LUND UNIVERSITY

ETIN35 – IC Project 1 Report

UDP readout system for the physics department

Dimitar Dikov

2/22/2017

Table of Contents

I. Assignment: Readout system ... 2
a. Assignment specification: ... 2
b. System Implementation: ... 2
II. Implementation of the Serial Readout Unit (SRU): ... 3
a. CMD/STATUS UDP Ethernet EMAC and UDP command/status payload. ... 4
b. UDP command distribution ... 7
c. DCS command decoder ... 8
d. DTC TX ... 10
e. DTC RX (Deserializer and Memory) ... 11
f. UDP Readout FSM and UDP EMAC. ... 12
g. Clock Generation ... 13
III. Implementation of the CPLD: ... 14
a. RDOClk and ADC clock ... 15
b. Trigger Detection ... 15
c. DTC RX Decoder ... 16
d. Address/Data demux and Reply packer. ... 17
e. Saltro Interface Controller ... 19
f. DTC TX FSM and Serializers ... 21
IV. ASIC Synthesis & Place and Route implementation ... 24
a. Synthesis constraints used ... 24
b. Place and Route ... 24
V. FPGA & CPLD Synthesis. ... 25
VI. Implementation Results & Verification ... 26
VII. Conclusion ... 27
VIII. Literature ... 27
Appendix 1: Synthesis script ... 28
Appendix 2: Place and route script ... 29

I. Assignment: Readout system

a. Assignment specification:

A prototype detector is to be developed that contains one SALTRO ADC chip and a CPLD Lattice LCMX02. The communication between the CPLD and SALTRO is described in [1]. The CPLD will need to communicate via Data Trigger Control (DTC) link [2] using Cat. 6 twisted pair cables with RJ45 connectors to a Scalable Readout Unit (SRU). The SRU consist of a Xilinx Virtex-6 FPGA and it needs to convert the DTC link data to UDP packages. The task of this project is to develop the logic in the CPLD and modify the existing FPGA RTL, including readout protocols, data formatting, system controller, and clock management.

b. System Implementation:

The system contains 1 CPLD (Lattice MACHXO 2) and 1 FPGA (Xilinx Virtex-6 130T). The testbench implementation of the system is shown on fig. 1. It consists of 5 main blocks:

- Saltro Emulation model – An emulation model of the digital part of the SALTRO ADC chip.
- Interposers – Delay blocks emulating a transmission delay over copper cables.
- CPLD – Provides decoding of DTC commands to the SALTRO chip and data serialization when reading out the channels.
- SRU (Serial Readout Unit) – Generates trigger sequence for the CPLD/SALTRO, encodes DTC commands and packages the readout data in UDP packets.
- Ethernet Simulation IP – Ethernet IP generated using Xilinx Custom IP Generator. Reads UDP packets in raw format and sends them to the SRU, in 8b/10b encoded format.

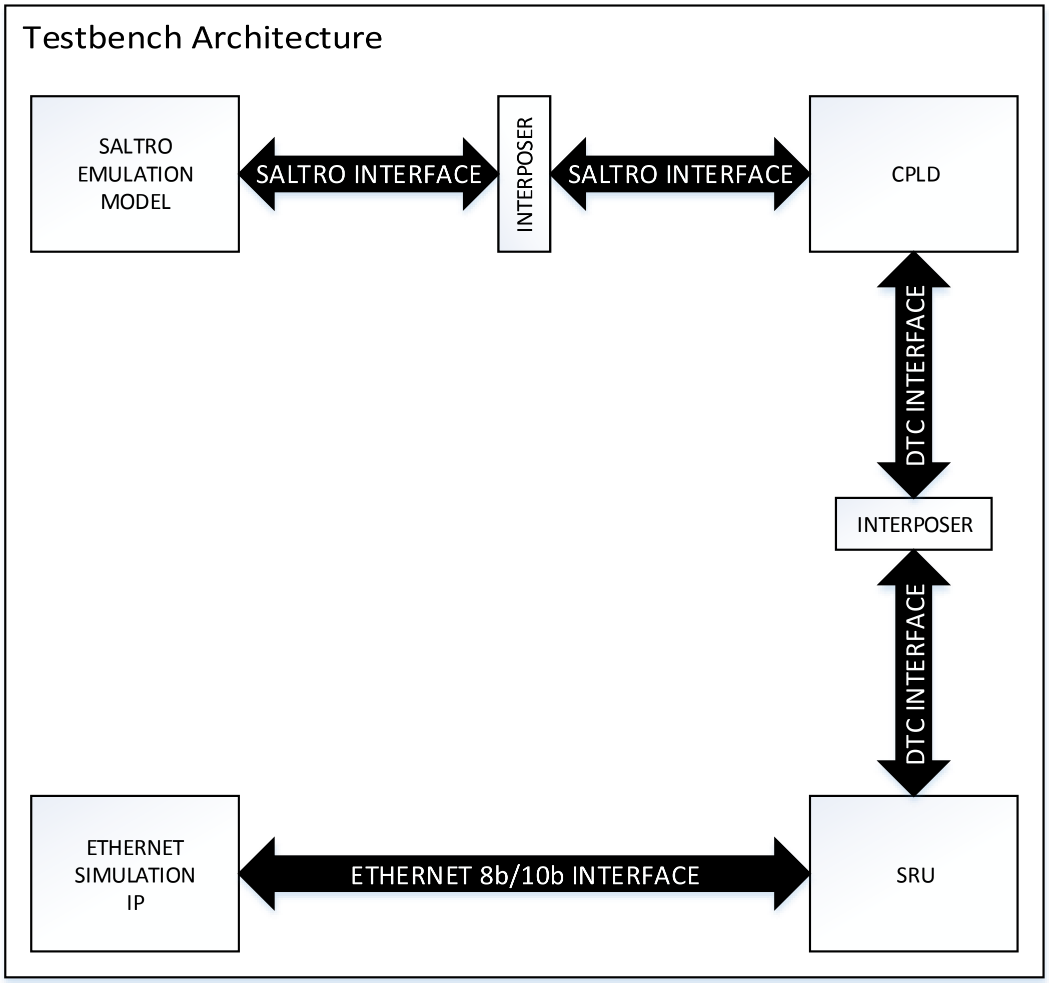

Figure 1: Testbench Architecture.

II. Implementation of the Serial Readout Unit (SRU):

On fig. 2. is shown the simplified SRU architecture.

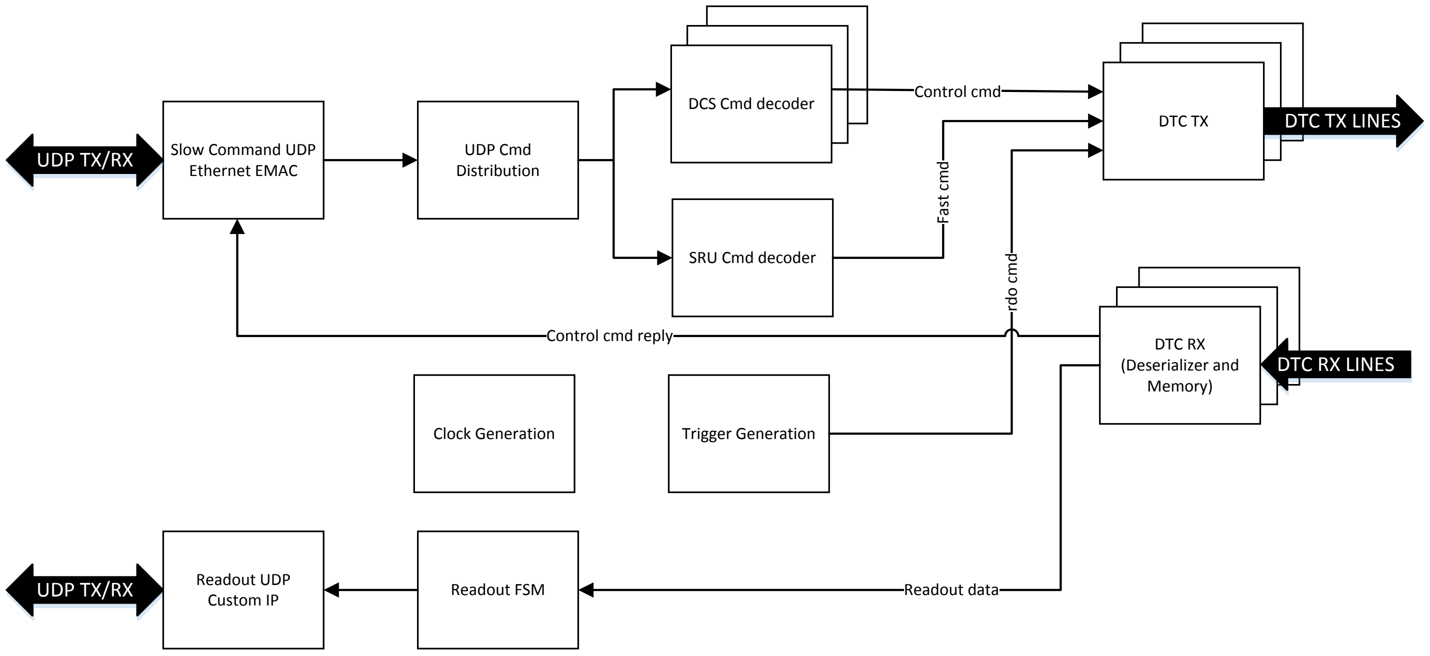

Figure 2: SRU architecture.

The SRU can support up to 40 DTC connections to 40 CPLDs. The architecture consists of:

- Slow control UDP Ethernet EMAC – This IP block contains Xilinx Custom IP Ethernet transceivers, MAC and IP frame decoding [3]. Takes care of ARP and ICMP replies. Read/Write to CSR commands must be send through this EMAC.
- UDP Cmd Distribution – Based on the Nodesel bits in the UDP packet, this module distributes the command to each of the DCS Cmd decoders or the SRU Cmd decoder.
- SRU Cmd Decoder – This module contains the SRU CSRs, when Nodesel[40] is 1, the SRU would access them.
- DCS Cmd Decoders – A total of 40 decoders. 1 for each DTC link. Decodes the 8bit data from the UDP/Cmd distribution to 32bit address & data.
- DTC TX – Serializer and encoder for DTC commands/trigger information.
- DTC RX – Deserializer and memory block. Once the event information is transmitted, it saves it into memory, which is accessed by the Read out FSM as soon as the event finishes.
- Readout FSM – Reads out the DTC RX memory and communicates with the Readout UDP Ethernet EMAC by using AXI4 protocol. Currently supports only a single DTC link.
- Readout UDP Ethernet EMAC – Packs the AXI4 data into UDP packets. Takes care of retransmitting packets if they are lost.
- Clock generation – Generates the necessary clocks and distributes them to each DTC module.
- Trigger generation – Generates the necessary trigger sequence for the SALTRO chip. Supports 3 different triggering schemes: periodic, external and via UDP write to a SRU CSR.

On fig. 3 is shown the ASMD diagram of the SRU.

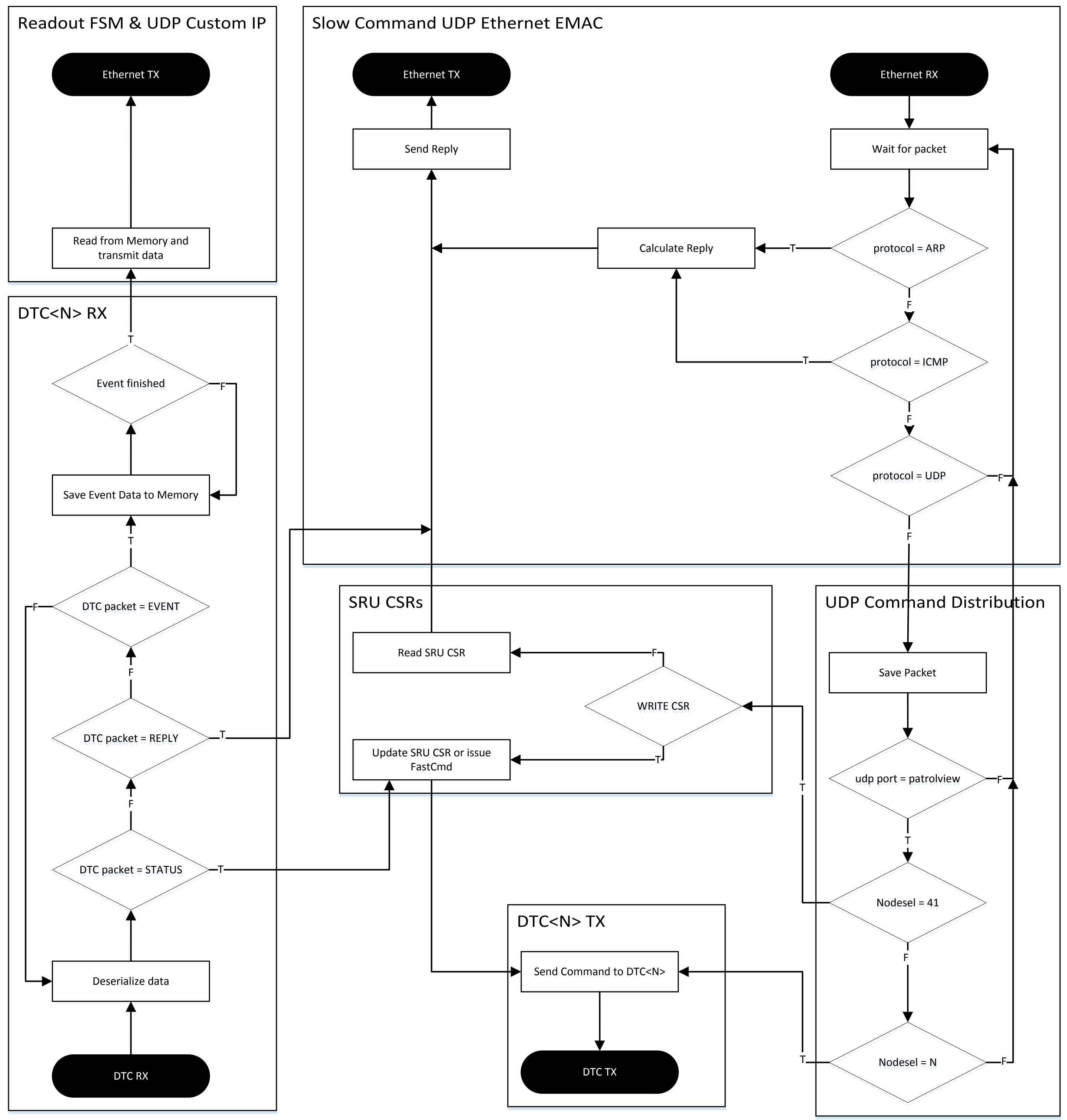

Figure 3: SRU ASMD diagram.

a. CMD/STATUS UDP Ethernet EMAC and UDP command/status payload.

On table 1 is shown the frame structure of the UDP payload of a SRU control packet. This packet must be crafted by the control software.

Table 1: UDP payload format for the SRU command frame.

31		Bit	0
RESERVED[10:0]			NodeSel[40:20]
RESERVED[11:0]			NodeSel[19:0]
WR	CType	RESERVED[9:0]	Address[19:0]
			Data[31:0]
WR	CType	RESERVED[9:0]	Address[19:0]
			Data[31:0]
		…	
WR	CType	RESERVED[9:0]	Address[19:0]
			Data[31:0]

- Nodesel[41:0] - This field has 41 active bits, corresponding to 41 slave nodes, including one SRU node and 40 DTC nodes. The 41st bit (NodeSel[40]) maps to the SRU. NodeSelH[39:20] maps to DTC39-DTC20. NodeSel[19:0] maps to DTC19-DTC0. If the corresponding bit is set, this command frame will be forwarded to the command buffers in the mapped slave DTC nodes.
- WR – This specifies the command type to be performed. “1” is Read, “0” is Write.
- Ctype – This bit specifies which chip the CPLD should address: “1” SALTRO, “0” CPLD.
- Address – This field depends on which chip is being addressed:
    - For SRU and CPLD – The lowest 16 bits are used as a 16 bit address.
    - For SALTRO – The lowest 20 bits are used as the address defined in the SALTRO specification[1].
- Data – Data to be written, if write command, or 0s if read command.
- Reserved – Reserved for future use. Set to 0s.

Due to fifo limitations the maximum amount of commands per packet is 500.

Once the command packet is send from the host computer it is received by the Ethernet CMD/Status Ethernet EMAC module in the SRU. It’s schematic is shown on fig. 4. It contains the following modules:

- Ethernet EMAC & LocalLink Wrapper – Generated IP core from Xilinx. Transmits/receives 8b/10b encoded signals to/from the Ethernet network. Uses 2 LocalLink interfaces for TX and RX.
- RX frame decoder FSM and FIFO. – From all received packets, this module filters out only the ones addressed to specific SRU IP/MAC address and protocol. Supported protocols are UDP, ARP and ICMP. This module extracts some of the necessary data, like Source host, in order to reply correctly on an ARP or ICMP request, Frame length. Contains a FIFO in order to buffer the incoming frame.
- IP Packet processing FSMs. – Contains FSMs that check if the packet frames are correct. If there is an error in the received packet it will drop it. Packages the ICMP reply data into a frame, to be transmitted, if the SRU is pinged via ICMP. Send the UDP payload for decoding of NODESEL fields.
- ARP reply FSM – Packs the ARP reply data from the RX Frame Decoder into LocalLink interface.
- IP Reply FSM – A FSM that select one of the 3 possible replies to send to the TX frame encoder.
- TX Frame encoder – Encapsulates the reply data with the Remote MAC/IP addresses and sends it to the Ethernet Xilinx core.

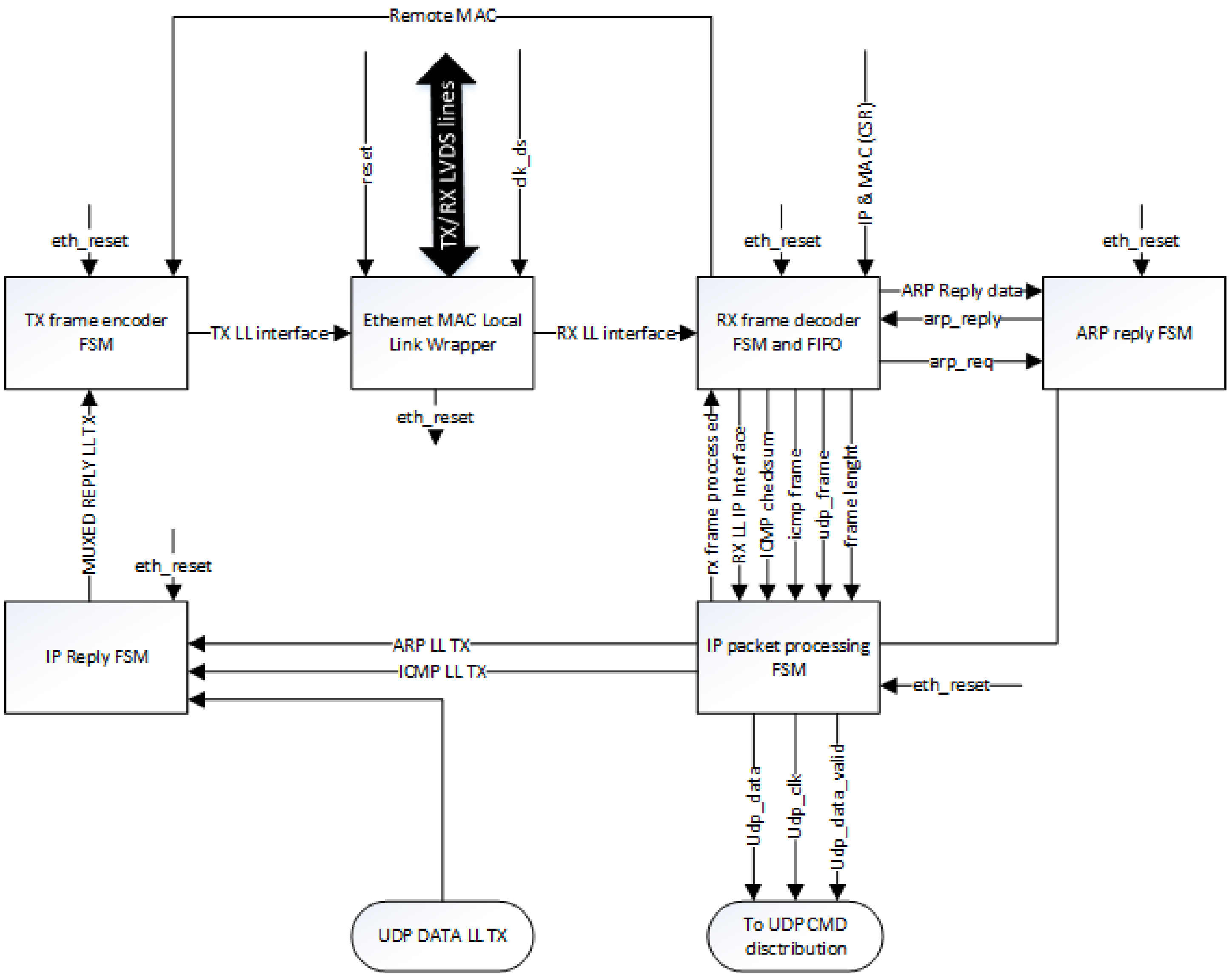

Figure 4: CMD/STATUS Ethernet schematic.

On fig. 5 is shown the ARP RX and TX frame processing. There are 3 time periods. The first one is showing the EMAC transmitting the received frame via the RX_LL interface. It is received by the RX frame decoder and it is decoded to be an ARP request frame. During this time the reply frame is calculated. In the second period the reply frame is send to the ARP Reply FSM, by asserting arp_req. This causes the FSM to store the frame in a FIFO and when the whole reply is stored into it, arp_tx_good_frame and arp_reply are asserted. In the third period the ARP LL TX interface is transmitting the reply frame to the TX reply mux (IP Reply FSM).

ICMP RX frame decoding and TX reply is shown on fig. 6. The reply is processed in 3 stages. In the first stage while the EMAC is transmitting the frame, the RX Frame decoder asserts icmp_frame signal (if the protocol, ip address and packet type matches) which turns on the calculation of the ICMP checksum. The second stage starts when the final byte of the ICMP request packet is decoded. The IP packet processing FSM starts reading the FIFO soon after that. Since the header of the TX reply is known at this time, the ICMP FSM packs it into another FIFO. After a while the payload of the request packet is inserted as the payload of the reply packet and the rx_frame_processed signal is asserted. This releases the RX frame decoder to monitor the next incoming packet, by deasserting the icmp_frame signal. In the third period the ICMP reply packet is transmitted to the IP reply FSM which selects a packet to be transmitted via the TX frame encoder.

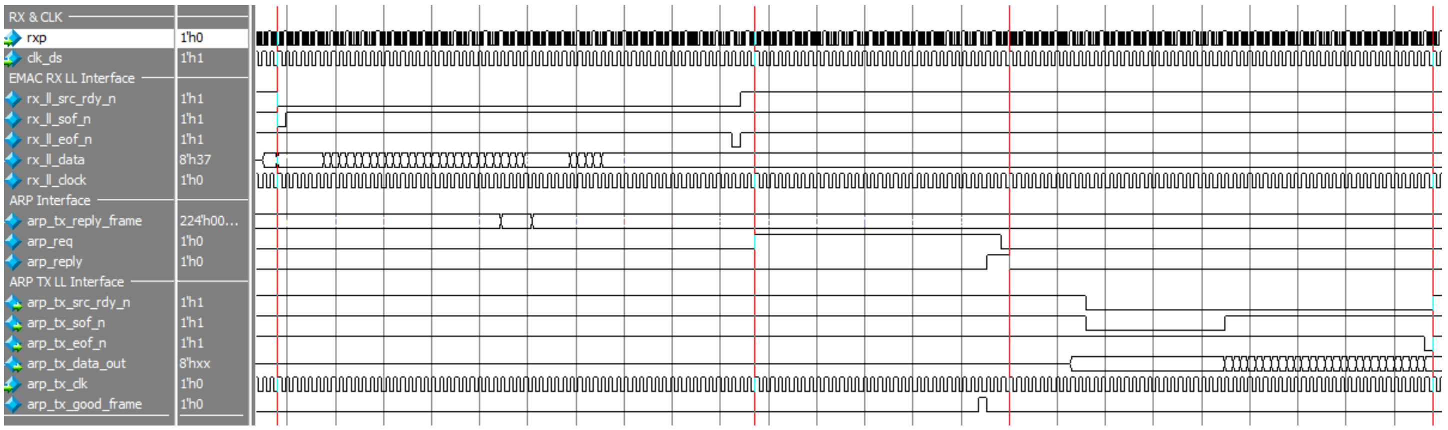

Figure 5: ARP RX frame and TX reply.

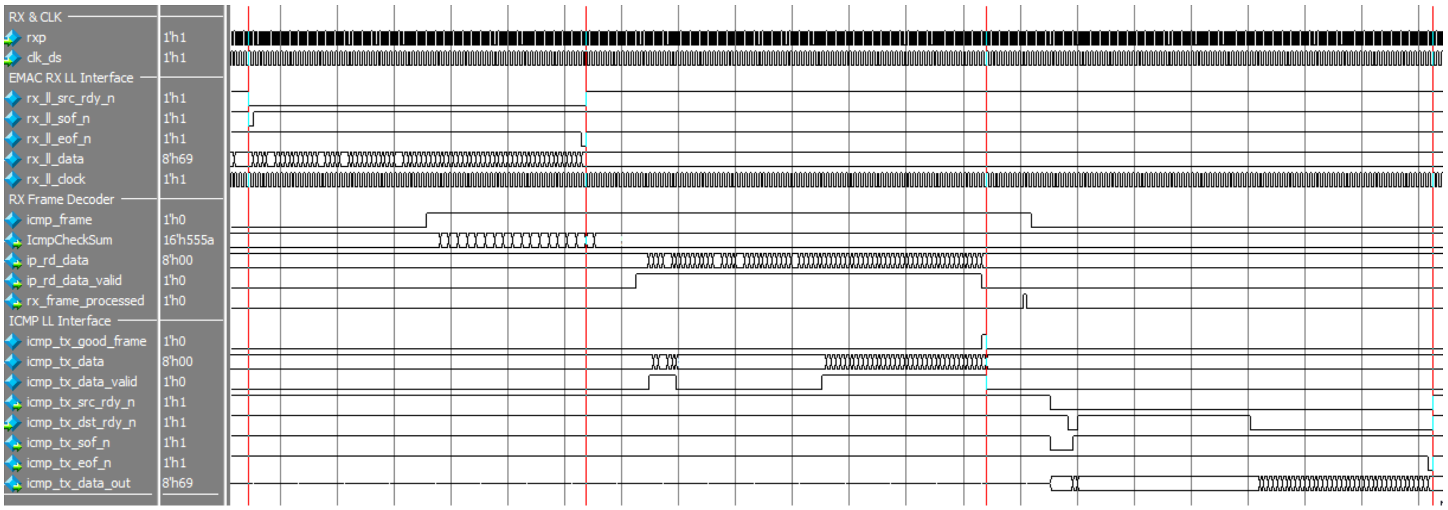

Figure 6: ICMP RX Frame and TX reply.

The decoding of a UDP Command packet is done very similarly to the ICMP frame. It consists of 3 periods, shown on fig. 7. While the EMAC is transmitting the packet, the frame decoder is monitoring. if the frame is using UDP and the IP and MAC addresses match, it asserts the udp_frame signal. When the whole packet is received, the IP packet processing FSM initiates the FIFO in the decoder. A very simple UDP FSM, counts how many bytes have been received, and asserts the udp_rx_dv signal, which signifies the UDP packet payload. The UDP bytes are transmitted on the udp_rxd bus. Once the frame is processed, rx_frame_processed is asserted, thus releasing the RX frame decoder to receive new frames.

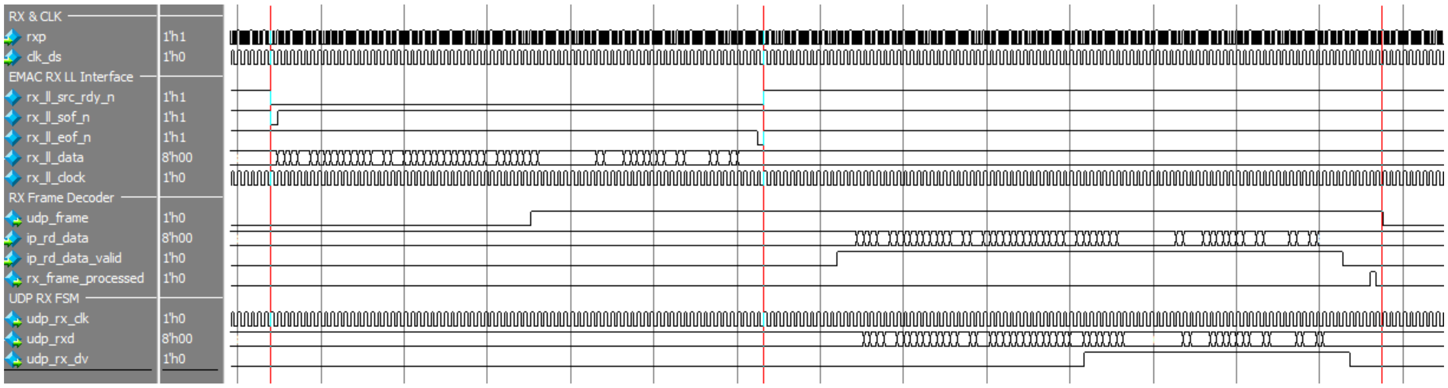

Figure 7: UDP RX Frame decoding.

b. UDP command distribution.

After the UDP command frame is extracted from the IPv4 packet, it is received by the UDP command distribution module. There the UDP destination port is matched. The NODESEL fields are decoded and demultiplexed into separate bus dcs_rx_dv[40:0]. Waveforms showing the decoding of a packet with NODESEL0 (i.e. 1st DTC board) and NODESEL40 (i.e. SRU) are shown on fig. 8 and fig. 9. The FSM takes one clock cycle to compute when to assert the dcs_rx_dv signal, thus the dcs_rxd is the delayed udp_rxd by a cycle. The first 16 bytes from the frame contain the UDP source & destination addresses, frame length and checksum. They also contain the 8 bytes encoding the NODESEL. Thus for the first 16 bytes the dcs_rx_dv is always kept low. Then based on what the NODESEL bits that are set in the packet, dcs_rx_dv[40:0] is asserted. Thus the dcs_rxd data is driving the 40 DTC transmitters and the SRU Cmd decoder with the same data, but different "data valid" signals.

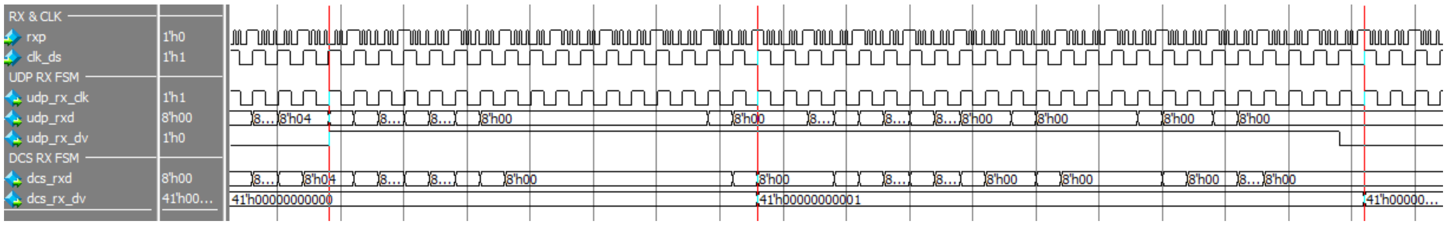

Figure 8: UDP payload addressed to NODESEL0, the first DTC board.

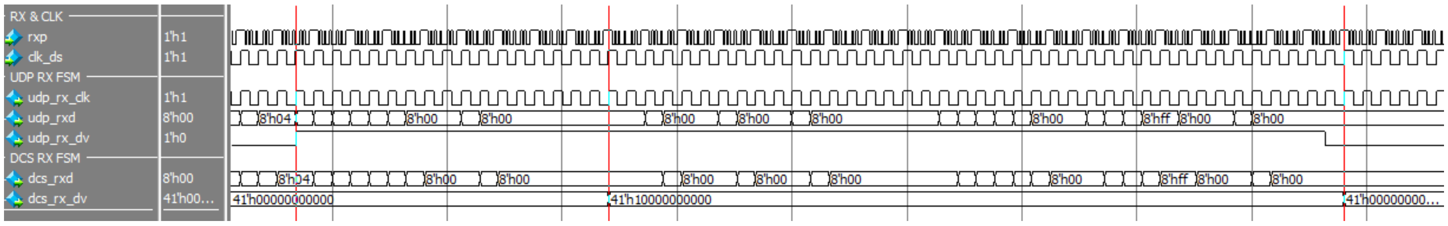

Figure 9: UDP payload addressed to NODESEL40, the SRU.

c. DCS command decoder

In order to support broadcasting read/write operations to 40 DTC boards each of the DTC link must have its own command decoder. The block diagram of the DCS command decoder is shown on fig. 10. It consists of a FIFO and an FSM. The FIFO is used to cross into the DTC clock domain and to store the commands from the packet, since the DTC link is orders of magnitude slower than Ethernet. The FSM reads 2 32bit words, containing the address and the data respectfully. The State diagram of the FSM is shown on fig. 11.

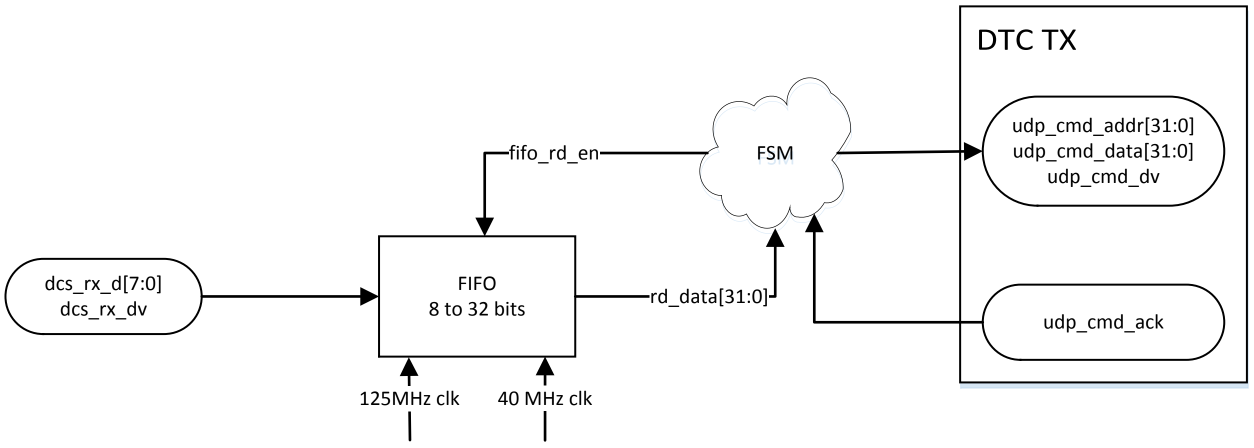

Figure 10: DCS command decoder block diagram.

The FSM initializes in st0, and waits until the FIFO is not empty to proceed to st1. St1 is a state where the UDP frame payload must be fully loaded into the FIFO, this ensures that there is always even number or 32 bit words in the FIFO. During st2 and st3, a single command containing 32bit address and data is readout, to the transmitter. Since the FSM is Moore and the FIFO has 1 cycle RD delay in st4 & st5 the udp_cmd_addr and udp_cmd_data are available to be assigned. In st5 udp_cmd_dv is asserted, signaling the DTC TX Serializer that there is a command valid and ready to transmit. When the DTC TX Serializer is starting to transmit the command, it asserts udp_cmd_ack, releasing the FSM into st6, which select which 2 watchdog states it should go into, based on if this was the last command in the FIFO. The watchdog is set to 160 cycles, since it takes a little over 128 cycles to transmit a command.

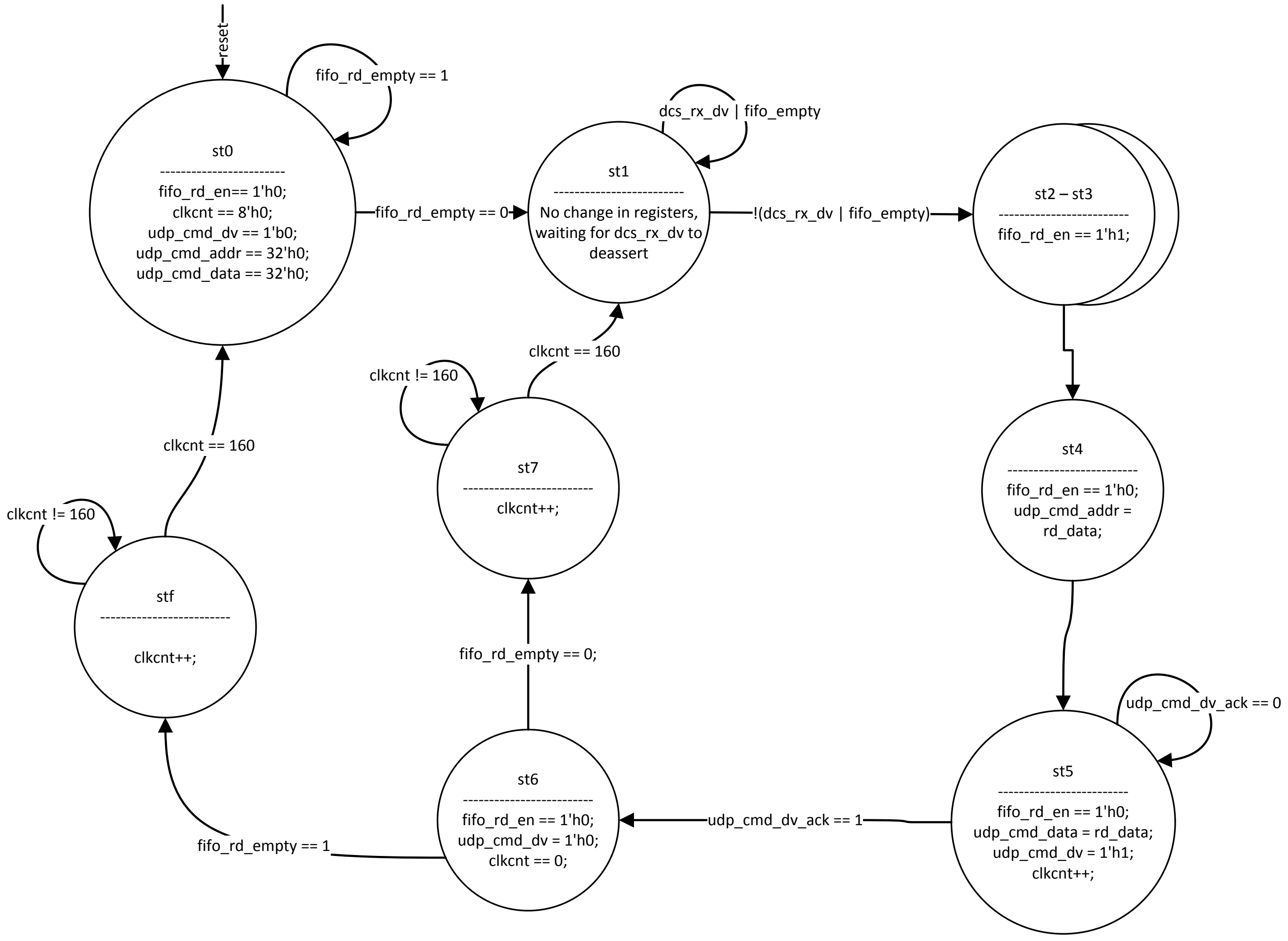

Figure 11: DCS Command decoder FSM state diagram.

On fig. 12 waveforms of filling the FIFO with a UDP frame payload (containing 2 command) from the DCS command distribution module, are shown. Soon after the first command is transmitted, followed by the second one in a while.

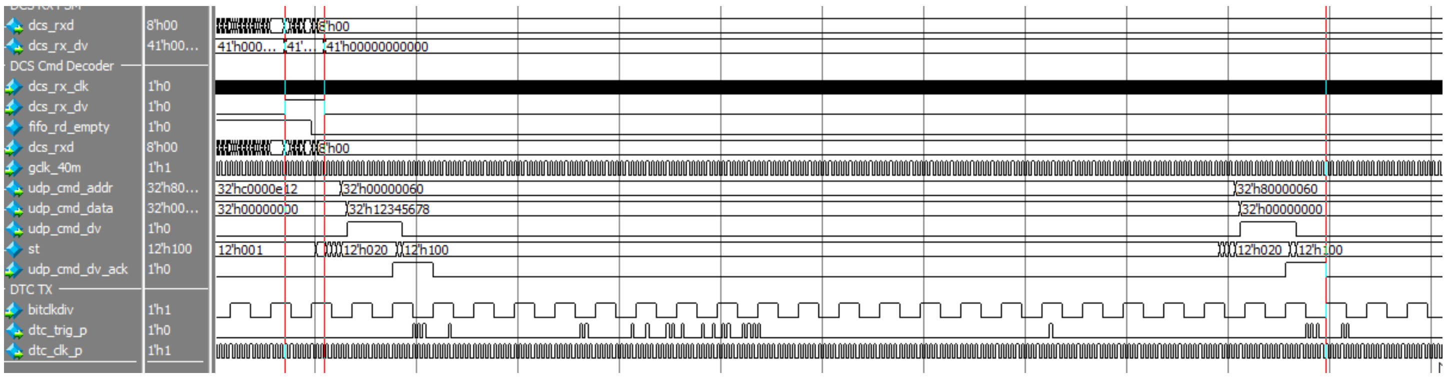

Figure 12: DCS Command decoder – waveforms.

d. DTC TX

In order to support the 3 types of DTC commands [2] (Trigger, Slow Control Commands and Fast Commands), the DTC TX module is connected to the three modules that issue them. To the DCS command decoder – when issuing a CSR write/read command; The SRU CSR module – when a fast command is issued; the trigger generation unit when triggering and abort command needs to be issued. The block diagram is shown on fig. 13. It consists of a DTC packing FSM, 3 smaller FSMs which convert the command pulses to a handshake interface, a serializer and output buffers. The handshake interface is required in order to ensure the command will be executed, if the FSM is busy with another command.

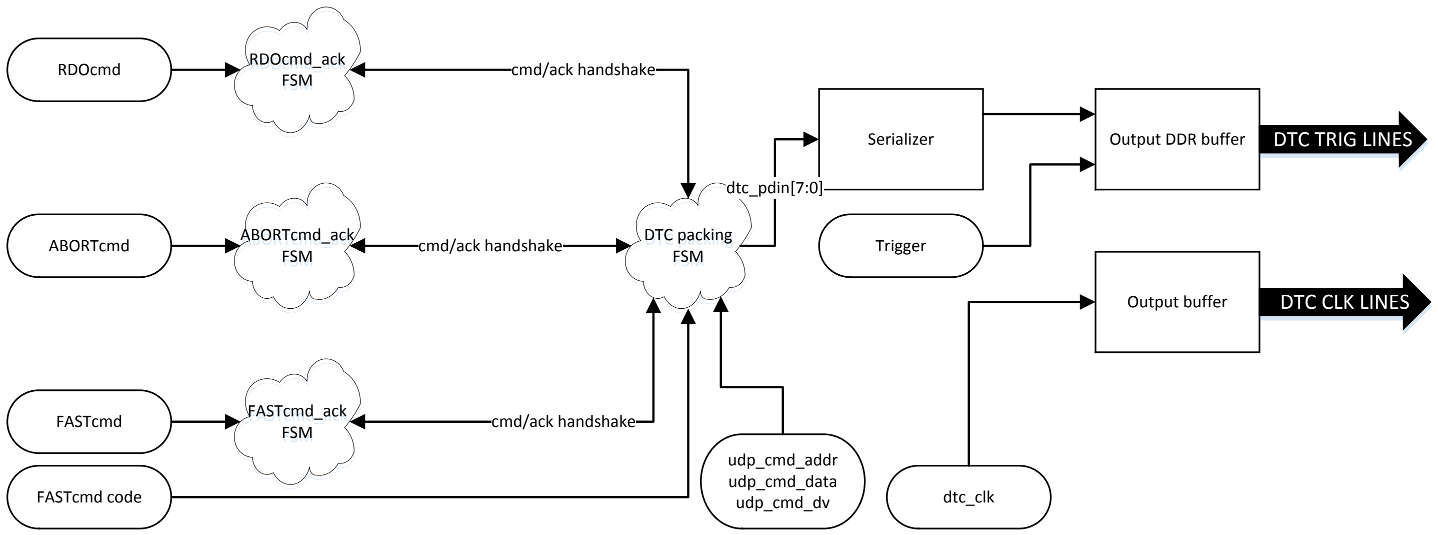

Figure 13: DTC TX block diagram.

The FSM diagram is shown on fig. 15. The FSM initializes in st0 and wait until a valid command is issued. Based on which command is executed, it enters st1, st2, st4 or st13. In st1, st2 and st13, the corresponding command code is asserted on dtc_pdin[7:0]. When entering st4, the read/write command code is asserted and then the next 8 states are used to assert the 2 32bit words for address and data. A complete waveform is shown on fig. 14. In the first period the read/write header code (0xE1) is asserted, then in the second period the address, and in the third – the data is asserted. At the end of the waveform, it just so happens there is a trigger sent unrelated to the CSR command.

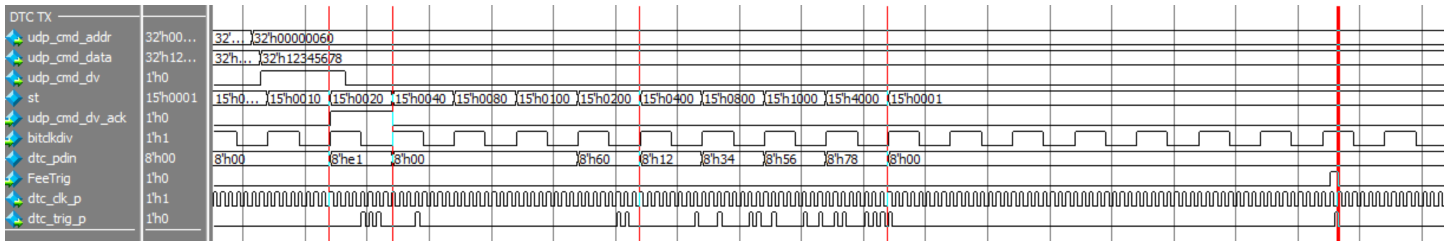

Figure 14: Read Write command serialization and Trigger waveforms.

The dtc_pdin[7:0] gets serialized and that signal is driving the Output DDR buffer on negedge, while the Trigger (FeeTrig) is driving it on the posedge, thus the DTC TX transfer is completed [2]. The 40 MHz dtc_clk is directly driving the DTC CLK lanes.

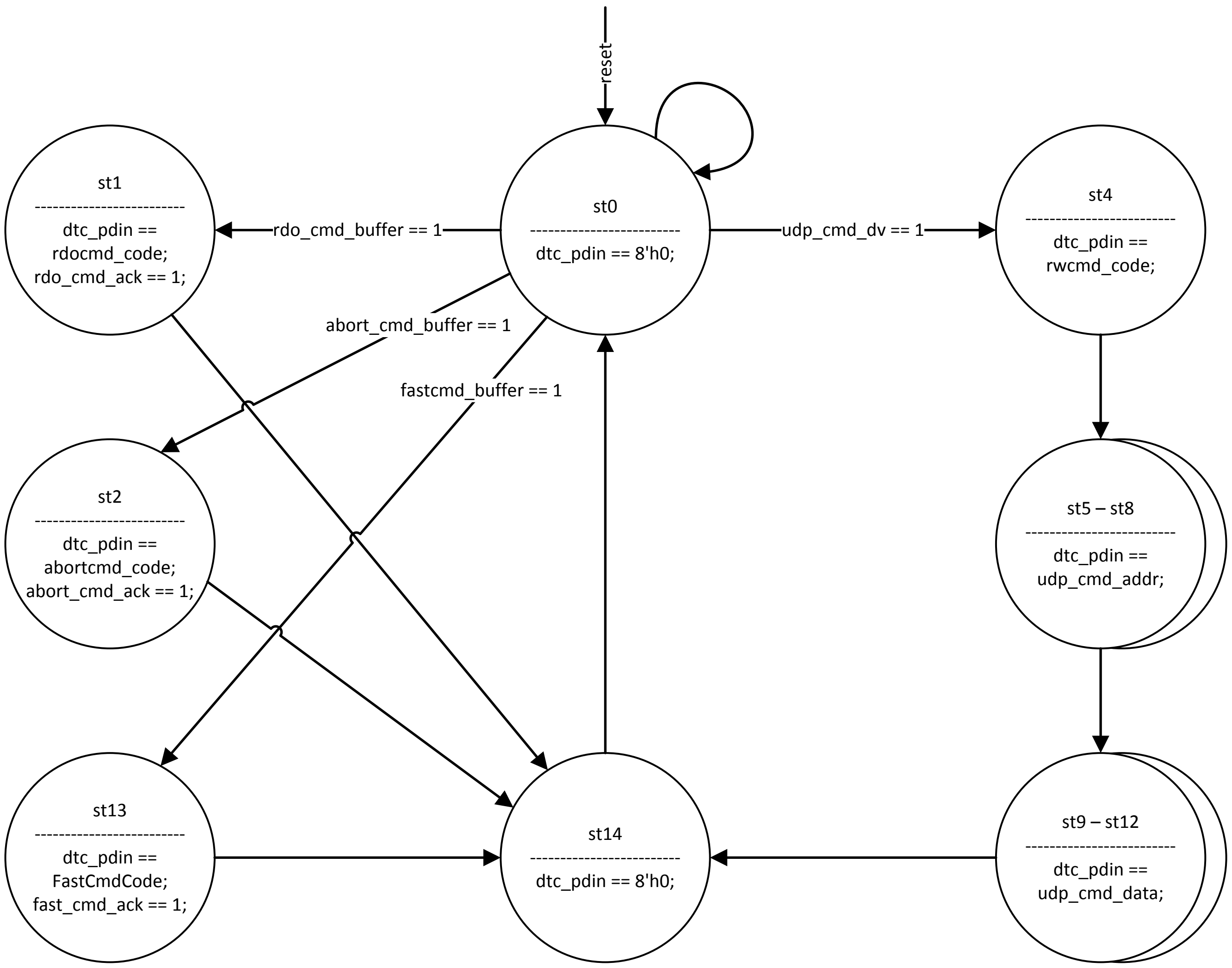

Figure 15: DTC TX FSM state diagram.

e. DTC RX (Deserializer and Memory)

The DTC protocol has 3 different packets [2] that are sent from the CPLD. The block diagram of the DTC RX module is shown on fig. 16. The data is deserialized and send to the DTX RX Decoder. It is an FSM whose state diagram is shown on fig. 17. In order to receive correct data, the serializer must be aligned to the sync word “0xBC50”. This is done in the ALIGN state. Once that is done, the Decoder enters WAIT state. Once a valid frame header is detected, it proceeds to the corresponding branch: STATUS, REPLY or EVENT. In the STATUS branch, just the LSB bit is used, it signifies if the SALTRO chip has asserted the ERROR flag. The REPLY branch consists of 4 stages, used to get the two 32bit words reply: address and data from a CSR read command. In the EVENT branch, every 2 cycles it writes to the memory a single 32 bit word, until the trailer is detected. Once the end of the event frame is detected, the Decoder signals to the Ram Management that the writing to the ram has finished. The Ram Management asserts RamFlag, which will stay high, until the Readout FSM asserts RdConfirm, when the final word was readout.

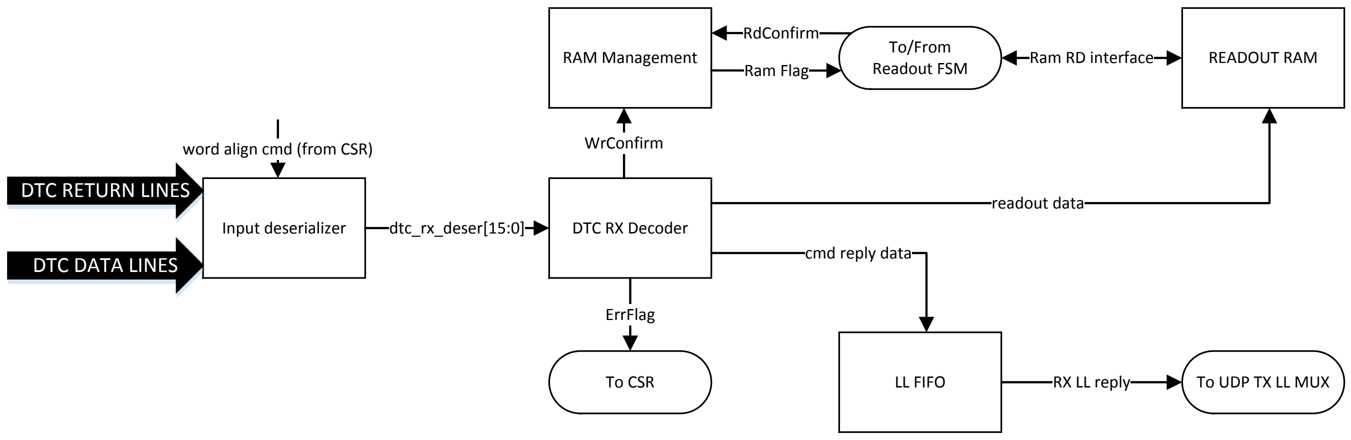

Figure 16: DTC RX block diagram.

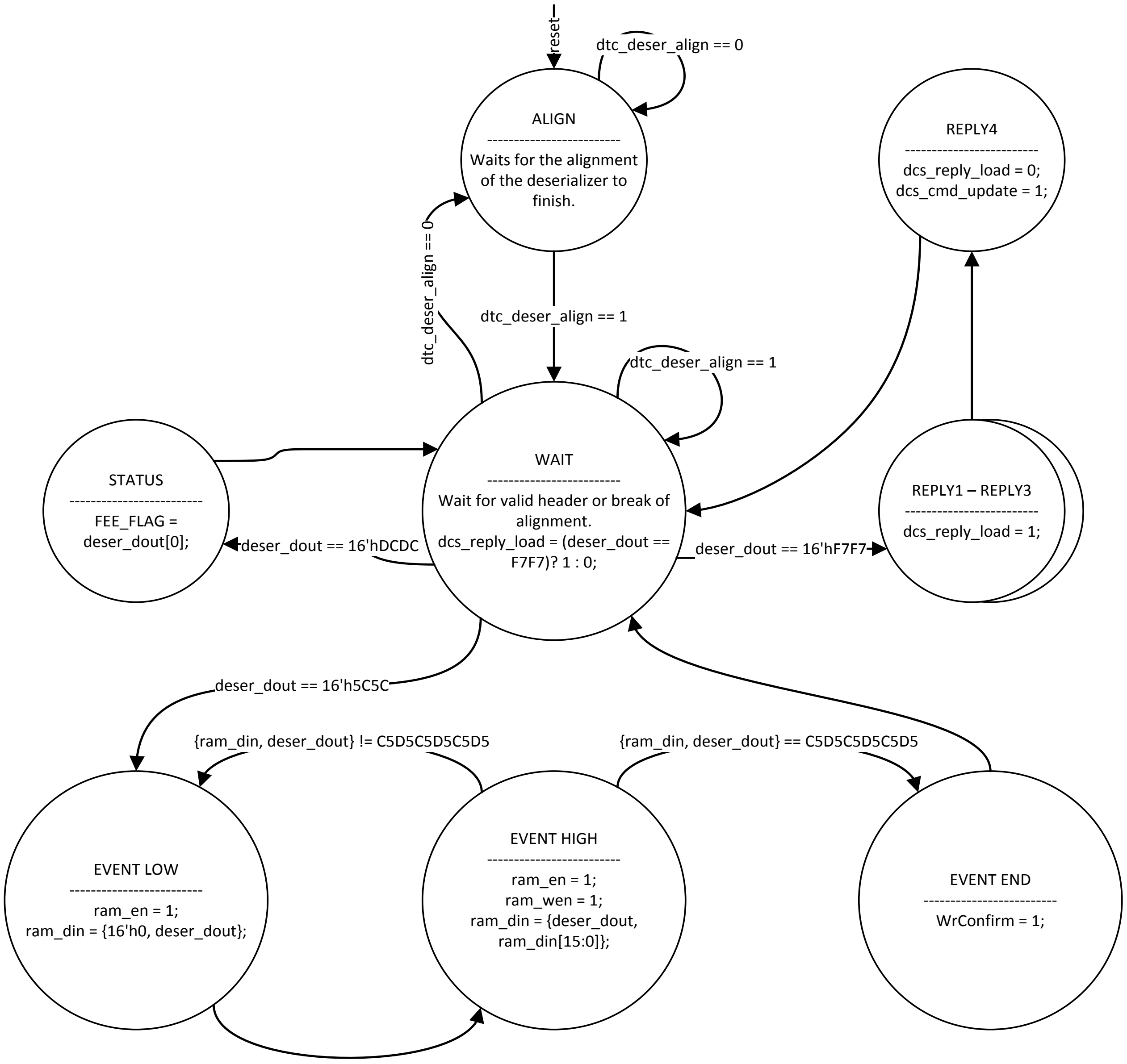

Figure 17: DTC RX Decoder state diagram.

f. UDP Readout FSM and UDP EMAC.

A temporary FSM is implemented in order to convert the readout of a single DTC board to AXI4 stream interface used by the Readout UDP EMAC, which is already synthesized IP, from another project.

In the beginning of a packet transfer shown on fig. 18 the FSM calculates the total amount of data in bytes that will be transferred and asserts it on the tuser input of the AXI interface. Tvalid is asserted, when the first byte is ready to be transmitted. After a while tready is asserted by the EMAC. The transfer continues by sending a new byte each cycle as long as tready is high. When the last byte is to be transmitted, tlast is asserted for a single cycle and tvalid is deasserted.

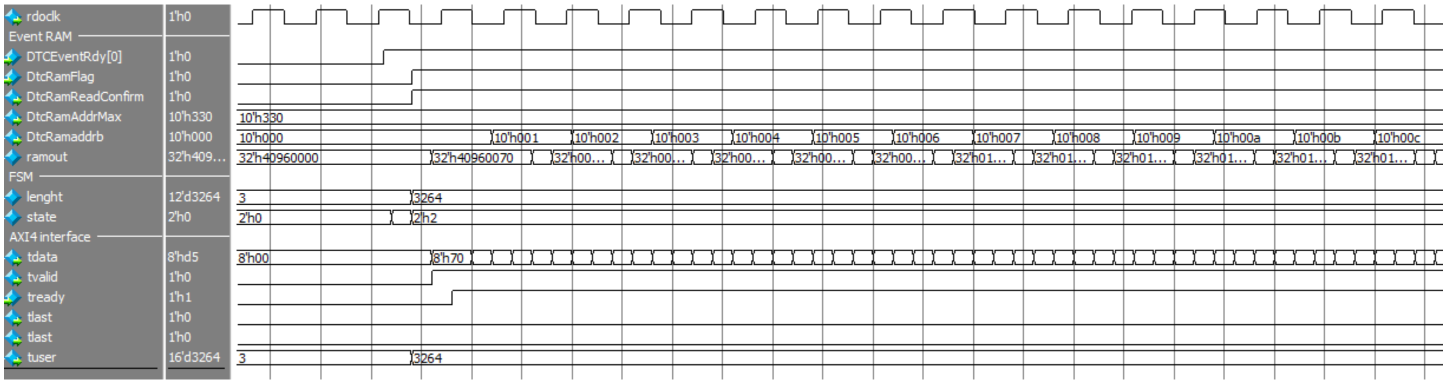

Figure 18: Beginning of event transmission to UDP Readout EMAC.

g. Clock Generation

The SRU supports clocks from external source or from on-board XTAL. The block diagram is shown on fig. 19. The LVDS clock lines come from their corresponding pads, and are buffered by an IBUFGS. Then a clock mux can be programmed by software by writing a CSR to switch to the external clock. The dcsclk output is used only when booting up, to drive the flash controller that programs the FPGA.

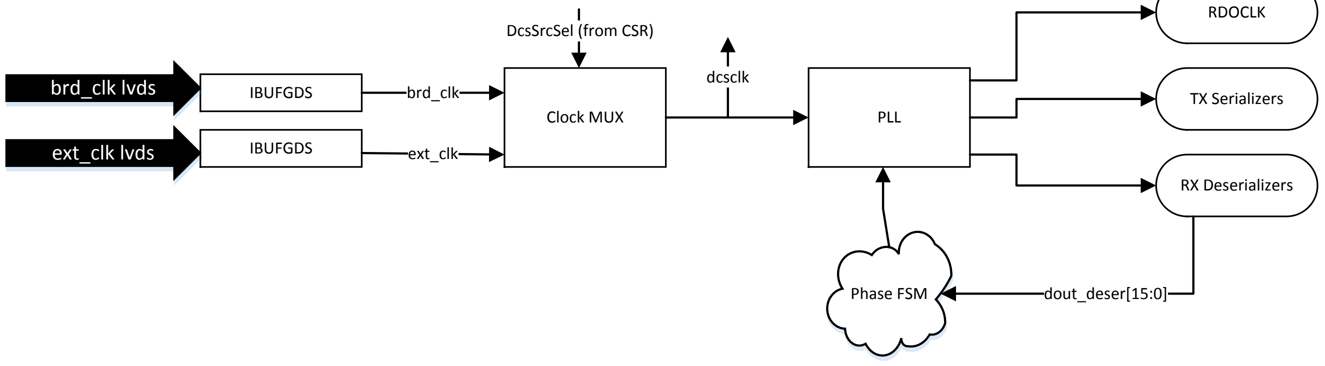

Figure 19: Clock generation block diagram.

After the clock mux locks, the PLL starts locking and asserts SerClkLockSt when locked. After this point the clocks are stable and can be used by the TX Serializers and the RX Deserializers. On fig. 20 is shown the simulation power up and locking of the clock generation circuit. Since it's crucial that the CPLD boards are triggered at the same moment, the TX serializer clocks' phase is constant and cannot be changed. The RX deserializer clocks' phase is tunable using an FSM and using the data feedback from the serializers. Though this functionality has not been tested as part of this project.

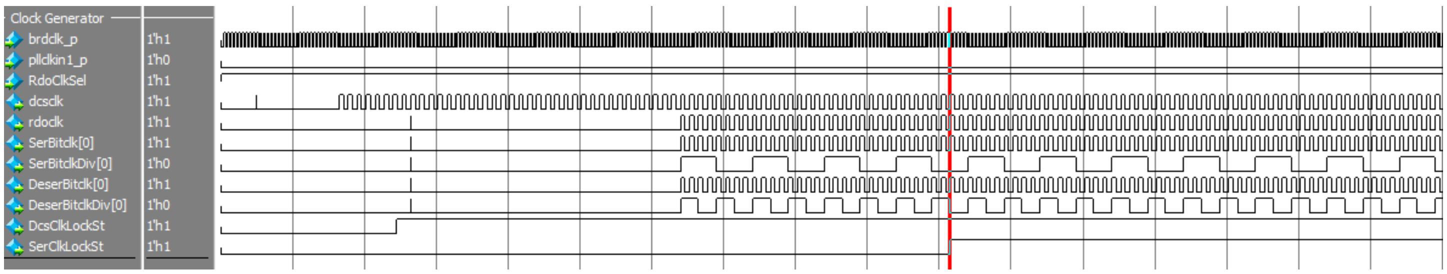

Figure 20: Power-up simulation of the clock generator.

III. Implementation of the CPLD:

The CPLD architecture is shown on fig. 21. The main functional block are:

- RDOClk input buffer and ADCClk PLL– This module converts the LVDS DTC Clk into single line RDOClk and from it a PLL generated the ADCClk sampling clock which is 2x or 4x slower than the RDOClk.
- DTC RX decoder – Decodes the DTC command on the DTC trig line. If it is a Write or Read CSR, it will be sent to the Cmd/Data demux. If it’s a fast command it will send it to the Saltro Controller.
- Trigger decode – Decodes the Trigger command, and toggles the L1 & L2 lines respectfully.
- Cmd and data demux, Reply data packer – Distributes the Address/Data lines according to where they are for, the CPLD CSRs or the SALTRO chips. Also packs the reply data when a read command is executed.
- CPLD CSRs – Control Status registers for the CPLD.
- Saltro Interface controller & Readout LIFO – Converts the commands in the Saltro interface format. Also converts the readout data format using a 40 bit wide LIFO and a 32 bit wide FIFO and sends it to the DTC TX FSM.
- DTC TX FSM and Serializers – An FSM drives 2 DDR serializers which are transmitting readout or reply data on the DTC data/return lines.

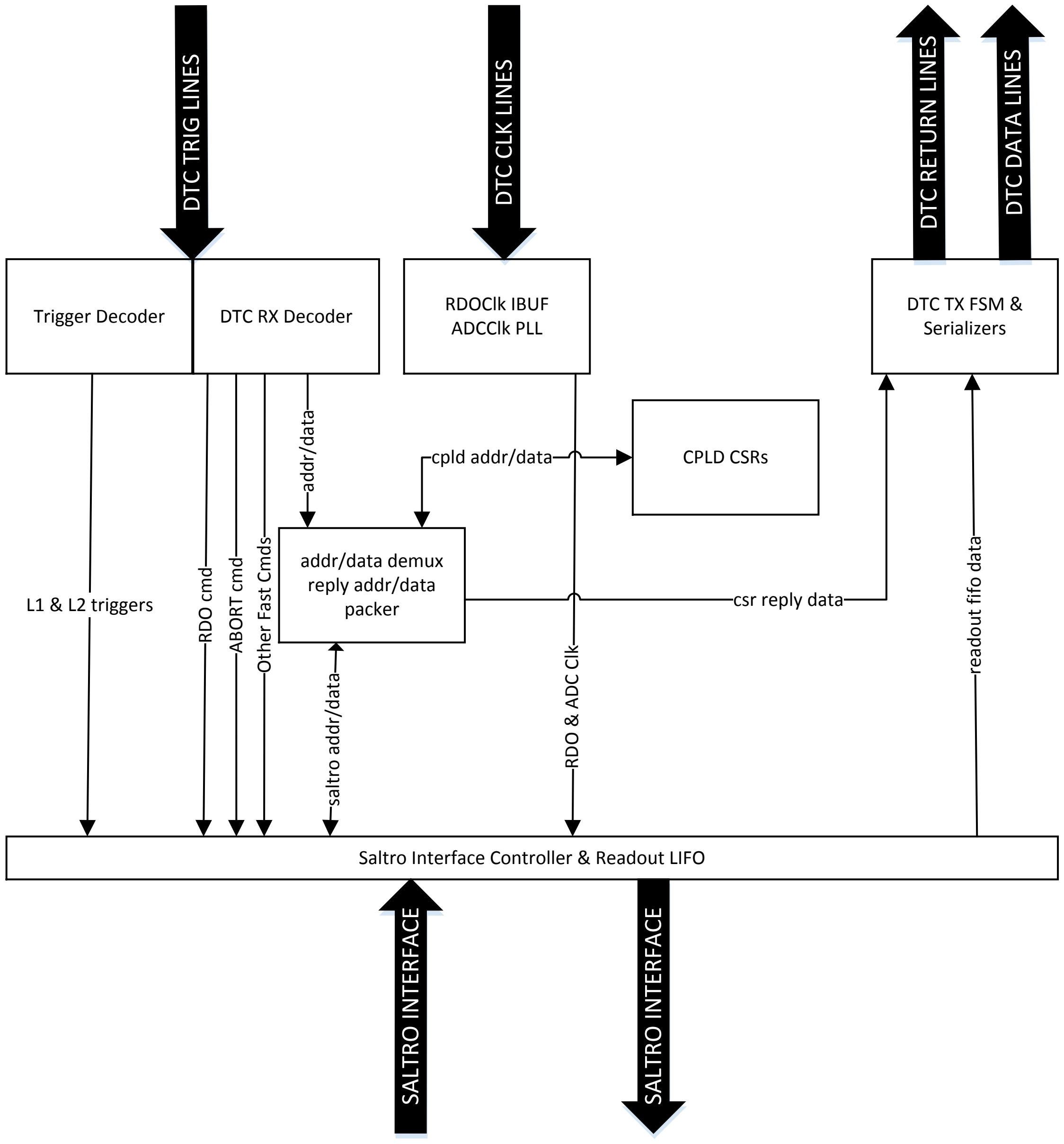

Figure 21: CPLD Architecture.

The ASMD of the CPLD is shown on fig. 22

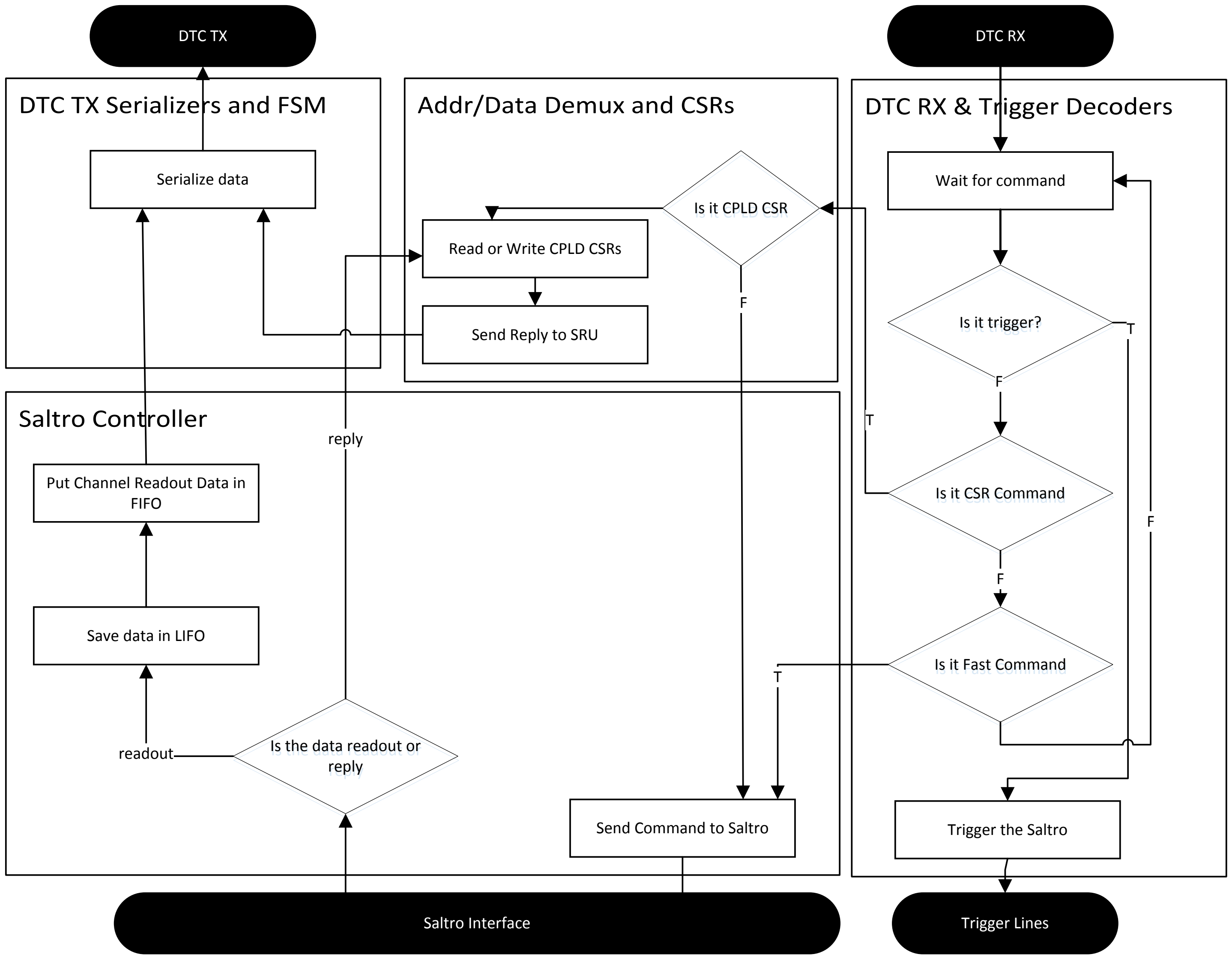

Figure 22: CPLD ASMD Diagram.

a. RDOClk and ADC clock

The CPLD works on 40 MHz RDOClk which is the same a the DTC clock. In order to control the resolution vs sampling time, the ADC clock can be 2 times or 4 times slower than the RDO clock. The block diagram of the clock generation is shown on fig. 23.

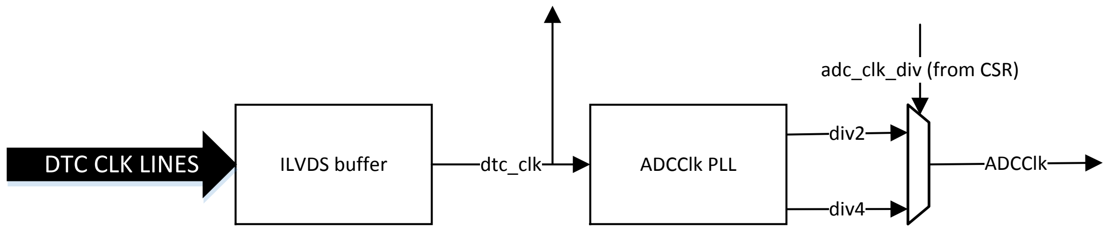

Figure 23: ADC clock PLL and DTC clock buffer.

b. Trigger Detection

The trigger sequence is detected by a detector, which consists of a shift register sampling the dtc_trig line on **negedge**. Then the output of it is monitored by an FSM, which generated the L1 and L2 trigger. The state diagram is shown on fig. 24. The FSM's initial state is WAIT0. When the shift_reg has a value of "0x2"

it means that it has received a L1 trigger. Thus it asserts it on the next cycle, and enters Trig_L1 state, where L1 trigger is hold low for another 10 clock cycles and then returns to the initial state.
When the shift_reg is "3", the FSM changes state to WAIT1 where it detects if the L2 trigger is received or there was an error in the trigger command. If a valid L2 trigger is received it enters Trig_L2 state, where trig_l2_n is asserted for 2 cycles. If the trigger was not valid in WAIT1 state, the FSM enters WAIT3 state. There a small timeout is added in order to ignore the false trigger. Since the trigger data is the only data sent on negedge, it can be sent together with a Slow or Fast command, an example of which is shown on fig. 25.

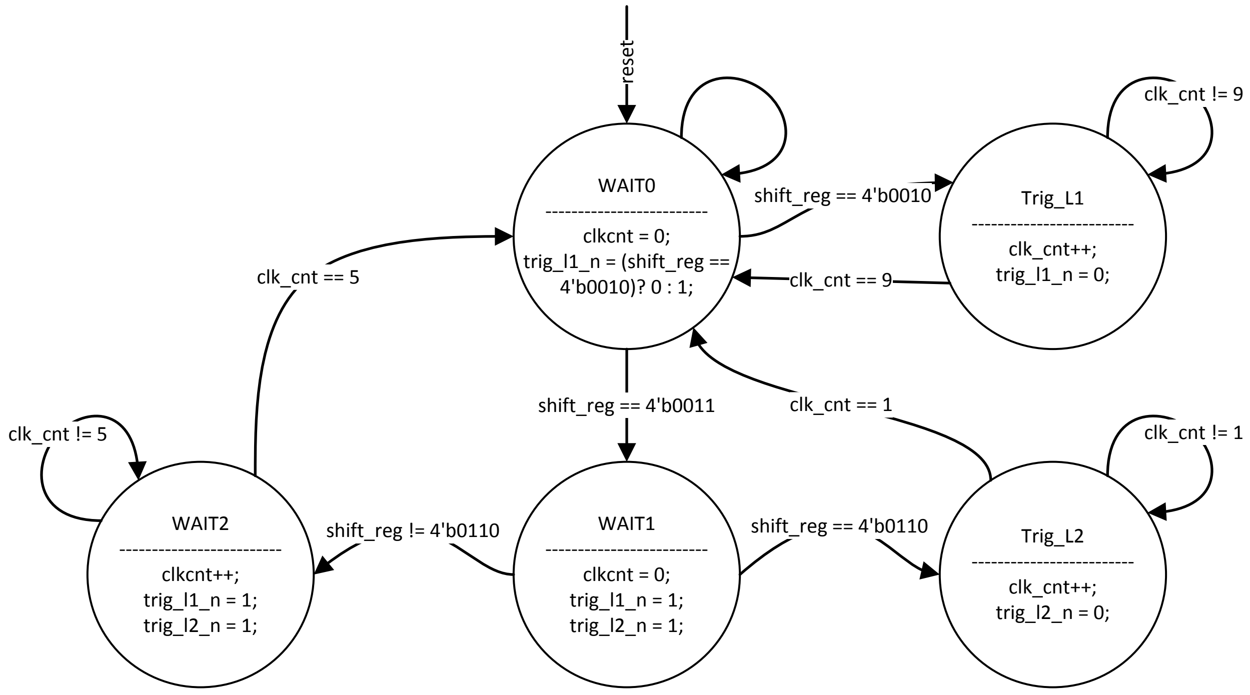

Figure 24: CPLD Trigger decode FSM.

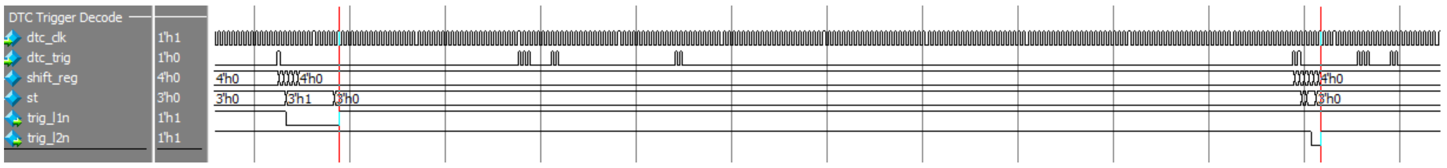

Figure 25: Trigger detection in the middle of WR/RD command.

c. DTC RX Decoder

The DTC RX decoder is implemented similarly to the Trigger decoder. The dtc_trig line is sampled on **posedge** by a shift register which trigger the FSM, that decodes the command. On fig.26 the state diagram of the FSM is shown. It initializes in st0, in which it waits to receive a valid header code in the shift_reg. If any of the 3 fast commands (channel readout, abort or reset) are received, the FSM enters the corresponding state and asserts the specific command flag and return to the initial state. If a slow WR/RD command header is received, the FSM enters stcmd1 state, where it waits for the shift_reg to receive the address, data, Ctype and WriteRead flag. Then it enters stcmd2, where it asserts dtc_cmd_exec and the address, data etc. fields. It waits in stcmd3 state until a dtc_cmd_ack is received from the cmd demux.

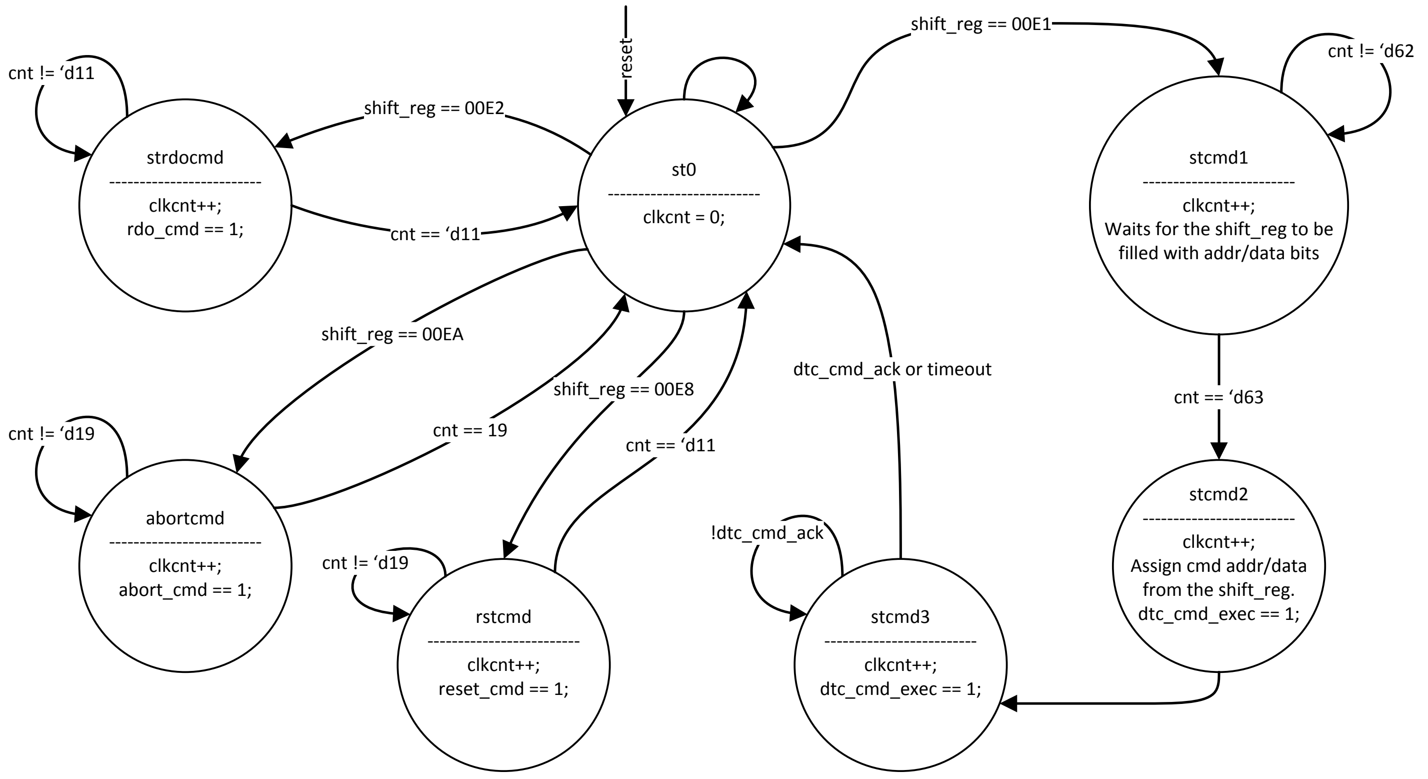

Figure 26: CPLD DTC RX Decoder FSM state diagram.

d. Address/Data demux and Reply packer.

When the RX Decoder receives a Write/Read command, based on the Ctype (dtc_cmd_feenal) the demux module sends the command to the CPLD CSRs or the Saltro Interface Controller. On fig.27 and fig.28 are shown waveforms of Read and Write to the CPLD and to the SALTRO.

When a RD command is executed and an ack is returned by ether the CPLD CSRs or the SALTRO Controller, and the command is READ, a small FSM shown on fig.29 is triggered. It asserts reply_ready. Which in turn triggers the DTC TX FSM to start transmitting the reply data. While the TX FSM Is transmitting the data frame_state is kept high, thus ensuring the CPLD will not send another reply_rdy request in that time. This handshake is shown on fig.30.

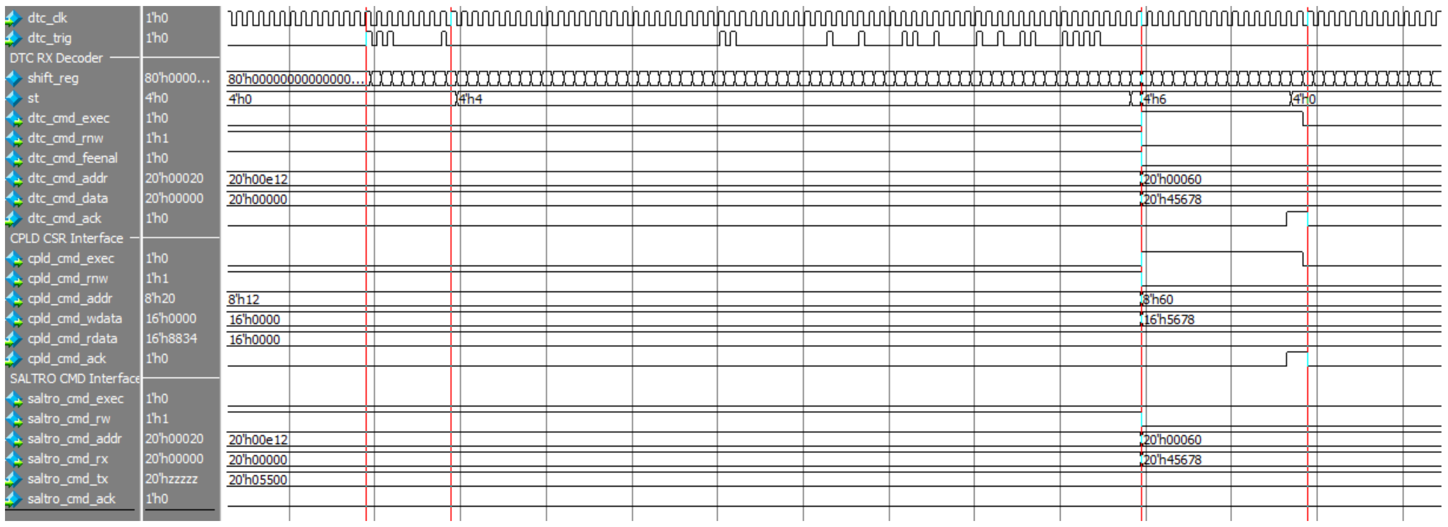

Figure 27: Write CPLD CSR command waveforms .

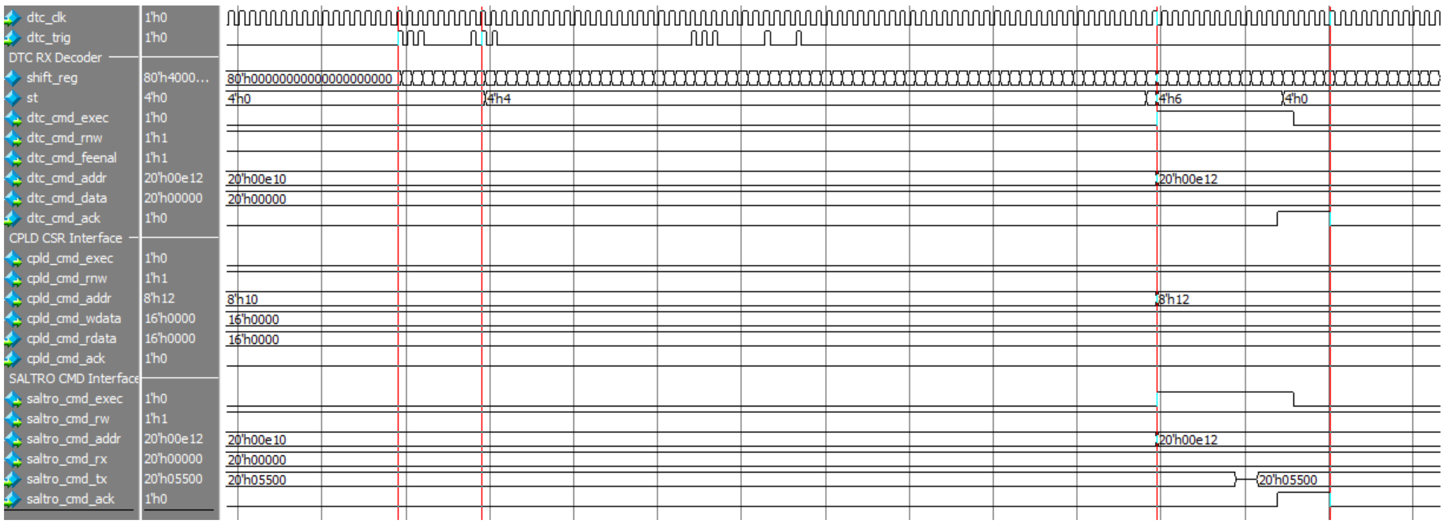

Figure 28: Read SALTRO CSR command waveforms.

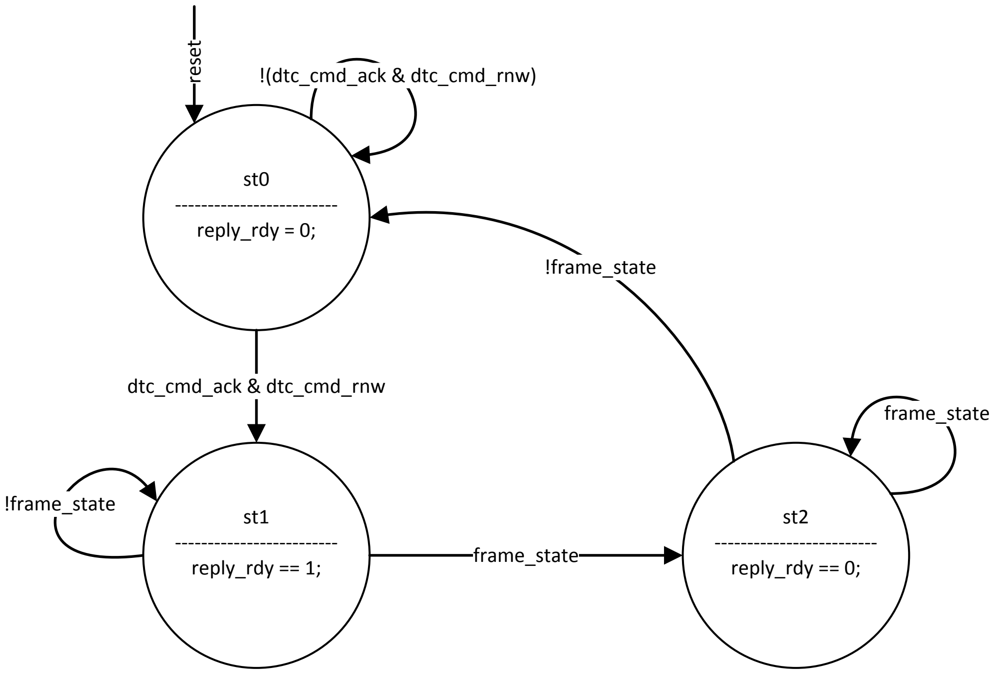

Figure 29: DTC Reply_Rdy FSM state diagram.

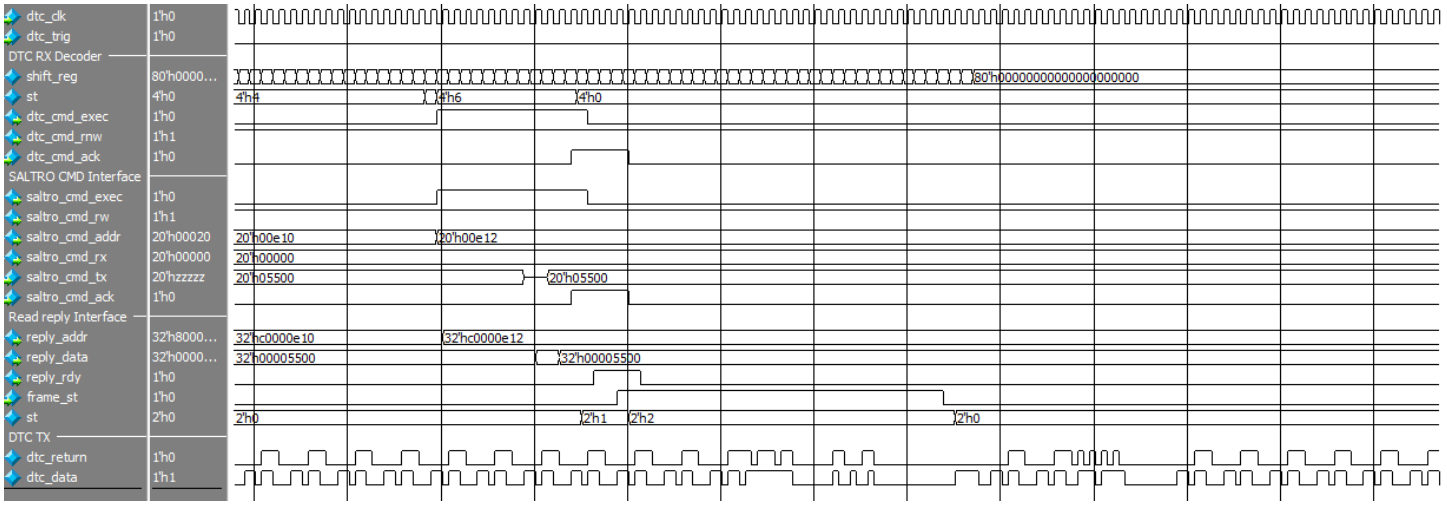

Figure 30: Reply_Rdy and frame_state handshake.

e. Saltro Interface Controller

The Saltro Controller consists of 2 FSMs, a dual port RAM and a FIFO. The block diagram is shown on fig.31. Every time the Saltro Controller receives a command, the Command Control FSM (CCFSM) takes control of the bi-directional BD bus. The state diagram of the CCFSM is shown on fig.32. If the command is Write/Read SALTRO CSR, the state machine enters st_cmd1, where the address and data are sampled, by the bdout registers (those registers drive the BD bus, when cstbn is asserted), also "writen" is sampling the command type. Then in st_cmd2, the chip select is asserted, sending the command to the SALTRO chip. Based on, if the command is broadcast or not, the state enters a st_cmd3 (where it waits for ackn to deassert and chip select is deasserted) or st_cmd4. In st_cmd4, after a few clock cycles the FSM returns to the initial state.

When a Channel Readout (CHRDO) command is send the FSM enters st_trsf1, which initializes the handshake with the second FSM in the controller. At his point the second FSM starts to wait for the trsfn to assert and then deassert, which means that a single channel had been readout and put into the Channel RAM. In st_trsf1 the Control FSM also checks, if the last channel has been readout or if the current channel is masked. If the channel was masked, then the ch_addr is incremented in st_mask and the channel mask register is shifted.

If the current channel has not been readout yet, the FSM goes to st_trsf2, where the bdout registers sample the correct command. In this state, if the second FSM has not yet readout the channel ram from the previous channel, it will delay the execution of the command. Also that will happen, if the 32bit fifo will not be able to fit the new data coming in, since the bottleneck is the DTC TX bandwidth.

Once the Channel RAM has been read and the fifo can accommodate the new data, the FSM executes a channel readout command for channel with address ch_addr in st_trsf3. The ram wr_addr is set to 0 in order to start writing the ram from the start.

In st_trsf4, the FSM waits for ackn to be asserted or for timeout to occur. In st_trsf5 the chip select is deasserted and the FSM is waiting for the transfer to initialize. If this doesn't happen a timeout will occurs.

In st_trsf6 the FSM is waiting for the transfer to finish, at this state the wr_addr of the ram increments, and the Channel ram is filled with data.

When a transfer is occurring the second FSM will assert con_busy, until all the data has been readout from the RAM. This is because the RAM is implemented as a LIFO, and no new channel data can be read until all of the previous channel data has been readout.

Once all the channels have been readout a Read Pointer increment command is issued in st_trsf7 & st_trsf8, which completes the readout of one event.

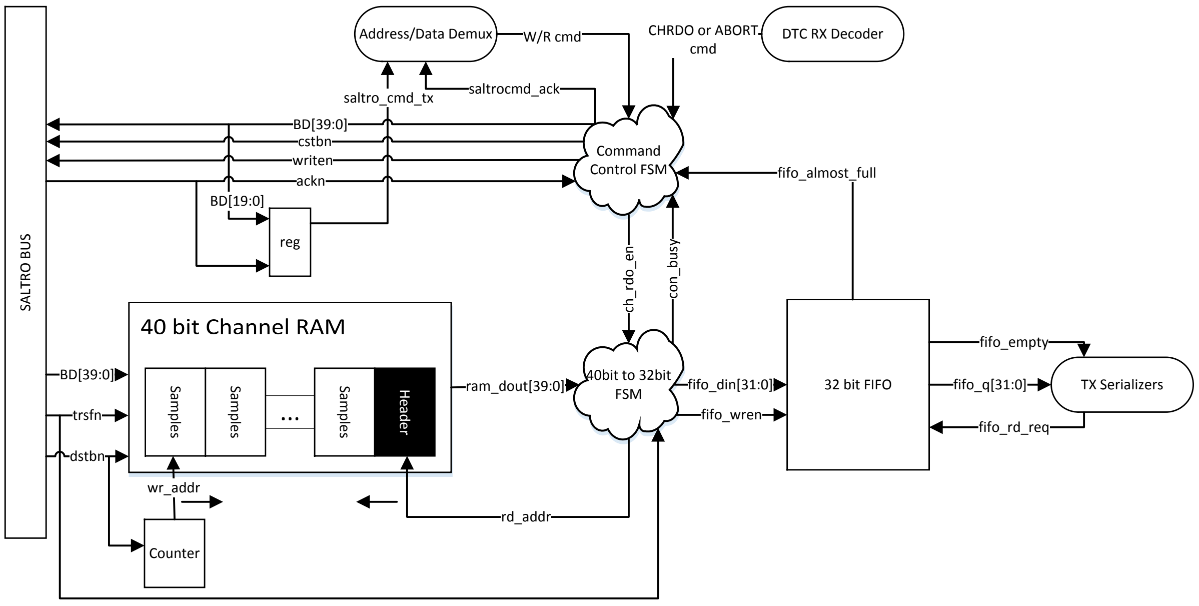

Figure 31: Saltro Controller block diagram.

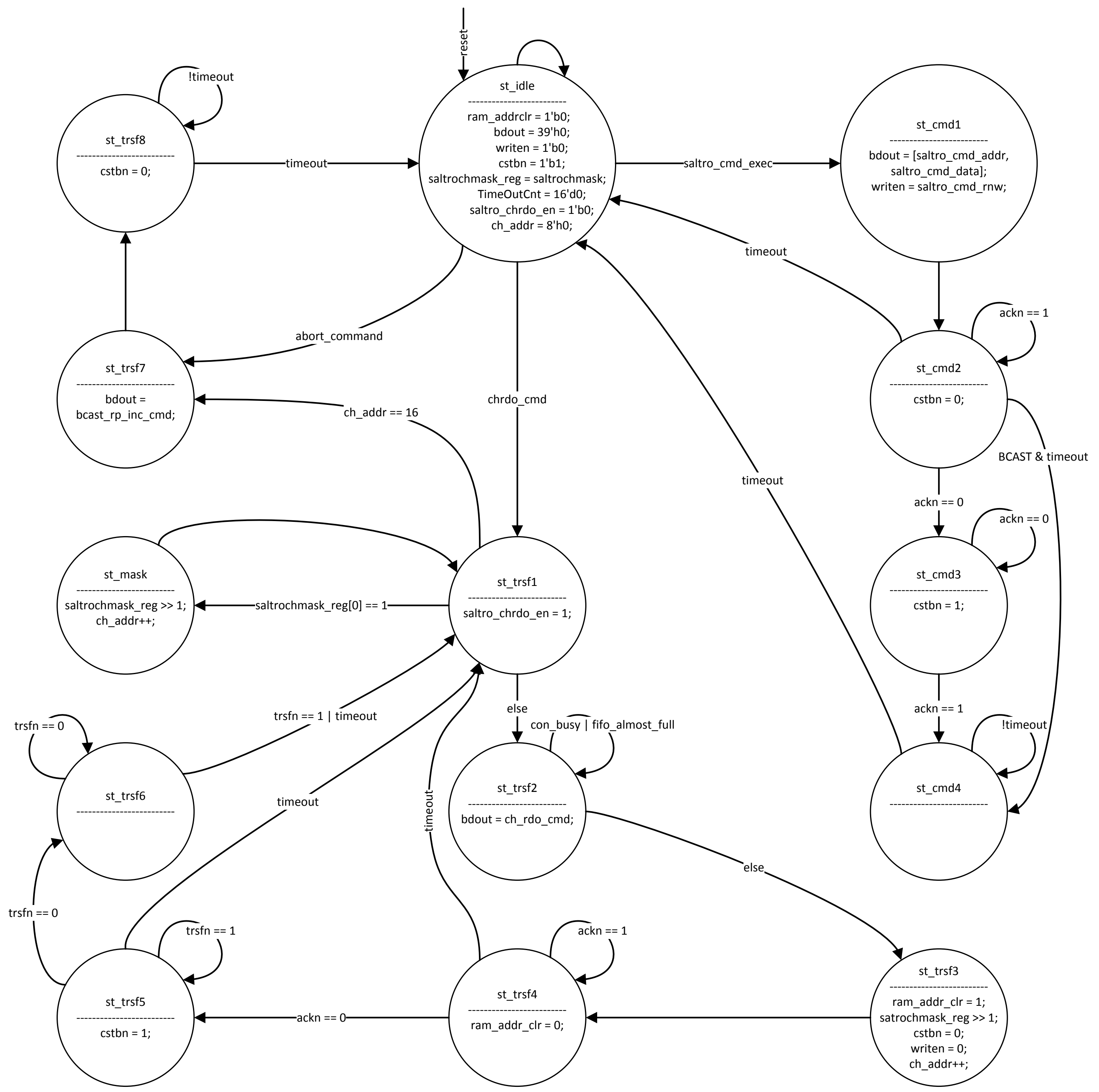

Figure 32: Saltro Controller CC FSM block diagram.

The readout waveforms of the first channel from the SALTRO are shown on fig.33. Once the readout command is issued and the SALTRO chip returns ackn, the transfer of data starts. This causes the wr_addr to increment and con_busy to be asserted. Once the transfer finishes, the fifo is starting to be filled with data. Once all the ram has been readout, the next readout command is issued. On fig.34 is shown a complete readout of the 16 analog channels. During the 5th channel readout, fifo_almost_full is asserted, which prevents the Control Command FSM from issuing readout commands. Thus the spacing between readouts is more spaced away.

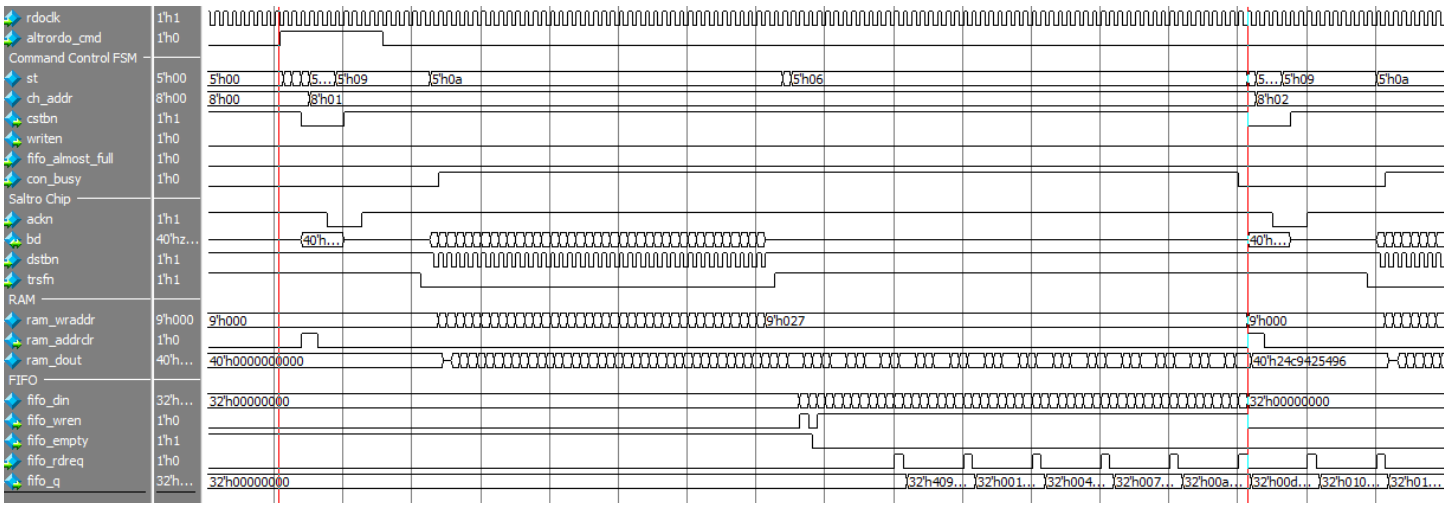

Figure 33: Readout of a single channel.

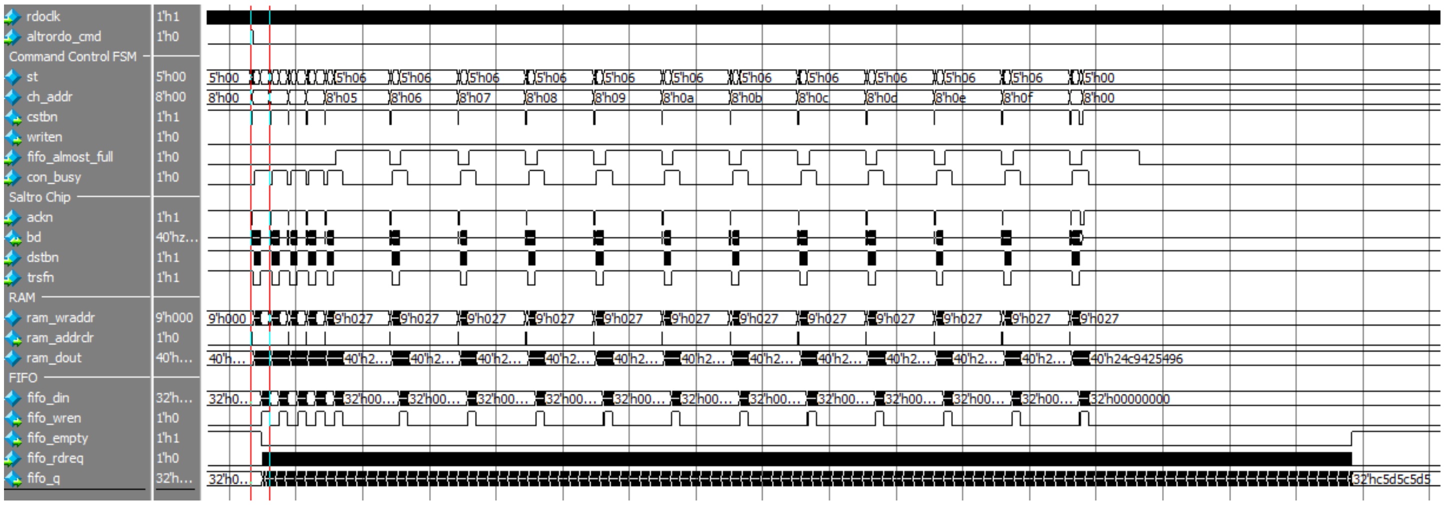

Figure 34: Readout of 16 channels.

f. DTC TX FSM and Serializers

The DTC TX module consists of an FSM, a 2 bit counter, shift register and 2 serializers. On fig.35 is shown the block diagram of the DTC TX module. The FSM is triggered when a Reply for a Read Cmd is received, or a status request, or the event fifo is not empty. The FSM State diagram is shown on fig.36.

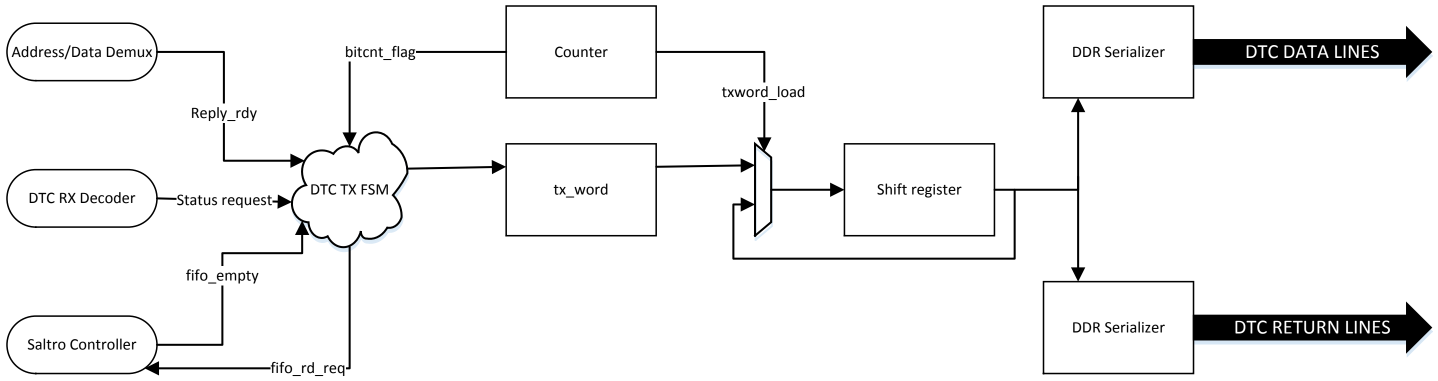

Figure 35: CPLD DTC TX block diagram.

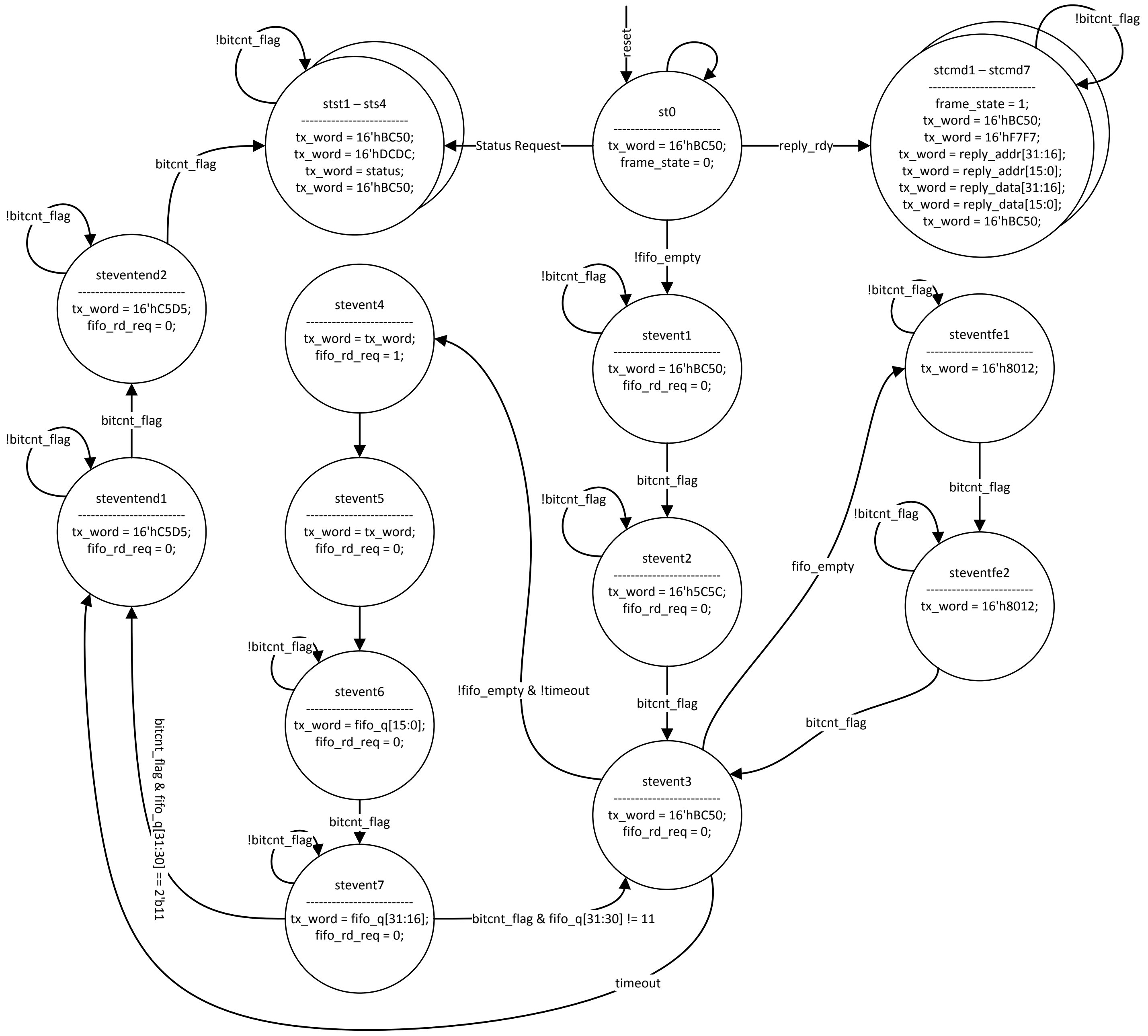

Figure 36: DTC TX FSM state diagram.

If a CSR reply is to be sent, the FSM enters stcmd1 to stcmd7, where on every 4th dtcclk cycle, the tx_word gets updated firstly with the DTC header code for a reply "0xF7F7", then the address and lastly, reply data. Similarly, if a status request is received, the FSM enters stst1 to stst4, where the DTC header code "DCDC", then the status bits (currently only the LSB bit of the status bits are used).

If the event fifo is not empty the FSM enters stevent1, where a synchronizing word BC50 is added in order to separate, the DTC packets. In stevent2, the DTC event header "0x5C5C" is asserted. In stevent3, the event fifo is checked if it is empty. If this is the case, the TX FSM will assert a dummy 32 bit word 32'h80128012 while the Saltro Controller is able to read more channels. In stevent4, a request for data from the fifo is issued. Then stevent5, is added as a delay state, while the fifo_q is available. In st_event6 and stevent7, the data from the fifo is transferred to tx_word. The FSM loops around to stevent3, if more data need to be transmitted. If the event end is detected in stevent7, one more 32bit event trailer word "C5D5C5D5C5D5" is transmitted and a status frame is added. The waveforms for the beginning and end of an event packet transmit are shown on fig.37 and fig.38.

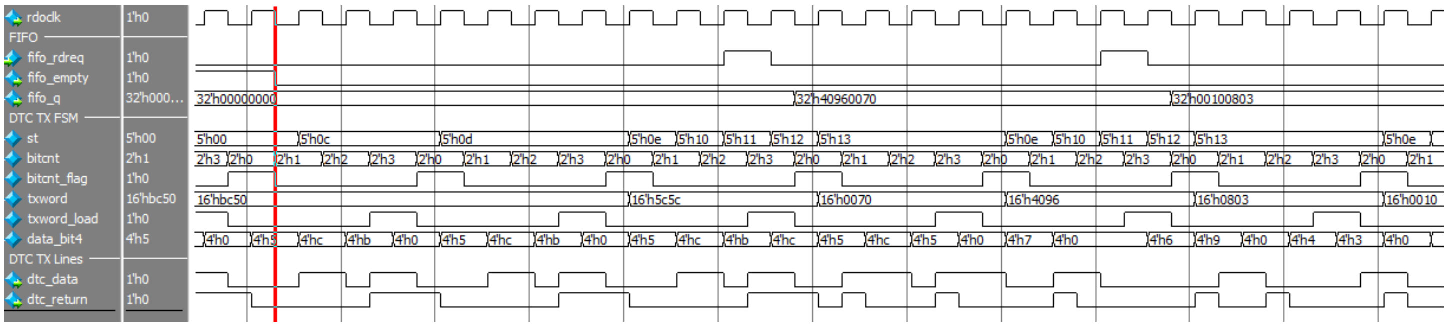

Figure 37: Beginning of event frame transfer.

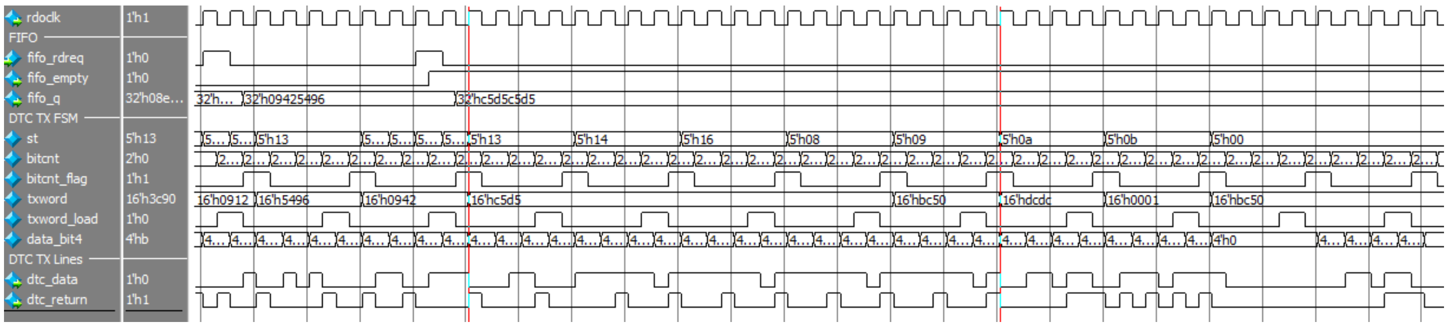

Figure 38: End of event frame transfer.

IV. ASIC Synthesis & Place and Route implementation

a. Synthesis constraints used

- Maximum speed.

The CPLD design was synthesized for maximum speed using an unrealistically fast clock of 2 ns clock period. Due to the CPLD containing proprietary Lattice IPs, they were black-boxed. Thus the area and timing report of the synthesis is unrealistic with only 5.8 % of the area taken by logic and the rest taken by the pads (Table 2).

Table 2: Area report for the ASIC synthesis.

Hierarchy	Max. speed [μm^2]	Percentage
pads (wrapper)	390 237	100
Top	22817	5.8

On table 3 is shown the number of sequential and combinational cells for the CPLD.

Table 3: Cell report for the ASIC synthesis.

Cell type	Number
Combinational	21
Sequential	82

Critical path of the synthesized design is way off than realistically with only 3.07 femto seconds.

b. Place and Route

Place and Route was performed on the synthesized netlist in 65nm technology. The layout is shown on fig.39, which is practically empty, due to the blackboxing of the custom IPs.

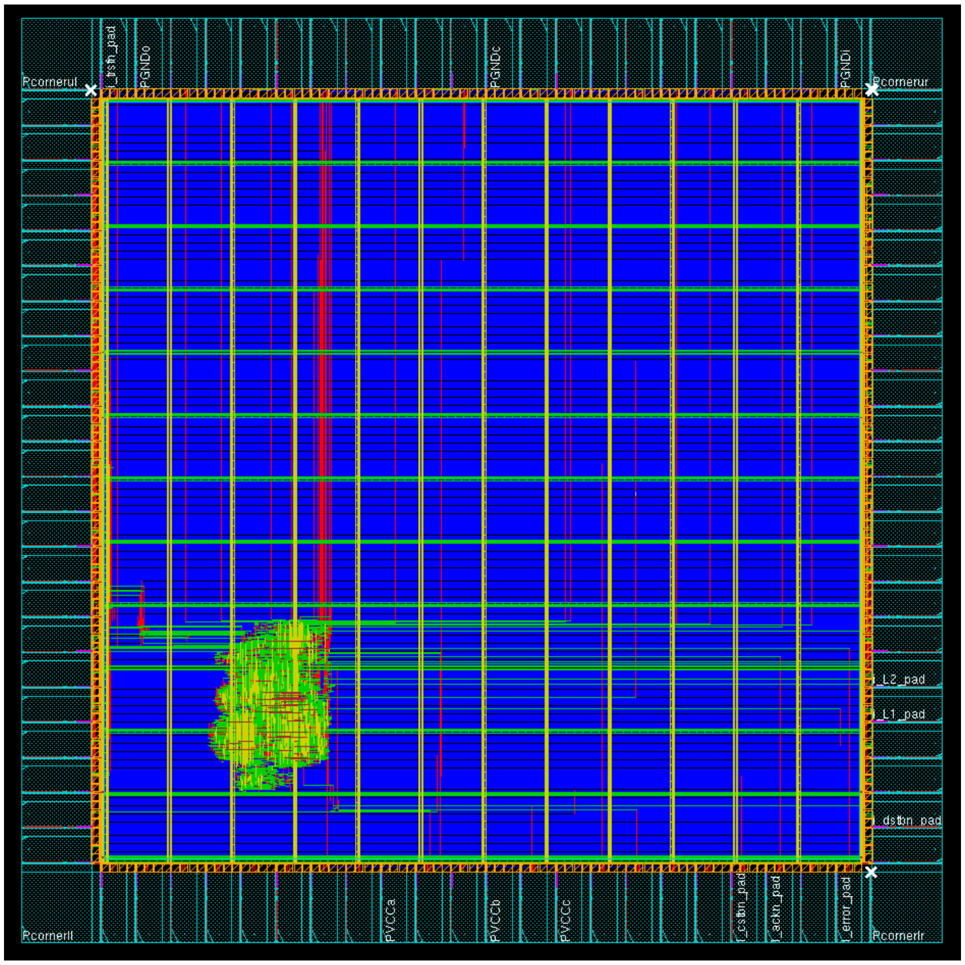

Figure 39: CPLD Layout.

V. FPGA & CPLD Synthesis.

On table 4 is shown the SRU post Place & Route report utilization of the Virtex 6 FPGA.

Table 4: SRU Virtex 6 Utilization.

Feature	Used	Available	Percentage
Slice Registers	37 900	160 000	23%
Slice LUTs	32 913	80 000	41%
Slices	12 181	20 000	60%
RAMB36E1/FIFO36E1	99	264	37%
RAMB18E1/FIFO18E1	6	528	1%
BUFG/BUFGCTRL	16	32	50%
ILOGICE1/ISERDESE1	160	600	26%
OLOGICE1/OSERDESE1	41	600	6%
BSCAN	1	4	25%
BUFR	3	30	10%
EFUSE_USR	1	1	100%
GTXE	2	20	10%
IBUFDS_GTXE1	1	10	10%
MMCM_ADV	2	10	20%
STARTUP	1	1	100%
SYSMON	1	1	100%
TEMAC_SINGLE	2	4	50%

On table 5 is shown the post Place and Route utilization of the Lattice MachXO2 CPLD.

Table 5: Lattice MachXO2 CPLD utilization.

Feature	Used	Available	Percentage
IOLOGIC	24	280	8%
SLICES	862	2160	39%
GSR	1	1	100%
EBR	7	10	70%
PLL	1	2	50%

VI. Implementation Results & Verification

The block diagram of the prototype system implemented in hardware is shown on fig.40. A PC is sending commands using custom software. A 1GBit/s switch is used to route the commands and readout data, since the PC has only 1 LAN card. The SRU is triggers using a special trigger UDP packet. The SRU generates a test pulse is send on a NIM output loaded with 50 Ω. This test pulse can be moved around by software with reference to the L1 trigger. The test pulse wire is taped across the PCB traces of the analog channels, which capacitively couples with the edges of the pulse. Thus is simulates the signals that will be in the detector. In order to verify internal and external signals, Xilinx ChipScope and Lattice Reveal were used with corresponding JTAG programmers.

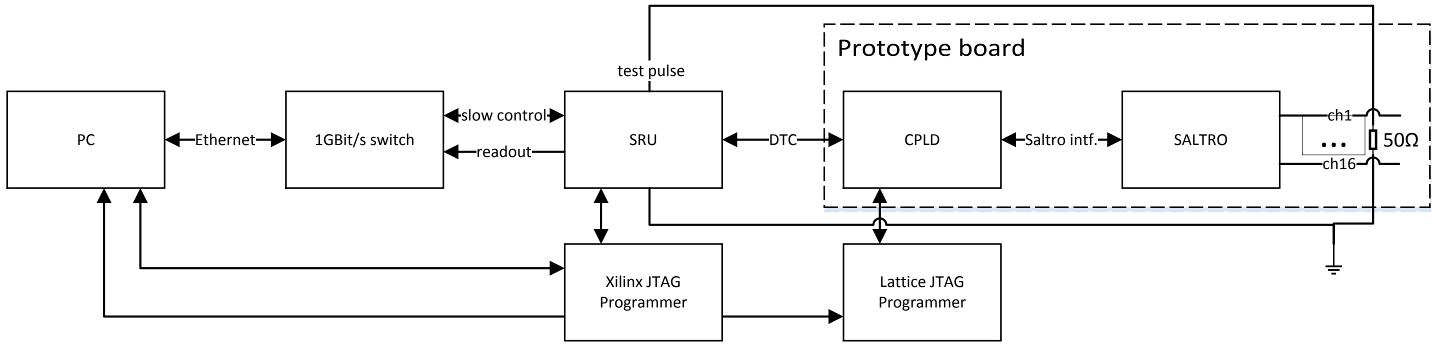

Figure 40: Hardware implementation block diagram.

Due to limitations of how deep the DTC RX memory in the SRU is, and the Readout FSM, only a single jumbo Ethernet frame per event can be readout. This is enough to readout 2-3 channels with up to 1024 samples. Or 16 channels with about 150 channels each. On fig.41 are shown 2 events, with different sampling clock (i.e. the resolution is different), each about 975 samples long.

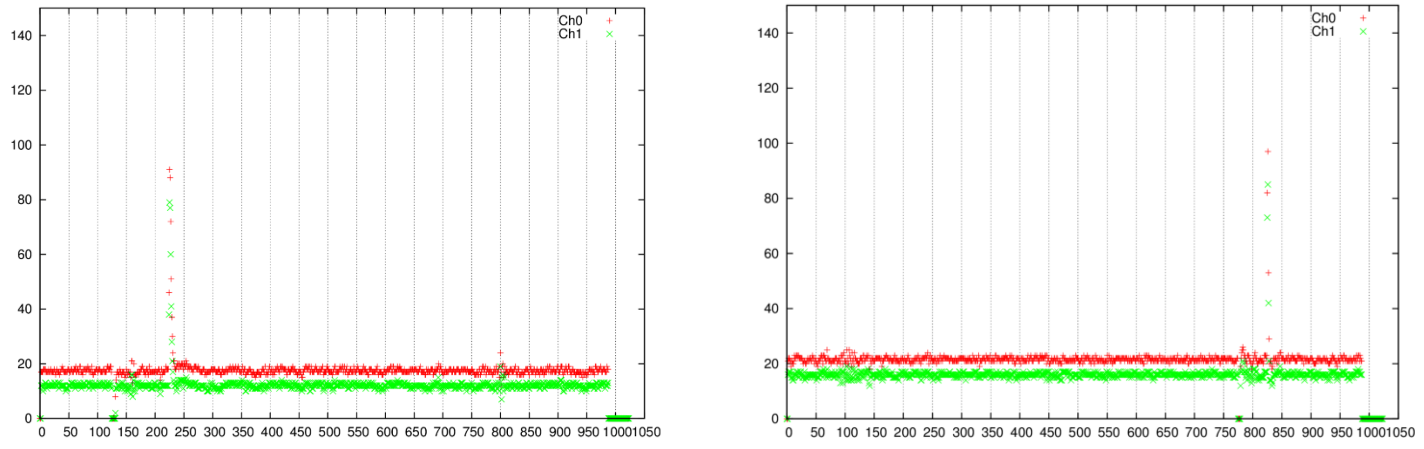

Figure 41: Readout of 2 channels with: a) 20MHz sampling clock b) 10 MHz sampling clock.

VII. Conclusion

It was a nice learning experience with this project and a lot of head scratching was done. The transition to 65nm was relatively painless and took only a couple of days, affirming the notion of reusability. Having experience and scripts from the 130nm synthesis and place & route saved a lot of time.

I learned than having a well-documented design is a must in being successful, since the code I took over, isn't well documented, and I spent a huge amount of time trying to figure out what it was doing.
The work on this projects will continue as my master thesis, by implementing support for 40 DTC boards readout each with 128 channels. And also future improvements could be investigated such as power-saving and error checking.

VIII. Literature

1. SALTRO ADC chip specification: http://www.hep.lu.se/eudet/saltro/S-ALTRO_Documentation-1.pdf
2. DTC Link protocol specification: https://twiki.cern.ch/twiki/pub/Sandbox/FanZhangSRUFiles/DTCLinkProtocolforEMCalPHOS20140311.pdf
3. UDP Command packet specification: http://www.hep.lu.se/eudet/saltro/developdoc/SRUGbEDetectorControlSpecification14032601.pdf

Appendix 1: Synthesis script

```
# Use: source cpld_synth.tcl
analyze -library WORK -format verilog "./bb_rtl_lib/ilvds.v";
analyze -library WORK -format verilog "./bb_rtl_lib/olvds.v";
analyze -library WORK -format verilog "./bb_rtl_lib/pll_adc.v";
analyze -library WORK -format verilog "./bb_rtl_lib/fifo_dc.v";
analyze -library WORK -format verilog "./bb_rtl_lib/ch_ram.v";
analyze -library WORK -format verilog "./bb_rtl_lib/event_fifo.v";
analyze -library WORK -format verilog "../cpld/bitddr.v";
analyze -library WORK -format verilog "../cpld/dtc_tx.v";
analyze -library WORK -format verilog "../cpld/dtc_rx.v";
analyze -library WORK -format verilog "../cpld/dtcTrigDecode.v";
analyze -library WORK -format verilog "../cpld/dtc_cmd.v";
analyze -library WORK -format verilog "../cpld/dtc_top_mcm.v";
analyze -library WORK -format verilog "../cpld/memory_interface.v";
analyze -library WORK -format verilog "../cpld/fed_build.v";
analyze -library WORK -format verilog "../cpld/altro_if.v";
analyze -library WORK -format verilog "../cpld/LGFlagModule.v";
analyze -library WORK -format verilog "../cpld/altro_top.v";
analyze -library WORK -format verilog "../cpld/Mcmtop.v";

analyze -library WORK -format verilog "./rtl_pads/boardcontroller_pad.v";

elaborate boardcontroller_pad -library WORK

# Specify clocks & area constraints..
# High Speed
create_clock -period 2 -name my_dtc_clk dtc_clk_p
set_clock_uncertainty 0.04 my_dtc_clk

set_fix_hold my_dtc_clk

# Do the real synthesis.
compile -map_effort high

change_names -rules verilog -hierarchy

# Save the stuff
write -hierarchy -format ddc -output ./netlists/boardcontroller.ddc
write -hierarchy -format verilog -output ./netlists/boardcontroller.v

write_sdf ./netlists/boardcontroller.sdf
write_sdc ./netlists/boardcontroller.sdc

# Check the design
check_design
report_timing
report_constraint -all_violators
report_area -hierarchy
```

Appendix 2: Place and route script

```
# Source the globals & the corners.
source ./boardcontroller.globals

# Initialize the design
init_design

# Connect the power nets.
clearGlobalNets
globalNetConnect VDD -type tiehi -inst *
globalNetConnect GND -type tielo -inst *
globalNetConnect VDD -type pgpin -pin VDD* -inst *
globalNetConnect GND -type pgpin -pin GND* -inst *

# Set the dimentions of the die.
floorPlan -site CORE -s 1200 1200 20 20 20 20

# Position the board controller instance.
setObjFPlanBox Module i_boardcontroller 200 200 500 500

# Add Power Rings & Slices
set sprCreateIeRingNets {}
set sprCreateIeRingLayers {}
set sprCreateIeRingWidth 1.0
set sprCreateIeRingSpacing 1.0
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
addRing -stacked_via_top_layer M7 -around core -jog_distance 0.4 -threshold 0.4 -nets {GND VDD} -
stacked_via_bottom_layer M1 -layer {bottom M3 top M3 right M4 left M4} -width 2 -spacing 2 -offset 2

# Add the stripes
set sprCreateIeStripeNets {}
set sprCreateIeStripeLayers {}
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeSpacing 2.0
set sprCreateIeStripeThreshold 1.0

addStripe -block_ring_top_layer_limit M5 -max_same_layer_jog_length 0.8 -
padcore_ring_bottom_layer_limit M3 -set_to_set_distance 100 -stacked_via_top_layer M7 -
padcore_ring_top_layer_limit M5 -spacing 2 -merge_stripes_value 0.4 -layer M4 -
block_ring_bottom_layer_limit M3 -width 2 -nets {GND VDD} -stacked_via_bottom_layer M1
addStripe -block_ring_top_layer_limit M4 -max_same_layer_jog_length 0.8 -
padcore_ring_bottom_layer_limit M2 -set_to_set_distance 100 -stacked_via_top_layer M7 -
padcore_ring_top_layer_limit M4 -spacing 2 -merge_stripes_value 0.4 -direction horizontal -layer M3
-block_ring_bottom_layer_limit M2 -width 2 -nets {GND VDD} -stacked_via_bottom_layer M1

# Place standard cells
setPlaceMode -fp false -prerouteAsObs {1 2 3 4}
placeDesign
setDrawView place

# Create clk tree
clockDesign -specFile ./soc/clock.ctstch -outDir clock_report -fixedInstBeforeCTS

# Add Fillers
addFiller -cell HS65_LH_FILLERCELL4 HS65_LH_FILLERCELL3 \
             HS65_LH_FILLERCELL2 HS65_LH_FILLERCELL1 -prefix fico

# Add Filler IO cells
addIoFiller -cell IOFILLERCELL64_ST_SF_LIN -prefix if64
addIoFiller -cell IOFILLERCELL32_ST_SF_LIN -prefix if32
addIoFiller -cell IOFILLER16_ST_SF_LIN -prefix if16
addIoFiller -cell IOFILLER8_ST_SF_LIN -prefix if8
addIoFiller -cell IOFILLER4_ST_SF_LIN -prefix if4
addIoFiller -cell IOFILLER2_ST_SF_LIN -prefix if2
addIoFiller -cell IOFILLER1_ST_SF_LIN -prefix if1

# Connect VDD & GND to the pads.
```

```
sroute -connect { blockPin padPin padRing corePin floatingStripe } -layerChangeRange { M1 M7 } -
blockPinTarget { nearestRingStripe nearestTarget } -padPinPortConnect { allPort oneGeom } -
stripeSCpinTarget { blockring padring ring stripe ringpin blockpin } -checkAlignedSecondaryPin 1 -
blockPin useLef -allowJogging 1 -crossoverViaBottomLayer M1 -allowLayerChange 1 -targetViaTopLayer M7
-crossoverViaTopLayer M7 -targetViaBottomLayer M1 -nets { GND VDD }

# NanoRoute normal wires
setNanoRouteMode -quiet -timingEngine {}
setNanoRouteMode -quiet -routeWithSiPostRouteFix 0
setNanoRouteMode -quiet -routeTopRoutingLayer default
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -drouteEndIteration default
setNanoRouteMode -quiet -routeWithTimingDriven false
setNanoRouteMode -quiet -routeWithSiDriven false
routeDesign -globalDetail

sroute -noBlockPins -noPadRings -jogControl { preferWithChanges differentLayer }

fit

globalDetailRoute

# Save design & netlists and other files.
saveDesign ./soc/boardcontroller.enc
saveNetlist ./netlists/boardcontroller_layout.v

#write_sdf -version 2.1 -interconn nooutport -view NOM ./netlists/boardcontroller_layout_NOM.sdf
#write_sdf -version 2.1 -interconn nooutport -view SS ./netlists/boardcontroller_layout_SS.sdf

#rcOut -spf ./netlists/boardcontroller_ss.spf -rc_corner SS
#rcOut -spef ./netlists/boardcontroller_ss.spef -rc_corner SS
#rcOut -spf ./netlists/boardcontroller_nom.spf -rc_corner NOM
#rcOut -spef ./netlists/boardcontroller_nom.spef -rc_corner NOM
```