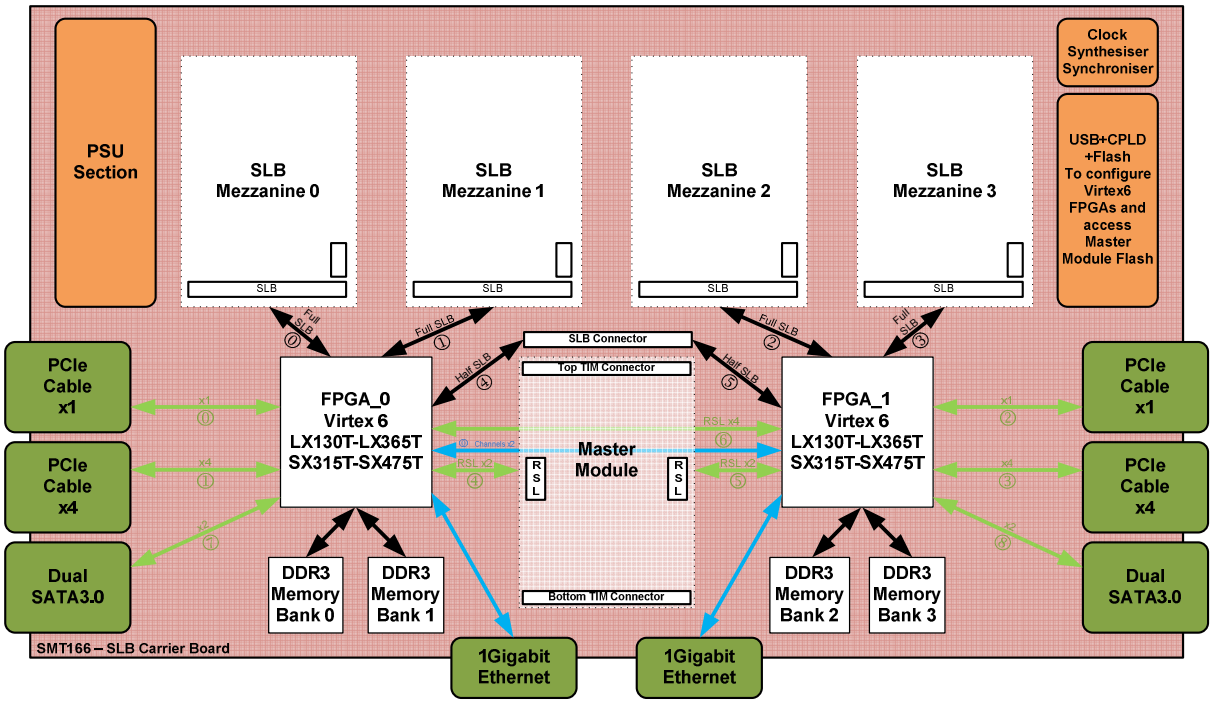
SMT166 FPGA Functionality

18th April 2012

Contents

Board Architecture

The SMT166 SLB carrier board architecture is shown below:



The design is mostly symmetrical with each Virtex 6 FPGA having identical connectivity (logical connections and PCB trace routing) to:

1) Two banks of DDR3 memory. Each bank is composed using 2 x 64Mx16 DDR3 devices. The clock frequency is 500MHz thus giving a data rate of 4GB/s.

2) Two SLB interfaces. Each SLB interface can carry 4 x 14-bit ADC interfaces from, for example, an SMT941 quad ADC module. The SLB interface runs at up to 250MHz, so the data rate is 2GB/s.

Each SLB interface has a separate serial programming/configuration protocol referred to as SPI.

3) A 1-lane and a 4-lane cable PCIe interface.

4) Two SATA interfaces.

5) A gigabit Ethernet interface.

6) RSL (Rocket serial link, also called GTP, GTX and/or MGT). These can operate up to 6.5Gb/s but run at 2.5Gb/s with the default firmware.

7) Half an interface to a 5th SLB connector.

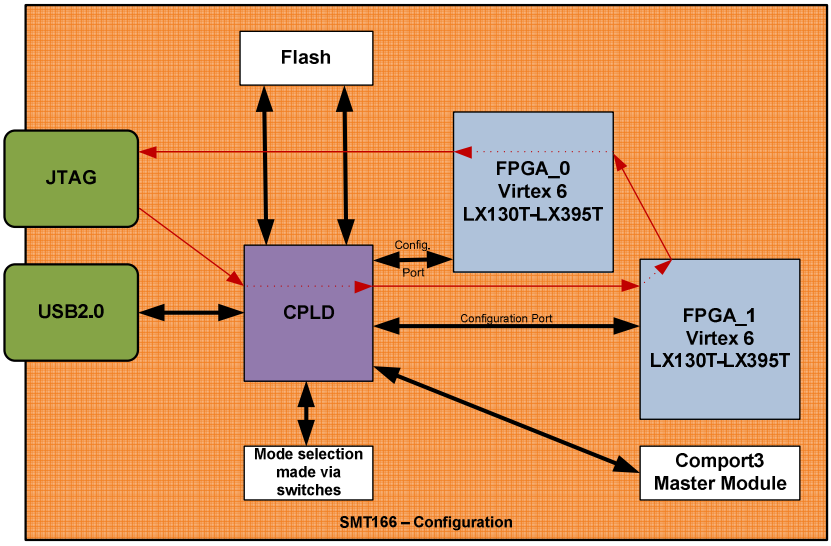
8) 4 LEDs.

FPGA Configuration

All three Xilinx devices (2 x Virtex 6 and a CPLD) are included in a single JTAG chain. The chain is defined as follows:

Device	ID	IR length
Virtex 6 LXT130 - FPGA_0	0x6424A093	10
Virtex 6 LXT130 - FPGA_1	0x6424A093	10
CPLD XC2C512	0x16D7E093	8

Diagrammatically this is shown below:



The CPLD contains a configuration that allows the two Virtex 6 FPGAs to be programmed/configured upon reset or power-up. The configuration data is stored in a flash memory device. The flash contents may be programmed using SMT6002.

The CPLD also contains a USB interface and a ComPort (8-bit bi-directional communication port) to a TIM site. The ComPort supports SMT6002 when programming the flash on the TIM, and also host connectivity from the TIM at data rates of up to 20MB/s.

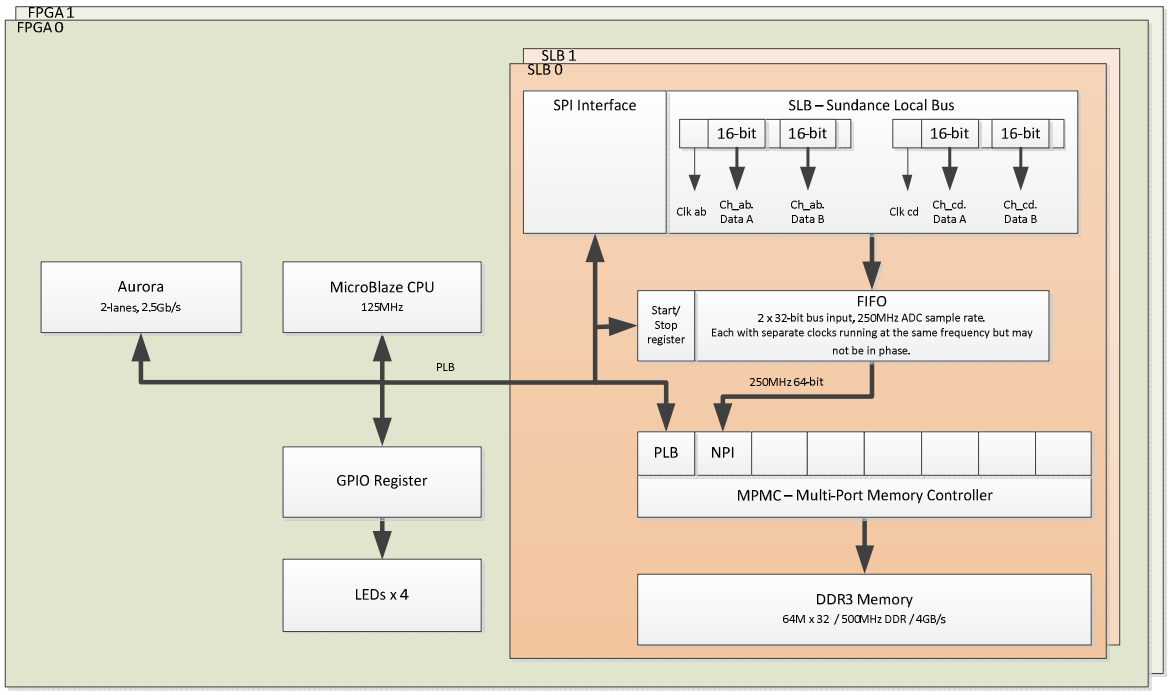
The various operational modes are selected using 2 x 4-way DIP switches as detailed here:

SW3				**SW2**				**Function**
1	2	3	4	1	2	3	4	
On	Off	Off	Off	On	On	Off	Off	SMT166 flash programming
Off	Off	Off	Off	On	On	Off	Off	TIM flash programming
				On	On			FPGA2000 configure from flash
				Off	Off			FPGA2000 no flash configuration
						On	On	FPGA1000 configure from flash
						Off	Off	FPGA1000 no flash configuration
On								USB to flash
Off								USB to ComPort

Notes:
FPGA1000 is also called FPGA_0.
FPGA2000 is also called FPGA_1.

FPGA Detailed Block Diagram

The Virtex 6 FPGA contains the following configuration shown below:



The FPGA is controlled using a Microblaze (MB) soft processor core. This runs at 125MHz, and has a number of peripherals attached.

The FPGA design was implemented using Xilinx EDK/SDK 13.4.

All of the custom peripherals were designed using ISE 13.4, and include:

Peripheral	Name	Description
Clock generator	clock_generator_v4_03_a	Generate clocks for MB etc.
Aurora	aurora_v1_00_a	RSL (GTX/MGT) serial interface to TIM.
SLB	slb_fifo_v1_00_a	SLB interface supporting 4 channels of 16 bits. Channels are in pairs with data and clock.
SPI	slb_spi_v1_00_a	Serial configuration interface for ADC module.
DDR3	mpmc_v6_05_a	Multi-ported memory interface.
SLB to MPMC	my_npi_v1_00_a	Fast DMA-type interface between SLB and DDR3.
LEDs	xps_gpio_v2_00_a	5-bit GPIO register to control LEDs and return the FPGA number.

All peripherals connect (where appropriate) to the PLB bus. The main VHDL code is called user_logic.vhd within the respective design folder.

FPGA Peripherals

Clock Generator

The clock generator has a 125MHz differential clock input from pins L23/M22 and outputs 6 clocks to the system as described below (clock_generator_A):

Clock	Frequency MHz	Description
CLKOUT0	125	MicroBlaze and PLB.
CLKOUT1	250	MPMC I/O clock.
CLKOUT2	500	DDR3 clock.
CLKOUT3	500	DDR3 clock.
CLKOUT4	200	DDR3 ref clock.
CLKOUT5	200	RSL initialisation clock.

Note: Clock_generator_B is used for MPMC_B and associated DDR3.

Aurora

The Aurora core is created using Xilinx CoreGen. This particular implementation has a 32-bit interface, 1-lane, and is set for streaming mode. The lanes used are from MGT112 with the REFCLK on pins AH6/AH5.

The entity contains the following ports:

Port	Description
TXP/TXN	Differential transmit pair from one lane of the quad-tile.
RXP/RXN	Differential receive pair to one lane of the quad-tile.
GTXQ0_P/N	Reference clock input to the quad-tile. These ports are in the top level (not user_logic).
aurora_clk_out	This is the single-ended refclk output from an IBUFDS_GTXE1. This is in the top level and is intended to drive the user_logic aurora_clk_in (to multiple lanes).
aurora_clk_in	Reference clock input. 125MHz.
init_clk_i	A 200MHz clock from the system clock generator, CLKOUT5.
LEDStat	4 bits used for debugging to drive LEDs.
probe_clk, probes	Interface to Chipscope used for debugging.

Two Aurora cores are implemented within this design. The MB addresses are shown here:

Address (hex)	Resource	
8740 0000	Aurora 0 CTRL	Use this lane on FPGA0 to connect to the TIM.
8840 0000	Aurora 0 FIFO	
8940 0000	Aurora 1 CTRL	Use this lane on FPGA1 to connect to the TIM.
8A40 0000	Aurora 1 FIFO	

The low-level aurora_8b10b_v5_3 core has a 32-bit TXD input coupled with a simple handshaking interface using TX_SRC_RDY_N and TX_DST_RDY_N. The 32-bit RXD output is coupled with a simple data ready signal RX_SRC_RDY_N. See the Xilinx user guide ug353 for full details http://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b_ug353.pdf.

The interface between the MB and the Aurora core is via a 1k x 32 FIFO. To transmit from the core, the MB writes to the FIFO then issues a send command. When the packet has been sent, the FIFO will become empty.

NOTE: This operation will be updated to include a DMA-type functionality.

The Aurora status and control registers are described here:

Aurora Status Register

Bit - little endian (31-0)	SDK bit (0-31)	Function
31-8	0 - 23	0 to 23 - not used
7	24	tx_full
6	25	tx_empty
5	26	rx_full
4	27	rx_empty
3	28	channel_up
2	29	lane_up
1	30	hard_error
0	31	soft_error

Aurora Control Register

Bit	Function
23-21	Loopback mode
20	State machine reset
19	Aurora GT reset
18	Aurora reset
17	RX FIFO reset
16	TX FIFO reset

The MB typically polls the status bits which show the FIFO flags. No communication is possible if the bits channel_up or lane_up are not high. Hard and soft errors should not occur. Do not over-fill the FIFO.

The lane initialisation is normally performed by writing 0x001F0000 followed by 0x00030000, and then 0. This will reset the Aurora core and the controlling state machine while keeping the FIFOs in reset to ensure no erroneous data reception.

DDR3 and MPMC and NPI

The DDR3 interface was implemented using the included IP from within Xilinx XPS. This was initially created using the BSB (base system builder) wizard. The second DDR3 interface has to be added manually as BSB does not support multiple DDR3 interfaces. Each DDR3 requires a separate clock generator.

MPMC means Multi-Port Memory Controller. This is an easy, efficient, and expandable mechanism to interface to the DDR3. The physical DDR3 interface is the Xilinx MIG which is called from the EDK tools.

The MPMC has 8 ports. This design currently uses 2 ports. Port0 is connected directly to the PLB and hence allows MB access. Port1 is connected to the NPI peripheral which is used to stream data from the SLB directly into DDR3 without MB intervention.

The MB memory addresses for the two DDR3 banks are given here:

Address (hex)	Resource	Comment
9000 0000	DDR3 A	
A000 0000	DDR3 B	

Note: The same addresses are used in both FPGAs.

The NPI protocol (Native Port Interface) is fully described in the MPMC datasheet from Xilinx http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf

In the SMT166 implementation of the NPI, the data bus is 64 bits wide, and we write 32 word bursts for maximum performance. The protocol follows essentially the following sequence and is implemented using a simple state machine:

> A WrFIFO_PUSH signal is asserted concurrently with the data that is to be written. The write data is first read from the SLB FIFO. A data word will be sent to the MPMC on every rising clock edge (250MHz) where WrFIFO_PUSH is asserted. When 32 words have been transferred then the AddrReq signal is asserted for one clock cycle.
>
> The data writing is continuous as long as there is data in the SLB FIFO to read. The SLB FIFO input will be running at a maximum speed of 245.76MHz. This is the local clock rate of the SMT941.
>
> The NPI interface runs at 250MHz with 64-bit data. This is translated to the physical DDR3 interface which runs at 500MHz DDR and 32-bit data. The DDR3 rate is equivalent to 500MHz and 64-bit data when taking the dual data rate into account. Hence the DDR3 rate far exceeds what is needed to store the ADC data at full (245.76MHz) speed.

The NPI implementation has a single control input rst_in, and a single output status stopped. The rst_in signal is controlled by an output port from the SLB interface (See later). The returned status signal is also made available in a register within the SLB interface. The stopped signal becomes asserted when the whole of the DDR3 has been written to at the end of an ADC data acquisition.

SLB

The SLB interface is implemented using the slb_fifo peripheral. The entity has the following ports:

Port	Description
adcab_da/db	14-bit inputs from the SMT941 ADC. Two channels a and b.
adcab_clkoutp/n	Differential clock for channels a and b.
adccd_da/db	14-bit inputs from the SMT941 ADC. Two channels c and d.
adccd_clkoutp/n	Differential clock for channels c and d.
fifo_dout	64-bit FIFO data output bus.
fifo_clk	FIFO read clock.
fifo_rden	FIFO read data enable.
fifo_empty	FIFO empty flag.
rst_out	Control signal to the NPI.
stopped	Status signal from the NPI.
debug_clk/data	Interface to Chipscope used for debugging.

In essence this core consists of data capture logic and two 1k x 32 FIFOs. Each of the two FIFOs receives data from the capture logic associated with a channel pair. The ADC clock is common to both channels in a pair and is used for the write clock of the FIFO. The two 32-bit wide FIFOs are read simultaneously by the NPI logic using a single read clock. The FIFO empty flag is a logic AND of the two empty flags from each 32-bit FIFO. This means that data can only be transferred if both channel pairs are running, and they must be at the same frequency (which is inherent in the SMT941 design).

Data from the ADC is sent using 7 differential pairs and DDR signalling. Each pair represents two bits of data; the LSB is sent on the rising edge, and the MSB on the falling edge. The ADC data capture logic first takes each pair and creates a DDR single-ended data bit using an IBUFDS. This single-ended bit is then sent to an IDDR to create the even and odd sample bits (at SDR). In this way, the 250MHz differential data (using 7 pairs) is converted into a 250MHz 14-bit data bus. This data bus is input to the capture FIFO.

The FIFO_CLK signal is input from the NPI (MPMC) core together with a synchronous FIFO_RDEN (read enable) signal. 64-bit data is returned on FIFO_DOUT.

The rst_out signal is inverted from the state of the control register bit (see below). When low this will hold the NPI in reset and not capture ADC data. When high, the NPI is released from reset and data capture starts.

The stopped signal is returned from the NPI and is made available in the status register (see below).

Address (hex)	Resource	
8540 0000	SLB_FIFO_0 channels a&b	W:bit 0 - 1 to enable capture. R:bit 16 - 1=capture complete.
8640 0000	SLB_FIFO_1 channels c&d	W:bit 0 - 1 to enable capture. R:bit 16 - 1=capture complete.

SPI

The SLB modules are typically configured (as is the case with the SMT941) using a serial bus comprising of clock, data, and latch enable.

The SMT941 has separate busses for each of its devices; ADC1, ADC2, and clock synthesizer.

Several other control signals are required to control the module's operation.

Sundance supply an SDK C source function that initialises the ADC. Two parameters are passed to this function, the SLB module (0 or 1), and the required sample frequency (125 or 250). This function is shown in the appendix.

The MB addresses are shown in the table below:

Address (hex)	Resource	
8240 0000	SLB_SPI_0	Offset 4: SPI_Data Offset 8: SPI_Control / Status
8340 0000	SLB_SPI_1	

The status / control register bit functions are shown in the table below:

SPI Control / Status Register

Bit	Function
11	ChAB_reset & ChCD_reset
10	ChCD_ctrl2
9	ChCD_ctrl1
8	ChAB_ctrl2
7	ChAB_ctrl1
6	Clk_Pwrdwn
5	Clk_nReset
4	
3	Busy
2	SPI_Sel bit 1
1	SPI_Sel bit 0
0	

GPIO

A 5-bit MB peripheral is included to control the state of 4 LEDs (per FPGA) and to return the FPGA device number.

The FPGA device number is currently implemented using a custom SLB plug on the spare (SLB5) interface. Most FPGA pins are connected equally on the two FPGAs with the exception of the 5th SLB interface which is shared. The only mechanism to allow the FPGAs to decide whether it is 0 or 1 is to use this interface. The custom plug simply grounds one of these inputs. The corresponding pin on the other FPGA is internally pulled high. So each FPGA can read this pin and make it available in the GPIO register for the MB.

This mechanism is needed if the bitstreams (FPGA configurations) are to be identical. As there is a PCB routing error (the RSL lanes are not connected symmetrically), the MB needs to know which Aurora lane to use to connect to the DSP module in the TIM site.

Address (hex)	Resource	
8140 0000	GPIO	Bits 4:1 LEDs. Bit 0: FPGA device number.

Really useful hints/tips:

Configure FPGA using impact as this allows the targeting of either device. There does not seem to be a simple way of configuring either FPGA from with SDK.

In the XMD% console type:

```
connect mb mdm –debugdevice deviceNr 2
```

this will target the second FPGA. When debug is launched you will get a message (I think) saying that there is already a connection - so I guess it is attempting to make a connection to maybe the 1st device. You'll also get a message saying that the FPGA is not programmed.

When changing debug devices type

```
disconnect 0
```

In SDK it you can also configure the JTAG chain manually using:
Virtex6: ID Code is 0x6424A093 and IR Length is 10, and
CoolRunner: ID Code is 0x16D7E093 and IR Length is 8.

When using SDK to display printf (or xil_printf) and this output is required to be in the SDK console, type the following two commands in the XMD console:

connect mdm -uart
terminal -jtag_uart_server

Also note that the debug configurations STDIO connection tab should have the check box ticked to allow STDIO to connect to console using port:JTAG UART.

Appendix

SPI Initialisation

```
void setup_941(int n, int freq)
{
	unsigned int *SPI_data;
	unsigned int *SPI_control;
	unsigned int rfreq;

	if(n==0) {
		SPI_data	= 0x82400004;
		SPI_control	= 0x82400008;
	}
	else {
		SPI_data	= 0x83400004;
		SPI_control	= 0x83400008;
	}

	if(freq==125)
		rfreq=0x00800000;
	else
		rfreq=0x00400000;

// Set nreset and npwrdwn high (inactive)
	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn;
	*SPI_control =                  slb_clk_npwrdwn;                    // pulse clock reset low
	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn | slb_chab_reset;   // pulse adc reset high
	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn | spi_rst;          // pulse spi reset high
	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn;

// Setup clock chip. Use "CDCE72010_Control_GUI.exe" to calculate register contents.
	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn | spi_sel_clk;      // select clk_pll for spi
                                                                        // transfers.
	*SPI_data=0x683C0240;  while((*SPI_control)&8);             // 0 - o/p 0 - nc      0x683C0200
	*SPI_data=0x82000001 | rfreq; while((*SPI_control)&8);      // 1 - o/p 1 - chcd clk1      0x80400001
	*SPI_data=0x83000002 | rfreq; while((*SPI_control)&8);      // 2 - o/p 2 - chcd clk2      0x83400002
	*SPI_data=0x82000003 | rfreq; while((*SPI_control)&8);      // 3 - o/p 3 - chab clk1 [8140* = low-
                                                                // swing, 8340* = high-swing] 0x68400003
	*SPI_data=0x83000004 | rfreq; while((*SPI_control)&8);      // 4 - o/p 4 - chab clk2      0x83400004
	*SPI_data=0x68800005;         while((*SPI_control)&8);      // 5 - o/p 5 - nc             0x68000005
	*SPI_data=0x68800006;         while((*SPI_control)&8);      // 6 - o/p 6 - nc             0x68000006
	*SPI_data=0x83000127 | rfreq; while((*SPI_control)&8);      // 7 - o/p 7 - ext clk out    0x83400127
	*SPI_data=0x68800178;         while((*SPI_control)&8);      // 8 - o/p 8 - nc             0x68000178
	*SPI_data=0x68050049;         while((*SPI_control)&8);      // 9 - o/p 9 - ext clk in     0x68050049
	*SPI_data=0x0BFC07CA;         while((*SPI_control)&8);      // A - M & N dividers         0x0BFC07CA
	*SPI_data=0x9000041B;         while((*SPI_control)&8);      // B - Control                0x9000041B
                                                                // (0x92..=ext clk)
	*SPI_data=0x0000180C;         while((*SPI_control)&8);      // C - Status & diagnostic    0x0000180C

// ADC SPI data needs to be bit reversed as firmware only shifts in one direction
	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn | spi_sel_adc;// select adc for spi transfers
	*SPI_data=0x0004;      while((*SPI_control)&8);      //
	*SPI_data=0x00FC;      while((*SPI_control)&8);      //
	*SPI_data=0x1002;      while((*SPI_control)&8);      // Normal operation
	*SPI_data=0x0182;      while((*SPI_control)&8);      // DDR output
	*SPI_data=0x2D22;      while((*SPI_control)&8);      // clock edge control. was 3F22
	*SPI_data=0x200A;      while((*SPI_control)&8);      // 2s comp 200A=single, 220A=independant
control
	*SPI_data=0x00CA;      while((*SPI_control)&8);      // offset corr.
	*SPI_data=0x00AA;      while((*SPI_control)&8);      // gain = 0dB
	*SPI_data=0x00EA;      while((*SPI_control)&8);      // fine gain
	*SPI_data=0x2046;      while((*SPI_control)&8);      // Use 0x2046 for ramp test pattern ch.A /
Normal 0046 / Toggle C046
	*SPI_data=0x00C6;      while((*SPI_control)&8);      // pedestal
	*SPI_data=0x20AE;      while((*SPI_control)&8);      // Use 0x20AE for ramp test pattern ch.B /
Normal 00AE / Toggle C0AE

	*SPI_control = slb_clk_nreset | slb_clk_npwrdwn | spi_sel_dac;      // select dac for spi transfers
                                                                        // (on a 941 dac is 2nd adc)
	*SPI_data=0x0004;      while((*SPI_control)&8);                     //
```

```
        *SPI_data=0x00FC;      while((*SPI_control)&8);      //
        *SPI_data=0x1002;      while((*SPI_control)&8);      // Normal operation
        *SPI_data=0x0182;      while((*SPI_control)&8);      // DDR output
        *SPI_data=0x2D22;      while((*SPI_control)&8);      // clock edge control
        *SPI_data=0x200A;      while((*SPI_control)&8);      // 2s comp
        *SPI_data=0x00CA;      while((*SPI_control)&8);      // offset corr.
        *SPI_data=0x00AA;      while((*SPI_control)&8);      // gain = 0dB
        *SPI_data=0x00EA;      while((*SPI_control)&8);      // fine gain
        *SPI_data=0x2046;      while((*SPI_control)&8);      // Use 0x2046 for ramp test pattern ch.A
        *SPI_data=0x00C6;      while((*SPI_control)&8);      // pedestal
        *SPI_data=0x20AE;      while((*SPI_control)&8);      // Use 0x20AE for ramp test pattern ch.B
}
```