AXI4-Lite communication between PL and PS

Applied to the data broadcast from the Charge Monitoring Board, through the EMP, to the Control Room

Context about EMP function in the CMB project

The EMP is a board developed at CERN, and in the Charge Monitoring System project, its function is to concentrate the data sent by multiple Charge Monitoring Boards and sent that information to the control Room. Each EMP will control 12 EMCIs, and each EMCI will handle 4 Charge Monitoring Boards.

Each EMP will be connected to, at most, 12 EMCIs, and each EMCI will input information using the lgGBT chip. This chip was customly designed at CERN, and enables optical fiber communication between the Charge Monitoring Boards and the EMP. As was presented in the document titled “Lpgbt communication explanation”, the data frame received from EMCI into the EMP has a particular structure that will depend on the communication frequency; for further information about that data structure, consult that document.

Each EMCI will deliver 230 bits into the EMP, 224 of them contain effectively data, while the rest are just zeros. Those 230 bits are the ones that must be sent from EMP’s Programmable Logic (its firmware) to its Processing System (its Petalinux). The HDL and software described in this document handles the communication that delivers those 230 bits to the Processing System.

Important conditions and requirements from AXI4-Lite IP block developed

1. The IP Core developed to communicate the Programmable Logic (PL) with the Processing System (PS) needs that each Charge Monitoring Board keeps sending the data they want to broadcast until they receive a signal to stop broadcasting.

1. The reason for this is that in that way it is not necessary to implement a memory in the EMP to store the data sent in one clock cyle by al the Charge Monitoring Boards; something useful as there is a risk to run out of space in EMP's, in terms of available flops.

2. AXI4-Lite IP block is based on a template automatically generated by Vivado, and that template was originally in Verilog; because of that, al the block is written in Verilog. This should be no problem for the user, as this IP block should not be opened by the user.

IP Core that implements AXI4-Lite communication from PL to PS

The IP Core that implements AXI4-Lite to communicate data from PL to PS is created starting from a template given by Vivado, but the user must modify it to get the communication he wants. This IP is named "Bin2AXI_slave_v3".

At this momment, the implemented IP Core has 9 internal registers. All the registers can be read by the Programmable Logic (PL) and the Processing System (PS), one of the registers can be written by the Processing System, and the rest can be written by the Programmable Logic. The summary of their names and the read/write permissions are in the table below:

PL permissions	PS permissions	Register name
read	read & write	slv_reg0
read & write	read	slv_reg1
read & write	read	slv_reg2
read & write	read	slv_reg3
read & write	read	slv_reg4
read & write	read	slv_reg5
read & write	read	slv_reg6
read & write	read	slv_reg7
read & write	read	slv_reg8

The register named slv_reg0 is used to send information from the Processign System to the IP Bin2AXI_slave_v3. At this moment[1], this register is used by the Processing System to command Bin2AXI_slave_v3 to start taking data, i.e. to start storing input data into their registers slv_reg2 to slv_reg8

1 i.e. 2024/08/26

Register slv_reg1 is used by Bin2AXI_slave_v3 to send information to the Processing System. This is used, at this moment[2], to indicate to the Processing System that registers are already full of information, and also to indicate that the system is ready to capture new information.

Registers from slv_reg2 to slv_reg8 are used to store the input data from lpGBT, so after the saving process, the PS can read them through AXI4-Lite protocol.

The exact implementation of AXI4-Lite protocol is done by the template given by Vivado. However, the saving of input information in specific registers, and the registers that command the handshakes between the Programmable Logic and the Processing System, are implemented by a custom Finite State Machine (FSM). Inside this FMS it is set the value sent and received by the IP block; registers slv_reg0 and slv_reg1. The state diagram for this FSM is at the Appendix.

What happens in each state of the FSM is the following:

1. **idle_ul**: This is the idle state, where Bin2AXI_slave_v3 is waiting to receive the order, from the PS, to start saving data. If register slv_reg1 equals 32'hFFFFFFFF, that means that all the data registers (i.e. from slv_reg2 to slv_reg8) already have data stored on them, so this data must be read by PS before initiate a new data capture; if slv_reg1 equals 0, that means Bin2AXI_slave_v3 is ready to capture new data. If register slv_reg0 equals 0 that means that PS does not want to capture data, but if it is 1, it means PS wants to capture data.
    1. Here there is an important caveat, when data capture ends, and register slv_reg1 = 32'hFFFFFFFF, the system will enter in capture data mode if and only if there is at least one cycle where slv_reg0 = 0. This safe measure ensures that it is not possible to enter into a continous data capture cycle if the Processing System stays with register slv_reg0 at 1 for a long period of time. If PS wants to start a new capture of data, it must take slv_reg0 to zero before make it 1 again.
2. **data_reg1 to data_reg7**: These states are the ones where data is stored at registers inside Bin2AXI_slave_v3. At state “data_reg1” input 32-bits data is stored at register “slv_reg2”, at state “data_reg2” input 32-bits data is stored at register “slv_reg3”, and so on, until state “data_reg7” stores input data in register “slv_reg8”.

2 i.e. 2024/08/26

3. **wr_ready_ul**: In this state, the register slv_reg1 is set to 32'hFFFFFFFF, which indicates that the input data has filled all data registers, and the PS must read that data before start a new data capture process.

PS C++ code to read data sent from PL to PS

The exact software that must be used to extract the data from the registers will depend on the Trenz board used and its available processing system. However, in the example implemented, by consulting the tab “Address Editor” available when working at step “IP Integrator” from the left-hand design flow menu, it is possible to see that the data sent from the block Bin2AXI_slave_v3 will be assigned to Processign System’s addresses from 0x43C0_0000 to 0x43C0_FFFF. You can do the same with your own design and Trenz board.

After knowing what registers match are assigned to the AXI4-Lute communication in your board, you can start reading the data from PL.

There is a 1-1 mapping between each Programmable Logic register and its corresponding register at the Processing System. This mapping is as follows:

Register name at PL	Memory address at PS
slv_reg0	Base register (0x43C0_0000 in this example)
slv_reg1	Base register + 4
slv_reg2	Base register + 8
slv_reg3	Base register + 12
slv_reg4	Base register + 16
slv_reg5	Base register + 20
slv_reg6	Base register + 24
slv_reg7	Base register + 28
slv_reg8	Base register + 32

By reading the addresses in the “Memory address at PS” column in your PS, you can access the corresponding data at their related registers. To read the wanted memory address, you can use the function Xil_In32(<ADDRESS>), and to write in a memory address, you can use the function Xil_Out32(<ADDRESS>, <VALUE>).

An example code to read data that is static at the input of Bin2AXI_slave_v3, i.e. it has no control over the input data to Bin2AXI_slave_v3 and it is constantly reading the data, is:

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "ps7_init.h"
#include "xparameters.h"
#include "xil_io.h"


int main()
{
    init_platform();
    ps7_post_config();

    /* Definitions for peripheral BIN2AXI_SLAVE_0 */
    //#define XPAR_BIN2AXI_SLAVE_0_BASEADDR 0x43c00000
    //#define XPAR_BIN2AXI_SLAVE_0_HIGHADDR 0x43c0ffff


    ////////////////////////////////
    ///// Instrucciones para leerlo
    // 1) Escribir un 1 en el registro base
    // 2) Escribir un 0 en el registro base
    // 3) Leer el segundo registro
    // 3.1) Si el valor del registro es 0, entonces hacer nada
    // 3.2) Si el valor del registro es 32'hFFFFFFFF (0xFFFFFFFF en C++), leer los registros de
interes.


    uint32_t regs_listos_leer = 0;
    uint32_t data_reg1 = 0;
    uint32_t data_reg2 = 0;
    uint32_t data_reg3 = 0;
    uint32_t data_reg4 = 0;
    uint32_t data_reg5 = 0;
    uint32_t data_reg6 = 0;
    uint32_t data_reg7 = 0;

    volatile uint32_t *register_ptr_0 = (uint32_t *)0x43c00000;
    volatile uint32_t *register_ptr_1 = (uint32_t *)0x43c00004;

    while (1) {
        uint32_t observa_reg0 = *register_ptr_0;
        uint32_t observa_reg = *register_ptr_1;
        regs_listos_leer = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 4);

        if (regs_listos_leer == 0xFFFFFFFF){
            // Reads data from input registers
            data_reg1 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 8);
            data_reg2 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 12);
            data_reg3 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 16);
            data_reg4 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 20);
            data_reg5 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 24);
            data_reg6 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 28);
            data_reg7 = Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 32);
            // Print the data
            //for (unsigned int i_regs=0; i_regs < 7; i_regs++) {
            //    xil_printf("value at address %x is %x\n", XPAR_BIN2AXI_SLAVE_0_BASEADDR + 8 +
i_regs*4, Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 8 + i_regs*4));
```

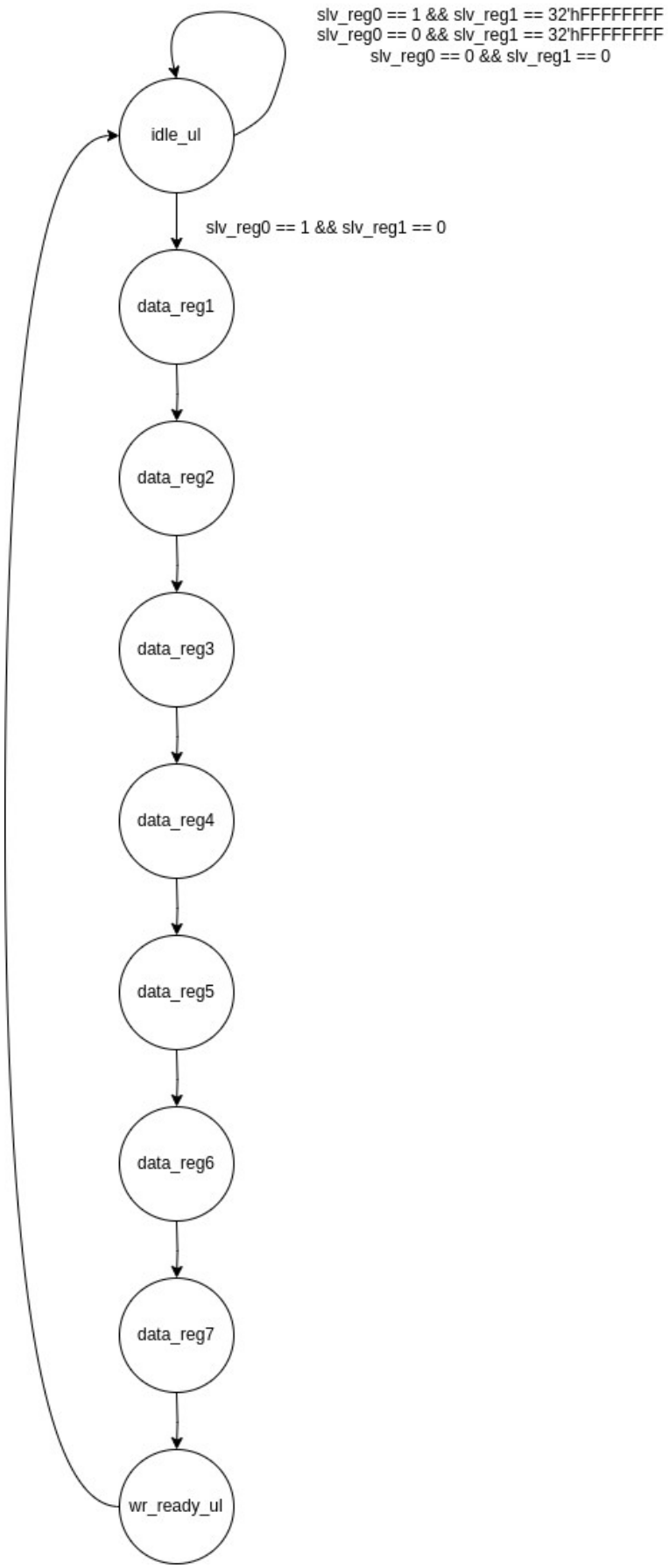
```
            //}

            for (unsigned int i_regs=0; i_regs < 7; i_regs++) {
                xil_printf("%lu\n", Xil_In32(XPAR_BIN2AXI_SLAVE_0_BASEADDR + 8 + i_regs*4));
            }

            // Write register reg0 = 0; this reg equals to 0 indicates that I don't want to aquire data
            Xil_Out32(XPAR_BIN2AXI_SLAVE_0_BASEADDR, 0);
        }
        else if (regs_listos_leer == 0) {
            // Write the register reg0 = 1; this indicates that I want to aquire data
            Xil_Out32(XPAR_BIN2AXI_SLAVE_0_BASEADDR, 1);
        }
        else {
            // Error, as reg0 took a value different to 0 or 0xFFFFFFFF
            printf("Error!!. Register associated to BIN2AXI_SLAVE_0, which is written by PL and read by
PS, took a value different to 0 or 0xFFFFFFFF");
            return 99;
        }
    }

    cleanup_platform();
    return 0;
}
```

Appendix


slv_reg0 == 1 && slv_reg1 == 32'hFFFFFFFF
slv_reg0 == 0 && slv_reg1 == 32'hFFFFFFFF
slv_reg0 == 0 && slv_reg1 == 0
idle_ul
slv_reg0 == 1 && slv_reg1 == 0
data_reg1
data_reg2
data_reg3
data_reg4
data_reg5
data_reg6
data_reg7
wr_ready_ul