Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

MEMPHIS MANY-CORE PLATFORM'S INSTALLATION AND CONFIGURATION TUTORIAL

MEMPHIS VERSION: 1.2

UTILIZED SOFTWARES' VERSIONS:

- SO: Ubuntu 18.04.3 LTS (Bionic Beaver)
- MIPS-GCC Cross Compiler: Version 4.1.1 (with *in-house modifications*)
- SystemC: 2.3.3
- Questa: 10.6e (with UNISIM libraries)

INTRODUCTION

This tutorial describes how to install and configure Memphis. After this tutorial you will be able to:

1. Compile Memphis in the following hardware models: SystemC-GCC, SystemC-Questa, VHDL
2. Run an example *testcase,* a simple producer-consumer application

NOTE: If you need to develop something in VHDL, you should use Questa, which is a paid tool, thus, you will need an access key to use it. You can get this key with the administrators of PUCRS **GAPH** (Grupo de Apoio de Projeto em Hardware, Hardware Design Support Group). In case you don't use VHDL, the compilation and simulation only use free software.

This tutorial is divided into 5 parts:

1. Build the compilation and simulation environment
2. Create and compile a hardware model
3. Create a simulation scenario, with a producer-consumer application
4. Simulate a platform utilizing the desired hardware model.
5. Debug the simulation utilizing a graphical debugging tool for Many-cores SoCs.

PART 1: COMPILATION AND SIMULATION ENVIRONMENT PREPARATION

PS: for use in the GAPH's laboratory, i.e., '/soft64', please type: source /soft64/source_gaph; source /soft64/source_memphis

Introduction: In this part of the tutorial you will see how to prepare the environment for Memphis compilation and simulation based on a new installation of the specified operating system.

1. Download and install Ubuntu in the version specified previously
2. With a fresh installation, install some useful packages using the following commands:

```
sudo apt-get update
sudo apt-get install gcc-multilib
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
sudo apt-get install build-essential
```

3. Restart the computer

MIPS-CROSS COMPILER INSTALLATION:

1. Create a Memphis directory in your user folder
2. Create a **tools_memphis** directory in your user folder

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

```
mkdir /home/user/memphis
mdkir /home/user/tools_memphis
```

3. Download the MIPS compiler (mips-elf-gcc-4.1.1.zip file) for Memphis, available in this link:

   https://github.com/GaphGroup/hemps/raw/master/tools/mips-elf-gcc-4.1.1.zip

4. Go to the Downloads directory, where you downloaded MIPS and unzip the file:

```
unzip mips-elf-gcc-4.1.1.zip
```

5. Move MIPS unzipped directory to **tools_memphis/** directory

```
mv mips-elf-gcc-4.1.1 /home/user/tools_memphis
```

6. Change the permission for all files inside the directory:

```
chmod -R 777 /home/user/tools_memphis/mips-elf-gcc-4.1.1
```

7. Define environment variables for MIPS. For that, open ".bashrc"

```
cd
gedit .bashrc
```

8. Insert the environment variables to the end of the file:

```
# MIPS
export PATH=/home/user/tools_memphis/mips-elf-gcc-4.1.1/bin:${PATH}
export MANPATH=/home/user/tools_memphis/mips-elf-gcc-4.1.1/man:${MANPATH}
```

9. Save and close the file. Close every terminal that you have opened.
10. Open a new terminal.
11. Test if the mips compiler is correctly accessible. For that, type in the new terminal

```
mips-elf-  (press TAB key 2 times)
```

   It should show something like this:

```
ruaro@ruaropuc:~$ mips-elf-
mips-elf-addr2line  mips-elf-gcc-4.1.1  mips-elf-ranlib
mips-elf-ar         mips-elf-gcov       mips-elf-readelf
mips-elf-as         mips-elf-gdb        mips-elf-run
mips-elf-c++        mips-elf-gdbtui     mips-elf-size
mips-elf-c++filt    mips-elf-ld         mips-elf-strings
mips-elf-cpp        mips-elf-nm         mips-elf-strip
mips-elf-g++        mips-elf-objcopy
mips-elf-gcc        mips-elf-objdump
```

12. In case it doesn't show anything, verify the process again, because something went wrong and the system doesn't have visibility of MIPS compiler binaries.

SYSTEMC-GCC INSTALLATION:

SystemC GCC is a C++ plug-in provided by *Accelera* that allows the language SystemC's compilation.

More details about SystemC versions can be found in Accellera's website (https://accellera.org/downloads/standards/systemc)

1. Download the SystemC source code (file systemc-2.3.3.tar.gz) for Memphis, available in this link:

   https://github.com/GaphGroup/hemps/raw/master/tools/systemc-2.3.3.tar.gz

2. Extract the downloaded file and move it to **tools_memphis**

```
tar xvf systemc-2.3.3.tar.gz
mv systemc-2.3.3 /home/user/tools_memphis
```

3. Go to directory **/home/user/tools_memphis** and runs the following commands

```
cd /home/user/tools_memphis
```

```
sudo mkdir /usr/local/systemc-2.3.3
cd systemc-2.3.3
mkdir objdir
cd objdir
sudo ../configure --prefix=/soft64/util/accelera/systemc/2.3.3
sudo make
sudo make install
```

4. Create environment variables for System C in the same way you did for mips, adding the following commands to the end of your .bashrc

```
# SYSTEMC
export SYSTEMC_HOME=/usr/local/systemc-2.3.3
export C_INCLUDE_PATH=${SYSTEMC_HOME}/include
export CPLUS_INCLUDE_PATH=${SYSTEMC_HOME}/include
export LIBRARY_PATH=${SYSTEMC_HOME}/lib-linux64:${LIBRARY_PATH}
export LD_LIBRARY_PATH=${SYSTEMC_HOME}/lib-linux64:${LD_LIBRARY_PATH}
```

QUESTA INSTALLATION:

1. Acquire the files for local execution of Questa in GAPH laboratory. The files will be in a directory called "10.6e". These files are the same as when you run the **module load questa** command inside GAPH labs.
2. Create and copy the directory to your computer

```
mkdir /soft64/mentor/ferramentas/questa
mv {caminho_remoto}10.6e /soft64/mentor/ferramentas/questa
```

3. Create environment variables for Questa

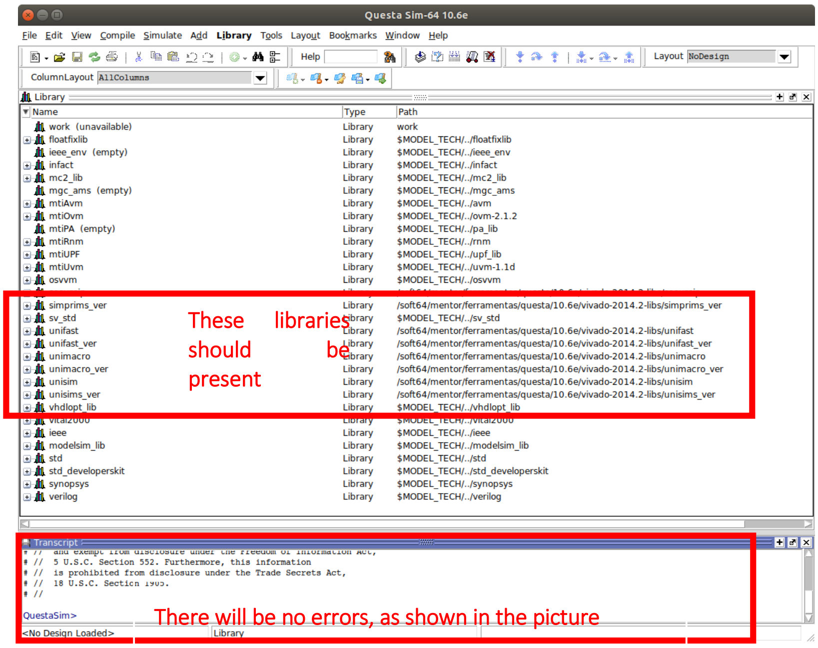
```
# QUESTA PONTING TO KRITI LICENCE
#==============
export LM_LICENSE_FILE=...adquira a licença no GAPH...
export MGLS_LICENSE_FILE=...adquira a licença no GAPH...
export PATH=/soft64/mentor/ferramentas/questa/10.6e/questasim/bin:$PATH
export PATH=/soft64/mentor/ferramentas/questa/10.6e/questasim/linux_x86_64:$PATH
export MODELSIM_HOME=/soft64/mentor/ferramentas/questa/10.6e/questasim
export MTI_BYPASS_SC_PLATFORM_CHECK=1
export MTI_VCO_MODE=64
export MTI_HOME=/soft64/mentor/ferramentas/questa/10.6e/questasim
export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LIBRARY_PATH
```

4. Open a new terminal window and test if Questa was correctly installed by typing the following command:

```
vsim
```

5. This command should open the graphical interface of Questa, without any library loading errors.

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019



MEMPHIS INSTALLATION:

1. Go to your root directory

```
cd ~
```

2. Install Git in your system

```
sudo apt-get install git
```

3. Download Memphis project in the following address on GitHub:

```
git clone https://github.com/GaphGroup/Memphis.git memphis
```

(in case you prefer, it is possible to download an old release by accessing the releases directory: https://github.com/GaphGroup/Memphis/releases)

Memphis directory is organized as follows:

```
.
├── applications ➔ Applications already implemented
├── build_env ➔ Scripts used to generate, compile and execute the platform
├── docs ➔ Sw documentation based in Doxygen tool
├── hardware ➔ Hardware model source code
├── README.txt
├── software ➔ Software model source code (kernel)
├── testcases ➔ Examples of testcase files
└── tutorials ➔ Contains this tutorial
```

4. Create a directory where there will be created new simulation scenarios. Usually, this directory is inside *the user* and is called ***sandbox_memphis***. You have the freedom to create this directory where you want, provided that you reference it appropriately in the environment variables that will be described in the next step (step 5).

```
mkdir /home/user/sandbox_memphis
```

5. Create an environment for Memphis

```
# MEMPHIS
export MEMPHIS_PATH=/home/user/memphis
export MEMPHIS_HOME=/home/user/sandbox_memphis
export PATH=${MEMPHIS_PATH}/build_env/bin:${PATH}
```

6. Test if Memphis commands are visible in anywhere of the system

```
memphis-    (press TAB key 2 times)
```

It should look something like this:

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br

Revision: **Dr. Fernando Moraes**

30/sep/2019

```
[rruaro]$ memphis-
memphis-all         memphis-debugger    memphis-help        memphis-sortdebug
memphis-app         memphis-gen         memphis-run
```

JAVA INSTALLATION:

1. Execute the following command:

```
sudo apt-get install default-jre
```

PYTHON INSTALLATION (with YAML support):

1. Execute the following commands:

```
sudo apt-get install python
sudo apt-get install python-yaml
```

PART 2: HARDWARE MODEL GENERATION

Introduction: In this part of the tutorial there will be seen how to generate the hardware model, choosing three types of description (language) provided: SystemC-GCC, SystemC-Questa e VHDL.

1. Go to the directory in MEMPHIS_HOME (*sandbox_memphis* in the case of this tutorial), then you will create the hardware model (remember to have run: ***mkdir /home/user/sandbox_memphis***)

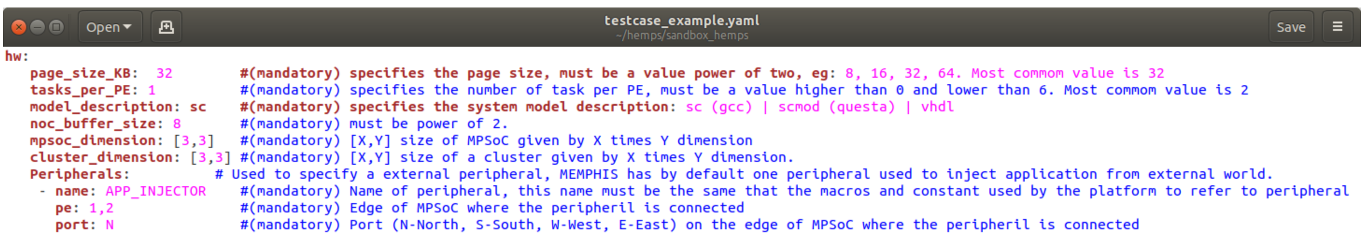
```
cd $MEMPHIS_HOME
```

2. Create the file ***testcase_example.yaml***. This file can have any name, provided that it has the YAML extension. This file is referenced as a ***testcase file***. A *testcase* is a file that is used by Memphis scripts to generate the hardware according to specifications provided by the user. In case you wish to work changing or adding new attributes, search in $MEMPHIS_PATH/build_env/scripts about how the scripts in Python use each attribute. Create a file **testcase_example.yaml**:

```
gedit my_testcase.yaml
```

3. Insert the parameters in the file according to the desired hardware configuration. The following picture demonstrates the content of a testcase example present in memphis/testcase directory. You will use this configuration as a reference from now on. The description of each field is commented next to each field.

```
hw:
  page_size_KB:  32        #(mandatory) specifies the page size, must be a value power of two, eg: 8, 16, 32, 64. The most common value is 32
  tasks_per_PE: 1          #(mandatory) specifies the number of task per PE, must be a value higher than 0 and lower than 6. The most common value is 2
  model_description: sc    #(mandatory) specifies the system model description: sc (gcc) | scmod (questa) | vhdl
  noc_buffer_size: 8       #(mandatory) must be power of 2.
  mpsoc_dimension: [3,3]   #(mandatory) [X,Y] size of MPSoC given by X times Y dimension
  cluster_dimension: [3,3] #(mandatory) [X,Y] size of a cluster given by X times Y dimension.
  Peripherals:             # Used to specify an external peripheral, MEMPHIS has by default one peripheral used to inject application from the external world.
   - name: APP_INJECTOR    #(mandatory) Name of peripheral, this name must be the same that the macros and constantly used by the platform to refer to peripheral
     pe: 1,2               #(mandatory) Edge of MPSoC where the peripheral is connected
     port: N               #(mandatory) Port (N-North, S-South, W-West, E-East) on the edge of MPSoC where the peripheral is connected
```



testcase_example.yaml
~/hemps/sandbox_hemps

```
hw:
  page_size_KB:  32        #(mandatory) specifies the page size, must be a value power of two, eg: 8, 16, 32, 64. Most commom value is 32
  tasks_per_PE: 1          #(mandatory) specifies the number of task per PE, must be a value higher than 0 and lower than 6. Most commom value is 2
  model_description: sc    #(mandatory) specifies the system model description: sc (gcc) | scmod (questa) | vhdl
  noc_buffer_size: 8       #(mandatory) must be power of 2.
  mpsoc_dimension: [3,3]   #(mandatory) [X,Y] size of MPSoC given by X times Y dimension
  cluster_dimension: [3,3] #(mandatory) [X,Y] size of a cluster given by X times Y dimension.
  Peripherals:           # Used to specify a external peripheral, MEMPHIS has by default one peripheral used to inject application from external world.
   - name: APP_INJECTOR    #(mandatory) Name of peripheral, this name must be the same that the macros and constant used by the platform to refer to peripheral
     pe: 1,2               #(mandatory) Edge of MPSoC where the peripheril is connected
     port: N               #(mandatory) Port (N-North, S-South, W-West, E-East) on the edge of MPSoC where the peripheril is connected
```

4. Next step is to generate the hardware model of the platform. Note that the field **model_description** specifies which description language is used. In the case of your file, the description **sc** means **SystemC-GCC**. To change hardware description just change this field to another available option (**scmod** for SystemC-Questa, **vhdl** for VHDL). To generate the hardware model run the **memphis-gen** command calling by reference the testcase file previously created.

```
memphis-gen my_testcase.yaml
```

After compiling the kernel (red messages) and the hardware (green messages), the model is generated and the following message should be displayed at the end of the generation:

```
Memphis platform generated and compiled successfully at:
-/home/user/sandbox_memphis/my_testcase
```

You may notice that a directory with the same name as the testcase was created.

OBS: The environment variable MEMPHIS_HOME is optional. In case it is not defined, the command memphis-gen will create a directory with the name of the testacase inside of the Memphis standard directory destined to testcases (/home/user/memphis/testcase)

Each testcase directory is self-contained. This means that it has a copy of every required file to compile the platform again. This is very useful to replicate the experiments. During your research, you can save the testcase directory for each experiment that you will do, this way you will have full conditions to know exactly which was the kernel and hardware utilized to obtain a certain result.

A directory created by **memphis-gen** has the following subdirectories (shown up to level 2):

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

```
ruaro@ruaropuc:~/hemps/sandbox_hemps/testcase_example$ tree -L 2
.
├── applications  ➔ Stores the application source code (which will be inserted in the future)
│   └── makefile
├── base_scenario ➔ Stores pre-loaded RAM data and the binary of testcase (only for SystemC-GCC)
│   ├── ram_pe
│   └── testcase_example
├── build ➔ Stores all scripts used to build and to compile testcase and applications
│   ├── app_builder.py
│   ├── banner.py
│   ├── build_utils.py
│   ├── build_utils.pyc
│   ├── deloream_env.py
│   ├── hw_builder.py
│   ├── kernel_builder.py
│   ├── scenario_builder.py
│   ├── testcase_builder.py
│   ├── wave_builder.py
│   ├── wave_builder.pyc
│   ├── yaml_intf.py
│   └── yaml_intf.pyc
├── hardware ➔ Stores the compiled hardware and its respective source code
│   ├── app_injector.o
│   ├── dmni.o
│   ├── makefile
│   ├── memphis.o
│   ├── mlite_cpu.o
│   ├── pe.o
│   ├── queue.o
│   ├── ram.o
│   ├── router_cc.o
│   ├── sc
│   ├── switchcontrol.o
│   ├── test_bench.o
│   └── testcase_example
├── include ➔ Stores files used and include during the kernel and hw compilation. These files
│   │          reflect parameter of .yaml file
│   ├── kernel_pkg.c
│   ├── kernel_pkg.h
│   ├── kernel_pkg.o
│   └── memphis_pkg.h
├── makefile
├── software ➔ Stores kernel source code and binaries. The .lst for each kernel is also
│   │           preserved, allowing to debug the CPU instruction flows using waveforms
│   ├── boot_master.o
│   ├── boot_slave.o
│   ├── boot_task
│   ├── include
│   ├── kernel
│   ├── kernel_master.bin
│   ├── kernel_master_debug.bin
│   ├── kernel_master_debug.map
│   ├── kernel_master.dump
│   ├── kernel_master.lst
│   ├── kernel_master.map
│   ├── kernel_master.o
│   ├── kernel_master.txt
│   ├── kernel_slave.bin
│   ├── kernel_slave_debug.bin
│   ├── kernel_slave_debug.map
│   ├── kernel_slave.dump
│   ├── kernel_slave.lst
│   ├── kernel_slave.map
│   ├── kernel_slave.o
│   ├── kernel_slave.txt
│   ├── makefile
│   └── modules
└── testcase_example.yaml ➔ A safe copy of the testcase file
```

PARTE 3: CREATION OF SIMULATION SCENARIO

Introduction: In this step of the tutorial you will create a ***scenario***. A ***scenario*** is a file that describes the applications set that will run in the system. These applications can be developed inside of the applications/ directory of the created testcase or can be imported from the directory applications/ in Memphis' root directory (/home/user/memphis/applications).

Go to the directory MEMPHIS_HOME

```
cd $MEMPHIS_HOME
```

As mentioned previously, it is necessary to have a developed application to use in the scenario. In this tutorial, you will develop an application inside of the testcase_example/applications directory and also will import an existing application of Memphis' root directory.

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

DEVELOPING A USER-MADE APPLICATION:

1. Go to the applications directory created inside of testcase (that should be empty, except a makefile file)

```
cd $MEMPHIS_HOME/my_testcase/applications
```

2. Create a new directory with your application name and access it.

```
mkdir prod_cons
cd prod_cons
```

Here you will be creating a 'prod_cons' application that has a producer task (prod.c), generating packets to a consumer task (cons.c).

3. Create a prod.c task. On Memphis, each task is represented by a file .c

```
gedit prod.c
```

4. Insert the following source code, which will make the prod task to send messages to the cons task

```
#include <api.h>
#include <stdlib.h>

void main(){

       //Creating a message data structure.
       //Message is a structure defined in api.h
       Message msg;

       //MSG_SIZE = max uint size allowed for a
       //single message in Memphis
       msg.length = MSG_SIZE;

       //Initializing the msg with some data
       for(int i=0; i<MSG_SIZE; i++){
                                 msg.msg[i] = 500 + i;
       }

       //Loop that sends 2000 messages to task cons.c
       for(int msg_numbr = 0; msg_numbr<2000; msg_numbr++){
           Send(&msg, cons); //Sends a message to cons task

           //Echo prints log strings
           Echo("Message produced - number: ");
           Echo(itoa(msg_numbr));
        }

     //Termianting the program
     exit();

}
```

5. Create the cons.c task

```
gedit cons.c
```

6. Insert the following source code, which will make the cons task to receive the prod task's messages

```
#include <api.h>
#include <stdlib.h>

void main(){

       //Creating a message data structure.
       //Message is a structure defined in api.h
       Message msg;

       //Loop that receives 2000 messages to task cons.c
       for(int msg_numbr = 0; msg_numbr<2000; msg_numbr++){
           Receive(&msg, prod); //Receives a message from prod task

           //Echo prints log strings
```

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

```
            Echo("Message received - number: ");
            Echo(itoa(msg_numbr));

            //Prints the message data
            Echo("Message content: ");
            for(int i=0; i<msg.length; i++){
                   Echo(itoa(msg.msg[i]));
            }
        }

        //Termianting the program
        exit();

}
```

CREATING THE SCENARIO FILE:

1. Go to the MEMPHIS_HOME directory

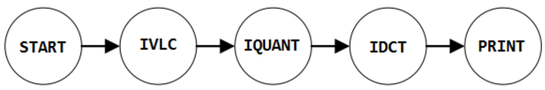
```
cd $MEMPHIS_HOME
```

2. With the user-made application created, now create a YAML file corresponding to the scenario. In this tutorial, you will call it **my_scenario.yaml**

```
gedit my_scenario.yaml
```

The purpose of the scenario file is to specify application characteristics, that is, obligatorily its name, and, optionally the allocation time in the system and the processor address where its tasks will be mapped (static mapping).

As mentioned previously we will use two types of application, a user-made application (already created) and the other application we will import from the Memphis directory. In this tutorial, we will import the MPEG application (whose purpose is to decodify MPEG images). The picture below demonstrates the communication graph between MPEG applications. It is clear that there are 5 tasks and that the communication pattern follows a single pipeline flow, where a task receives an input, applies some kind of processing and sends to the next task until the data flow arrives at the OUTPUT task.



MPEG

Let's create a scenario where the application prod_cons is dynamically mapped in the system and that its execution begins at 2 ms. The MPEG application will be dynamically mapped except for **print** and **start** tasks that will be targeted to PEs specified by the scenario. Static mapping favors scenarios where, for example, the input and output flow of data must be near to extern chip resources, in other words, in its edge, hence we will map the task **start** in PE (Processing elements) 2x2 and the task **print** in PE 2x0. To create a scenario of this kind, edit the file *my_scenario.yaml with the following content*. Tag cluster specifies which cluster (Processor set) in which the application will be mapped. In this example, we are working with only 1 cluster, so the specified value of the cluster is 0.

```
apps:
  - name: prod_cons
  - name: mpeg        #(mandatory) name of application, must be equal to the application folder name
    cluster: 0        #(optinal) index of the statically mapped cluster - dynamic mapping by default
    start_time_ms: 15 #(optinal) any unsigned integer number - 0 by default
    static_mapping:   #(optinal) field, used to store static mapping information of tasks
      start: [2,2]    # Task start from app mpeg will be mapped as static at address X=2, Y=2
      print: [2,0]    # Task print from app mpeg will be mapped as static at address X=2, Y=0
```

apps:
 - name: prod_cons
 - name: mpeg
   cluster: 0
   start_time_ms: 15

```
static_mapping:
 start: [2,2]
 print: [2,0]
```

COMPILING APPLICATIONS:

1. After creating a scenario file, let's compile the applications (prod_con and MPEG) by the memphis-app command. There are two ways of using memphis-app.

   (a) Passing the application names as parameters

```
memphis-app my_testcase.yaml prod_cons
memphis-app my_testcase.yaml mpeg
```

   (b) Passing the scenario file as parameter

```
memphis-app my_testcase.yaml -all my_scenario.yaml
```

(a) allows compiling each application individually passing its full name. Memphis-app command will search for a directory with the same name inside of the $MEMPHIS/HOME/testcase_example/application/ directory and will compile all existing '.c' files, assuming that each '.c' represents a task. In the instance of the command not finding any directory with this name, it will search in the standard memphis application directory: $MEMPHIS_PATH/applications, it will copy the application to MEMPHIS/HOME/testcase_example/application/ and it will compile the application.

(b) causes all application present in the **my_scenario.yaml** file to be compiled. In practice, each application present in the file is compiled individually as in option 'a)'. The command in b) is just convenient when you don't want to compile application by application.

PART 4: SIMULATION

Introduction: In this part of the tutorial let's simulate a scenario utilizing the hardware and kernel model generated previously and the application previously compiled.

1. Run memphis-run

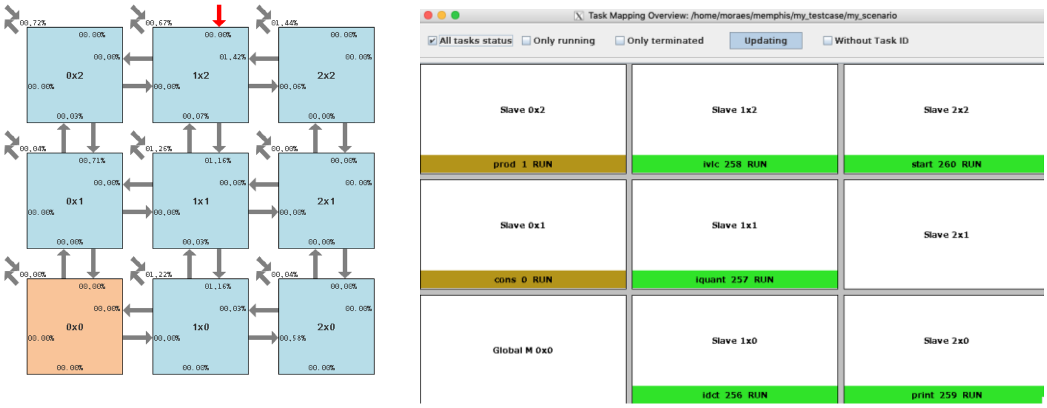
```
memphis-run my_testcase.yaml my_scenario.yaml 20
```

memphis-run command requires

- **1st argument**: testcase file (hardware)
- **2nd argument**: scenario file (software)
- **3rd argument**: simulation time (milliseconds).

When running the command, a directory with the scenario name will be created inside of the testcase directory, the scenario directory contains log and debug information. **memphis-run** starts the simulation automatically by the specified time. If the specific model description in testcase_example.yaml is 'sc', the simulation will be similar to how a normal '.c' file is executed, meaning, a series of logs will show up in the terminal, similarly to the picture below. **The debugging graphical interface must open**:

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019



Tools-> Task Mapping

```
        SystemC 2.3.1-Accellera --- Jan 17 2019 14:04:54
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
Creating PE PE0x0
Creating PE PE1x0
Creating PE PE2x0
Creating PE PE0x1
Creating PE PE1x1
Creating PE PE2x1
Creating PE PE0x2
Creating PE PE1x2
Creating PE PE2x2
App Injector requesting app prod_cons
Master receiving msg
Master sending msg
Master sending msg
Master sending msg
Master sending msg
Master sending msg
Master sending msg
Master sending msg
Master sending msg
Master sending msg
Manager sent ACK
Master receiving msg
Master sending msg
Master receiving msg
App Injector requesting app mpeg
Master receiving msg
Master sending msg
Master sending msg
Master receiving msg
Master receiving msg
Master receiving msg
Master receiving msg
Master receiving msg
```

2. If the specified description model in testcase_example.yaml is 'scmod' or 'vhdl', the command will call Questa simulator, which will load a waveform with the main Memphis signals and start the system simulation.

TERMINATING THE SIMULATION BEFORE THE SPECIFIED TIME:

To terminate a 'sc' simulation just type in any terminal window the following command:

```
killall my_scenario
```

To terminate a simulation in Questa just click the 'stop' icon in Questa's toolbar

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

PART 5: DEBUGGING

Introduction: In this part, let's debug the simulated scenario using a graphic tool developed specifically to debug MPSoCs. It was developed in Java. It also consumes a lot of memory resources.

VIDEO TUTORIAL

There's a YouTube video tutorial showing the main tool functionalities, in this link:

https://youtu.be/nvgtvFcCc60

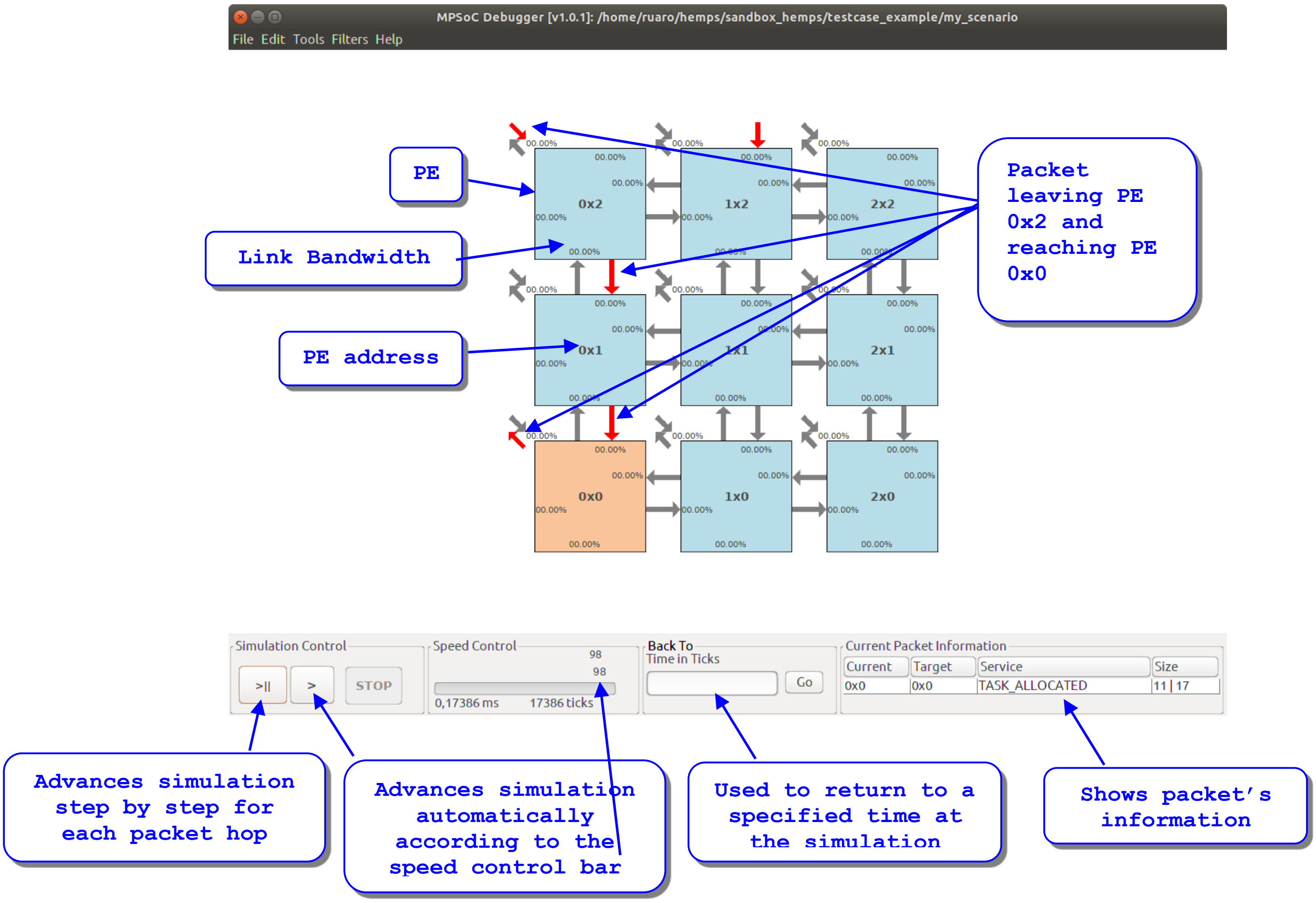
The debugging tool opens automatically after the simulation starts. If you want to open manually, just type the following command in any path in the terminal.

```
memphis-debugger $MEMPHIS_HOME/my_testcase/my_scenario
```

memphis-debugger command requires the scenario directory, created inside of the testcase.

MAIN WINDOW

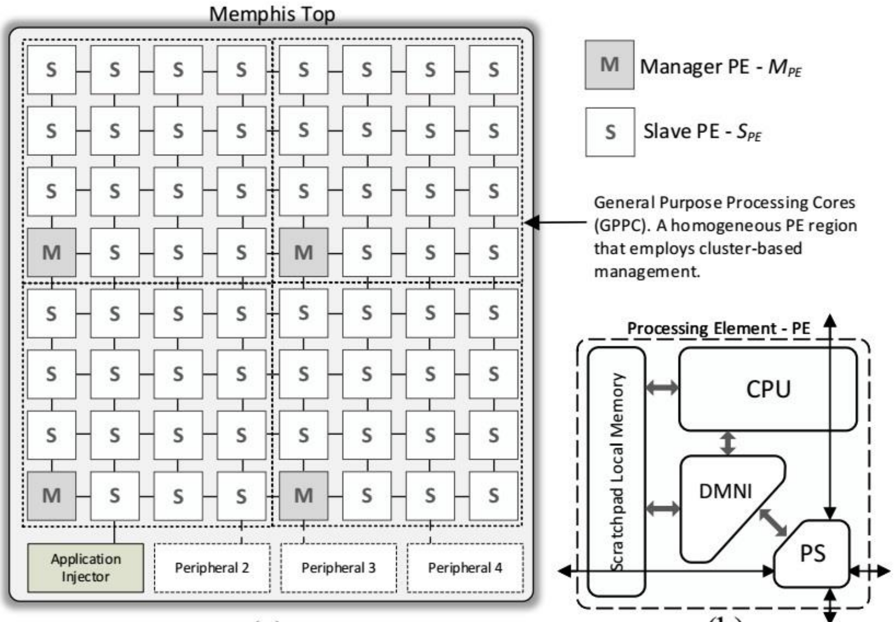
The main window provides a general view of the packets that are traveling in the system, as well as evaluating if there were any errors during execution time and some kind of service. The picture below details the main components of the main window.



Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

APPENDIX I – NEW PERIPHERALS INSTANCIATION

Introduction: This tutorial will explain how to instantiate new peripherals in Memphis, it will approach both hardware descriptions (SystemC and VHDL). Peripherals are hardware components that can be connected to the MPSoC edge. The Memphis architecture is divided into two regions (picture below):



(a)
Overview of the Memphis hardware model, with peripherals connected at the unused south ports of the NoC mesh

(b)
GPPC contains a set of identical PEs that execute general purpose applications

- GPPC: *General Purpose Processing Cores*, consists in the internal chip region, composed by homogeneous PEs and dedicated to applications executing.
- Peripherals: Edge chip region, that allows peripherals implementation that implements hardware acceleration and I/O interface services.

1. The first step to instantiate a new peripheral is to implement it in SystemC (Currently only SystemC is supported). The peripheral interface is a standard Hermes interface, with the following signals:

```
a. Input:
        clock (1 bit)
        reset (1 bit)

        rx (1 bit)
        data_in (FLIT bits – mesma quantidade de bits a 1 flit)
        credit_in (1 bit)

b.          Output:
        tx (1 bit)
        data_out (FLIT bits – mesma quantidade de bits a 1 flit)
        credit_out (1 bit)
```

This interface implements the flow-control protocol based on credits, more details can be obtained in NoC Hermes materials.

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

VHDL

2. With the peripheral being already implemented, and with its interface according to a NoC Hermes's router, the next step consists in opening a new Testbench file and instantiate the peripheral there:

    - Edit the **hardware/vhdl/test_bench.vhd** file in order to instantiate the peripheral and connect it to GPPC.
    - Create signals that will link the peripheral to GPPC (Memphis)

```vhdl
        signal clock                              : std_logic := '0';
        signal reset                              : std_logic;

        -- IO signals connecting App Injector and Memphis
        signal memphis_injector_tx          : std_logic;
        signal memphis_injector_credit_i     : std_logic;
        signal memphis_injector_data_out     : regflit;

        signal memphis_injector_rx          : std_logic;
        signal memphis_injector_credit_o     : std_logic;
        signal memphis_injector_data_in      : regflit;

        -- Create the signals of your IO component here:

```

    - Instantiate the Peripheral

```vhdl
    -- Peripheral 1 - Instantiation of App Injector
    App_Injector : entity work.app_injector
    port map(
        clock         => clock,
        reset         => reset,

        rx            => memphis_injector_tx,
        data_in       => memphis_injector_data_out,
        credit_out    => memphis_injector_credit_i,

        tx            => memphis_injector_rx,
        data_out      => memphis_injector_data_in,
        credit_in     => memphis_injector_credit_o
    );

    -- Peripheral 2 - Instantiate your IO component here:

```

    - Connect the Peripheral to Memphis GPPC

```vhdl
    Memphis : entity work.Memphis
    port map(
        clock                    => clock,
        reset                    => reset,

        -- Peripheral 1 - App Injector
        memphis_app_injector_tx          => memphis_injector_tx,
        memphis_app_injector_credit_i => memphis_injector_credit_i,
        memphis_app_injector_data_out => memphis_injector_data_out,

        memphis_app_injector_rx      => memphis_injector_rx,
        memphis_app_injector_credit_o    => memphis_injector_credit_o,
        memphis_app_injector_data_in     => memphis_injector_data_in

        -- Peripheral 2 - Connect your IO component to Memphis here:

    );

```

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

3. Edit the **hardware/vhdl/memphis.vhd** file in order to connect the peripheral interface to the router

- Insert the interface to the *portmap* with the peripheral

```
entity Memphis is
        port(
            clock                : in  std_logic;
            reset                : in  std_logic;

            -- IO interface - App Injector
            memphis_app_injector_tx          : out std_logic;
            memphis_app_injector_credit_i : in std_logic;
            memphis_app_injector_data_out : out regflit;

            memphis_app_injector_rx          : in std_logic;
            memphis_app_injector_credit_o : out std_logic;
            memphis_app_injector_data_in     : in regflit

            -- IO interface - Create the IO interface for your component here:

        );
end;
```

- Connect the peripheral to the router

```
                --IO App Injector connection
                memphis_app_injector_tx <=  tx(APP_INJECTOR)(io_port(i));
                memphis_app_injector_data_out <= data_out(APP_INJECTOR)(io_port(i));
                credit_i(APP_INJECTOR)(io_port(i)) <= memphis_app_injector_credit_i;

                rx(APP_INJECTOR)(io_port(i)) <= memphis_app_injector_rx ;
                memphis_app_injector_credit_o  <= credit_o(APP_INJECTOR)(io_port(i));
                data_in(APP_INJECTOR)(io_port(i)) <= memphis_app_injector_data_in;

            end generate;

            --Insert the IO wiring for your component here:
```

4. Edit makefile: **build_env/makes/make_vhdl**

- Add the .cpp file name that contains the peripheral implementation

```
MEMPHIS_PKG =memphis_pkg
STAND       =standards
TOP         =memphis_test_bench
IO          =app_injector  meu_periferico
PE          =pe
DMNI        =dmni
```

SYSTEMC

5. With the peripheral being already implemented, and with its interface according to a NoC Hermes's router, the next step consists in opening a new Testbench file and instantiate the peripheral there:

- Edit the **hardware/sc/test_bench.h** file in order to instantiate the peripheral and connect it to GPPC.

- Create signals that will link the peripheral to GPPC (Memphis)
- Instantiate the Peripheral

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

```
    app_injector * io_app;

    char aux[255];
    FILE *fp;

    SC_HAS_PROCESS(test_bench);
    test_bench(sc_module_name name_, char *filename_= "output_master.txt") :
    sc_module(name_), filename(filename_)
    {
        fp = 0;

        MPSoC = new memphis("Memphis");
        MPSoC->clock(clock);
        MPSoC->reset(reset);
        MPSoC->memphis_app_injector_tx(memphis_injector_tx);
        MPSoC->memphis_app_injector_credit_i(memphis_injector_credit_i);
        MPSoC->memphis_app_injector_data_out(memphis_injector_data_out);
        MPSoC->memphis_app_injector_rx(memphis_injector_rx);
        MPSoC->memphis_app_injector_credit_o(memphis_injector_credit_o);
        MPSoC->memphis_app_injector_data_in(memphis_injector_data_in);


        io_app = new app_injector("App_Injector");
        io_app->clock(clock);
        io_app->reset(reset);
        io_app->rx(memphis_injector_tx);
        io_app->data_in(memphis_injector_data_out);
        io_app->credit_out(memphis_injector_credit_i);
        io_app->tx(memphis_injector_rx);
        io_app->data_out(memphis_injector_data_in);
        io_app->credit_in(memphis_injector_credit_o);

        //Instantiate your IO component  here
        //...

        SC_THREAD(ClockGenerator);
```

Connect the peripheral to the router

- Connect the Peripheral to Memphis GPPC

```
        MPSoC->memphis_app_injector_tx(memphis_injector_tx);
        MPSoC->memphis_app_injector_credit_i(memphis_injector_credit_i);
        MPSoC->memphis_app_injector_data_out(memphis_injector_data_out);
        MPSoC->memphis_app_injector_rx(memphis_injector_rx);
        MPSoC->memphis_app_injector_credit_o(memphis_injector_credit_o);
        MPSoC->memphis_app_injector_data_in(memphis_injector_data_in);


        io_app = new app_injector("App_Injector");
```

6. Edit the **hardware/sc/memphis.h** file in order to connect the peripheral interface to the router

- Insert the interface to the *portmap* with the peripheral

```
    //IO interface - App Injector
    sc_out< bool >          memphis_app_injector_tx;
    sc_in< bool >           memphis_app_injector_credit_i;
    sc_out< regflit >       memphis_app_injector_data_out;

    sc_in< bool >           memphis_app_injector_rx;
    sc_out< bool >          memphis_app_injector_credit_o;
    sc_in< regflit >        memphis_app_injector_data_in;

    //IO interface - Create the IO interface for your component here
```

- Connect the peripheral to the router

```
        SC_METHOD(pes_interconnection);
        sensitive << memphis_app_injector_tx;
        sensitive << memphis_app_injector_credit_i;
        sensitive << memphis_app_injector_data_out;
        sensitive << memphis_app_injector_rx;
        sensitive << memphis_app_injector_credit_o;
        sensitive << memphis_app_injector_data_in;
        for (j = 0; j < N_PE; j++) {
            for (i = 0; i < NPORT - 1; i++) {
                sensitive << tx[j][i];
                sensitive << data_out[j][i];
                sensitive << credit_i[j][i];
                sensitive << data_in[j][i];
                sensitive << rx[j][i];
                sensitive << credit_o[j][i];
            }
        }
    }
};
```

Author: **Dr. Marcelo Ruaro** – marcelo.ruaro@acad.pucrs.br
Revision: **Dr. Fernando Moraes**
30/sep/2019

Open memphis.cpp to implement the Peripheral interface connection with the router

```
        //--IO Wiring (Memphis <-> IO) ----------------------
        if (i == APP_INJECTOR && io_port[i] != NPORT) {
            p = io_port[i];
            memphis_app_injector_tx.write(tx[APP_INJECTOR][p].read());
            memphis_app_injector_data_out.write(data_out[APP_INJECTOR][p].read());
            credit_i[APP_INJECTOR][p].write(memphis_app_injector_credit_i.read());

            rx[APP_INJECTOR][p].write(memphis_app_injector_rx.read());
            memphis_app_injector_credit_o.write(credit_o[APP_INJECTOR][p].read());
            data_in[APP_INJECTOR][p].write(memphis_app_injector_data_in.read());
        }
        //Insert the IO wiring for your component here
```

7. Edit makefile: **build_env/makes/make_systemc and build_env/makes/make_systemc_mod**

   Add the .cpp file name that contains the peripheral implementation Add the .cpp file name that contains the peripheral implementation

```
#SystemC files
TOP          =memphis test_bench
IO           =app_injector
PE           =pe
DMNI         =dmni
MEMORY       =ram
PROCESSOR    =mlite_cpu
ROUTER       =queue switchcontrol router_cc
```

8. The next steps are independent of VHDL or SystemC. Once the source code files modifications are completed, the next step is to specify the peripheral name in the testcase YAML file and the position where it will be connected to GPPC.

ATTENTION: The peripheral name **must** be the same name used in the source code to reference it, in case of AppInjector, the hardware is using APP_INJECTOR macro, thus the testcase must specify the name APP_INJETCTOR in the testcase space destinated to describe the peripheral.

```
  mpsoc_dimension: [2,2]    #(mandatory) [X,Y] siz
  cluster_dimension: [2,2]  #(mandatory) [X,Y] siz
  Peripherals:              # Used to specify a exter
   - name: APP_INJECTOR     #(mandatory) Name of p
     pe: 1,1                #(mandatory) Edge of M
     port: E                #(mandatory) Port (N-N
```

When creating a peripheral in testcase, like the one made for APP_INJECTOR, APP_INJECTOR macro stays visible to hardware files.

This macro also stays visible in kernel slave and master. These kernels must use the APP_INJECTOR macro to fill any packet header that the kernel wants to send to the AppInjector. Look at this example of kernel_master.c, where it sends an APP_ALLOCATION_REQUEST packet to AppInjector peripheral, in line 158, kernel, uses the APP_INJECTOR macro to fill the field content p->header:

```
/** Assembles and sends a APP_ALLOCATION_REQUEST packet to the global master
 *  \param app The Application instance
 *  \param task_info An array containing relevant task informations
 */
void send_app_allocation_request(Application * app, unsigned int * task_info){

    ServiceHeader *p;

    p = get_service_header_slot();

    p->header = APP_INJECTOR;

    p->service = APP_ALLOCATION_REQUEST;
```