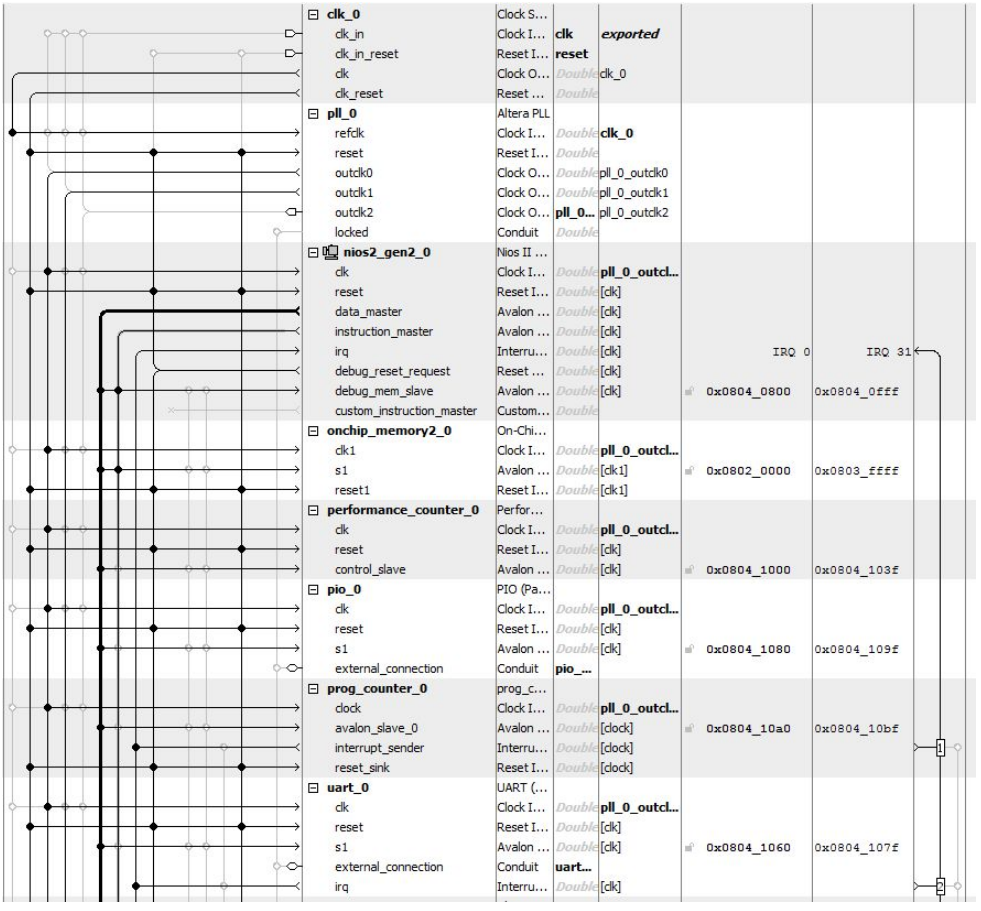
Real-Time Embedded Systems
Report on laboratory 3
« Multiprocessor design »

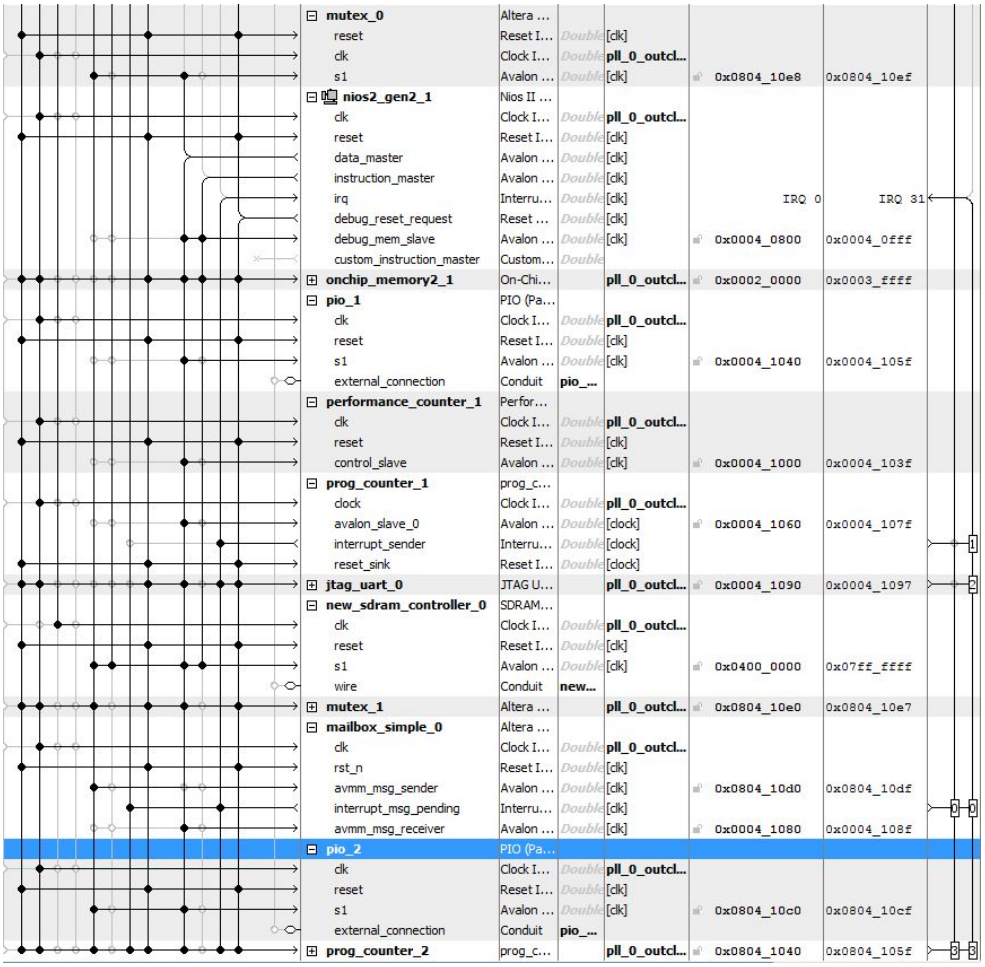
I. Introduction

In this report we present our software and hardware design of a multiprocessor system using 2 NIOS-II processors on DE1-NanoSoc FPGA. We also present the methodology we followed to realize the different measurements, and finally we present and discuss the obtained results.

II. Hardware design

The hardware system is quite complex for this laboratory since there are 2 processors, the basic components that are required need to be duplicated, and their access limited to the corresponding processor. On top of that, some components may be accessed by both the processors, such as a bidirectional parallel port and a hardware mutex per shared peripheral to arbitrate the accesses to the peripheral. For the second part of the laboratory, we use a hardware mailbox to pass messages from one processor to the other. The figure below shows the contents and the interconnects of the hardware system.





The mutex_0 is used for the bidirectional parallel port, and mutex_1 is used for the shared SDRAM memory. To implement that, both mutexes are connected to the global reset network and to the 2 data_master ports of the processors. The mailbox has been connected in such a way that processor_0 is the message sender and processor_1, the receiver. Processor_0 is connected to a serial UART interface and processor_1 is connected via the usual JTAG-UART interface, that we have been using for the other laboratories.

III. Software code, measurements & results

a. Parallel port test

In this part, no shared peripheral is accessed, so no need to use mutexes to access the peripherals. The code is the same on both processors except for the names of the peripherals and their base addresses. What is done in this code is, for each processor, the initialization of the specific counter (our custom IP component which we used for Lab 1 as well) to make it start counting at 50 MHz and after a fixed amount of cycles (2'500'000 cycles at 50 MHz which corresponds to 50 ms), the unidirectional parallel port in output-mode is accessed (same for both processors). Both the parallel ports seemed to work correctly (we mapped the parallel ports to 2 different 7-Segment units).

b. Hardware mutex

At first the mutex must be opened, to make the mutex core accessible to the other functions, then the mutex is acquired by calling the following function: "altera_avalon_mutex_lock" which

makes the current processor the owner of the mutex. Then the access to the shared peripheral can be made because the program counter will not move until the lock is acquired, so it is a safe access. Finally, when all the accesses to the shared resource have been made, the mutex should be unlocked to make it available to the other processors.

How does the mutex core work? Basically, the mutex is an Avalon Memory-Mapped slave with 2 32-bit registers. We included the mutex core register map provided in the altera documentation. The first register contains 16 bits for the Owner and 16 bits for the value. The Owner value will be filled with the ID of the processor that first acquires the lock. The VALUE bits equal 0 if unlocked and 1 if locked by the owner. The VALUE bits can only be accessed for a write when the VALUE is 0 OR the processor ID corresponds to the ID stored

Table 27–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 16	15 1	0
0	mutex	RW	OWNER	VALUE	
1	reset	RW	Reserved		RESET

in the OWNER register.

We included screenshots of the software part. As explained before, the mutex is first opened and locked by the corresponding processor. Then, the common parallel port is initialized at max value because the decrementing is done twice as fast as the incrementing, and the mutex is unlocked. After that, the specific parallel port (for the 7-Segment display) is initialized and the specific counter is started. The empty for loop was used as a delay buffer in case the counter was not working, we had problems getting the counters work on both processors (it worked well for processor 0 but not for processor 1 even when starting them up separately). Then the operation of incrementing the value/decrementing depending on the processor ID is done for thousands of cycles in order to have an averaged value of the measurement. The overhead of locking and unlocking the mutex are measured by using the Altera Performance counter. It can be done simultaneously by using the different section counters available (up to 3 in our Qsys settings). We obtained the following results:

203 cycles for a lock, 182 cycles for an unlock. These values are taken by averaging the results over 2000 iterations of the same portion of code to have more reliable results. Locking the mutex demands more cycles on average which seems to make sense since it waits for it to be unlocked before accessing it, while unlocking it only requires to check the processor ID. Still both these 2 operations create significant overhead in terms of execution time.

```c
#define START_COUNTER 8
#define STOP_COUNTER 12
#define IRQ_ENABLE_COUNTER 16 //bit 0
#define CLR_EOT_COUNTER 20 //bit 0
#define SET_COMP_COUNTER 28 //set a compare value
#define MS50 1000000

#define READ_VAL_COUNTER 0 //bit 0
#define READ_IRQ_COUNTER 16 //bit 0
#define READ_EOT_COUNTER 20 //bit 0
#define READ_COMP_COUNTER 28 //read compare value


//address offsets for the custom parallel port
#define DIR_OFFSET 0
#define WRITE_OFFSET 4
#define SET_OFFSET 12
#define CLEAR_OFFSET 16

int value_showhex(int value)
{
    static unsigned char _hex_digits_data[] = {0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x90};
    int led_value;
    led_value = _hex_digits_data[value & 0xF];
    //printf("led value %d\n",led_value);
    return led_value;
}

int main()
{
    alt_mutex_dev* mutex=altera_avalon_mutex_open(MUTEX_0_NAME);

    altera_avalon_mutex_lock(mutex,1);
    IOWR_32DIRECT(PIO_2_BASE,4,0); //set direction 0 input 1 out
    IOWR_32DIRECT(PIO_2_BASE,0,0b11111111); //decrement value
    altera_avalon_mutex_unlock(mutex);

    printf("Hello from Nios II %d \n",ALT_CPU_CPU_ID_VALUE);
    int i=0;
    int read;
    IOWR_32DIRECT(PIO_0_BASE,4,1); //output mode
    IOWR_32DIRECT(PIO_0_BASE,4,1); //set direction 0 input 1 out

    IOWR_32DIRECT(PIO_0_BASE,0,value_showhex(i++));
```

```c
    IOWR_32DIRECT(PROG_COUNTER_0_BASE, RESET_COUNTER, 0);
    IOWR_32DIRECT(PROG_COUNTER_0_BASE, SET_COMP_COUNTER, MS50);
    IOWR_32DIRECT(PROG_COUNTER_0_BASE, CLR_EOT_COUNTER, 1);
    IOWR_32DIRECT(PROG_COUNTER_0_BASE, START_COUNTER, 0);
    int j;
    for(j=0;j<315000;j++)
    {

    }
    printf("counter_start %d\n",IORD_32DIRECT(PROG_COUNTER_0_BASE, START_COUNTER));
    PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
    PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);

    while(i<=5000){
        if(IORD_32DIRECT(PROG_COUNTER_0_BASE, READ_EOT_COUNTER)==3){

            IOWR_32DIRECT(PIO_0_BASE,0,value_showhex(i%10));
            IOWR_32DIRECT(PROG_COUNTER_0_BASE, RESET_COUNTER, 0);
            IOWR_32DIRECT(PROG_COUNTER_0_BASE, CLR_EOT_COUNTER, 1);
            IOWR_32DIRECT(PROG_COUNTER_0_BASE, START_COUNTER, 0);

            PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,1);
            altera_avalon_mutex_lock(mutex,1);
            PERF_END(PERFORMANCE_COUNTER_0_BASE,1);

            IOWR_32DIRECT(PIO_2_BASE,4,0); //set direction 0 input 1 out
            read=IORD_32DIRECT(PIO_2_BASE,0);
            IOWR_32DIRECT(PIO_2_BASE,4,1); //set direction 0 input 1 out
            IOWR_32DIRECT(PIO_2_BASE,0,read++); //decrement value

            PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,2);
            altera_avalon_mutex_unlock(mutex);
            PERF_END(PERFORMANCE_COUNTER_0_BASE,2);
            i++;

        }
    }
    PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
    perf_get_section_time(PERFORMANCE_COUNTER_0_BASE,1);
    perf_get_section_time(PERFORMANCE_COUNTER_0_BASE,2);
    perf_print_formatted_report(PERFORMANCE_COUNTER_0_BASE,ALT_CPU_CPU_FREQ,2,1,2);// environ 100/105 clock cycle

  return 0;

}
```

c. Hardware Mailbox

The objective for this task was to send a message to print from one processor to the other via a hardware mailbox, and the receiving processor should be waiting on a queue, which is filled with the message sent from the other processor. The mailbox is unidirectional, meaning

Table 82. Mailbox Register Map

Word Address Offset	Register/ Queue Name	Width	Attribute	
			Sender	Receiver
0x0	Command register	32-bit	R/W	RO
0x1	Pointer register	32-bit	R/W	RO
0x2	Status register	32-bit	RO	RO
0x3	Interrupt Masking register	32-bit	R	R

it is connected to one processor as sender and the other as a receiver in our Qsys system. It is also connected to a shared memory space, on an SDRAM controller. First, we briefly present how the mailbox register map and how it is working.

One can use this mailbox in two different ways. The first one is to write directly into the registers of the IP core, and the second one is to use the software functions of the HAL library in the "altera_avalon_mailbox_simple.h" header file. The steps that should be done when writing directly to the registers, are the following:

A processor that intends to send a message will write to the Mailbox's Pointer register at address offset 0x1, then only to the Command register at address offset 0x0 (the value that is written in the command register is not important). Writing to the Command register indicates the completion of a message passing into the Mailbox. Then if using polling to retrieve the message on the receiver processor, it must poll the Status register at address offset 0x2 to determine if any message has arrived. Interrupts could also be used instead of polling. After polling the status register, if a message is pending to be retrieved, the value of the Status register will be 0x01 and the processor that needs to receive the message reads the Mailbox's Pointer register and then the Command register through the connected Avalon-MM interface. Upon reading of Command register, the message is considered delivered, and the Mailbox is empty.

We implemented this method but observed that the registers at the sender side were not copied to the receiver side (the addresses to which the processors read/write from the mailbox are different: one for the sender, another for the receiver, which means the values in the registers at the sender side are not necessarily equal to the values of the registers at the receiver side). We didn't manage to find out the cause of this issue, so we tried implementing the second method, which did work, and we attached the code of the 2 processors below.

1) Sending processor (nios_1 on JTAG UART, bare metal)

```
#include <stdio.h>
#include "system.h"
#include "io.h"

#include <altera_avalon_mutex.h>
#include "altera_avalon_performance_counter.h"
#include "altera_avalon_mailbox_simple.h"
#include "altera_avalon_mailbox_simple_regs.h"

void tx_cb (void* report, int status) {
    if (!status) {
       printf("Transfer done");
    } else {
       printf ("error in transfer");
    }
}
int main(){

    printf("Hello from Nios II!\n");

    alt_u32 message= 1;
    int timeout     = 50000;
    alt_u32 status;
    altera_avalon_mailbox_dev* mailbox_sender;
    mailbox_sender = altera_avalon_mailbox_open("/dev/mailbox_simple_0", tx_cb, NULL);
    if(!mailbox_sender){
        printf ("FAIL: Unable to open mailbox_simple");
        return 1;
    }

    timeout = 0;
    status = altera_avalon_mailbox_send(mailbox_sender, message,timeout, POLL);
    if (status)
        printf ("error in transfer");
    else
        printf ("Transfer done");
    altera_avalon_mailbox_close(mailbox_sender);

    return 0;
}
```

On the sender side, the sending device is set as *mailbox_sender,* then the mailbox is opened with a specified callback. If this operation failed, an error message is printed in the console. Then the message is sent to the mailbox with the corresponding function, and the status register is read. If the status corresponds to empty mailbox, an error message is printed. Finally, the mailbox is closed on the sender side.

2) Receiving processor (nios_0 on UART, uC-OSII)

```
#define TASK_STACKSIZE 2048
OS_STK task_stk[TASK_STACKSIZE];
OS_STK task1_stk[TASK_STACKSIZE];

#define QUEUE_SIZE 10
OS_EVENT *queue_res;

typedef struct {
    int message;
} msg;

msg* msg_queue[10];
msg *some_msg;
alt_u32 *message;
unsigned int start, stop;


void rx_cb (void* message) {
    /* Get message read from mailbox */
    //alt_u32* data, message;
    alt_u32*data=message;
    if (message!= NULL) {
    printf ("Message received");
    }
else
  printf ("Incomplete receive");

}
void task_mailbox_polling(void* pdata)
{
    alt_u32* message;
    int timeout     = 50000;
    altera_avalon_mailbox_dev* mailbox_rcv;
    /* This example is running on receiver processor */
    mailbox_rcv = altera_avalon_mailbox_open("/dev/mailbox_simple_0", NULL,rx_cb);
    if (!mailbox_rcv){
        printf ("FAIL: Unable to open mailbox_simple");

    }
    /* For interrupt disable system */

    altera_avalon_mailbox_retrieve_poll(mailbox_rcv,message, timeout);
    if (message == NULL)
        printf ("Receive Error");
    else{
        printf ("Message received with Command 0x %x \n", message);
        some_msg->message=message;
        OSQPost(queue_res, some_msg);
    }
    altera_avalon_mailbox_close (mailbox_rcv);
}
```

```
void task_queue(void* pdata){
    msg* bla;
    INT8U err;
    printf (" task queue start \n");

    while(1){
        bla= (struct msg*)OSQPend(queue_res,0,&err);

        if(err==OS_NO_ERR)
            printf("getting message from queue and mailbox: %u \n",bla->message);
        else
            printf("Not getting message");
    }
}
int main()
{
    INT8U err;
    queue_res = OSQCreate(msg_queue, QUEUE_SIZE);
    OSTaskCreateExt(task_queue,
                    NULL,
                    (void *)&task1_stk[TASK_STACKSIZE-1],
                    1,1,
                    task1_stk,
                    TASK_STACKSIZE,
                    NULL,
                    0);
    OSTaskCreateExt(task_mailbox_polling,
                    NULL,
                    (void *)&task_stk[TASK_STACKSIZE-1],
                    2,2,
                    task_stk,
                    TASK_STACKSIZE,
                    NULL,
                    0);

    OSStart();
    return 0;

}
```

In the *main()* function, 2 tasks are created: one for polling the mailbox status, retrieving the message and posting it to a message Queue, and another task pending for a message in this queue.

In the mailbox polling task, the steps are similar the ones taken at the sender side, except now we want to retrieve a message. Thus, the receiving device is set, the mailbox opened, (if not successful → error message), then the status is polled until a certain timeout, after which the received message is printed (if successfully retrieved). Finally, the message is sent into the Queue which the second task is pending for, using *OSPost()* and *queue_res OS_EVENT*, and the mailbox is closed on the receiver side.

The second task basically pends indefinitely for a message in the queue via the *OSPend()* function and prints the message if received correctly. Otherwise prints “not received” and the loop restarts again.

d. Hardware counter

In this part, the hardware design is very similar to the part described in b., where the shared peripheral was a bidirectional parallel port. Now, the shared peripheral is a specific counter. The goal of this experience is to verify that the counter can be successfully accessed from both processors with a hardware mutex and to observe whether removing the mutex results in conflictual accesses or if the counter still behaves as initially wanted. To do that, we

have chosen that the processor 1 starts the counter and verifies it is counting, and the processor 0 stops it and checks it is indeed disabled. These actions will be done repeatedly and at the same frequency. What we observed is that with a mutex, the value of the counter on processor 0 is constant between to calls inside the same mutex window but varies between two different mutex locks, which means that the processor 1 indeed started the counter in-between 2 consecutive locks of processor 0: the counter behaves as expected and both processor manage to have access to the counter. After removing the mutex, the behavior is not the same. The counter value is constant always on processor 0 UART terminal, which means the processor 1 is never starting the counter up because both processors are writing a conflictual value in the enable register. Thus, the use of a mutex is necessary for a correct behavior of the system.

IV. Conclusions

In this laboratory we learned how to use the hardware mutex and mailbox cores, which are the necessary tools to build multiprocessor systems. Mutexes allow multiple masters or processors to access the same peripheral and ensures unique access at a given time and avoids data corruption, when the mailbox allows message passing from one processor to the other, and combining the use of two of these cores, message passing can be done in a bilateral fashion which of course can be very useful in multiple applications. The use of an embedded-system-type operating system like uC-OSII to realize multitasking on a single processor has also proven to be a very useful tool. We had an overview of the different multiprocessing techniques, and have a better idea of how to implement them on embedded systems.