Lab assignment 4:

Nios V multiprocessor implementation, parallel programming, and performance evaluation

Computer Architecture (40969)
School of Computer Science (EII)
University of Las Palmas de Gran Canaria

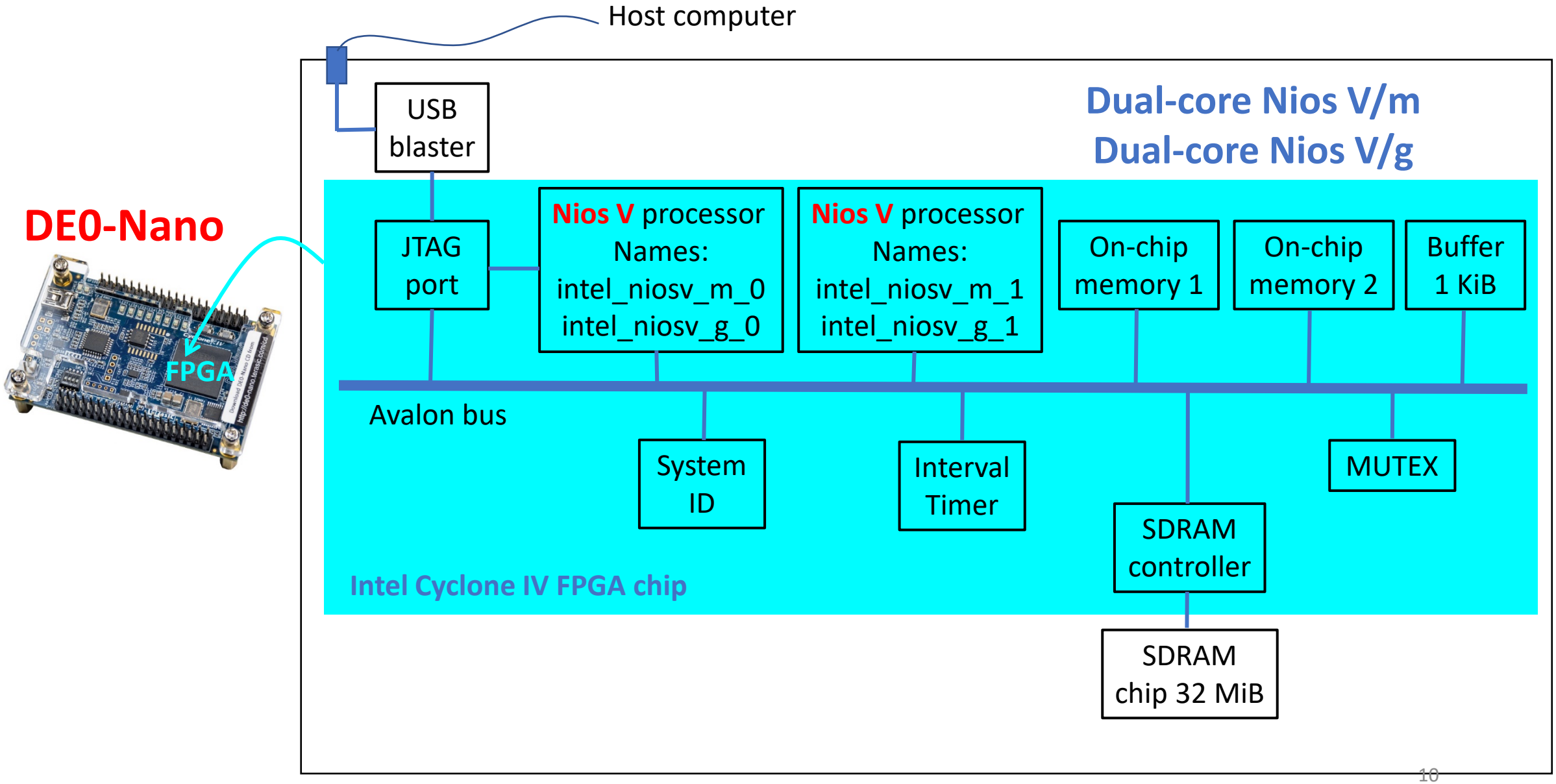
Main goals

- Implement parallel programs on two Nios V multiprocessors with the goal of significantly reducing the execution time of the programs.
- Measure and compare the execution times of the parallel programs with the corresponding sequential versions using the DE0-Nano board.
- Evaluate the performance of the Nios V multiprocessors for different amounts of processed data.
- Compare the performance of the various Nios V multiprocessors.
- Implement the parallel programs on the DE0-Nano board.
- Keywords: multithreaded programming, parallelism, multiprocessors, thread synchronization, performance evaluation, Nios V, DE0-Nano.

Scheduling: 3 sessions, 1 session/week

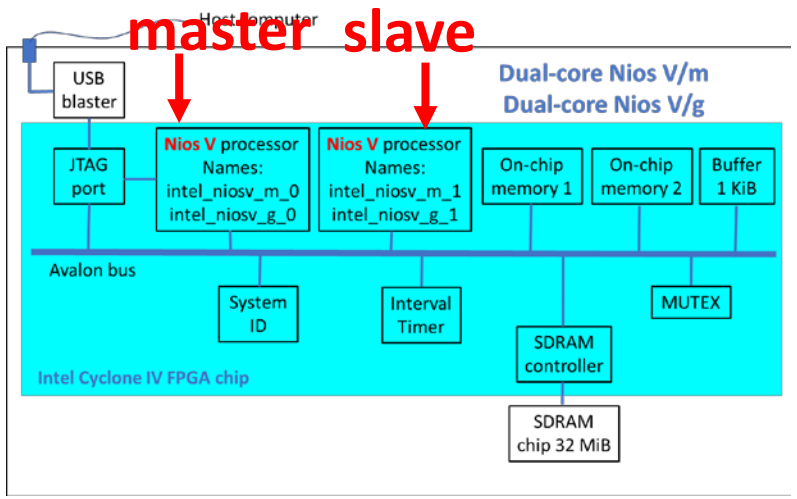
- Session 1:
  - Microarchitecture of Nios V-based parallel SoC computer + memory hierarchy + software
- tools + Nios V Command Shell.
  - Tutorial-1: `hi_guys` (0.5 h).
  - Tutorial-2: `hi_mutex` (0.5 h).
- Session 2:
  - Tutorial-3: Matrix × Vector. • Proposal: Matrix × Matrix.
- Session 3:
  - Implementation of Matrix × Matrix algorithm using dual-core Nios V/m and Nios V/g multiprocessors

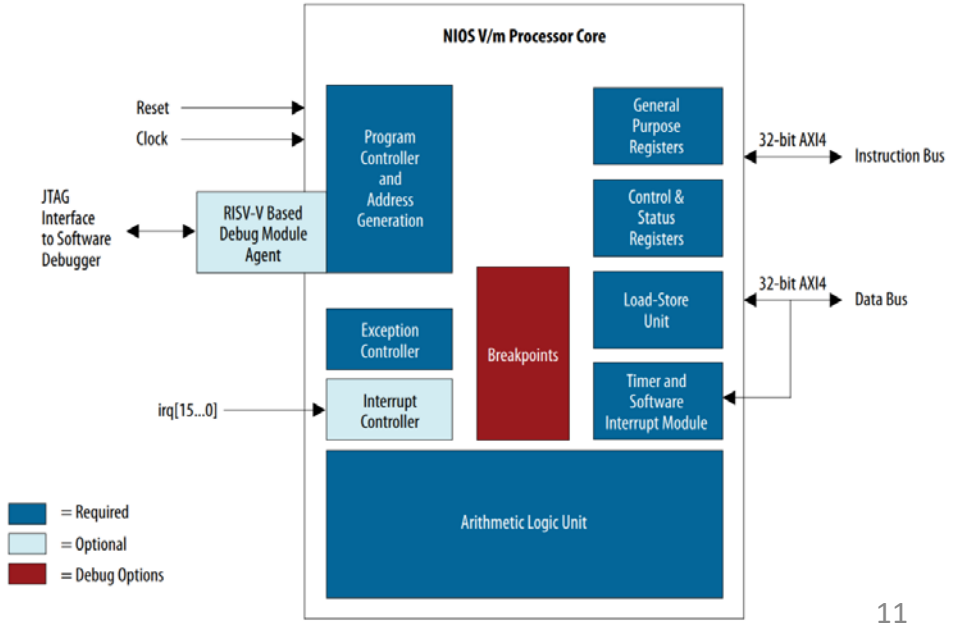
Soft SoC based on Nios V dual-core multiprocessors



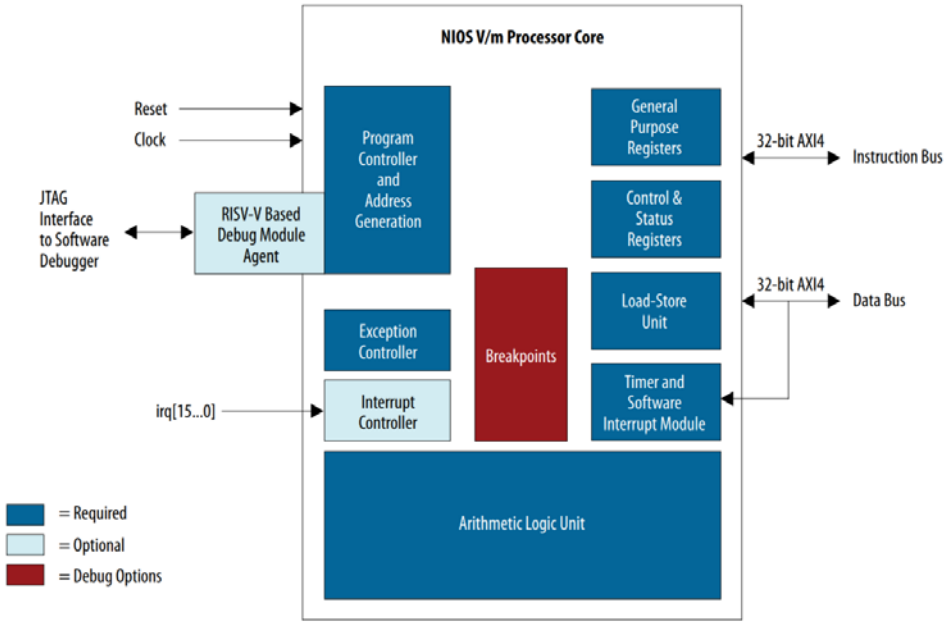
Implementation of Nios V dual-core multiprocessors on DE0-Nano board

- FPGA desing: Intel Quartus Prime Standard Edition Design Suite 24.1
- Two different Nios V dual cores (cores: **master** & **slave**)
  - 2 x Nios V/m (DualCoreNiosVm): pipelined, without cache
  - 2 x Nios V/g (DualCoreNiosVg): pipelined, iCache (4 KiB) + dCache (4 KiB) + hardware multiplier



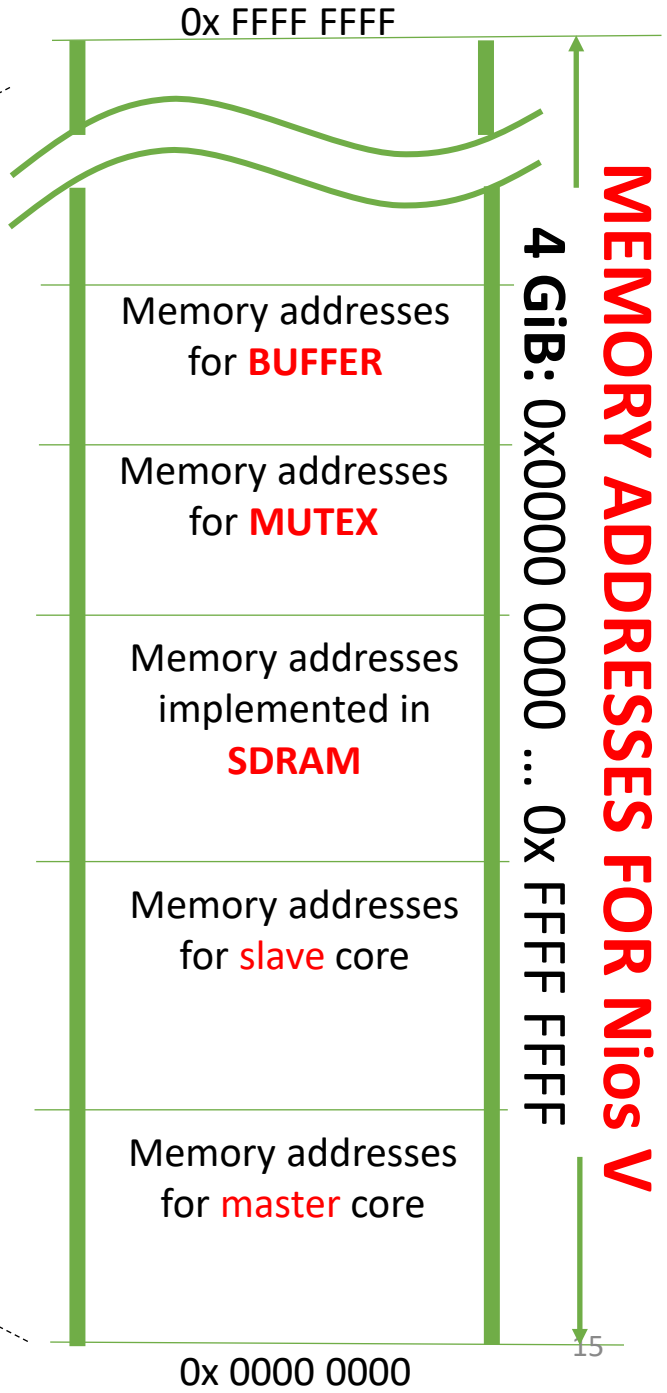


Memory addresses for Nios V cores



32 bits architecture

32 bits address

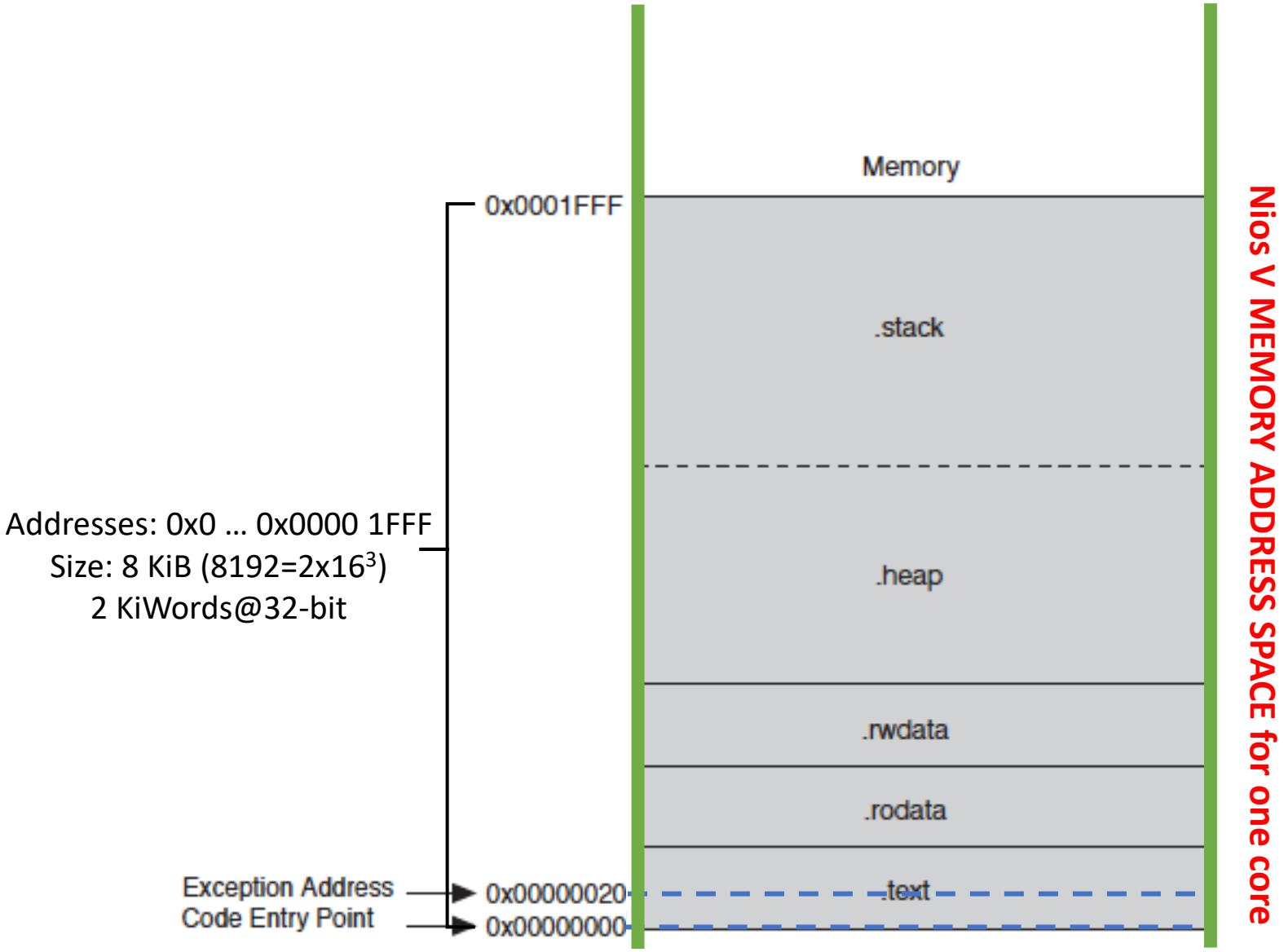


Map of memory addresses

- Interval Timer.s1 — 0x 1000 2000 - 0x 1000 201F (32 bytes)
- on-chipMEM2.s1 — 0x 0900 0000 - 0x 0900 1FFF (8 KiB)
- on-chipMEM_SRAM.s1 — 0x 0800 0000 - 0x 0800 1FFF (8 KiB)
- BUFFER (message_buffer_ram.s1) — MESSAGE_BUFFER_RAM_BASE → 0x 0820 0000 - 0x 0820 03FF (1 KiB)
- MUTEX (message_buffer_mutex.s1) — 0x 0820 0400 - 0x 0820 0407 (8 bytes)
- SDRAM.s1 — 0x 0000 0000 - 0x 01FF FFFF (32 MiB)
- **intel_niosv_m_0** (dentro de SDRAM) — **0x 0000 0000 - 0x 003F FFFF (4 MiB)**
  - resetVector={SDRAM.s1,0x 0000 0000},
  - exceptionVector={SDRAM.s1,0x 0000 0020},
  - mhartid CSR value = 0x0
- **intel_niosv_m_1** (dentro de SDRAM) — **0x 0040 0000 - 0x 007F FFFF (4 MiB)**
  - resetVector={SDRAM.s1,0x 0040 0000},
  - exceptionVector={SDRAM.s1,0x 0040 0020},
  - mhartid CSR value = 0x1

Memory organization for one core

- Code entry point
- Exception address
- Memory limits
- Memory size
- Address range
- Linking zones :
  - instructions, data, stack, heap

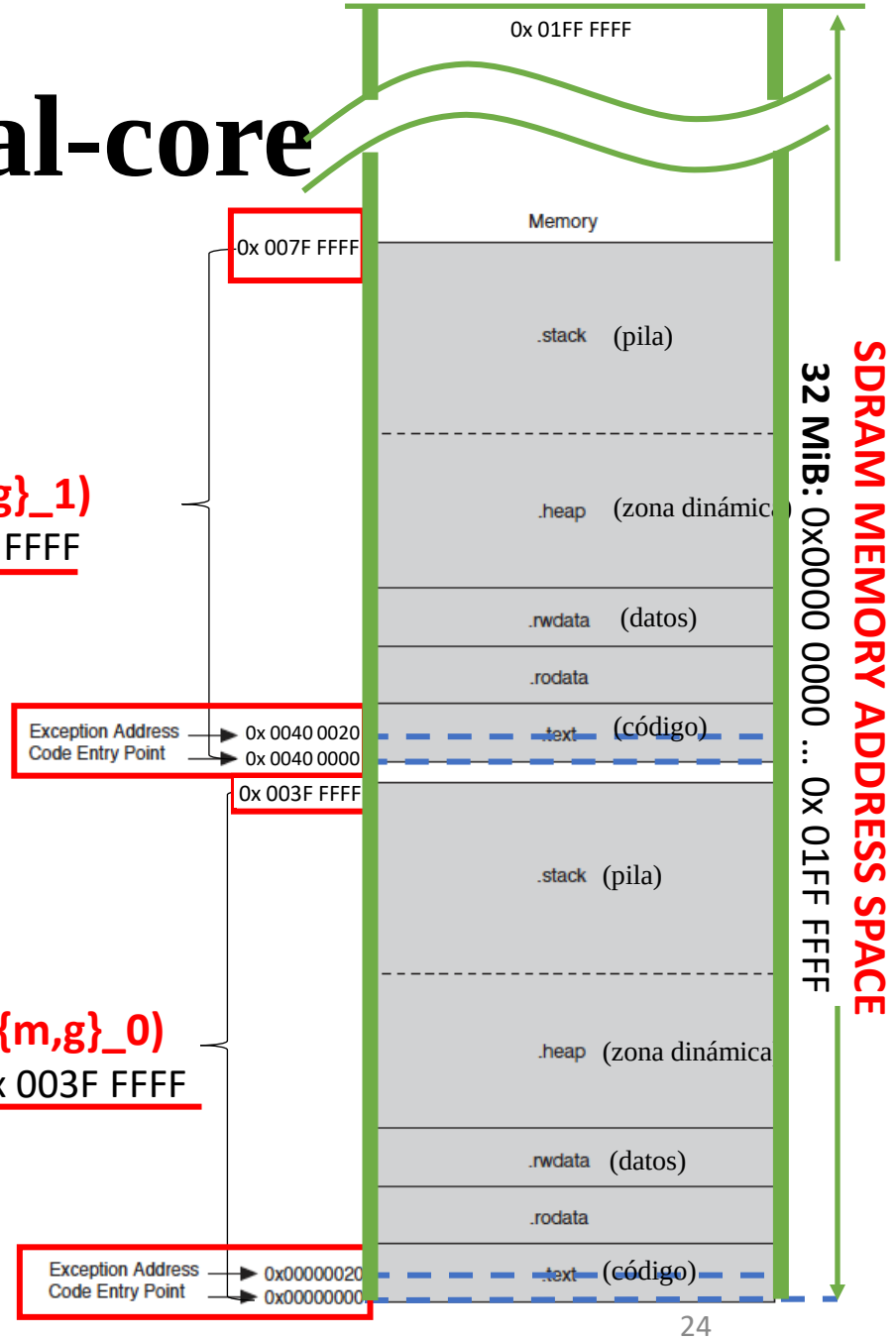


Memory partition for Nios V dual-core

- Code entry points
- Exception addresses
- Memory limits
- Memory size
- Address range
- Linking zones :
  - instructions, data, stack, heap

CORE-2 (intel_niosv_{m,g}_1)
Addresses: 0x 0040 0000 … 0x 007F FFFF
Size: 4 MiB ($4x16^5$)

CORE-1 (intel_niosv_{m,g}_0)
Addresses: 0x0000 0000 … 0x 003F FFFF
Size: 4 MiB ($4x16^5$)



Software projects for Nios V

- Software Build Tools:
  - BSP projects: `niosv-bsp.exe` → `system.h` file is generated
  - Compile and link C projects: `niosv-app.exe`, `cmake`, `make` → `*.elf` files are generated
- Nios V Command Shell, comands:
  - Configure FPGA of DE0-Nano board: $ `quartus pgm <file>.sof` (file DE0-nano= DE0_Nano_DualCore_NiosVm.sof)
  - Download programs in memory and begin execution:
    ```
    $ niosv-download –r –g –i <ID processor> <file>.elf
    ```
  - View messages on the display: `$ juart-terminal.exe`
  - Register measures: pen + paper
- Tutorials for two Nios V dual-core in this lab assigment:
  - Tutorial-1 (sequential): hi_guys
  - Tutorial-2 (parallel): hi_mutex
  - Tutorial-3 (sequential & parallel): Matrix × Vector

Tutorial-1 (one core), hi_guys: `dualCoreNVm_app0_Q24.c`

```
#include <stdio.h>

int main()
{
 printf("Hello everybody from Nios V/m soft processor Q24.1 !\n");
 return 0;
}
```

Tutorial-1 (1 core): hi_guys

Please, follow steps described in Part 2 of the document for Lab Assignment 4.

1. Open: Nios V Command Shell Tools
2. BSP files for both cores of Nios V multiprocessor are created
3. Compile and link Tutorial-1 application whose source file is `dualCoreNVm_app0_Q24.c`
4. DE0-Nano board configuration
5. Program download and execution
6. Display output message on the screen

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

Tutorial-2 (2 cores): multithread programming

- Programming methodology: multithread programming, two programs are developed, one for each core of the dual-core multiprocessor.
- Synchronizing threads use
  - Shared variables stored in the memory address space
  - Mutual exclusion controller called *MUTEX*
- Functional description of parallel program:
  - Thread-1: increases and updates a shared variable like a counter
  - Thread-2: reads the shared variable and shows it on display using `printf`

Tutorial-2: multithread programming

2 cores: intel_niosv_m_{0,1} 2 threads: dualCoreNVm_app1_semaforo_{0,1}.c

intel_niosv_m_0 (core 1)

```
/* Source code for master thread: dualCoreNVm_app1_semaforo_0.c */
#include <stdio.h>
#include <system.h>
#include <altera_avalon_mutex.h>
#include <unistd.h>

int main(){

// address memory for a shared message buffer: 0x 0820 0000
volatile int * message_buffer_ptr = (int *) MESSAGE_BUFFER_RAM_BASE;

printf("Hello, I am Semaforo_0\n");

/* driver for mutex controller */
alt_mutex_dev* mutex = altera_avalon_mutex_open("/dev/message_buffer_mutex");

int message_buffer_val   = 0x0;
int iterations           = 0x0;

while(1) {
  iterations++;
  /* Master core wants to lock the mutex controller, using an ID with value 1 */
  altera_avalon_mutex_lock(mutex,1);
  message_buffer_val = *(message_buffer_ptr); /* read the value from shared buffer */
  altera_avalon_mutex_unlock(mutex); /* free mutex */
  printf("CPU - iter: %i - message_buffer_val: %08X\n", iterations, message_buffer_val);
  usleep(4000000); /* wait 4 seg = 4000000 useg = 4 10^6 useg */
  }
return 0;
}
```

intel_niosv_m_0 READ shared variable message_buffer_val

intel_niosv_m_1 (core 2)

```
/* Source code for slave thread: dualCoreNVm_app1_semaforo_1.c */
#include <stdio.h>
#include <system.h>
#include <altera_avalon_mutex.h>

int main(){

// address memory for a shared message buffer: 0x 0820 0000
volatile int * message_buffer_ptr  = (int *)
                                     MESSAGE_BUFFER_RAM_BASE;

/* driver for mutex controller */
alt_mutex_dev* mutex =
altera_avalon_mutex_open("/dev/message_buffer_mutex");

int message_buffer_val   = 0x0;

while(1) {
  /* Slave core wants to lock the mutex controller, using an ID 2 */
  altera_avalon_mutex_lock(mutex,2);
   /* save message_buffer_val variable in the message buffer */
  *(message_buffer_ptr) = message_buffer_val;
  altera_avalon_mutex_unlock(mutex); /* free mutex */
  /* shared variable message_buffer_val is increased */
  message_buffer_val++;
  }
return 0;
}
```

Only intel_niosv_m_1 UPDATES message_buffer_val

Map of memory addresses

- Interval Timer.s1 — 0x 1000 2000 - 0x 1000 201F (32 bytes)
- on-chipMEM2.s1 — 0x 0900 0000 - 0x 0900 1FFF (8 KiB)
- on-chipMEM_SRAM.s1 — 0x 0800 0000 - 0x 0800 1FFF (8 KiB)
- BUFFER (message_buffer_ram.s1) — 0x 0820 0000 - 0x 0820 03FF (1 KiB)
- MUTEX (message_buffer_mutex.s1) — 0x 0820 0400 - 0x 0820 0407 (8 bytes)
- SDRAM.s1 — 0x 0000 0000 - 0x 01FF FFFF (32 MiB)
- **intel_niosv_m_0** (dentro de SDRAM) — **0x 0000 0000 - 0x 003F FFFF (4 MiB)**
  - resetVector={SDRAM.s1,0x 0000 0000},
  - exceptionVector={SDRAM.s1,0x 0000 0020},
  - mhartid CSR value = 0x0
- **intel_niosv_m_1** (dentro de SDRAM) — **0x 0040 0000 - 0x 007F FFFF (4 MiB)**
  - resetVector={SDRAM.s1,0x 0040 0000},
  - exceptionVector={SDRAM.s1,0x 0040 0020},
  - mhartid CSR value = 0x1

MESSAGE_BUFFER_RAM_BASE

mutex

Tutorial-2 (2 cores): hi_mutex

Please, follow steps described in Part 2 of the document for Lab Assignment 4.

1. Open: Nios V Command Shell Tools
2. Reuse BSP files for both cores of Nios V multiprocessor created in Tutorial-1
3. Compile and link Tutorial-2 master application whose source file is `dualCoreNVm_app1_semaforo_0_Q24.c`
4. Compile and link Tutorial-2 slave application whose source file is `dualCoreNVm_app1_semaforo_1_Q24.c`
5. DE0-Nano board configuration
6. Program download and execution
7. Display output message on the screen

```
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
```

Tutorial-3: multithread parallel programming and performance evaluation of dual-core Nios V multiprocessors

- Benchmark : multiplication **Matrix × Vector**
- Objective 3-1: performance evaluation of a sequential benchmark
- Objective 3-2: performance evaluation of a parallel benchmark for two 2-core multiprocessors: 2 x Nios V/m y 2 x Nios V/g
- Objective 3-3: develop a program for **Matrix × Matrix** multiplication on 2-core Nios V multiprocessors and evaluate performance

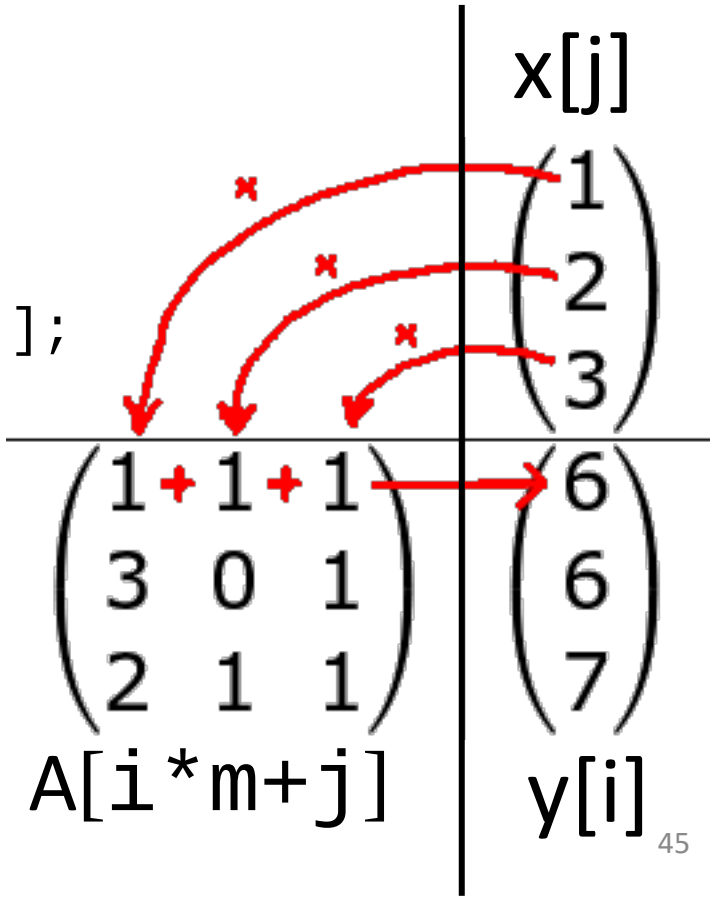
Benchmark: multiplication Matrix × Vector

- Mathematics operation : $y = A \,.\, x$
- Main C source code:

n : number of rows and elements of vectors: x and y

m : number of columns

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



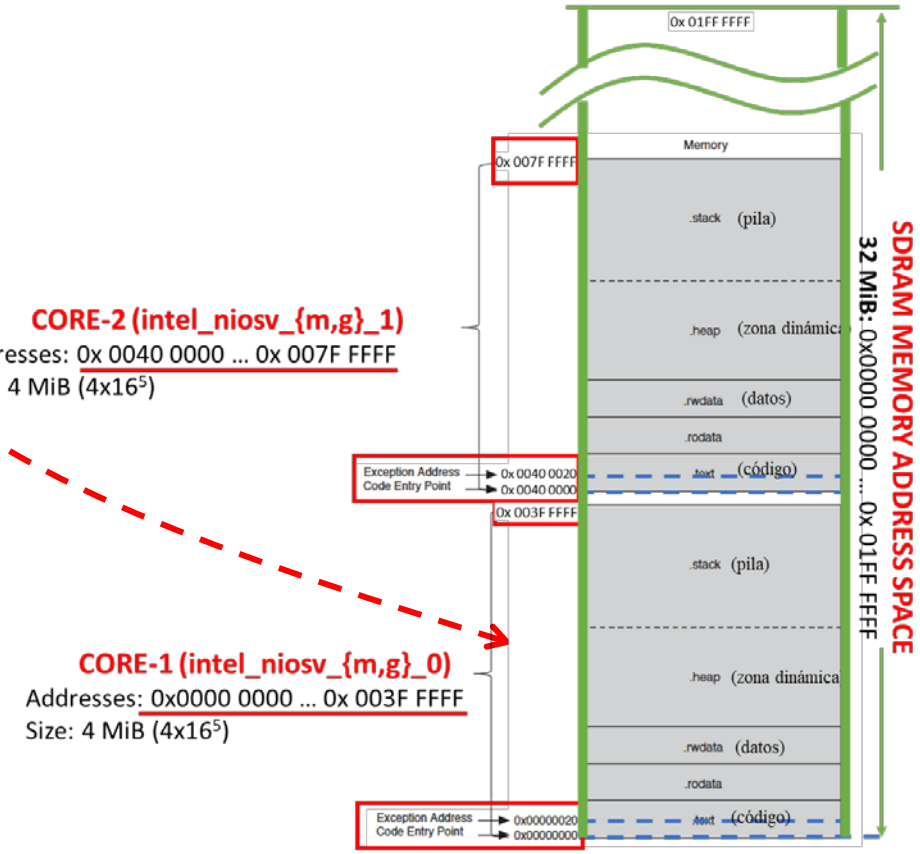
Sequential benchmark for 1-core: source code (MV_serie_2025.c)

```
int main(){
// Shared memory addresses 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
// COMPUTING - Matrix x Vector repeated Niter times
int local_n          = n;
int my_first_row = 0;                // first matrix raw
int my_last_row  = local_n - 1;  // last matrix raw
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()
```

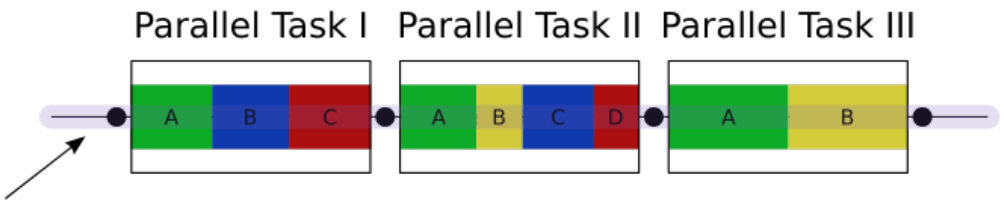
Repetitions of the Matrix-Vector loop

Computing load

Intel_niosv_{m,g}_0 (core 1)

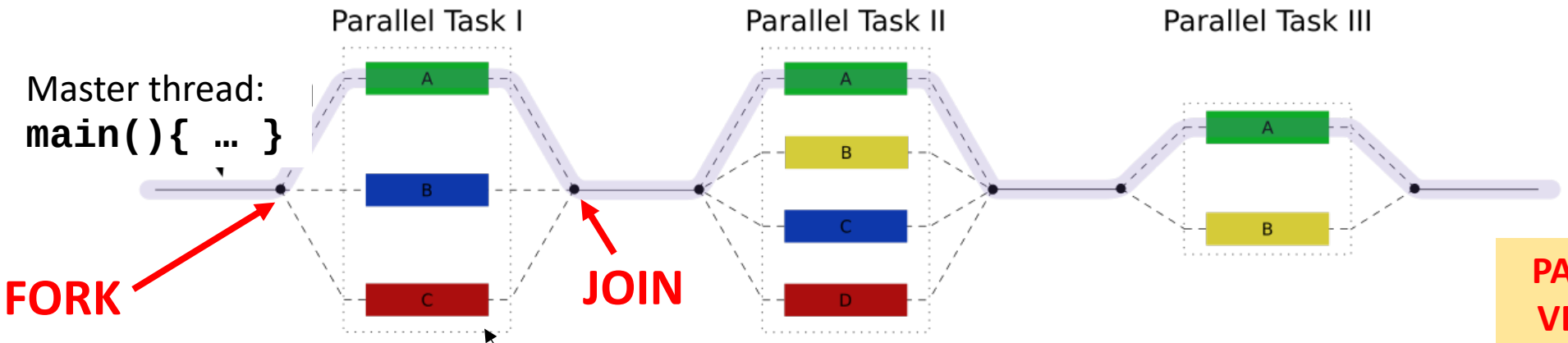


Fork/Join model for parallel programming



Hilo maestro: **`main(){ … }`**

SEQUENTIAL VERSION



Master thread: **`main(){ … }`**

PARALLEL VERSION

Slave threads can interchange data using shared variables: **`main(){ … }`**

Multithread parallel program

Master thread

```
main() {
```

MEMORY INITIALIZATION

FORK SYNCHRONIZATION

```
// Master begins FORK
*(message_buffer_ptr)        = 15
// Master is ready and wait for slave response
*(message_buffer_ptr_fork) = 1
// Master sends number of threads
*(message_buffer_threads)  = thread_count;
//Master sends the number of iterations,
each of them the Matrix x Vector is done
*(message_buffer_Niter)      = Niter;

// Master sends both threads are
synchronized in FORK stage
*(message_buffer_ptr)        = 5
```

COMPUTING

JOIN SYNCHRONIZATION

```
// Master is ready and wait for slave response
*(message_buffer_ptr_join) |= 1

// Master sends both threads are synchronized
in JOIN stage
*(message_buffer_ptr) = 6
```

DISPLAY OF RESULTS

```
}
```

Slave thread

```
main() {
```

MEMORY INITIALIZATION

FORK SYNCHRONIZATION

```
// Slave reads shared variables in RAM
message_buffer_val
                = *(message_buffer_ptr);
message_buffer_val_fork
                = *(message_buffer_ptr_fork);
thread_count
                = *(message_buffer_threads);
Niter           = *(message_buffer_Niter);

// Slave updates shared variable
*(message_buffer_val_fork) |= 2;
```

COMPUTING

JOIN SYNCHRONIZATION

```
// Slave reads shared variables in RAM
message_buffer_val_join
                = *(message_buffer_ptr_join);

// Slave updates shared variable
*(message_buffer_ptr_join) |= 2;
```

```
}
```

Benchmark: partition of the computing load

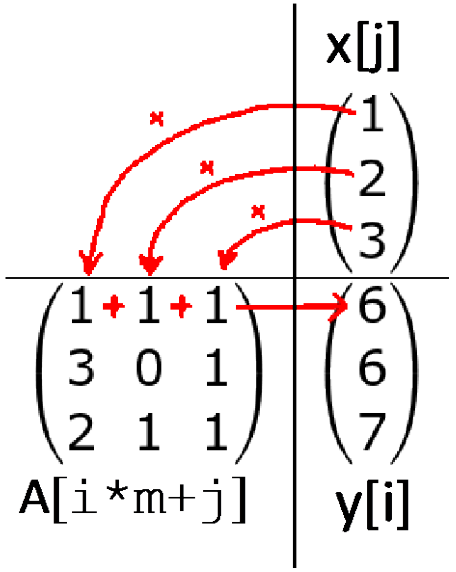
COMPUTING

- C ource code for both threads/programs:

Core/Thread 1: rank = 0
Core/Thread 2: rank = 1
thread_count = 2
Niter = …

```
int local_n          = n / thread_count;
int my_first_row     = rank * local_n;          // first matrix raw
int my_last_row      = (rank+1) * local_n - 1;  // last matrix raw

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



Performance evaluation of the Nios V multiprocessor, sequential version (MV_serie_2025.c)

- **Objective 3-1**:
  - Register execution times for 4 computing loads. Parameters:
    - Niter = 1000, 2000, 5000, 10000.
  - Performance evaluation: fill in Table 2 (only total execution time)
    - Computing: time needed for Matrix × Vector algorithm
    - Total: time needed from the begining to the end of the program
  - Repeat measurements for:
    - Nios V/m (FPGA configuration: `DE0_Nano_DualCoreNiosVm.sof`)
    - Nios V/g (FPGA configuration : `DE0_Nano_DualCoreNiosVg.sof`)
  - Justify results:
    - 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?

Performance evaluation of Nios V/{m,g}

Table 2: Measurements of execution time of the Matrix × 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
Nios V/**m** Multiprocessor	DualCoreNiosVm	1000		1
	DualCoreNiosVm	2000		1
	DualCoreNiosVm	5000		1
	DualCoreNiosVm	10000		1
Nios V/**g** Multiprocessor	DualCoreNiosVg	1000		
	DualCoreNiosVg	2000		
	DualCoreNiosVg	5000		
	DualCoreNiosVg	10000		

Serial execution **using only one core**

Performance evaluation for the 2-core Nios V/{m,g} multiprocessors

- **Objective 3-2** – Execution and performance evaluations of the Matrix × Vector algorithm using 2-core Nios V multiprocessors:
  - Run using **DualCoreNiosVm** multiprocessor (2-core: 2 × Nios V/m)
  - Register execution times activating one thread, using the intel_niosv_m_0 core of the multiprocessor for 4 working loads. Parameters:
    - Nthreads = 1
    - Niter = 1000, 2000, 5000, 10000.
  - Register execution times activating two threads, using the intel_niosv_m_0 amd intel_niosv_m_1 cores of the multiprocessor for 4 working loads. Parameters:
    - Nthreads = 2
    - Niter = 1000, 2000, 5000, 10000.
  - Performance evaluation: fill in Table 3 (see next slide)
  - Repeat experiments using **DualCoreNiosVg** (2-core: 2 × Nios V/g)
  - Justify results: Are these results similar to those obtained for Objective 3-1?, why?

Performance evaluation for the 2-core Nios V/{m,g} multiprocessors

Table 3: Performance evaluation of the Matrix × 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 × Vector algorithm.

	FPGA configuration	Number of threads	N_{iter}	Time (ms)	Speed-up	Parallelism efficiency	
Nios V/**m** Multiprocessor	DE0_Nano_DualCoreNiosVm.sof	1	1000		1	100 %	1-thread
	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				2-threads
	DE0_Nano_DualCoreNiosVm.sof	2	2000				
	DE0_Nano_DualCoreNiosVm.sof	2	5000				
	DE0_Nano_DualCoreNiosVm.sof	2	10000				
Nios V/**g** Multiprocessor	DE0_Nano_DualCoreNiosVg.sof	1	1000		1	100 %	1-thread
	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				2-threads
	DE0_Nano_DualCoreNiosVg.sof	2	2000				
	DE0_Nano_DualCoreNiosVg.sof	2	5000				
	DE0_Nano_DualCoreNiosVg.sof	2	10000				

Performance evaluation of the Nios V/{m,g} multiprocessors using the Matrix × Matrix benchmark

- **Objective 3-3 –** Develop, run, and evaluate the performance of a Matrix × Matrix (C[] = A[] × B[]) benchmark that multiplies two matrices:
  - Code the algorithm using 8 × 8 matrices
  - Run the program using DualCoreNiosVm (2 cores: 2 × Nios V/m) y DualCoreNiosVg (2 cores: 2 × Nios V/g). Parameters:
    - thread_count =1, 2.
    - Niter = 1000, 2000, 5000, 10000.
  - Performance evaluation: fill in a table similar to Table 3 for this new algorithm
  - Justify results: Are these results similar to those obtained for Objective 3-2?, why?

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