

POLITECNICO DI TORINO

**Corso di Laurea Magistrale in
MECHATRONIC ENGINEERING (INGEGNERIA MECCATRONICA)**

Tesi di Laurea Magistrale
**Study and development of a real-time
monitoring system for parametric experimental
testbed**



Appendix I

Supervisors

Prof. Luca Sterpone

.....
Prof. Boyang Du

Student
Patricia Arribas Guerrero

Index

Appendix I.1 Monitor board: Sending information module	1
Appendix I.2 Monitor board: Receiving information module	3
Appendix I.3 Monitor board: Software application	6
Appendix I.4 DUT board: receiving information module	18
Appendix I.5 DUT board: sending information module	22
Appendix I.6 Unit Under Test module: Shift register	25

This appendix includes all the code generated for this project. Each section has the VHDL or C lines that were used to create the modules and the software application. To understand the functioning of each function, go to the corresponding chapter of the document that collects all the working principles.

Appendix I.1 Monitor board: Sending information module

This module is the one designed for the process of sending the inputs for the test from the monitor to the DUT board. The language used is VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Communication_module_input is
generic(
    NumberInputs_MON:integer :=8;
    --The max size of package that can be received or sent is 16
    MaxSizePack_MON: integer :=8);

Port ( clk: in STD_LOGIC;
       Reset_MON: in STD_LOGIC;
       Load: in STD_LOGIC;
       flag_load: in STD_LOGIC;

       Enable_in_MON: out STD_LOGIC;
       led_input: out STD_LOGIC;
       Ready_in_MON: in STD_LOGIC;

       Input_MON: in STD_LOGIC_VECTOR(MaxSizePack_MON-1 downto 0);

       Data_MON: out STD_LOGIC_VECTOR(MaxSizePack_MON-1 downto 0)
);
end Communication_module_input;

architecture Behavioral of Communication_module_input is
begin

    Send_Data: process(Reset_MON, clk) is
        variable start_flag: natural range 0 to 1:=0;
        variable time_flag: natural range 0 to 1:=0;
    begin

        if(Reset_MON='1') then
            Data_MON<=(others=>'0');
            Enable_in_MON<='0';
            Start_flag:=0;
            led_input<='0';
        end if;
        if(flag_load='1') then
            if(start_flag=0) then
                time_flag:=1;
            else
                time_flag:=time_flag+1;
            end if;
            if(time_flag=MaxSizePack_MON) then
                start_flag:=1;
                time_flag:=0;
            end if;
        end if;
        if(Enable_in_MON='1') then
            Data_MON<=(Input_MON);
        end if;
        if(Ready_in_MON='1') then
            Enable_in_MON<='0';
        end if;
        if(flag_load='0') then
            if(start_flag=1) then
                led_input<='1';
            else
                led_input<='0';
            end if;
        end if;
    end process;
end Behavioral;
```

```
elsif(clk'event and clk='1') then  
  
    --When the software application has loaded the data in the  
    signal, it is transferred to the channel  
    if(Load='1' and start_flag=0) then  
        --The flag is set to one, since this is executed only once  
        start_flag:=1;  
        Data_MON<=Input_MON;  
  
    --It is waited a clock cycle to assure the data is loaded in  
    the channel  
    elsif(flag_load='1' and Ready_in_Mon='0')then  
        Enable_in_mon<='1';  
  
    --After a clock cycle, the enable is set to zero again  
    elsif(flag_load='0') then  
        Enable_in_MON<='0';  
        start_flag:=0;  
  
    end if;  
end if;  
  
end process Send_Data;  
  
end Behavioral;
```

Appendix I.2 Monitor board: Receiving information module

This module is the one designed for the process of receiving the results of the test from the DUT. The language used is VHDL.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Communication_module_output is
generic(
    NumberOutputs_MON:integer :=8;
    --The max size of package that can be received or sent is 16
    MaxSizePack_MON: integer :=8);

Port (
    clk : in STD_LOGIC;
    Reset_MON: in STD_LOGIC;

    data_available: in STD_LOGIC;
    Ready_MON: out STD_LOGIC;

    store: out STD_LOGIC;
    Flag_store: in STD_LOGIC;

    led_com: out STD_LOGIC;

    end_transmission: out STD_LOGIC

);
end Communication_module_output;

architecture Behavioral of Communication_module_output is

    constant div_output: integer :=NumberOutputs_MON REM MaxSizePack_MON;

    function rest(div:integer)
        return integer is
    begin
        if div=0 then
            return 0;
        else
            return 1;
        end if;
    end function;

    constant NumberOPack: integer := (NumberOutputs_MON/MaxSizePack_MON)
+1*rest(div_output);

begin

-----
Receiving_process: process (clk, Reset_MON,data_available)
-----

```

```

variable Ready_MON_flag: natural range 0 to 1:=0;
variable numpackage: natural range 1 to 100:= 1;
variable flag_end_transmission: natural range 0 to 1:=0;
variable flag_end_store: natural range 0 to 1:=0;
begin

    if(reset_MON='1') then
        Ready_MON<='0';
        led_com<='0';
        end_transmission<='0';
        Ready_MON_flag:=0;
        flag_end_transmission:=0;
        numpackage:=1;
        flag_end_store:=0;

    elsif(clk'event and clk='1') then
        --This condition is triggered to turn off the led that
        indicates the end of the process
        if(data_available='1' and numpackage=1) then
            led_com<='0';
            end if;

        --When the DUT has loaded the results from the test, the
        monitor must store them
        if(data_available='1' and Ready_MON_flag=0 and
        numpackage<NumberOPack+1) then
            Ready_MON_flag:=1;
            numpackage:=numpackage+1;
            store<='1';

        --When the software application has saved the data, the ready
        signal must be activated
        elsif(Flag_store='1' and data_available='0' and
        flag_end_store=0) then
            store<='0';
            Ready_MON<='1';
            Ready_MON_flag:=0;
            flag_end_store:=1;

        --When all the packages have been stored, the process sets the
        flag to end it
        elsif(data_available='1' and numpackage>NumberOPack and
        flag_end_transmission=0) then
            flag_end_transmission:=1;

        --This last condition sets the signal that interrupts the SA
        and re-starts the test
        elsif(flag_end_transmission=1 and data_available='0') then
            end_transmission<='1';
            flag_end_transmission:=0;
            numpackage:=1;
            Ready_MON_flag:=0;
            led_com<='1';

```

```
    elsif (Flag_store='0') then
        Ready_MON<='0';
        flag_end_store:=0;
        end_transmission<='0';

    end if;

end if;

end process Receiving_process;

end Behavioral;
```

Appendix I.3 Monitor board: Software application

This is the software application implemented in the monitor board. It is designed in SDK and the language used is C.

```
/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * -----
 * | UART TYPE      BAUD RATE |
 * -----
 * |   uartns550    9600          |
 * |   uartlite     Configurable only in HW design |
 * |   ps7_uart     115200 (configured by bootrom/bsp) |
 */
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

#include "xil_io.h"
#include "xparameters.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "sleep.h"

#include "xbram.h"

#define GPIO0_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID //START
#define GPIO1_DEVICE_ID XPAR_AXI_GPIO_1_DEVICE_ID //READY
#define GPIO2_DEVICE_ID XPAR_AXI_GPIO_2_DEVICE_ID //LOAD
#define GPIO3_DEVICE_ID XPAR_AXI_GPIO_3_DEVICE_ID //FLAG_LOAD
#define GPIO4_DEVICE_ID XPAR_AXI_GPIO_4_DEVICE_ID //DATA
#define GPIO5_DEVICE_ID XPAR_AXI_GPIO_5_DEVICE_ID //LOAD
#define GPIO6_DEVICE_ID XPAR_AXI_GPIO_6_DEVICE_ID //FLAG_STORE
#define GPIO7_DEVICE_ID XPAR_AXI_GPIO_7_DEVICE_ID //RPI_IN
#define GPIO8_DEVICE_ID XPAR_AXI_GPIO_8_DEVICE_ID //END_TRANSMISSION

#define INTC_DEVICE_ID XPAR_PS7_SCUGIC_0_DEVICE_ID

#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define INTC_GPIO_INTERRUPT_ID1 XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR
#define INTC_GPIO_INTERRUPT_ID5 XPAR_FABRIC_AXI_GPIO_5_IP2INTC_IRPT_INTR
#define INTC_GPIO_INTERRUPT_ID8 XPAR_FABRIC_AXI_GPIO_8_IP2INTC_IRPT_INTR

#define Start_INT XGPIO_IR_CH1_MASK

#define NumberofInputs 16
#define NumberofOutputs 16
```

```

#define MaxSizePackage 8

#define NPacksInput 2
#define NPacksOutput 2

XScuGic INTCInst;
static XBram Bram;
XGpio START, READY, LOAD, FLAG_LOAD, DATA, STORE, FLAG_STORE, RPI_IN,
END_TRANSMISSION;

int flag_start=0;
int flag_ready=0;
int flag_store=0;
int flag_end_transmission=0;
int package=1;

int number_word_input=0;
int number_word_output=12;

//-----
// PROTOTYPE FUNCTIONS
//-----
static void START_Intr_Handler(void *baseaddr_p);
static void READY_Intr_Handler(void *baseaddr_p);
static void STORE_Intr_Handler(void *baseaddr_p);
static void END_TRANSMISSION_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DevideId, XGpio *GpioInstancePtr, XGpio
*GpioInstancePtr2, XGpio *GpioInstancePtr3,XGpio *GpioInstancePtr4);

/*************
** Initialization BRAM**
*****/

int init_bram(void){

    int Status;
    XBram_Config *ConfigPtr;

    ConfigPtr = XBram_LookupConfig(XPAR_AXI_BRAM_CTRL_0_DEVICE_ID);

    if (ConfigPtr == (XBram_Config *) NULL) {
        return XST_FAILURE;
    }

    Status = XBram_CfgInitialize(&Bram, ConfigPtr, ConfigPtr-
>CtrlBaseAddress);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    return XST_SUCCESS;
}

```

```

void init_value(void){
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+0,0x04); //0x08
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+1,0x02); //0x04
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+2,0x0A); //0x14
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+3,0x01); //0x02
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+4,0x08); //0x10
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+5,0x03); //0x06
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+6,0x0B); //0x08
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+7,0x05); //0x0a
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+8,0x07); //0x0c
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+9,0x13); //0x0e
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+10,0x09); //0x10
    XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+11,0xF1); //0x12

}

/*****************/
/** Initialization GPIO**/
/*****************/

int init_GPIO0(void){

    int status;

    // Initialize Enable_in (GPIO0)
    status = XGpio_Initialize(&START, GPIO0_DEVICE_ID);
    if(status != XST_SUCCESS) {
        print("Initialization of GPIO0 failed");
        return XST_FAILURE;
    }
    // Set all buttons direction to input
    XGpio_SetDataDirection(&START, 1, 0xff);

    return XST_SUCCESS;
}

int init_GPIO1(void){

    int status;

    // Initialize DATA_IN1 (GPIO1)
    status = XGpio_Initialize(&READY, GPIO1_DEVICE_ID);
    if(status != XST_SUCCESS) {
        print("Initialization of GPIO1 failed");
        return XST_FAILURE;
    }
    // Set all buttons direction to INPUT
    XGpio_SetDataDirection(&READY, 1, 0xFF);

    return XST_SUCCESS;
}

```

```
int init_GPIO2(void){  
    int status;  
  
    // Initialize DATA_IN2 (GPIO2)  
    status = XGpio_Initialize(&LOAD, GPIO2_DEVICE_ID);  
    if(status != XST_SUCCESS) {  
        print("Initialization of GPIO2 failed");  
        return XST_FAILURE;  
    }  
    // Set all buttons direction to output  
    XGpio_SetDataDirection(&LOAD, 1, 0x00);  
  
    return XST_SUCCESS;  
}  
  
int init_GPIO3(void){  
    int status;  
  
    // Initialize RPI_IN (GPIO3)  
    status = XGpio_Initialize(&FLAG_LOAD, GPIO3_DEVICE_ID);  
    if(status != XST_SUCCESS) {  
        print("Initialization of GPIO3 failed");  
        return XST_FAILURE;  
    }  
    // Set all buttons direction to output  
    XGpio_SetDataDirection(&FLAG_LOAD, 1, 0x00);  
  
    return XST_SUCCESS;  
}  
  
int init_GPIO4(void){  
    int status;  
  
    // Initialize Ready (GPIO4)  
    status = XGpio_Initialize(&DATA, GPIO4_DEVICE_ID);  
    if(status != XST_SUCCESS) {  
        print("Initialization of GPIO4 failed");  
        return XST_FAILURE;  
    }  
    // Set all buttons direction to OUTPUT  
    XGpio_SetDataDirection(&DATA, 1, 0x00);  
  
    return XST_SUCCESS;  
}  
  
int init_GPIO5(void){  
    int status;  
  
    // Initialize Ready (GPIO4)  
    status = XGpio_Initialize(&STORE, GPIO5_DEVICE_ID);
```

```

        if(status != XST_SUCCESS) {
            print("Initialization of GPIO5 failed");
            return XST_FAILURE;
        }
        // Set all buttons direction to OUTPUT
        XGpio_SetDataDirection(&STORE, 1, 0xFF);

        return XST_SUCCESS;
    }

int init_GPIO6(void){
    int status;

    // Initialize Ready (GPIO4)
    status = XGpio_Initialize(&FLAG_STORE, GPIO6_DEVICE_ID);
    if(status != XST_SUCCESS) {
        print("Initialization of GPIO6 failed");
        return XST_FAILURE;
    }
    // Set all buttons direction to OUTPUT
    XGpio_SetDataDirection(&FLAG_STORE, 1, 0x00);

    return XST_SUCCESS;
}

int init_GPIO7(void){
    int status;

    // Initialize Ready (GPIO4)
    status = XGpio_Initialize(&RPI_IN, GPIO7_DEVICE_ID);
    if(status != XST_SUCCESS) {
        print("Initialization of GPIO7 failed");
        return XST_FAILURE;
    }
    // Set all buttons direction to INPUT
    XGpio_SetDataDirection(&RPI_IN, 1, 0xFF);

    return XST_SUCCESS;
}

int init_GPIO8(void){
    int status;

    // Initialize Ready (GPIO4)
    status = XGpio_Initialize(&END_TRANSMISSION, GPIO8_DEVICE_ID);
    if(status != XST_SUCCESS) {
        print("Initialization of GPIO8 failed");
        return XST_FAILURE;
    }
    // Set all buttons direction to INPUT
    XGpio_SetDataDirection(&END_TRANSMISSION, 1, 0xFF);
}

```

```

        return XST_SUCCESS;
}

/*****************/
/** Initialization Interrupt controller**/
/*****************/
int init_interrupt(void){
    int statuss;
    // Initialize interrupt controller

    //statuss = IntcInitFunction1(INTC_DEVICE_ID, &START, &READY, &STORE,
    &END_TRANSMISSION, &ENABLE);
    statuss = IntcInitFunction(INTC_DEVICE_ID, &START, &READY, &STORE,
    &END_TRANSMISSION);

    if(statuss != XST_SUCCESS) {
        print("Initialization of interruptions failed");
        return XST_FAILURE;
    }
    return statuss;
}

/*****************/
/** MAIN *****/
/*****************/

int main()
{
    int status_load;
    int status_flag_store;
    int status_flag_load;

    init_platform();

    print("Initialization start\n");

    init_bram();
    init_value();

    init_GPIO0(); //Initialize START
    init_GPIO1(); //Initialize READY
    init_GPIO2(); //Initialize LOAD
    init_GPIO3(); //Initialize FLAG_LOAD
    init_GPIO4(); //Initialize DATA
    init_GPIO5(); //Initialize STORE
    init_GPIO6(); //Initialize FLAG_STORE
    init_GPIO7(); //Initialize RPI_IN
    init_GPIO8(); //Initialize END_TRANSMISSION

    init_interrupt();
}

```

```

//Initialize value for the output GPIO
XGpio_DiscreteWrite(&LOAD, 1,0);
XGpio_DiscreteWrite(&FLAG_LOAD, 1,0);
XGpio_DiscreteWrite(&FLAG_STORE, 1,0);
print("Initialization succeeded\n\r");

while(1){
    //In this while the flags used to execute only once each
    interruption are set to zero
    status_load=XGpio_DiscreteRead(&LOAD, 1);
    status_flag_load=XGpio_DiscreteRead(&FLAG_LOAD, 1);
    status_flag_store=XGpio_DiscreteRead(&FLAG_STORE, 1);

    if(status_load==1){
        XGpio_DiscreteWrite(&LOAD, 1,0);;
        flag_end_transmission=0;
    }

    if(status_flag_load==1){
        XGpio_DiscreteWrite(&FLAG_LOAD, 1,0);
    }

    if(status_flag_store==1){
        XGpio_DiscreteWrite(&FLAG_STORE, 1,0);
        flag_store=0;
    }
}

cleanup_platform();
return 0;
}

/*****************/
***** Interruption *****/
/*****************/

void START_Intr_Handler(void *InstancePtr){
    int word;
    // Disable GPIO interrupts
    XGpio InterruptDisable(&START, Start_INT);
    // Ignore additional button presses
    if ((XGpio InterruptGetStatus(&START) & Start_INT) !=Start_INT)
{return;}

    if(flag_start==0){

        //The input data is loaded in the signal that sends it to the
        channel

        word=XBram_In8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+number_word_input)
        ;
        XGpio_DiscreteWrite(&DATA, 1,word);
        //The signals for the VHDL process are set
        XGpio_DiscreteWrite(&LOAD, 1,1);
    }
}

```

```

XGpio_DiscreteWrite(&FLAG_LOAD, 1,1);
//The input data is printed in the terminal
print("Start interruption\n");
printf("IN_START=0x%04x\n\r", word);

flag_start=1;

number_word_input=number_word_input+1;
}

(void)XGpio_InterruptClear(&START, Start_INT);
// Enable GPIO interrupts
XGpio_InterruptEnable(&START, Start_INT);
}

void READY_Intr_Handler(void *InstancePtr){
    int word;
    // Disable GPIO interrupts
    XGpio_InterruptDisable(&READY, Start_INT);
    // Ignore additional button presses
    if ((XGpio_InterruptGetStatus(&READY) & Start_INT) !=Start_INT)
{return;}

        //The input data is loaded in the signal that sends it to the
channel

word=XBram_In8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+number_word_input)
;

XGpio_DiscreteWrite(&DATA, 1,word);
//The signals for the VHDL process are set
XGpio_DiscreteWrite(&LOAD, 1,1);
XGpio_DiscreteWrite(&FLAG_LOAD, 1,1);
//The input data is printed in the terminal
print("READY interruption ");
printf("IN_READY=0x%04x\n\r", word);

flag_ready=1;

number_word_input=number_word_input+1;

(void)XGpio_InterruptClear(&READY, Start_INT);
// Enable GPIO interrupts
XGpio_InterruptEnable(&READY, Start_INT);
}

void STORE_Intr_Handler(void *InstancePtr){
    int word;
    int word2;
    // Disable GPIO interrupts
    XGpio_InterruptDisable(&STORE, Start_INT);
    // Ignore additional button presses
}

```

```

    if ((XGpio_InterruptGetStatus(&STORE) & Start_INT) !=Start_INT)
{return;}

    if(flag_store==0){
        //It is printed the package number that has arrived
        printf("package=%d\n", package);
        //The data is stored in the corresponding space of the memory
        word=XGpio_DiscreteRead(&RPI_IN, 1);

        XBram_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+number_word_output,wor
d);
        XGpio_DiscreteWrite(&FLAG_STORE, 1,1);
        //The result of the test is printed in the terminal

        word2=XBram_In8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+number_word_outpu
t);
        printf("RPI_IN=0x%04x\n", word2);

        flag_store=1;

        package=package+1;

        number_word_output=number_word_output+1;
    }

    (void)XGpio_InterruptClear(&STORE, Start_INT);
    // Enable GPIO interrupts
    XGpio_InterruptEnable(&STORE, Start_INT);
}

void END_TRANSMISSION_Intr_Handler(void *InstancePtr){
    int word;

    // Disable GPIO interrupts
    XGpio_InterruptDisable(&END_TRANSMISSION, Start_INT);
    // Ignore additional button presses
    if ((XGpio_InterruptGetStatus(&END_TRANSMISSION) & Start_INT)
!=Start_INT) {return;}

    if(flag_end_transmission==0){

        //If there are more inputs available in the memory
        if(number_word_input<12){
            //The input data is loaded in the signal that sends it to
the channel

            word=XBram_In8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+number_word_input)
;
            XGpio_DiscreteWrite(&DATA, 1,word);
            //The signals for the VHDL process are set
            XGpio_DiscreteWrite(&LOAD, 1,1);
            XGpio_DiscreteWrite(&FLAG_LOAD, 1,1);
            //The input data is printed in the terminal
    }
}

```

```

        print("\rEnd interruption \n");
        printf("IN_END=0x%04x\n\r", word);

        flag_end_transmission=1;

        number_word_input=number_word_input+1;
    }

    //If there are no more available inputs in the memory, then we
    must take again the same values as before
    else{
        number_word_input=0;
        number_word_output=12;
        //The input data is loaded in the signal that sends it to
        the channel

        word=XBram_In8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR+number_word_input)
        ;
        XGpio_DiscreteWrite(&DATA, 1,word);
        //The signals for the VHDL process are set
        XGpio_DiscreteWrite(&LOAD, 1,1);
        XGpio_DiscreteWrite(&FLAG_LOAD, 1,1);
        //The input data is printed in the terminal
        print("End interruption \n");
        printf("IN_END=0x%04x\n\r", word);
        flag_end_transmission=1;
        number_word_input=number_word_input+1;
    }

}

(void)XGpio InterruptClear(&END_TRANSMISSION, Start_INT);
// Enable GPIO interrupts
XGpio InterruptEnable(&END_TRANSMISSION, Start_INT);
}

//-----
// INITIAL SETUP FUNCTIONS
//-----


int InterruptSystemSetup(XScuGic *XScuGicInstancePtr){
    // Enable interrupt

    XGpio InterruptEnable(&START, Start_INT);
    XGpio InterruptGlobalEnable(&START);

    XGpio InterruptEnable(&READY, Start_INT);
    XGpio InterruptGlobalEnable(&READY);

    XGpio InterruptEnable(&STORE, Start_INT);
    XGpio InterruptGlobalEnable(&STORE);
}

```

```

XGpio_InterruptEnable(&END_TRANSMISSION, Start_INT);
XGpio_InterruptGlobalEnable(&END_TRANSMISSION);

Xil_ExceptionInit();
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler,XScuGicInstancePtr);
Xil_ExceptionEnable();
return XST_SUCCESS;
}

int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr, XGpio
*XGpioInstancePtr2, XGpio *GpioInstancePtr3, XGpio *GpioInstancePtr4){
    XScuGic_Config *IntcConfig;
    int status;

    // Interrupt controller initialization
    IntcConfig = XScuGic_LookupConfig(DeviceId);
    status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
    if(status != XST_SUCCESS) return XST_FAILURE;

    // Call to interrupt setup
    status = InterruptSystemSetup(&INTCInst);
    if(status != XST_SUCCESS) return XST_FAILURE;

    // Connect GPIO interrupt to handler
    status = XScuGic_Connect(&INTCInst,
    INTC_GPIO_INTERRUPT_ID,(Xil_ExceptionHandler)START_Intr_Handler,(void
    *)GpioInstancePtr);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XScuGic_SetPriorityTriggerType(&INTCInst,INTC_GPIO_INTERRUPT_ID ,
    0xA0, 0x3);

    // Connect GPIO interrupt to handler
    status = XScuGic_Connect(&INTCInst,
    INTC_GPIO_INTERRUPT_ID1,(Xil_ExceptionHandler)READY_Intr_Handler,(void
    *)GpioInstancePtr2);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XScuGic_SetPriorityTriggerType(&INTCInst,INTC_GPIO_INTERRUPT_ID ,
    0xA0, 0x3);

    // Connect GPIO interrupt to handler
    status = XScuGic_Connect(&INTCInst,
    INTC_GPIO_INTERRUPT_ID5,(Xil_ExceptionHandler)STORE_Intr_Handler,(void
    *)GpioInstancePtr3);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XScuGic_SetPriorityTriggerType(&INTCInst,INTC_GPIO_INTERRUPT_ID ,
    0xA0, 0x3);

    // Connect GPIO interrupt to handler
}

```

```
    status = XScuGic_Connect(&INTCInst,
INTC_GPIO_INTERRUPT_ID8,(Xil_ExceptionHandler)END_TRANSMISSION_Intr_Handle
r,(void *)GpioInstancePtr4);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XScuGic_SetPriorityTriggerType(&INTCInst,INTC_GPIO_INTERRUPT_ID ,
0xA0, 0x3);

    // Enable GPIO interrupts interrupt
    XGpio_InterruptEnable(GpioInstancePtr, 1);
    XGpio_InterruptGlobalEnable(GpioInstancePtr);

    // Enable GPIO interrupts interrupt
    XGpio_InterruptEnable(GpioInstancePtr2, 1);
    XGpio_InterruptGlobalEnable(GpioInstancePtr2);

    // Enable GPIO interrupts interrupt
    XGpio_InterruptEnable(GpioInstancePtr3, 1);
    XGpio_InterruptGlobalEnable(GpioInstancePtr3);

    // Enable GPIO interrupts interrupt
    XGpio_InterruptEnable(GpioInstancePtr4, 1);
    XGpio_InterruptGlobalEnable(GpioInstancePtr4);

    // Enable GPIO and timer interrupts in the controller
    XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);
    XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID1);
    XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID5);
    XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID8);

return XST_SUCCESS;}
```

Appendix I.4 DUT board: receiving information module

This module is designed for the DUT. It is in charge of receiving the inputs from the monitor, gathering them together and sending them to the test. The language used is VHDL.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Communication_module_input is
    --This two generics are the ones that control the total number of inputs
    and outputs
    generic(NumberInputs_COM:integer :=8;
            --The max size of package that can be received or sent is 16
            MaxSizePack_COM: integer :=8);

    Port ( clk : in STD_LOGIC;
           Reset_COM: in STD_LOGIC;

           Enable_in_COM: in STD_LOGIC;
           Ready_in_COM: out STD_LOGIC;

           led_in: out STD_LOGIC;
           Enable_shift_COM: out STD_LOGIC;

           IN_UUT_COM: out STD_LOGIC_VECTOR(NumberInputs_COM-1 downto 0);
           IN_COM : in STD_LOGIC_VECTOR(MaxSizePack_COM-1 downto 0)
    );
end Communication_module_input;

architecture Behavioral of Communication_module_input is

    --This two constants allow to calculate the number of packages
    depending on the remain of the division
    constant div_input: integer :=NumberInputs_COM REM MaxSizePack_COM;

    --This functions returns a 0 in case that the packages are all complete
    or 1 if there is a package incomplete
    function rest(div:integer)
        return integer is
    begin
        if div=0 then
            return 0;
        else
            return 1;
        end if;
    end function;

    --These constants are the number of input/output packages that have to
    be sent in total for the number of I/O
    --If there is one package incomplete, then an extra package is added

```

```

constant NumberIPack: integer := (NumberInputs_COM/MaxSizePack_COM)
+1*rest(div_input);

--Auxiliar signals used inside the input and shift register processes
signal aux_in : STD_LOGIC_VECTOR(NumberInputs_COM-1 downto
0):=(others=>'0');
-- signal flag : integer:=0;

begin

    --This process is used for recollecting all the input packages in only
one variable: aux_in
-----
Input_package: process(clk, enable_in_COM, reset_COM)
-----

    --Variable that allows the for loops to locate the packages in their
correct position of aux_in
    variable numpackage: natural range 1 to 100:= 1;
    --This variable is a flag to assure that the enable_shift only reaches
value one when the correct conditions are given
    --Avoids the enable_shift getting one in the beggining or when the out
process finishes, by assuring that in only can be set to one if an input
process has been carried out
    variable input_flag: natural range 0 to 1:=0;
    variable end_input_flag: natural range 0 to 1:=0;
    variable flag: natural range 0 to 1:=0;
    variable flag_ready: natural range 0 to 1:=0;
    variable flag_enable: natural range 0 to 1:=0;

begin

    if(reset_COM='1') then
        IN_UUT_COM<=(others=>'0');
        Enable_shift_com<='0';
        READY_IN_COM<='0';
        led_in<='0';
        end_input_flag:=0;
        input_flag:=0;
        flag:=0;
        numpackage:=1;
        flag_ready:=0;
        flag_enable:=0;

    elsif(clk'event and clk='1') then
        if(enable_in_COM='1' and numpackage=1) then
            led_in<='0';
            end if;
            --When the monitoring system activates the enable_in, then
the input is collected
            --The inputs come in packages of up to 16 signals, the
number of packages is NumberIPack

```

```

--For as many times as packages there are, the inputs are
moved from the input to aux_in, which's size is the total number of inputs
if (enable_in_COM='1' and flag_enable=0) then

    --The flag is set to one to prove that this
process is being carried out
    input_flag:=1;
    flag_enable:=1;

    --If this package is complete, has 8 signals then,
the loop is complete
    if(numpackage< NumberIPack) then

        --READY_IN_COM<='1';
        flag_ready:=1;

        for j in 0 to MaxSizePack_COM-1 loop
            --16*(numcycles-1) is in case that there is
more than one package, to get all the inputs in the same input signal
(aux_in)
            --aux_in(j+MaxSizePack_COM*(numpackage-
1))<=IN_COM(j);
            --IN_UUT_COM(j+MaxSizePack_COM*(numpackage-
1))<=IN_COM(j);
            --IN_UUT_COM(j+MaxSizePack_COM*(numpackage-
1))<='1';
            end loop;
            if(flag=0) then
                numpackage:=numpackage+1;

                flag:=1;
            end if;

        --Here we treat the two left cases
        --First: The last package of a transmission that
is complete
        --Second: If the package size is smaller than 8,
then the loop has to go from zero to the remainder
        else
            end_input_flag:=1;

        if(numpackage = NumberIPack and div_input=0)
then

            for j in 0 to MaxSizePack_COM-1 loop
                --16*(numcycles-1) is in case that there is
more than one package, to get all the inputs in the same input signal
(aux_in)
                --
aux_in(j+MaxSizePack_COM*(numpackage-1))<=IN_COM(j);

                IN_UUT_COM(j+MaxSizePack_COM*(numpackage-1))<=IN_COM(j);

```

```

--  

IN_UUT_COM(j+MaxSizePack_COM*(numpackage-1))<='1';  

--IN_UUT_COM(j+MaxSizePack_COM)<='1';  

    end loop;  

else  

    for i in 0 to ((NumberInputs_COM REM  

MaxSizePack_COM)-1) loop  

--  

aux_in(i+MaxSizePack_COM*(numpackage-1))<=IN_COM(i);  

--  

IN_UUT_COM(i+MaxSizePack_COM*(numpackage-1))<=IN_COM(i);  

--  

IN_UUT_COM(i+MaxSizePack_COM*(numpackage-1))<='1';  

--IN_UUT_COM(i+MaxSizePack_COM)<='1';  

    end loop;  

    end if;  

end if;  

  

--When a communication has ended, then the process of the  

shift register is allowed to start  

elsif(flag_ready=1 and enable_in_COM='0') then  

    READY_IN_COM<='1';  

    flag_ready:=0;  

    flag_enable:=0;  

  

--If this communication has not ended, then the ready is reset  

and the system waits for the next enable signal  

elsif(enable_in_COM='0' and input_flag=1 and  

end_input_flag=0) then  

    READY_IN_COM<='0';  

    flag:=0;  

  

--If the last package has been processed, then the process must  

end  

elsif (enable_in_COM='0' and input_flag=1 and  

end_input_flag=1) then  

    led_in<='1';  

    enable_shift_COM<='1';  

    READY_IN_COM<='0';  

    --Here we assure, that we cannot enter this if again  

if an input communication has not happened again  

    end_input_flag:=0;  

    input_flag:=0;  

    numpackage:=1;  

    flag_enable:=0;  

  

else  

    enable_shift_COM<='0';  

end if;  

  

end if;  

end process Input_package;  

  

end Behavioral;

```

Appendix I.5 DUT board: sending information module

This module is designed for the DUT. It is in charge of sending the results from the test to the monitor, diving them in packages. The language used is VHDL.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Communication_module_output is
--This two generics are the ones that control the total numer of inputs
and outputs
generic(
    NumberOutputs_COM:integer :=8;
    --The max size of package that can be received or sent is 16
    MaxSizePack_COM: integer :=8);

Port ( clk : in STD_LOGIC;
       Reset_COM: in STD_LOGIC;

       Enable_out_COM: in STD_LOGIC; --Starts process of sending out

       OUT_COM : out STD_LOGIC_VECTOR(MaxSizePack_COM-1 downto 0);
       Ready_Mon: in STD_LOGIC;--Output of the module

       Data_available: out STD_LOGIC;

       finish:out std_logic;
       led_out: out std_logic;
       OUT_UUT_COM: in STD_LOGIC_VECTOR(NumberOutputs_COM-1 downto 0)
--Comes from UUT
       );
end Communication_module_output;

architecture Behavioral of Communication_module_output is

constant div_output: integer :=NumberOutputs_COM REM MaxSizePack_COM;

function rest(div:integer)
    return integer is
begin
    if div=0 then
        return 0;
    else
        return 1;
    end if;
end function;

constant Number0Pack: integer := (NumberOutputs_COM/MaxSizePack_COM)
+1*rest(div_output);

signal flag_end_out: STD_LOGIC :='0';

signal en_out_aux_com: STD_LOGIC :='0';

```

```

begin
    --This process is used for splitting the variable aux_out in as
many output packages as necessary
    -----
    Output_package: process(clk, enable_out_COM, Reset_COM)
    -----
        --This variable counts the package that is being transmitted to the
output
        variable numpackage: natural range 1 to 100:= 1;
        variable flag_data_available: natural range 0 to 1:=0;
        variable flag_pack: natural range 0 to 1:=0;

begin

if (reset_COM='1') then
    OUT_COM<=(others=>'0');
    numpackage:=1;
    flag_end_out<='0';
    en_out_aux_com<='0';
    finish<='0';
    flag_data_available:=0;
    led_out<='0';
    flag_pack:=0;

elsif(CLK'event and CLK='1') then

    if(enable_out_COM='1') then
        numpackage:=1;
        finish<='0';
        led_out<='0';
    end if;

    if((enable_out_COM='1' or Ready_mon='1') and
flag_data_available=0 ) then

        --This 'if' checks whether or not there are more packages
to send to the output
        if (numpackage/= NumberOPack+1) then
            flag_data_available:=1;
            --This signal allows to extend the duration of the
output enable
            led_out<='1';
            --If all the packages of a transmission are
complete or if it is not the last one of an incomplete transmission, then
we copy the info to the output
            --div_output contains the size of the last package,
in case it is incomplete and allows to take care of the last package
depending on its size
            if(div_output=0 OR (div_output/=0 and
numpackage<NumberOPack)) then

                for j in 0 to MaxSizePack_COM-1 loop

```

```

Out_COM(j)<=OUT_UUT_COM(j+MaxSizePack_COM*(numpackage-1));
      end loop;

numpackage:=numpackage+1;

      --If it is the last package of an incomplete
      transmission, then the rest of the info is copied, and the package is
      completed with '-'
      elsif (div_output/=0 and numpackage=NumberOPack)
then
      flag_data_available:=1;
      for i in 0 to ((NumberOutputs_COM REM
MaxSizePack_COM)-1) loop

Out_COM(i)<=OUT_UUT_COM(i+MaxSizePack_COM*(numpackage-1));
      end loop;
      for j in (NumberOutputs_COM REM
MaxSizePack_COM) to MaxSizePack_COM-1 loop
          Out_COM(j)<='0';
      end loop;
      numpackage:=numpackage+1;

end if;

--Once all the info has been sent, the flag that warns this
is set to one
      elsif(numpackage= NumberOPack+1) then
          flag_data_available:=1;
          flag_end_out<='1';
          numpackage:=numpackage+1;
      end if;

      elsif (flag_data_available=1 and Ready_mon='0') then
          data_available<='1';
          flag_data_available:=0;

--Once the info has been sent, the output is set to '-' and the
flag is set to zero to start another round of communication
      elsif(flag_end_out='1') then
          finish<='1';
          flag_end_out<='0';
          data_available<='0';
          flag_data_available:=0;

      elsif(flag_data_available=0) then
          data_available<='0';
      end if;

end if;

end process Output_package;
end Behavioral;

```

Appendix I.6 Unit Under Test module: Shift register

This is the test circuit used to verify the correct functioning of the system, and to measure the its performance. The language used is VHDL.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UUT_ShiftRegister is
    --This two generics are the ones that control the total numer of
    inputs and outputs
    generic(NumberInputs_UUT:integer:=8;
            NumberOutputs_UUT:integer :=8
            );
    Port ( clk : in STD_LOGIC;
           Reset_UUT: in STD_LOGIC;
           Enable_out_UUT: out STD_LOGIC;
           Enable_shift_UUT: in STD_LOGIC;
           led_uut: out STD_Logic;
           IN_UUT: in STD_LOGIC_VECTOR(NumberInputs_UUT-1 downto 0);
           OUT_UUT: out STD_LOGIC_VECTOR(NumberOutputs_UUT-1 downto 0)
           );
end UUT_ShiftRegister;

architecture Behavioral of UUT_ShiftRegister is

begin
    --This process is the one that shifts the input to get the output, and
    stores it in aux_out
    -----
    Shift_register: process(clk, Reset_UUT)
    -----
    --This variable is used as flag to enable the output process.
    --When the shifting process has finished, then the output one can
    start
        --It also prevents the output process to start in any other ocassion
        than when the shift process has taken place
    variable flag_end_shift: natural range 0 to 1:=0;
    variable flag_shift: natural range 0 to 1:=0;

    begin
        if(Reset_UUT='1')then
            OUT_UUT<=(others=>'0');
            Enable_out_UUT<='0';
            aux_out<=(others=>'0');

```

```

led_uut<='0';
flag_shift:=0;
flag_end_shift:=0;

elsif (CLK'event and clk='1') then
    if(enable_shift_UUT='1') then
        led_uut<='0';
    end if;

    --If the input communication has finished and has enable the shift
process, then this starts
    --Depending on the number of total inputs and outputs, the process
carried out is different
    if(enable_shift_UUT='1' and flag_end_shift=0) then

        --When there are more outputs than inputs, then all the input
is shifter and stores in the aux_out
        --The rest of the variable is set to zero
        if(NumberOutputs_UUT>NumberInputs_UUT) then
            for i in 0 to (NumberInputs_UUT-1) loop
                OUT_UUT(i+1) <= IN_UUT(i);
                --aux_out(i+1) <= aux_in(i);
            end loop;
            OUT_UUT(0) <= '0'; --The last bit is set to zero
            flag_end_shift:=1;

            if(flag_shift=0) then
                flag_shift:=1;
            else
                flag_shift:=0;
            end if;

        --When there are more inputs than outputs, then the loop has
to be limited by the number of the outputs
        else

            for i in 0 to (NumberOutputs_UUT-2) loop
                OUT_UUT(i+1) <= IN_UUT(i);
            end loop;
            OUT_UUT(0) <= '0'; --The last bit is set to zero
            flag_end_shift:=1;

            if(flag_shift=0) then
                flag_shift:=1;
            else
                flag_shift:=0;
            end if;
        end if;

    elsif(flag_end_shift=1)then
        enable_out_UUT<='1';
        led_uut<='1';
        flag_end_shift:=0;

```

```
    else
        enable_out_UUT<='0';

    end if;

end if;

end process Shift_register;

end Behavioral;
```