the PCIeV4Base can purchase the PCIeV4Base source code package which contains the schematics of the board as well all all sources (API, DLL). Please contact CESYS for details.

Feature list

Form factor	standard short size PCIe board
XILINXTM Virtex-4TM FPGA	XC4VLX25-10FFG668C
PCIe bridge	PEX8311
Memory	512 MB DDR2 SODIMM module
JTAG Interface	connects to XILINXTM Download cable
FPGA Configuration	using PCIe or JTAG
External connector	SUB-D, gender: female, 78-pin
Expansion board slot	standard CESYS PIB Format
Leds	4 green and 4 yellow leds
Clocks	3 on board, additional PIB user clocks possible
Example code	sample VHDL and C++ code can be used as a starting point for user's designs. Included in delivery

The standard delivery, PCIeV4Base, includes:

- PCIeV4Base board
- 512 MByte DDR2 memory module (SODIMM)
- PIB64IO Plug-In-board

- 78-pin SUB-D connector with housing
- One CD-ROM containing the user's manual (English), drivers, libraries, tools and example source code.

All parts are ROHS compliant.

Single boards and very low quantities can be ordered in this configuration only. OEM customers may have a different scope of supply based on individual agreements. If you have questions, please call.

Hardware

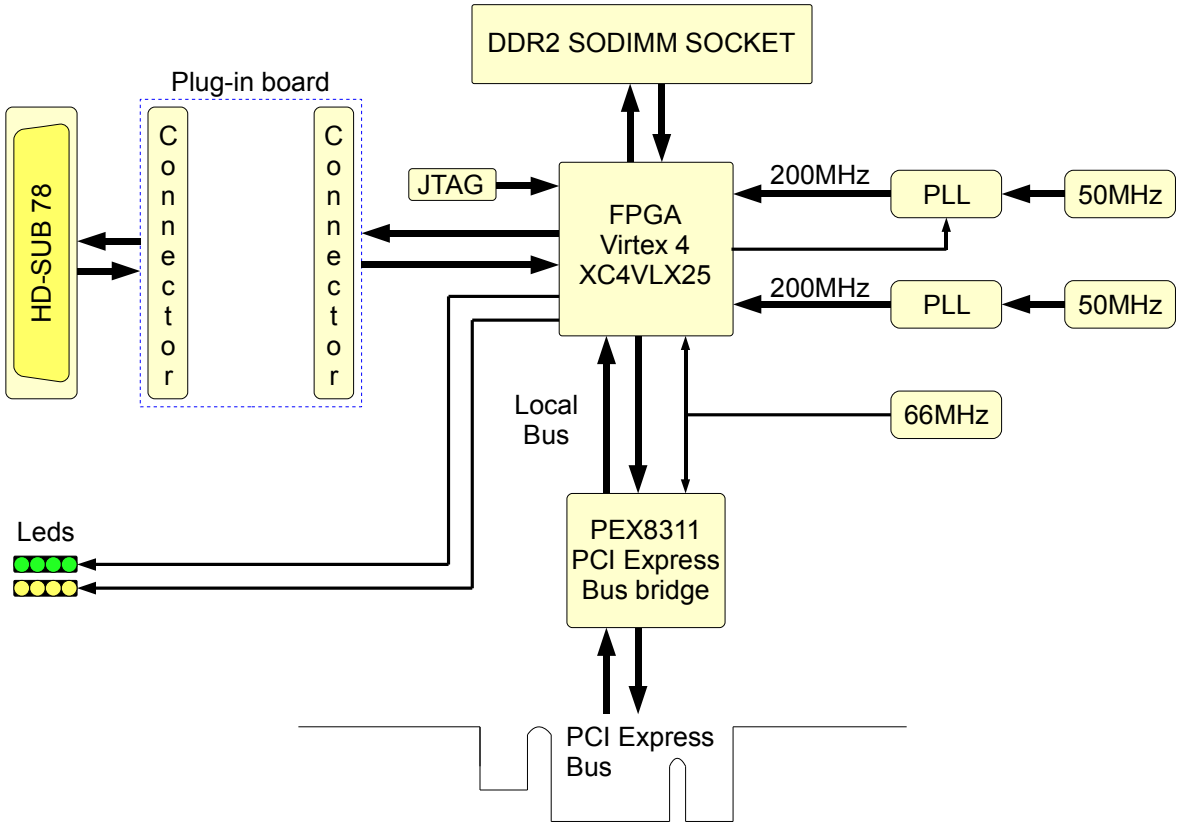


Figure 1: PCIeV4Base block diagram

Virtex-4 FPGA

XC4VLX25-10FFG668C FPGA features:

Configurable Logic Blocks (CLBs):	96 x 26
Logic Cells	24,192
Max Distributed Ram	168 (kBit)
XtremeDSP Slices	48[1]
Block RAM	72 blocks (1,296 kBit)
DCMs	8
PMCDs	4

For details of the Virtex-4™ FPGA device, please look at the data sheet at:
http://direct.xilinx.com/bvdocs/publications/ds112.pdf

1 Each XtremeDSP slice contains one 18 x 18 multiplier, an adder, and an accumulator

SODIMM Memory module

A SODIMM slot populated with a MIG[2] compatible 512 MByte DDR2 memory module is available on board. All examples that come with the board require this reference module. Users may replace the module, but also have to rewrite the FPGA code to support other sizes and parameters. CESYS recommends to keep the default module.

PCI Express interface

To implement a PCI Express interface on a FPGA board, there are two possibilities. The PCI Express interface can be implemented in the FPGA or an additional PCIe bridge chip can be used. The second was preferred for the *PCIeV4BASE* board. Although this makes the costs per board a bit higher, it has some advantages for low to medium volume products like the *PCIeV4BASE:* the local bus of the bridge is easier to handle than a PCIe core, no PCIe core must be purchased and last but not least - the PCIe bridge chip is well tested and will likely run fine on current and future computer systems.

2 Memory Interface Generator

CESYS PIB slot

Like some other CESYS boards, the *PCIeV4BASE* board has a Plug-In-Board slot. The PIB slot consists of two 100-pin connectors. One is wired to some FPGA I/O balls, the other is wired to an external connector. The default PIB that comes with the *PCIeV4BASE* board has 5 Volt tolerant fast buffers and 5 Volt fast drivers for 64 signals. Please check the PIB64IO PIB documentation for the pin out and other details.

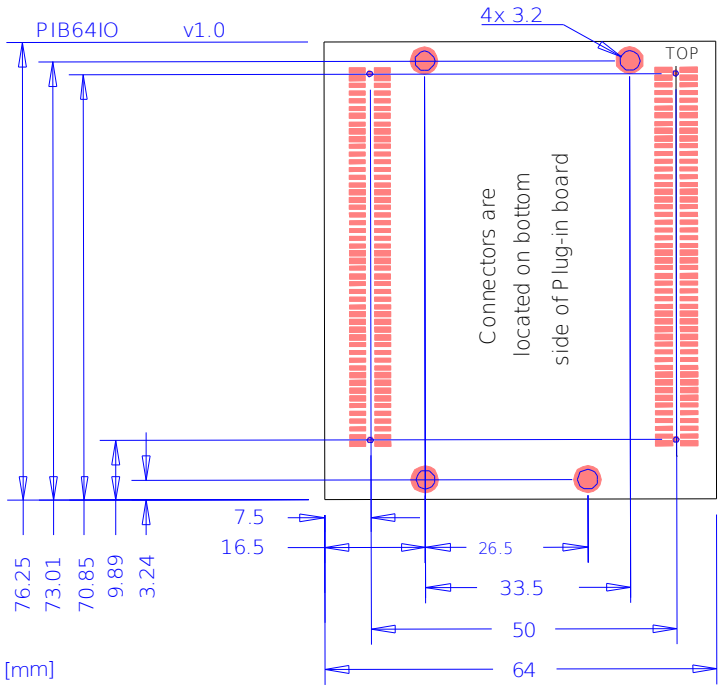


Figure 2: PIB outline drawing and dimensions

Board size

The *PCIeV4BASE* board has Standard PCI-Express half- length card dimensions.

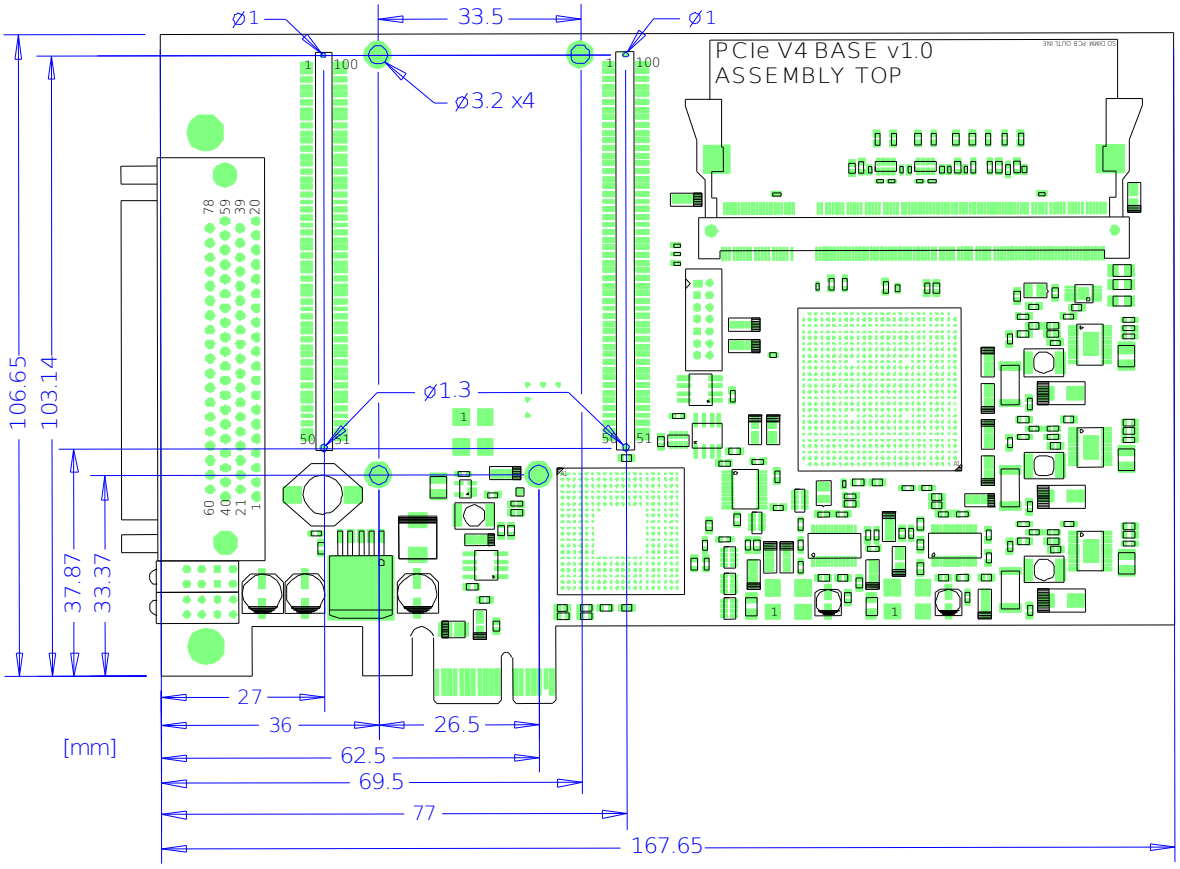


Figure 3: PCIeV4BASE outline drawing and dimensions

Connectors and FPGA pinout

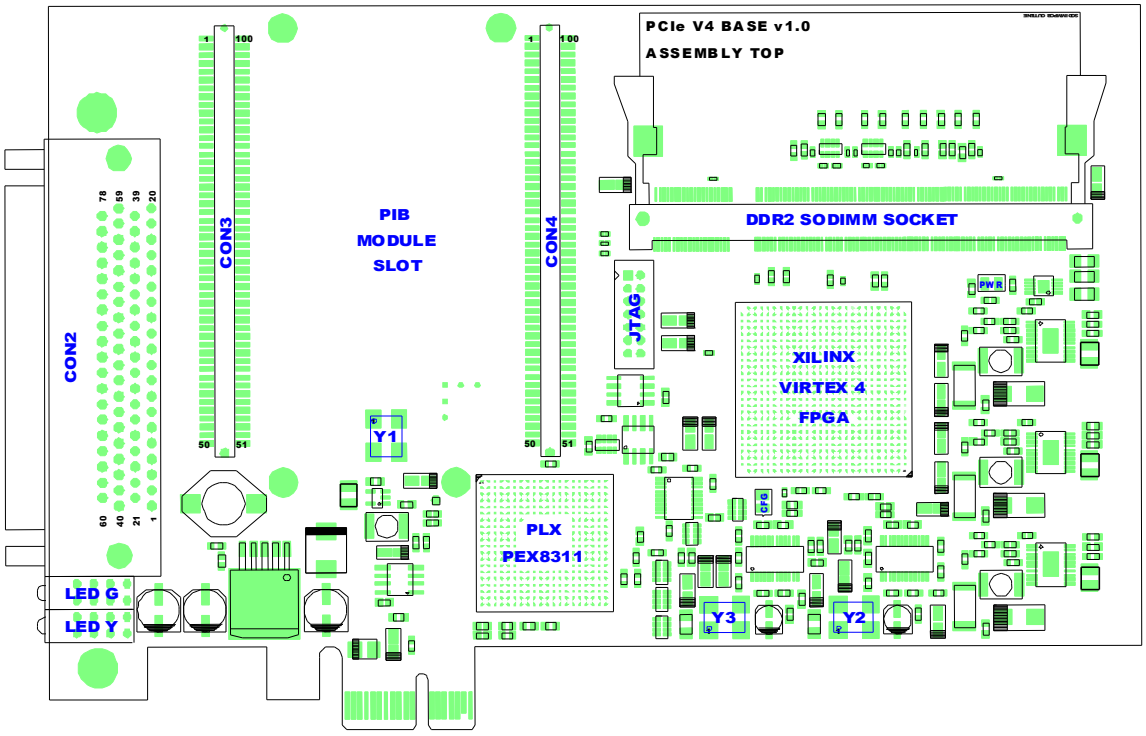


Figure 4: PCIeV4BASE connector diagram

Clock signals

The user can choose between various clock signals available on the *PCIeV4BASE* board. Via the zero delay buffer CY2305 the 66 MHz clock oscillator output signal is distributed to XCV4LX25, PEX8311 and the Plug-In board slot. This clock is used as CCLK for the configuration of the FPGA through the PCIe interface and is usable as system clock for user designs after startup. The same clock also serves as local bus clock for the local bus interface between FPGA and PLX PCIe bus bridge PEX8311. Through an external PLL circuit the 200 MHz differential clock CLK200/CLK200B is provided. This clock can be used as reference clock for the 'idelay' option using the memory interface generator from XIILINX. The differential clock DDR_PLLCLK / DDR_PLLCLKB is generated independently by another PLL circuit. By default DDR_PLLCLK is used as SYS_CLK for the memory interface generator. The default clock rate for DDR_PLLCLK is 200 MHz. Through configuration pins routed to the FPGA the user can optionally set this frequency to other values, for example 50 MHz or 100 MHz. For further information about DDR_PLLCLK configuration see section “DDR PLLCLK”.

Clock signals				
Signal name	**Clock rate**	**I/O Standard**	**FPGA I/O**	**Comment**
CCLK_66MHz	66 MHz	LVCMOS33	G14	Configuration clock for FPGA
PIBCLK_66MHz	66 MHz	LVCMOS33		Clock available at PIB slot clock pin
FPGACLK_66MHz	66 MHz	LVCMOS33	AD12	System clock for FPGA designs
PLX_LCLK_66MHz	66 MHz	LVCMOS33		PLX local bus clock
CLK200	200 MHz	LVPECL25	C15	Reference clock for 'idelay'
CLK200B		LVPECL25	C14	Complementary clock signal
DDR_PLLCLK	200 MHz, adj.	LVPECL25	B15	DDR2 SDRAM clock
DDR_PLLCLKB		LVPECL25	B14	Complementary clock signal

DDR PLLCLK

The *PCIeV4BASE* board provides one configurable PLL clock through the use of the clock multiplier CDCF5801A (http://focus.ti.com/docs/prod/folders/print/cdcf5801a.html) from Texas Instruments. By default DDR_PLLCLK is used as SYS_CLK for the DDR2 IP core generated with the *memory interface generator v1.7* provided by XILINX™. The user should only change this clock with no DDR2 SODIMM memory module inserted or if SYS_CLK is provided by other means.

DDR PLLCLK Interface			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DDR_PLLCLK	LVPECL25	B15	PLL clock output signal
DDR_PLLCLKB	LVPECL25	B14	Complementary clock signal
DDR_PLL_P0	LVCMOS33	C4	Mode control input 0 – Normal operation (default) 1 – High-Z outputs and special settings
DDR_PLL_P1	LVCMOS33	D4	Post divider control input P[1:2] = 11: div2 (default) P[1:2] = 10: div4 P[1:2] = 01: div8
DDR_PLL_P2	LVCMOS33	D1	
DDR_PLL_MULT0	LVCMOS33	D3	PLL multiplication factor select MULT[0:1] = 10: x16 MULT[0:1] = 11: x8 (default) MULT[0:1] = 00: x4

DDR PLLCLK Interface			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DDR_PLL_MULT1	LVCMOS33	D2	MULT[0:1] = 01: x2
FPGA_CLKPLL[3]	LVCMOS33	AE10	Optional PLL clock input driven by FPGA
FPGA_CLKPLL_FB	LVCMOS33	AD17	Optional feedback for FPGA driven PLL clock input

By default the PLL clock input signal is generated via a 50 MHz clock oscillator. Optionally the FPGA can source the PLL clock input. To enable clock sourcing by the FPGA it is necessary to remove resistor R123 and insert R120 and R121. Standard 0603 resistors in the range of 22 – 50 Ohms should be adequate.

SODIMM socket

The *PCIeV4BASE* board provides one standard DDR2 SODIMM module socket for use with 1,8V DDR2 SODIMM modules. Due to the various configurations existing for DDR2 modules it cannot be guaranteed that a specific module is supported. The user has to check the table “Slot1 SODIMM Socket”, whether all I/O needed by the memory module are available on Slot 1.

Slot 1 SODIMM Socket					
Pin	**Signal name**	**FPGA I/O Ball**	**Pin**	**Signal name**	**FPGA I/O Ball**
1	**0,9 Volt**		2	**GND**	
3	**GND**		4	DQ4	W21
5	DQ 0	V21	6	DQ5	W22
7	DQ 1	V22	8	**GND**	
9	**GND**		10	DM0	V20
11	DQS0#	Y24	12	**GND**	
13	DQS0	AA24	14	DQ6	W23
15	**GND**		16	DQ7	W20
17	DQ2	W25	18	**GND**	
19	DQ3	W26	20	DQ12	AB26
21	**GND**		22	DQ13	AA26
23	DQ8	Y25	24	**GND**	

3 To enable DDR_PLLCLK clock sourcing by the FPGA is an expert option and should only be used by experienced users for whom it is normal to do SMD soldering jobs. Warranty will be lost.

Slot 1 SODIMM Socket					
Pin	**Signal name**	**FPGA I/O Ball**	**Pin**	**Signal name**	**FPGA I/O Ball**
25	DQ9	Y26	26	DM1	Y22
27	**GND**		28	**GND**	
29	DQS1#	AC26	30	CK0	AA19
31	DQS1	AC25	32	CK0#	AA20
33	**GND**		34	**GND**	
35	DQ10	AB24	36	DQ14	AD25
37	DQ11	AB25	38	DQ15	AD26
39	**GND**		40	**GND**	
41	**GND**		42	**GND**	
43	DQ16	AC22	44	DQ20	AD22
45	DQ17	AB22	46	DQ21	AD23
47	**GND**		48	**GND**	
49	DQS2#	AB21	50	NC	
51	DQS2	AC21	52	DM2	AF19
53	**GND**		54	**GND**	
55	DQ18	AB23	56	DQ22	AC23
57	DQ19	AA23	58	DQ23	AC24
59	**GND**		60	**GND**	
61	DQ24	AF20	62	DQ28	AE23
63	DQ25	Y19	64	DQ29	Y20
65	**GND**		66	**GND**	
67	DM3	AF24	68	DQS3#	AC19
69	NC		70	DQS3	AD19
71	**GND**		72	**GND**	
73	DQ26	W19	74	DQ30	AA18
75	DQ27	AF23	76	DQ31	Y18
77	**GND**		78	**GND**	
79	CKE0	K2	80	CKE1	L4
81	**1,8 Volt**		82	**1,8 Volt**	
83	NC		84	NC/A15	M1
85	NC/BA2	AF9	86	NC/A14	K1
87	**1,8 Volt**		88	**1,8 Volt**	
89	A12	AE24	90	A11	AB20
91	A9	AB18	92	A7	AF22

Slot 1 SODIMM Socket					
Pin	**Signal name**	**FPGA I/O Ball**	**Pin**	**Signal name**	**FPGA I/O Ball**
93	A8	AF21	94	A6	AF18
95	**1,8 Volt**		96	**1,8 Volt**	
97	A5	AE18	98	A4	AE21
99	A3	AD21	100	A2	J7
101	A1	J6	102	A0	J5
103	**1,8 Volt**		104	**1,8 Volt**	
105	A10	AC18	106	BA1	J4
107	BA0	K7	108	RAS#	K6
109	WE#	L7	110	CS0#	L6
111	**1,8 Volt**		112	**1,8 Volt**	
113	CAS#	J2	114	ODT0	K4
115	CS1#	K5	116	NC/A13	L1
117	**1,8 Volt**		118	**1,8 Volt**	
119	ODT1	K3	120	NC	
121	**GND**		122	**GND**	
123	DQ32	W2	124	DQ36	W7
125	DQ33	W1	126	DQ37	V7
127	**GND**		128	**GND**	
129	DQS4#	Y3	130	DM4	W5
131	DQS4	Y4	132	**GND**	
133	**GND**		134	DQ38	W4
135	DQ34	V6	136	DQ39	W6
137	DQ35	V5	138	**GND**	
139	**GND**		140	DQ44	AB1
141	DQ40	Y2	142	DQ45	AA1
143	DQ41	Y1	144	**GND**	
145	**GND**		146	DQS5#	Y5
147	DM5	AB3	148	DQS5	Y6
149	**GND**		150	**GND**	
151	DQ42	AA4	152	DQ46	AC4
153	DQ43	AA3	154	DQ47	AB4
155	**GND**		156	**GND**	
157	DQ48	AC5	158	DQ52	AF3
159	DQ49	AB5	160	DQ53	AE3

Slot 1 SODIMM Socket					
Pin	**Signal name**	**FPGA I/O Ball**	**Pin**	**Signal name**	**FPGA I/O Ball**
161	**GND**		162	**GND**	
163	NC		164	CK1	Y17
165	**GND**		166	CK1#	AA17
167	DQS6#	AB6	168	**GND**	
169	DQS6	AC6	170	DM6	AF4
171	**GND**		172	**GND**	
173	DQ50	AC2	174	DQ54	AD2
175	DQ51	AC1	176	DQ55	AD1
177	**GND**		178	**GND**	
179	DQ56	AE4	180	DQ60	AF5
181	DQ57	AD3	182	DQ61	AA7
183	**GND**		184	**GND**	
185	DM7	AD5	186	DQS7#	AF7
187	**GND**		188	DQS7	AF8
189	DQ58	AC3	190	**GND**	
191	DQ59	AF6	192	DQ62	AA9
193	**GND**		194	DQ63	Y9
195	SDA	AF10	196	**GND**	
197	SCL	AF11	198	SA0	AE14
199	**3,3 Volt**		200	SA1	AE13

By default the DDR2 SODIMM module MT8HTF6464HDY -512MB from Micron is used. The specific memory interface for the Micron MT8HTF6464HDY – 512MB is described in table “DDR2 – Memory Interface for Micron MT8HTF6464HDY – 512MB” for reference. Further information on using this DDR2 SODIMM can be found in chapter D, section “Interfacing DDR2 memory”.

DDR2 – Memory Interface for Micron MT8HTF6464HDY – 512MB			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DQ 0	SSTL18_II_DCI	V21	Data I/O: Bidirectional data bus
DQ 1	SSTL18_II_DCI	V22	Data I/O: Bidirectional data bus
DQ 2	SSTL18_II_DCI	W25	Data I/O: Bidirectional data bus
DQ 3	SSTL18_II_DCI	W26	Data I/O: Bidirectional data bus
DQ 4	SSTL18_II_DCI	W21	Data I/O: Bidirectional data bus

DDR2 – Memory Interface for Micron MT8HTF6464HDY – 512MB			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DQ 5	SSTL18_II_DCI	W22	Data I/O: Bidirectional data bus
DQ 6	SSTL18_II_DCI	W23	Data I/O: Bidirectional data bus
DQ 7	SSTL18_II_DCI	W20	Data I/O: Bidirectional data bus
DQ 8	SSTL18_II_DCI	Y25	Data I/O: Bidirectional data bus
DQ 9	SSTL18_II_DCI	Y26	Data I/O: Bidirectional data bus
DQ 10	SSTL18_II_DCI	AB24	Data I/O: Bidirectional data bus
DQ 11	SSTL18_II_DCI	AB25	Data I/O: Bidirectional data bus
DQ 12	SSTL18_II_DCI	AB26	Data I/O: Bidirectional data bus
DQ 13	SSTL18_II_DCI	AA26	Data I/O: Bidirectional data bus
DQ 14	SSTL18_II_DCI	AD25	Data I/O: Bidirectional data bus
DQ 15	SSTL18_II_DCI	AD26	Data I/O: Bidirectional data bus
DQ 16	SSTL18_II_DCI	AC22	Data I/O: Bidirectional data bus
DQ 17	SSTL18_II_DCI	AB22	Data I/O: Bidirectional data bus
DQ 18	SSTL18_II_DCI	AB23	Data I/O: Bidirectional data bus
DQ 19	SSTL18_II_DCI	AA23	Data I/O: Bidirectional data bus
DQ 20	SSTL18_II_DCI	AD22	Data I/O: Bidirectional data bus
DQ 21	SSTL18_II_DCI	AD23	Data I/O: Bidirectional data bus
DQ 22	SSTL18_II_DCI	AC23	Data I/O: Bidirectional data bus
DQ 23	SSTL18_II_DCI	AC24	Data I/O: Bidirectional data bus
DQ 24	SSTL18_II_DCI	AF20	Data I/O: Bidirectional data bus
DQ 25	SSTL18_II_DCI	Y19	Data I/O: Bidirectional data bus
DQ 26	SSTL18_II_DCI	W19	Data I/O: Bidirectional data bus
DQ 27	SSTL18_II_DCI	AF23	Data I/O: Bidirectional data bus
DQ 28	SSTL18_II_DCI	AE23	Data I/O: Bidirectional data bus
DQ 29	SSTL18_II_DCI	Y20	Data I/O: Bidirectional data bus
DQ 30	SSTL18_II_DCI	AA18	Data I/O: Bidirectional data bus
DQ 31	SSTL18_II_DCI	Y18	Data I/O: Bidirectional data bus
DQ 32	SSTL18_II_DCI	W2	Data I/O: Bidirectional data bus
DQ 33	SSTL18_II_DCI	W1	Data I/O: Bidirectional data bus
DQ 34	SSTL18_II_DCI	V6	Data I/O: Bidirectional data bus
DQ 35	SSTL18_II_DCI	V5	Data I/O: Bidirectional data bus
DQ 36	SSTL18_II_DCI	W7	Data I/O: Bidirectional data bus
DQ 37	SSTL18_II_DCI	V7	Data I/O: Bidirectional data bus
DQ 38	SSTL18_II_DCI	W4	Data I/O: Bidirectional data bus

DDR2 – Memory Interface for Micron MT8HTF6464HDY – 512MB			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
DQ 39	SSTL18_II_DCI	W6	Data I/O: Bidirectional data bus
DQ 40	SSTL18_II_DCI	Y2	Data I/O: Bidirectional data bus
DQ 41	SSTL18_II_DCI	Y1	Data I/O: Bidirectional data bus
DQ 42	SSTL18_II_DCI	AA4	Data I/O: Bidirectional data bus
DQ 43	SSTL18_II_DCI	AA3	Data I/O: Bidirectional data bus
DQ 44	SSTL18_II_DCI	AB1	Data I/O: Bidirectional data bus
DQ 45	SSTL18_II_DCI	AA1	Data I/O: Bidirectional data bus
DQ 46	SSTL18_II_DCI	AC4	Data I/O: Bidirectional data bus
DQ 47	SSTL18_II_DCI	AB4	Data I/O: Bidirectional data bus
DQ 48	SSTL18_II_DCI	AC5	Data I/O: Bidirectional data bus
DQ 49	SSTL18_II_DCI	AB5	Data I/O: Bidirectional data bus
DQ 50	SSTL18_II_DCI	AC2	Data I/O: Bidirectional data bus
DQ 51	SSTL18_II_DCI	AC1	Data I/O: Bidirectional data bus
DQ 52	SSTL18_II_DCI	AF3	Data I/O: Bidirectional data bus
DQ 53	SSTL18_II_DCI	AE3	Data I/O: Bidirectional data bus
DQ 54	SSTL18_II_DCI	AD2	Data I/O: Bidirectional data bus
DQ 55	SSTL18_II_DCI	AD1	Data I/O: Bidirectional data bus
DQ 56	SSTL18_II_DCI	AE4	Data I/O: Bidirectional data bus
DQ 57	SSTL18_II_DCI	AD3	Data I/O: Bidirectional data bus
DQ 58	SSTL18_II_DCI	AC3	Data I/O: Bidirectional data bus
DQ 59	SSTL18_II_DCI	AF6	Data I/O: Bidirectional data bus
DQ 60	SSTL18_II_DCI	AF5	Data I/O: Bidirectional data bus
DQ 61	SSTL18_II_DCI	AA7	Data I/O: Bidirectional data bus
DQ 62	SSTL18_II_DCI	AA9	Data I/O: Bidirectional data bus
DQ 63	SSTL18_II_DCI	Y9	Data I/O: Bidirectional data bus
A 0	SSTL18_I_DCI	J5	Address input 0
A 1	SSTL18_I_DCI	J6	Address input 1
A 2	SSTL18_I_DCI	J7	Address input 2
A 3	SSTL18_I_DCI	AD21	Address input 3
A 4	SSTL18_I_DCI	AE21	Address input 4
A 5	SSTL18_I_DCI	AE18	Address input 5
A 6	SSTL18_I_DCI	AF18	Address input 6
A 7	SSTL18_I_DCI	AF22	Address input 7
A 8	SSTL18_I_DCI	AF21	Address input 8

DDR2 – Memory Interface for Micron MT8HTF6464HDY – 512MB			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
A 9	SSTL18_I_DCI	AB18	Address input 9
A 10	SSTL18_I_DCI	AC18	Address input 10
A 11	SSTL18_I_DCI	AB20	Address input 11
A 12	SSTL18_I_DCI	AE24	Address input 12
BA 0	SSTL18_I_DCI	K7	Bank address input 0
BA 1	SSTL18_I_DCI	J4	Bank address input 1
RAS#	SSTL18_I_DCI	K6	Row address strobe
CAS#	SSTL18_I_DCI	J2	Column address strobe
WE#	SSTL18_I_DCI	L7	Write enable
CS# 0	SSTL18_I_DCI	L6	Chip select 0
CS# 1	SSTL18_I_DCI	K5	Chip select 1
ODT 0	SSTL18_I_DCI	K4	On-die termination 0
ODT 1	SSTL18_I_DCI	K3	On-die termination 1
CKE 0	SSTL18_I_DCI	K2	Clock enable 0
CKE 1	SSTL18_I_DCI	L4	Clock enable 1
DM 0	SSTL18_II_DCI	V20	Input data mask 0
DM 1	SSTL18_II_DCI	Y22	Input data mask 1
DM 2	SSTL18_II_DCI	AF19	Input data mask 2
DM 3	SSTL18_II_DCI	AF24	Input data mask 3
DM 4	SSTL18_II_DCI	W5	Input data mask 4
DM 5	SSTL18_II_DCI	AB3	Input data mask 5
DM 6	SSTL18_II_DCI	AF4	Input data mask 6
DM 7	SSTL18_II_DCI	AD5	Input data mask 7
DQS 0	DIFF_SSTL18_II_DCI	AA24	Data strobe 0
DQS# 0	DIFF_SSTL18_II_DCI	Y24	Complementary Data strobe 0
DQS 1	DIFF_SSTL18_II_DCI	AC25	Data strobe 1
DQS# 1	DIFF_SSTL18_II_DCI	AC26	Complementary Data strobe 1
DQS 2	DIFF_SSTL18_II_DCI	AC21	Data strobe 2
DQS# 2	DIFF_SSTL18_II_DCI	AB21	Complementary Data strobe 2
DQS 3	DIFF_SSTL18_II_D	AD19	Data strobe 3

DDR2 – Memory Interface for Micron MT8HTF6464HDY – 512MB			
Signal name	**I/O Standard**	**FPGA I/O**	**Comment**
	CI		
DQS# 3	DIFF_SSTL18_II_DCI	AC19	Complementary Data strobe 3
DQS 4	DIFF_SSTL18_II_DCI	Y4	Data strobe 4
DQS# 4	DIFF_SSTL18_II_DCI	Y3	Complementary Data strobe 4
DQS 5	DIFF_SSTL18_II_DCI	Y6	Data strobe 5
DQS# 5	DIFF_SSTL18_II_DCI	Y5	Complementary Data strobe 5
DQS 6	DIFF_SSTL18_II_DCI	AC6	Data strobe 6
DQS# 6	DIFF_SSTL18_II_DCI	AB6	Complementary Data strobe 6
DQS 7	DIFF_SSTL18_II_DCI	AF8	Data strobe 7
DQS# 7	DIFF_SSTL18_II_DCI	AF7	Complementary Data strobe 7
CK 0	DIFF_SSTL18_II_DCI	AA19	Clock 0
CK# 0	DIFF_SSTL18_II_DCI	AA20	Complementary Clock 0
CK 1	DIFF_SSTL18_II_DCI	Y17	Clock 1
CK# 1	DIFF_SSTL18_II_DCI	AA17	Complementary Clock 1
SCL	LVCMOS33	AF11	Serial clock for presence-detect
SDA	LVCMOS33	AF10	Serial presence-detect data
SA 0	LVCMOS33	AE14	Presence-detect address input 0
SA 1	LVCMOS33	AE13	Presence-detect address input 1

PIB signals and SUB-D connector

CON4 and CON3 serve as the Plug-In board socket. Through CON4 93 FPGA I/O are accessible. The I/O standard to be used is LVCMOS33. Pins marked as CC are connected to lower capacitance clock capable IO which lack the LVDS driver to reduce parasitic capacitance and therefore are the perfect choice for very high clock rates. For further information about CC and LC IO the user is encouraged to check the appropriate user guides provided by XILINX™. On Pin 44 the 66 MHz local bus clock is available. The complete pinout is provided in table “CON4 Plug-In board to FPGA I/O-pin connector”.

CON 4 Plug-In board to FPGA I/O-pin connector					
Pin	**Signal name**	**FPGA I/O Ball**	**Pin**	**Signal name**	**FPGA I/O Ball**
1	PIB_IO0	AB17	**100**	PIB_IO92	V25
2	PIB_IO1	U21	**99**	PIB_IO91	V26
3	PIB_IO2	U22	**98**	PIB_IO90	U26
4	**GND**	**--**	**97**	PIB_IO89	T26
5	PIB_IO3	T19	**96**	PIB_IO88	U24
6	PIB_IO4	U20	**95**	PIB_IO87	U25
7	PIB_IO5	R23	**94**	PIB_IO86	V23
8	PIB_IO6	R24	**93**	PIB_IO85	U23
9	PIB_IO7	R21	**92**	PIB_IO84	T20
10	PIB_IO8	R22	**91**	PIB_IO83	T21
11	PIB_IO9	T23 (LC[4])	**90**	PIB_IO82	R25
12	PIB_IO10	T24 (CC[5])	**89**	PIB_IO81	R26
13	PIB_IO11	R20 (CC)	**88**	PIB_IO80	P24
14	PIB_IO12	R19 (LC)	**87**	PIB_IO79	P25
15	PIB_IO13	P19	**86**	PIB_IO78	P22
16	PIB_IO14	P20	**85**	PIB_IO77	P23
17	PIB_IO15	N19 (LC)	**84**	PIB_IO76	N25
18	PIB_IO16	M19 (CC)	**83**	PIB_IO75	N24
19	PIB_IO17	K26 (CC)	**82**	PIB_IO74	N23
20	PIB_IO18	K25 (LC)	**81**	PIB_IO73	N22
21	PIB_IO19	M23	**80**	PIB_IO72	N21
22	PIB_IO20	M22	**79**	PIB_IO71	N20
23	**GND**	**--**	**78**	PIB_IO70	M26

4 Lower capacitance pin
5 Lower capacitance clock pin

CON 4 Plug-In board to FPGA I/O-pin connector					
Pin	**Signal name**	**FPGA I/O Ball**	**Pin**	**Signal name**	**FPGA I/O Ball**
24	PIB_IO21	M21	**77**	PIB_IO69	L26
25	PIB_IO22	M20	**76**	PIB_IO68	M25
26	PIB_IO23	L21	**75**	PIB_IO67	M24
27	PIB_IO24	L20	**74**	PIB_IO66	L24
28	PIB_IO25	L19	**73**	PIB_IO65	L23
29	PIB_IO26	K20	**72**	PIB_IO64	K24
30	PIB_IO27	K22	**71**	PIB_IO63	K23
31	PIB_IO28	K21	**70**	PIB_IO62	J26
32	PIB_IO29	J21	**69**	PIB_IO61	J25
33	PIB_IO30	J20	**68**	PIB_IO60	J23
34	PIB_IO31	F24 (CC)	**67**	PIB_IO59	J22
35	PIB_IO32	F23 (LC)	**66**	PIB_IO58	H26
36	PIB_IO33	E23	**65**	PIB_IO57	H25
37	PIB_IO34	E22	**64**	PIB_IO56	G26
38	PIB_IO35	D24	**63**	PIB_IO55	G25
39	PIB_IO36	C24	**62**	PIB_IO54	H24
40	PIB_IO37	D23	**61**	PIB_IO53	H23
41	PIB_IO38	C23	**60**	PIB_IO52	G24
42	PIB_IO39	A21	**59**	PIB_IO51	G23
43	PIB_IO40	A22	**58**	PIB_IO50	F26
44	PIBCLK (66MHz)	--	**57**	PIB_IO49	E26
45	**GND**	**--**	**56**	PIB_IO48	E25
46	PIB_IO41	A23 (LC)	**55**	PIB_IO47	E24
47	PIB_IO42	A24 (CC)	**54**	PIB_IO46	D26 (CC)
48	**+3,3 Volt**	**--**	**53**	PIB_IO45	D25 (LC)
49	**+3,3 Volt**	**--**	**52**	PIB_IO44	C26
50	**+3,3 Volt**	**--**	**51**	PIB_IO43	C25

CON3 connects to the 78-pin HD-SUBD connector CON2. 32 IO are routed as 16 differential pairs with a typical differential impedance of 100 Ohms +/-10%. The pinout with the existing differential pairs marked is available in table “CON 3 Plug-In board to External HD-SUB connector”.

To power active devices on Plug-In boards supply voltages +3,3 Volt, +5 Volt and +12 Volt are provided on CON3 and CON4. +3,3 Volt and +12 Volt are fed from the PCIe-

Connector and can supply current up to the limits of the PCIe specification. The +5 Volt supply is provided through a LM2576S step-down voltage regulator from the +12 Volt supply rail. Current supplied by +5 Volt rail should not exceed 1A.

CON 3 Plug-In board to External 78-pin HD-SUB connector CON 2					
Pin	**HD-SUB**	**Differential**	**Pin**	**HD-SUB**	**Differential**
1	**GND**	**--**	**100**	**GND**	**--**
2	**GND**	**--**	**99**	**GND**	**--**
3	**GND**	**--**	**98**	**GND**	**--**
4	HD-Sub Pin 39	Diff.- Pair 0	**97**	HD-Sub Pin 59	Diff.- Pair 8
5	HD-Sub Pin 20	Diff.- Pair 0	**96**	HD-Sub Pin 78	Diff.- Pair 8
6	HD-Sub Pin 38	Diff.- Pair 1	**95**	HD-Sub Pin 58	Diff.- Pair 9
7	HD-Sub Pin 19	Diff.- Pair 1	**94**	HD-Sub Pin 77	Diff.- Pair 9
8	HD-Sub Pin 37	Diff.- Pair 2	**93**	HD-Sub Pin 57	Diff.- Pair 10
9	HD-Sub Pin 18	Diff.- Pair 2	**92**	HD-Sub Pin 76	Diff.- Pair 10
10	HD-Sub Pin 36	Diff.- Pair 3	**91**	HD-Sub Pin 56	Diff.- Pair 11
11	HD-Sub Pin 17	Diff.- Pair 3	**90**	HD-Sub Pin 75	Diff.- Pair 11
12	HD-Sub Pin 35	Diff.- Pair 4	**89**	HD-Sub Pin 55	Diff.- Pair 12
13	HD-Sub Pin 16	Diff.- Pair 4	**88**	HD-Sub Pin 74	Diff.- Pair 12
14	HD-Sub Pin 34	Diff.- Pair 5	**87**	HD-Sub Pin 54	Diff.- Pair 13
15	HD-Sub Pin 15	Diff.- Pair 5	**86**	HD-Sub Pin 73	Diff.- Pair 13
16	HD-Sub Pin 33	Diff.- Pair 6	**85**	HD-Sub Pin 53	Diff.- Pair 14
17	HD-Sub Pin 14	Diff.- Pair 6	**84**	HD-Sub Pin 72	Diff.- Pair 14
18	HD-Sub Pin 32	Diff.- Pair 7	**83**	HD-Sub Pin 52	Diff.- Pair 15
19	HD-Sub Pin 13	Diff.- Pair 7	**82**	HD-Sub Pin 71	Diff.- Pair 15
20	HD-Sub Pin 12	--	**81**	HD-Sub Pin 51	--
21	HD-Sub Pin 31	--	**80**	HD-Sub Pin 70	--
22	HD-Sub Pin 11	--	**79**	HD-Sub Pin 50	--
23	HD-Sub Pin 30	--	**78**	HD-Sub Pin 69	--
24	HD-Sub Pin 10	--	**77**	HD-Sub Pin 68	--
25	HD-Sub Pin 9	--	**76**	HD-Sub Pin 67	--
26	HD-Sub Pin 8	--	**75**	HD-Sub Pin 66	--
27	HD-Sub Pin 7	--	**74**	HD-Sub Pin 65	--
28	HD-Sub Pin 6	--	**73**	HD-Sub Pin 64	--
29	HD-Sub Pin 5	--	**72**	HD-Sub Pin 63	--
30	HD-Sub Pin 4	--	**71**	HD-Sub Pin 62	--
31	HD-Sub Pin 3	--	**70**	HD-Sub Pin 61	--

CON 3 Plug-In board to External 78-pin HD-SUB connector CON 2					
Pin	**HD-SUB**	**Differential**	**Pin**	**HD-SUB**	**Differential**
32	HD-Sub Pin 2	--	**69**	HD-Sub Pin 60	--
33	HD-Sub Pin 29	--	**68**	HD-Sub Pin 49	--
34	HD-Sub Pin 28	--	**67**	HD-Sub Pin 48	--
35	HD-Sub Pin 27	--	**66**	HD-Sub Pin 47	--
36	HD-Sub Pin 26	--	**65**	HD-Sub Pin 46	--
37	HD-Sub Pin 25	--	**64**	HD-Sub Pin 45	--
38	HD-Sub Pin 24	--	**63**	HD-Sub Pin 44	--
39	HD-Sub Pin 23	--	**62**	HD-Sub Pin 43	--
40	HD-Sub Pin 22	--	**61**	HD-Sub Pin 42	--
41	HD-Sub Pin 21	--	**60**	HD-Sub Pin 41	--
42	**GND**	**--**	**59**	**GND**	**--**
43	**GND**	**--**	**58**	**GND**	**--**
44	**GND**	**--**	**57**	**GND**	**--**
45	**+5 Volt**	**--**	**56**	**+5 Volt**	**--**
46	**+5 Volt**	**--**	**55**	**+5 Volt**	**--**
47	**+5 Volt**	**--**	**54**	**+5 Volt**	**--**
48	**+12 Volt**	**--**	**53**	**+12 Volt**	**--**
49	**+12 Volt**	**--**	**52**	**+12 Volt**	**--**
50	**+12 Volt**	**--**	**51**	**+12 Volt**	**--**

CON 2 External 78-pin HD-SUB to Plug-In board connector CON 3							
HD-SUB	**PIB**	**HD-SUB**	**PIB**	**HD-SUB**	**PIB**	**HD-SUB**	**PIB**
HD-Pin 1	**GND**	HD-Pin 21	**41**	HD-Pin 40	**EARTH**	HD-Pin 60	**69**
HD-Pin 2	**32**	HD-Pin 22	**40**	HD-Pin 41	**60**	HD-Pin 61	**70**
HD-Pin 3	**31**	HD-Pin 23	**39**	HD-Pin 42	**61**	HD-Pin 62	**71**
HD-Pin 4	**30**	HD-Pin 24	**38**	HD-Pin 43	**62**	HD-Pin 63	**72**
HD-Pin 5	**29**	HD-Pin 25	**37**	HD-Pin 44	**63**	HD-Pin 64	**73**
HD-Pin 6	**28**	HD-Pin 26	**36**	HD-Pin 45	**64**	HD-Pin 65	**74**
HD-Pin 7	**27**	HD-Pin 27	**35**	HD-Pin 46	**65**	HD-Pin 66	**75**
HD-Pin 8	**26**	HD-Pin 28	**34**	HD-Pin 47	**66**	HD-Pin 67	**76**
HD-Pin 9	**25**	HD-Pin 29	**33**	HD-Pin 48	**67**	HD-Pin 68	**77**
HD-Pin 10	**24**	HD-Pin 30	**23**	HD-Pin 49	**68**	HD-Pin 69	**78**
HD-Pin 11	**22**	HD-Pin 31	**21**	HD-Pin 50	**79**	HD-Pin 70	**80**

CON 2 External 78-pin HD-SUB to Plug-In board connector CON 3							
HD-SUB	**PIB**	**HD-SUB**	**PIB**	**HD-SUB**	**PIB**	**HD-SUB**	**PIB**
HD-Pin 12	**20**	HD-Pin 32	**18**	HD-Pin 51	**81**	HD-Pin 71	**82**
HD-Pin 13	**19**	HD-Pin 33	**16**	HD-Pin 52	**83**	HD-Pin 72	**84**
HD-Pin 14	**17**	HD-Pin 34	**14**	HD-Pin 53	**85**	HD-Pin 73	**86**
HD-Pin 15	**15**	HD-Pin 35	**12**	HD-Pin 54	**87**	HD-Pin 74	**88**
HD-Pin 16	**13**	HD-Pin 36	**10**	HD-Pin 55	**89**	HD-Pin 75	**90**
HD-Pin 17	**11**	HD-Pin 37	**8**	HD-Pin 56	**91**	HD-Pin 76	**92**
HD-Pin 18	**9**	HD-Pin 38	**6**	HD-Pin 57	**93**	HD-Pin 77	**94**
HD-Pin 19	**7**	HD-Pin 39	**4**	HD-Pin 58	**95**	HD-Pin 78	**96**
HD-Pin 20	**5**	--	**--**	HD-Pin 59	**97**	--	--

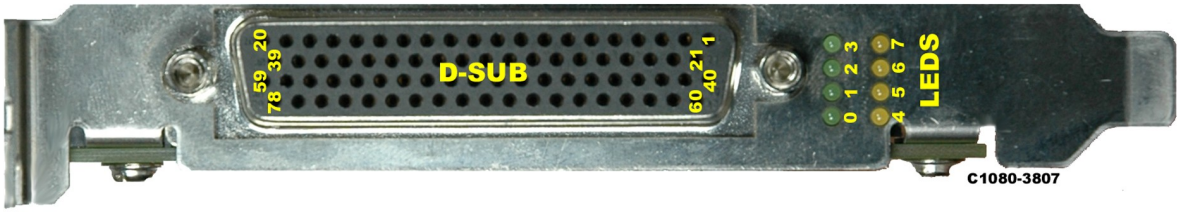


Figure 5: PCIeV4BASE PCIe slot bracket

Leds

The *PCIeV4BASE* is equipped with several LEDs. The PWR LED gives information upon the power supply state. The varying supply voltages needed by the Xilinx™ FPGA are ramped up sequential. PWR LED starts to light after all voltages have reached an appropriate level. Upon successful configuration the CFG LED lights up and stays on as long as the device is configured. Additionally 8 user- configurable LEDs allow to make internal monitoring states visible by driving the appropriate FPGA I/O high.

LEDs	
LED	**Comment**
LED0 Green	FPGA I/O Ball D22
LED1 Green	FPGA I/O Ball C22
LED2 Green	FPGA I/O Ball E21
LED3 Green	FPGA I/O Ball D21
LED4 Yellow	FPGA I/O Ball C21
LED5 Yellow	FPGA I/O Ball B24
LED6 Yellow	FPGA I/O Ball C20
LED7 Yellow	FPGA I/O Ball B23

LEDs	
LED	**Comment**
CFG LED	Configuration LED
PWR LED	POWER

PCIe Controller local bus signals

This section describes in short the interface between Virtex™4 FPGA and PLX PEX8311. PEX8311 supports three types of local bus processor interface. For *PCIeV4BASE* only J mode with multiplexed address/data bus is available. From the three existing data transfer modes of PEX8311 Direct Slave mode and DMA mode are implemented. For data transmission 32 Bit single read/write and DMA single and continuous burst cycles are supported. Further information about the usage of the local bus interface can be found in chapter D section 'design “pciev4base” '. It may also be useful to have a look at the documentation for the x1 Lane PCIe Bridge PEX8311 at PLX (http://www.plxtech.com/products/expresslane/pex8311.asp).

The following spreadsheet “Local bus signals” gives an overview of the local bus signals and which FPGA I/O they are connected to.

Local bus signals				
FPGA I/O	**I/O Standard**	**Signal name**	**External pull-up /down**	**Comment**
E18	LVCMOS33	ADS#	pull-up	Address strobe
A7	LVCMOS33	ALE	pull-down	Address latch enable
C19	LVCMOS33	BIGEND#	pull-up	Big- endian select
D18	LVCMOS33	BLAST#	pull-up	Burst last
E17	LVCMOS33	BREQi	pull-down	Bus request in
F17	LVCMOS33	BREQo	pull-down	Bus request out
D10	LVCMOS33	BTERM#	pull-up	Burst terminate
D19	LVCMOS33	CCS#	pull-up	Configuration register select
A20	LVCMOS33	DACK0#	pull-up	DMA channel 0 demand mode acknowledge
B20	LVCMOS33	DACK1#	pull-up	DMA channel 1 demand mode acknowledge
A6	LVCMOS33	DEN#	pull-up	Data enable
F10	LVCMOS33	DP0	--	Data parity 0
A9	LVCMOS33	DP1	--	Data parity 1
B9	LVCMOS33	DP2	--	Data parity 2

Local bus signals				
FPGA I/O	**I/O Standard**	**Signal name**	**External pull-up /down**	**Comment**
D9	LVCMOS33	DP3	--	Data parity 3
D20	LVCMOS33	DREQ0#	pull-up	DMA channel 0 demand mode request
B21	LVCMOS33	DREQ1#	pull-up	DMA channel 1 demand mode request
A5	LVCMOS33	DT/R#	pull-up	Data transmit / receive
E20	LVCMOS33	EOT#	pull-up	End of transfer for current DMA channel
AD13	LVCMOS33	LAD 0	--	Multiplexed data address bus
AC13	LVCMOS33	LAD 1	--	Multiplexed data address bus
AC15	LVCMOS33	LAD 2	--	Multiplexed data address bus
AC16	LVCMOS33	LAD 3	--	Multiplexed data address bus
AA11	LVCMOS33	LAD 4	--	Multiplexed data address bus
AA12	LVCMOS33	LAD 5	--	Multiplexed data address bus
AD14	LVCMOS33	LAD 6	--	Multiplexed data address bus
AC14	LVCMOS33	LAD 7	--	Multiplexed data address bus
AA13	LVCMOS33	LAD 8	--	Multiplexed data address bus
AB13	LVCMOS33	LAD 9	--	Multiplexed data address bus
AA15	LVCMOS33	LAD 10	--	Multiplexed data address bus
AA16	LVCMOS33	LAD 11	--	Multiplexed data address bus
AC11	LVCMOS33	LAD 12	--	Multiplexed data address bus
AC12	LVCMOS33	LAD 13	--	Multiplexed data address bus
AB14	LVCMOS33	LAD 14	--	Multiplexed data address bus
AA14	LVCMOS33	LAD 15	--	Multiplexed data address bus
D12	LVCMOS33	LAD 16	--	Multiplexed data address bus
E13	LVCMOS33	LAD 17	--	Multiplexed data address bus
C16	LVCMOS33	LAD 18	--	Multiplexed data address bus
D16	LVCMOS33	LAD 19	--	Multiplexed data address bus
D11	LVCMOS33	LAD 20	--	Multiplexed data address bus
C11	LVCMOS33	LAD 21	--	Multiplexed data address bus
E14	LVCMOS33	LAD 22	--	Multiplexed data address bus
D15	LVCMOS33	LAD 23	--	Multiplexed data address bus
D13	LVCMOS33	LAD 24	--	Multiplexed data address bus
D14	LVCMOS33	LAD 25	--	Multiplexed data address bus

Local bus signals				
FPGA I/O	**I/O Standard**	**Signal name**	**External pull-up /down**	**Comment**
F15	LVCMOS33	LAD 26	--	Multiplexed data address bus
F16	LVCMOS33	LAD 27	--	Multiplexed data address bus
F11	LVCMOS33	LAD 28	--	Multiplexed data address bus
F12	LVCMOS33	LAD 29	--	Multiplexed data address bus
F13	LVCMOS33	LAD 30	--	Multiplexed data address bus
F14	LVCMOS33	LAD 31	--	Multiplexed data address bus
A8	LVCMOS33	LBE0#	pull-up	Local byte enable 0
F9	LVCMOS33	LBE1#	pull-up	Local byte enable 1
C8	LVCMOS33	LBE2#	pull-up	Local byte enable 2
D8	LVCMOS33	LBE3#	pull-up	Local byte enable 3
	LVCMOS33	LCLK	--	Local processor clock (66MHz)
C17	LVCMOS33	LHOLD	pull-down	Local hold request
D17	LVCMOS33	LHOLDA	pull-down	Local hold acknowledge
F19	LVCMOS33	LINTi#	pull-up	Local interrupt input
A18	LVCMOS33	LINTo#	pull-up	Local interrupt output
B18	LVCMOS33	LRESET#	pull-up	Local bus reset
E10	LVCMOS33	LSERR#	pull-up	Local system error interrupt output
E9	LVCMOS33	LW/R#	pull-up	Local write/read
F18	LVCMOS33	READY#	pull-up	Ready I/O
C10	LVCMOS33	WAIT#	pull-up	Wait I/O
A19	LVCMOS33	USERo	pull-up	General purpose user output
B3	LVCMOS33	GPIO[6] 0	--	General purpose I/O 0
A3	LVCMOS33	GPIO 1	--	General purpose I/O 1
A4	LVCMOS33	GPIO 2	--	General purpose I/O 2
B4	LVCMOS33	GPIO 3	--	General purpose I/O 3

6 GPIO 0,1,2,3 are not accessible by standard PCIBase API

JTAG interface

In addition to configuration via PCIe, it is possible to download configuration data using a JTAG interface. The *PCIeV4BASE* is equipped as standard with a 2- row 14- pin connector to plug-in the *Parallel Cable IV* [7] from Xilinx™. But the JTAG interface is not only suitable to download designs for testing purposes but enables the user to check a running design by the help of software tools provided by Xilinx™, for instance ChipScope[8].

CON1 JTAG connector	
Pin	**Comment**
Pin 1, 3, 5, 7, 9, 11, 13	**GND**
Pin 2	**+3,3 Volt**
Pin 4	TMS
Pin 6	TCK
Pin 8	TDO
Pin 10	TDI
Pin 12, 14	Not connected

7 Parallel Cable IV is not included

8 ChipScope is not included. A demo version is available at the Xilinx™ webpage. (http://www.xilinx.com/ise/optional_prod/cspro.htm)

FPGA design

Introduction

The CESYS PCIeV4BASE Card is shipped with some demonstration FPGA designs to give you an easy starting point for own development projects. The whole source code is written in VHDL. Verilog and schematic entry design flows are not supported.

- The design “pciev4base” demonstrates the implementation of a system-on-chip (SOC) with host software access to the peripherals like GPIOs and DDR2 SODIMM over PCIe.
- The design “performance_test” allows high speed data transfers from and to the FPGA over PCIe and can be used for software benchmarking purposes.

The Virtex4 XCV4LX25 Device is supported by the free Xilinx™ ISE Webpack development software. You will have to change some options of the project properties for own applications. There are some control signals of the PLX PEX8311 PCIe controller routed to FPGA pins, but not used in FPGA designs. These signals must not be pulled into any direction! Otherwise the whole host system could stall! Right click on process “generate programming file”. Then you will have to change properties=>configuration options “unused iob pins” to “float”:

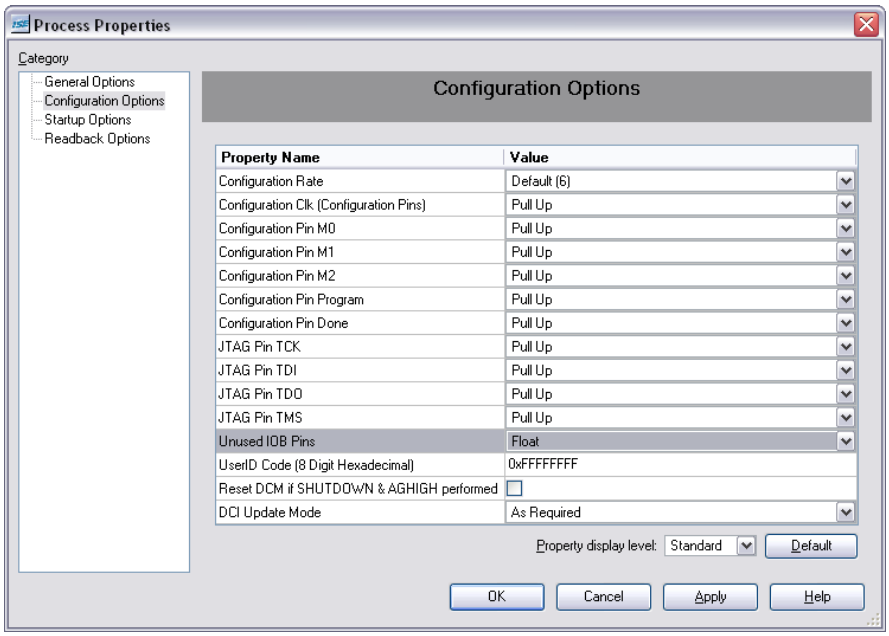


Figure 6: ISE Generate Programming File Properties (Config. Opt.)

A bitstream in the “*.bin”-format is needed, if you want to download your FPGA design with the CESYS software API-functions `LoadBIN()` and `ProgramFPGA()`. The generation of this file is disabled by default in the Xilinx™ ISE development environment. Check “create binary configuration file” at right click “generate programming file”=>properties=>general options:

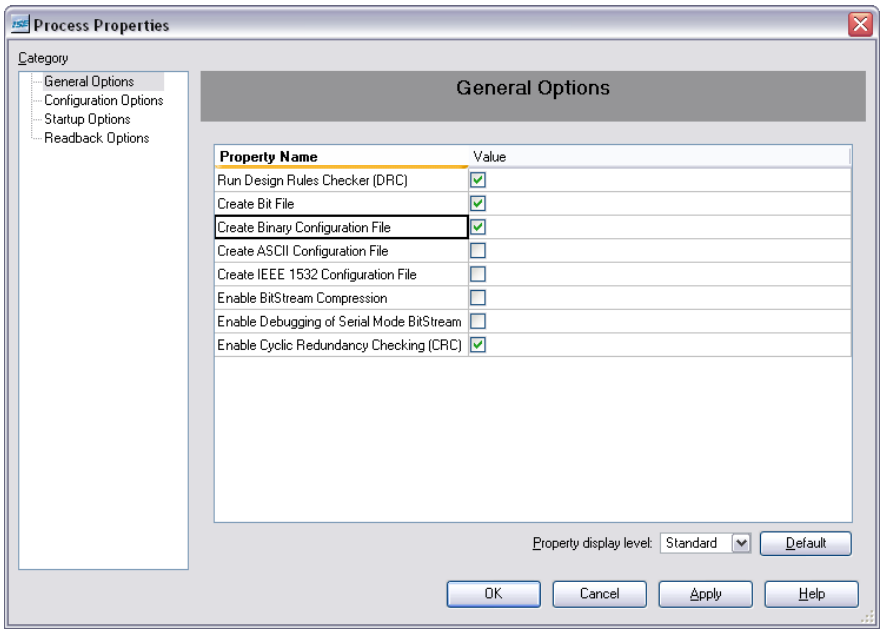


Figure 7: ISE Generate Programming File Properties (Gen. Opt.)

After `ProgramFPGA()` is called and the FPGA design is completely downloaded, the pin LRESET# (note: the postfix # means, that the signal is active low) is automatically pulsed (HIGH/LOW/HIGH). This signal can be used for resetting the FPGA design. The API-function `ResetFPGA()` can be called to initiate a pulse on LRESET# at a user given time. The following sections will give you a brief introduction about the data transfer from and to the FPGA over the PLX PEX8311 PCIe controller's local bus, the WISHBONE interconnection architecture and the provided peripheral controllers. The PCIeV4BASE uses J mode, direct slave, 32 Bit single read/write and DMA single and continuous burst cycles for transferring data. For further information about the PLX local bus see PEX8311 Data Book (PEX_8311AA_Data_Book_v0.90_10Apr06.pdf, chapters 6 to 8) and about the WISHBONE architecture see specification B.3 (wbspec_b3.pdf).

FPGA source code copyright information

This source code is copyrighted by CESYS GmbH / GERMANY, unless otherwise noted.

FPGA source code license

THIS SOURCECODE IS NOT FREE! IT IS FOR USE TOGETHER WITH THE CESYS *PCIeV4BASE* PCIe CARD ONLY! YOU ARE NOT ALLOWED TO MODIFY AND DISTRIBUTE OR USE IT WITH ANY OTHER HARDWARE, SOFTWARE OR ANY OTHER KIND OF ASIC OR PROGRAMMABLE LOGIC DESIGN WITHOUT THE EXPLICIT PERMISSION OF THE COPYRIGHT HOLDER!

Disclaimer of warranty

THIS SOURCECODE IS DISTRIBUTED IN THE HOPE THAT IT WILL BE USEFUL, BUT THERE IS NO WARRANTY OR SUPPORT FOR THIS SOURCECODE. THE COPYRIGHT HOLDER PROVIDES THIS SOURCECODE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THIS SOURCECODE IS WITH YOU. SHOULD THIS SOURCECODE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT WILL THE COPYRIGHT HOLDER BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SOURCECODE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THIS SOURCECODE TO OPERATE WITH ANY OTHER SOFTWARE-PROGRAMS, HARDWARE-CIRCUITS OR ANY OTHER KIND OF ASIC OR PROGRAMMABLE LOGIC DESIGN), EVEN IF THE COPYRIGHT HOLDER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Design "pciev4base"

An on-chip-bus system is implemented in this design. The VHDL source code shows you, how to build a 32 Bit WISHBONE based shared bus architecture. All devices of the WISHBONE system support only SINGLE READ / WRITE Cycles. Files and modules having something to do with the WISHBONE system are labeled with the prefix "wb_". The WISHBONE master is labeled with the additional prefix "ma_" and the slaves are labeled with "sl_".

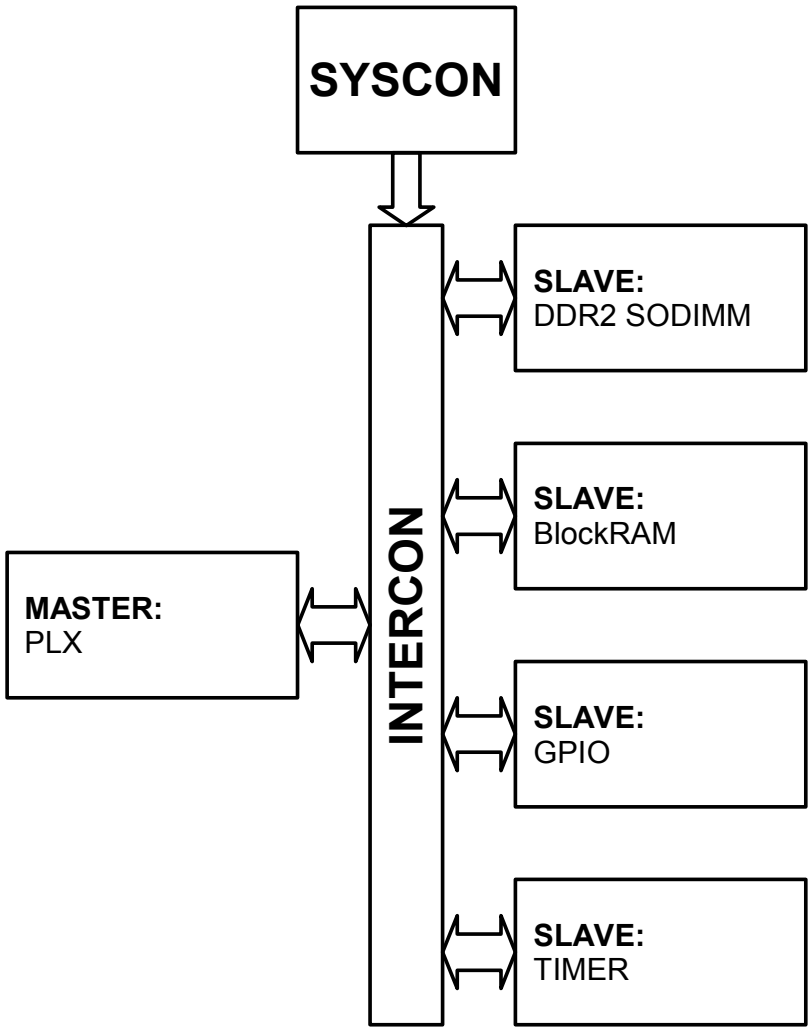


Figure 8: WISHBONE system overview

Files and modules

src/wishbone.vhd:

A package containing datatypes, constants, components, signals and information for software developers needed for the WISHBONE system. You will find C/C++-style "#define"s with important addresses and values to copy and paste into your software source code after VHDL comments ("- -").

src/pciev4base_top.vhd:

This is the top level entity of the design. The WISHBONE components are instantiated here. The internal VHDL signals are mapped to the 100 pin connector of the general purpose I/O plug in boards, so the pinout of the user constraints file does not need to be changed for other plug in boards. You will find a table with the column "HDL Pin" and some pin explanations in the plug in board documentation at the end of this document. This table associates the pin numbers of the FPGA and the 100 pin connector with the bidirectional VHDL data bus port "pin_gpiomoduleport_io".

src/wb_syscon.vhd:

This entity provides the WISHBONE system signals RST and CLK. It uses LRESET# and SYSTEMCLOCK as external reset and clock source.

src/wb_intercon.vhd:

All WISHBONE devices are connected to this shared bus interconnection logic. Some MSBs of the address are used to select the appropriate slave.

src/wb_ma_plx.vhd:

This is the entity of the WISHBONE master, which converts the local bus protocol for 32 Bit single read/write-cycles of the PLX PCI controller into a WISHBONE conform one.

src/wb_sl_bram.vhd:

A internal BlockRAM is instantiated here and simply connected to the WISHBONE architecture.

src/wb_sl_gpio.vhd:

This entity shows you, how to control the dual 8-bit bus transceiver circuits (see 74FCT162245T_Datasheet.pdf for details) on the plug in board and use them as general purpose I/Os. The eight LEDs are controlled by this module as well.

src/wb_sl_timer.vhd:

A 32 Bit timer with programmable period (15 ns steps). The timer starts running if the period is not null. It generates an interrupt at overflow time. The interrupt output is asserted as long as the interrupt is not acknowledged.

src/sl_ddr2.vhd:

This entity connects the memory interface IP core memory_interface_top.vhd created by the Xilinx™ MIG Tool to the WISHBONE system. Xilinx™ LogiCORE FIFOs are used for clock domain translation (DDR2 clock domain <=> WISHBONE clock domain). The prefix "wb_" is missing here, because this slave module is not exactly a WISHBONE device, but uses the WISHBONE architecture for transferring data.

src/xil_mig_ddr2sodimm/:

This directory contains the IP core memory_interface_top.vhd and all other VHDL source code files generated by the Xilinx™ MIG Tool version 1.72. Note that these source code files are copyrighted by Xilinx™ and are absolutely not supported by CESYS! For details on the generated IP see the MIG user guide (ug086.pdf).

ddr2_addr_fifo.vhd|*.ngc|*.xco, ddr2_ram2user_fifo.vhd|*.ngc|*.xco, ddr2_user2ram_fifo.vhd|*.ngc|*.xco:

Design files for Xilinx™ LogiCORE FIFOs. For further details see Xilinx™ FIFO Generator v3.3 user guide and data sheet (fifo_generator_ug175.pdf, fifo_generator_ds317.pdf)

pciev4base.ise:

Project file for Xilinx™ ISE version 9.1.03i

pciev4base.ucf:

User constraint file with timing and pinout constraints

Bus transactions

The API-functions `ReadRegister()`, `WriteRegister()` lead to direct slave single cycles and `ReadBlock()`, `WriteBlock()` to DMA transfers. Bursting is not allowed in the WISHBONE demo application. You can find details on enabling/disabling the local bus continuous burst mode in the software API and the source code of the software examples. There is no difference in the PLX local bus cycles "direct slave" and "DMA", if continuous burst is disabled for DMA transfers. Continuous burst mode can be disabled by the FPGA using the BTERM# signal as well. The address is incremented automatically in block transfers.

Local bus signals driven by the PLX PCIe controller:

- LW/R#: local bus write/not read, indicates, if a read or write cycle is in progress
- ADS#: address strobe, indicates a valid address, if asserted low by PLX
- BLAST#: burst last, indicates the last data cycle in burst mode, can be ignored in single cycle mode

Local bus signals driven by the FPGA:

- READY#: handshake signal, FPGA indicates a successful data transfer for writing and valid data on bus for reading by asserting this signal low, FPGA can insert wait states by delaying this signal
- BTERM#: burst terminate signal, FPGA can break the current burst and request a new address cycle by asserting this signal low. If READY# signal is assigned to the BTERM# pin too, then PLX PCIe controller is forced to transfer all data in single cycles.

Local bus signal driven by the PLX PCI controller and the FPGA:

- LAD[31:0]: 32 Bit multiplexed address/data bus, FPGA drives valid data on this bus in read cycles while asserting the READY# signal low, the FPGA LAD[31:0] output drivers have to be in a high impedance state at all other times

The PLX local bus protocol is converted into a WISHBONE based one. So the PLX becomes a master device in the internal WISHBONE architecture. Input signals for the WISHBONE master are labeled with the postfix “_I”, output signals with “_O”.

WISHBONE signals driven by the master:

- STB_O: strobe, qualifier for the other output signals of the master, indicates valid data and control signals
- WE_O: write enable, indicates, if a write or read cycle is in progress
- ADR_O[31:0]: 32-Bit address bus, the PLX local bus uses BYTE addressing, but the WISHBONE system uses DWORD (32-Bit) addressing. The address is shifted two bits inside the WISHBONE master module
- DAT_O[31:0]: 32-Bit data out bus for data transportation from master to slaves

WISHBONE signals driven by slaves:

- DAT_I[31:0]: 32-Bit data in bus for data transportation from slaves to master
- ACK_I: handshake signal, slave devices indicate a successful data transfer for writing and valid data on bus for reading by asserting this signal, slaves can insert wait states by delaying this signal, this delay leads to a delay of the READY# signal on the local bus side

The signals LHOLD (local hold request) driven by PLX and LHOLDA (local hold acknowledge) driven by the FPGA are used for local bus arbitration. LHOLD can be simply looped back to LHOLDA, because the PLX PCI controller is the one and only master on the local bus.

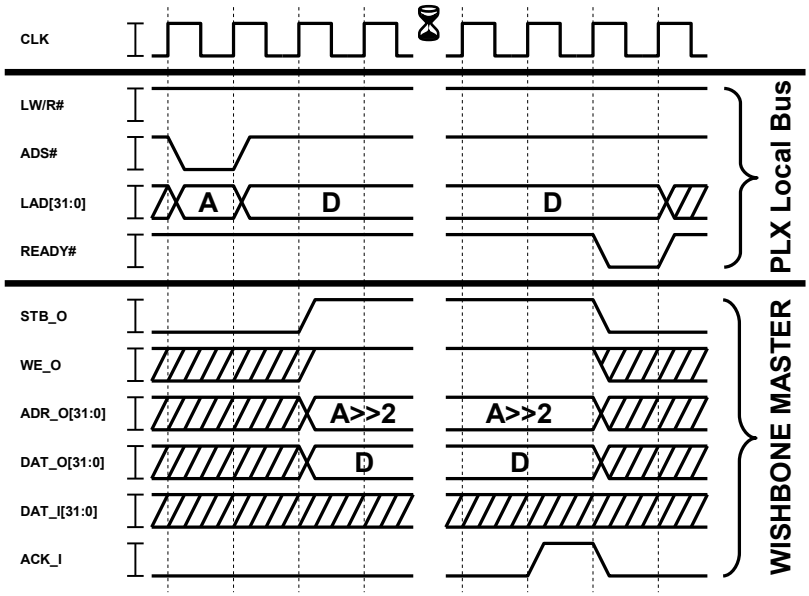


Figure 9: Bus transactions with `WriteRegister()` *and* `WriteBlock()`

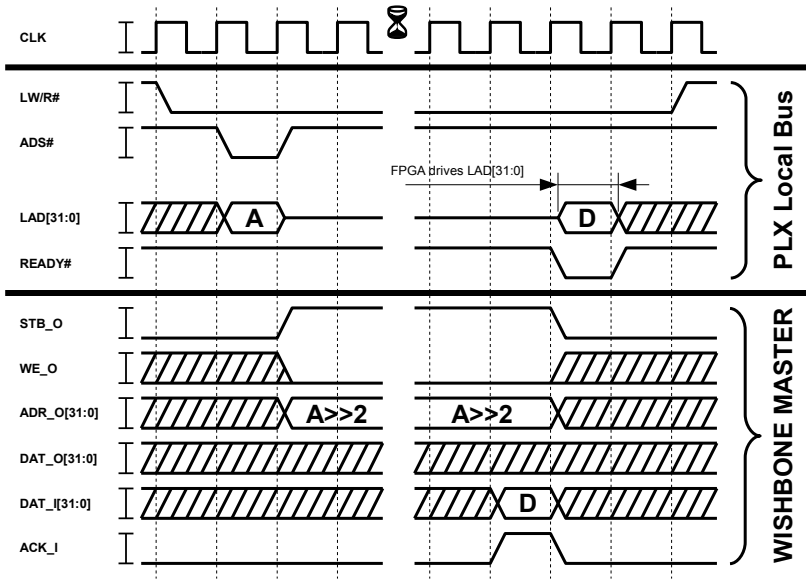


Figure 10: Bus transactions with `ReadRegister()` *and* `ReadBlock()`

The WISHBONE signals in these illustrations and explanations are shown as simple bit types or bit vector types, but in the VHDL code these signals could be encapsulated in extended data types like arrays or records.

Example:

```
...

port map
(
      ...
      ACK_I => intercon.masters.slave(2).ack,
      ...
```

Port ACK_I is connected to signal ack of element 2 of array slave, of record masters, of record intercon.

PCIe interrupt

The FPGA has the possibility to cause PCIe interrupts. The interrupt state can be checked by calling the API-function `WaitForInterrupt()`. If the FPGA asserts the LINTi# (local interrupt input) signal low, then the function returns immediately else it returns after the programmed timeout period. The return value shows you if an interrupt event has been occurred or not. The software has to acknowledge an interrupt, i. e. by writing to a special address. The FPGA deasserts the LINTi# pin after recognizing the acknowledgment. The interrupt functionality is demonstrated by the slave timer module.

Design "performance_test"

Small and simple design to achieve maximum data rates over PCIe. This design has limited functionality. It handles the PLX local bus protocol as fast as possible. The last value transferred to the FPGA is stored in a 32 Bit register. The local bus address information is completely ignored. Some of the stored data bits are routed to the LEDs to demonstrate the Bit/Byte-order.

LEDs	
LED	**Register data bit**
LED0 Green	D00
LED1 Green	D04
LED2 Green	D08
LED3 Green	D12
LED4 Yellow	D16
LED5 Yellow	D20
LED6 Yellow	D24
LED7 Yellow	D28

Files and modules

src/performance_test.vhd:

This is the one and only VHDL source code module. It encapsulates a small and simple local bus protocol engine as well as a single 32-Bit register.

performance_test.ise:

Project file for Xilinx™ ISE version 9.1.03i.

performance_test.ucf:

User constraint file with timing and pinout constraints.

Bus transactions

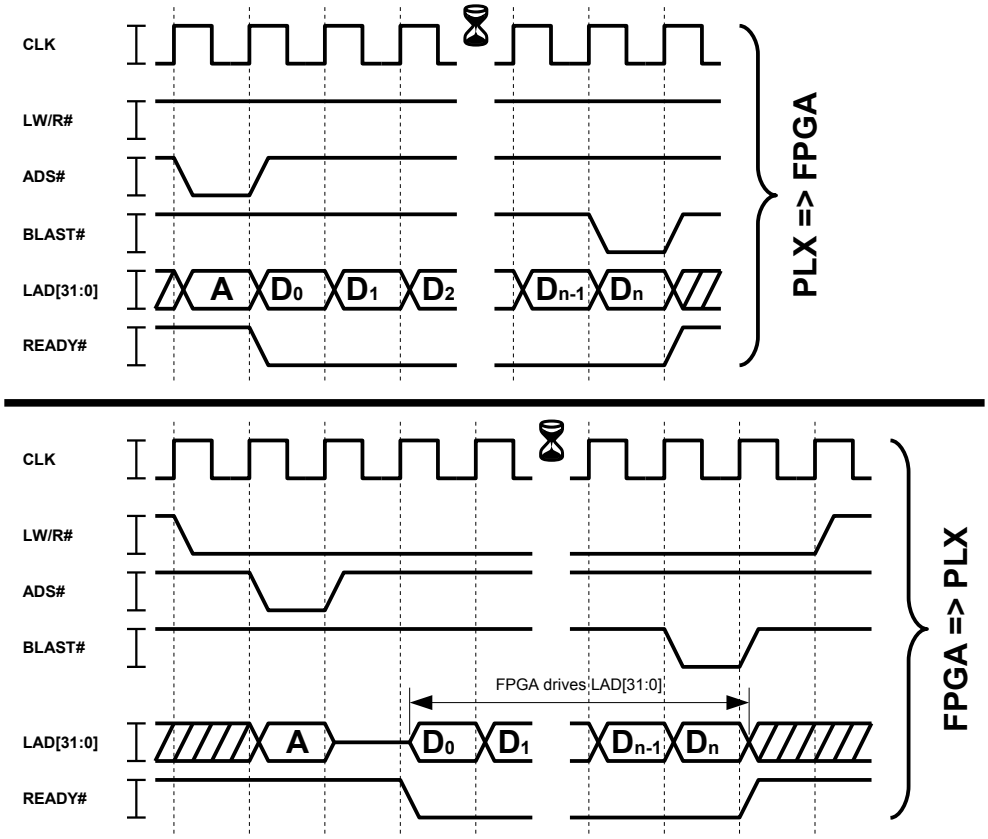


Figure 11: Bus transactions with `ReadBlock()` and `WriteBlock()` in continuous burst mode

This design supports the local bus continuous burst transfers as well as the single cycle transfers. For burst transfers the additional signal BLAST# (burst last) is needed, which is driven by the PLX PCIe controller. If this signal is asserted low, the PLX indicates the last LWORD it wants to transmit or receive. The FPGA can use the READY# signal for inserting wait states like in the single cycle mode. Furthermore the FPGA can drive the additional signal BTERM# (burst terminate) to break the current burst transfer and request a new address cycle. Note that the use of BTERM# is not demonstrated in “performance_test”, because it would decrease the performance.

Software

Introduction

The UDK (Unified Development Kit) is used to allow developers to communicate with Cesys's USB and PCI(e) devices. Older releases were just a release of USB and PCI drivers plus API combined with some shared code components. The latest UDK combines all components into one single C++ project and offers interfaces to C++, C and for .NET (Windows only). The API has functions to mask-able enumeration, unique device identification (runtime), FPGA programming and 32bit bus based data communication. PCI devices have additional support for interrupts.

Changes to previous versions

Beginning with release 2.0, the UDK API is a truly combined interface to Cesys's USB and PCI devices. The class interface from the former USBUni and PCIBase API's was saved at a large extend, so porting applications from previous UDK releases can be done without much work.

Here are some notes about additional changes:

- Complete rewrite
- Build system cleanup, all UDK parts (except .NET) are now part of one large project
- 64 bit operating system support
- UDK tools combined into one application (UDKLab)
- Updated to latest PLX SDK (6.31)
- Identical C, C++ and .NET API interface (.NET ⇒ Windows only)
- Different versions of components collapsed to one UDK version
- Windows only:
  - Microsoft Windows Vista / Seven(7) support (PCI drivers are not released for Seven at the moment)
  - Driver installation / update is done by an installer now
  - Switched to Microsoft's generic USB driver (WinUSB)
  - Support moved to Visual Studio 2005, 2008 and 2010(experimental), older Visual Studio versions are not supported anymore
- Linux only:
  - Revisited USB driver, tested on latest Ubuntu distributions (32/64)
  - Simpler USB driver installation

Windows

Requirements

To use the UDK in own projects, the following is required:

- Installed drivers
- Microsoft Visual Studio 2005 or 2008; 2010 is experimental
- CMake 2.6 or higher ⇒ http://www.cmake.org
- wxWidgets 2.8.10 or higher (must be build separately) ⇒ http://www.wxwidgets.org [optionally, only if UDKLab should be build]

Driver installation

The driver installation is part of the UDK installation but can run standalone on final customer machines without the need to install the UDK itself. During installation, a choice of drivers to install can be made, so it is not necessary to install i.e. PCI drivers on machines that should run USB devices only or vice versa. If USB drivers get installed on a machine that has a pre-2.0 UDK driver installation, we prefer the option for USB driver cleanup offered by the installer, this cleanly removes all dependencies of the old driver installation.

Note: There are separate installers for 32 and 64 bit systems.

Important: At least one device should be present when installing the drivers !

Build UDK

Prerequisites

The most components of the UDK are part of one large CMake project. There are some options that need to be fixed in *msvc.cmake* inside the UDK installation root:

- **BUILD_UI_TOOLS** If *0*, UDKLab will not be part of the subsequent build procedure, if *1* it will. This requires an installation of an already built wxWidgets.
- **WX_WIDGETS_BASE_PATH** Path to wxWidgets build root, only needed if **BUILD_UI_TOOLS** is not *0*.
- **USE_STATIC_RTL** If *0*, all projects are build against the dynamic runtime libraries. This requires the installation of the appropriate Visual Studio redistributable pack on every machine the UDK is used on. Using a static build does not create such dependencies, but will conflict with the standard wxWidgets build configuration.

Solution creation and build

The preferred way is to open a command prompt inside the installation root of the UDK,

lets assume to use *c:\\udkapi*.

```
c:
cd \udkapi
```

CMake allows the build directory separated to the source directory, so it's a good idea to do it inside an empty sub-directory:

```
mkdir build
cd build
```

The following code requires an installation of CMake and at least one supported Visual Studio version. If CMake isn't included into the **PATH** environment variable, the path must be specified as well:

```
cmake ..
```

This searches the preferred Visual Studio installation and creates projects for it. Visual Studio Express users may need to use the command prompt offered by their installation. If multiple Visual Studio versions are installed, CMake's command parameter '-G' can be used to specify a special one, see CMake's documentation in this case. This process creates the solution files inside *c:\\udkapi\\build*. All subsequent tasks can be done in Visual Studio (with the created solution), another invocation of cmake isn't necessary under normal circumstances.

Important: The UDK C++ API must be build with the same toolchain and build flags like the application that uses it. Otherwise unwanted side effects in exception handling will occur ! (See example in *Add project to UDK build*).

Info: It is easy to create different builds with different Visual Studio versions by creating different build directories and invoke CMake with different '-G' options inside them:

```
c:
cd \udkapi
mkdir build2005
cd build2005
cmake -G"Visual Studio 8 2005" ..
cd ..
mkdir build2008
cd build2008
cmake -G"Visual Studio 9 2008" ..
```

Linux

There are too many distributions and releases to offer a unique way to the UDK installation. We've chosen to work with the most recent Ubuntu release, 9.10 at the moment. All commands are tested on an up to date installation and may need some tweaking on other systems / versions.

Requirements

- GNU C++ compiler toolchain
- zlib development libraries
- CMake 2.6 or higher ⇒ http://www.cmake.org
- wxWidgets 2.8.10 or higher ⇒ http://www.wxwidgets.org [optionally, only if UDKLab should be build]

```
sudo apt-get install build-essential cmake zlib1g-dev libwxbase2.8-dev
libwxgtk2.8-dev
```

The Linux UDK comes as gzip'ed tar archive, as the Windows installer won't usually work. The best way is to extract it to the home directory:

```
tar xzvf UDKAPI-x.x.tgz ~/
```

This creates a directory */home/[user]/udkapi[version]* which is subsequently called udkroot. The following examples assume an installation root in *~/udkapi2.0*.

Important: Commands sometimes contain a ` symbol, have attention to use the right one, refer to command substitution if not familiar with.

Drivers

The driver installation on Linux systems is a bit more complicated than on Windows systems. The drivers must be build against the installed kernel version. Updating the kernel requires a rebuild.

USB

As the USB driver is written by Cesys, the installation procedure is designed to be as simple and automated as possible. The sources and support files reside in directory *<udkroot>/drivers/linux/usb*. Just go there and invoke *make*.

```
cd ~/udkapi2.0/drivers/linux/usb
make
```

If all external dependencies are met, the build procedure should finish without errors. Newer kernel releases may change things which prevent success, but it is out of the scope of our possibilities to be always up-to-date with latest kernels. To install the driver, the

following command has to be done:

```
sudo make install
```

This will do the following things:

- Install the kernel module inside the module library path, update module dependencies
- Install a new udev rule to give device nodes the correct access rights (0666) (/etc/udev/rules.d/99-ceusbuni.rules)
- Install module configuration file (/etc/dev/modprobe.d/ceusbuni.conf)
- Start module

If things work as intended, there must be an entry */proc/ceusbuni* after this procedure.

The following code will completely revert the above installation (called in same directory):

```
sudo make remove
```

The configuration file, */etc/modprobe.d/ceusbuni.conf*, offers two simple options (Read the comments in the file):

- Enable kernel module debugging
- Choose between firmware which automatically powers board peripherals or not

Changing these options require a module reload to take affect.

PCI

The PCI drivers are not created or maintained by Cesys, they are offered by the manufacturer of the PCI bridges that were used on Cesys PCI(e) boards. So problems regarding them can't be handled or supported by us.

Important: If building PlxSdk components generate the following error / warning:

```
/bin/sh [[: not found
```

Here's a workaround: The problem is Ubuntu's default usage of *dash* as *sh*, which can't handle command *[[*. Replacing *dash* with *bash* is accomplished by the following commands that must be done as root:

```
sudo rm /bin/sh
sudo ln -s /bin/bash /bin/sh
```

Installation explained in detail:

PlxSdk decompression:

```
cd ~/udkapi2.0/drivers/linux
tar xvf PlxSdk.tar
```

Build drivers:

```
cd PlxSdk/Linux/Driver
PLX_SDK_DIR=`pwd`/../../ ./buildalldrivers
```

Loading the driver manually requires a successful build, it is done using the following commands:

```
cd ~/udkapi2.0/drivers/linux/PlxSdk
sudo PLX_SDK_DIR=`pwd` Bin/Plx_load Svc
```

PCI based boards like the **PCIS3Base** require the following driver:

```
sudo PLX_SDK_DIR=`pwd` Bin/Plx_load 9056
```

PCIe based boards like the **PCIeV4Base** require the following:

```
sudo PLX_SDK_DIR=`pwd` Bin/Plx_load 8311
```

Automation of this load process is out of the scope of this document.

Build UDK

Prerequisites

The whole UDK will be build using CMake, a free cross platform build tool. It creates dynamic Makefiles on unix compatible platforms.

The first thing should be editing the little configuration file *linux.cmake* inside the installation root of the UDK. It contains the following options:

- **BUILD_UI_TOOLS** If *0* UDKLab isn't build, if *1* UDKLab is part of the build, but requires a compatible wxWidgets installation.
- **CMAKE_BUILD_TYPE** Select build type, can be one of *Debug, Release, RelWithDebInfo, MinSizeRel*. If there should be at least 2 builds in parallel, remove this line and specify the type using command line option *-DCMAKE_BUILD_TYPE=….*

Makefile creation and build

Best usage is to create an empty build directory and run cmake inside of it:

```
cd ~/udkapi2.0
mkdir build
cd build
cmake ..
```

If all external dependencies are met, this will finish creating a Makefile. To build the UDK, just invoke make:

```
make
```

Important: The UDK C++ API must be build with the same toolchain and build flags like

the application that uses it. Otherwise unwanted side effects in exception handling will occur ! (See example in *Add project to UDK build*).

Use APIs in own projects

C++ API

- Include file: udkapi.h
- Library file:
  - Windows: udkapi_vc[ver]_[arch].lib, [ver] is *8, 9, 10*, [arch] is *x86* or *amd64*, resides in *lib/[build]/*
  - Linux: libusbapi.so, resides in *lib/*
- Namespace: ceUDK

As this API uses exceptions for error handling, it is really important to use the same compiler and build settings which are used to build the API itself. Otherwise exception based stack unwinding may cause undefined side effects which are really hard to fix.

Add project to UDK build

A simple example would be the following. Let's assume there's a source file *mytest/mytest.cpp* inside UDK's root installation. To build a *mytestexe* executable with UDK components, those lines must be appended:

```
add_executable(mytestexe mytest/mytest.cpp)
target_link_libraries(mytestexe ${UDKAPI_LIBNAME})
```

Rebuilding the UDK with these entries in Visual Studio will create a new project inside the solution (and request a solution reload). On Linux, calling *make* will just include *mytestexe* into the build process.

C API

- Include file: udkapic.h
- Library file:
  - Windows: udkapic_vc[ver]_[arch].lib, [ver] is *8, 9, 10*, [arch] is *x86* or *amd64*, resides in *lib/[build]/*
  - Linux: libusbapic.so, resides in *lib/*
- Namespace: Not applicable

The C API offers all functions from a dynamic link library (Windows: .dll, Linux: .so) and uses standardized data types only, so it is usable in a wide range of environments.

Adding it to the UDK build process is nearly identical to the C++ API description, except that *${UDKAPIC_LIBNAME}* must be used.

.NET API

- Include file: -
- Library file: udkapinet.dll, resided in *bin/[build]*
- Namespace: cesys.ceUDK

The .NET API, as well as it example application is separated from the normal UDK build. First of all, CMake doesn't have native support .NET, as well as it is working on Windows systems only. Building it has no dependency to the standard UDKAPI, all required sources are part of the .NET API project. The Visual Studio solution is located in directory *dotnet/* inside the UDK installation root. It is a Visual Studio 8/2005 solution and should be convertible to newer releases. The solution is split into two parts, the .NET API in mixed native/managed C++ and an example written in C#.

To use the .NET API in own projects, it's just needed to add the generated DLL *udkapinet.dll* to the projects references.

API Functions in detail

Notice: To prevent overhead in most usual scenarios, the API does not serialize calls in any way, so the API user is responsible to serialize call if used in a multi-threaded context !

Notice: The examples for .NET in the following chapter are in C# coding style.

API Error handling

Error handling is offered very different. While both C++ and .NET API use exception handling, the C API uses a classical return code / error inquiry scheme.

C++ and .NET API

UDK API code should be embedded inside a try branch and exceptions of type *ceException* must be caught. If an exception is raised, the generated exception object offers methods to get detailed information about the error.

C API

All UDK C API functions return either *CE_SUCCESS* or *CE_FAILED*. If the latter is returned, the functions below should be invoked to get the details of the error.

Methods/Functions

GetLastErrorCode

API	Code
C++	unsigned int ceException::GetErrorCode()
C	unsigned int GetLastErrorCode()
.NET	uint ceException.GetLastErrorCode()

Returns an error code which is intended to group the error into different kinds. It can be one of the following constants:

Error code	Kind of error
ceE_TIMEOUT	Errors with any kind of timeout.
ceE_IO_ERROR	IO errors of any kind, file, hardware, etc.
ceE_UNEXP_HW_BEH	Unexpected behavior of underlying hardware (no response, wrong data).
ceE_PARAM	Errors related to wrong call parameters (NULL pointers, …).
ceE_RESOURCE	Resource problem, wrong file format, missing dependency.
ceE_API	Undefined behavior of underlying API.
ceE_ORDER	Wrong order calling a group of code (i.e. deinit()→init()).
ceE_PROCESSING	Occurred during internal processing of anything.
ceE_INCOMPATIBLE	Not supported by this device.
ceE_OUTOFMEMORY	Failure allocating enough memory.

GetLastErrorText

API	Code
C++	const char *ceException::GetLastErrorText()
C	const char *GetLastErrorText()
.NET	string ceException.GetLastErrorText()

Returns a text which describes the error readable by the user. Most of the errors contain problems meant for the developer using the UDK and are rarely usable by end users. In most cases unexpected behavior of the underlying operation system or in data transfer is reported. (All texts are in english.)

Device enumeration

The complete device handling is done by the API internally. It manages the resources of all enumerated devices and offers either a device pointer or handle to API users. Calling Init() prepares the API itself, while DeInit() does a complete cleanup and invalidates all device pointers and handles.

To find supported devices and work with them, Enumerate() must be called after Init(). Enumerate() can be called multiple times for either finding devices of different types or to find newly plugged devices (primary USB at the moment). One important thing is the following: Enumerate() does **never** remove a device from the internal device list and so invalidate any pointer, it just add new ones or does nothing, even if a USB device is removed. For a clean detection of a device removal, calling DeInit(), Init() and Enumerate() (in exactly that order) will build a new, clean device list, but invalidates all previous created device pointers and handles.

To identify devices in a unique way, each device gets a UID, which is a combination of device type name and connection point, so even after a complete cleanup and new enumeration, devices can be exactly identified by this value.

Methods/Functions

Init

API	Code
C++	static void ceDevice::Init()
C	CE_RESULT Init()
.NET	static void ceDevice.Init()

Prepare internal structures, must be the first call to the UDK API. Can be called after invoking DeInit() again, see top of this section.

DeInit

API	Code
C++	static void ceDevice::DeInit()
C	CE_RESULT DeInit()
.NET	static void ceDevice.DeInit()

Free up all internal allocated data, there must no subsequent call to the UDK API after this call, except Init() is called again. All retrieved device pointers and handles are invalid after this point.

Enumerate

API	Code
C++	static void ceDevice::Enumerate(ceDevice::ceDeviceType DeviceType)
C	CE_RESULT Enumerate(unsigned int DeviceType)
.NET	static void ceDevice.Enumerate(ceDevice.ceDeviceType DeviceType)

Search for (newly plugged) devices of the given type and add them to the internal list. Access to this list is given by GetDeviceCount() / GetDevice(). DeviceType can be one of the following:

DeviceType	Description
ceDT_ALL	All UDK supported devices.
ceDT_PCI_ALL	All UDK supported devices on PCI bus.
ceDT_PCI_PCIS3BASE	Cesys PCIS3Base
ceDT_PCI_DOB	DOB (*)
ceDT_PCI_PCIEV4BASE	Cesys PCIeV4Base
ceDT_PCI_RTC	RTC (*)
ceDT_PCI_PSS	PSS (*)
ceDT_PCI_DEFLECTOR	Deflector (*)
ceDT_USB_ALL	All UDK supported devices.
ceDT_USB_USBV4F	Cesys USBV4F
ceDT_USB_EFM01	Cesys EFM01
ceDT_USB_MISS2	MISS2 (*)
ceDT_USB_CID	CID (*)
ceDT_USB_USBS6	Cesys USBS6

***** Customer specific devices.

GetDeviceCount

API	Code
C++	static unsigned int ceDevice::GetDeviceCount()
C	CE_RESULT GetDeviceCount(unsigned int *puiCount)
.NET	static uint ceDevice.GetDeviceCount()

Return count of devices enumerated up to this point. May be larger if rechecked after calling Enumerate() in between.

GetDevice

API	Code
C++	static ceDevice *ceDevice::GetDevice(unsigned int uiIdx)
C	CE_RESULT GetDevice(unsigned int uiIdx, CE_DEVICE_HANDLE *pHandle)
.NET	static ceDevice ceDevice.GetDevice(uint uiIdx)

Get device pointer or handle to the device with the given index, which must be smaller than the device count returned by GetDeviceCount(). This pointer or handle is valid up to the point DeInit() is called.

Information gathering

The functions in this chapter return valuable information. All except GetUDKVersionString() are bound to devices and can be used after getting a device pointer or handle from GetDevice() only.

Methods/Functions

GetUDKVersionString

API	Code
C++	static const char *ceDevice::GetUDKVersionString()
C	const char *GetUDKVersionString()
.NET	static string ceDevice.GetUDKVersionString()

Return string which contains the UDK version in printable format.

GetDeviceUID

API	Code
C++	const char *ceDevice::GetDeviceUID()
C	CE_RESULT GetDeviceUID(CE_DEVICE_HANDLE Handle, char *pszDest, unsigned int uiDestSize)
.NET	string ceDevice.GetDeviceUID()

Return string formatted unique device identifier. This identifier is in the form of *type@location* while type is the type of the device (i.e. *EFM01*) and location is the position the device is plugged to. For PCI devices, this is a combination of bus, slot and function (PCI bus related values) and for USB devices a path from device to root hub, containing the port of all used hubs. So after re-enumeration or reboot, devices on the same machine can be identified exactly.

Notice C API: pszDest is the buffer were the value is stored to, it must be at least of size uiDestSize.

GetDeviceName

API	Code
C++	const char *ceDevice::GetDeviceName()
C	CE_RESULT GetDeviceName(CE_DEVICE_HANDLE Handle, char *pszDest, unsigned int uiDestSize)
.NET	string ceDevice.GetDeviceName()

Return device type name of given device pointer or handle.

Notice C API: pszDest is the buffer were the value is stored to, it must be at least of size uiDestSize.

GetBusType

API	Code
C++	ceDevice::ceBusType ceDevice::GetBusType()
C	CE_RESULT GetBusType(CE_DEVICE_HANDLE Handle, unsigned int *puiBusType)
.NET	ceDevice.ceBusType ceDevice.GetBusType()

Return type of bus a device is bound to, can be any of the following:

Constant	Bus
ceBT_PCI	PCI bus
ceBT_USB	USB bus

GetMaxTransferSize

API	Code
C++	unsigned int ceDevice::GetMaxTransferSize()
C	CE_RESULT GetMaxTransferSize(CE_DEVICE_HANDLE Handle, unsigned int *puiMaxTransferSize)
.NET	uint ceDevice.GetMaxTransferSize()

Return count of bytes that represents the maximum in one transaction, larger transfers must be split by the API user.

Using devices

After getting a device pointer or handle, devices can be used. Before transferring data to or from devices, or catching interrupts (PCI), devices must be accessed, which is done by calling Open(). All calls in this section require an open device, which must be freed by calling Close() after usage.

Either way, after calling Open(), the device is ready for communication. As of the fact, that Cesys devices usually have an FPGA on the device side of the bus, the FPGA must be made ready for usage. If this isn't done by loading contents from the on-board flash (not all devices have one), a design must be loaded by calling one of the ProgramFPGA*() calls. These call internally reset the FPGA after design download. From now on, data can be transferred.

Important: All data transfer is based on a 32 bit bus system which must be implemented inside the FPGA design. PCI devices support this natively, while USB devices use a protocol which is implemented by Cesys and sits on top of a stable bulk transfer implementation.

Methods/Functions

Open

API	Code
C++	void ceDevice::Open()
C	CE_RESULT Open(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.Open()

Gain access to the specific device. Calling one of the other functions in this section require a successful call to Open().

Notice: If two or more applications try to open one device, PCI and USB devices behave a bit different. For USB devices, Open() causes an error if the device is already in use. PCI allows opening one device from multiple processes. As PCI drivers are not developed by Cesys, it's not possible to us to prevent this (as we see this as strange behavior). The best way to share communication of more than one application with devices would be a client / server approach.

Close

API	Code
C++	void ceDevice::Close()
C	CE_RESULT Close(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.Close()

Finish working with the given device.

ReadRegister

API	Code
C++	unsigned int ceDevice::ReadRegister(unsiged int uiRegister)
C	CE_RESULT ReadRegister(CE_DEVICE_HANDLE Handle, unsigned int uiRegister, unsigned int *puiValue)
.NET	uint ceDevice.ReadRegister(uint uiRegister)

Read 32 bit value from FPGA design address space (internally just calling ReadBlock() with size = 4).

WriteRegister

API	Code
C++	void ceDevice::WriteRegister(unsiged int uiRegister, unsigned int uiValue)
C	CE_RESULT WriteRegister(CE_DEVICE_HANDLE Handle, unsigned int uiRegister, unsigned int uiValue)
.NET	void ceDevice.WriteRegister(uint uiRegister, uint uiValue)

Write 32 bit value to FPGA design address space (internally just calling WriteBlock() with size = 4).

ReadBlock

API	Code
C++	void ceDevice::ReadBlock(unsiged int uiAddress, unsigned char *pucData, unsigned int uiSize, bool bIncAddress)
C	CE_RESULT ReadBlock(CE_DEVICE_HANDLE Handle, unsigned int uiAddress, unsigned char *pucData, unsigned int uiSize, unsigned int uiIncAddress)
.NET	void ceDevice.ReadBlock(uint uiAddess, byte[] Data, uint uiLen, bool bIncAddress)

Read a block of data to the host buffer which must be large enough to hold it. The size should never exceed the value retrieved by GetMaxTransferSize() for the specific device. bIncAddress is at the moment available for USB devices only. It flags to read all data from the same address instead of starting at it.

WriteBlock

API	Code
C++	void ceDevice::WriteBlock(unsiged int uiAddress, unsigned char *pucData, unsigned int uiSize, bool bIncAddress)
C	CE_RESULT WriteBlock(CE_DEVICE_HANDLE Handle, unsigned int uiAddress,

	unsigned char *pucData, unsigned int uiSize, unsigned int uiIncAddress)
.NET	void ceDevice.WriteBlock(uint uiAddess, byte[] Data, uint uiLen, bool bIncAddress)

Transfer a given block of data to the 32 bit bus system address uiAddress. The size should never exceed the value retrieved by GetMaxTransferSize() for the specific device. bIncAddress is at the moment available for USB devices only. It flags to write all data to the same address instead of starting at it.

WaitForInterrupt

API	Code
C++	bool ceDevice::WaitForInterrupt(unsigned int uiTimeOutMS)
C	CE_RESULT WaitForInterrupt(CE_DEVICE_HANDLE Handle, unsigned int uiTimeOutMS, unsigned int *puiRaised)
.NET	bool ceDevice.WaitForInterrupt(uint uiTimeOutMS)

(PCI only) Check if the interrupt is raised by the FPGA design. If this is done in the time specified by the timeout, the function returns immediately flagging the interrupt is raised (return code / *puiRaised). Otherwise, the function returns after the timeout without signaling.

Important: If an interrupt is caught, EnableInterrupt() must be called again before checking for the next. Besides that, the FPGA must be informed to lower the interrupt line in any way.

EnableInterrupt

API	Code
C++	void ceDevice::EnableInterrupt()
C	CE_RESULT EnableInterrupt(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.EnableInterrupt()

(PCI only) Must be called in front of calling WaitForInterrupt() and every time an interrupt is caught and should be checked again.

ResetFPGA

API	Code
C++	void ceDevice::ResetFPGA()
C	CE_RESULT ResetFPGA(CE_DEVICE_HANDLE Handle)
.NET	void ceDevice.ResetFPGA()

Pulses the FPGA reset line for a short time. This should be used to sync the FPGA design with the host side peripherals.

ProgramFPGAFromBIN

API	Code
C++	void ceDevice::ProgramFPGAFromBIN(const char *pszFileName)
C	CE_RESULT ProgramFPGAFromBIN(CE_DEVICE_HANDLE Handle, const char *pszFileName)
.NET	void ceDevice.ProgramFPGAFromBIN(string sFileName)

Program the FPGA with the Xilinx tools .bin file indicated by the filename parameter. Calls ResetFPGA() subsequently.

ProgramFPGAFromMemory

API	Code
C++	void ceDevice::ProgramFPGAFromMemory(const unsigned char *pszData, unsigned int uiSize)
C	CE_RESULT ProgramFPGAFromMemory(CE_DEVICE_HANDLE Handle, const unsigned char *pszData, unsigned int uiSize)
.NET	void ceDevice.ProgramFPGAFromMemory(byte[] Data, uint Size)

Program FPGA with a given array created with UDKLab. This was previously done using fpgaconv.

ProgramFPGAFromMemoryZ

API	Code
C++	void ceDevice::ProgramFPGAFromMemoryZ(const unsigned char *pszData, unsigned int uiSize)
C	CE_RESULT ProgramFPGAFromMemoryZ(CE_DEVICE_HANDLE Handle, const unsigned char *pszData, unsigned int uiSize)
.NET	void ceDevice.ProgramFPGAFromMemoryZ(byte[] Data, uint Size)

Same as ProgramFPGAFromMemory(), except the design data is compressed.

SetTimeOut

API	Code
C++	void ceDevice::SetTimeOut(unsigned int uiTimeOutMS)
C	CE_RESULT SetTimeOut(CE_DEVICE_HANDLE Handle, unsigned int uiTimeOutMS)
.NET	void ceDevice.SetTimeOut(uint uiTimeOutMS)

Set the timeout in milliseconds for data transfers. If a transfer is not completed inside this timeframe, the API generates a timeout error.

EnableBurst

API	Code
C++	void ceDevice::EnableBurst(bool bEnable)
C	CE_RESULT EnableBurst(CE_DEVICE_HANDLE Handle, unsigned int uiEnable)
.NET	void ceDevice.EnableBurst(bool bEnable)

(PCI only) Enable bursting in transfer, which frees the shared address / data bus between PCI(e) chip and FPGA by putting addresses on the bus frequently only.

UDKLab

Introduction

UDKLab is a replacement of the former cesys-Monitor, as well as cesys-Lab and fpgaconv. It is primary targeted to support FPGA designers by offering the possibility to read and write values from and to an active design. It can further be used to write designs onto the device's flash, so FPGA designs can load without host intervention. Additionally, designs can be converted to C/C++ and C# arrays, which allows design embedding into an application.

The main screen

The following screen shows an active session with an EFM01 device. The base view is intended to work with a device, while additional functionality can be found in the tools menu.

The left part of the screen contains the device initialization details, needed to prepare the FPGA with a design (or just a reset if loaded from flash), plus optional register writes for preparation of peripheral components.

The right side contains elements for communication with the FPGA design:

- Register read and write, either by value or bit-wise using checkboxes.
- Live update of register values.
- Data areas (like RAM or Flash) can be filled from file or read out to file.
- Live view of data areas.
- More on these areas below.

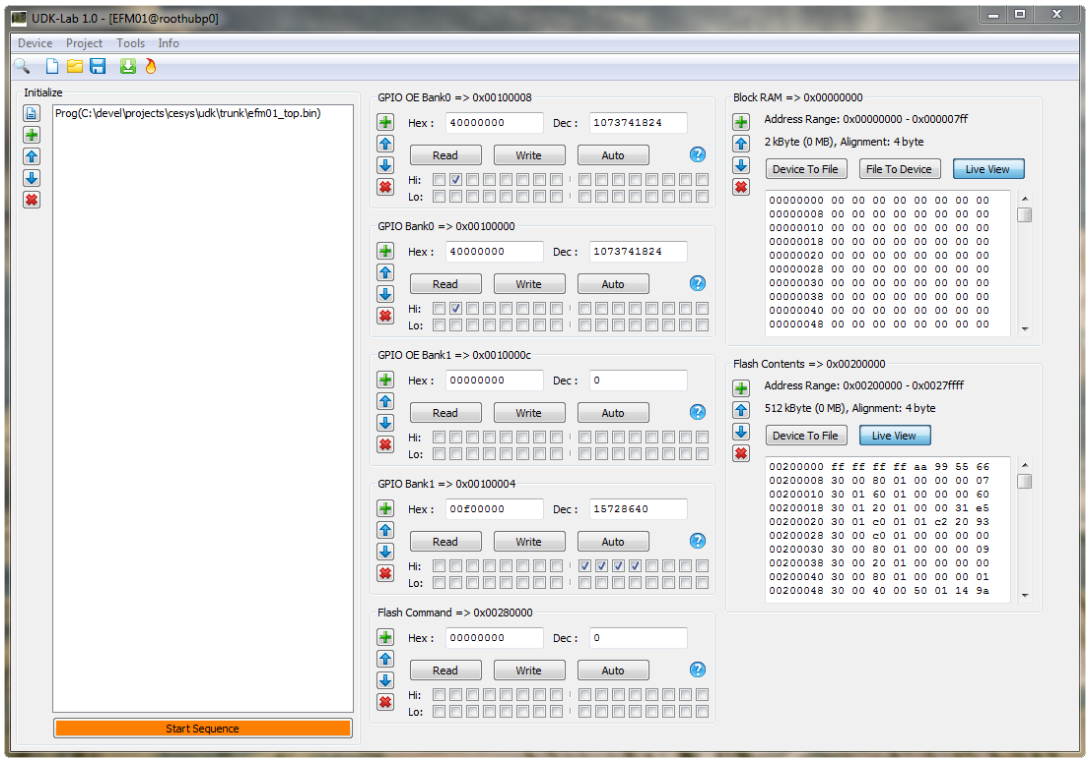


Figure 12: UDKLab Main Screen

Using UDKLab

After starting UDKLab, most of the UI components are disabled. They will be enabled at the point they make sense. As no device is selected, only device independent functions are available:

- The FPGA design array creator
- The option to define USB Power-On behavior
- Info menu contents

All other actions require a device, which can be chosen via the device selector which pops up as separate window:

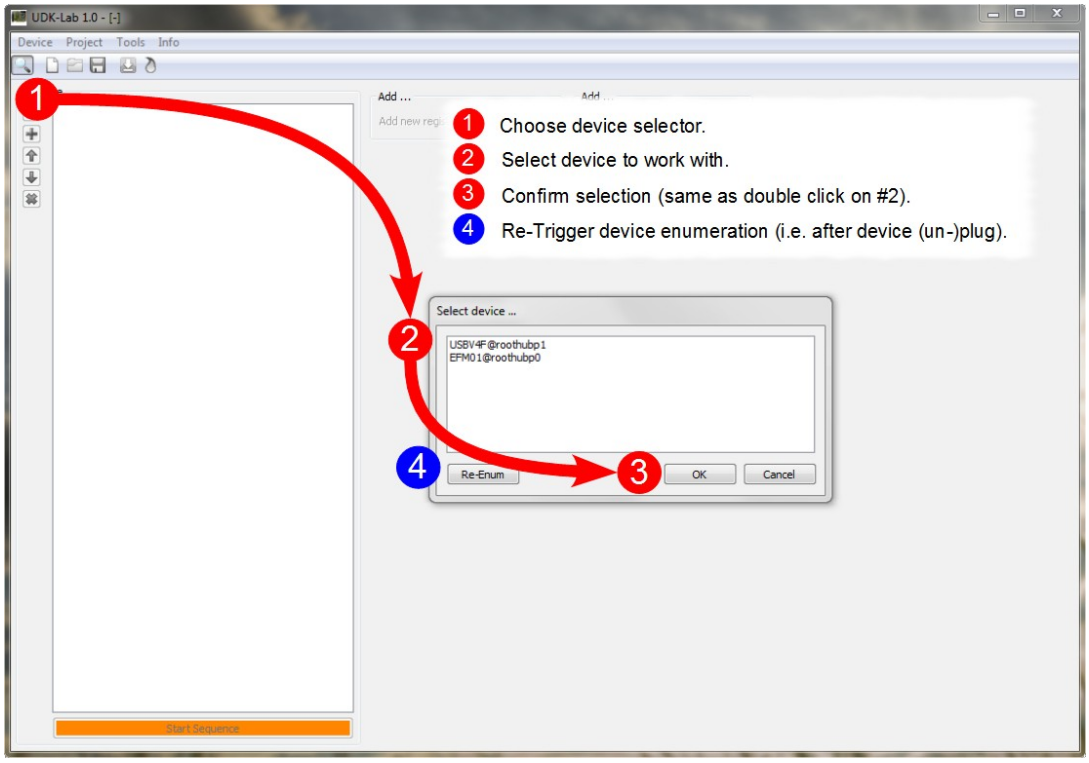


Figure 13: Device selection flow

If the device list is not up to date, clicking Re-Enum will search again. A device can be selected by either double clicking on it or choosing *OK*.

Important: Opening the device selector again will internally re-initialize the underlying API, so active communication is stopped and the right panel is disabled again (more on the state of this panel below).

After a device has been selected, most UI components are available:

- FPGA configuration
- FPGA design flashing [if device has support]
- Project controls
- Initializer controls (Related to projects)

The last disabled component at this point is the content panel. It is enabled if the initialization sequence has been run. The complete flow to enable all UI elements can be seen below:

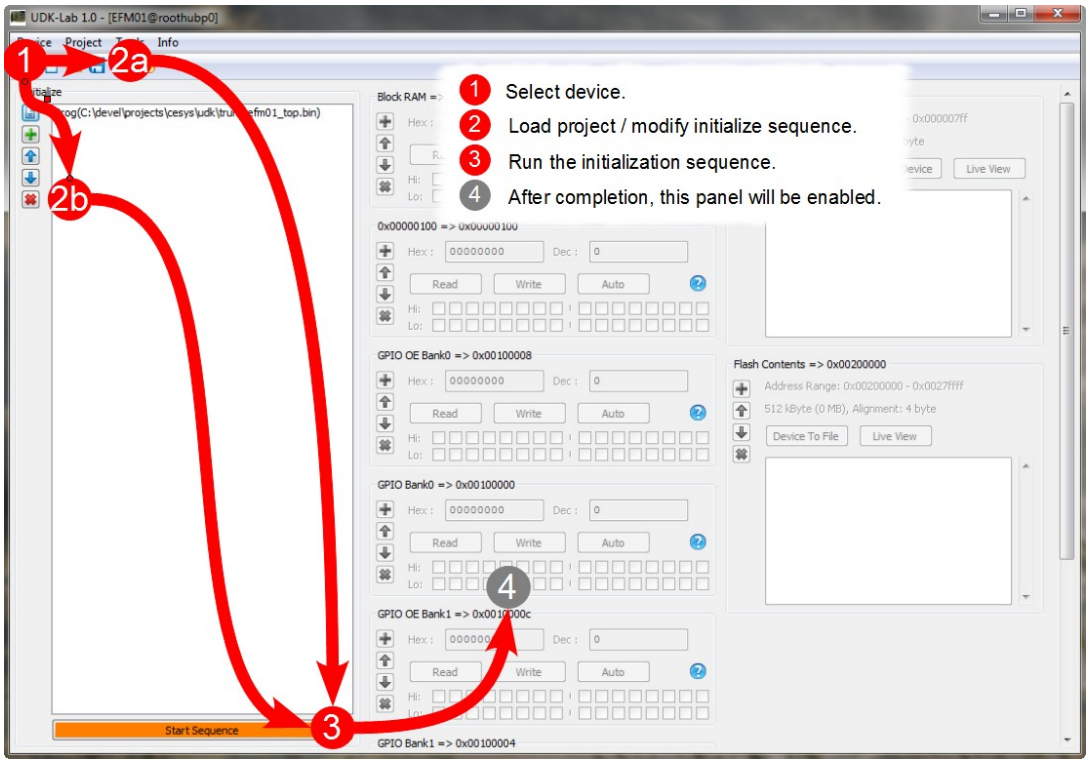


Figure 14: Prepare to work with device

FPGA configuration

Choosing this will pop up a file selection dialog, allowing to choose the design for download. If the file choosing isn't canceled, the design will be downloaded subsequent to closing the dialog.

FPGA design flashing

This option stores a design into the flash component on devices that have support for it. The design is loaded to the FPGA after device power on without host intervention. How and under which circumstances this is done can be found in the hardware description of the corresponding device. The following screen shows the required actions for flashing:

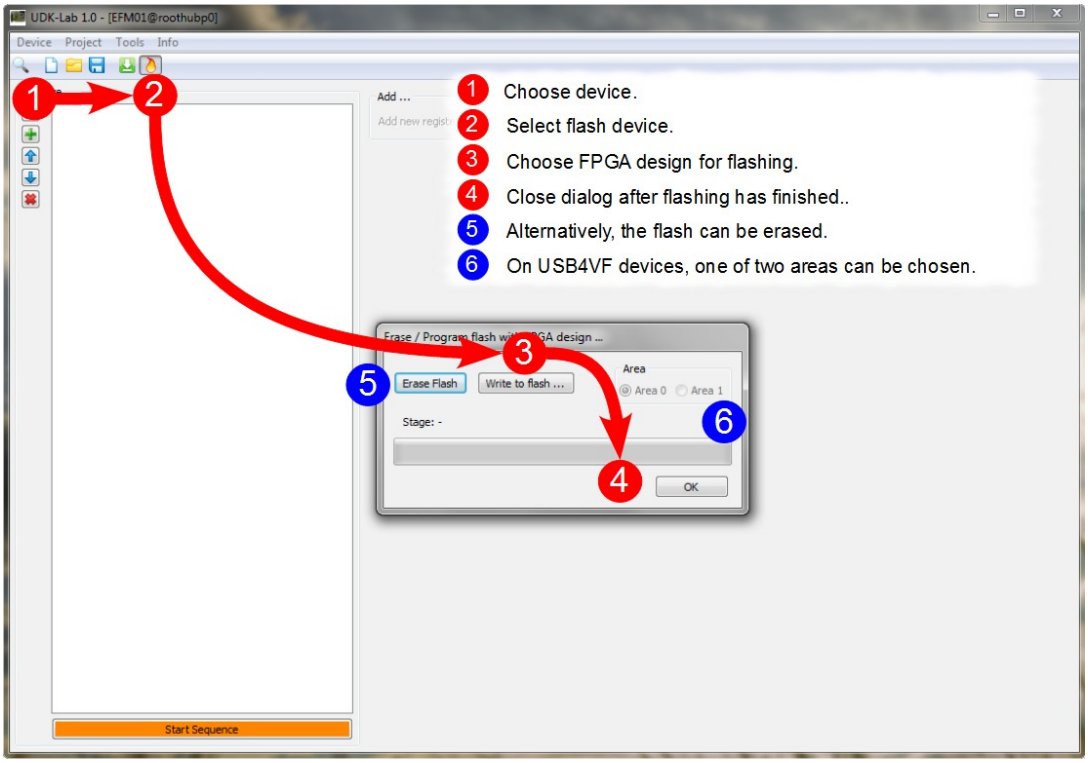


Figure 15: Flash design to device

Projects

Device communication is placed into a small project management. This reduces the actions from session to session and can be used for simple service tasks too. A projects stores the following information:

- Device type it is intended to
- Initializing sequence
- Register list
- Data area list

Projects are handled like files in usual applications, they can be loaded, saved, new

projects can be created. Only one project can be active in one session.

Initializing sequence

The initializing sequence is a list of actions that must be executed in order to work with the FPGA on the device. (The image shows an example initializing list of an EFM01, loading our example design and let the LED blink for some seconds):

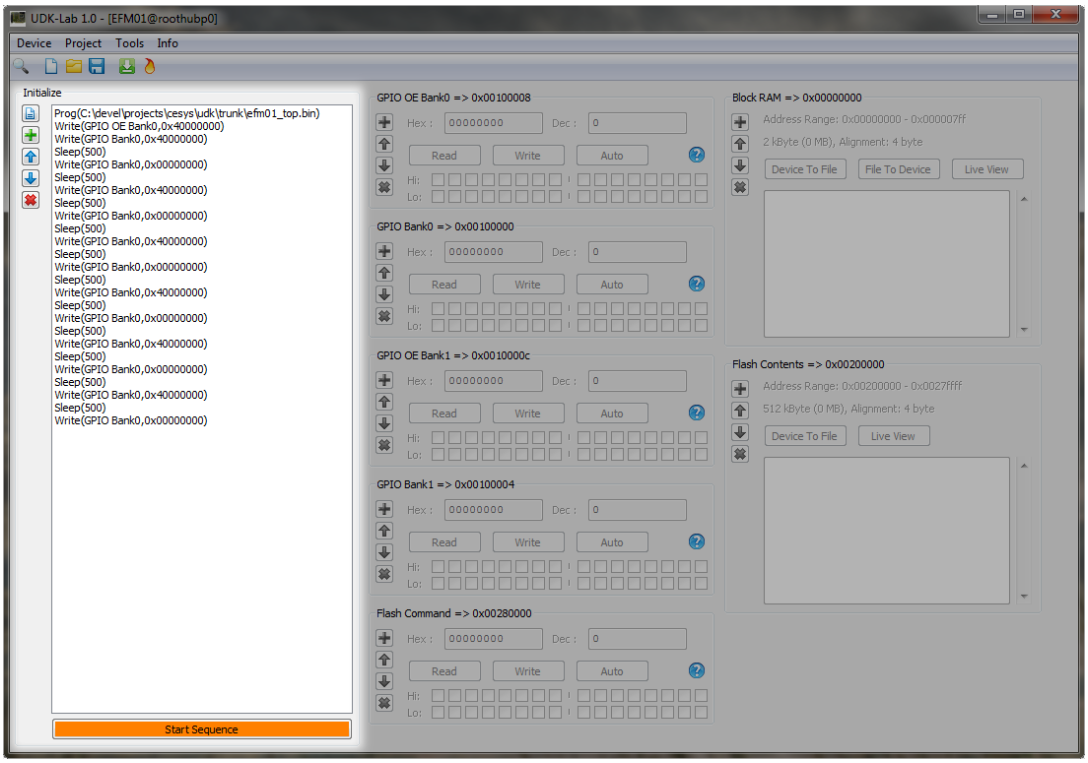


Figure 16: Initializing sequence

Sequence contents

UDKLab supports the following content for initialization:

- FPGA programming
- FPGA reset
- Register write
- Sleep

Without a design, an FPGA does nothing, so it must be loaded before usage. This can be ensured in two ways:

- Download design from host
- Load design from flash (supported on EFM01, USBV4F and USBS6)

So the first entry in the initialize list must be a program entry or, if loaded from flash, a reset entry (To sync communication to the host side). Subsequent to this, a mix of register write and sleep commands can be placed, which totally depends on the underlying FPGA design. This can be a sequence of commands sent to a peripheral component or to fill data structures with predefined values. If things get complexer, i.e. return values must be checked, this goes beyond the scope of the current UDKLab implementation and must be solved by a host process.

To control the sequence, the buttons on the left side can be used. In the order of appearance, they do the following (also indicated by tooltips):

- Clear complete list
- Add new entry (to the end of the list)
- Move currently selected entry on position up
- Move currently selected entry on position down
- Remove currently selected entry

All buttons should be self explanatory, but here's a more detailed look on the add entry, it opens the following dialog:

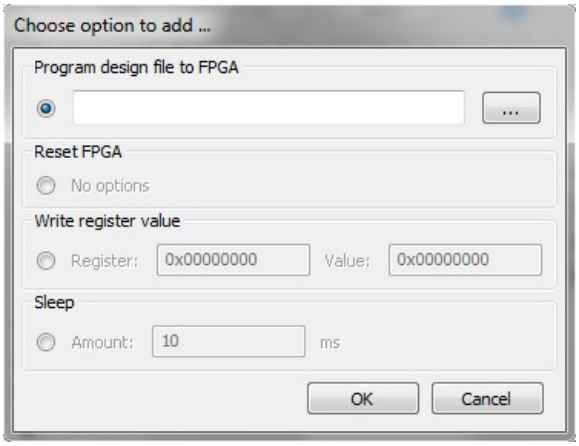


Figure 17: Add new initializing task

One of the four possible entries must be selected using the radio button in front of it. Depending on the option, one or two parameters must be set, *OK* adds the new action to initializer list.

Sequence start

The button sitting below the list runs all actions from top to bottom. In addition to this, the remaining UI components, the content panel, will be enabled, as UDKLab expects a working communication at this point. The sequence can be modified an started as often as wished.

Content panel

The content panel can be a visual representation of the FPGA design loaded during initialization. It consists of a list of registers and data areas, which can be visit and modified using UDKLab. The view is split into two columns, while the left part contains the registers and the right part all data area / block entries.

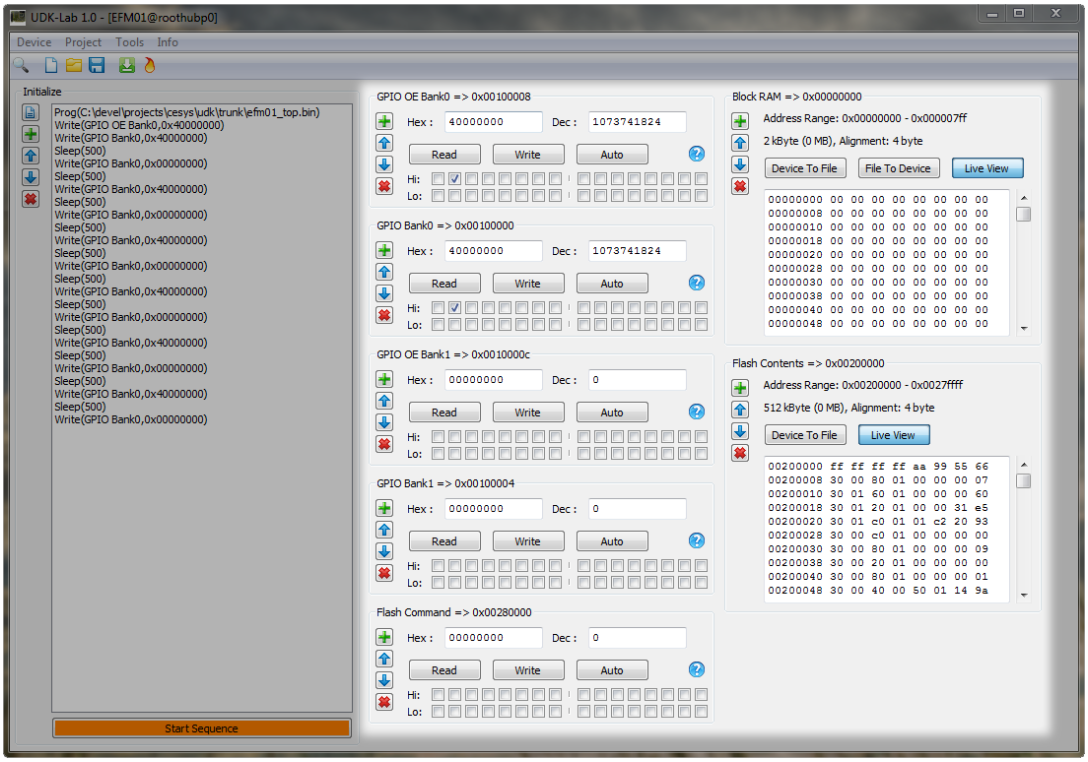


Figure 18: Content panel

Register entry

A register entry can be used to communicate with a 32 bit register inside the FPGA. In UDKLab, a register consists of the following values:

- Address
- Name
- Info text

The visual representation of one register can be seen in the following image:

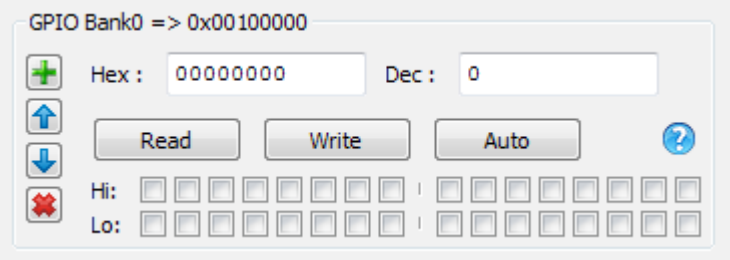


Figure 19: Register panel

The left buttons are responsible for adding new entries, move the entry up or down and removing the current entry, all are self explanatory. The header shows it's mapping name as well as the 32 bit address. The question mark in the lower right will show a tooltip if the mouse is above it, which is just a little help for users. Both input fields can be used to write in a new value, either hex- or decimal or contain the values if they are read from FPGA design. The checkboxes represent one bit of the current value. Clicking the *Read* button will read the current value from FPGA and update both text boxes as well as the checkboxes, which is automatically done every 100ms if the *Auto* button is active. Setting register values inside the FPGA is done in a similar way, clicking the *Write* button writes the current values to the device. One thing needs a bit attention here:

Clicking on the checkboxes implicitly writes the value without the need to click on the *Write* button !

Data area entry

A data area entry can be used to communicate with a data block inside the FPGA, examples are RAM or flash areas. Data can be transfered from and to files, as well as displayed in a live view. An entry constits of the following data:

- Address
- Name
- Data alignment
- Size
- Read-only flag

The visual representation is shown below.

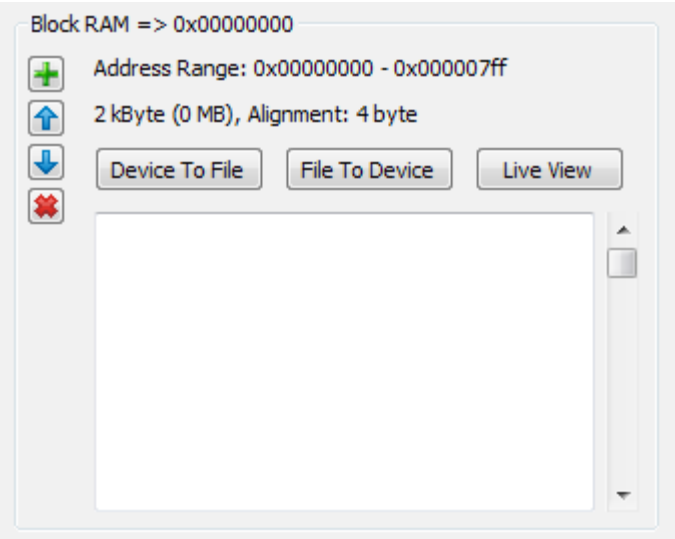


Figure 20: Data area panel

Similar to the register visualization, the buttons on the right side can be used to add, move and remove data area panels. The header shows the name and the address followed by the data area details. Below are these buttons:

- Device To File: The complete area is read and stored to the file which is defined in the file dialog opening after clicking the button.
- File To Device: This reads the file selected in the upcoming file dialog and stores the contents in the data area, limited by the file size or data area size. This button is not shown if the Read-only flag is set.
- Live View: If this button is active, the text view below shows the contents of the area, updated every 100 ms, the view can be scrolled, so every piece can be visited.

Additional Information

References

- CESYS PCIeV4BASE software API and sample code
- PLX PEX8311 PCIe controller data book (PEX_8311AA_Data_Book_v0.90_10Apr06.pdf)
- Specification for the “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores” Revision B.3, released September 7, 2002 (wbspec_b3.pdf)
- Dual 8 Bit transceiver data sheet (74FCT162245T_Datasheet.pdf)

Links

- http://www.vhdl-online.de/
- Informations about the VHDL language, including a tutorial, a language reference, design hints for describing state machines, synthesis and the synthesizable language subset
- http://www.opencores.org/projects.cgi/web/wishbone/
- Home of the WISHBONE standard
- http://www.plxtech.com/
- Provider of the PLX PEX8311 PCIe controller
- http://www.xilinx.com/
- Provider of the Virtex-4™ FPGA and the free FPGA development environment ISE WebPACK

Data Rates

PCIe and DDR2 SDRAM technology are build for achieving very high data transfer rates. The theoretical maximum values of these data rates differ from real life applications. This chapter provides information about measured data rates over PLX local bus and DDR2 memory bus connected to FPGA balls. There can be no “one-for-all” architecture to process huge amounts of data in high performance applications. So normally a lot of optimization has to be done on software and FPGA design side. Buffer sizes, buffer thresholds, prefetch logic and latency of peripherals have great influence on data throughput, especially in data streaming/bursting applications.

The FPGA is always a slave at PLX local bus. A special design was used for benchmarking, which responds to PLX local bus master requests with zero cycle latency. The FPGA handles the local bus protocol and acts as a endless data source/sink. The local bus clock frequency is 66 MHz.

PCIe Benchmark	
Direction	**Result**
PCIe => Local Bus	125 MBytes/sec
Local Bus => PCIe	135 MBytes/sec

The hardware interface FPGA/DDR2 is designed to work together with IP-cores generated by the Xilinx MIG tool. The memory controller has a FIFO-like interface for addresses/commands and data. The benchmark FPGA design generates a neverending datastream with consecutive addresses for write accesses and neverending data requests with consecutive addresses for read accesses. The data direction is software defined. Addresses, data and commands are written as soon as there is space available in the memory controller's FIFO interface.

The memory controller has been generated using Xilinx MIG Tool version 1.72, the DDR2 burst length has been set to four and the selected device has been MT8HTF6464HDY-40E. The DDR2 clock frequency is 200 MHz.

DDR2 SODIMM Benchmark	
Direction	**Result**
FPGA => DDR2	3 007 MBytes/sec
DDR2 => FPGA	3 038 MBytes/sec

Host system configuration:

Operating system:	Windows XP™ Pro 32 Bit, SP2
PCIe driver:	PLX Technology device driver version 5.0
Mainboard:	Asus P5B-E, Intel P965 Chipset
CPU:	Intel Core2 Duo 6320, 2 x 1866 MHz
RAM:	2 GByte PC2-5300 CL5, Kingston KVR667D2N5K2/2G
HDD:	250 GByte SATA, WDC WD2500YD-01NVB1
Graphics adapter:	Asus EN7300TC512

PIBIO Etch length report

In most cases it is not necessary to consider etch lengths on *PCIeV4BASE* when designing user specific Plug-In boards. For the rare cases, where timing margins are critical or nets have to be length matched, the following chart gives information about etch lengths between FPGA I/O ball and respective PIB pin on connector CON4.

PIBIO etch length report				
PIB pin	**Net name**	**FPGA I/O**	**Etch length (mm)**	**Comment**
1	PIB_IO0	AB17	82,743	Bank4, L3P GC
2	PIB_IO1	U21	77,571	Bank 9, L29N
3	PIB_IO2	U22	79,092	Bank 9, L29P
4	**GND**	--	--	--
5	PIB_IO3	T19	74,304	Bank 9, L31N
6	PIB_IO4	U20	73,349	Bank 9, L31P
7	PIB_IO5	R23	76,405	Bank 9, L22N
8	PIB_IO6	R24	76,412	Bank 9, L22P
9	PIB_IO7	R21	76,579	Bank 9, L23N
10	PIB_IO8	R22	84,736	Bank 9, L23P
11	PIB_IO9	T23	67,348	Bank 9, L24N CC
12	PIB_IO10	T24	67,579	Bank 9, L24P CC
13	PIB_IO11	R20	66,617	Bank 9, L25P CC
14	PIB_IO12	R19	70,156	Bank 9, L25N CC
15	PIB_IO13	P19	69,738	Bank 9, L21N
16	PIB_IO14	P20	66,467	Bank 9, L21P
17	PIB_IO15	N19	69,713	Bank 9, L9N CC
18	PIB_IO16	M19	82,691	Bank 9, L9P CC
19	PIB_IO17	K26	58,770	Bank 9, L8P CC
20	PIB_IO18	K25	62,150	Bank 9, L8N CC
21	PIB_IO19	M23	63,851	Bank 9, L14P
22	PIB_IO20	M22	62,141	Bank 9, L14N
23	**GND**	--	--	--
24	PIB_IO21	M21	60,650	Bank 9, L13P
25	PIB_IO22	M20	74,899	Bank 9, L13N
26	PIB_IO23	L21	58,991	Bank 9, L6P
27	PIB_IO24	L20	69,975	Bank 9, L6N
28	PIB_IO25	L19	71,487	Bank 9, L5P
29	PIB_IO26	K20	59,517	Bank 9, L5N
30	PIB_IO27	K22	53,493	Bank 9, L3P
31	PIB_IO28	K21	53,000	Bank 9, L3N
32	PIB_IO29	J21	59,164	Bank 9, L1P
33	PIB_IO30	J20	71,060	Bank 9, L1N
34	PIB_IO31	F24	53,008	Bank 5, L24P CC

PIBIO etch length report				
PIB pin	**Net name**	**FPGA I/O**	**Etch length (mm)**	**Comment**
35	PIB_IO32	F23	56,102	Bank 5, L24N CC
36	PIB_IO33	E23	63,894	Bank 5, L18P
37	PIB_IO34	E22	64,332	Bank 5, L18N
38	PIB_IO35	D24	54,130	Bank 5, L16P
39	PIB_IO36	C24	144,104	Bank 5, L16N
40	PIB_IO37	D23	54,288	Bank 5, L21P
41	PIB_IO38	C23	50,959	Bank 5, L21N
42	PIB_IO39	A21	54,098	Bank 5, L15N
43	PIB_IO40	A22	47,495	Bank 5, L15P
44	PIBCLK	--	--	66MHz clock signal
45	**GND**	--	--	--
46	PIB_IO41	A23	48,579	Bank 5, L8N CC
47	PIB_IO42	A24	49,467	Bank 5, L8P CC
48	**+3,3 Volt**	--	--	--
49	**+3,3 Volt**	--	--	--
50	**+3,3 Volt**	--	--	--
51	PIB_IO43	C25	29,844	Bank 5, L20N
52	PIB_IO44	C26	30,034	Bank 5, L20P
53	PIB_IO45	D25	31,010	Bank 5, L25N CC
54	PIB_IO46	D26	29,649	Bank 5, L25P CC
55	PIB_IO47	E24	40,062	Bank 5, L27N
56	PIB_IO48	E25	42,733	Bank 5, L27P
57	PIB_IO49	E26	32,405	Bank 5, L29N
58	PIB_IO50	F26	33,051	Bank 5, L29P
59	PIB_IO51	G23	40,754	Bank 5, L28N
60	PIB_IO52	G24	39,907	Bank 5, L28P
61	PIB_IO53	H23	58,906	Bank 5, L30N
62	PIB_IO54	H24	72,742	Bank 5, L30P
63	PIB_IO55	G25	41,899	Bank 5, L31N
64	PIB_IO56	G26	57,826	Bank 5, L31P
65	PIB_IO57	H25	47,961	Bank 5, L32N
66	PIB_IO58	H26	56,948	Bank 5, L32P
67	PIB_IO59	J22	56,817	Bank 9, L2N
68	PIB_IO60	J23	57,026	Bank 9, L2P

PIBIO etch length report				
PIB pin	**Net name**	**FPGA I/O**	**Etch length (mm)**	**Comment**
69	PIB_IO61	J25	42,968	Bank 9, L4N
70	PIB_IO62	J26	40,162	Bank 9, L4P
71	PIB_IO63	K23	69,578	Bank 9, L7N
72	PIB_IO64	K24	50,584	Bank 9, L7P
73	PIB_IO65	L23	48,473	Bank 9, L10N
74	PIB_IO66	L24	53,364	Bank 9, L10P
75	PIB_IO67	M24	48,694	Bank 9, L11N
76	PIB_IO68	M25	45,996	Bank 9, L11P
77	PIB_IO69	L26	51,158	Bank 9, L12P
78	PIB_IO70	M26	47,973	Bank 9, L12N
79	PIB_IO71	N20	65,031	Bank 9, L17N
80	PIB_IO72	N21	54,915	Bank 9, L17P
81	PIB_IO73	N22	56,137	Bank 9, L16N
82	PIB_IO74	N23	54,690	Bank 9, L16P
83	PIB_IO75	N24	60,018	Bank 9, L15N
84	PIB_IO76	N25	55,095	Bank 9, L15P
85	PIB_IO77	P23	61,166	Bank 9, L19P
86	PIB_IO78	P22	63,766	Bank 9, L19N
87	PIB_IO79	P25	59,243	Bank 9, L18P
88	PIB_IO80	P24	60,737	Bank 9, L18N
89	PIB_IO81	R26	59,402	Bank 9, L20P
90	PIB_IO82	R25	63,008	Bank 9, L20N
91	PIB_IO83	T21	62,649	Bank 9, L30P
92	PIB_IO84	T20	66,250	Bank 9, L30N
93	PIB_IO85	U23	67,624	Bank 9, L27P
94	PIB_IO86	V23	63,089	Bank 9, L27N
95	PIB_IO87	U25	67,634	Bank 9, L28P
96	PIB_IO88	U24	65,147	Bank 9, L28N
97	PIB_IO89	T26	76,830	Bank 9, L26P
98	PIB_IO90	U26	66,616	Bank 9, L26N
99	PIB_IO91	V26	71,907	Bank 9, L32P
100	PIB_IO92	V25	74,963	Bank 9, L32N

78-pin HD-Sub Connector diagram

The following diagram links 100-pin PIB- to 78-pin HD-Sub connector.

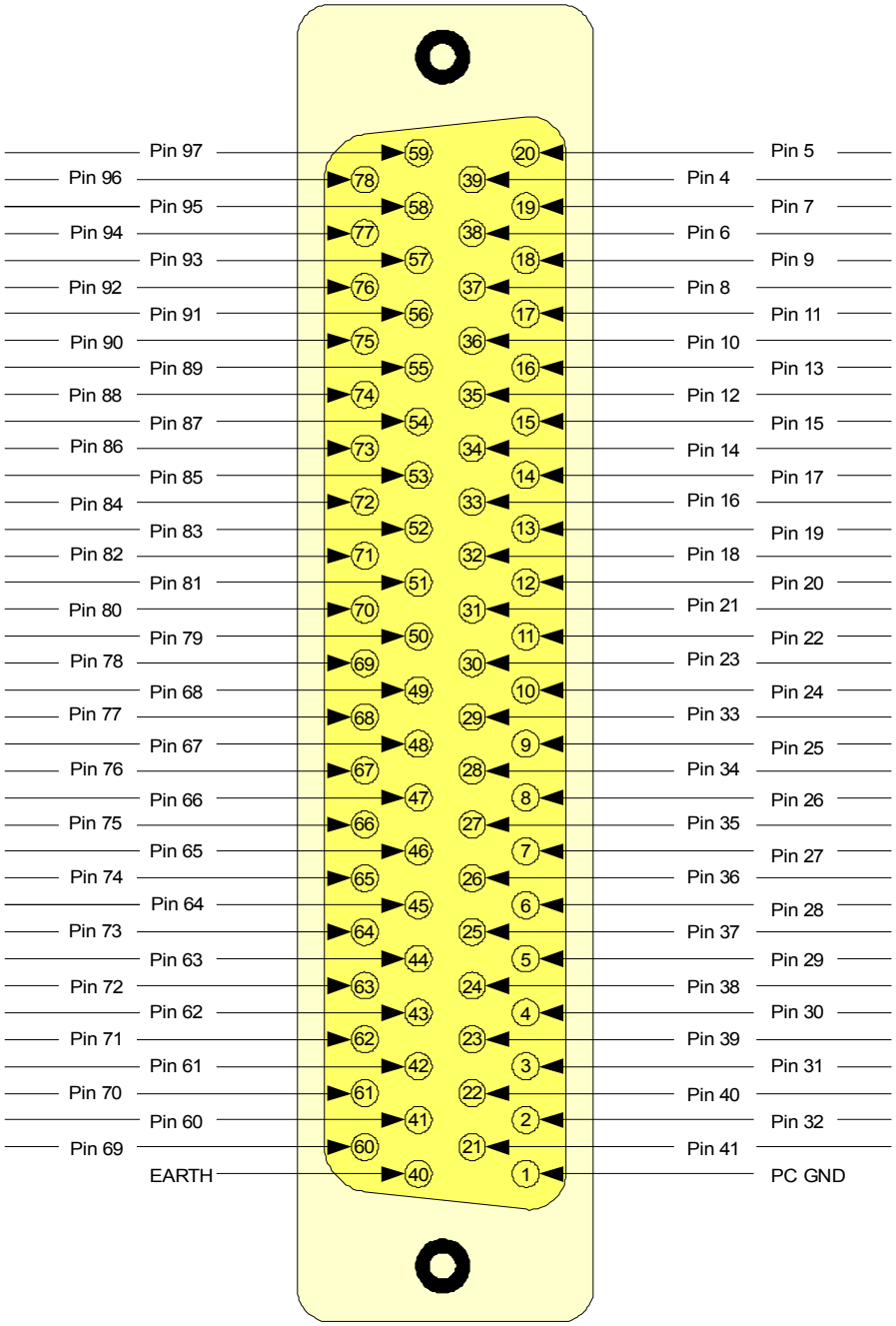


Table of contents

Table of Contents

Copyright information....2

Overview....3

Summary of PCIeV4Base....3

Feature list....3

Hardware....5

Virtex-4 FPGA....5

SODIMM Memory module....6

PCI Express interface....6

CESYS PIB slot....7

Board size....8

Connectors and FPGA pinout....9

Clock signals....9

DDR PLLCLK....10

SODIMM socket....11

PIB signals and SUB-D connector....19

Leds....23

PCIe Controller local bus signals....24

JTAG interface....27

FPGA design....28

Introduction....28

FPGA source code copyright information....30

FPGA source code license....30

Disclaimer of warranty....30

Design “pciev4base”....31

Files and modules....32

src/wishbone.vhd:....32

src/pciev4base_top.vhd:....32

src/wb_syscon.vhd:....32

src/wb_intercon.vhd:....32

src/wb_ma_plx.vhd:....32

src/wb_sl_bram.vhd:....32

src/wb_sl_gpio.vhd:....32

src/wb_sl_timer.vhd:....33

src/sl_ddr2.vhd:....33
src/xil_mig_ddr2sodimm/:....33
ddr2_addr_fifo.vhd|*.ngc|*.xco, ddr2_ram2user_fifo.vhd|*.ngc|*.xco, ddr2_user2ram_fifo.vhd|*.ngc|*.xco:....33
pciev4base.ise:....33
pciev4base.ucf:....33
Bus transactions....33
Local bus signals driven by the PLX PCIe controller:....34
Local bus signals driven by the FPGA:....34
Local bus signal driven by the PLX PCI controller and the FPGA:....34
WISHBONE signals driven by the master:....34
WISHBONE signals driven by slaves:....34
Example:....36
PCIe interrupt....36
Design “performance_test”....37
Files and modules....37
src/performance_test.vhd:....37
performance_test.ise:....37
performance_test.ucf:....37
Bus transactions....38
Software....39
Introduction....39
Changes to previous versions....39
Windows....40
Requirements....40
Driver installation....40
Build UDK....40
Prerequisites....40
Solution creation and build....40
Linux....42
Requirements....42
Drivers....42
USB....42
PCI....43
Build UDK....44
Prerequisites....44
Makefile creation and build....44
Use APIs in own projects....46
C++ API....46
Add project to UDK build....46
C API....46
.NET API....47

API Functions in detail....47
API Error handling....47
C++ and .NET API....47
C API....47
Methods/Functions....48
Device enumeration....49
Methods/Functions....49
Information gathering....52
Methods/Functions....52
Using devices....54
Methods/Functions....54
UDKLab....59
Introduction....59
The main screen....60
Using UDKLab....61
FPGA configuration....62
FPGA design flashing....63
Projects....63
Initializing sequence....64
Content panel....66

Additional Information....69
References....69
Links....69
Data Rates....69
PIBIO Etch length report....70
78-pin HD-Sub Connector diagram....74

Table of contents....75