

Computer Architecture - Lab Assignment 4

Nios II multiprocessors implementation, parallel programming and performance evaluation

Introduction

The main objective of this laboratory assignment consists on the performance evaluation of a Nios II dual-core multiprocessor and the comparison with a single Nios II processor. For the programming of the multiprocessor, two programs that are synchronized through a mutual exclusion hardware mechanism called semaphore will be used. These programs will run on a DE0-Nano board.

This assignment contains three parts that are summarized below. For the preparation of this guide, the bibliographical references [1, 5, 2, 4, 3, 6, 8, 7] have been used.

Part 1. Intel-Altera Software Building Tools and Command Shell will be introduced for C language program development.

Part 2. Two C programming tutorials will be developed using the tools mentioned in Part 1. These tutorials will be executed on the DE0-Nano board found in one of the ULPGC laboratories (see Figure 1). To do this, each student will connect the board to a computer in the laboratory and will run a virtual machine called "Altera" where all needed software tools to interact with the mentioned board are located. The first tutorial runs on a single processor and the second runs on a dual-core multiprocessors in parallel.

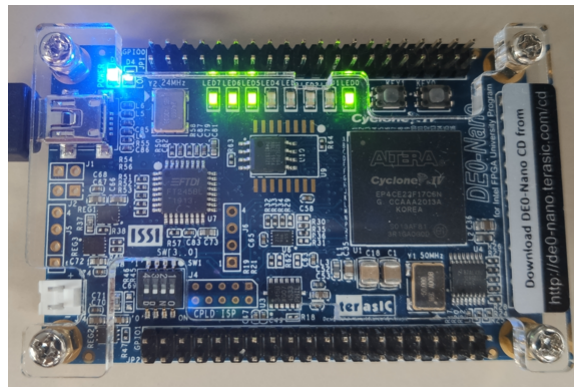


Figure 1: DE0-Nano board

Part 3. Another tutorial consisting on the implementation of the Matrix \times Vector algorithm will be developed. Additionally, the performance of this program will be analyzed using two dual-core multiprocessors based on two members of the Nios II family of processors: $2 \times$ Nios II/e, $2 \times$ Nios II/s. Finally, an exercise is proposed to evaluate the performance of a matrix multiplication algorithm.

The computer material that accompanies this practice can be found in the following folders (they are found in the folder called `code`):

- Tutorial1
- Tutorial2

- Tutorial3
- DualCoreNios2e
- DualCoreNios2s

Part 1. Software-hardware infrastructure of the lab assignment

General description: This part describes the software tools that will be used to develop the C programs. Additionally, the microarchitecture of the Nios II multiprocessors will also be described. The multiprocessors will be configured in the FPGA device of the DEO-Nano board, in which the C programs will run.

Software Build Tools (SBT) for Eclipse: developing the C programs

The SBT tool integrates an user interface that automatize the compiling of C programs for the Nios II instruction set architecture (see Figure 2). SBT includes text editor, program debugger and FPGA device programmer.

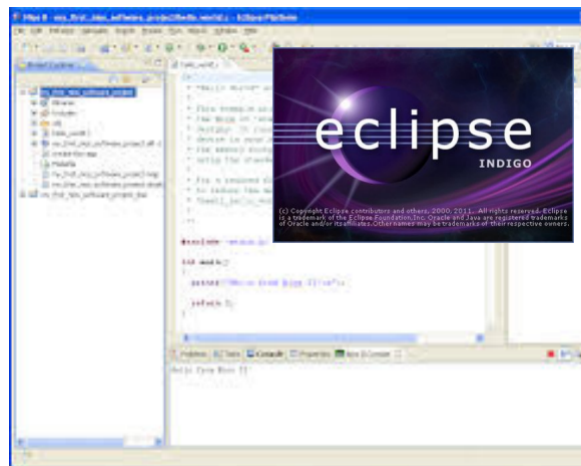


Figure 2: SBT user interface for Eclipse

Nios II Command Shell: board setup, downloading and running the programs

With the Command Shell software tool you can create, modify, compile and run programs for Nios II processors (see Figure 3). To do this, the commands are written through the keyboard in the Command Shell window or inserted into a script file. The commands that will be used in this practice are the following:

Command Shell command: Use a `sof` file to configure the FPGA on the DE0-Nano board

```
$ nios2-configure <file>.sof
```

Command Shell command: Download a compiled `elf` program in the main memory that is located in the DE0-Nano board

```
$ nios2-download -r -g -i <core number> <file>.elf
```

Command Shell command: Displays on the screen the results of `printf` messages that are coded in the C programs

```
$ nios2-terminal
```

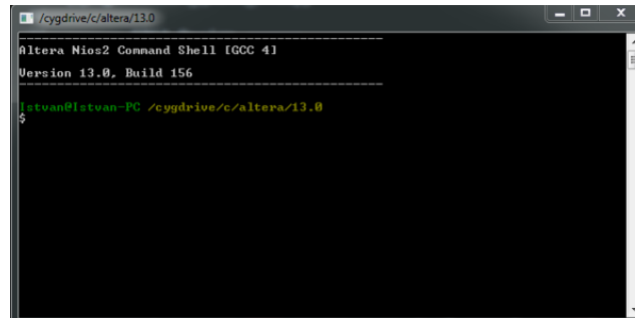


Figure 3: Command Shell window with prompt ready to enter commands that allow to interact with the DE0-Nano board

Nios II multiprocessor with shared memory parallel architecture

When the FPGA circuit of the DE0-Nano board is configured in this lab assignment, a multiprocessor that is integrated by two Nios II cores called CPU and CPU2 is physically implemented. Additionally, the FPGA also integrates the following hardware modules:

- A semaphore device for 8-byte atomic operations (MUTEX). It ensures that a given device is accessed only by one of the processors at a given time. For this reason, it is also called *mutual exclusion device*.
- A 1 KiB SRAM memory to storage variables of the semaphore (BUFFER).

One feature of this multiprocessor is that both processor cores share the same physical RAM memory devices. For this reason, the multiprocessor architecture is called *shared memory*. Additionally, the multiprocessor can be used to execute the same program in parallel. That is why the architecture is also called *parallel*. The MUTEX device is used to synchronize the execution of the various threads into which a parallel program can be divided.

Other element of the DE0-Nano board that is necessary for this lab, is a 32 MiB SDRAM memory. In the Figure 4 a diagram of the Nios II multiprocessor micro-architecture is shown.

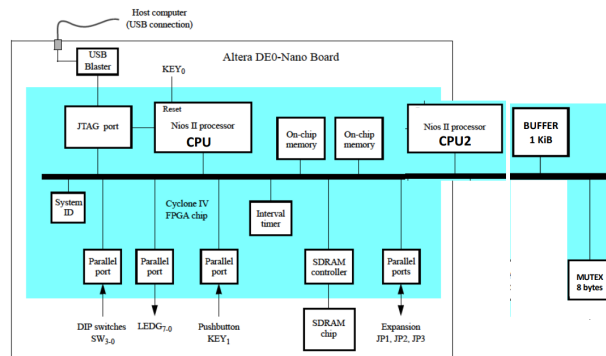


Figure 4: Nios II dual-core multiprocessor with shared memory parallel architecture. In this lab, two FPGA configurations that have a Nios II multiprocessor, called *DualCoreNios2e* and *DualCoreNios2s*, are handled.

Two different Nios II multiprocessor configurations that differ in the processor will be handled. One of them, called *DualCoreNios2e*, integrates two copies of the Nios II/e core and the other configuration, called *DualCoreNios2s*, integrates two copies of the Nios II/s core. The Nios II/s processor includes 1 KiB instruction cache and 2 KiB data cache, both with 32 byte blocks.

Main memory organization

When creating a multiprocessor system, it would be nice to run the software for multiple processors using the same physical memory device. The software for each processor is located in a memory region that is not shared by any other processor but resides on the same physical device.

The SBT tool provides memory partitioning that allows multiple processors to run their software from different regions of the same physical memory device. Additionally, SBT ensures that the software of the processors is linked, determining the correct memory addresses where the different parts in which this software is divided reside. To do this, SBT uses the exception address to calculate the memory area allocated for each processor.

Each processor has five linking zones (see Figure 5):

- **.text** – memory region where the executable code resides
- **.rodata** – memory region where any data that is read-only and used in code execution is located
- **.rwdata** – memory region where read-write variables and memory pointers are located
- **.heap** – memory region where dynamic memory resides
- **.stack** – memory region where the parameters that are handled in calls to functions and subroutines as well as other temporary data are located

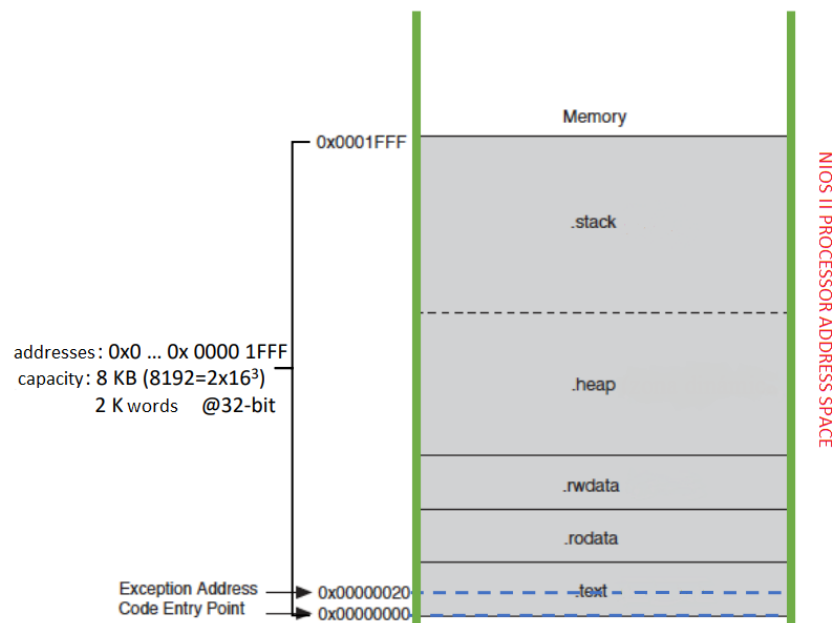


Figure 5: Example of main memory link zones for 1 core.

The SBT tool ensures that these memory sections are located (linked) at fixed memory addresses.

In Figure 5 you can see a memory map that shows how these link sections are assigned to fixed addresses for a single processor. In this figure it has been assumed that the processor software requires eight kilobytes (8 KiB) of memory. Therefore, the processor uses the region between addresses 0x0 and 0x1FFF. Figure 5 also shows the partitioning of the processor's main memory into several linking zones: code (**.text**), data (**.rodata**, **.rwdata**), stack (**.stack**) and dynamic zone (**.heap**). The exception address 0x0000 0020 is included in the code area.

In a multiprocessor system, it would be advantageous to use a single memory device to store all sections of code for each processor core. In our case, each processor's exception address is used to define the boundaries between the end of one processor's code sections and the beginning of the next processor core.

For example, in the case of this practice, the SDRAM main memory takes 32 MiB, between addresses 0x0000 0000 and 0x01FF FFFF. Within this range, the addresses assigned to each of the two Nios II cores, CPU and CPU2, which are arranged in the FPGA configurations mentioned above: DualCoreNios2e and DualCoreNios2s, are established. In Figure 6 it can be seen that the address ranges assigned to the processor cores are:

- Each core is allocated 4 MiB of main memory to run its software. The SBT tool automatically partitions main memory by looking at the exception address. For both cores of the multiprocessor, this address has the values 0x0000 0020 for CPU and 0x0040 0020 for CPU2. See Figure 6 where the address assignment for the two cores is shown.

The six least significant bits of the exception address are always 0x20. At the address whose six least significant bits are 0x00 is called *entry point*, where the Nios II processor expects to find the reset code. The address offset 0x20 is because of the instruction cache line is 32 bytes long (eight 32-bit instructions). These 32 bytes coincides with the extension of the reset code, which ends up skipping the exception handling code area to later start executing the main code of a certain processor core.

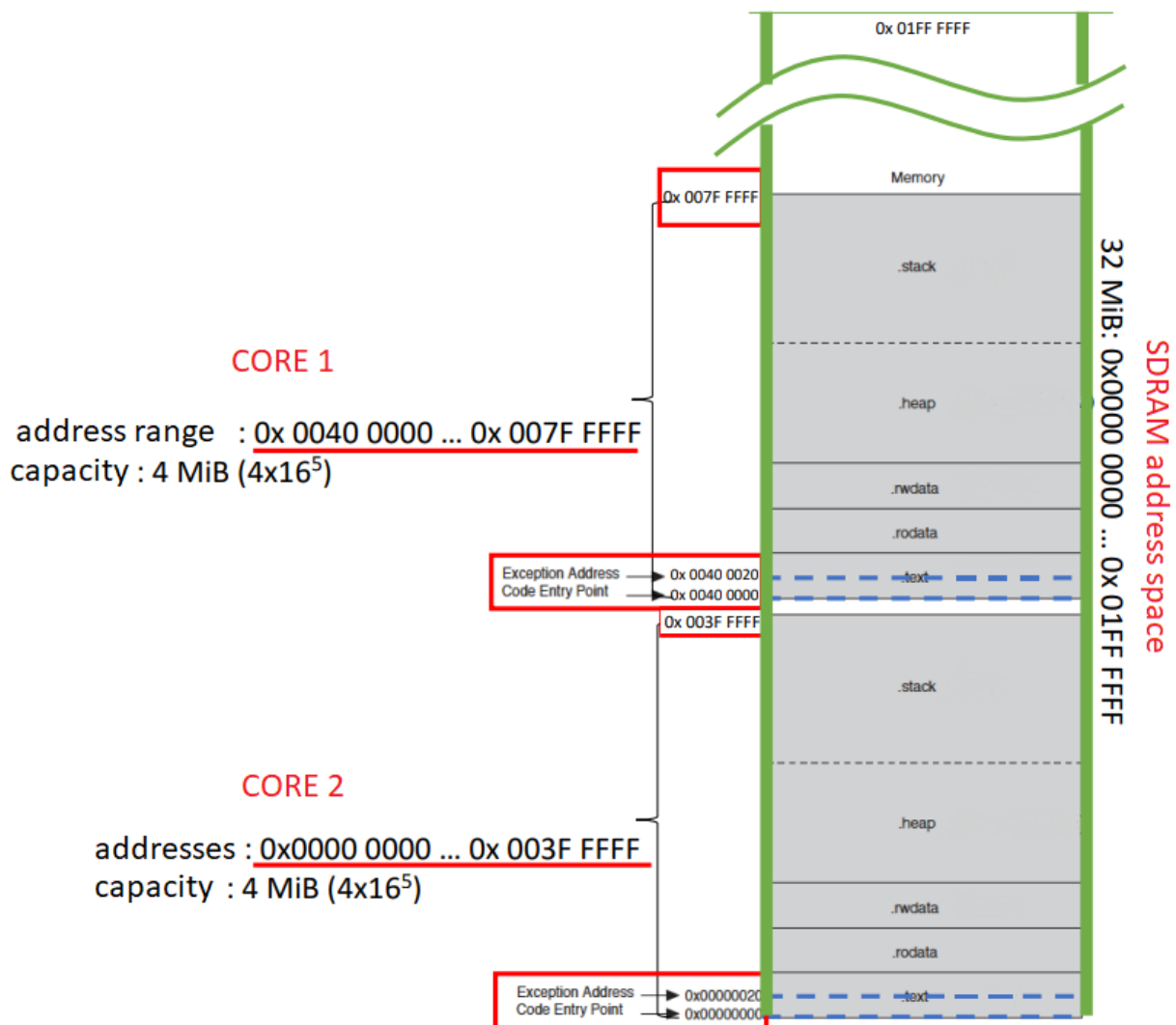


Figure 6: Linking areas of the main memory for the dual-core Nios II multiprocessors that are handled in this lab assignment

Part 2. Tutorials 1 and 2 for programming the Nios II multiprocessor

General description: In this part, two C programs are implemented on the Nios II multiprocessor using the DualCoreNios2e configuration in addition to the SBT and Command Shell tools. These two C programs

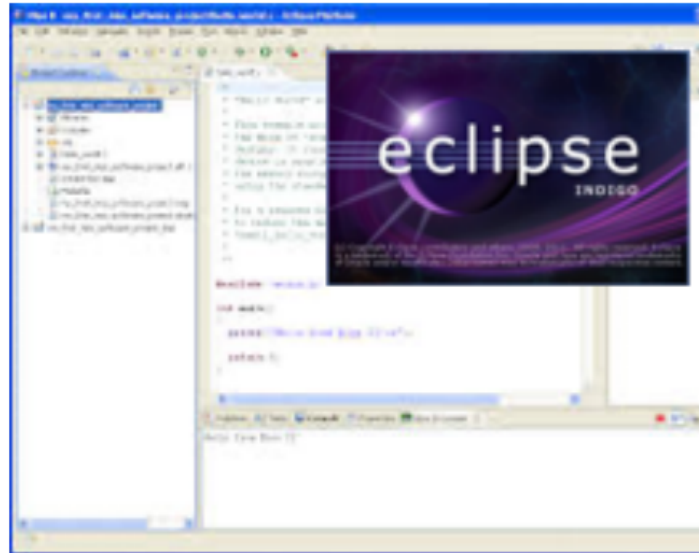


Figure 7: SBT user interface for Eclipse

make the CPU and CPU2 cores to access the same main memory location concurrently. This means that the processors never access at the same time. The MUTEX device allows coordinating access to the memory area shared by both cores.

Objective 2-1: Perform the steps mentioned below to run **Tutorial-1** in which only one processor core (CPU) is used.

1. Open: Nios II – Software Build Tools - Eclipse (see Figure 7).
2. File > New > Nios II app & BSP from Template (see Figure 8).
 - SOPC Information File name: DE0_Nano_DualCoreNios2e.sopcinfo
 - CPU name: CPU (do not select CPU2)
 - Project name: hola_chicos_y_chicas_0
 - Project Location: choose <ProjectDir>
 - Templates: select “Hello World Small”
 - Click “Next >”
3. Select “Create BSP”, project name: hola_chicos_y_chicas_0_bsp, use default location, click on “Finish” (see Figure 9).
 - Result: in the Eclipse “Project Explorer” window, a project of type “app C/C++” and another project “BSP” (Board Support Package) are created.
4. Display the project hola_chicos_y_chicas_0 in the “Project Explorer” window and click twice on `hello_world.c` (see Figure 10).
 - Result: a window with the source code is opened
5. Replace the content of the `hello_world.c` file in the project directory with the content of the file `Tutorial1/hola_chicos_y_chicas.c` (see Figure 11)
6. Compile and link (see Figure 12)
 - Right click on the C/C++ project in the Project Explorer window.
 - Select: Build Project.
 - Result: if the process ends successfully, the message “hola_chicos_y_chicas_0 Build complete” appears in the lower “Console” window. Two projects are created in the Project Explorer window and two folders in <ProjectDir>.

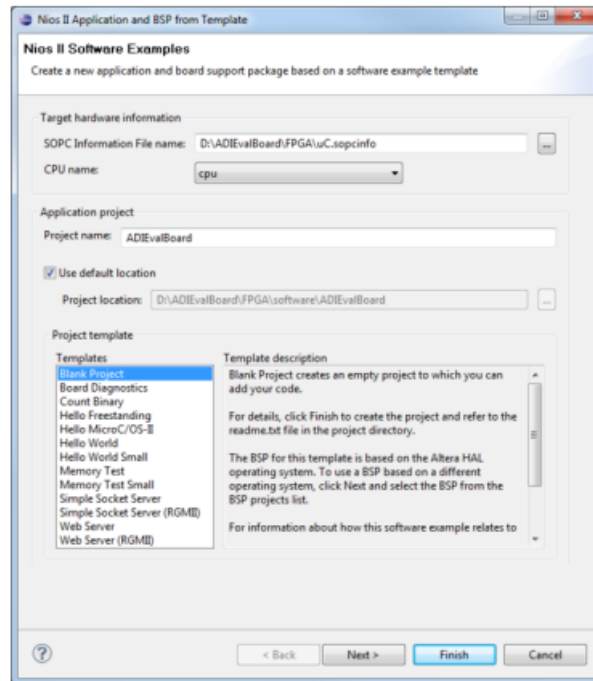


Figure 8: Templates for the generation of C programs using SBT.

7. DE0-Nano board configuration (see Figure 13):

- Connect the board to the USB connector
- Open: Start > Altera > Nios II Command Shell
- File DE0_Nano_Basic_Computer.sof is loaded in the board:
 - (a) `$ cd <directory with the FPGA configurations from lab assignment 4>`
 - (b) `$ nios2-configure-sof DE0_Nano_DualCoreNios2e.sof` (try it several times if it gives JTAG error)
- Result: the message “Quartus II 32-bit Programmer was successful” should appear (see Figure 13).

8. Program execution (see Figure 14):

- `$ cd <project directory hola_chicos_y_chicas.0>`
- `$ nios2-download -r -g -i 0 hola_chicos_y_chicas_0.elf`

9. View results (see Figure 15):

- `$ nios2-terminal`
 - (a) Result: the corresponding messages to the printf of the source code will display on the screen
- Close Command Shell window

Files for Tutorial-1:

- C source code: `hola_chicos_y_chicas.c` (folder: Tutorial1).
- FPGA configuration files: `DE0_Nano_DualCoreNios2e.sopcinfo`, `DE0_Nano_DualCoreNios2e.sof` (folder: DualCoreNios2e).

Objective 2-2: Perform the steps mentioned below to run **Tutorial 2** using two processor cores (CPU and CPU2). Two projects are created following the steps followed in Tutorial-1.

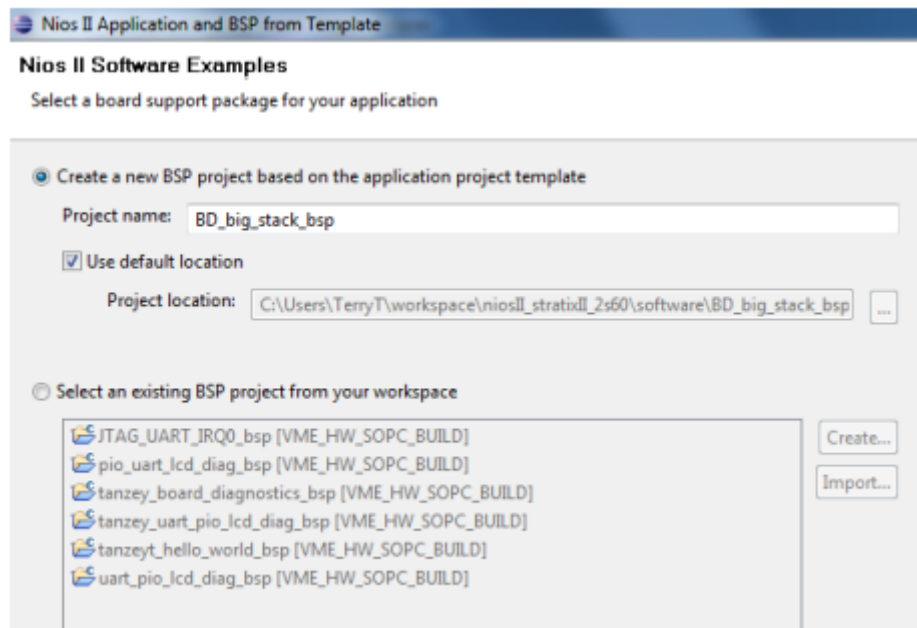


Figure 9: Window where the BSP (Board Support Package) project is defined.

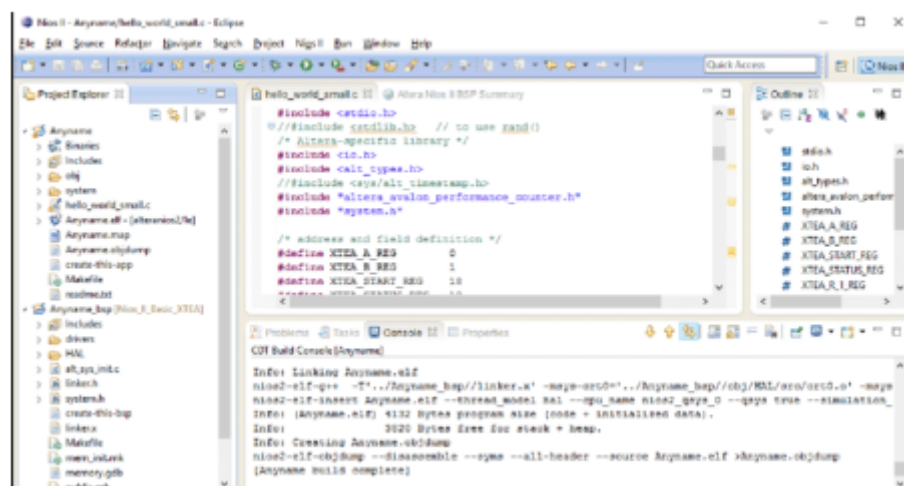


Figure 10: Window where the C source code is displayed.

1. Common elements of both projects:

- SOPC Information File name: DE0_Nano_DualCoreNios2e.sopcinfo
- Project Location: choose <ProjectDir>
- Templates: select “Hello World Small”

2. Project 1:

- CPU name: CPU (do not select CPU2)
- Project name: hola_semaforo.0
- BSP project, project name: hola_chicos_y_chicas.0_bsp
- Replace the content of the `hello_world.c` file in the project directory with the content of the file `Tutorial2/hola_semaforo.0.c` (see Figure ??). Note that CPU only reads the `message_buffer_val` variable. The display of results obtained by CPU is done using the `printf()` function. Only CPU is connected to the JTAG interface so its program is the only one that can include `printf`.
- Compile and link


```

#include <stdio.h>

int main()
{
    printf("Hola chicos de AC!\n");
    printf("Hola chicas de AC!\n");

    return 0;
}

```

Figure 11: Source code in C of Tutorial 1. File: hola_chicos_y_chicas.c

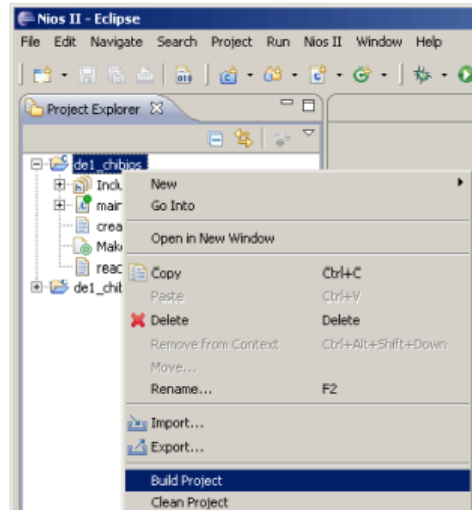


Figure 12: Window where the compile and link command (Build Project) of the C code is displayed.

3. Project 2:

- CPU name: CPU2
- Project name: hola_semaforo.1
- BSP project, project name: hola_chicos_y_chicas.1_bsp
- Replace the content of the `hello_world.c` file in the project directory with the content of the file `Tutorial12/hola_semaforo.1.c` (see Figure 17). Note that CPU2 modifies the `message_buffer_val` variable.
- Compile and link

4. DE0-Nano board configuration:

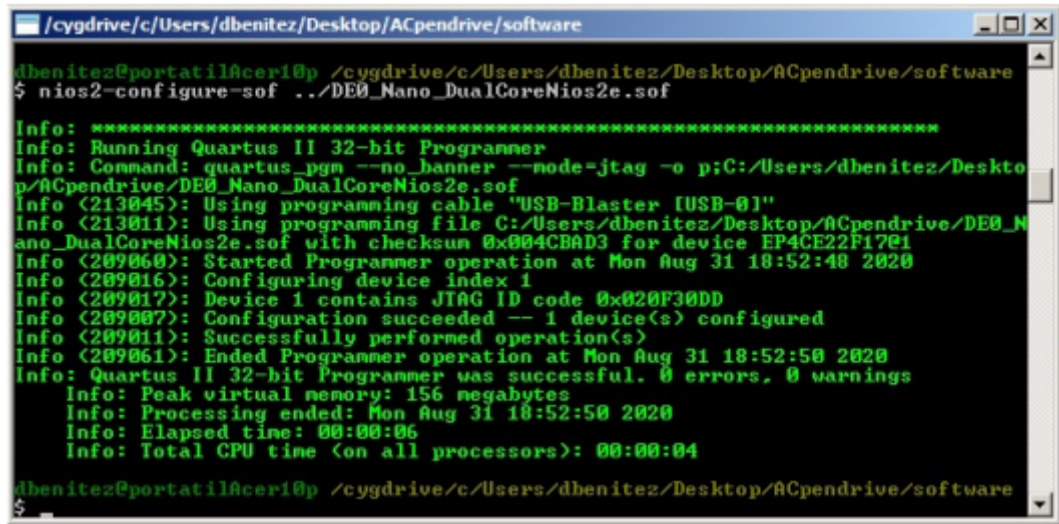
- Connect the board to the USB connector
- Open: Start > Altera > NiosII Command Shell
- File `DE0_Nano_Basic_Computer.sof` is downloaded into the board:
 - (a) `$ cd <directory with the FPGA configurations from lab assignment 4>`
 - (b) `$ nios2-configure-sof DE0_Nano_DualCoreNios2e.sof` (try it several times if a JTAG error is displayed)
- Result: the message "Quartus II 32-bit Programmer was successful" should appear

5. Run the program (see Figure 18):

- `$ nios2-download -r -g -i 0 hola_semaforo_0/hola_semaforo_0.elf`
- `$ nios2-download -r -g -i 1 hola_semaforo_1/hola_semaforo_1.elf`

6. View results (see Figure 19):

- `$ nios2-terminal`
- `$ CTRL-C`



```

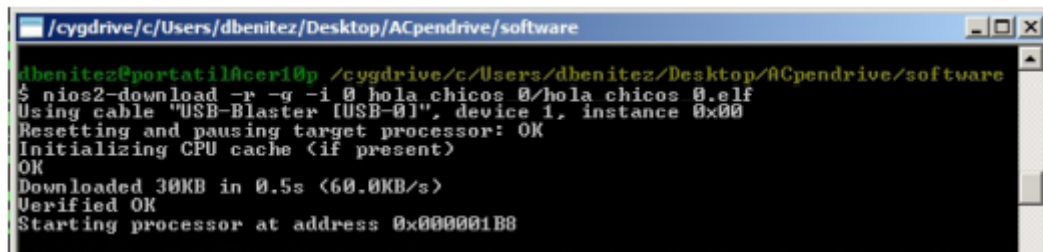
/cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
dbenitez@portatilaceri0p /cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
$ nios2-configure-sof ../DE0_Nano_DualCoreNios2e.sof

Info: *****
Info: Running Quartus II 32-bit Programmer
Info: Command: quartus_pgm --no_banner --mode-jtag -o p:C:/Users/dbenitez/Desktop/ACpendrive/DE0_Nano_DualCoreNios2e.sof
Info (213045): Using programming cable "USB-Blaster [USB-01]"
Info (213011): Using programming file C:/Users/dbenitez/Desktop/ACpendrive/DE0_Nano_DualCoreNios2e.sof with checksum 0x004CBAD3 for device EP4CE22F17E1
Info (209060): Started Programmer operation at Mon Aug 31 18:52:48 2020
Info (209016): Configuring device index 1
Info (209017): Device 1 contains JTAG ID code 0x020F30DD
Info (209007): Configuration succeeded -- 1 device(s) configured
Info (209011): Successfully performed operation(s)
Info (209061): Ended Programmer operation at Mon Aug 31 18:52:50 2020
Info: Quartus II 32-bit Programmer was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 156 megabytes
Info: Processing ended: Mon Aug 31 18:52:50 2020
Info: Elapsed time: 00:00:06
Info: Total CPU time (on all processors): 00:00:04

dbenitez@portatilaceri0p /cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
$

```

Figure 13: Result of the `nios2-configure-sof` command to configure the FPGA of the DE0-Nano board.



```

/cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
dbenitez@portatilaceri0p /cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
$ nios2-download -r -g -i 0 hola_chicos 0/hola_chicos 0.elf
Using cable "USB-Blaster [USB-01]", device 1, instance 0x00
Resetting and pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 30KB in 0.5s (60.0KB/s)
Verified OK
Starting processor at address 0x000001B8

```

Figure 14: Result of the `nios2-download` command to load the executable program into the SDRAM memory of the DE0-Nano board.

Files for Tutorial-2:

- C source code: `hola_semaforo_0.c`, `hola_semaforo_1.c` (folder: Tutorial2).
- FPGA configuration files: `DE0_Nano_DualCoreNios2e.sopcinfo`, `DE0_Nano_DualCoreNios2e.sof` (folder: DualCoreNios2e).

Part 3. Multithreaded parallel programming and performance evaluation of Nios II dual-core multiprocessors

General description: In this part of the lab, **Tutorial 3** is developed where the Matrix \times Vector multiplication algorithm is implemented. This algorithm will run on a single processor core and two multiprocessors based on Nios II/e and Nios II/s, respectively. Finally, an exercise is proposed where it is implemented the matrix multiplication algorithm.

The C code of Tutorial 3 consists of a loop of `Niter` iterations in which each iteration multiplies a matrix of $n=16$ rows \times $m=16$ columns ($A[i \times m + j]$) and a vector of 16 components ($x[j]$). The result consists of 16 components ($y[i]$): $y[i] = A[i \times m + j] \times x[j]$. Matrix and vectors components are integer values. The mathematical operation is: $y = A \times x$ (see Figure 20).

The main loop in C of Tutorial 3 is shown in Figure 21.

In this tutorial, execution time will be measured using the Timer and its HAL (Hardware Abstraction Layer) function/driver is called `alt_timestamp()`. To correctly configure the time recording, it is necessary to perform the following steps.

```

/cygdrive/c/altera/12.1sp1
dhenitez@portatilaceri8p /cygdrive/c/altera/12.1sp1
$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-01]", device 1, instance 0
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>

Hola chicos de AC?
Hola chicas de AC?

nios2-terminal: exiting due to ^C on host

```

Figure 15: Display of Tutorial 1 messages after running the `nios2-terminal` command in a new Command Shell window.

Timer setting of the CPU processor (first dual core processor):

1. Generate a SBT project with the file `DE0_Nano_DualCoreNios2e.sopcinfo` for the CPU processor following the same steps as in Tutorial 1. The following project name can be used: `MV_serie`. The software project includes a BSP project.
2. Edit the configuration of the BSP project (`MV_serie_bsp`). To do this, you need to perform the following actions: (right click on the BSP project) Nios II > BSP editor > timestamp_timer > Value= interval_timer > Save.
3. In the BSP editor and in the "Target BSP Directory" window: click on Generate.
4. Compile + link C program: (right click on C project) > Build Project.

Note: If the execution of the `*.elf` program indicates that the execution time is 0, it means that the Timer driver is not properly configured. In this case, try to compile the program by executing `make` in the `MV_serie` project directory

As in Tutorial 2, display of results can only be done through the CPU core using the `printf()` function.

Objective 3-1: Follow the steps in Tutorial 1 using `MV_serie.c` so that a SBT project is generated. Next, configure the DE0-Nano board using the FPGA configuration: *DualCoreNios2e*, and run the sequential program on the CPU core. Next, do the following exercises:

1. Register the execution times for four workloads.
 - Parameters: `Niter` = 1000, 2000, 5000, 10000.
2. Performance evaluation: fill in Tables 1 and 2, whose columns represent the following items:
 - `add Noperations`: number of add operations in all Niter iterations
 - `mult Noperations`: number of mult operations in all Niter iterations
 - `Nloads`: number of load instructions
 - `Nstores`: number of store instructions
 - Time: execution time of the entire program
3. Repeat the results for Nios II/s using the processor core called CPU that is integrated in the FPGA configuration: *DualCoreNios2s*.

Questions to justify the results

1. Is it reasonable that doubling the number of arithmetic operations and memory accesses would cause the execution time of one of the Nios II/e processors in the DualCoreNios2e multi-processor to be doubled? Why?
2. Is it reasonable that the results of execution times provided by Nios II/s are significantly lower than those obtained when Nios II/e is used? Why? What is the average speed-up provided by the Nios II/s core relative to the Nios II/e core?

```
#include <stdio.h>
#include <system.h>
#include <altera_avalon_mutex.h>
#include <unistd.h>

int main(){

// Message buffer memory address: 0x 0400 0000
volatile int * message_buffer_ptr = (int *) MESSAGE_BUFFER_RAM_BASE;

printf("Hola, soy CPU!\n");

/* the driver of the mutex type hardware device is saved */
alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");

int message_buffer_val= 0x0;
int iteraciones      = 0x0;

while(1) {
    iteraciones++;
    /* CPU asks to be owner of mutex, assigning the value 1 */
    altera_avalon_mutex_lock(mutex,1);
    message_buffer_val = *(message_buffer_ptr); /* reads buffer's value */
    altera_avalon_mutex_unlock(mutex);          /* release mutex */
    printf("CPU - iter: %i - message_buffer_val: %08X\n",
           iteraciones, message_buffer_val);
    usleep(1000000); /* wait 1 sec      = 106 useg */
}
return 0;
}
```

Figure 16: Source C code (hola_semaforo_0.c) for CPU in Tutorial-2.

```
#include <stdio.h>
#include <system.h>
#include <altera_avalon_mutex.h>

int main(){

// Message buffer memory address: 0x 400 0000
volatile int * message_buffer_ptr = (int *) MESSAGE_BUFFER_RAM_BASE;

/* the driver of the mutex type hardware device is saved */
alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");

int message_buffer_val      = 0x0;

while(1) {
    /* CPU asks to be owner of mutex, assigning the value 2 */
    altera_avalon_mutex_lock(mutex,2);
    /* Saves in buffer the value modified in CPU2 */
    *(message_buffer_ptr) = message_buffer_val;
    altera_avalon_mutex_unlock(mutex); /* release mutex */
    message_buffer_val++;
}

return 0;
}
```

Figure 17: C source code (hola_semaforo_1.c) for CPU2 processor used in Tutorial-2.

```

/cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
dbenitez@portatilAceri0p /cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
$ nios2-download -r -g -i 0 hola_semaforo_0/hola_semaforo_0.elf
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Resetting and pausing target processor: OK
Initializing CPU cache <if present>
OK
Downloaded 59KB in 1.0s <59.0KB/s>
Verified OK
Starting processor at address 0x000001B8

dbenitez@portatilAceri0p /cygdrive/c/Users/dbenitez/Desktop/ACpendrive/software
$ nios2-download -r -g -i 1 hola_semaforo_1/hola_semaforo_1.elf
Using cable "USB-Blaster [USB-0]", device 1, instance 0x01
Resetting and pausing target processor: OK
Initializing CPU cache <if present>
OK
Downloaded 8KB in 0.1s
Verified OK
Starting processor at address 0x004001B4

```

Figure 18: Results of the `nios2-download` commands to download the executable programs into the SDRAM memory of the DE0-Nano board.

```

/cygdrive/c/altera/12.isp1
dbenitez@portatilAceri0p /cygdrive/c/altera/12.isp1
$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>

Hola, soy CPU!
CPU - iter: 1 - message_buffer_val: 0002AA34
CPU - iter: 2 - message_buffer_val: 0002AA34
CPU - iter: 3 - message_buffer_val: 0002AA34
CPU - iter: 4 - message_buffer_val: 000115AB
CPU - iter: 5 - message_buffer_val: 00027914
CPU - iter: 6 - message_buffer_val: 0003DCD6
CPU - iter: 7 - message_buffer_val: 00054052
CPU - iter: 8 - message_buffer_val: 0006A3CE
CPU - iter: 9 - message_buffer_val: 00080748

nios2-terminal: exiting due to ^C on host

```

Figure 19: Messages displayed in Tutorial-2 after executing the command `nios2-terminal` in a Command Shell window.

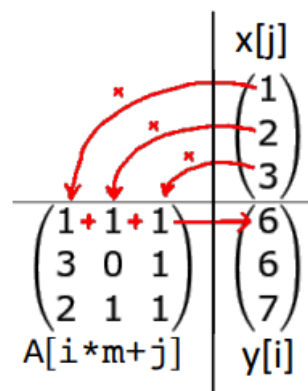


Figure 20: Matrix \times Vector algorithm: $y[i] = A[i \times m + j] \times x[j]$.

```

for (i=my_first_row; i<=my_last_row; i++) {
    for(j=0; j<m; j++) y[i] += A[i*m+j] * x[j]; }

int main(){

// Shared memory zone for vector and matrix
volatile int * x      = (int *) 0x6440; // 16x1 x4 =64B: 0x6440 - 0x647F
volatile int * y      = (int *) 0x6480; // 16x1 x4 =64B: 0x6480 - 0x64BF
volatile int * A      = (int *) 0x6000; // 16x16x4=1KB: 0x6000 - 0x63FF

// Matrix x Vector operation
int local_n           = n;
int my_first_row = 0; // 1st row assigned to this core
int my_last_row  = local_n - 1; // last row assigned to this core

for (k = 0; k < Niter; k++) {
    iteraciones++;
    for (i=my_first_row; i<=my_last_row; i++) {
        for(j=0; j<m; j++) y[i] += A[i*m+j] * x[j];
    }
}
} // main()

```

Figure 21: MV_serie.c

Table 1: Analysis of the computational load and number of memory accesses of the Matrix \times Vector algorithm.

N_{iter}	add $N_{operations}$	mult $N_{operations}$	N_{loads}	N_{stores}
1000				
2000				
5000				
10000				

Table 2: Measurements of execution time of the Matrix \times Vector sequential algorithm for one of the processors (CPU) integrated into a Nios II dual-core multiprocessor.

FPGA configuration	N_{iter}	Time (ms)	Speed-up
DualCoreNios2e	1000		1
DualCoreNios2e	2000		1
DualCoreNios2e	5000		1
DualCoreNios2e	10000		1
DualCoreNios2s	1000		
DualCoreNios2s	2000		
DualCoreNios2s	5000		
DualCoreNios2s	10000		

Objective 3-2: Follow the steps done in Tutorial-2 using `MV_paralelo_maestro.c` (see Figure 22) and `MV_paralelo_esclavo.c` (see Figure 23) for generating two projects in SBT tool. Next, configure the DE0-Nano board employing the FPGA configuration: *DualCoreNios2e*. Finally, run the parallel program using CPU and CPU2 cores.

Activating the Timer controller for the CPU processor (first processor of the dual-core):

1. Generate two SBT projects with the `DE0_Nano_DualCoreNios2e.sopcinfo` file, one for the CPU processor and the other for the CPU2 processor. The following project names can be used: `MV_paralelo_maestro`, `MV_paralelo_esclavo`. Each project includes a C sub-project and another BSP sub-project.
2. Edit the configuration of the BSP project for the CPU processor (`MV_paralelo_maestro.bsp`): (right click on the BSP project) Nios II > BSP editor > timestamp.timer > Value = interval.timer > Save.
3. In the BSP editor > "Target BSP Directory" window: Generate.
4. Compile and link C program: (right click on C project) > Build Project. A `*.elf` file is generated.

Note: If the execution of the `*.elf` program indicates that the execution time is 0, it means that the driver for the Timer controller is not properly configured. In this case, try to compile the two parallel programs by executing the command `make` in both projects directories `MV_paralelo_maestro` and `MV_paralelo_esclavo`.

Next, the following activities should be done:

1. Configure the FPGA using `DualCoreNios2e.sof` file and `nios2-configure-sof` command (dual core: $2 \times$ Nios II/e).
2. Download the program using `nios2-download` command.
3. Register the execution times activating a single thread and using CPU processor for 4 workloads. The parameters are different for each run:
 - `thread_count = 1`
 - `Niter = 1000, 2000, 5000, 10000`.
4. Register the execution times activating two threads, using CPU and CPU2 cores for 4 workloads. Parameters:
 - `thread_count = 2`
 - `Niter = 1000, 2000, 5000, 10000`.
5. Performance evaluation: fill in Table 3 using the execution times obtained in steps 3 and 4 above. Additionally, calculate speed-up considering that the reference configuration is a Nios II/e processor core.
6. Repeat the results using *DualCoreNios2s* multiprocessor (dual core: $2 \times$ Nios II/s). For the performance analysis, consider the reference configuration to be a single Nios II/s processor.

Questions to justify the results

1. Are the results similar to those obtained in Objective 3-1?
2. Why? Justify the results involving an analysis of parallelism efficiency ($100\% \times \text{speed-up} / \text{number of threads}$)


```

int main(){
alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");
int thread_count = 2;    // number of threads
int rank          = 0;    // master thread for CPU core
int Niter         = 2000; // times that repeat matrix - vector; other values = 1000, 2000, 5000

// FORK - Synchronization of Separation      : *MESSAGE_BUFFER_RAM_BASE = 15
message_buffer_val = 15; // ID=15 (0xF) indicates that fork starts
message_buffer_val_join = 0; // ID=0 indicates that memory is initialized
altera_avalon_mutex_lock(mutex,1);           // blocks mutex
*(message_buffer_ptr) = message_buffer_val;   // initializes RAM FORK
*(message_buffer_ptr_join)= message_buffer_val_join; // initializes RAM JOIN
altera_avalon_mutex_unlock(mutex);           // releases mutex

// Master computation - Matrix x Vector
int local_n = n / thread_count;
int my_first_row = rank * local_n;           // 1st row assigned to this core
int my_last_row = (rank+1) * local_n - 1;    // last row assigned to this core
for (k = 0; k < Niter; k++) {
    iteraciones++;
    for (i=my_first_row; i<=my_last_row; i++){
        for(j=0; j<m; j++) y[i] += A[i*m+j] * x[j];
    }
}

// JOIN Synchronization = thread unification barrier
altera_avalon_mutex_lock(mutex,1);           /* blocks mutex */
message_buffer_val_join = *(message_buffer_ptr_join); /* reads value in RAM */
message_buffer_val_join |= 0x1;               /* ID=1: ends thread 0 */
*(message_buffer_ptr_join) = message_buffer_val_join; /* stores the value in RAM */
altera_avalon_mutex_unlock(mutex);           /* releases mutex */
while( (message_buffer_val != 6) ){
    altera_avalon_mutex_lock(mutex,1);        /* blocks mutex */
    message_buffer_val = *(message_buffer_ptr); /* reads value in RAM */
    message_buffer_val_join = *(message_buffer_ptr_join);
    altera_avalon_mutex_unlock(mutex);        /* releases mutex */
    if ( (message_buffer_val_join == 0x3 && thread_count == 2 ) ||
        (message_buffer_val_join == 0x1 && thread_count == 1) ){
        dummy = 6;
        altera_avalon_mutex_lock(mutex,1);    /* blocks mutex */
        *(message_buffer_ptr) = dummy;        /* writes value in buffer */
        altera_avalon_mutex_unlock(mutex);    /* releases mutex */
        message_buffer_val = dummy;
    }
}
} // main()

```

Figure 22: MV.paralelo_maestro.c

```

int main() {
alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");
int thread_count = 2;          // Number of threads
int rank          = 1;          // Master thread for CPU2 core

// FORK of thread 1 , Synchronization from thread 0 , message_buffer_val=15
while(message_buffer_val != 15) {
    altera_avalon_mutex_lock(mutex, 2);          /* blocks mutex */
    message_buffer_val = *(message_buffer_ptr); /* reads value in buffer */
    altera_avalon_mutex_unlock(mutex);           /* releases mutex */
}

// Slave Computation - matrix-vector
int local_n = n / thread_count;
int my_first_row = rank * local_n;          // 1st row assigned to this core
int my_last_row = (rank+1) * local_n - 1; // last row assigned to this core
for (k = 0; k < Niter; k++) {
    for (i=my_first_row; i<=my_last_row; i++){
        for(j=0; j<m; j++) y[i] += A[i*m+j] * x[j];    } }

// JOIN - threads unification
altera_avalon_mutex_lock(mutex, 2);
message_buffer_val_join = *(message_buffer_ptr_join);
message_buffer_val_join |= 0x2; /* ID=2 thread1synchronization */
*(message_buffer_ptr_join) = message_buffer_val_join;
altera_avalon_mutex_unlock(mutex);

} // main()

```

Figure 23: MV_paralelo_esclavo.c

Table 3: Performance evaluation of the Matrix \times Vector algorithm for 1 and 2 threads using CPU and CPU2 processor of two Nios II multiprocessors.

FPGA configuration	Number of threads	N_{iter}	Time (ms)	Speed-up	Parallelism efficiency
DualCoreNios2e	1	1000		1	100 %
DualCoreNios2e	1	2000		1	100 %
DualCoreNios2e	1	5000		1	100 %
DualCoreNios2e	1	10000		1	100 %
DualCoreNios2e	2	1000			
DualCoreNios2e	2	2000			
DualCoreNios2e	2	5000			
DualCoreNios2e	2	10000			
DualCoreNios2s	1	1000		1	100 %
DualCoreNios2s	1	2000		1	100 %
DualCoreNios2s	1	5000		1	100 %
DualCoreNios2s	1	10000		1	100 %
DualCoreNios2s	2	1000			
DualCoreNios2s	2	2000			
DualCoreNios2s	2	5000			
DualCoreNios2s	2	10000			

```

void Matrix-Matriz (int n, int* A, int* B, int* C)
{
    int i,j,k;
    for (i = 0; i < n; ++i)          /* i: column of the matrix */
        for (j = 0; j < n; ++j) {    /* j: row of the matrix */
            int cji = C[j*n+i];      /* cji ← C[j][i] */
            for (k = 0; k < n; k++)    /* cji: row -B(j) x column -A(i) */
                cji += B[j*n+k] * A[k*n+i]; /* cji += B[j][k]*A[k][i] */
            C[j*n+i] = cji;          /* C[j][i] ← cji */
        }
    }
}

```

Figure 24: Procedure for matrix multiplication: $C = B \times A$.**Objective 3-3:**

Develop a C source code, run, and evaluate the performance of a Matrix \times Matrix ($C[] = B[] \times A[]$) algorithm that performs matrix multiplication. The steps to be performed are as follows:

1. Code the matrix multiplication algorithm using matrices with n rows \times n columns and 4-byte integer data. The input matrices are called A and B , and the result matrix is called C : $C = B \times A$ (see Figure 24). To do this, you need to develop two source programs that will be used to generate the programs for the master and slave threads. In each program you need:
 - Initialize the pointers to the start address of each matrix: A, B, C .
 - Assign the workload to each thread.
 - Develop the FORK and JOIN synchronization events for each thread.
 - Compile and link.
2. Run the program using DualCoreNios2e (2 cores: $2 \times$ Nios II/e) and DualCoreNios2s (2 cores: $2 \times$ Nios II/s) multiprocessors. Parameters:
 - thread_count = 1, 2.
 - Niter = 1000, 2000, 5000, 10000.
3. Performance evaluation: fill in a table similar to Table 3 for the matrix multiplication algorithm. This table should show the execution times, speed-up and parallel efficiency of the programs.

Questions to justify the results

1. Are the results similar to Objective 3-2?
2. Why? Justify the results involving the analysis of the parallelism efficiency ($100\% \times \text{speed-up} / \text{number of threads}$).

Material to be provided

1. You should provide the following material related to Objective 3-3: two C files of the parallel version corresponding to the master and slave threads and a table with results of the performance evaluation.

Files for Part 3:

- C source code: MV_paralelo_maestro.c, MV_paralelo_esclavo.c (folder: Tutorial3).
- FPGA configuration files: DE0_Nano_DualCoreNios2e.sopcinfo, DE0_Nano_DualCoreNios2e.sof (folder: DualCoreNios2e); DE0_Nano_DualCoreNios2s.sopcinfo, DE0_Nano_DualCoreNios2s.sof (folder: DualCoreNios2s).

References

- [1] Altera. Quartus II Handbook Version 9.0 Volume 5: Embedded Peripherals, 2009.
- [2] Altera. Creating Multiprocessor Nios II Systems - Tutorial. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tt/tt_nios2_multiprocessor_tutorial.pdf, 2011.
- [3] Altera. Nios II Hardware Development Tutorial, 2011.
- [4] Altera. My First Nios II Software - Tutorial. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tt/tt_my_first_nios_sw.pdf, 2012.
- [5] Altera. Basic Computer System for the Altera DE0 Nano Board. ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/12.0/Computer_Systems/DE0/DE0_Basic_Computer.pdf, 2013. Accessed: 2022-01-19.
- [6] Intel. Embedded Design Handbook. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/edh_ed_handbook.pdf, 2020.
- [7] Intel. Embedded Peripherals IP User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf, 2021.
- [8] Intel. Nios II Software Developer Handbook. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2sw_nii5v2gen2.pdf, 2021.