# Saturn Project Notes

This document describes the FPGA for a new radio project using a Xilinx FPGA and an attached processing module. In the first instance this will be a Raspberry Pi 4 compute module but could be another embedded processor board. The interface to the FPGA will be PCI Express; the protocol code for data transfer to the “Thetis” PC will execute on the processor, not in the FPGA. The radio could be implemented on an oversize M.2 board, to fit onto several embedded processors.

This could be used in several ways:

|  |  |
| --- | --- |
| Diagram  Description automatically generated | The processor module is used simply to move data using protocol 1 – like the Red Pitaya’s processor. Low demand on the processor. Wired ethernet connection to PC running Thetis or other app. |
| Diagram  Description automatically generated | The processor module is used simply to move data using protocol 2. This will have higher demand because the data rate is higher. Wired ethernet connection to PC running Thetis or other app. |
| Diagram  Description automatically generated | The processor executes an SDR app such as Pihpsdr or linhpsdr. No PC required, and high quality display outputs are available. |
| Diagram  Description automatically generated | The processor execute Pihpsdr and has an attached 7” RPi touchscreen display. Possibly single ADC or 14 bit ADC version, with Apollo-like RF module. This could be a lower cost small radio, but there may be no market for it now. |

## FPGA Block Diagram

Diagram

Description automatically generated

Figure : Overall block Diagram

# Interfaces

## FPGA

|  |  |
| --- | --- |
| ADC | 2x LTC2208. Each with 5 bit attenuator. |
| DAC | The “normal” TX DAC. With 6 bit attenuator and 8 bit analogue PWM drive level. |
| Envelope DAC | The current design allows for a PWM DAC like Hermes used. An SPI DAC would give faster update. |
| Codec | TLV320AIC23B, as used on Hermes etc  Its I2S port driven by FPGA. the I2C port interfaced through the FPGA but with data transactions initiated by the host processor. |
| Audio interfaces | Orion-like software settable connections to 3.5mm jack  Separate audio out for each of speaker and headphone, with separate gains  Balanced XLR mic input for “pro” audio people |
| RF | SPI-like signals for Alex header. Follows the ANAN7000 standard, 16 bit TX data, 32 bit RX data. An SPI ADC (78H90) is used for the analogue level measurement. |
| Clock generation | 122.88MHz VCXO, with C/R filter on control voltage feed.  10MHz reference XTAL oscillator  10MHz external reference in, with auto select |
| PTT, keyer | 3 strobe inputs for PTT and dot/dash input. Buffered, and active low. No dash/dot keying yet added; it just implements “key down” |
| Config PROM | 256Mbit QSPI. Must be programmable through the FPGA with no jumper changes. Able to hold processor in reset until FPGA has configured if needed. |
| JTAG | Standard Xilinx JTAG connector for debug |

## Processor (Raspberry Pi 4 Compute Module)

|  |  |
| --- | --- |
| PCIe | PCI express connection to FPGA. Gen2 x1. Provide additional PCI signals for X4; just not used with this module.  The design aim is for the connection to the radio to be entirely via PCIe; no other signals. This to make sure we can use other processor modules too. |
| USB | Tracked out to rear panel I/O connectors. |
| HDMI | Rear panel |
| Ethernet | Tracked to rear panel |
| Wi-Fi | Wi-Fi antenna connectors to connect to rear panel. |
| Power, reset | We need to investigate the power sequence / reset arrangements – if we can hold the processor in reset while the FPGA configures for perhaps 150ms, we can use a cheaper QSPI config prom. |

## RF Module Interfaces

### Alex header

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin** | **Function** | **Orion mk2: J15** | **7000 RF board: J7** |
| 1 | ANT\_TUNE | Buffered input to FPGA | (grounded) |
| 2 | +12V ALEX | +12V | Not connected |
| 3 | ALEX\_SPI\_SDO | SPI verilog | SPI registers |
| 4 | ALEX\_SPI\_SCK | SPI verilog | SPI registers |
| 5 | ALEX\_RX\_LOAD | SPI verilog | SPI registers |
| 6 | ALEX\_TX\_LOAD | SPI verilog | SPI registers |
| 7 | FWD\_POWER | Analogue in | Directional coupler FWD |
| 8 | PTT | Buffered input to FPGA | Not connected |
| 9 | REV\_POWER | Analogue in | Directional coupler REV |
| 10 | GND | GND | GND |

ANT\_TUNE and PTT appear to have no relevance; may be historical?

Note this is the only connection to the RF board. The RF board also has J22, for possible connection to an Arduino-like modules for LCD display.

# Power Supply, PCB Issues

This section covers issues related to the board & interfaces to the FPGA. Much of this is as per Orion mk2, with noted differences.

## Main interfaces- Changes from Orion mk2

This section notes deltas from the Orion mk2 design:

**DAC:** The DAC is driven by the FPGA which has differential LVDS outputs: the driver is not needed. This will be essentially as per Orion.

**ADCs:** The ADCs data to the FPGA is differential LVDS. There are 18 pairs per ADC (16 data + overflow + o/p clock). The shutdown pin SHDN is hardwired rather than FPGA driven: wire to 0v. Mode=2/3 VCC (2’s complement data, clock adjustment enabled). The LVDS pin connects to 3.3v to select the signal level. Other than that its schematic should be largely as per Orion. I assume there should be 100R terminating resistors at the FPGA between the +and- differential inputs.

**RF attenuators:** ADC attenuators “as is”. The DAC attenuator is an IDT F1912. Orion has pins to drive it in serial or parallel modes. Saturn will use “Direct Parallel” mode only. Dac\_Atten\_Mode = 0 in FPGA; Dac\_Atten\_LE=1

**7000DLE board interface:** The SPI data interface for RX and TX is from the FPGA. The analogue inputs (eg fwd, rev power) from the RF board go to an ADC connected via an SPI port.

**Codec & audio:** The CODEC SPI configuration interface needs to be driven by the FPGA. The discrete logic signals to control the audio path (eg tip/ring/bias etc) are on GPIO[4:0] pins. There is a new signal GPIO5, to select a differential XLR mic input amplifier. 2 spare signals GPIO6&7 if needed.

## Floorplan

The current selected FPGA is Xilinx **XC7A200TFBG676-2**. It may be able to downsize to **XC7A100TFGG676-2** which is footprint compatible. I have allocated pins to the FPGA one the assumption that this approximate layout of main interfaces around the chip will be OK. IO pins are assigned to the FPGA banks to accommodate this. There are four sets of constraints:

1. The PCI Express data transceivers are fixed in bank 216 and can’t be changed;
2. The DAC LVDS output levels mean we can’t have CMOS 3.3V outputs in the same bank. (The DAC attenuator will be driven with lower voltage CMOS for that reason);
3. To enable moving down to the XC7A100T device, do not use banks 12 & 33;
4. Analogue inputs are in bank 35.

Chart, funnel chart

Description automatically generated

## Power Supplies

FPGA I/O levels are set “per bank”.

|  |  |  |
| --- | --- | --- |
| **Bank** | **Purpose** | **I/O Voltage** |
| Bank 0 | Configuration | 3.3V |
| Bank 13 | CODEC | 3.3V |
| Bank 14 | Config + ADC2 connections | 3.3V |
| Bank 15 | ADC1 connections | 3.3V |
| Bank 16 | DAC connections | 2.5V |
| Bank 34 | General logic | 3.3V |
| Bank 35 | General logic; analogue | 3.3V |
| Banks 12, 33 | Not used | 3.3V (I assume it still needs power) |
| GTP Quad 216 | GTP buffers (PCIe) | See below |

The analogue inputs (XADC) connect to pins in bank 35. These are no longer being used for external functions, but power supply voltage and die temp can still be monitored.

Xilinx 7 series devices have power supply sequencing requirements. Supplies should come up in this order, & be shut down in reverse order:

|  |  |  |
| --- | --- | --- |
| 1 | VCCINT | 1V |
| 1 | VCCBRAM | 1V |
| 2 | VMGTAVCC[[1]](#footnote-1) | 1V (separate regulator, <10mV p-p noise) |
| 3 | VMGTAVTT[[2]](#footnote-2) | 1.2V (separate regulator, <10mV p-p noise) |
| 3 | VCCAUX | 1.8V |
| 3 | VCCADC | 1.8V |
| 4 | VCCO | Set per bank; 2.5V and 3.3V both needed (see above) |

The on-chip VCCADC supply is derived from VCCAUX: same voltage, but via a ferrite bead. See Xilinx UG480 figure 1-2 (p15). VREFP, VREFN, GNDADC are all at ground potential but with a ferrite bead to ground. 100nF and 470nF decoupling capacitors across VCCADC and GNDADC.

Diagram

Description automatically generated

## Decoupling

UG483 recommends a set of capacitors for each supply. These are package dependent. Assume that we will use XC7A200TFBG676.

|  |  |  |
| --- | --- | --- |
| **Pin** | **Voltage** | **Decoupling** |
| VCCINT | 1.0V | 1x 680uF; 1x100uF; 12x 4.7uF;17x 0.47uF |
| VCCBRAM | 1.0V |
| VCCAUX | 1.8V | 1x 47uF; 4x 4.7uF; 7x 0.47uF |
| VCCO\_0 | 3.3V | 1x 47uF |
| VCCO\_13 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_14 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_15 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_16 | 2.5V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_34 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_35 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_12 (unused | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_32 (unused) | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| MGTAVCC | 1.0V | 4.7uF ceramic; 2x 0.1uF |
| MGTAVTT | 1.2V | 4.7uF ceramic; 2x 0.1uF |
| MGTRREF\_216 | Reference input. Connect via 100R resistor to MGTAVTT | 4.7uF ceramic. |

Xilinx UG483 table 2-5 recommends suitable capacitors:

|  |  |
| --- | --- |
| 680uF Tantalum D package | T530X687M006ATE018 |
| 100uF tantalum or ceramic X7R 1210 | GRM32ER60J107ME20L |
| 47uF ceramic X7R 1210 | GRM32ER70J476ME20L |
| 4.7uF ceramic X7R 0805 | GRM21BR71A475KA73 |
| 0.47uF ceramic X7R 0603 | GRM188R70J474KA01 |

For placement of the MGTAVCC/MGTAVTT capacitors see Xilinx UG482 p230 (placement within the BGA pads is recommended)

## Pinouts

There is a separate spreadsheet “iopins.csv” created by the Xilinx tools with FPGA pinouts. This is a cross reference to that describing what the signals are for.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin(s)** | **Group/Interface** | **Dir’n** | **Comment** |
| BCLK | CODEC | Out | Bit clock to CODEC |
| LRCLK | CODEC | Out | L/R clock to Codec |
| MCLK | CODEC | Out | 12.288MHz master clock output to codec |
| i2stxd | CODEC | Out | Audio TX data to codec |
| i2srxd | CODEC | In | Audio RX data from codec |
| CODEC\_SPI\_CLK | CODEC | Out | SPI signal to codec |
| CODEC\_SPI\_DATA | CODEC | Out | SPI signal to codec |
| CODEC\_CS | CODEC | Out | SPI chip select to codec |
| pcie\_reset\_n | PCI Express | In | Reset in from processor |
| pcie\_diff\_clock\_rtl\_clk\_n[0]  pcie\_diff\_clock\_rtl\_clk\_p[0] | PCI Express | In | PCIe reference clock. differential |
| pcie\_7x\_mgt\_rtl\_0\_rxn[3:0]  pcie\_7x\_mgt\_rtl\_0\_rxp[3:0] | PCI Express | In | PCIe RX data pairs |
| pcie\_7x\_mgt\_rtl\_0\_txn[3:0]  pcie\_7x\_mgt\_rtl\_0\_txp[3:0] | PCI Express | Out | PCIe TX data pairs |
| PCI\_LINK\_LED | PCI Express | Out | 0 (LED lit) when PCI link established. See 8.5 |
| PCIe\_CLK\_REQn | PCI Express | Out | Clock request output. Driven to 0V by configured FPGA. Needs pullup resistor. |
| MGTRREF\_216 | PCI Express | in | PCIe buffer reference. Needs 100R to MGTAVTT supply (see also section 3.4 & 3.8) |
| ADC1Ovr\_in\_N  ADC1Ovr\_in\_P | ADC | In | RF ADC1 overscale. Differential LVDS |
| ADC1\_In\_N[15:0]  ADC1\_In\_P[15:0] | ADC | In | RF ADC1 16 bit input data. Differential LVDS |
| ADC1\_CLKin\_N  ADC1\_CLKin\_P | ADC | In | RF ADC1 clock from ADC to FPGA  (not used but tracked; differential) |
| ADC2Ovr\_in\_N  ADC2Ovr\_in\_P | ADC | In | RF ADC2 overscale. Differential LVDS |
| ADC2\_In\_N[15:0]  ADC2\_In\_P[15:0] | ADC | In | RF ADC2 16 bit input data. Differential LVDS |
| ADC2\_CLKin\_N  ADC2\_CLKin\_P | ADC | In | RF ADC2 clock from ADC to FPGA  (not used but tracked; differential) |
| DAC\_Out\_N[15:0]  DAC\_Out\_P[15:0] | DAC | Out | RF DAC 16 bit output data. Differential LVDS |
| clock\_122\_in\_n  clock\_122\_in\_p | Sample Clock | In | 122.88MHz VCXO clock in; PECL with level shift |
| pll\_cr | Sample Clock | Out | PLL output to C-R circuit for VCXO control input |
| ref\_in\_10 | Sample Clock | In | 10MHz reference input from ext/oscillator select |
| TX\_DAC\_PWM | TX DAC related | Out | PWM o/p for DAC bias voltage (like DAC\_ALC) |
| Dac\_Atten[5:0]  Dac\_Atten\_LE  Dac\_Atten\_MODE  Dac\_Atten\_CLK  Dac\_Atten\_DATA | DAC attenuator | Out | TX DAC attenuator parallel data (see 3.1)  (1.8V logic signals)  (the serial wires are not used)  LE: drive to 1  MODE, CLK, DATA: drive to 0 |
| ADC1\_ATTEN\_CLK | ADC attenuator | Out | ADC1 atten control |
| ADC1\_ATTEN\_DAT | ADC attenuator | Out | ADC1 atten control |
| ADC1\_ATTEN\_LE | ADC attenuator | Out | ADC1 atten control |
| ADC2\_ATTEN\_CLK | ADC attenuator | Out | ADC2 atten control |
| ADC2\_ATTEN\_DAT | ADC attenuator | Out | ADC2 atten control |
| ADC2\_ATTEN\_LE | ADC attenuator | Out | ADC2 atten control |
| RF\_SPI\_CK | RF interface | Out | RF data control to Anan 7000 |
| RF\_SPI\_DATA | RF interface | Out | RF data control to Anan 7000 |
| RF\_SPI\_TX\_LOAD | RF interface | Out | RF data control to Anan 7000 |
| RF\_SPI\_RX\_LOAD | RF interface | Out | RF data control to Anan 7000 |
| Buf\_Alex\_Pin1 | RF interface | In | Unused Input |
| Buf\_Alex\_Pin8 | RF interface | In | Unused input |
| (see also configuration wiring diagram section 3.7) | FPGA Configuration |  |  |
| GPIO\_OUT[23:0] | Various | Out | Parallel data; see also section 7.4 |
| GPIO\_OUT[0] | Audio | Out | Mic bias enable. =1 to provide electret bias on 3.5mm jack |
| GPIO\_OUT[1] | Audio | Out | Input\_PTT\_Select  0=PTT on ring; 1=PTT on tip |
| GPIO\_OUT[2] | Audio | Out | Mic\_Signal\_Select  0=mic on ring, 1 = mic on tip |
| GPIO\_OUT[3] | Audio | Out | Mic\_Bias\_Select  0=bias on ring; 1= bias on tip |
| GPIO\_OUT[4] | Audio | Out | Spkr\_amp\_Mute |
| GPIO\_OUT[5] | Audio | Out | Balanced\_Mic\_Select  =1 to enable balanced mic input |
| GPIO\_OUT[7:6] | Audio | Out | Spare, uncommitted output |
| GPIO\_OUT[8] | RF ADC | Out | ADC1 RAND =1 to randomise data |
| GPIO\_OUT[9] | RF ADC | Out | ADC1 PGA =1 to enable ADC 3dB amplifier |
| GPIO\_OUT[10] | RF ADC | Out | ADC1 DITHER =1 to dither the clock |
| GPIO\_OUT[11] | RF ADC | Out | ADC2 RAND =1 to randomise data |
| GPIO\_OUT[12] | RF ADC | Out | ADC2 PGA =1 to enable ADC 3dB amplifier |
| GPIO\_OUT[13] | RF ADC | Out | ADC2 DITHER =1 to dither the clock |
| GPIO\_OUT[15:14] | RF | Out | Spare, uncommitted output |
| GPIO\_OUT[22:16] | General | Out | Open collector outputs (7 bits) |
| GPIO\_OUT[23] | General | Out | Spare, uncommitted output |
| DRIVER\_PA\_EN | TX Strobes | Out | Enables power to 0.5W amp |
| MOX\_strobe | TX Strobes | Out | 1=TX. See Orion FPGA\_PTT signal |
| TXRX\_RELAY | TX Strobes | Out | 0 if TX. Drive LED, lit for TX |
| CTRL\_TRSW | TX Strobes | Out | Drives relay by 0.5W amp |
| BUFF\_OUT | Strobe | Out | Unused. Wired to 0v |
| ATU\_TUNE | General CMOS | Out | =1 to initiate TUNE by external ATU |
| STATUS\_IN[9:0] |  | In | Parallel data; see section 7.5 |
| STATUS\_IN[0] | General CMOS | In | PTT In (3.5mm jack & rear panel combined)  1= PTT pressed |
| STATUS\_IN[1] | General CMOS | In | Not used |
| STATUS\_IN[2] | General CMOS | In | Key in 1 (Dot) 1= pressed |
| STATUS\_IN[3] | General CMOS | In | Key in 2 (Dash) 1= pressed |
| STATUS\_IN[7:4] | General CMOS | In | User IO4,5,6,8 as drawn  IO5 used as a TX inhibit input  IO8 used as a CW input  (these are NOT inverted by hardware, and have pullups) |
| STATUS\_IN[8] | General CMOS | In | 13.8v detect in. 1= power valid |
| STATUS\_IN[9] | General CMOS | In | ATU tune complete. 1= complete (needs pullup) |
| STATUS\_IN[10] | General CMOS | In | 1=10MHz/122.88MHz PLL locked |
| STATUS\_IN[31] | General CMOS | In | TX enable. 1 if TX allowed. Ext input J54. |
| TX\_ENABLE | General CMOS | In | External input; if 0, TX is gated off. Needs pullup. |
| LEDOutputs[15:0] | Debug | Out | Drivers for status / debug LED. Active high, See section 8.1 |
| BLINK\_LED | Debug | Out | LED blinking at about 1Hz rate. See 8.1 |
| EMC\_CLK | Config | In | 122.88MHz CMOS clock in |
| PROM\_SPI\_ss\_io[0]  PROM\_SPI\_io3\_io  PROM\_SPI\_io2\_io  PROM\_SPI\_io1\_io  PROM\_SPI\_io0\_io  VCCBATT\_0  DONE\_0  CCLK\_0  INIT\_B\_0  M0\_0, M1\_0, M2\_0  PROGRAM\_B\_0  CFGBVS\_0  PUDC\_B | Config |  | Configuration signals. See diagram in section 3.7 |
| TCK\_0  TMS\_0  TDI\_0  TDO\_0 | Config |  | JTAG signals. See diagram in section 3.7 |
| ADC\_MOSI  ADC\_MISO  ADC\_CLK  nADC\_CS | Aux ADC; connects an SPI A-D converter |  | Pins added, and custom Verilog IP core. Not fully tested! |
| FPGA\_CM4\_EN |  | Out | Not currently used. Powers on/off the compute module 4. Drive to 1 for normal operation. |
| PCIe-T-SMBCLK  PCIe-T-SMBDAT |  | bidir | Not currently used: PCIe bus config signals |
| EXTCLKDET | General CMOS | In | =0: internal 0MHz reference.  =1: external 10MHz reference |

A number of signals need pullup resistors (eg 4K7 to +3.3v):

STATUS\_IN[9]; TX\_ENABLE;

(not on the FPGA) RUN\_PG output to Raspberry pi4 CM

## Clocking

The clock arrangement should be as per Figure 2. The ADCs, DAC and FPGA should all get differential LVPECL clocks with trace lengths matched as far as possible to make the trace delays equal. The FPGA also requires a 122.88MHz clock to the EMCCLK pin (V22) used during configuration.



Figure : 122.88MHz Clock Distribution

Connect the FPGA LVPECL inputs with a similar resistor network to that on the ADCs. (The FPGA is set to LVDS levels – so we need PECL to LVDS translation).

ADC, DAC data timing at the FPGA can then be worked out (Figure 3):



Figure : Clock Timing For FPGA I/O

(In rev 1 Saturn, the ADC clocks are probably 0.7ns before the FPGA, DAC clocks)

DAC timing adjusted: a 135o clock phase gives a timing diagram something like:

Th 2.6ns min

Tsu 1.5ns max

## FPGA Configuration

The FPGA loads its configuration from an external memory device at startup. By far the best arrangement will be to use a QSPI Flash device, as long as it is fast enough. If necessary we can hold off the Raspberry pi starting up until FPGA configuration is complete. This will use Master SPI X4 Flash config mode (see Xilinx UG470).

Wire the JTAG and config PROM as per the Figure 4 & notes below:

Diagram, schematic

Description automatically generated

Figure : Schematic for Configuration Prom

|  |  |  |
| --- | --- | --- |
| **Pin** | **Package Pin** | **Comment** |
| D[00] | P22 | in-system programming of QSPI Prom. Signals PROM\_SPI\_MOSI, PROM\_SPI\_MISO, PROM\_SPI\_SSn[0] in the FPGA pin list connect to these 3 pins. |
| DIN/D[01] | R22 |
| FCS\_B | T19 |
| D[02] | P21 |  |
| D[03] | R21 |  |
| CCLK | L12 | Does not appear in the FPGA pin list |
| INIT\_B | U12 |  |
| DONE | G11 | Drives Raspberry Pi RUN\_PG via an open collector buffer and jumper |
| TDO | U13 | Use a 2mm pitch 2x7 pin header for Xilinx download cable. Xilinx suggested parts (DS593 page 15):  Molex 87832-1420 (SMT) or 87831-1420 (leaded)  FCI 98424-G52-14 (SMT) or 98414-G06-16 (leaded) |
| TDI | R13 |
| TMS | T13 |
| TCK | V12 |
| PROGRAM\_B | N12 |  |
| M0 | U11 |  |
| M1 | U10 |  |
| M2 | U9 |  |
| EMCCLK | V22 | 3.3V CMOS clock for configuration only |
| PUDC\_B | U22 |  |
| CFGBVS | U8 | Selects the correct configuration voltage. |
| VCCO\_0 | F12, T12 | 3.3V power rail |

Config LEDs lit after successful config

The S25FL256 devices are ready for read 300us after power applied. This is significantly faster than the FPGA TPOR (10-35ms) so the Flash memory will be ready when the FPGA begins.

The XC7A75T & XC7A100T both require approx. 30Mbit configuration data. The XC7A200T requires 75Mbit. Xilinx only supports commodity devices. The de-facto config prom is a QSPI device costing perhaps £2 and consuming only 8 pins. These read on quad output, fast read mode; not DDR. XAPP586 describes using a QSPI serial prom in master SPI mode with 7 series devices.

UG908 lists flash PROMs specifically supported by Xilinx. Cypress S25FL256SAGMFI003is suitable & available from Farnell. (Supported by Xilinx under manufacturer “Spansion”). These can clock at 133MHz but including setup time into FPGA, 80MHz is more realistic. Using 61.44MHz clock XC7A200T would configure in approx. 300ms in quad mode.

PCI Express has a commonly quoted “must be configured in 100ms” requirement. Actually it’s nearer 70ms:

* A PC PSU requirement is 100ms from power good until the power supply releases reset (we might have some flexibility here, as we aren’t using a PC power supply);
* PCIe says be ready for configuration 20ms after that;
* But Xilinx on-chip reset may take 50ms to release after it senses power OK.

QSPI unlikely to meet that requirement on its own BUT if we can hold off the CPU reset for a further 200ms say (Can we hold RUN\_PG low until ready – needs to be an open drain signal?), a cheap QSPI device would be fine. The on-chip RC oscillators have huge variation in clock rate caused by temp & process variation. We do have a good quality 122.88MHz VCXO available, which can be divided.

UG953 & XAPP1020 describes how to access the reserved configuration pin CCLK to be able to program the PROM: need to use a STARTUPE2 primitive.

There is an app note XAPP518 that covers in-system programming the flash device through a PCI express endpoint. There’s code for a host application to read the BIT file; and code for the FPGA to provide the data path. There is a mechanism to provide an “emergency” configuration in case the write process is interrupted and the code corrupted.

There are settings to be made in Vivado to specify config details into the bitstream. This will cover selecting the x4 data width and switching to the EMCCLK pin with a correct division ratio. See XAPP586.

### Creating the PROM file

After Vivado’s ”generate bitstream” command, it is necessary to run a further step to create a file ready to be written into the prom: “Tools > Generate Memory Configuration File”. The options for this should look something like the screenshot below. The memory part should be as shown; SPIx4 is important; and the newly generated bit file (normally found in folder <projectname>.runs\impl\_1) should be selected.

Graphical user interface, text, application, email

Description automatically generated

Figure : Vivado Settings to Generate PROM File

### Programming the Config PROM

I have created a simple linux GUI app (Figure 6) to program the configuration prom from that newly generated file. Simple select the filename and click “program”.

Graphical user interface, application

Description automatically generated

Figure : GUI App to Write Configuration PROM

There is also a command line linux application that writes the config prom. Typical usage:

cd ~/software/saturn

./spi-loader/build/spi-loader -a 0 -f prom.bin -v

The -a 0 is important: it sets the destination address to the start of the config PROM. The default is some random location elsewhere!

### Fallback Configuration

See Xilinx 7 series Configuration Guide, and particularly XAPP1247

Fallback configuration is a special case of multiboot configuration. It allows for a fallback “golden” configuration to be loaded if the main one fails to load. We would use this in the following way:

* The config PROM writer will be able to write either configuration;
* The config PROM writer and any application code (eg p2app) will report to the user if the fallback configuration is in use;
* There will be one Saturn “fallback” PROM file SaturnGolden.bin;
* The fallback is identified in user code by identifier:
  + Product id = 1 (Saturn)
  + Product version is any
  + Software ID = 3 (fallback)
  + Software version = any

There are some constraints on how fallback configurations work, and how they must be programmed.

* The “Golden” configuration loads into the Flash memory at address 0
* The “normal” (also called “update” or “multiboot”) configuration address is programmable.
* The fallback configuration must be loaded in SPIx1 mode
* Fallback isn’t triggered by CRC error or watchdog timer if the “update” image is just erased (which could happen)
* Therefore the preferred approach is the XAPP1247 method with “barrier” images

Bitstream settings need to be different for the two FPGA configurations, as a consequence.

|  |
| --- |
| **Normal configuration** |
| set\_property CONFIG\_VOLTAGE 3.3 [current\_design]  set\_property CFGBVS VCCO [current\_design]  set\_property BITSTREAM.CONFIG.EXTMASTERCCLK\_EN Div-2 [current\_design]  set\_property BITSTREAM.CONFIG.SPI\_BUSWIDTH 4 [current\_design]  set\_property CONFIG\_MODE SPIx4 [current\_design]  set\_property BITSTREAM.CONFIG.SPI\_FALL\_EDGE YES [current\_design]  set\_property BITSTREAM.CONFIG.CONFIGFALLBACK ENABLE [current\_design]  set\_property BITSTREAM.CONFIG.SPI\_32BIT\_ADDR YES [current\_design] |
| **Golden (fallback) configuration** |
| set\_property CONFIG\_VOLTAGE 3.3 [current\_design]  set\_property CFGBVS VCCO [current\_design]  set\_property BITSTREAM.CONFIG.EXTMASTERCCLK\_EN Div-2 [current\_design]  set\_property BITSTREAM.CONFIG.SPI\_BUSWIDTH 1 [current\_design]  set\_property CONFIG\_MODE SPIx1 [current\_design]  set\_property BITSTREAM.CONFIG.SPI\_FALL\_EDGE YES [current\_design]  set\_property BITSTREAM.CONFIG.CONFIGFALLBACK ENABLE [current\_design]  set\_property BITSTREAM.CONFIG.NEXT\_CONFIG\_ADDR 0x097FC00 [current\_design]  set\_property BITSTREAM.CONFIG.SPI\_32BIT\_ADDR YES [current\_design] |
| **Barrier images** |
| the multiboot\_address\_table.tcl script is hard to run, but it works out the required settings. Open a vivado console by getting a command window, then  cd c:\xilinx\vivado\2021.2\bin  vivado -mode tcl  cd c:/xilinxdesigns/Saturn/FPGA/multiboot\_address\_table  source multiboot\_address\_table.tcl  (That runs the script in interactive mode). for Saturn, enter:  spi  4  61  256  9730652  the results give load addresses:  0x00000000 golden image  0x0097FC00 timer 1  0x00980000 multiboot image  0x01300000 timer 2  The images should NOT be compressed, or the script will have to be run again  Having made those images, copy the .BIT files into the folder and issue command:  write\_cfgmem -format bin -size 32 -interface SPIx1 -loadbit "up 0x00000000 saturn\_top\_wrapper\_golden.bit up 0x00980000 saturn\_top\_wrapper.bit" -loaddata "up 0x0097FC00 timer1.bin up 0x01300000 timer2.bin" saturngolden.bin -force |

# BITSTREAM.CONFIG.SPI\_BUSWIDTH 1 sets the actual serial load rate

#CONFIG\_MODE SPIx4 is for Vivado tool to check DRC, but doesn’t set the rate itself.

#The advice is to match both settings to the mode you are using.

The “golden” image being built on 5/3/2023

The main image address has been set to at address 16Mbyte into the config PROM. The programming app needs to erase the correct half, and program the correct half.

## Processor PCI Express interface

Ideally we should allow a 4 lane PCI express interface to allow for future expansion. Initially we will use 1 lane Gen2 (Gen2x1); Allow eventually for up to Gen2x4. The PCI Express bus signals are:

|  |  |
| --- | --- |
| PCIe\_nRST | Reset signal, driven by processor |
| PCIECLKREQN | Clock request. Permanently low output from FPGA, once configured. |
| PCIe\_CLK\_P,  PCIe\_CLK\_N | Differential clock; 100MHz as required by PCI Express spec; HCSL levels. Clocks the IP core. AC coupled at the RPi source. |
| PCIe\_TX\_P,  PCIe\_TX\_N | Differential RPi data out; to FPGA data in. AC coupled at the RPi source.  Lane 0 only used for RPi4. |
| PCIe\_RX\_P,  PCIe\_RX\_N | Differential RPi data in; from FPGA data out. AC coupled near the FPGA; use 0.1uF X7R. Lane 0 only used for RPi4. |
| MGTRREF\_216 | Needs 100R to MGTAVTT supply (see also section 3.4) |

Note that the ordering of the high speed GTPE2 transceivers is set in the XDMA core (actually the hard PCIe block). The pinouts can’t be reassigned.

See NVidia OEM design guide for advice on tracking: it is probably still relevant. Raspberry pi recommendation is for 90Ω Traces, and with lengths within a pair matched within 0.1mm

Diagram, schematic

Description automatically generated

Figure : PCI Express Signals

PCIe\_CLK\_nREQ: PCI express “clock request” to processor. 3.3V level. Needs a pullup resistor; driven to 0 by an FPGA output.

PCIe\_nRST: active low PCI express reset from processor. Treat as a CMOS 3.3V level input. This should reset the PCI express IP core in the FPGA.

The PCIe core has “preferred” GTP transceivers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PCIe Lane** | **GTP transceiver** | **Pin names**  **XC7A100T-FGG484** | **Pin Names**  **XC7A100T-FGG676** | **Pin Names**  **XC7A200T-FBG676** |
| Lane 0 | X0Y7 | D7 MGTPTXP3\_216  C7 MGTPTXN3\_216  D9 MGTPRXP3\_216  C9 MGTPRXN3\_216 | D10 MGTPTXP3\_216  C10 MGTPTXN3\_216  D12 MGTPRXP3\_216  C12 MGTPRXN3\_216 | D10 MGTPTXP3\_216  C10 MGTPTXN3\_216  D12 MGTPRXP3\_216  C12 MGTPRXN3\_216 |
| Lane 1 | X0Y6 | B6 MGTPTXP2\_216  A6 MGTPTXN2\_216  B10 MGTPRXP2\_216  A10 MGTPRXN2\_216 | B9 MGTPTXP2\_216  A9 MGTPTXN2\_216  B13 MGTPRXP2\_216  A13 MGTPRXN2\_216 | B9 MGTPTXP2\_216  A9 MGTPTXN2\_216  B13 MGTPRXP2\_216  A13 MGTPRXN2\_216 |
| Lane 2 | X0Y5 | D5 MGTPTXP1\_216  C5 MGTPTXN1\_216  D11 MGTPRXP1\_216  C11 MGTPRXN1\_216 | D8 MGTPTXP1\_216  C8 MGTPTXN1\_216  D14 MGTPRXP1\_216  C14 MGTPRXN1\_216 | D8 MGTPTXP1\_216  C8 MGTPTXN1\_216  D14 MGTPRXP1\_216  C14 MGTPRXN1\_216 |
| Lane 3 | X0Y4 | B4 MGTPTXP0\_216  A4 MGTPTXN0\_216  B8 MGTPRXP0\_216  A8 MGTPRXN0\_216 | B7 MGTPTXP0\_216  A7 MGTPTXN0\_216  B11 MGTPRXP0\_216  A11 MGTPRXN0\_216 | B7 MGTPTXP0\_216  A7 MGTPTXN0\_216  B11 MGTPRXP0\_216  A11 MGTPRXN0\_216 |
| (the preferred GTP transceivers are the same for each of: XC7A100TFGG484, XC7A100TFGG676, XC7A200TFBG676; but the pins numbers change between packages) | | | | |

# FPGA DSP Design Notes - Conventional Time Domain Processing

The board will be used in one of 3 ways:

1. With an external application, data sent over ethernet using protocol 1 by Raspberry pi. A C code app will have to collect data from the FPGA and forward packets over ethernet, and vice versa. This will include interleaving multiple DDC streams on RX. This is not demanding.
2. With an external application, data sent over ethernet using protocol 2 by Raspberry pi. This is the most stressing case and the data format at the FPGA interface should try to match protocol 2 data structures to minimise processor load.
3. With the Raspberry pi running an SDR application (eg pihpsdr). The C program will need to perform the DSP and UI functions as well as data transfer.

We know that a lesser processor can handle the data transfers for protocol 1; so managing the various control etc data registers won’t be a problem. It is only the main sample flows that need to be designed carefully.

## FPGA Requirements

* Use PCI express to communicate with host computer
* TX function
  + Accept I/Q baseband TX samples
    - 48KHz, 16 bit I/Q (protocol 1)
    - 192KHz 24 bit I/Q (protocol 2)
  + Upconvert to required TX frequency
  + Present samples to DAC
  + Be able to gate off the DAC samples when TX strobe not asserted
  + Be able to TX a test source waveform
  + For CW modes – ramp amplitude up, down as key pressed/released
  + Ramp up/down time and waveform needs to be programmable in the range 1-6ms
* Downconvert RX
  + Have at least 5 separate RX DDC; Warren requests at least 10.
    - If they were all at max sample rate, data transfer is likely a problem
    - If they were all full performance, FPGA resources may be a problem
    - Could we consider “lesser” receivers for some of them?
  + Option to interleave output samples for two DDC
  + Each DDC select samples from ADC1, ADC2, TX samples, RX test source
  + Separate selection for each of RX and TX (current design does this, but it goes beyond what protocol 2 requires)
  + Downconvert to baseband I/Q
  + Present samples to PC for each DDC streams
  + Latency is a known concern, so data transfers will have to be optimised
  + Selectable sample rate 48/96/192/384/768/1536KHz
  + Accept “overscale” input from ADC & latch
  + Present latched “overscale” to processor, which clears when read
* Clock Management
  + Accept 10MHz ref signal from local xtal or external input
  + Control 122.88MHz VCXO
  + Take clock inputs from VCXO, ADC
  + Generate 122.88MHz DAC clock
  + Derive CODEC master clock (12.288MHz)
* CODEC interface
  + CODEC setup will be via I2C; connected to the FPGA but software on the host will write the registers
  + Accept 48 KHz mic scalar samples from CODEC, send via FIFO to host computer
  + Accept 48KHz L/R speaker samples, send via FIFO to CODEC
  + When in CW mode, add to the TX path a CW sidetone
  + CW sidetone from freq programmable DDS, amplitude ramps up and down to avoid keyclick.
  + Overall sidetone amplitude set according to user preference
* RF interface
  + Provide interface signals for ANAN7000 series RF
  + Drive SPI-like interface for RX (set entirely by processor)
  + Drive SPI-like interface for TX (set by processor, updated by TX strobe)
  + Send control signals to RF attenuators
* Configuration
  + Be able to read back various strobes
  + Be able to assert TX strobe, FIFO resets, control lines for TX & RX
  + Be able to rewrite FPGA configuration PROM without jumper change
  + Be able to reset all FIFOs
  + Be able to read back an FPGA code version number
* I/O Signals
  + Accept PTT, Keyer signals and send to host PC
  + Accept analogue inputs: either attached ADC or Xilinx ADC
  + Drive I2C for codec etc

## Sampling Architecture

### RX

Uses conventional 122.88MHz clock; 16 bit ADC/DAC; designed for 2 ADC. The required Output Rates are:



I planned to adopt the concept of Phil Harman’s approach with CIC filter followed by decimate-by-8 FIR. That means the CIC is used well within its passband and is almost flat (0.4dB droop predicted at +- Fs/2). However Warren recommends a compensating FIR filter.



Consequently the FIR has the same shape and the same coefficients can be used for each sample rate setting.



Figure 8: CIC Decimate by 40, 6 stages (FIR width shown in red)

The CIC is decimate by 40 for final Fs=384KHz. Passband droop at +/-192KHz is ~ 0.4dB. The FIR is needed to accelerate the cutoff at the Fs/2 point; the number of taps driven by the sharpness desired.

The FIR aliases at its input sample rate Fs. For the diagram above for final Fs=384KHz, FIR Fs = 8\*384 = 3072KHz. So the filter aliases (at 3072KHz , 6144KHz etc) map onto the nulls in the CIC spectrum. The filter width is narrow; the CIC provides ~140dB rejection at the FIR alias points.

6 stage CIC, decimating by a variable rate



Figure : Decimate by 8 Filter

From the diagram (Figure 9) the operation of the decimating filter is clear. The bandwidth is defined at the start; decimation simply reduces the sample rate to 1/8 of the original. The spectrum display will be +/- Fs/2. A signal just above Fs/2 will alias to being just inside the passband at just above -Fs/2. So we need the filter to pass the required signals inside +/-FS/2 and reject others to the stop band required. My initial thoughts were that a stopband depth 120dB may be appropriate; Warren thinks 140dB should be a better target.

For final Fs=48KHz we could consider having the filter select 40KHz of “useful” spectrum. That would make the cutoff +/-20KHz. Potentially, CFIR compensation could be built into the FIR filter.

### TX

TX has fewer choices. It will implement a 24 bit datapath feeding a 16 bit DAC; 24 bit I/24 bit Q input samples @Fs=48KHz (protocol1) or 24 bit I/24 bit Q input samples @192KHz (protocol2). If used for protocol 1, the software will need to zero-pad the samples to 24 bits.

The DDS does not seem to need to be used in “unit circle” mode. Amplitude variation negligible after filters have settled. However it may impact Puresignal linearisation if not in that mode.



## Receiver Implementation

The receiver currently contains 10 DDCs. Each channel can accept data from ADC1, ADC2, TX samples or a test source. I/Q samples from all 10 channels go to separate FIFOs, with an option for DDC0 and DDC1 to interleave. The receiver is clocked at 122.88MHz and each DDC has an individually selectable sample rate.

### DDC Architecture

The DDC is the receiver building block; it outputs downconverted and decimated / filtered I/Q samples having processed an ADC input stream. The Vivado block design flowgraph is shown in Figure 10 and Figure 11. The first is unfortunately hard to read! The DSP is implemented entirely using Xilinx IP blocks provided as part of the Vivado package. Each IP core is customised according to user entered parameters. The DSP cores are marked in red in the first figure; the rest are infrastructure.



Figure : DDC Flowgraph in Vivado

Diagram

Description automatically generated

Figure Single Channel DDC

The processing is as follows:

* Input samples are chosen from 4 sources according to the channel select bits. Sources available are:
  + ADC1
  + ADC2
  + TX samples that go to the DAC
  + A “test sources” DDS
* The downconversion frequency is generated using a quadrature DDS. Its NCO sets the frequency, and controls a phase accumulator. sin/cos lookup tables generate I/Q. The DDS free-runs at 122.88MHz. The DDC parameters are:
  + SFDR 95dB (implies 16 bit I/Q output)
  + Frequency resolution 0.05Hz (implies 32 bit phase accumulator)
* The selected sample stream connects to one port of a complex multiplier. The data sources provides the I samples; the Q samples are zeroed. The other port connects to the 16 bit I/Q DDS. The complex multiplier parameters are:
  + 16+16 bit I/Q inputs
  + 16+16 bit I/Q outputs
  + Set to random rounding of output data width 24 bits
* The output of the complex multiplier is an I/Q stream at zero centre frequency. It now needs to be filtered and decimated to the final bandwidth and matching sample rate. A dual stage filtering scheme is used with CIC decimating to 8x the final sample rate, then a further decimate-by-8 FIR to set the final output sample rate.
* The complex sample stream is split into two separate scalar streams for the individual CIC filters
* Each stream is filtered, with decimation set in binary steps from 10 to 320 depending on required final sample rate. The CIC parameters of the core are:
  + 6 CIC stages
  + Differential delay = 1
  + Input width = 24 bits
  + Output data width = 32 bits
* The FPGA has 25x18 multipliers; input word width into the FIR filter is a significant call upon the FPGA resources. 25 bits of output data are selected for the FIR filter: bits 29:4.
* The two scalar filtered streams are recombined into a single complex stream. An FIR filter IP processes both the I and the Q streams, re-using its multiplier and coefficient storage resources. This has been found by experimentation to use fewer resources that two separate filters, but numerically makes no difference.
* The coefficients were originally generated using a web filter design site, and converted to .coe format using an excel spreadsheet. More recently Warren Pratt has designed a filter with CI equalisation, increasing the gain slightly towards the edge of the filter passband. The core takes floating point coefficients, and normalises them.
* The FIR parameters are as follows:
  + 1024 taps
  + Coefficient file: 1024tap\_TX\_filter\_tfilter.coe (yes I know it says TX!)
  + Decimate by 8
  + Input width 25 bits
  + Coefficient width 18 bits, fractional bits 20
  + Output rounding is “non symmetric rounding up”
  + Output width 27 bits
* The output data is limited to 24 bits by taking the 24 LSBs.
* The single stream is expanded back to an I/Q stream giving 24+24 bit I/Q data at the required final sample rate.
* DDCs are paired, for possible interleaving of their sample streams. When the DDCs are set to interleave, DDCn will fill its FIFO with alternate DDCn and DDCn+1 samples. The DDS I/Q output for DDCn downconversion will be used for DDCn+1 too.

In protocol 2 each receiver can have a different sample rate, so the same FIR can’t be shared across all receivers. Each receiver slice has one FIR shared by I and Q. The max input sample rate is 12.288MHz (CIC decimating by 10, FIR decimate by 8 to give 1536KHz I/Q sample rate). The filter implements 2 channels for each of I and Q. The final AXI stream width change gives parallel I/Q data output at the required sample rate.

Each DDC FIR uses 27 DSP48 slices and 13 Block RAMs. The CIC filters have been set to not use DSP48 slices and use FPGA logic instead; it seems to make little overall impact: it seems to fit combinatorial logic well.

The filter has been time domain simulated to check the word sizes. (I also found at the same time that the DDS needs a reset strobe, or its output is forever XXXX because it has accumulated XXXX when starting up). The test bench collects bit-true integer I/Q samples into a file. An Octave script is used to plot those samples to view the downconverted spectrum.

As the test source, a single DDS is used with user programmable frequency.

### DDC Registers

Each DDC has a 32 bit frequency-setting delta phase word. A single register provides the DDC input selection, and a further register provides the DDC sample rates.

ADC overflow is detected and latched: see section 9.1.5.

ADC randomise inverts bits 15:1 if bit0 is 1. This is automatically removed in the FPGA input circuit. ADC gain and dither are also programmable: see section 7.4.

DDCs can be interleaved, as long as the 1st one is an even DDC. If DDCs N and N+1 are set to interleaved, the LO for DDC N+1is automatically selected to the LO for DDC N, to ensure phase coherence.

|  |  |  |
| --- | --- | --- |
| **DDC Rate Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| DDCRate (2:0) | DDC0 sample rate | Sets CIC decimation  0: disabled (operates at 48KHz);  1: 48KHz;  2: 96KHz;  3: 192KHz;  4: 384 KHz;  5: 768 KHz;  6: 1536 KHz  7: interleaved with next DDC; rate set by bits for next DDC (only valid for even DDCs) |
| DDCRate (5:3) | DDC1 sample rate |
| DDCRate (8:6) | DDC2 sample rate |
| DDCRate (11:9) | DDC3 sample rate |
| DDCRate (14:12) | DDC4 sample rate |
| DDCRate (17:15) | DDC5 sample rate |
| DDCRate (20:18) | DDC6 sample rate |
| DDCRate (23:21) | DDC7 sample rate |
| DDCRate (26:24) | DDC8 sample rate |
| DDCRate (29:27) | DDC9 sample rate |

|  |  |  |
| --- | --- | --- |
| **DDC Input Select Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| DDCInSel(1:0) | DDC0 input select | DDC select bits:  00: ADC1  01: ADC2  10: Test source  11: TX samples |
| DDCInSel(3:2) | DDC1 input select |
| DDCInSel(5:4) | DDC2 input select |
| DDCInSel(7:6) | DDC3 input select |
| DDCInSel(9:8) | DDC4 input select |
| DDCInSel(11:10) | DDC5 input select |
| DDCInSel(13:12) | DDC6 input select |
| DDCInSel(15:14) | DDC7 input select |
| DDCInSel(17:16) | DDC8 input select |
| DDCInSel(19:18) | DDC9 input select |
| DDCInSel(30) | DDCEnabled | =1 if DDC data transfer is enabled |
| Initial register value | 0 |  |

|  |  |  |
| --- | --- | --- |
| **DDC Tune Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| DDC0Tune (31:0) | DDC0 LO tuning | 32 bit phase word for DDS tune  Centre frequency = N.122.88MHz/(2^32) |
| DDC1Tune (31:0) | DDC1 LO tuning |
| DDC2Tune (31:0) | DDC2 LO tuning |
| DDC3Tune (31:0) | DDC3 LO tuning |
| DDC4Tune (31:0) | DDC4 LO tuning |
| DDC5Tune (31:0) | DDC5 LO tuning |
| DDC6Tune (31:0) | DDC6 LO tuning |
| DDC7Tune (31:0) | DDC7 LO tuning |
| DDC8Tune (31:0) | DDC8 LO tuning |
| DDC9Tune (31:0) | DDC9 LO tuning |
| RXTestTune(31:0) | Test source tuning |

There are two FIFOs in the RX path. After each DDC there is a small FIFO before the data multiplexer. After the data multiplexer there is the “main” RX FIFO to take up peaks and troughs in Raspberry Pi data readout. The smaller FIFOs have overflow monitored by an IP: each will get a latched bit set if overflow occurs. See section 9.1.5.

## Transmitter Implementation

The transmitter is clocked at the full output sample rate (122.88MHz). It ultimately provides sample data to the TX DAC (MAX5891) with offset binary data format. That is simply converted from 2’s complement by inverting the MSB.

### DSP Architecture

The TX uses the same approach; interpolate by 8 FIR then a CIC interpolator. For CW TX, the DDS is adjusted to offset by the sidetone frequency; sidetone is generated using a DDS. CW keying simple scales its amplitude through the I/Q sample path. We don’t need a TX test source – just need to be able to turn on the DDS output at selectable amplitude.

For protocol 1 the TX sample rate is 48KHz, 16+16 bit I/Q. For Protocol 2 it is 192KHz, 24 + 24 bit I/Q.

The TX flowgraph is shown in Figure 12 and Figure 13. This includes a partly implemented path for TX envelope generation, for EER; but this is incomplete. (I know for example it needs a different I/Q feed with a delay). The TX again uses Xilinx IP cores, marked in red.



Figure : Transmitter Vivado Flow Graph



Figure : Transmitter

The TX signal path is as follows:

* I/Q modulation samples are selected from 1 of 4 sources:
  + The TX samples from the DSP application (eg Thetis)
  + A test DDS source;
  + A CW keyer;
  + A fixed amplitude, 0Hz sample.
* The I/Q samples are multiplexed into a single scalar data stream.
* An interpolating FIR filter bandwidth limits the samples and increases the same rate. Like the RX, the FIR filter has designed using a web filter designer, then the Xilinx coefficient file is generate using an excel spreadsheet. The FIR filter parameters are:
  + Input sample width 16 its
  + 512 filter taps
  + Coefficient file: 512TapLPF\_corner\_20KHz\_tfilter.coe
  + Interpolate by 8
  + Coefficient width 22 bits, fractional bits 24
  + Output width 20 bits
* Filtered samples are converted back to an I/Q stream
* The stream is split into separate I and Q samples.
* Each stream is filtered by identical interpolating CIC filters. The CIC parameters are as follows:
  + 6 stages
  + Differential delay = 1
  + Interpolate by either 80 (protocol2) or 320 (protocol1)
  + Input data width 20 bits
  + Output data width 23 bits
* The filtered samples, now at the final sample rate of 122.88MHz, are converted back into a single I/Q stream and connected to one port of a complex multiplier.
* A quadrature DDS generated samples of the local oscillator for upconversion. The DDS parameters are:
  + SFDR 95dB (implies 16 bit I/Q output)
  + Frequency resolution 0.05Hz (implies 32 bit phase accumulator)
* A complex multiplier multiples the complex modulating samples by the complex DDS samples. The multiplier parameters are:
  + Channel A (modulation) width 23 bits
  + Channel B(DDS) width 16 bits
  + Output width 20 bits
* The In-phase output is selected and scaled in amplitude by a processor-defined 18 bit word. The top 16 bits are taken and used to drive the output DAC. The samples can be gated to 0 when TX is not in progress. The DAC samples are also passed back to the receiver to be downconverted for Puresignal processing.
* Although not fully implemented, a cordic IP core extracts the TX signal magnitude. This would then be used to generate the envelop output signal using a PWM DAC. Alternatively an SPI DAC could be used, but this is not included in the current design.

### Transmitter Registers

|  |  |  |
| --- | --- | --- |
| **TX Config Register registers** | | |
| **Input Bits** | **Function** | **Meaning** |
| TXLOTune[31:0] | TX DDS frequency | 32 bit phase word |
| TXTestFreq[31:0] | Test source tune word | The DDS is clocked at 122.88MHz, but the effective sample rate for frequency calculations is 48KHz or 192KHz depending on protocol in use. |
| TXConfig[1:0] | Select the TX data source | 00: TX I/Q Data  01: Fixed amplitude 0Hz  10: Test DDS source  11: CW keyer |
| TXConfig[2] | Output sample gating | 0: TX/RX controlled  1: always on |
| TXConfig[3] | Protocol | 0: protocol 1  1: protocol 2  (selects interpolation rate) |
| TXConfig[21:4] | Output amplitude | 18 bit amplitude scale word, applied at output of the DDC before the DAC  0x1FFFF (half amplitude) results in DAC levels being just within the 16 bits available |
| TXConfig[29] | Mux Reset | =1: reset 64 to 48 multiplexer |
| TXConfig[30] | IQ Deinterleave | 0: single TX I/Q stream  1: dual stream; odd samples for envelope |
| TXConfig[31] | IQ Modulation enable | 0: I/Q stream from processor disabled  1: I/Q stream enabled |
| Initial value | 0 |  |

The TX LO DDS is 29 bits with a resolution~0.4Hz. The I/Q test source DDS is 29 bits with a resolution ~0.4Hz

### EER

Note that there is no code in Orion to drive this signal; we may be able to remove it completely.

The EER function uses the TX signal envelope, a few times faster than TX I/Q sample rate (Orion is 5Fs). It is viable to calculate the envelope at the full DAC output rate which will need to be decimated to a suitable DAC speed. Note that the envelope should be generated using a different I/Q stream: it has a group delay imparted by the Puresignal software to adjust out the delays between I/Q signal and envelope arriving at the HPA.

Hermes uses a PWM DAC. Orion has no EER code at all. One option was an SPI DAC with 12 bit resolution. There is available Verilog code for an axi stream to SPI IP core. Use the Verilog code to decimate the o/p sample rate to an acceptable rate (eg 384kS/s, compatible with protocol 1 and 2 rates). A suitable DAC is MCP4821.

## Codec & Audio Interface

### Architecture

This interfaces to a TLV320AIC23B codec (same as Hermes). It uses I2S Slave mode, with timing strobes derived by the FPGA. An SPI interface connected to the processor is used for configuration.

Audio sample rate = 48KHz for both protocols. Data is shifted MSB first. The data is signed, 2s complement.



Figure : CODEC Interface

The I2S TS and RX Verilog modules present data: right data = tdata[31:16]; left data = tdata[15:0]. For speaker data, both left and right audio are transferred (2x16 bit samples per 48KHz clock). For mic data, only the left channel data is transferred (1 16 bit sample per 48KHz clock). Data is 2s complement.

### Codec Clocks

The Codec has several clocks that the FPGA needs to generate. The audio sample rate is 48KHz. The Codec has a master clock of 12.288MHz; sample clock divided by 10. This is generated by a Xilinx clock generator.

There is a data clock BCLK, generated by the I2S interface logic.



Currently the entire Codec interface including sidetone DDS is clocked at 12.288MHz rate. But the DDS is “throttled” by the data rate that the I2S interface will accept (ultimately 48KHz word rate) and the TREADY signal as part of the AXI stream interface sets the effective clock rate. The Codec interface and the TX need to maintain constant latency regardless of whether CW or other modes are used. Sidetone is added to the speaker path, not replacing it (this avoids clicks through gating off an active audio signal).

### FPGA Codec Interface Registers

|  |  |  |
| --- | --- | --- |
| **Codec Config Register registers** | | |
| **Input Bits** | **Function** | **Meaning** |
| CodecConfig(31:16) | Sidetone volume | 16 bit ampl word signed |
| CodecConfig(15:0) | Sidetone frequency | 16 bit phase word (note 48KHz effective Fs) |

### Codec I2C or SPI Register Settings

Taken from the Hermes “hermes\_TLV320\_SPI.v” code: these settings will need to be made by the processor at power up. Also some settings at runtime.

The codec MODE pin can be wired (by changing resistors) to 0 or 1 allowing selection of SPI or I2C interface. SPI looks much faster.

The Codec has its ~CS pin wired to the FPGA. This can be used to set 7 bit address = 0x1A, or is used directly for SPI.

The CODEC uses two bytes for a register write: a 7 bit register address and 9 bit register data. The 16 bit word is as follows:

<A6 A5 A4 A3 A3 A1 A0> <D8 D7 D6 D5 D4 D3 D2 D1 D0>

The I2C interface requires 3 bytes to be transferred for a bus transaction:

1. 7 bit address + R/W (noting it is write only)
2. MSB of data word: A6 A5 A4 A3 A3 A1 A0 D8
3. LSB of data word: D7 D6 D5 D4 D3 D2 D1 D0

The SPI interface to the codec just needs a 16 bit SPI write.

|  |  |  |  |
| --- | --- | --- | --- |
| **Register (hex)** | **Value (9 bits, hex)** | **16 bit word** | **Meaning** |
| 0F | 000 | 0x1E00 | Reset device |
| 09 | 01 | 0x1201 | Digital interface activation: set to ACTIVE |
| 04 | 10: line  14: mic no boost  15 mic, boost | 0x0810  0x0814  0x0815 | Analogue audio path control  Line: mic not muted; line input; bypass disabled; sidetone disabled  Mic: mic not muted; mic input; bypass disabled; sidetone disabled  (set bit 0 for 20dB boost) |
| 06 | 00 | 0x0C00  0x0CFF | Power down control.  All elements powered on  All elements powered down; mic bias output disabled. |
| 07 | 02 | 0x0E02 | Digital interface format.  Slave; no swap; right when LRC high; 16 bit; I2S format |
| 08 | 00 | 0x1000 | Sample rate control  No clock divide; sample rate ctrl=0; normal mode, oversample 256Fs (suitable for MCLK=12.288MHz, 48KHz ADC & DAC) |
| 05 | 00 | 0x0A00 | Digital audio path control  DAC soft mute disabled; de-emphasis disabled; ADC high pass filter enabled |
| 00 | Line in gain  0000nnnnn | 0x00nn | Left line input volume  No mute; no simultaneous update; gain=nnnnn |

The CODEC is write only. A series of 16 bit register writes will be needed with no mechanism to determine if the interface is functioning correctly.

### Codec Hardware Interface

|  |  |  |
| --- | --- | --- |
| **CODEC Pin** | **Connection** | **Function** |
| MODE | Hardwired to 0 | Selects I2C |
| CS~ | Hardwired to 0 | Selects address = 0x1A. For SPI needed to latch data after transaction |
| SCLK | FPGA I2C\_SCK | I2C clock; 400KHz, or SPI clock |
| SDIN | FPGA I2C\_SDA | I2C data or SPI data |
| MCLK | FPGA MCLK | 12.288MHz clock |
| BCLK | FPGA BCLK | I2S bit clock |
| LRCIN  LRCOUT | FPGA LRCLK | Left/right select. Both driven by the same FPGA signal. |
| DIN | FPGA I2STXD | I2S serial speaker audio data to CODEC |
| DOUT | FPGA I2SRXD | I2S serial microphone data from CODEC |

### Sidetone

The sidetone path is as drawn in Figure 15. 16 bit signed data is used at each stage. A 16 bit DDS clocked at 48KHz generates the sidetone, generating a 16 bit signed sinusoidal output. This is multiplied by the sidetone volume, then the CW keyer ramp (both 16 bit signed values but only using positive values, 0 to 0x7F). The result is a sinusoid with the correct level, keyed on and off. This is then added to each of the left and right speaker samples. That add outputs the 16 LSB of a 17 bit result: there is potential for overflow at extreme amplitudes.

A diagram of a multiplying process

Description automatically generated

Figure : Sidetone Audio Path

## DSP Improvements

There are performance improvements needed for the DSP implementation:

* TX composite noise performance is now a key discriminator between radios
* For Puresignal the TX filters need to be flatter, with CFIR compensation
* For RX the required performance isn’t clear but is important
* We are likely to want more DDCs

### FIR Filter Sizing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Channels | Taps | DSP | BRAM |
| Original filter, as specified above | 2 | 512 | 4 | 1 |
| Original filter, rounding mode set to full precision (removes symmetric rounding to zero) | 2 | 512 | 3 | 1 |
| Original filter, 1 channel | 1 | 512 | 2 | 1 |
| Warren’s 1024 tap filter, symmetric rounding to zero | 1 | 1024 | 4 | 2 |
| Warren’s 1024 tap filter, input data size = 24 bits o/p data size 47 bits 24 bit coefficients, full precision | 1 | 1024 | 7 | 2 |
| Warren’s 1024 tap filter, input data size = 24 bits o/p data size 47 bits, coefficient size 26 bit, full precision | 1 | 1024 | 7 | 4 |
| Warren’s 1024 tap filter, input data size = 24 bits o/p data size 47 bits, coefficient size 32 bit, full precision | 1 | 1024 | 15 | 4 |
| Warren’s 1024 tap filter, input data size = 24 bits o/p data size 47 bits, coefficient size 32 bit, full precision, 2 channel (this filter implements both I and Q channels) | 2 | 1024 | 23 | 4 |

# Clock Generation

The TX and RX sample path is clocked by a 122.88MHz VCXO. That is phase locked to a 10MHz reference.

## FPGA Clocks

The FPGA uses two clocks:

122.88MHz ADC/DAC sample clock, for all RX/TX paths

Derived 12.288MHZ clock for audio codec, debounce, SPI data shifting etc. Used as the CODEC MCLK source.

The 122.88MHz sample clock may have three different phases in the FPGA:

* The “main” clock input
* The ADCs have a “clock output” connected to the FPGA. Those can be used to register samples into the FPGA; those signals will be an additional “clock capable” input to the FPGA.
* The DAC needs its data driving from a different clock phase to meet the output timing.
* There is a 122.88MHz configuration clock input. This is driven by a single ended source and is probably poorer in quality.

## PLL

The phase lock is:

1. Divide 122.88MHz clock by 3072
2. Divide 10MHz clock by 250
3. Exor the two signals
4. That output goes to the loop filter on the VCXO control voltage.

The Orion board has an auto detector for external reference, and automatically selects it if detected.



Figure : PLL For 122.88MHz VCXO

The VCXO control output needs the same C/R filter that Hermes etc have.

## AXI Bus

The PCI Express interface receives a 100MHz clock. The AXI output from the PCIe core has a 125MHz clock rate.

The AXI Buses after the AXI interconnect operate at two speeds. The AXI infrastructure IP inserts FIFOs as required to manage the different clock rates.

1. Those associated with the radio hardware operate at 122.88MHz;
2. Those providing processor peripherals operate at the native 125MHz.

## Clock Monitor

A “clock monitor” IP receives several input clocks and if it detects transitions, it outputs a 1Hz blink to an LED. If that LED is lit it indicates that all clocks are present. The monitor is clocked at 125MHz (the reasoning being: without that clock it can’t be read back externally anyway). It has monitors for 122.88MHz clock, 122.88MHz configuration clock and 10MHz reference. They can be read back to the processor (see section 9.1.9).

# CW Keyer

## Required Behaviour

The required settings are:

* When breakin == OFF: keyer disabled except when in TX.
* When breakin == SEMI: Keyer enabled. Client app notified of TX while key down or hang delay counted.
* When breakin == QSK: Keyer enabled. Client App notified of TX when key is down including ramp period. (There is no delay after CW ramp)
* When the keyer activates TX mode, and using Protocol 2, the PTT bit in the high priority status message needs to be set.

In all cases the hardware should be enabling high RX attenuation when in TX.

When a key is first pressed, and using protocol 2: a high priority status message should be sent straightaway – not after the normal delay. A speed-up to the normal “sleep for 200ms when in RX” required.

## Ramp Generator

The CW ramp generator generates an “S” shaped ramp to minimise keyclicks. The ramp is used both to modulate a CW drive to the TX, and to generate the audio sidetone. The ramp generator Verilog code has been designed using code from profile.v and Pavel Demin’s code. It is clocked at 122.88MHz (together with the remainder of the TX) and generates an I/Q ramp signal to go to the TX modulation input. The effective clock rate for ramp generation is throttled by the AXI stream tready, with an effective clock rate of 48KHz or 192KHz. The keyer also generates a PTT signal.

The keyer will generate a ramp period in the range 1-10 ms with software programmable ramp shape. A single keyer provides both the I/Q modulation and an amplitude signal to the codec sidetone generation code. The sample rate for sidetone amplitude is always 48KHz, and the clock is different and needs to be remapped using an AXI stream clock converter. The top 16 bits are passed to the sidetone generator, as the audio path is all 16 bits wide.

When TX is enabled and the CW “Enable” bit is set, a key down will initiate TX.

|  |  |  |
| --- | --- | --- |
| **CW Keyer Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| CW\_Keyer[7:0] | CW PTT Delay | 0-255 ms; units ms |
| CW\_Keyer[17:8] | CW Hang time | 0-1023 ms; units ms |
| CW\_Keyer[30:18] | Ramp length | Ramp length in words \* 4 (this sets the byte address it steps up to) |
| CW\_Keyer[31] | Enable | 0=off; 1=enabled |

The Ramp amplitude is set by a dual port RAM. The RAM data needs to be configured after power up by the processor; it sets a 24 bit signed amplitude vs time as the keyer is pressed and released. Data with an amplitude range between 0 & 8388607 is appropriate; providing negative values is meaningless. The RAM holds amplitude samples at either 192KHz or 48KHz sample rate; the memory should hold an “S” shape waveform.

|  |  |
| --- | --- |
| Ramp length (words) | 2048 |
| Max Ramp duration (protocol 1) | 42.6ms |
| Max ramp duration (protocol 2) | 10.6ms |
| Sample length for 5ms ramp (protocol 1) | 240 |
| Sample length for 5ms ramp (protocol 2) | 960 |

Samples of a suitable waveform have been calculated using a spreadsheet for testing. See “cw-shaping-in-dsp.pdf” by VE3NEA for an algorithm.

## Iambic Keyer

Not yet implemented: but the intent will be to re-use the Verilog code from Orion “as is”. The iambic keyer will take “dot” and “dash” inputs and a programmed speed, and generate a CW “key down” bit to drive the ramp generator. Inputs can be from connected hardware, or passed through from a PC application using the “CWX” data encoded into protocol 2.

|  |  |  |
| --- | --- | --- |
| **Iambic Keyer Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| IambicConfig(7:0) | Keyer speed | Speed in WPM |
| IambicConfig[15:8] | Keyer weight | Typ 33-66 |
| IambicConfig[16] | Reversed | Swap dot/dash inputs if 1 |
| IambicConfig[17] | Iambic Enable | 1=Enable iambic keyer |
| IambicConfig[18] | ModeB | Keyer mode B if 1; mode A otherwise |
| IambicConfig[19] | StrictSpacing | =1 if strict character spacing enforced. |
| IambicConfig[20] | CWX Straightkey | Host key down |
| IambicConfig[21] | CWX Host Dot | Host Dot |
| IambicConfig[22] | CWX host dash | Host Dash |
| IambicConfig[23] | Breakin | If set, CW sidetone should add to existing audio  Else should replace it |

Note the sidetone bit from protocol 2 should affect audio, not carrier ramp. Suggest set volume to 0 if sidetone disabled.

## Protocol 2 CW bits

Not strictly Saturn documentation, but… the various CW bits on the protocol 2 messages are confusing and in some places inaccurate.

TX specific byte 5 bit 1 “CW”-> should enable CW mode in Saturn, activating the keyer.

TX specific byte 5 bits 7 “break in”: if set, sidetone audio should add to RX audio; if not set, it should replace RX audio.

High priority byte 5 bit 0 “CWX”: this is NOT an enable for CWX. It is “key down” for PC initiated CW.

High priority byte 5 bits 1,2: PC “dot” and “dash” inputs should OR with h/w inputs; but probably no longer used.

Right now, Break-In bit from protocol 2 enables the CW keyer.

# RF System Control & GPIO Signals

This function is clocked at 12.288MHz (Fs/10). To be included into the FPGA block:

* 2x5 bit atten control output
* 6 bit atten control output
* DAC drive level PWM output
* Aux DAC output (EER)
* RX/TX SPI control

## RF SPI Interfaces

The radio uses the ANAN7000DLE RF hardware, and its SPI control interface. Two words are used – 16 bits TX, and 32 bit RX. The data is transferred to the radio whenever a new bus write is executed.

Most significant bit shifted first

U3/U5: TX Settings: 16 bit SR. Serial data = ALEX\_SPI\_SDO; Serial clock = ALEX\_SPI\_SCK; LOAD clock = ALEX\_TX\_LOAD

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TX\_SPI[15:0]** | | | | | |
| **Bit** | **Function** | **Notes** | **Bit** | **Function** | **Notes** |
| 0 | N/A |  | 8 | ANT1 |  |
| 1 | N/A |  | 9 | ANT2 |  |
| 2 | TXRX\_STATUS | unsure | 10 | ANT3 |  |
| 3 | LED D9 | Yellow LED | 11 | TXRX\_RELAY | Operates T/R relay. 1=TX |
| 4 | BPF3 | 20-30m LPF | 12 | LED-D7 | Red LED |
| 5 | BPF2 | 40-60m LPF | 13 | BYPASS | 6m LPF |
| 6 | BPF1 | 80m LPF | 14 | BPF5 | 10-10m LPF |
| 7 | BPF0 | 160m LPF | 15 | BPF4 | 15-17m LPF |

(Note bit 11 isn’t a CPU register and is provided by a hardwired TX/RX signal)

U6/U10/U7/U13: RX Settings: 32 bit SR. Serial data = ALEX\_SPI\_SDO; Serial clock = ALEX\_SPI\_SCK; LOAD clock = ALEX\_RX\_LOAD. (See Protocol 2 document for documentation on these settings)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **RX\_SPI[31:0]** | | | | | |
| **Bit** | **Function** | **Notes** | **Bit** | **Function** | **Notes** |
| 0 | YELLOWLED |  | 16 | YELLOWLED 2 |  |
| 1 | 13HPF | 10-22MHz BPF | 17 | 13HPF 2 | 10-22MHz BPF |
| 2 | 20HPF | 22-35MHz BPF | 18 | 20HPF 2 | 22-35MHz BPF |
| 3 | 6MLNA | 50MHz BPF&LNA | 19 | 6MLNA 2 | 50MHz BPF & LNA |
| 4 | 9.5HPF | 6-10MHz BPF | 20 | 9.5HPF 2 | 6-10MHz BPF |
| 5 | 6.5HPF | 2.5-6MHz BPF | 21 | 6.5HPF 2 | 2.5-6MHz BPF |
| 6 | 1.5HPF | 1-2.5MHz BPF | 22 | 1.5HPF 2 | 1-2.5MHz BPF |
| 7 | N/A |  | 23 | N/A |  |
| 8 | XVTR RELAY | Transverter in | 24 | RX2\_GROUND | When 1, RX2 i/p disconnected |
| 9 | EXT1 RELAY | Ext 1 in | 25 | N/A |  |
| 10 | N/A |  | 26 | N/A |  |
| 11 | RX BYPASS RELAY | PS sample select: Selects main or RX\_BYPASS\_OUT | 27 | N/A |  |
| 12 | HPF\_BYPASS | RX1 Filter bypass | 28 | HPF\_BYPASS 2 | RX2 filter bypass |
| 13 | N/A |  | 29 | N/A |  |
| 14 | RX MASTER IN RELAY | (selects main, or transverter/ext1) | 30 | N/A |  |
| 15 | REDLED |  | 31 | REDLED 2 |  |

## RX Attenuators

|  |  |  |
| --- | --- | --- |
| **RX Data Conversion Register 1** | | |
| ADC1\_Ctrl[4:0] | ADC1 atten when RX | 5 bit atten setting for RX state; 1dB step |
| ADC1\_Ctrl[9:5] | ADC1 atten when TX | 5 bit atten setting for TX state; 1dB step |
| ADC2\_Ctrl[14:10] | ADC2 atten when RX | 5 bit atten setting for RX state; 1dB step |
| ADC2\_Ctrl[19:15] | ADC2 atten when TX | 5 bit atten setting for TX state; 1dB step |

Easy to derive from a local SDR app. If using Protocol 2, the RX attenuation during TX is specified in the TX specific message.

## TX Attenuators & Drive Level

|  |  |  |
| --- | --- | --- |
| **TX Data Conversion Register** | | |
| DAC\_CTRL[7:0] | RX DAC drive level | PWM DAC drive level when RX |
| DAC\_CTRL[15:8] | TX DAC drive level | PWM DAC drive level when TX |
| DAC\_CTRL[21:16] | RX DAC Attenuation | 6 bit atten value when RX (0.5dB steps) |
| DAC\_CTRL[29:24] | TX DAC Attenuation | 6 bit atten value when TX (0.5dB steps) |

## GPIO register

|  |  |  |
| --- | --- | --- |
| **General Purpose I/O Register** | | |
| GPIO[0] | MIC Bias Enable | =1 to provide electret bias on 3.5mm jack |
| GPIO[1] | Input\_PTT\_Select | 0=PTT on ring; 1=PTT on tip |
| GPIO[2] | Mic\_Signal\_Select | 0=mic on tip, 1 = mic on ring |
| GPIO[3] | Mic\_Bias\_Select | 0=bias on ring; 1= bias on tip |
| GPIO[4] | Spkr\_amp\_Mute | 1=mute. 0 = normal operation |
| GPIO[5] | Balanced\_Mic\_Select | =1 to enable balanced mic input |
| GPIO[8] | ADC1 RAND | =1 to randomise data |
| GPIO[9] | ADC1 PGA | =1 to enable ADC 3dB amplifier |
| GPIO[10] | ADC1 DITHER | =1 to dither the clock |
| GPIO[11] | ADC2 RAND | =1 to randomise data |
| GPIO[12] | ADC2 PGA | =1 to enable 3dB amplifier |
| GPIO[13] | ADC2 DITHER | =1 to dither the clock |
| GPIO[15:14] |  | Spare outputs from FPGA |
| GPIO[22:16] | User outputs | Open collector o/p (6 bits) |
| GPIO[24] | MOX (TX strobe) | 1=TX |
| GPIO[25] | TX enable | 1=TX enabled. Enabled MOX bit or keyer to initiate TX. |
| GPIO[26] | Byteswap | =0: DMA transferred I/Q & codec data is in native byte order  =1: data is in network byte order |
| GPIO[27] | TX\_Relay\_Disable | 0=normal; 1=TXRX relay & PA disabled |
| GPIO[28] | Puresignal enable | Not used. |
| GPIO[29] | ATU TUNE output | 1=tune |
| GPIO[30] | Transverter enable | 1=transverter |

(note PGA bits set to zero in protocol 2 Orion, and not transferred in protocol 2)

Some of these signals need gating to provide the required strobes:

|  |  |  |
| --- | --- | --- |
| **Strobe** | **Purpose** | **Logic** |
| MOX | =1 for TX | ((CPU\_MOX || keyer\_MOX) && TX\_ENABLED) |
| DRIVER\_PA\_EN | Controls driver amplifier after DAC. =1 to enable. | Same as MOX |
| CTRL\_TRSW | Additional TX/RX relay | MOX && transverter\_enable |
| TXRX\_RELAY | Controls relay drive into SPI and as an LED output | MOX && ! TX\_RELAY\_DISABLE |

Diagram

Description automatically generated

## Status Readback

Various strobes and status signals can be read back into the processor.

|  |  |  |
| --- | --- | --- |
| **Status Readback Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| Status[0] | PTT in | 1=TX request |
| Status[1] | - | Not used |
| Status[2] | Dot Key in | 1=key down |
| Status[3] | Dash Key in | 1=key down |
| Status [4] | User input IO4 |  |
| Status [5] | User input IO5 | IO5 used as a ~TX inhibit input |
| Status [6] | User input IO6 |  |
| Status [7] | User input IO8 | IO8 used as a CW key input |
| Status [8] | 13.8v detect in | 1= power valid |
| Status[9] | ATU tune complete | Feedback. Valid state TBD |
| Status[10] | PLL locked | 1= 10MHz/122.88MHz PLL is locked |
| Status[11] | CW Key Down | 1 if CW TX in progress |
| Status[31] | TX\_ENABLE | External input J54; if 0, TX is gated off |
| Status[63:32] | FPGA Version ID (4 bytes) | 32 bit user value, holding f/w ID from USR\_ACCESS register  (currently holds a date code) |

Note IO4/5/6/8 present the true input logic state; not inverted through FPGA.

|  |  |  |
| --- | --- | --- |
| Device input | Diagram, schematic  Description automatically generated | FPGA transfer to processor |

## Clock Regions For interfaces

The AXI4 and AXI4-Lite buses from the PCI Express interface all have 125MHz bus rate. This can be changed using an AXI interconnect IP. The Various bus rates are defined here.

### AXI4 DMA Transfer bus

DMA transfers are all to or from FIFO, and FIFO devices can change the clock region. All AXI4 DMA interfaces should be full speed 125MHz on the processor side. The FIFO monitor IPs should also be clocked at 125MHz.

### AXI4-Lite Register Access bus

There is a mix of clock rates use on this bus. I’m not clear whether it is more efficient to have one split between clock rates then fully synchronous AXI interconnects, or whether I can pick & choose. Interfaces feeding control data to the radio DSP should be operated at 122.88MHz.

|  |  |  |
| --- | --- | --- |
| **Interface** | **Interface IP** | **Clock Rate** |
| FIFO monitor IPs | FIFO monitor | 125MHz |
| Debug LEDs | Config 64 | 125MHz |
| SPI interface to config PROM | Xilinx SPI | 125MHz |
| XADC (for temp, PSU monitoring) | XADC | 125MHz |
| I2C interface to CODEC | Xilinx I2C | 122.88MHz |
| TX registers | Config 64 | 122.88MHz |
| Codec Registers | Config 64 | 122.88MHz |
| RX registers | Config 256 | 122.88MHz |
| Keyer Registers | Config 64 | 122.88MHz |
| CPU Readback registers, user register readback | Read64 | 122.88MHz |
| ADC overrange latch | AXI overrange | 122.88MHz |
| SPI ADC (for RF level monitoring) | AXI SPI ADC | 122.88MHz |
| Alex RF interface | AXI Alex interface | 122.88MHz |

# Processor Data Interface

This section describes the connections between the processor Raspberry Pi Compute module and the FPGA. It could eventually be repurposed into an API description.

## Physical Layer Interface

The physical layer interface between FPGA and Raspberry PI is through PCI Express. This is supported natively in the Pi4 CM. A Xilinx IP core offering DMA and bus access is used to provide appropriate bus interfaces internal to the FPGA.

### XDMA Interfaces

There are primarily two different data interfaces into the FPGA:

1. A 32 bit AXI4-Lite bus provides read/write access to the many registers in the FPGA. This is a relatively slow interface (only about 4Mbyte/s) which is fine for configuration settings and no use at all for sample data transfer.
2. A 64 bit AXI4 bus interface is accessible for Direct Memory Access transfer of blocks of sample data. There are two DMA transfer engines for each or read and write. Each can achieve ~40Mbyte/s with 4Kbyte transfers, and double that for 8Kbyte transfers; they get slower because of the setup overhead for smaller transfers.

From the FPGA hardwire side there are 8 or more AXI-4 streams of data: two providing data to the hardware (speaker data, I/Q TX data) and 6 or more providing data from the hardware (microphone samples and DDC I/Q samples).

From the processor side there are 3 options for reading and writing data via the PCI express DMA/bridge subsystem:

1. Processor reads and writes via an AXI4-lite interface. Bandwidth available ~4Mbyte/s. OK for register setting but inappropriate for I/Q data transfer. Not considered further.
2. DMA reads and writes to separate AXI-4 streams directly interfaced to the IP core. This would be easiest, but the device driver for ARM processors is VERY slow. Requires TLAST to be asserted in data streams; see <https://github.com/XavierAudier/tlast_generator>
3. DMA reads and writes via an AXI-4 bus interface. Smaller FIFOs may be OK. This can achieve measured rates of 100Mbyte/s over a 64 bit AXI-4 bus but does need IP to access the FIFOs. **This is the selected option.**

Diagram, schematic

Description automatically generated

Figure : AXI4 Stream connection to data FIFOs

Diagram, schematic

Description automatically generated

Figure : AXI4 Bus connection to data FIFOs

For AXI-4 bus interface to the FIFOs: The choice seems to be to use an AXI streaming FIFO, or to have some simple IP that translates an AXI-4 lite bus transaction to a stream master write (asserting TVALID) or read (accepting TREADY). The stream reader/writer IP has been created for this purpose. It occupies an address range so that a DMA (with incrementing target address) still reads or writes the same stream data.

### Data Endian-ness

The ARM processor seems to be little endian, like the Intel PC world. The PCI express interface does byte lane translation so the registers directly accessed on the AXI4-Lite bus come out OK. No additional translation required. DMA sample data arrives in a form suitable for local processing; for transfer to a PC for protocol 1 or protocol 2 use, the sample data needs to be converted to network byte order.

An AXI4-Stream with a 64 bit data payload 0x8877665544332211, when DMA transferred to memory, arrives with the bytes as follows:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Addr** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| Byte | 0x11 | 0x22 | 0x33 | 0x44 | 0x55 | 0x55 | 0x77 | 0x88 |

A 16 bit read at address 0 gives data 0x2211; a 32 bit read at address 0 gives data 0x44332211. A 64 bit read would give data 0x8877665544332211. So without any byte swapping in the FPGA, the data is naturally aligned for local processor reads and writes.

Data written to AXI4-lite bus registers will automatically be written with correctly ordered bytes.

The data sent to or received from the PC for Thetis is in “network endian” format which is big endian. This means that byte swapping is needed so that data DMA’d out into the ARM processor can be sent directly to the PC. It does mean that translation would be required if a local DSP app (eg Pihpsdr) is used.

Programmable hardware byte swapping has been implemented on the data paths in the FPGA. Bit “Byteswap” in the GPIO register controls byte swapping. This bit should be set to 0 for native access to the data (for example a locally running instance of Pihpsdr). It should be set to 1 to put I/Q and codec data into network byte order (for example if running a protocol 1 or protocol 2 server application).

### DMA Engine

The XDMA IP core contains 4 DMA engines: 2 for raspberry Pi to FPGA write transfers, and two for FPGA to Raspberry Pi read transfers. The DMA engines work from DMA descriptors, which describe a transfer; transfers can be chained so several are executed one after the other. The DMA engine supports Scatter-Gather DMA (SGDMA) modes; and FPGA hardware can optionally push DMA descriptors direct to the FPGA through a Verilog interface (Descriptor bypass mode).

In the simplest mode, user code calls a function to initiate a transfer. The XDMA device driver creates appropriate DMA descriptors in host (RPi) memory. It tells the DMA engine their location and the DMA engine fetches them from host memory. The DMA engine executes the transfer, and informs the driver by interrupt when the transfer is complete. The whole process is dominated by the overheads of initiating the process for small transfers.

|  |  |  |
| --- | --- | --- |
| **Bytes** | **Average time (ms)** | **Speed (Mbyte/s)** |
| 128 | 0.114 | 1.12 |
| 256 | 0.106 | 2.42 |
| 512 | 0.134 | 3.82 |
| 1024 | 0.114 | 8.98 |
| 2048 | 0.128 | 16.00 |
| 4096 | 0.137 | 29.90 |
| 8192 | 0.13 | 63.02 |
| 16384 | 0.162 | 101.14 |
| 32768 | 0.237 | 138.26 |

Table : DMA Transfer timings

A 2048 byte transfer is appropriate for a single DDC protocol 2 packet (1440 payload bytes); but the speed would not be acceptable for several DDC transfers at higher sample rates (~10Mbyte/s at 1536KHz Fs). However all DDCs are multiplexed into the same DMA stream, so when more DDCs are enabled a larger transfer can be selected. There is a theoretical route to use a hardware initiated DMA controlled from inside the FPGA but that would require very complex IP and a device driver, because the DMA engine needs the hardware address of destination memory and the user application sees it through a memory management unit. Each DDC channel would have a block of DMA descriptors, which were set up by the Raspberry pi in advance. When a FIFO reaches a certain depth it initiates a transfer and the data is DMA’d to processor memory. FIFO depths would be smaller, and there would be less processor overhead.

There are some tutorials on XDMA:

<https://www.hackster.io/Roy_Messinger/pci-express-with-dma-sub-system-241d15>

<https://support.xilinx.com/s/article/71435?language=en_US>

<https://www.xilinx.com/video/technology/getting-the-best-performance-with-dma-for-pci-express.html>

## Register Access

All FPGA registers are 32 bit registers accessed via an AXI4-lite bus. They are accessed using 32 bit reads or writes. Byte or word access is not possible. The registers are listed in section 9.3.

In general all registers can be read or written without side effect, unless otherwise noted. There are a few registers where a read clears a flag (for example FIFO overflow, ADC overflow).

## Sample Data Transfer

### FIFO sizes

There are several FIFOs required – see Figure 18:

A picture containing text, calculator

Description automatically generated

Figure : CPU to DSP FIFOs

In all cases the FIFOs on the CPU side are 64 bits; the data needs to be resized (eg using AXI stream datawidth converters) to match that width.

There 3 cases to consider:

1. RX data: 5 or more parallel RX streams, variable sample rate
2. TX data: 1 I/Q sample stream, 48KHz (protocol 1) or 192 KHz (protocol 2) sample rate
3. Audio Codec data: 48KHz sample rate, fixed size.

Each block RAM can be 4Kx9, 2Kx18 or 1Kx36 size. (There are many more options!)

### Audio Codec FIFO

This is the simplest. The data set is 16 bits mic samples, 16+16 bits L/R speaker samples at fixed 48KHz rate. At that clock rate a 1K deep FIFO holds 21ms of audio data – far more than required. Separate Codec needed for Mic samples (16 bits wide), L/R speaker samples (32 bits wide)

Original design decision: 256x64 FIFO for each of read, write to CODEC. Upgraded to 1024x64 for speaker data after finding the incoming data often backed up waiting for the FIFO to have space.

### TX FIFO

Input data:

* 16 bit I / 16 bit Q samples @ 48KHz Fs (protocol 1)
* 24 bit I / 24 bit Q samples @ 192KHz Fs (protocol 2)

Do the sums for protocol 2; protocol 1 has much lower rate. Work this out for a FIFO width of 4 bytes, and assume that can be read out to unpack the samples.



A reasonable conclusion might be that a 1Kx32 FIFO will be adequate for both protocol 1 and 2 as long as it can be serviced by a new data transfer in <4ms.

Original design decision: 1kx64 FIFO for TX path. Doubled to create more headroom for incoming data, although no specific issues identified.

### RX FIFO

Input 24 bit I / 24 bit Q samples @ variable Fs. several parallel receiver channels merged eventually into a single FIFO.



The RX FIFOs are the largest memory structure in the design. Consider limiting the sample rate on DDC5 and above, and giving it a smaller FIFO if needed. However, this does hinge on the protocol used!

Initial design decision: Saturn will have has 8kx64 FIFO for the interleaved RX DDCs. Overflows occurred with RX1 & RX2 at 1536KHz, so FIFO sized doubled to 16384x64

### FIFO Clearing

FIFOs are reset by assertion of the asynchronous reset signal. They can also be reset under processor control by asserting bits in the FIFO clearing register to zero. This gives the software a way to recover from error conditions.

These bits have been moved to a central register from RX and TX registers to avoid possibility that different software processes set (or forget to set) the bits erroneously.

|  |  |  |
| --- | --- | --- |
| **FIFO Clear Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| FIFOClear[0] | Mic FIFO reset | =0: reset FIFO  =1: normal operation |
| FIFOClear[1] | Spk FIFO reset | =0: reset FIFO  =1: normal operation |
| FIFOClear[2] | DDC FIFO reset | =0: reset FIFO  =1: normal operation |
| FIFOClear[3] | DUC FIFO reset | =0: reset FIFO  =1: normal operation |

## DMA Data Transfer Formatting

This is about the organisation of the data and its organisation so that it can be efficiently transferred between FPGA and Raspberry Pi. Required data transfer rates could reach about 40Mbytes/s; an efficient transfer is important.

### RX DDC Data

#### DDC Operation

For Thetis to work it must be possible reliably to start and stop operation and reconfigure the DDCs and still get deterministic data. The required criterion is: the phase difference between any pair of DDCs used for Puresignal (on TX) or diversity (on receive) must remain constant even if the sample rate is changed[[3]](#footnote-3). To ensure this, the DDC sample rates for any pair of DDCs than needs to be phase related must be changed in the same clock cycle.

An new approach has been developed which multiplexes all data streams from all enabled DDCs into a single FIFO regardless of their sample rate. A custom multiplexer is used. Its strategy is to transfer data per fixed duration “beat” with an initial plan being one beat = one 48KHz sample period. Within a beat, each DDC transfers N samples depending on sample rate (48KHz = 1 sample, 192KHz = 4 samples etc) and each enabled DDC transfer samples in DDC order. The DDC rate register is only transferred to the DDCs at the start of a beat, and the register content is added into the data stream so that software will always know exactly the meaning of the data stream.

At the start of a beat, the DDC rate register (see section 4.3.2) is copied into bits 31:0 of the output stream with the top data bits set to 0x8000: the top bit set is the marker indicating this is the DDC word. The DDC word is used to frame the output stream into data beats. Thereafter N I/Q words are transferred for the enabled DDCs, starting from DDC0, with the top 16 bits of the 64 bit word set to zero. N is worked out from the DDC rate register. Pairs of even/odd numbered DDC with the same sample rate can be interleaved; the data from both is written into the “even” output stream. Non-enabled DDC have the correct number of samples read from their stream. But not written to the FIFO.

The multiplexer has 48 bit AAX stream inputs from DDC, A 64 bit AXI stream output, an enable input and 3 bit codes for each DDC to set the sample rate. When enabled it starts from DDC0 and goes through to DDC9 in turn; and read out N samples where N is the number that the DDC generates in a beat. If enable has been cleared it stops, otherwise it repeats. This means the stream sent to the FIFO is always for a complete data beat covering all DDCs.

If we have DDC0 & 2 enabled at 96KHz, DDC 1 enabled at 192KHz and DDC5 enabled at 48 KHz a single beat will result in sequential outputs:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DDC** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| Enabled | Y | Y | Y | N | N | Y | N | N | N | N |
| Fs (KHz) | 96 | 192 | 96 |  |  | 48 |  |  |  |  |
| Data beat 0 | DDC0n DDC0n+1 | DDC1m DDC1m+1 DDC1m+2 DDC1m+3 | DDC2p DDC2p+1 |  |  | DDC5q |  |  |  |  |
| Data beat 1 | DDC0n+2 DDC0n+3 | DDC1m+4 DDC1m+5 DDC1m+6 DDC1m+7 | DDC2p+2 DDC2p+3 |  |  | DDC5q+1 |  |  |  |  |
| Data beat 2 | DDC0n+4 DDC0n+5 | DDC1m+8 DDC1m+9 DDC1m+10 DDC1m+11 | DDC2p+4 DDC2p+5 |  |  | DDC5q+2 |  |  |  |  |

This leads to an output stream as follows:

<DDC rate register> DDC0n DDC0n+1 DDC1m DDC1m+1 DDC1m+2 DDC1m+3 DDC2p DDC2p+1 DDC5q DDC0n+2 DDC0n+3 DDC1m+4 DDC1m+5 DDC1m+6 DDC1m+6 DDC2p+2 DDC2p+3 DDC5q+1 DDC0n+4 DDC0n+5 DDC1m+8 DDC1m+9 DDC1m+10 DDC1m+11 DDC2p+4 DDC2p+5 DDC5q+1 and so on

The stream is deterministic, as long as no beats are lost. The Raspberry pi software will always be able to find the correct data. The DDC rate register word can be uniquely identified because bit 63=1. If the DDC rate register is updated, the processor does not need to find the correct point in the FIFO to start the new settings: the DDC word will appear in the data stream at the right point.

The multiplexer implements interleaving of pairs of DDCs. If an even DDC has its rate bits set to 111, the rate is taken from the next DDC and sample data from DDC N and DDC N+1 are interleaved. So with DDC4 set to 111 and DDC 5 set to 011, the resulting output data for that pair would be (in sequence):

DDC4n DDC5n DDC4n+1 DDC5n+1 DDC4n+2 DDC5n+2 DDC4n+3 DDC5n+3

Diagram

Description automatically generated

Figure : Alternative DDC to CPU Connection Concept

For the design to work the appropriate number of samples per beat must already be available, requiring a FIFO between DDC and multiplexer. This FIFO needs simply to hold one “data beat” of samples and can be small, eg 64 samples. The output FIFO has the largest data rate; if it fills, then the input FIFOs will begin to fill (because it is a stream).

For non enabled DDC, the multiplexer consumes data so the FIFOs maintain depth alongside the others. All data transfers should be initiated or stopped by setting a single “Active” signal.

The multiplexer isn’t synchronised to time; it is data driven. It works out a “beat” by counting the right number of samples from each source. At the end of a beat, it will have read in the required number of samples for each DDC and also written them out (or be a few clock cycles from that) – there is no persistent storage.

The multiplexer will transfer 48 bit (6 byte) words to a 64 bit FIFO. The additional (top) 16 bits of data will be used as follows:

* Bit 63 = 1: bits 31-0 contain a DDC configuration word; bits 47-32=0
* Bit 63 = 0: bits 47-0 contain an I/Q sample word
* Bits 62-48 are zero, but could be used for debug

This way the DMA process is continuous. It is data driven, and self locates the points where data format changes.

#### DMA Transfer & Processor Operation

With many DDC enabled and high sample rates the output rate could get quite high. BUT the DMA transfer chunks can be larger, and only one DMA needs to be operated to service all DDC. Overall it is likely more efficient in terms of CPU & bus utilisation. Processor loading is implied for P1 and P2 transfers to Thetis, because data needs to be reordered and packed; but the processor has little else to do in those circumstances.

In this design the multiplexer will transfer a DDC rate word, then loop through all enabled DDC and transfer the required number of samples for one data beat. After processing DDC0-9 it will repeat. This way once started the data transfer never needs be stopped, and the input FIFOs can all be small (eg 64 samples). It does mean that if it ever is stopped, data will be lost and the input FIFOs (and probably output FIFO) should be reset.

* To start up:
  + Setup DDC rate register
  + Enable multiplexer
    - DDC o/p FIFOs are reset automatically
  + Start DMA transfer
  + Handle received data: loop through this procedure
    - Read 1st word, check top bit is set.
    - Decode as DDC rate word, establish number of words per DDC
    - Read 1-320 words and transfer to local buffers for each DDC
* To make changes:
  + Change DDC rate register
  + The IP will synchronously change DDC rate settings at the end of a data beat
  + The IP will read and output the configuration before processing all current DDCs
* If an error is detected:
  + Stop multiplexer
    - Multiplexer completes the current set of DDC reads up to & including DDC9
  + DMA out all data from FIFO (or just delete)
  + Enable multiplexer
    - DDC o/p FIFOs are reset automatically
  + Re-start DMA transfer

A spreadsheet (“DDC output FIFO depth calculator.xlsx”) takes DDC configuration to predict output data rate and FIFO capacity in milliseconds. It shows that a 4kx48 FIFO would have a duration of 0.88ms if DDC0,1 at 1536KHz; DDC 2,3 at 192KHz and others all at 192KHz. Seems reasonable.

This does mean that the configuration is sent each beat. That’s a 100% overhead with one DDC enabled at 48KHz, but the processor has little to do in this case. But the overhead gets much smaller as the number of DDCs and/or the sample rates are increased; it also means the PC doesn’t need to search for the word – it will always be in a known location AND it has bits set to identify it for error detection.

It would also be possible to only transfer the DDC config at the start, and when it changes. Then the RaspberryPi would need to check for a new config word. This is not currently implemented.

The DDC multiplexer interleaves DDC N and DDC N+1 data if DDC N has rate inputs set to 111. So the processor does NOT need to worry about interleaving paired DDCs.

#### Example data

With DDC0&1 = 192KHz, DDC2=384KHz, DDC3=48KHz and DDC7=96KHz the output stream for each data “beat” (20.8us) will be (using randomised data):

|  |  |
| --- | --- |
| **output data for a single beat** | |
| 800000001B83FFFF | Rate reg Header |
| 0000EFB8165E22E9 | DDC0 |
| 0000E2CC5C33D1A2 |  |
| 000026BB315B49EA |  |
| 00005148B9A13644 |  |
| 00001ED999A9B06C | DDC1 |
| 00007C83BB2AF529 |  |
| 0000288B1E95D38D |  |
| 00008919B9CCEF73 |  |
| 00006C9AB7EC5211 | DDC2 |
| 00000339A05855F7 |  |
| 0000B5444920AF82 |  |
| 0000F3F9C2FE4E14 |  |
| 0000B7D4BF1EEBA4 |  |
| 0000924C9D9162C3 |  |
| 00006BE5A771F08F |  |
| 0000F4189ECCBE4A |  |
| 0000AC6D0D9E2627 | DDC3 |
| 00006878189F1D42 | DDC7 |
| 00002A6C332867B2 |  |

In this case, one config word per beat adds a 5% overhead.

#### Sample Data format at the Raspberry Pi

(see also section 8.5 for byte ordering)

Samples are transferred as 24 bit I, 24 bit Q complex samples. These appear into the processor memory map, depending on the setting of Byteswap (GPIO bit 26) as follows:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |
| Data | DDC | DDC | DDC | DDC | 0x00 | 0x00 | 0x00 | 0x80 | (rate reg) |
| Byte # | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  |
| Data | I07:0 | I015:8 | I023:16 | Q07:0 | Q015:8 | Q023:16 | 0x00 | 0x00 | (I/Q data row) |
| Byte # | 16 | 17 | 18 | 19 | 20 | 21 | 23 | 23 |  |
| Data | I17:0 | I115:8 | I123:16 | Q17:0 | Q115:8 | Q123:16 | 0x00 | 0x00 | (I/Q data row) |
| Byte # | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |  |
| Data | I27:0 | I215:8 | I223:16 | Q27:0 | Q215:8 | Q223:16 | 0x00 | 0x00 | (I/Q data row) |

Table : RX IQ Data, with Byteswap==0

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |
| Data | DDC | DDC | DDC | DDC | 0x00 | 0x00 | 0x00 | 0x80 | (rate reg) |
| Byte # | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  |
| Data | I023-16 | I015:8 | I07:0 | Q023-16 | Q015:8 | Q07:0 | 0x00 | 0x00 | (I/Q data row) |
| Byte # | 16 | 17 | 18 | 19 | 20 | 21 | 23 | 23 |  |
| Data | I123-16 | I115:8 | I17:0 | Q123-16 | Q115:8 | Q17:0 | 0x00 | 0x00 | (I/Q data row) |
| Byte # | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |  |
| Data | I223-16 | I215:8 | I27:0 | Q223-16 | Q215:8 | Q27:0 | 0x00 | 0x00 | (I/Q data row) |

Table : RX IQ Data, with Byteswap==1

### TX I/Q data

Regardless of protocol I/Q samples are sent to the TX and these are asynchronous to other transfers. The source data is either 16+16 bits @Fs=48KHz (protocol 1) or 24+24 bits @Fs=192KHz (protocol 2). The hardware transfer is always 24 bits, zero padded at the LSBs for protocol 1.

48 bit I/Q pairs are packed into 64 bit words for DMA transfer. The AXI stream deinterleaver IP demultiplexes from 64 to 48 bits, and optionally demultiplexes to I/Q words for each of normal I/Q modulation and envelope output (which isn’t properly supported yet).

For protocol 1 assume that the ARM processor will unpack the I/Q data from the speaker samples.

If envelope data is enabled, then even I/Q samples go to I/Q modulation and odd samples go to envelope.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Data | I07-0 | I015:8 | I023:16 | Q07:0 | Q015:8 | Q023:16 | I17:0 | I115:8 |
| Byte # | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Data | I123:16 | Q17:0 | Q115:8 | Q123:16 | I27:0 | I215:8 | I223:16 | Q27:0 |
| Byte # | 16 | 17 | 18 | 19 | 20 | 21 | 23 | 23 |
| Data | Q215:8 | Q223:16 | I37:0 | I315:8 | I323:16 | Q37:0 | Q315:8 | Q323:16 |

Table : TX I/Q Data, Byteswap == 0

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Data | I023:16 | I015:8 | I07:0 | Q023:16 | Q015:8 | Q07:0 | I123:16 | I115:8 |
| Byte # | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Data | I17:0 | Q123:16 | Q115:8 | Q17:0 | I223:16 | I215:8 | I27:0 | Q223:16 |
| Byte # | 16 | 17 | 18 | 19 | 20 | 21 | 23 | 23 |
| Data | Q215:8 | Q27:0 | I323:16 | I315:8 | I37:0 | Q323:16 | Q315:8 | Q37:0 |

Table : TX I/Q Data, Byteswap == 1

### Codec Microphone Data

Microphone data requires one stream of 16 bit scalar samples (from the codec left ADC) at a sample rate of 48KHz. These will be read by the DMA engine directly. Transfers are 64 bits wide.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte #** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **Swap bit** |
| Data | M07:0 | M015:8 | M17:0 | M115:8 | M27:0 | M215:8 | M37:0 | M315:8 | (Byteswap==0) |
| Data | M015:8 | M07:0 | M115:8 | M17:0 | M215:8 | M27:0 | M315:8 | M37:0 | (Byteswap==1) |

### Codec Speaker Data

Speaker data requires one stream of 16+16 bit Left and Right sample pairs at a sample rate of 48KHz. These will be written by the DMA engine directly. DMA transfers are 64 bits wide.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** | **Swap bit** |
| Data | L07:0 | L015:8 | R07:0 | R015:8 | L17:0 | L115:8 | R17:0 | R115:8 | (Byteswap==0) |
| Data | L015:8 | L07:0 | R015:8 | R07:0 | L115:8 | L17:0 | R115:8 | R17:0 | (Byteswap==1) |

### Data Transfer Software Process

The details will need to be worked out, but each data stream will need to be serviced by a thread in the Raspberry pi. There are two read and two write DMA engines available; these can be statically allocated to I/Q read, I/Q write, codec mic data read, codec speaker data write.

In all cases, each data path has its own 64 bit wide FIFO. The FIFO depth can be read using the FIFO monitor IP. FIFO data is read (DDC, codec mic) by reading from one address; it is written (TX, codec speaker) by writing to one address. It is intended that the DMA transfers will do those writes and reads.

The FIFOs can be reset by deasserting the FIFO reset bit in the appropriate config registers. (DDC input select bit 21; TX config bit 22). The codec FIFOs aren’t resettable, but the data path consistently transfers multiples of 16 bit words so they will always end up on a “safe” boundary if the FIFO over or underflows.

## LED Outputs

Various LED outputs are provided, mostly for debugging. 3.3V logic, LED should connect to ground / Vdd via a suitable resistor. Today these are all software driven, but some could be assigned to hardware functions.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **LED Output Register** | | |
| **Input Bits** | **LED** | **Function** | **Meaning** |
| LED\_Out [15:0] | D44, D22, D41, D40, D35, D10, D11, D14, D12, D13, D15, D16, D17, D18, D19, D74 | =1 to light LED | To be determined. Initially software driven but could be remapped to internal h/w lines. |
| BLINK\_LED | D75 | 1Hz blink | Blinks when FPGA configured, reset not active and all clocks are present |
| PCI\_LINK\_LED | D80 | PCIe | Lit when PCIe interface has been initialised by the operating system |
| Config LEDs | D64, D73 | Lit if successful FPGA config | See section 3.7 |

# Software API

## IP Registers

This section describes the register interface for project-specific IP blocks. The header files for the Verilog modules should describe these too!

### 64 bit Config Register

The 64 bit config register IP presents two 32 bit registers. Each can be read or written; the value written is available as a 32 bit bus for hardware control.

|  |  |
| --- | --- |
| **IP** | AXIL\_ConfigReg\_64 |
| **File** | axil\_config64\_reg.v |
| **Addr space used** | 8 bytes |
| **Register Address** | **Function** |
| 0x0 | 32 bit register 0 |
| 0x4 | 32 bit register 1 |

### 256 bit Config Register

The 64 bit config register IP presents two 32 bit registers. Each can be read or written; the value written is available as a 32 bit bus for hardware control.

|  |  |
| --- | --- |
| **IP** | AXIL\_ConfigReg\_256 |
| **File** | axil\_config256\_reg.v |
| **Addr space used** | 32 bytes |
| **Register Address** | **Function** |
| 0x00 | 32 bit register 0 |
| 0x04 | 32 bit register 1 |
| 0x08 | 32 bit register 2 |
| 0x0c | 32 bit register 3 |
| 0x10 | 32 bit register 4 |
| 0x14 | 32 bit register 5 |
| 0x18 | 32 bit register 6 |
| 0x1C | 32 bit register 7 |

### Status Readback Register

The 64 bit config register IP presents two 32 bit registers. Each can be read or written; writes are ignored. The value read back is that presented on a 32 bit input bus.

|  |  |
| --- | --- |
| **IP** | AXIL\_ReadReg\_64 |
| **File** | axil\_read64\_reg.v |
| **Addr space used** | 8 bytes |
| **Register Address** | **Function** |
| 0x0 | 32 bit register 0 |
| 0x4 | 32 bit register 1 |

### FIFO Monitor

This IP monitors 4 FIFOs. It provides a way to read the FIFO depth. It latches the FIFO overflow flag, and latches over or underflow from the FIFO depth reaching the programmed depth or zero respectively. These latched values are held until the FIFO’s status register is read, at which point they are cleared.

|  |  |
| --- | --- |
| **IP** | FIFO\_Monitor |
| **File** | FIFO\_Monitor.v |
| **Addr space used** | 32 bytes |
| **Register Address** | **Function** |
| 0x00 | Status register 1 (read only, with side effect) |
| 0x04 | Status register 2 (read only, with side effect) |
| 0x08 | Status register 3 (read only, with side effect) |
| 0x0C | Status register 4 (read only, with side effect) |
| Each Status reg: | bit(15:0) Current FIFO Depth  bit 31 1 if an overflow has occurred. Cleared by read.  Bit 30 1 if an overflow occurred, measured from depth value  Bit 29 1 if an underflow occurred, measured from depth value  Bits 31-29 cleared by read. |
| 0x10 | Control register 1 (read/write, with no read side effect) |
| 0x14 | Control register 2 (read/write, with no read side effect) |
| 0x18 | Control register 3 (read/write, with no read side effect) |
| 0x1C | Control register 4 (read/write, with no read side effect) |
| Each control reg: | bit(15:0) Threshold FIFO depth  bit 31 Interrupt enable |

The 4 channels are allocated as follows:

|  |  |  |
| --- | --- | --- |
| **Channel** | **FIFO** | **FIFO size** |
| 1 | RX DDC I/Q | 8192x64 |
| 2 | TX DUC I/Q | 1024x64 |
| 3 | Codec mic samples | 256x64 |
| 4 | Codec speaker samples | 256x64 |

All FIFOs are transferred at 8 bytes (64 bits) wide

Mic & DDC FIFO – rd\_data\_count connected to FIFO monitor IP

Spk & DUC FIFO – wr\_data\_count connected to FIFO monitor IP

Both words read out the 7 value: with 10 items in the FIFO they will read out a value of 10. They differ only in that they may be in different clock domains.

### ADC Overflow Register, FIFO overflow registers

A single AXI-lite read register that passes latched overrange readings for two ADCs, or several FIFOs. Any overrange/overflow is latched and stored until the register is read; a read operation clears the latch.

|  |  |
| --- | --- |
| **IP** | AXI\_FIFO\_overflow\_reader |
| **File** | AXI\_FIFO\_overflow\_latch\_reader.v |
| **Addr space used** | 4 bytes |
| **Register Address** | **Function** |
| 0x0 | bit 0: =1 if ADC1 overrange has occurred  bit 1: =1 if ADC2 overrange has occurred  side effect: read clears both bits |

|  |  |
| --- | --- |
| **IP** | AXI\_FIFO\_overflow\_reader |
| **File** | AXI\_FIFO\_overflow\_latch\_reader.v |
| **Addr space used** | 4 bytes |
| **Register Address** | **Function** |
| 0x0 | bit 0: =1 if DDC FIFO1 overflow has occurred  bit 1: =1 if DDC FIFO2 overflow has occurred  bit 2: =1 if DDC FIFO3 overflow has occurred  bit 3: =1 if DDC FIFO4 overflow has occurred  bit 4: =1 if DDCFIFO5 overflow has occurred  bit 5: =1 if DDC FIFO6 overflow has occurred  bit 6: =1 if DDC FIFO7 overflow has occurred  bit 7: =1 if DDC FIFO8 overflow has occurred  bit 8: =1 if DDC FIFO9 overflow has occurred  bit 9: =1 if DDC FIFO10 overflow has occurred  side effect: read clears both bits |

(Note that this instance checks small FIFOs before the main data multiplexer, not the main data FIFO)

### SPI ADC Reader Registers

This IP reads an SPI A-D converter (78H90) periodically and stores the data is AXI readable registers. The VSWR forward/reverse power readings store the peak value; the peak reading is cleared when the register is read.

For avoidance of doubt the mappings to protocol 2 bytes are shown, as which user ADC is which isn’t very clear!

|  |  |  |
| --- | --- | --- |
| **IP** | AXI\_SPI\_ADC |  |
| **File** | axi\_spi\_adc.v |  |
| **Addr space used** | 32 bytes |  |
| **Register Address** | **Function** | **Protocol 2 High priority status bytes (starting at 0)** |
| 0x00 | [11:0] AIN1 reading; has peak hold (Fwd\_power) | 14, 15 |
| 0x04 | [11:0] AIN2 reading; has peak hold (Rev\_power) | 22, 23 |
| 0x08 | [11:0] AIN3 reading (J16 pin 12; 7000DLE PA voltage) | 57, 58 (user\_ADC0) |
| 0x0C | [11:0] AIN4 reading (J16 pin 11, 7000DLE PA current) | 55, 56 (user\_ADC1) |
| 0x10 | [11:0] AIN5 reading (Exciter\_power) | 6, 7 |
| 0x14 | [11:0] AIN6 reading (13.8V monitor) | 49, 50 (supply volts) |
| 0x18 | [11:0] AIN7 reading (TX driver PA current) |  |
| 0x1C | Reads AIN1 |  |

#### PA Drive Current

PA Drive current scaling: there is a 0.04R resistor in the drain supply to the amplifiers, and the voltage drop sensor (INA199A1RSWR) has a gain of 50. ADC full scale reading = 5V.

ADC reading = 1638.4I (I in Amps) or current = ADC reading/1638.4

100mA bias recommended; that’s a reading of 0xA4 (200mA gives 0x 148)

To make the reading: write 0x0 to TX config (0x2008); write 0x03000000 to GPIO (0x2014)

#### 7000DLE RF PA Voltage

PA voltage = 0.0256 \* ADC reading (0 to 4095)

#### 7000DLE RF PA Current

This can be used for bias current measurement. To work out the scaling we need to know if U12 on the RF board is an ACS713ELCTR-20A-T or ACS713ELCTR-30A-T.

If -30 device: PA current = ADC reading \* 0.01387

If -20 device: PA current = ADC reading \* 0.00996 (it is this)

### Alex SPI Registers

This IP writes data from the AXI4-Lite bus to the Alex registers for RX and TX RF control. Data is shifted following a change in the data presented.

|  |  |
| --- | --- |
| **IP** | AXILite\_Alex\_SPI |
| **File** | AXILite\_Alex\_SPI.v |
| **Addr space used** | 8 bytes |
| **Register Address** | **Function** |
| 0x00 | 16 bit TX SPI data  After shift, data should be latched by a rising edge on Strobe\_0  For Alex RF interface: see section 7.1 for data bits |
| 0x04 | 32 bit RX SPI data  After shift, data should be latched by a rising edge on Strobe\_1  For Alex RF interface: see section 7.1 for data bits |

### Codec SPI Registers

This IP writes data from the AXI4-Lite bus to the Codec “processor access” registers. Data is shifted following a processor write operation. If there are subsequent write transactions while data is being shifted, the bus will stall until the shifter is idle and then the write will complete. Thus there is no data loss, but check how long a stall is allowed before a bus error is declared.

The SPI clock rate is a programmable parameter, expressed as a clock divide. The core is clocked at 122.88MHz; a clock divide of 6 gives a core clock rate or around 20MHz and SPI clock around 10MHz.

|  |  |
| --- | --- |
| **IP** | AXIL\_SPIWriter |
| **File** | axil\_SPIWriter.v |
| **Addr space used** | 16 bytes |
| **Register Address** | **Function** |
| 0x00 | 16 bit SPI data  After shift, data should be latched by a rising edge on Strobe\_0  For Codec: see codec datasheet |
| 0x04 | 32 bit SPI data  After shift, data should be latched by a rising edge on Strobe\_1  For Codec: do not use |
| 0x08 | Status register  Bit0: =1 if data is being shifted; =0 if idle. |

### Product & Version ID registers

A 64 bit read register provides version number design time constants and 4 bits readback of the clock monitor status.

The code in version.c should match the version numbers as it expands the integer to a user string

|  |  |  |
| --- | --- | --- |
| **Address** | **Bits** | **Function** |
| 0x0C000 | 31:20 | Software ID  This is to identify the software build on a specific product  0=invalid  1=1st prototype, board test with no DSP  2=1st prototype/production, Saturn DSP  3=fallback configuration  4=full function saturn |
| 0x0C000 | 19:4 | Software Version |
| 0x0C000 | 3 | Clock monitor =1 if 122MHz clock present (duplicates bit 0) |
| 0x0C000 | 2 | Clock monitor: =1 if 122MHz EMC clock present |
| 0x0C000 | 1 | Clock monitor: =1 if 10MHz ext reference clock present |
| 0x0C000 | 0 | Clock monitor: =1 if 122MHz clock present |
| 0x0C004 | 31:16 | Product ID  This is to identify the board on which the FPGA runs  0=invalid  1= Saturn |
| 0x0C004 | 15:0 | Product Version  1=1st prototype  2 = 2nd prototype/production V1 |

## AXI4 DMA Bus Address Map

Each AXI stream reader/writer supports one RX stream and one TX stream. The IP is designed so that a DMA (with linearly incrementing address) maps to the port but the address isn’t used: the data is written to / read from the one FIFO for all address matching the IP address range. This means that the maximum size of DMA is 256Kbyte, beyond which data would go to the wring IP core or a bus error would occur. It will be the case that the FIFO size isn’t sufficient to cause that to be a problem!

There is a 1Mbyte PCI BAR window. Any data transfers to addresses higher than 0x7FFFF will result in a non recoverable bus error.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IP Device** | **Streams supported** | **Address Start** | **Address End** | **Size** | **FIFO monitor port** |
| Reader/writer 0 | RX DDC, TX DDC | 0x00000 | 0x3FFFF | 256K | RX: Mon 0 i/p 0  TX: Mon 0 i/p 1 |
| Reader/writer 1 | Codec speaker, mic audio | 0x40000 | 0x7FFFF | 256K | mic: Mon 0 i/p 2  spk: Mon 0 i/p 3 |

## AXI4-Lite Register Bus Address map

There are a lot of I/O registers! These can be read/write accessed from the Raspberry Pi easily via the device driver. I’ve created AXI4-Lite IP to provide 64 bits of config data (2x32 bit words) and another with 256 bits of config data (8x32 bit words).

Byte addresses given as an offset address within the AXI4-lite BAR. All addresses are byte addresses, but the bus only accepts 32 bit accesses with an address step of 4. Access to unmapped addresses will result in a non recoverable bus error.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IP** | **Byte Address** | **Addr**  **Width** | **Register** |  | **Reference** |
| Config256\_RX\_0 | 0x00000 | 4K | DDC0 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x00004 | 4K | DDC1 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x00008 | 4K | DDC2 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x0000C | 4K | DDC3 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x00010 | 4K | DDC4 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x00014 | 4K | DDC5 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x00018 | 4K | DDC6 Frequency |  | See 4.3.2 |
| Config256\_RX\_0 | 0x0001C | 4K | DDC7 Frequency |  | See 4.3.2 |
| Config256\_RX\_1 | 0x01000 | 4K | DDC8 Frequency |  | See 4.3.2 |
| Config256\_RX\_1 | 0x01004 | 4K | DDC9 Frequency |  | See 4.3.2 |
| Config256\_RX\_1 | 0x01008 | 4K | RX Test DDS Frequency |  | See 4.3.2 |
| Config256\_RX\_1 | 0x0100C | 4K | DDC Rate register |  | See 4.3.2 |
| Config256\_RX\_1 | 0x01010 | 4K | DDC Input Select Register |  | See 4.3.2 |
| Config256\_RX\_1 | 0x01014 | 4K | Unused |  |  |
| Config256\_RX\_1 | 0x01018 | 4K | Unused |  |  |
| Config256\_RX\_1 | 0x0101C | 4K | Unused |  |  |
| Config256\_2 | 0x02000 | 4K | KeyerConfig |  | See section 6 |
| Config256\_2 | 0x02004 | 4K | CodecConfig |  | See section 4.5.3 |
| Config256\_2 | 0x02008 | 4K | TXConfig |  | See section 4.4.2 |
| Config256\_2 | 0x0200C | 4K | TXFrequency |  | See section 4.4.2 |
| Config256\_2 | 0x02010 | 4K | TX Modulation test source |  | See section 4.4.2 |
| Config256\_2 | 0x02014 | 4K | RF GPIO |  | See section 7.4 |
| Config256\_2 | 0x02018 | 4K | ADC\_Ctrl |  | See section 7.2 |
| Config256\_2 | 0x0201C | 4K | DAC\_Ctrl |  | See section 7.3 |
| ConfigReg\_64\_0 | 0x03000 | 4K | Processor LED |  | See section 8.5 |
| ReadReg\_64\_0 | 0x04000 | 4K | Status |  | See section 7.5 |
| ReadReg\_64\_0 | 0x04004 | 4K | Date code Register |  | See section 7.5 |
| Axi\_FIFO\_overflow\_0 | 0x05000 | 4K | ADC Overflow | ADC1, 2 latched overflow bits | See 9.1.5 |
| Receiver/ Axi\_FIFO\_overflow\_0 | 0x06000 | 4K | FIFO Overflow | DDC mux input FIFO overflow bits | See 9.1.5 |
| ConfigReg\_64\_0 | 0x07000 | 4K | FIFO Clear | DDC, DUC, Mic & Spk FIFO reset bits | See section 8.3.5 |
| ConfigReg\_64\_0 | 0x07004 | 4K | IambicConfig |  | See section 6.2 |
| Fifo\_Monitor\_0 | 0x09000 | 4K | FIFO monitor | RX DDC FIFO, TX DUC FIFO, Codec RX FIFO, Codex TX FIFO | See 9.1.4 |
| AXI\_SPI\_ADC\_0 | 0x0A000 | 4K | SPI ADC reader | Alex analogue inputs | See 9.1.6 |
| Axilite\_Alex\_SPI\_0 | 0x0B000 | 4k | AXILite\_Alex\_SPI | SPI interface to Alex | See 9.1.7 |
| ReadReg\_64\_ID | 0x0C000 | 4K | ID1 | Version ID (31:16)  Revision (15:4)  Clock monitor bits(3:0) | See 9.1.9 |
| ReadReg64\_ID | 0x0C004 | 4K | ID2 | Product ID (31:16)  S/W ID (15:0) | See 9.1.9 |
| Axi\_quad\_spi\_0 | 0x10000 | 16K | SPI Config Prom | Xilinx SPI interface |  |
| Axil\_SPIWriter\_0 | 0x14000 | 16K | SPI codec bus | Codec register access | See 9.1.8 |
| Axadwiz\_0 | 0x18000 | 16K | On-chip XADC | Xilinx XADC interface | See Xilinx PG091 |
| Axi\_BRAM\_ctrl\_0 | 0x1C000 | 16K | AXI block RAM access | CW keyer ramp waveform |  |

## XADC registers

The XADC block is set to measure 4 analogue values:

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Current reading Register | Min Reading Register | Max Reading Register |
| Temperature | 0x18200 | 0x18290 | 0x18280 |
| VCCINT | 0x18204 | 0x18294 | 0x18284 |
| VCCAUX | 0x18208 | 0x18298 | 0x18288 |
| VCCBRAM | 0x18218 | 0x1829C | 0x1828C |

All values are 12 bits in bit positions 15-4, zero padded in bits 3-0; so treat as a 16 bit read.

To scale to meaningful measurements, according to UG480 p23:

Temperature (Celsius) = (16 bit ADC reading \* 503.975)/65536 – 273.15

Voltage = 16 bit ADC reading \* 3.0)/65536

# IP Modules used in Design

## Xilinx IP

|  |  |  |
| --- | --- | --- |
| **IP** | **Used For** | **Documentation** |
| DMA/Bridge Subsystem for PCI Express | PCI express interface, with DMA and individual read/write capability | PG195 |
| AXI IIC Bus Interface  v2.0 | I2C interface to Codec (IIC is the same thing) | PG090 |
| DDS |  |  |
| FIR |  |  |
| CIC |  |  |

## Local Verilog Modules

Verilog modules used in the design:

|  |  |  |
| --- | --- | --- |
| **Module** | **File** | **Description** |
| Usr\_Reg\_Access | usr\_reg\_access.v | Wrapper around Xilinx IP to access the user register, whose content is specified in the constraints file when the bitstream is generated. Used to provide a version number. |
| reg\_to\_axis | reg\_to\_axis.v | Takes parallel input and adds a permanently asserted valid signal. |
| double\_D\_register | double\_register.v | Two cascaded D registers, for example to synchronise asynchronous inputs |
| D\_register | register.v | A simple D register |
| axis\_mux\_2 | axis\_mux\_2.v | Simple 4:1 selector for AXI stream inputs. |
| axis\_mux\_4 | axis\_mux\_4.v | Simple 4:1 selector for AXI stream inputs. |
| regmux\_2\_1 | regmux\_2\_1.v | Simple 2:1 data selector |
| regmux\_4\_1 | regmux\_4\_1.v | Simple 4:1 data selector |
| regmux\_8\_1 | regmux\_8\_1.v | Simple 8:1 data selector |
| axis\_variable | axis\_variable.v | Takes a parallel input and asserts its Valid output for one cycle if the data changes. |
| axis\_constant | axis\_constant.v | Takes a parallel input and appends a permanently asserted valid output. |
| axis\_adder | axis\_adder.v | Simple signed or unsigned adder for two AXI streams. |
| Axis\_multiplier |  | Signed pipelined axi stream multiplier |
| i2s\_clk\_lrclk\_gen | i2s\_clk\_lrclk\_gen.v | Phil Harman/Kirk Weedman code – generate a divided clock for the Codec |
| I2S\_xmit | I2S\_xmit.v | Kirk Weedman code for codec TX data, modified Laurence Barker to present an AXI Stream data interface |
| I2S\_rcv | I2S\_rcv.v | Kirk Weedman code for codec RX data, modified Laurence Barker to present an AXI Stream data interface |
| cw\_key\_ramp | cw\_key\_ramp.v | CW keyer to generate sidetone and ramp up/down the amplitude as key pressed/released |
| debounce | debounce.v | Asynchronous input debounce. Used for PTT, key and other external inputs. |
| ClockDivider | clockdivider.v | N:1 clock divider. Still used in input debouncing. |
| PWM\_DAC | pwm\_dac.v | Generate a PWM pseudo-DAV output for an 8 bit DAC. May need to be parameterizable to a different width. |
| Serial\_Atten | attenuator.v | Phil Harman’s code – serial data shift for Minicircuits DAT-33-SP+ attenuator |
| LTC2208\_derandomise | ltc2208\_derand.v | Removes the “randomised” data option for the LTC2208 ADC input, to give 2’s complement data |
| cvt\_offsetbinary | cvt\_offsetbinary.v | Takes 2’s complement data and provides offset binary data for MAX5891 TX RF DAC |
| FIFO\_overflow\_reader | FIFO\_overflow\_latch\_reader.v | AXI4-Lite bus reader for ADC overrange or FIFO overflow signals. Latches and holds the overrange indication until read. |
| AXI\_Stream\_Reader\_Writer | Stream\_reader\_writer.v | AXI stream interface to full AXI-4 bus. Provides a burst-capable data transfer from the processor / DMA bus to and from an AXI4-Stream to connect FIFOs. |
| AXI\_SPI\_ADC | axi\_spi\_adc.v | 78H90 SPI ADC data reader, modified by Laurence Barker to add an AXI-Lite bus interface |
| FIFO\_Monitor |  | Overflow and depth monitor for FIFO; AXI4-Lite interface |
| AXIL\_SPIWriter | Axil\_SPIWriter.v | Axi-4 lite bus interface; dual SPI 9one 16 bit, one 32 bit) serial shifter. Designed for CODEC bus interface. |
| AXILite\_Alex\_SPI | AXILite\_Alex\_SPI.v | Axi-4 lite bus interface and SPI shifter from Orion. |
| AXIS\_Sizer\_48to64 | axi\_stream\_resizer.v | Resizes an axi stream from 48 to 64 bits; resettable. RX DDC datapath. |
| AXIS\_Sizer\_64to48 | axi\_stream\_resizer\_64to48.v | Resizes an axi stream from 64 to 48 bits; resettable. TX DUC datapath. |
| AXIS\_Interleaver | axi\_stream\_interleaver.v | Either passes separately or interleaves two AXI4 streams, 48 bits wide. To interleave pairs of DDC eg DDC0/1. |
| AXIL\_ConfigReg\_64 | axil\_config64\_reg.v | 64 bit config write register, with axi4-lite interface. |
| AXIL\_ConfigReg\_256 | axil\_config256\_reg.v | 256 bit config write register, with axi4-lite interface. |
| AXIL\_ReadReg\_64 | axil\_read64\_reg.v | 64 bit status read register, with axi4-lite interface. |
| AXIS\_DDC\_Multiplexer | DDCMux.v | Protocol-aware AXI stream data multiplexer. Multiplexes sample data into a single stream for the enabled DDCs, with data embedded in the stream to indicate the boundaries between one data beat and the next. |

There are several testbenches to test some of the IP modules.

# Packaging the Project

This is about getting the FPGA source design into Github.

The approach I have used is:

1. Begin with a simple folder (in my case E:\xilinxdesigns\Pluto)
2. It holds just a few files:
3. Constraints folder – holds the 3 constraints files
4. Sources folder with 3 subfolders:
   1. sources\coefficientfiles – files (generated by spreadsheets) with filter coefficients and keyer waveshape
   2. sources\verilogmodules – HDL sources for the Verilog needed
   3. sources\wrapper – the HDL wrapper which is not automatically managed by Vivado.
5. create\_pluto\_project.tcl: this is a TCL file which reconstructs the project and its block diagram
6. The various git files. .gitignore includes the folder “pluto\_project”

The consequence of this is that git manages the sources files and the TCL script; it does not store all the reconstructed project files. To reconstruct follow the guidance in readme.md:

To use this repository:

1. Install vivado 2020.2

2. Copy this repository to c:\xilinxdesigns\pluto

3. Open vivado and find the TCL command line

4. type: cd c:/xilinxdesigns/pluto

5. type: source create\_pluto\_project.tcl

As the design evolves:

1. create any new Verilog sources in the “sources” folder
2. Periodically recreate the TCL script using the command **File > Project > Write TCL**…
3. Tick “recreate block designs using TCL”
4. Select the “create\_pluto\_project.tcl” file
5. Press OK
6. Publish changes to github



# Linux DMA Device Driver

The XDMA IP core has a Xilinx supplied device driver. See AR65444. Unfortunately it isn’t as simple as it could be. There is a folder missing (/etc/udev/rules.d) and you can get it here: <https://github.com/ramonaoptics/xilinx-dma-driver>

## Building & Patching The Module

The build process requires the files for building kernel modules. The simple way to get them is

**sudo apt install raspberrypi-kernel-headers**

but the documentation says the file could be out of date by several weeks; if the kernel build is recent, you may have to rebuild the kernel from scratch to get them. See <https://www.raspberrypi.org/documentation/linux/kernel/headers.md> and if necessary follow the “build section” link.

The newest code published by Xilinx (with those /etc files added) compiles OK on the raspberry pi. However the /dev/xdma0\_user access to axi4-lite bus does not work. Function bridge\_mmap() in file xdma\_cdev.c maps the memory segment: but pci\_resource\_start (around line 196) returns a 64 bit number which is stored into a 32 bit value. Resize the 4 local variables to uint64\_t and it works correctly.

sudo ./load\_driver.sh runs OK but doesn’t actually need running as the module loads automatically.

./perform\_hwcount.sh runs, and appears to report success

./dma\_memory\_mapped\_test.sh 1024 16 1 1 runs and reports success

./dma\_streaming\_test.sh 1024 16 1 1 does not run (but there are no streaming DMA channels)

In /dev I how have:

* /dev/xdma/card0
* /dev/xdma0\_c2h\_0
* /dev/xdma0\_h2c\_0
* /dev/xdma0\_control
* /dev/xdma0\_user (axi4-lite bus)
* /dev/xdma0\_xvc
* 16 more drivers: /dev/xdma0\_eventsn (/dev/xdma0\_events0 to /dev/xdma0\_events15)

See instructions in the AR65444 document:

Here is an example of how to read from the bypass channel at a specified offset (0x0000).

$Linux> ./reg\_rw /dev/xdma0\_bypass 0x0000 w

Here is an example of how to write to the bypass channel at a specified offset (0x0000) with specific data (0x1234567): $Linux> ./reg\_rw /dev/xdma0\_bypass 0x0000 w 0x1234567

Application program ‘reg\_rw’ has 32Kbytes allocated space as default. If ‘PCIe to AXI Lite Master’ or ‘PCIe to DMA Bypass’ interface selected size is less than 32Kbytes and try to use ‘reg\_rw’ application for read/write will produce an error. If selected size is less than 32Kbytes modify this define in ‘reg\_rw.c’ to corresponding value and compile (make) the file. E.g. #define MAP\_SIZE (128\*1024UL)

# Raspberry Pi Issues

## LCD DSI port

The “normal” Raspberry Pi boards have a 15 pin, 2 lane DSI connector for the LCD. The CM4 has a 22 pin connector; you can get an adapter or a special flexi PCB with 15 pin at one end and 22 pin at the other.

Need to provide 5V power to the display. Although there is 5V on the flexi PCB, the display does not take power from it. There is a 5 pin 0.1” pitch connector on the display itself with this pinout:

GND; SCL; SDA; (cut off, as an index pin); 5V

The CM4 has 2 display connectors, DISP0 and DISP1. DISP1 is the default, with 4 DSI lanes. There is a documented process to download an overlay (/boot/dt\_blob.bin) for the device tree for the display. As downloaded this drives DISP1. To make it drive DISP0, make these edits in the CM4 section:

DISPLAY\_I2C\_PORT set to <0> (default=0, so no change)

DISPLAY\_DSI\_PORT set to <0> (default=1)

DISPLAY\_SCL set to <1> (default=45)

DISPLAY\_SDA set to <0> (default = 44)

## IDE

Microsoft Visual Studio Code has proven to be a usable IDE on the Raspberry pi

## Simple Apps

I have created simple apps for programming a serial configuration prom and for reads and writes to/from the AXI4-Lite register bus.

### Flashwriter

Graphical user interface, application

Description automatically generated

This is an application to program the FPGA configuration memory from a BIT file. Usage:

|  |  |
| --- | --- |
| Open File | Opens a dialog to select the file to be programmed into the memory,  The only format accepted is a .BIN file (a Xilinx binary format) |
| Boot Image | Selects which image to reporogram  Primary (this is the normal image to program)  Fallback (CAUTION: only reprogram if absolutely necessary) |
| Program | When pressed initiated programming; this may take 2 minutes or so. The steps are displayed and progress for each step is shown in the bar display. |
| Close | Close the application. |

### AXI ReaderWriter

Graphical user interface, text, application

Description automatically generated

This is a simple application for register reading and writing. It is intended for development, not for normal use and it is very possible to crash the raspberry pi by incorrect usage!

|  |  |
| --- | --- |
| Address | Hexadecimal address to be accessed. Should be in the range 0 to 0x1FFC, in steps of 4 |
| Data | Hexadecimal data to be written, or read from a register. |
| Write | Causes the data value entered to be written into the register with specified address |
| Read | Reads data from the register at the specified address. |
| Close | Causes the data value entered to be written into the register with specified address |

Note that if the address does not correspond to a valid register, this may hang the computer!

### Audiotest

A screenshot of a computer

Description automatically generated

This is a simple application for checking audio in and out; for example to test if a new headset or microphone is working.

|  |  |
| --- | --- |
| Test tone L, R | Plays a test tone with selectable volume while the button is pressed into the Left or Right audio channels. |
| Mic Test | Enables recording for the (adjustable) duration followed by playback of the recording. A progress bar shows how much time has elapsed, and the current operation (record/replay/idle) is indicated.  The mic level bar shows a linear representation of signal level, with max signal at the right. |
| XLR Input | If checked, selects the XLR input instead of the 3.5mm jack. |
| Mic Bias | If selected, applies bias to the 3.5mm mic input. |
| Mic Boost | If checked, increases the audio gain of the Codec by 20dB. Typically needed for dynamic microphones and not for electret ones. |
| Mic on Jack Tip | Selects the microphone signal to be on the tip of the 3.5mm jack. |
| Close | Close the application. |

### Biascheck

A screenshot of a computer

Description automatically generated

This is a simple application for setting up driver and PA bias currents. It puts the radio into transmit with no signal present, so the potentiometers to adjust bias can be set. **This is not intended for user operation.**

|  |  |
| --- | --- |
| Enter TX | When checked, the radio enters TX with no signal present. Any current in the driver stage or PA is purely the bias current/ |
| Driver Current | Shows the current into the driver stage on the Saturn board. Each potentiometer should be fully anticlockwise initially; advance the 1st to obtain 0.1A reading then advance the second until 0.2A is displayed. |
| PA Current | Shows the current into the PA stage on the RF board. Each potentiometer should be fully anticlockwise initially; advance the 1st to obtain 1.0A reading then advance the second until 2.0A is displayed. |
| Close | Close the application. |

## Power Management

The Raspberry Pi will need to be properly shut down to avoid file system corruption. That can be done by “shutdown” for a system with a KWM and UI. However a “headless” system will just haver an on/off switch.

Suggested way forward (from Rick): put pi into a read-only filesystem mode.

<https://learn.adafruit.com/read-only-raspberry-pi>

When one wants to update or install new software the user can type a one line command:

sudo mount -o remount,rw /boot

Then ftp files to the right location.

## Protocol 2 App

“p2app” is a server program that interfaces Saturn as a protocol 2 based radio. On the ethernet side it sends and receives UDP packets as per the protocol 2 spec; on the radio side it reads and writes registers, and DMA transfers I/Q and audio data to and from the hardware.

Other than debug it has no user interface.

To run for normal use:

./p2app

Or it can be run as ./p2app <optional arguments>

Where <optional arguments> can be:

-f <freq in Hz> Turns on DDC test frequency source for development

-i saturn Identifies board as “Saturn” (default)

-i orionmk2 Identified board as “Orion mk 2”

-m xlr selects XLR microphone input

-m jack selects 3.5mm microphone jack (default)

-s skip checking for “x” button to exit (used when run at startup)

To exit: type x<enter>

## Auto-loading p2app at Startup

There is a recommended way to make applications load automatically at startup.

Use your favourite text editor to edit the /etc/rc.local file so it comes out like this:

pi@raspberrypi:~ $ **sudo geany /etc/rc.local**   
#!/bin/sh -e  
#  
# rc.local  
#  
# This script is executed at the end of each multiuser runlevel.  
# Make sure that the script will "exit 0" on success or any other  
# value on error.  
#  
# In order to enable or disable this script just change the execution  
# bits.  
#  
# By default this script does nothing.  
  
# Print the IP address  
\_IP=$(hostname -I) || true  
if [ "$\_IP" ]; then  
  printf "My IP address is %s\n" "$\_IP"  
fi  
  
cd /home/pi/github/Saturn/sw\_projects/P2\_app  
./p2app &  
  
exit 0

The save and exit; p2app will auto-run at startup

If you need to terminate the auto-running app you need to find its process id, then kill it

pi@raspberrypi:~ $ **ps-ax | grep p2app**

741 ? Sl 30.00 ./p2app

pi@raspberrypi:~ $ **sudo kill 741**

# Useful Information

Phil Harman provided this post about PWM: [[hpsdr] EER Support in PowerSDR and Hermes (openhpsdr.org)](http://lists.openhpsdr.org/pipermail/hpsdr-openhpsdr.org/2014-June/045625.html)

# Notes

* I removed 2 strobe signals from the original Andromeda design:
  + Puresignal\_Enable (which should never have been a strobe – it is used in Orion purely to choose which frequency is used in DDC4)
  + Bias\_Ctrl(which was never an Orion strobe);
* TXRX\_Relay is retained as a strobe, to drive an LED when TX is asserted & enabled
* P2app: Temporarily at least removed a line of code responding to byte 1400 in the received high priority message. That bit is set; it is decoded by Thetis from “user digital input 1”. Bit is set if input 1 was 0

1. Source: Xilinx AR# 51017 [↑](#footnote-ref-1)
2. There appears to be no constraint between GTP supplies and VCCAUX & VCCO: so they could be powering up while MGTAVCC and MGTAVTT power up. [↑](#footnote-ref-2)
3. This is because Thetis, today, will reconfigure DDC0 & 1 between RX and TX because only DDC0&1 are paired & interleaved. A change to Thetis explicitly to support Saturn would stop this being an issue [↑](#footnote-ref-3)