

## Computer Architecture - Lab Assignment 4

# Nios V multiprocessors implementation, parallel programming and performance evaluation

## Introduction

The main objective of this laboratory assignment consists on the performance evaluation of a Nios V dual-core multiprocessor and the comparison with a single Nios V processor [1, 3]. 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 [2, 5]. This assignment contains three parts that are summarized below.

Part 1. Intel-Altera Software Building Tools and Nios V 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 all needed software tools to interact with the mentioned board. The first tutorial runs on a single processor and the second runs on a dual-core multiprocessors in parallel.

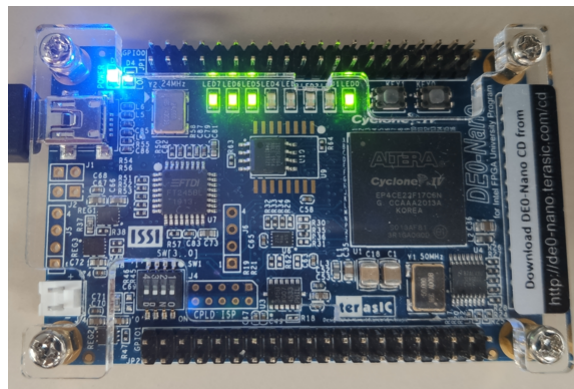


Figure 1: DE0-Nano board [2].

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 V family of processors:  $2 \times$  Nios V/m,  $2 \times$  Nios V/g. 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`):

- Tutorial-1
- Tutorial-2
- Tutorial-3

- DualCoreNiosVm
- DualCoreNiosVg

## 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 V multiprocessors will also be described. The multiprocessors will be configured in the FPGA device of the DEO-Nano board (see Figure 1), in which the C programs will run.

Nios V Command Shell: board setup, compiling, linking, downloading and running the programs

With the Nios V Command Shell of Quartus Prime 24.1 software tool you can create, modify, compile and run programs for Nios V processors (see Figure 2). To do this, the commands are written through the keyboard in the Command Shell window or inserted into a script file [4]. The patterns of the commands that will be used in this assignment are the following:

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

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

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

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

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

```
$ niosv-terminal.exe
```

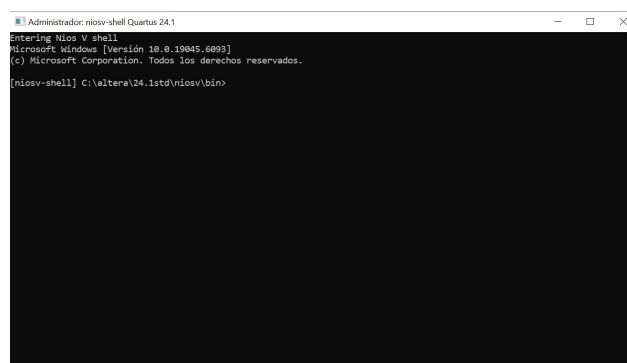


Figure 2: Nios V Command Shell window with prompt ready to enter commands that allow to interact with the DE0-Nano board.

### Nios V 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 V cores called `intel_niosv_m_0` and `intel_niosv_m_1` is physically implemented. Additionally, the FPGA also integrates the following hardware modules:

- A semaphore device for 8-byte atomic operations. 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 (MUTEX)*.

- 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 and address space. 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 is divided.

Other element of the DE0-Nano board that is necessary for this lab is a 32 MiB SDRAM memory. In Figure 3, a diagram of the Nios V multiprocessor micro-architecture is shown.

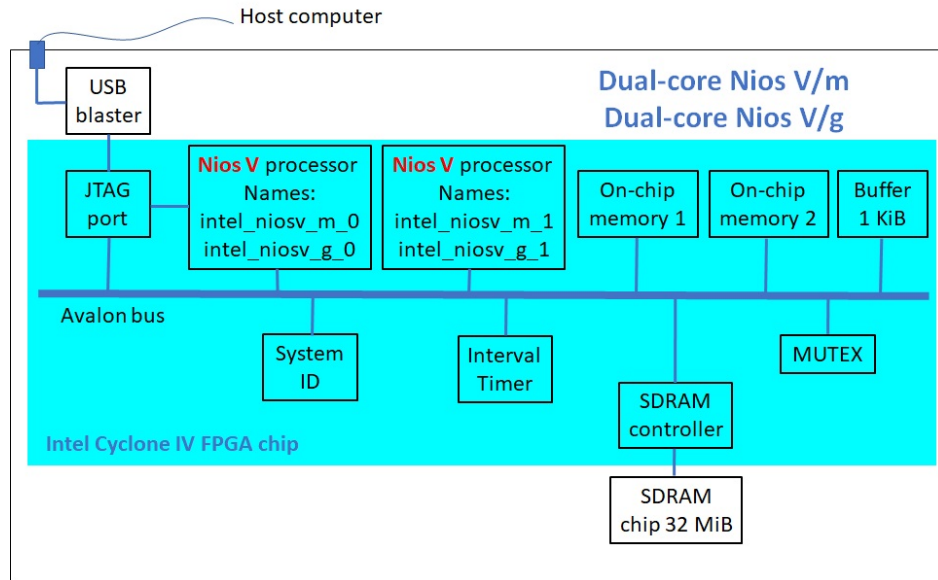


Figure 3: Nios V dual-core multiprocessor with shared memory parallel architecture. In this lab, two FPGA configurations that have a Nios V multiprocessor, called *DualCoreNiosVm* and *DualCoreNiosVg*, are handled.

Two different Nios V multiprocessor configurations that differ in the processor will be handled. One of them, called *DualCoreNiosVm*, integrates two copies of the Nios V/m core and the other configuration, called *DualCoreNiosVg*, integrates two copies of the Nios V/g core. The Nios V/g processor includes 4 KiB instruction cache and 4 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 Nios V software tools provide memory partitioning that allows multiple processors to run their software from different regions of the same physical memory device. Additionally, these tools ensure 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, software tools for Nios V use the exception address to calculate the memory area allocated for each processor.

Each processor has five linking zones (see Figure 4):

- **.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

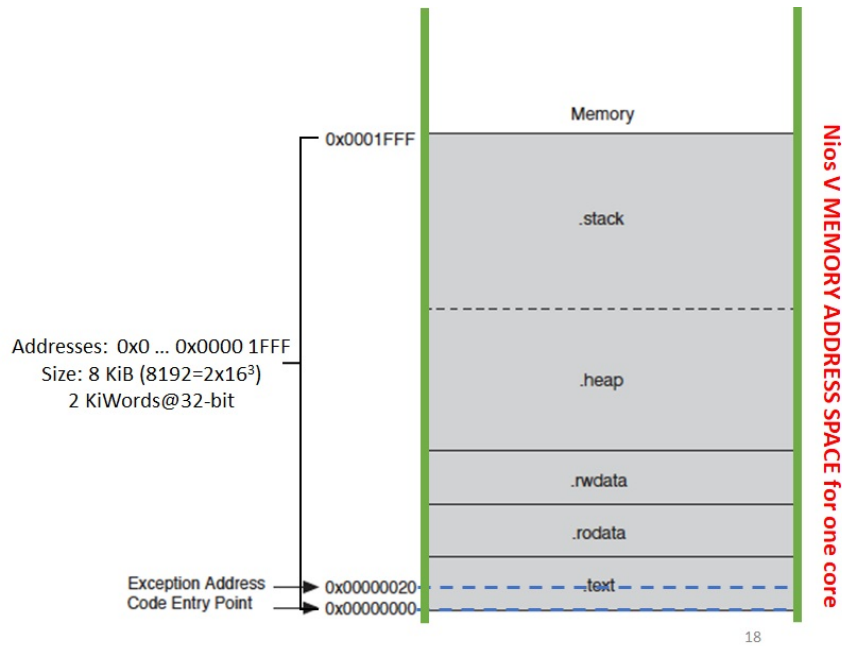


Figure 4: Example of main memory link zones for one Nios V core.

The Nios V software tools ensure that these memory sections are located (linked) at fixed memory addresses.

In Figure 4 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 4 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 assignment, 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 V cores, `intel_niosv_m_0` and `intel_niosv_m_1`, which are arranged in the FPGA configurations mentioned above: `DualCoreNiosVm` and `DualCoreNiosVg`, are established. In Figure 5 it can be seen that the address ranges assigned to the processor cores are:

- 0x0000 0000 - 0x003F FFFF - `intel_niosv_m_0` core
- 0x0040 0000 - 0x007F FFFF - `intel_niosv_m_1` core

Each core is allocated 4 MiB of main memory to run its software. The Nios V software tools automatically partition main memory by looking at the *reset address*, also called *Code Entry Point*. For both cores of the multiprocessor, this address has the values 0x0000 0000 for `intel_niosv_m_0` and 0x0040 0000 for `intel_niosv_m_1`. See Figure 5 where the address assignments for the two cores are shown. The *code entry point* is the memory address where each Nios V processor expects to find the reset code.

## Part 2. Tutorials 1 and 2 for programming the Nios V multiprocessor

**General description:** In this part, two C programs are implemented on the Nios V multiprocessor using the `DualCoreNiosVm` configuration in addition to the Nios V Command Shell tools. These two C programs make

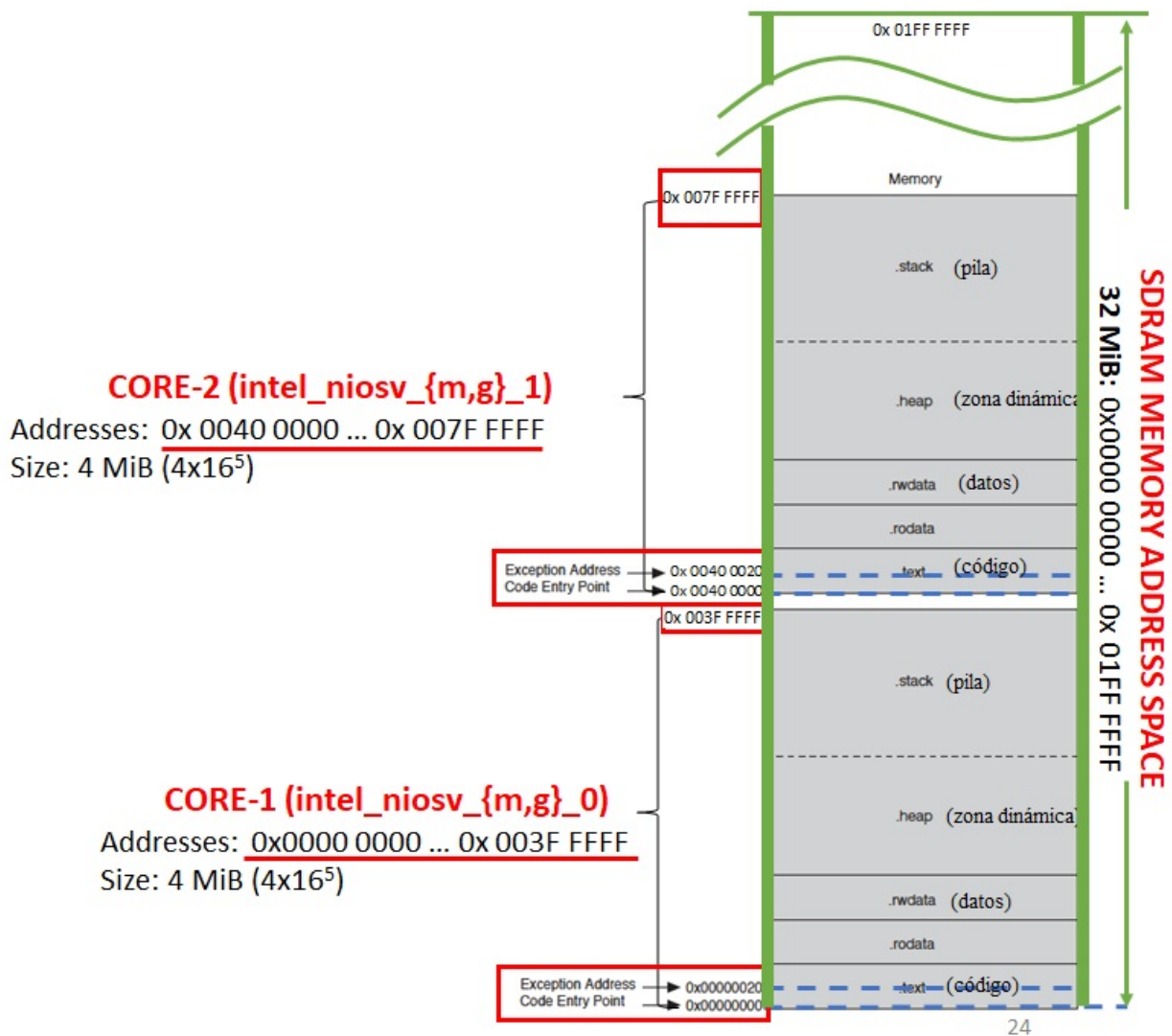


Figure 5: Linking areas of the main memory for the dual-core Nios V multiprocessors that are handled in this lab assignment

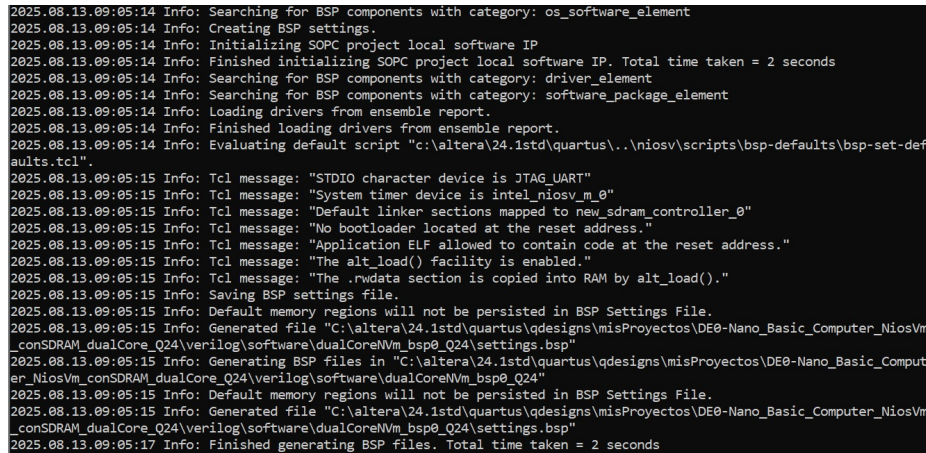
the `intel_niosv_m_0` and `intel_niosv_m_1` cores to access the same main memory location concurrently. This means that the processors never access at the same time. The MUTEX device allows coordinating access to the memory area shared by both cores.

**Objective 2-1:** Perform the steps mentioned below to run **Tutorial-1** in which only one processor core (`intel_niosv_m_0`) is used.

1. Open: Nios V Command Shell Tools (see Figure 2).
2. The BSP files for both cores of Nios V multiprocessor are created following the next steps.

dualCoreNVg\_bsp0\_Q24 folder contains BSP files for first core: intel\_niosv\_g\_0

```
$ sh
$ cd <working directory for this lab assignment>
$ mkdir dualCoreNVg_bsp0_Q24
$ cd dualCoreNVg_bsp0_Q24
$ niosv-bsp.exe -c -t=hal -s=DE0_Nano_DualCoreNiosVm.sopcinfo -i=intel_niosv_g_0
settings.bsp
Output: Info messages will be shown on the display (see Figure 6).
The altera_avalon_mutex.c file should be copied into the drivers/src folder
after executing the niosv-bsp command
$ cp altera_avalon_mutex.c drivers/src
```



```
2025.08.13.09:05:14 Info: Searching for BSP components with category: os_software_element
2025.08.13.09:05:14 Info: Creating BSP settings.
2025.08.13.09:05:14 Info: Initializing SOPC project local software IP
2025.08.13.09:05:14 Info: Finished initializing SOPC project local software IP. Total time taken = 2 seconds
2025.08.13.09:05:14 Info: Searching for BSP components with category: driver_element
2025.08.13.09:05:14 Info: Searching for BSP components with category: software_package_element
2025.08.13.09:05:14 Info: Loading drivers from ensemble report.
2025.08.13.09:05:14 Info: Finished loading drivers from ensemble report.
2025.08.13.09:05:14 Info: Evaluating default script "c:\altera\24.1std\quartus\...\niosv\scripts\bsp-defaults\bsp-set-def
aults.tcl".
2025.08.13.09:05:15 Info: Tcl message: "STDIO character device is JTAG UART"
2025.08.13.09:05:15 Info: Tcl message: "System timer device is intel_niosv_m_0"
2025.08.13.09:05:15 Info: Tcl message: "Default linker sections mapped to new sdram_controller_0"
2025.08.13.09:05:15 Info: Tcl message: "No bootloader located at the reset address."
2025.08.13.09:05:15 Info: Tcl message: "Application ELF allowed to contain code at the reset address."
2025.08.13.09:05:15 Info: Tcl message: "The alt_load() facility is enabled."
2025.08.13.09:05:15 Info: Tcl message: "The .rwdata section is copied into RAM by alt_load()."
2025.08.13.09:05:15 Info: Saving BSP settings file.
2025.08.13.09:05:15 Info: Default memory regions will not be persisted in BSP Settings File.
2025.08.13.09:05:15 Info: Generated file "C:\altera\24.1std\quartus\qdesigns\misProjectos\DE0-Nano_Basic_Computer_NiosVm
_conSDRAM_dualCore_Q24\verilog\software\dualCoreNvm_bsp0_Q24\settings.bsp"
2025.08.13.09:05:15 Info: Generating BSP files in "C:\altera\24.1std\quartus\qdesigns\misProjectos\DE0-Nano_Basic_Comput
er_NiosVm_conSDRAM_dualCore_Q24\verilog\software\dualCoreNvm_bsp0_Q24"
2025.08.13.09:05:15 Info: Default memory regions will not be persisted in BSP Settings File.
2025.08.13.09:05:15 Info: Generated file "C:\altera\24.1std\quartus\qdesigns\misProjectos\DE0-Nano_Basic_Computer_NiosVm
_conSDRAM_dualCore_Q24\verilog\software\dualCoreNvm_bsp0_Q24\settings.bsp"
2025.08.13.09:05:17 Info: Finished generating BSP files. Total time taken = 2 seconds
```

Figure 6: Result of the niosv-bsp.exe command to generate BSP files.

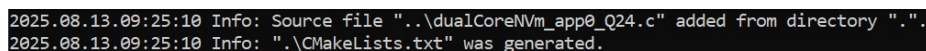
dualCoreNVg\_bsp1\_Q24 folder contains BSP files for second core: intel\_niosv\_g\_1

```
$ mkdir dualCoreNVg_bsp1_Q24
$ cd dualCoreNVg_bsp1_Q24
$ niosv-bsp.exe -c -t=hal -s=DE0_Nano_DualCoreNiosVm.sopcinfo -i=intel_niosv_g_1
settings.bsp
$ cp altera_avalon_mutex.c drivers/src
```

3. Compile and link Tutorial-1 application whose source file is dualCoreNVm\_app0\_Q24.c.

build folder contains ELF file for for Tutorial-1 program

```
$ mkdir ../dualCoreNVm_app0_Q24
$ cd ../dualCoreNVm_app0_Q24
$ cp dualCoreNVm_app0_Q24.c .
$ niosv-app.exe -a=. -b=../dualCoreNVm_bsp0_Q24 -s=.
Output: Info messages will be shown on the display (see Figure 7).
$ cmake -S . -G "Unix Makefiles" -B build
Output: Info messages will be shown on the display (see Figure 8).
$ make -C build
Output: the build folder is created and the dualCoreNVm_app0_Q24.elf file is
also created (see Figure 9).
```



```
2025.08.13.09:25:10 Info: Source file "..\dualCoreNVm_app0_Q24.c" added from directory ".".
2025.08.13.09:25:10 Info: ".\CMakeLists.txt" was generated.
```

Figure 7: Result of the niosv-app.exe command to generate ELF file.



```
-- Defaulting build type to Debug.
-- The C compiler identification is GNU 13.2.0
-- The CXX compiler identification is GNU 13.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/altera/24.1std/riscfree/toolchain/riscv32-unknown-elf/bin/riscv32-unknown-elf-gcc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/altera/24.1std/riscfree/toolchain/riscv32-unknown-elf/bin/riscv32-unknown-elf-g++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- The ASM compiler identification is GNU
-- Found assembler: C:/altera/24.1std/riscfree/toolchain/riscv32-unknown-elf/bin/riscv32-unknown-elf-gcc.exe
-- Configuring done (4.0s)
-- Generating done (0.1s)
-- Build files have been written to: C:/altera/24.1std/quartus/qdesigns/misProyectos/DE0-Nano_Basic_Computer_NiosVm_conSDRAM_dualCore_Q24/verilog/software/dualCoreNvm_app0_Q24/build
```

Figure 8: Result of the cmake command to prepare Makefile compilation files.

```
[ 1%] Building C object dualCoreNvm_bsp0_Q24/CMakeFiles/hal2_bsp.dir/drivers/src/altera_avalon_mutex.c.obj
[ 2%] Building C object dualCoreNvm_bsp0_Q24/CMakeFiles/hal2_bsp.dir/drivers/src/altera_avalon_sysid_qsys.c.obj
[ 3%] Building C object dualCoreNvm_bsp0_Q24/CMakeFiles/hal2_bsp.dir/drivers/src/altera_avalon_timer_sc.c.obj
[ 4%] Building C object dualCoreNvm_bsp0_Q24/CMakeFiles/hal2_bsp.dir/drivers/src/altera_avalon_timer_ts.c.obj
[ 5%] Building C object dualCoreNvm_bsp0_Q24/CMakeFiles/hal2_bsp.dir/drivers/src/altera_avalon_timer_vars.c.obj
[ 6%] Building C object dualCoreNvm_bsp0_Q24/CMakeFiles/hal2_bsp.dir/drivers/src/altera_up_avalon_parallel_port.c.obj
[ 7%] Linking C static library libhal2_bsp.a
[ 93%] Built target hal2_bsp
[ 94%] Building C object CMakeFiles/dualCoreNvm_app0_Q24.elf.dir/dualCoreNvm_app0_Q24.c.obj
[ 95%] Linking C executable dualCoreNvm_app0_Q24.elf
[ 95%] Built target dualCoreNvm_app0_Q24.elf
[ 96%] Creating dualCoreNvm_app0_Q24.elf.objdump.
[ 96%] Built target create-objdump
[ 97%] Reporting memory available for stack + heap in dualCoreNvm_app0_Q24.elf.
dualCoreNvm_app0_Q24.elf
* 35.47 KB - Program size (code + initialized data).
* 4054.34 KB - Free for stack + heap.
[ 97%] Built target niosv-stack-report
[ 98%] Creating new_sdram_controller_0.hex.
[ 99%] Creating message_buffer_ram.hex.
[ 99%] Creating Onchip_memory.hex.
[100%] Creating Onchip_memory_SRAM.hex.
[100%] Built target create-hex
```

Figure 9: Result of the make -C build command to generate ELF file.

#### 4. DE0-Nano board configuration :

##### Board connection test

Connect the board to the USB connector.

\$ jtagconfig.exe

Output: The message shown on the display should be similar to those seen in Figure 10.

```
1) USB-Blaster [USB-0]
   020F30DD 10CL025(Y|Z)/EP3C25/EP4CE22
```

Figure 10: Result of the jtagconfig.exe command to test board connection to computer.

##### DE0-Nano board configuration

\$ quartus\_pgm -c 1 -m JTAG -o "p;DE0\_Nano\_DualCoreNiosVm.sof@1"

Output: Some messages are shown on the display as can be seen in Figure 11.

```

Info: *****
Info: Running Quartus Prime Programmer
Info: Version 24.1std.0 Build 1077 03/04/2025 SC Standard Edition
Info: Copyright (C) 2025 Altera Corporation. All rights reserved.
Info: Your use of Altera Corporation's design tools, logic functions
Info: and other software and tools, and any partner logic
Info: functions, and any output files from any of the foregoing
Info: (including device programming or simulation files), and any
Info: associated documentation or information are expressly subject
Info: to the terms and conditions of the Altera Program License
Info: Subscription Agreement, the Altera Quartus Prime License Agreement,
Info: the Altera IP License Agreement, or other applicable license
Info: agreement, including, without limitation, that your use is for
Info: the sole purpose of programming logic devices manufactured by
Info: Altera and sold by Altera or its authorized distributors. Please
Info: refer to the Altera Software License Subscription Agreements
Info: on the Quartus Prime software download page.
Info: Processing started: Wed Aug 13 09:47:47 2025
Info: Command: quartus_pgm -c 1 -m JTAG -o p;../DE0_Nano_Basic_Computer.sof@1
Info (213045): Using programming cable "USB-Blaster [USB-0]"
Info (213011): Using programming file ../DE0_Nano_Basic_Computer.sof with checksum 0x00A06297 for device EP4CE22F17@1
Info (209060): Started Programmer operation at Wed Aug 13 09:47:48 2025
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 Wed Aug 13 09:47:49 2025
Info: Quartus Prime Programmer was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 4446 megabytes
Info: Processing ended: Wed Aug 13 09:47:49 2025
Info: Elapsed time: 00:00:02
Info: Total CPU time (on all processors): 00:00:00

```

Figure 11: Result of the `quartus_pgm` command to configure the FPGA of the DE0-Nano board.

##### 5. Program execution (see Figure 12):

###### Program execution

```
$ niosv-download.exe -i 0 -g build/dualCoreNVm_app0_Q24.elf
```

Output: Some messages are shown on the display as can be seen in Figure 12.

```

INFO: Starting GDB server. Running "ash-riscv-gdb-server --auto-detect true --probe-type usb-blaster-2 --instance 1 --device 020F30DD --core-number 0".
[GDB server output] Ashling GDB Server for RISC-V (ash-riscv-gdb-server).
[GDB server output] v25.1.1, 31-Jan-2025, (c)Ashling Microsystems Ltd 2024.
[GDB server output]
[GDB server output] Initializing connection ...
[GDB server output] Failed to get JTAG frequency from the debug probe
[GDB server output] Connected to target device with IDCODE 0x20F30dd using USB-Blaster-2 (1) via JTAG at 0.00MHz.
[GDB server output] Info : Active Harts Detected : 1
[GDB server output] Info : Core[0] Hart[0] is in halted state
[GDB server output] Info : [0] System architecture : RV32
[GDB server output] Info : [0] Debug version : v1.00
[GDB server output] Info : [0] Number of hardware breakpoints available : 1
[GDB server output] Info : [0] Number of program buffers: 8
[GDB server output] Info : [0] Number of data registers: 2
[GDB server output] Info : [0] Memory access -> Program buffer
[GDB server output] Info : [0] Memory access -> Abstract access memory
[GDB server output] Info : [0] CSR & FP Register access -> Abstract commands
[GDB server output]
[GDB server output] Waiting for debugger connection on port 55958.
INFO: Found gdb port 55958
INFO: Starting gdb. Running "riscv32-unknown-elf-gdb -batch -ex set arch riscv:rv32 -ex set remotetarget mout 60 -ex target extended-remote localhost:55958 -ex load build/dualCoreNVm_app0_Q24.elf -ex set $mstatus &= ~(0x00000000) -ex continue".
The target architecture is set to "riscv:rv32".
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.

Program received signal SIGINT, Interrupt.
0x00000004 in ?? ()
Loading section .entry, size 0x20 lma 0x0
Loading section .exceptions, size 0x2f4 lma 0x20
Loading section .text, size 0x6cf0 lma 0x314
Loading section .rodata, size 0x140 lma 0x7004
Loading section .rdata, size 0x18c0 lma 0x8a04
Start address 0x00000348, load size 35332
Transfer rate: 44 KB/sec, 7066 bytes/write.
[Inferior 1 (Remote target) detached]
C:/altera/24.1std/quartus/qdesigns/misProyectos/DE0-Nano_Basic_Computer_NiosVm_conSDRAM_dualCore_Q24/verilog/software/dualCoreNVm_app0_Q24 #

```

Figure 12: Result of the `niosv-download` command to load the executable program into the SDRAM memory of the DE0-Nano board.

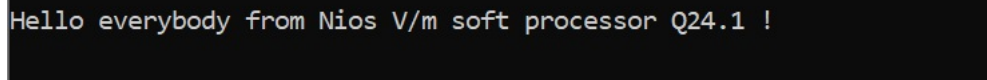
##### 6. Display output message on the screen (see Figure 14):

###### View results

```
$ juart-terminal.exe
```

Output: the message 'Hello everybody from Nios V/m soft processor Q24.1 !' should appear as can be seen in Figure 14.





```
Hello everybody from Nios V/m soft processor Q24.1 !
```

Figure 13: Display of Tutorial-1 messages after running the `juart-terminal.exe` command in a Nios V Command Shell window.

### Files for Tutorial-1:

- C source code: `dualCoreNVm_app0_Q24.c` (folder: Tutorial-1).
- FPGA configuration files: `DE0_Nano_DualCoreNiosVm.sopcinfo`, `DE0_Nano_DualCoreNiosVm.sof` (folder: `DualCoreNiosVm`).
- C source file to be copied in folders `dualCoreNVm_bsp0_Q24/drivers/src` and `dualCoreNVm_bsp1_Q24/drivers/src`: `altera_avalon_mutex.c`

**Objective 2-2:** Follow the steps described below to run **Tutorial-2** using two processor cores (`intel_niosv_m_0` and `intel_niosv_m_1`). Two BSP projects are created following the steps explained in Tutorial-1.

1. Common elements for both BSP projects:

- SOPC Information File name: `DE0_Nano_DualCoreNiosVm.sopcinfo`

2. BSP Projects 1 and 2:

- Reuse BSP files saved in folders: `dualCoreNVm_bsp0_Q24` for `intel_niosv_m_0` core and `dualCoreNVm_bsp1_Q24` for `intel_niosv_m_1` core.

3. C program for the first Nios V core:

- Folder name: `dualCoreNVm_app1_semaforo_0`
- Copy the `dualCoreNVm_app1_semaforo_0_Q24.c` file in the directory. Note that `intel_niosv_m_0` only reads the `message_buffer_val` variable. The display of results obtained by `intel_niosv_m_0` is done using the `printf()` function. Only `intel_niosv_m_0` is connected to the JTAG interface so its program is the only one that can include `printf`.
- Compile and link

build folder contains ELF file for for the first Tutorial-2 program

```
$ mkdir dualCoreNVm_app1_semaforo_0_Q24
$ cd dualCoreNVm_app1_semaforo_0_Q24
$ cp dualCoreNVm_app1_semaforo_0_Q24.c .
$ niosv-app.exe -a=. -b=../dualCoreNVm_bsp0_Q24 -s=.
Output: Info messages will be shown on the display (see Figure 7).
$ cmake -S . -G "Unix Makefiles" -B build
Output: Info messages will be shown on the display (see Figure 8).
$ make -C build
Output: the build folder and the dualCoreNVm_app1_semaforo_0_Q24.elf file
are created (see Figure 9).
```

4. C program for the second Nios V core:

- Folder name: `dualCoreNVm_app1_semaforo_1`

- Copy the dualCoreNvm\_app1\_semaforo\_1\_Q24.c file in the directory.
- Compile and link

build folder contains ELF file for for the first Tutorial-2 program

```
$ mkdir dualCoreNvm_app1_semaforo_1_Q24
$ cd dualCoreNvm_app1_semaforo_1_Q24
$ cp dualCoreNvm_app1_semaforo_1_Q24.c .
$ niosv-app.exe -a=. -b=../dualCoreNvm_bsp1_Q24 -s=.
Output: Info messages will be shown on the display (see Figure 7).
$ cmake -S . -G "Unix Makefiles" -B build
Output: Info messages will be shown on the display (see Figure 8).
$ make -C build
Output: the build folder and the dualCoreNvm_app1_semaforo_1_Q24.elf file
are created (see Figure 9).
```

##### 5. DE0-Nano board configuration and program execution:

Board connection, configuration, program download and execution

```
Connect the board to the USB connector.
$ jtagconfig.exe
Output: The message shown on the display should be similar to those seen in
Figure 10. $ quartus_pgm -c 1 -m JTAG -o "p;DE0_Nano_DualCoreNiosVm.sof@1"
Output: Some messages are shown on the display as can be seen in Figure 11.
Download first program for intel_niosv_m_0 core.
$ niosv-download.exe -i 0 -g build/dualCoreNvm_app1_semaforo_0_Q24.elf
Output: Some messages are shown on the display as can be seen in Figure 12.
Download second program for intel_niosv_m_1 core.
$ niosv-download.exe -i 1 -g build/dualCoreNvm_app1_semaforo_1_Q24.elf
Output: Some messages are shown on the display as can be seen in Figure 12.
$ juart-terminal.exe
Output: the messages shown in Figure 14 should appear on display.
$ CTRL-C
```

```
Hello, I am Semaforo_0
CPU - iteration: 1 - message_buffer_val: 00000000
CPU - iteration: 2 - message_buffer_val: 00000000
CPU - iteration: 3 - message_buffer_val: 0000274F
CPU - iteration: 4 - message_buffer_val: 00011BDF
CPU - iteration: 5 - message_buffer_val: 00021086
CPU - iteration: 6 - message_buffer_val: 00030530
CPU - iteration: 7 - message_buffer_val: 0003F9DA
CPU - iteration: 8 - message_buffer_val: 0004EE85
```

Figure 14: Display of Tutorial-2 messages after running the juart-terminal.exe command in a Nios V Command Shell window.

##### Files for Tutorial-2:

- C source code: dualCoreNvm\_app1\_semaforo\_0\_Q24.c, dualCoreNvm\_app1\_semaforo\_1\_Q24.c (folders: dualCoreNvm\_app1\_semaforo\_0\_Q24, dualCoreNvm\_app1\_semaforo\_1\_Q24).
- FPGA configuration files: DE0\_Nano\_DualCoreNiosVm.sopcinfo, DE0\_Nano\_DualCoreNiosVm.sof (folder: DualCoreNiosVm).

### Part 3. Multithreaded parallel programming and performance evaluation of Nios V 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 V/m and Nios V/g, respectively. Finally, an exercise is proposed where the matrix multiplication algorithm is implemented.

The C code for 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 elements ( $x[j]$ ). The result is a vector with 16 elements ( $y[i]$ ):  $y[i] = A[i \times m + j] \times x[j]$ . Matrix and vectors components are integer values. The mathematical operation is:  $y = A \times x$  (see Figure 15).

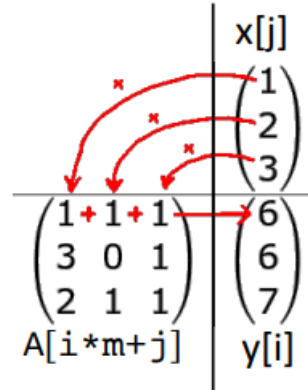


Figure 15: Matrix  $\times$  Vector algorithm:  $y[i] = \sum_j A[i \times m + j] \times x[j]$ .

The main loop of the unique C source file for Tutorial-3 (`MV_serie_2025.c`) is shown in Figure 16.

```
// Shared memory for A matrix and x, y vectors
volatile int * A = (int *) 0x100000; // 16x16x4=1KiB: 0x100000 - 0x1003FF
volatile int * x = (int *) 0x100400; // 16x1 x4=64 B: 0x100400 - 0x10043F
volatile int * y = (int *) 0x100800; // 16x1 x4=64 B: 0x100800 - 0x10083F

int main()
{
    unsigned int k1;
    unsigned int local_n = n;
    unsigned int my_first_row = 0; // first row assigned to this core
    unsigned int my_last_row = local_n - 1; // last row assigned to this core

    // Matrix-Vector operations
    for (k1 = 0; k1 < Niter; k1++) {
        iteraciones++;
        for (i=my_first_row; i<=my_last_row; i++){
            dummy = y[i];
            for(j=0; j<m; j++){
                dummy += A[i*m+j] * x[j];
            }
            y[i] = dummy;
        }
    }
}
```

Figure 16: `MV_serie_2025.c`

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

$N_{iter}$	<b>add</b> $N_{operations}$	<b>mult</b> $N_{operations}$	$N_{loads}$	$N_{stores}$
1000				
2000				
5000				
10000				

Table 2: Measurements of execution time of the Matrix  $\times$  Vector sequential algorithm for one of the processors (intel\_niosv\_m.0) integrated into a Nios V/m dual-core multiprocessor. The measurements for the Nios V/g processor are also included (intel\_niosv\_g.0).

FPGA configuration	$N_{iter}$	Time ( $ms$ )	Speed-up
DualCoreNiosVm	1000		1
DualCoreNiosVm	2000		1
DualCoreNiosVm	5000		1
DualCoreNiosVm	10000		1
DualCoreNiosVg	1000		
DualCoreNiosVg	2000		
DualCoreNiosVg	5000		
DualCoreNiosVg	10000		

In this tutorial, execution time will be measured using the Timer and its HAL (Hardware Abstraction Layer) function/driver is called `alt_timestamp()`.

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_series` project directory

As in Tutorial 2, display of results can only be done through the `intel_niosv_m.0` core using the `printf()` function.

**Objective 3-1:** Follow the steps described in Part 2 for Tutorial 1 using `MV_serier.c` so that the `.elf` file is generated. Next, configure the DE0-Nano board using the FPGA configuration: `DE0_Nano_DualCoreNiosVm`, and run the sequential program on the `intel_niosv_m.0` core. Next, do the following exercises:

- Register the execution times for four workloads.
  - Parameters: `Niter` = 1000, 2000, 5000, 10000.
- Performance evaluation: fill in Tables 1 and 2, whose columns represent the following items:
  - add**  $N_{operations}$ : number of add operations in all Niter iterations
  - mult**  $N_{operations}$ : number of mult operations in all Niter iterations
  - $N_{loads}$ : number of load instructions
  - $N_{stores}$ : number of store instructions
  - Time: execution time of the entire program
- Repeat the results for Nios V/g using the processor core called `intel_niosv_g.0` that is integrated in the FPGA configuration: `DE0_Nano_DualCoreNiosVg.sof`. For the BSP projects of the Nios V/g multiprocessor, the following file is needed: `DE0_Nano_DualCoreNiosVm.sopcinfo`.

## Questions to justify the results of this Objective 3-1

1. Is it reasonable that doubling the number of arithmetic operations and memory accesses would cause the execution time of one of the Nios V/m processors in the DualCoreNiosVm multiprocessor to be doubled? Why?
2. Is it reasonable that the execution times provided by Nios V/g are significantly lower than those obtained when Nios V/m is used? Why?
3. What is the average speed-up provided by the Nios V/g core relative to the Nios V/m core?

**Objective 3-2:** Follow the steps done in Tutorial-2 using `MV_paralelo_maestro_2025.c` (see Figure 17) and `MV_paralelo_escravo_2025.c` (see Figure 18) for generating two C software projects, one for each Nios V/m core of the *DualCoreNiosVm* multiprocessor: `intel_niosv_m_0` and `intel_niosv_m_1`. Next, configure the DE0-Nano board employing the FPGA configuration: `DE0_Nano_DualCoreNiosVm.sof`. Finally, run the parallel program using `intel_niosv_m_0` and `intel_niosv_m_1` cores.

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 again the BSP projects and the two parallel programs.

Next, the following activities should be completed:

1. Configure the FPGA using `DE0_Nano_DualCoreNiosVm.sof` file and `quartus_pgm` command (dual core:  $2 \times$  Nios V/m).
2. Download the program using `nios5-download` command.
3. Register the execution times activating a single thread and using `intel_niosv_m_0` processor for four 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 `intel_niosv_m_0` and `intel_niosv_m_1` cores for four workloads. Parameters:
  - `thread_count = 2`
  - `Niter = 1000, 2000, 5000, 10000`.
5. Performance evaluation: fill in Table 3 using the execution times obtained in previous steps 3 and 4. Additionally, calculate speed-up considering that the reference configuration is a single Nios V/m core.
6. Repeat the results using *DualCoreNiosVg* multiprocessor (dual core:  $2 \times$  Nios V/g). For the performance analysis, consider the reference configuration to be a single Nios V/g processor.

## Questions to justify the results of this Objective 3-2

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

**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 are described as follows:

1. Code the matrix multiplication algorithm using input matrices with  $n$  rows  $\times$   $n$  columns and 4-byte integer data. The input matrices are called *A* and *B*, and the output matrix is called *C*:  $C = B \times A$  (see Figure 19). 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:



```

main() {
    // driver for mutex controller
    alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");
    int rank      = 0;    // master thread
    int thread_count = 2;  // number of threads: 1,2
    int Niter      = 1000; // options: 1000,2000,10000; repetitions of algorithm
    int message_buffer_val      = 0x0; // local variable
    int message_buffer_val_fork = 0x0; // local variable
    int message_buffer_val_join = 0x0; // local variable
    // Fork for master thread
    message_buffer_val = 15;          // Fork starts
    message_buffer_val_fork = 1;      // master thread is ready
    altera_avalon_mutex_lock(mutex,1); // lock mutex
    *(message_buffer_ptr) = message_buffer_val; // pointer initialization
    *(message_buffer_ptr_fork) = message_buffer_val_fork; // pointer initialization
    *(message_buffer_threads) = thread_count; // pointer initialization
    *(message_buffer_Niter) = Niter; // pointer initialization
    altera_avalon_mutex_unlock(mutex); // free mutex
    while(message_buffer_val != 5) {
        altera_avalon_mutex_lock(mutex,1); // lock mutex
        message_buffer_val_fork = *(message_buffer_ptr_fork); // read from synchronizing pointer
        altera_avalon_mutex_unlock(mutex); // free mutex
        if ( (message_buffer_val_fork == 0x3 && thread_count == 2 ) ||
            (message_buffer_val_fork == 0x1 && thread_count == 1) ){
            dummy = 5;
            altera_avalon_mutex_lock(mutex,1); // lock mutex
            *(message_buffer_ptr) = dummy; // write buffer
            *(message_buffer_ptr_join) = 0; // write buffer
            altera_avalon_mutex_unlock(mutex); // free mutex
            message_buffer_val = dummy;
        }
    }
    // Computation
    int i, j, k1, iteraciones = 0, dummy;
    int local_n = n / thread_count;
    int my_first_row = rank * local_n; // first row assigned to this thread
    int my_last_row = (rank+1) * local_n - 1; // last row assigned to this thread
    for (k1 = 0; k1 < Niter; k1++) {
        iteraciones++;
        for (i=my_first_row; i<=my_last_row; i++){
            dummy = y[i];
            for(j=0; j<m; j++) dummy = dummy + A[i*m+j] * x[j];
            y[i] = dummy;
        }
    }
    // Thread join
    message_buffer_val_join = 1; // master thread has finished its work
    altera_avalon_mutex_lock(mutex,1); // lock mutex
    *(message_buffer_ptr_join) |= message_buffer_val_join; // pointer is updated by master thread
    altera_avalon_mutex_unlock(mutex); // free mutex
    while(message_buffer_val != 6 && thread_count == 2) {
        altera_avalon_mutex_lock(mutex,1); // lock mutex
        message_buffer_val = *(message_buffer_ptr); // read from synchronizing pointer
        message_buffer_val_join = *(message_buffer_ptr_join); // read from synchronizing pointer
        altera_avalon_mutex_unlock(mutex); // free 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); // lock mutex
            *(message_buffer_ptr) = dummy; // both threads are synchronized at JOIN
            altera_avalon_mutex_unlock(mutex); // free mutex
            message_buffer_val = dummy; // variable is updated
        }
    }
}

```

Figure 17: MV.paralelo.maestro.2025.c

```

int main()
{
    int rank = 1;          // slave thread
    // driver for mutex controller
    alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");

    int message_buffer_val    = 0x0; // local variable
    int message_buffer_val_fork = 0x0; // local variable
    int message_buffer_val_join = 0x0; // local variable
    int thread_count          = 0; // local variable
    int Niter                 = 0; // local variable, number of repetitions of the algorithm
    *(message_buffer_ptr_fork) |= 0; // pointer initialization

    // Fork for slave thread
    while(message_buffer_val != 5) {
        altera_avalon_mutex_lock(mutex,2); // lock mutex

        message_buffer_val = *(message_buffer_ptr); // read from synchronizing pointer
        thread_count       = *(message_buffer_threads); // read number of active threads
        Niter              = *(message_buffer_Niter); // read number of repetitions

        if(message_buffer_val == 15 && thread_count == 2) {
            message_buffer_val_fork = *(message_buffer_ptr_fork); // read from shared synchronizing pointer
            message_buffer_val_fork |= 2; // slave thread is ready
            *(message_buffer_ptr_fork) = message_buffer_val_fork; // pointer is updated
        }
        altera_avalon_mutex_unlock(mutex); // free mutex
    }

    // Computation
    int i, j, k1, iteraciones = 0, dummy;
    int local_n = n / thread_count;
    int my_first_row = rank * local_n; // first row assigned to this thread
    int my_last_row = (rank+1) * local_n - 1; // last row assigned to this thread

    for (k1 = 0; k1 < Niter; k1++) {
        iteraciones++;
        for (i=my_first_row; i<=my_last_row; i++){
            dummy = y[i];
            for(j=0; j<m; j++){
                dummy = dummy + A[i*m+j] * x[j];
            }
            y[i] = dummy;
        }
    }

    // Thread join
    while(message_buffer_val != 6 && thread_count == 2) {
        altera_avalon_mutex_lock(mutex,2); // lock mutex

        message_buffer_val = *(message_buffer_ptr); // read from synchronizing pointer
        message_buffer_val_join = *(message_buffer_ptr_join); // read from shared synchronizing pointer
        message_buffer_val_join |= 2; // slave thread has finished its work
        *(message_buffer_ptr_join) = message_buffer_val_join; // pointer is updated
        altera_avalon_mutex_unlock(mutex); // free mutex
    }
    return 0;
}

```

Figure 18: MV.paralelo\_esclavo\_2025.c

Table 3: Performance evaluation of the Matrix  $\times$  Vector algorithm for 1 and 2 threads using `intel_niosv_m_0`, `intel_niosv_m_1` and `intel_niosv_g_0`, `intel_niosv_g_1` processors of the two Nios V/{m,g} multiprocessors. Legend:  $N_{iter}$  is the number of repetitions of the Matrix  $\times$  Vector algorithm.

FPGA configuration	Number of threads	$N_{iter}$	Time (ms)	Speed-up	Parallelism efficiency
DE0_Nano_DualCoreNiosVm.sof	1	1000		1	100 %
DE0_Nano_DualCoreNiosVm.sof	1	2000		1	100 %
DE0_Nano_DualCoreNiosVm.sof	1	5000		1	100 %
DE0_Nano_DualCoreNiosVm.sof	1	10000		1	100 %
DE0_Nano_DualCoreNiosVm.sof	2	1000			
DE0_Nano_DualCoreNiosVm.sof	2	2000			
DE0_Nano_DualCoreNiosVm.sof	2	5000			
DE0_Nano_DualCoreNiosVm.sof	2	10000			
DE0_Nano_DualCoreNiosVg.sof	1	1000		1	100 %
DE0_Nano_DualCoreNiosVg.sof	1	2000		1	100 %
DE0_Nano_DualCoreNiosVg.sof	1	5000		1	100 %
DE0_Nano_DualCoreNiosVg.sof	1	10000		1	100 %
DE0_Nano_DualCoreNiosVg.sof	2	1000			
DE0_Nano_DualCoreNiosVg.sof	2	2000			
DE0_Nano_DualCoreNiosVg.sof	2	5000			
DE0_Nano_DualCoreNiosVg.sof	2	10000			

```

void Matrix-Matrix (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 19: Procedure for matrix multiplication:  $C = B \times A$ .

- 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 DualCoreNiosVm (2 cores:  $2 \times$  Nios V/m) and DualCoreNiosVg (2 cores:  $2 \times$  Nios V/g) 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 of Objective 3-3**

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 for Objective 3-3**

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, a table with results of the performance evaluation and some evidences of the program execution, for example, screenshots obtained of display messages.

**Files for Part 3:**

- C source code: MV\_paralelo\_maestro.2025.c, MV\_paralelo\_esclavo.2025.c (folder: Tutorial3).
- FPGA files for Nios V/m: DE0\_Nano\_DualCoreNiosVm.sopcinfo, DE0\_Nano\_DualCoreNiosVm.sof.
- FPGA files for Nios V/g: DE0\_Nano\_DualCoreNiosVg.sopcinfo, DE0\_Nano\_DualCoreNiosVg.sof.

**References**

- [1] 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](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tt/tt_nios2_multiprocessor_tutorial.pdf), 2011.
- [2] 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](ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/12.0/Computer_Systems/DE0/DE0_Basic_Computer.pdf), 2013. Accessed: 2022-01-19.
- [3] Intel. Nios V Processor Reference Manual. Updated for Intel Quartus Prime Design Suite: 23.1. <https://cdrdv2-public.intel.com/776470/ug-683632-776470.pdf>, 2023.
- [4] Intel. Nios V Processor Software Developer Handbook. Updated for Intel Quartus Prime Design Suite: 23.1. <https://cdrdv2-public.intel.com/774362/ug-743810-774362.pdf>, 2023.
- [5] Terasic. DE0-Nano User Manual. <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English\&No=593\&PartNo=4>, 2021.