Booster Cogging with the MFC VXI Module

Transferring Process Data to ACNET

Memory Maps and Transfer Protocol

**DRAFT** May 9, 2013

Craig Drennan, Kiyomi Seiya, Alex Waller

## Introduction

This document describes the low level logic that manages the transfer of data generated by the Booster Cogging control processes to ACNET for diagnostics and evaluation. It also provides the memory locations and the data transfer protocol that helps ensure data gathered by the front end crate processor is valid.

Section II is a description of the data generated by the Cogging control process.

Section III is a description of the FPGA logical process that copies and buffers the process data for access by the front end crate processor and manages access to this buffer memory by the copying process and the front end processor.

Section IV presents tables listing the memory location of data and status and control registers used in the Cogging application.

## Cogging Control Process Data

Four buffers of 4096, 16 bit signed values are generated by the Cogging control process. These buffers are currently undefined and made available for application development.

## Memory and FPGA Logic for Buffering Data

### III.1 Accessing the Data Buffer

The Cogging control process generates data during Booster beam acceleration. This is an interval of approximately 36 ms following the occurrence of a Booster reset, TClk trigger. Booster resets occur at 15 Hz, at 66 ms intervals. The final 30 ms between the completion of the Booster acceleration cycle and the start of the next is typically the time when control parameters are updated in the VXI LLRF modules. It is this interval in which the Cogging process data is copied into the buffer memory.

Copying the Four buffers of 4096 values takes approximately 1 ms. This is the only time we wish to hold off the front end crate processor from accessing this data. To manage this, a simple protocol of setting a memory request bit and waiting for a memory permit bit to be set is used. Once the front end is finished reading the memory it must clear the memory request bit it set.

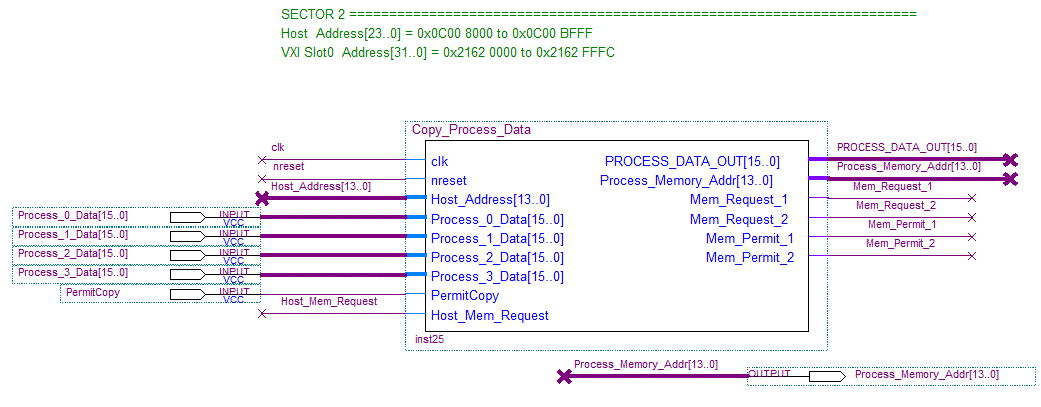
***The front end processor Memory Request bit is set and cleared at BASE\_OFFSET+0x01670004, bit 0.***

***The front end processor Memory Permit bit is read at BASE\_OFFSET+0x01660000, bit 3.***

***The Cogging process data begins at BASE\_OFFSET+0x01620000 to BASE\_OFFSET+0x0162FFFC in address steps of 4 bytes. That is data is at base offset plus 0x01620000, 0x01620004, 0x01620008, etc.***

### III.2 Interfacing to the “Copy\_Process\_Data” Logic Block

The highest level block for the “Copy\_Process\_Data” logic block is shown below.



This logic block copies the contents of four dual port memories, each containing 4096, 16 bit signed values. The PermitCopy signal should be active High from the end of the Booster acceleration interval until the next Booster cycle or just before. The copy process is triggered by the rising edge of PermitCopy. If PermitCopy goes Low before the copy process is complete, the copy process stops immediately and resets it memory address output back to zero. The Process\_Memory\_Addr[11..0] will address 4096 values. All four memorys are addressed at the same time, but a Mux using the address bits [13..12] route the appropriate data into the internal copy/buffer memory.

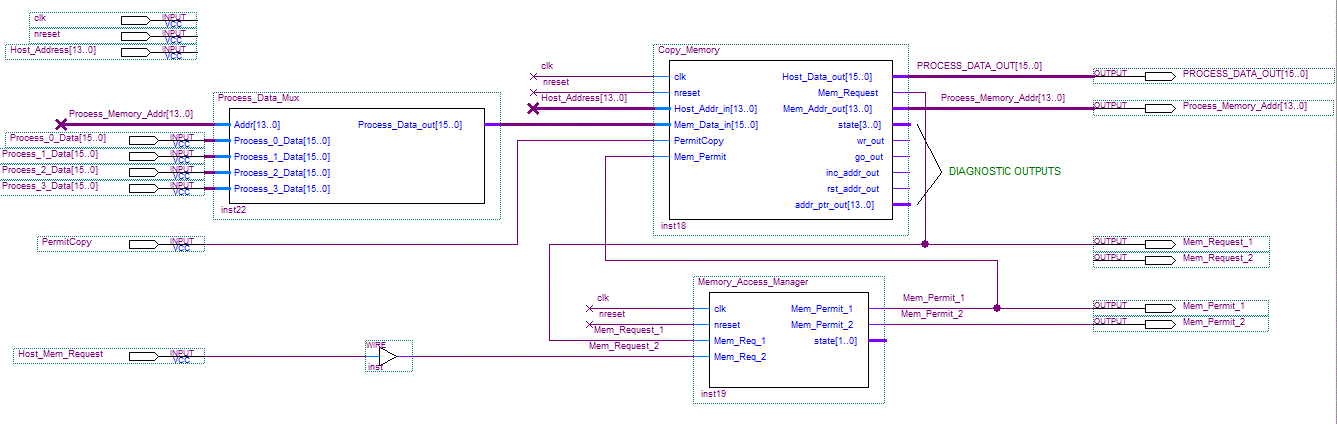
The files to be included in the Quartus project for this logic are

1. Copy\_Process\_Data.bdf
2. Copy\_Process\_Data.bsf
3. Process\_Data\_Mux.vhd
4. Process\_Data\_Mux.bsf
5. Memory\_Access\_Manager.vhd
6. Memory\_Access\_Manager.bsf
7. Copy\_Memory.vhd
8. Copy\_Memory.bsf
9. dualport\_16x16k.vhd
10. Copy\_Memory.mif

These files can currently be found on the Booster.BD drive at

X:\projects\DSP\_Projects\00 - MFC-LLRF Documentation\FPGA Copy Memory Logic Block Files

The figure on the next page gives the block interconnections for the Copy\_Process\_Data logic block.



## Mapped Memory and Registers

Table IV.1 lists the addresses of the memory and registers used by the Cogging application. It also includes some generic memory items that can easily be applied as the Cogging application is developed further.

==== This Table Assumes a Base Offset of the MFC Module of 0x2000 0000. ====

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Item Description | Dimension | Slot 0 Address | | DSP Address |
| Parameter 0 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 0A24  0x204C 0A28  0x204C 0A2C  0x2161 0000 | 0x0023 0289  0x0023 028A  0x0023 028B  0x0C00 4000 |
| Parameter 1 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 0C44  0x204C 0C48  0x204C 0C4C  0x2161 0004 | 0x0023 0311  0x0023 0312  0x0023 0313  0x0C00 4001 |
| Parameter 2 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 0E64  0x204C 0E68  0x204C 0E6C  0x2161 0008 | 0x0023 0399  0x0023 039A  0x0023 039B  0x0C00 4002 |
| Parameter 3 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 1084  0x204C 1088  0x204C 108C  0x2161 000C | 0x0023 0421  0x0023 0422  0x0023 0423  0x0C00 4003 |
| Parameter 4 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 12A4  0x204C 12A8  0x204C 12AC  0x2161 0010 | 0x0023 04A9  0x0023 04AA  0x0023 04AB  0x0C00 4004 |
| Parameter 5 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 14C4  0x204C 14C8  0x204C 14CC  0x2161 0014 | 0x0023 0531  0x0023 0532  0x0023 0533  0x0C00 4005 |
| Parameter 6 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 16E4  0x204C 16E8  0x204C 16EC  0x2161 0018 | 0x0023 05B9  0x0023 05BA  0x0023 05BB  0x0C00 4006 |
| Parameter 7 | 1 | Flag:  Reading:  Setting:  FPGA Direct: | 0x204C 1904  0x204C 1908  0x204C 190C  0x2161 001C | 0x0023 0641  0x0023 0642  0x0023 0643  0x0C00 4007 |
| Cogging Process Data Set 0 (1) | 4096 |  | 0x2162 0000  To  0x2162 3FFF | 0x0C00 8000  To  0x0C00 8FFF |
| Cogging Process Data Set 1 (2) | 4096 |  | 0x2162 4000  To  0x2162 7FFF | 0x0C00 9000  To  0x0C00 9FFF |
| Cogging Process Data Set 2 (3) | 4096 |  | 0x2162 8000  To  0x2162 BFFF | 0x0C00 A000  To  0x0C00 AFFF |
| Cogging Process Data Set 3 (4) | 4096 |  | 0x2162 C000  To  0x2162 FFFF | 0x0C00 B000  To  0x0C00 BFFF |
| Slot 0 Memory Request Bit |  | Bit 0 | 0x2167 0004 | 0x0C01 C001 |
| Slot 0 memory Permit Bit |  | Bit 3 | 0x2166 0000 | 0X0C01 8000 |

## APPENDIX A. VHDL For The Memory Access Manager

Library IEEE;

USE IEEE.std\_logic\_1164.all;

USE IEEE.std\_logic\_arith.all;

USE ieee.std\_logic\_unsigned.all;

entity Memory\_Access\_Manager is

port( clk : in std\_logic;

nreset : in std\_logic;

Mem\_Req\_1 : in std\_logic;

Mem\_Req\_2 : in std\_logic;

Mem\_Permit\_1 : out std\_logic;

Mem\_Permit\_2 : out std\_logic;

state : out std\_logic\_vector(1 downto 0)

);

end Memory\_Access\_Manager;

architecture main of Memory\_Access\_Manager is

signal timer : std\_logic\_vector(23 downto 0);

signal timer\_ena : std\_logic;

signal syncd\_Req\_1 : std\_logic;

signal syncd\_Req\_2 : std\_logic;

signal q1,q2 : std\_logic;

-- State Machine declarations ------------------------------------

type STATE\_TYPE is

(idle, permit1, permit2, idle2);

signal CS, NS : STATE\_TYPE;

-- ## MAIN ##############################################################

begin

SYNC\_REQUESTS:

process(clk, nreset, Mem\_Req\_1, Mem\_Req\_2, q1, q2)

begin

if (nreset = '0') then q1 <= '0';

q2 <= '0';

elsif rising\_edge(clk) then

q1 <= Mem\_Req\_1;

q2 <= Mem\_Req\_2;

syncd\_Req\_1 <= q1;

syncd\_Req\_2 <= q2;

end if;

end process;

TIMER\_COUNT: process(clk, nreset, timer\_ena)

begin -- timer resets when not enabled

if (nreset = '0') then timer <= (others => '0');

elsif rising\_edge(clk) then

if (timer\_ena = '1') then timer <= timer + 1;

else timer <= (others => '0');

end if;

end if;

end process;

----------------------------------------------------------

----------------------------------------------------------

sm\_reg: process (clk, nreset)

begin

if(nreset='0') then CS <= idle;

elsif(clk'event and clk='1')then CS <= NS;

end if;

end process; -- sm\_reg

sm\_combinatorial:

process (CS, timer, syncd\_Req\_1, syncd\_Req\_2)

begin

timer\_ena <= '0';

Mem\_Permit\_1 <= '0';

Mem\_Permit\_2 <= '0';

case CS is

when idle =>

state <= "00";

if (syncd\_Req\_1 = '1') then NS <= permit1;

elsif(syncd\_Req\_2 = '1') then NS <= permit2;

else NS <= idle;

end if;

when permit1 =>

state <= "01";

Mem\_Permit\_1 <= '1';

timer\_ena <= '1';

if (timer > X"27AC40") then NS <= idle2; -- 40ms timeout

elsif(syncd\_Req\_1 = '0') then NS <= idle2; -- alternate idle

else NS <= permit1;

end if;

when permit2 =>

state <= "10";

Mem\_Permit\_2 <= '1';

timer\_ena <= '1';

if (timer > X"27AC40") then NS <= idle; -- 40ms timeout

elsif(syncd\_Req\_2 = '0') then NS <= idle;

else NS <= permit2;

end if;

when idle2 => -- reverses the priority of the requests

state <= "11";

if (syncd\_Req\_2 = '1') then NS <= permit2;

elsif(syncd\_Req\_1 = '1') then NS <= permit1;

else NS <= idle2;

end if;

when others => NS <= idle;

state <= "11";

end case;

end process;

end main;