

TUTORIAL DE INSTALAÇÃO E CONFIGURAÇÃO DA PLATAFORMA MANY-CORE **MEMPHIS**

VERSÃO DA MEMPHIS: 1.2

VERSÕES DOS PRINCIPAIS SOFTWARES UTILIZADOS:

- SO: Ubuntu 18.04.1 LTS (bionic)
- MIPS-GCC Cross Compiler: Versão 4.1.1 (com modificações feitas *in-house*)
- SystemC: 2.3.3
- Questa: 10.6e (com inserções de bibliotecas UNISIM)

INTRODUÇÃO

Este tutorial descreve como instalar e configurar a Memphis. Após esse tutorial você será capaz de:

1. Compilar a Memphis nos seguintes modelos de hardware: SystemC-GCC, SystemC-Questa, VHDL
2. Executar um *testcase* exemplo, com uma aplicação simples de produtor e consumidor

OBS: Se você precisar desenvolver algo em VHDL, deverá utilizar o Questa, que é uma ferramenta paga, por isso, será necessário ter acesso a chave para uso da ferramenta. Essa chave você consegue com os administradores do GAPH (Grupo de Apoio de Projeto em Hardware) da PUCRS. Caso não utilize VHDL, toda a compilação e simulação utilizam software gratuitos.

Este tutorial é dividido em 5 partes:

1. Montar o ambiente de compilação e simulação
2. Criar e compilar um modelo de hardware
3. Criar um cenário de simulação, com uma aplicação produtora-consumidora
4. Simular a plataforma utilizando o modelo de hardware desejado.
5. Depurar a simulação utilizando uma ferramenta gráfica de depuração para Many-cores SoCs.

PARTE 1: PREPARAÇÃO DO AMBIENTE DE COMPILAÇÃO E SIMULAÇÃO

Para uso no laboratório, ou seja /soft64, digitar: `source /soft64/source_gaph; source /soft64/source_memphis`

Introdução: Nesta parte do tutorial será visto como preparar o ambiente pra compilação e simulação da Memphis assumindo como base uma instalação nova do sistema operacional especificado.

1. Faça download e instale o Ubuntu na versão especificada anteriormente
2. Com uma instalação limpa, instale os pacotes que darão suporte a execução de aplicações através dos seguintes comandos:

```
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. Reinicie o computador

INSTALAÇÃO DO MIPS-CROSS COMPILER:

1. Crie um diretório memphis dentro do seu user
2. Crie um diretório tools_memphis dentro do seu user

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

3. Baixe o compilador mips (arquivo mips-elf-gcc-4.1.1.zip) para Memphis (e Hemps também), disponível neste link:

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

4. Vá para o diretório de Downloads, onde você baixou o mips e descompacte o arquivo:

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

5. Mova a o diretório descompactado do mips para o diretório tools_memphis/

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

6. Altere as permissões de todos os arquivos de dentro do diretório:

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

7. Defina variáveis de ambiente para o MIPS. Para isso, abra o arquivo .bashrc

```
cd
gedit .bashrc
```

8. Insira as variáveis de ambiente ao final do arquivo:

```
# 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. Salve e feche o arquivo. Feche todos os terminais que você tem aberto.

10. Abra um novo terminal.

11. Teste se o compilador mips está corretamente visível em qualquer parte do sistema. Para isso digite no novo terminal

```
mips-elf- (pressione a tecla TAB 2 vezes)
```

Deverá aparecer algo assim:

```
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. Caso não mostrar nada, verifique o processo novamente, pois alguma coisa aconteceu de errado e o sistema não tem visibilidade dos binários do compilador mips.

INSTALAÇÃO DO SYSTEMC-GCC:

O SystemC GCC é um plugin C++ fornecido pela Accellera que permite a compilação da linguagem SystemC.

Mais detalhes sobre versões do SystemC podem ser obtidos no site da Accellera (<https://accellera.org/downloads/standards/systemc>)

1. Baixe o código fonte do SystemC (arquivo systemc-2.3.3.tar.gz) para Memphis (e Hemps também), disponível neste link:

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

2. Descompacte o arquivo baixado e mova-o para o diretório tools_memphis

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

3. Vá para o diretório /home/user/tools_memphis e execute os seguintes comandos

```
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. Crie as variáveis de ambiente para o SystemC da mesma forma que fez com o mips, adicionando os seguintes comandos ao final do seu .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}
```

INSTALAÇÃO DO QUESTA:

1. Adquira os arquivos para execução local do Questa no laboratório GAPH. Os arquivos deverão estar em um diretório chamado 10.6e. Esses arquivos são os mesmos que são carregados quando se aplica o comando **module load questa** de dentro do GAPH.
2. Crie e copie o diretório para o local do seu computador

```
mkdir /home/user/tools/memphis/questa
mv {caminho_remoto}10.6e /soft64/mentor/ferramentas/questa
```

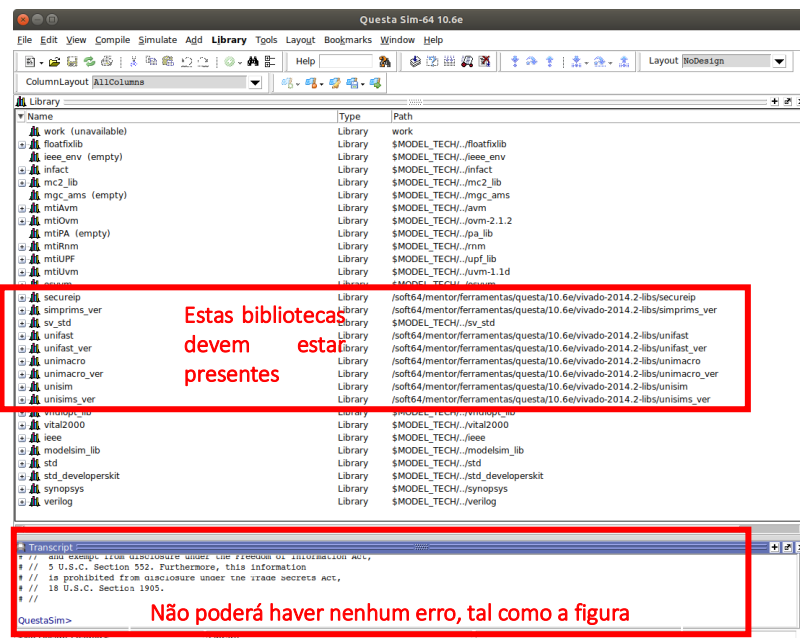
3. Crie as variáveis de ambiente para o Questa

```
# 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=/home/user/tools/memphis/questa/10.6e/questasim/bin:$PATH
export PATH=/home/user/tools/memphis/questa/10.6e/questasim/linux_x86_64:$PATH
export MODELSIM_HOME=/home/user/tools/memphis/questa/10.6e/questasim
export MTI_BYPASS_SC_PLATFORM_CHECK=1
export MTI_VCO_MODE=64
export MTI_HOME=/home/user/tools/memphis/questa/10.6e/questasim
export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LIBRARY_PATH
```

4. Abra um novo terminal e teste se o questa foi instalado corretamente digitando o seguinte comando:

```
vsim
```

5. Este comando deverá abrir a interface gráfica do Questa, sem nenhum erro de carregamento de bibliotecas.



INSTALAÇÃO DA MEMPHIS:

1. Vá para o diretório raiz do seu usuário

```
cd ~
```

2. Instale o suporte a git no seu sistema

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

3. Baixe o projeto da Memphis do seguinte endereço no GitHub:

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

(caso preferir, é possível baixar um release antigo acessando o diretório de releases: <https://github.com/GaphGroup/Memphis/releases>)

O diretório da Memphis é organizado da seguinte maneira

```
├── 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. Crie um diretório onde serão criados novos cenários de simulação. Por costume, esse diretório fica dentro de *user* e chama-se *sandbox_memphis*. Você tem liberdade para criar esse diretório onde quiser e chamá-lo como quiser, desde que faça referência ele apropriadamente nas variáveis de ambiente que serão descritas no próximo passo.

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

5. Crie variáveis de ambiente para a Memphis

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

6. Teste se os comandos da memphis estão visíveis em qualquer lugar do sistema

```
memphis- (pressione a tecla TAB 2 vezes)
```

Deverá aparecer algo assim:

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

INSTALAÇÃO DO JAVA:

1. Execute o seguinte comando:

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

INSTALAÇÃO DO PYTHON com suporte YAML e Tkinter:

1. Execute os seguintes comandos:

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

PARTE 2: GERAÇÃO DO MODELO DE HARDWARE

Introdução: Nesta parte 2 do tutorial será visto como gerar o modelo de hardware escolhendo os três tipos de descrição (linguagem) fornecidos: SystemC-GCC, SystemC-Quarta e VHDL.

1. Vá para o diretório apontado por MEMPHIS_HOME (sandbox_memphis no caso desde tutorial), lá iremos criar o modelo de hardware (lembrar de ter feito: `mkdir /home/user/sandbox_memphis`)

```
cd $MEMPHIS_HOME
```

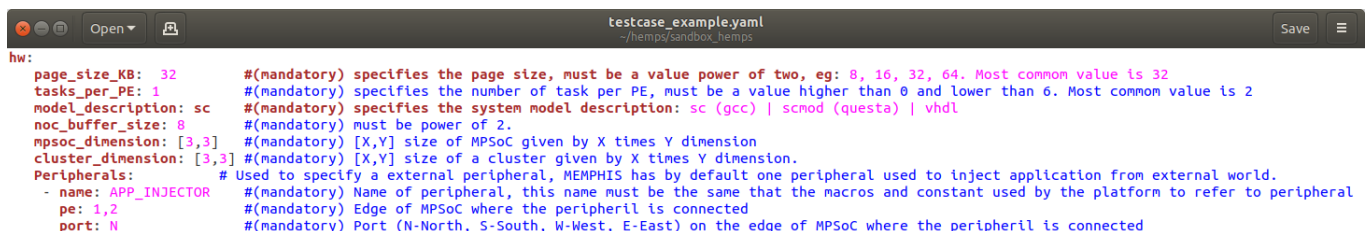
2. Criar o arquivo `testcase_example.yaml`. Este arquivo pode ter qualquer outro nome, desde que mantenha extensão e o formato YAML. Este arquivo é referenciado como um arquivo de `testcase`. Um `testcase` é um arquivo que é utilizado pelos scripts da memphis para gerar o hardware de acordo com especificações fornecidas pelo usuário. Caso deseje trabalhar alterando ou adicionando novos atributos, pesquise dentro do diretório `$MEMPHIS_PATH/build_env/scripts` sobre como os scripts em Python usam cada atributo. Crie o arquivo `testcase_example.yaml`:

```
gedit my_testcase.yaml
```

3. Insira no arquivo os parâmetros de acordo com a configuração de hardware desejada. A figura abaixo demonstra o conteúdo de um testcase exemplo presente dentro do diretório `memphis/testcase`. Iremos utilizar essa configuração como referência daqui para frente. A descrição de cada campo está inserida como comentário ao lado de cada campo.

```
hw:
  page_size_KB: 32      #(mandatory) specifies the page size, must be a value power of two, eg: 8, 16, 32, 64. 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. 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 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 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
```

4. Verifique novamente o formato do arquivo .yaml. O Yaml é muito sensível aos espaços, que são usados para indentação (TAB não é permitido e produz um erro). Para evitar qualquer problema, use dois espaços para recuar cada linha. Problema com o formato .yaml, visite o site oficial em <https://yaml.org/spec/1.2/spec.html>



5. Como o arquivo de testcase criado, o próximo passo é gerar o modelo de hardware da plataforma. Note que o campo `model_description` do testcase especifica qual a linguagem de descrição de hardware utilizada. No caso do nosso arquivo, a descrição `sc` significa **SystemC-GCC**. Para alterar a descrição do hardware basta alterar este campo com as outras opções disponíveis (`scmod` para SystemC-Quarta, `vhdl` para VHDL). Para gerar o modelo de hardware execute o comando `memphis-gen` passando como referência o arquivo testcase criado anteriormente.

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

Após compilar o kernel (mensagens em vermelho) e o hardware (mensagens em verde), o modelo estará gerado e a seguinte mensagem deverá ser exibida ao final da geração:

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

Você poderá perceber que um diretório com o mesmo nome do testcase foi criado.

OBS: A variável de ambiente MEMPHIS_HOME é opcional. Caso não esteja definida, o comando memphis-gen irá criar um diretório com o nome do testcase dentro do diretório padrão da memphis destinado a testcases (/home/user/memphis/testcase)

Todo diretório de testcase é **autocontido**. Isso significa que ele possui uma cópia de todos os arquivos necessário para compilar a plataforma novamente. Isso é muito útil para replicar experimentos. Durante sua pesquisa, você pode salvar o diretório de testcase para cada experimento que você irá fazer, assim terá plenas condições de saber exatamente qual era o kernel e hardware utilizado para obter determinado resultado.

Um diretório de testcase criado pelo **memphis-gen** possui os seguintes subdiretórios (mostrado até o nível 2):

```
ruaro@ruaropuc:~/hempis/sandbox_hempis/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
│   ├── delorean_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: CRIAÇÃO DE CENÁRIO DE SIMULAÇÃO

Introdução: Nesta parte 3 do tutorial vamos criar um **cenário**. Um **cenário** é um arquivo que descreve o **conjunto de aplicações** que irão executar no sistema. Estas aplicações podem ser desenvolvidas dentro do diretório applications/ do testcase criado, ou podem ser importadas do diretório applications/ do diretório raiz da memphis (/home/user/memphis/applications).

Vá para o diretório MEMPHIS_HOME

```
cd $MEMPHIS_HOME
```

Como mencionando anteriormente, é necessário ter uma aplicação desenvolvida para usar no cenário. Neste tutorial, vamos desenvolver uma aplicação própria dentro do diretório testcase_example/applications e também iremos importar uma aplicação existente do diretório raiz da memphis.

DESENVOLVENDO UMA APLICAÇÃO PRÓPRIA:

1. Vá para o diretório applications criado dentro do testcase (que deve estar vazio exceto por um arquivo de makefile)

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

2. Crie um novo diretório com o nome da sua aplicação e acesse ele

```
mkdir prod_cons  
cd prod_cons
```

Aqui estamos criando a aplicação 'prod_cons' que possui uma tarefa produtora (prod.c), gerando pacotes para uma tarefa consumidora (cons.c).

3. Crie a tarefa prod.c. Na Memphis, cada tarefa é representada por um arquivo .c

```
gedit prod.c
```

4. Insira o seguinte código fonte, o qual fará com que a tarefa prod envie mensagens para a tarefa cons

```
#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));  
    }  
  
    //Terminating the program  
    exit();  
}
```

5. Crie a tarefa cons.c

```
gedit cons.c
```

6. Insira o código fonte abaixo, o qual fará com que a tarefa cons receba as mensagens da tarefa prod

```
#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
        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]));
        }
    }

    //Terminating the program
    exit();
}
```

CRIANDO O ARQUIVO DE CENÁRIO:

1. Vá para o diretório MEMPHIS_HOME

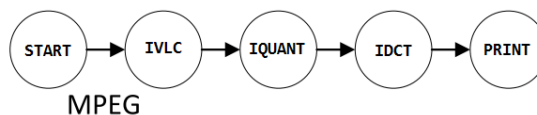
```
cd $MEMPHIS_HOME
```

2. Com a aplicação própria criada crie agora o arquivo yaml correspondente ao cenário. Neste tutorial, vamos chamá-lo de **my_scenario.yaml**

```
gedit my_scenario.yaml
```

O objetivo do arquivo de cenário é especificar características das aplicações, que são, obrigatoriamente o seu nome, e, opcionalmente o tempo de alocação no sistema e o endereço dos processadores onde suas tarefas serão mapeadas (mapeamento estático).

Como mencionando anteriormente vamos usar dois tipos de aplicações, uma aplicação própria (já criada) e a outra aplicação iremos importar do diretório da memphis. Neste tutorial vamos importar a aplicação MPEG (cujo objetivo é decodificar imagens MPEG). A figura abaixo demonstra o grafo de comunicação entre as aplicações da MPEG. Percebe-se que existem 5 tarefas e que o padrão de comunicação segue um único fluxo de pipeline, onde uma tarefa recebe uma entrada aplica algum processamento e envia para a próxima tarefa até que o fluxo de dados chegue na tarefa OUTPUT.



Vamos criar um cenário onde a aplicação **prod_cons** seja mapeada dinamicamente no sistema e que inicie sua execução no tempo de 2 ms. Já a aplicação **mpeg** irá ser mapeada dinamicamente com exceção das tarefas **print** e **start** que irão ser direcionadas para PEs especificados pelo cenário. O mapeamento estático favorece cenários onde, por exemplo, o fluxo de entrada e saída de dados devem estar próximo a recursos externos do chip, ou seja em suas bordas, por isso iremos mapear a tarefa **start** no PE (Elemento de Processamento) 2x2 e a tarefa **print** no PE 2x0. Para criar um cenário desse tipo, edite o arquivo **my_scenario.yaml** com o seguinte conteúdo. A tag **cluster** especifica qual o cluster (conjunto de processadores) em que a aplicação será mapeada, como neste exemplo estamos trabalhando com 1 cluster somente, o valor especificado é **cluster: 0**.


```

1 apps:
2   - name: prod_cons
3   - name: mpeg       #(mandatory) name of application, must be equal to the application folder name
4     cluster: 0       #(optional) index of the statically mapped cluster - dynamic mapping by default
5     start_time_ms: 15  #(optional) any unsigned integer number - 0 by default
6     static_mapping:   #(optional) field, used to store static mapping information of tasks
7       start: [2,2]    # Task start from app mpeg will be mapped as static at address X=2, Y=2
8       print: [2,0]    # Task print from app mpeg will be mapped as static at address X=2, Y=0
9

```

apps:

```

- name: prod_cons
- name: mpeg
cluster: 0
start_time_ms: 15
static_mapping:
  start: [2,2]
  print: [2,0]

```

COMPILANDO AS APLICAÇÕES:

1. Após criar um arquivo de cenário, vamos compilar as aplicações (prod_con e mpeg) através do comando `memphis-app`. Existem duas formas de usar o comando `memphis-app`.

- (a) Passando o nome das aplicações como parâmetro

```

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

```

- (b) Passando o arquivo de cenário como parâmetro

```

memphis-app my_testcase.yaml -all my_scenario.yaml

```

A **opção (a)** permite compilar cada aplicação individualmente passando o seu nome por extenso. O comando `memphis-app` irá procurar por um diretório com este mesmo nome dentro do diretório `$MEMPHIS/HOME/testcase_example/application/` e irá compilar todos os arquivos `.c` existentes dentro desse diretório, assumindo que cada arquivo `.c` representa uma tarefa. Caso o comando não encontre nenhum diretório com esse nome ele irá procurar no diretório padrão da memphis de aplicações: `$MEMPHIS_PATH/applications`, irá copiar a aplicação para `MEMPHIS/HOME/testcase_example/application/` e irá compilar a aplicação.

A **opção (b)** faz com que todas as aplicações presentes no arquivo `my_scenario.yaml` sejam compiladas. Na prática, cada aplicação presente neste arquivo é compilada individualmente tal como na opção a). O comando em b) é apenas conveniente quando não se deseja compilar aplicação por aplicação.

PARTE 4: SIMULAÇÃO

Introdução: Nesta parte 4 do tutorial vamos simular um cenário usando o modelo de hardware e kernel previamente gerado e as aplicações previamente compiladas.

1. Execute o comando `memphis-run`

```

memphis-run my_testcase.yaml my_scenario.yaml 20

```

O comando ***memphis-run*** pede como

- **1º argumento:** arquivo de testcase (hardware)
- **2º argumento:** o arquivo de cenário (software)
- **3º argumento:** o tempo de simulação em milissegundos.

Ao executar o comando um diretório com o nome do cenário será criado dentro do diretório do testcase, o diretório do cenário contém informações de log e de depuração. O comando ***memphis-run*** inicia a simulação automaticamente pelo tempo especificado. Se a descrição do modelo especificada em `testcase_example.yaml` for 'sc' a simulação será parecida como uma execução de um arquivo `.c` normal, ou seja uma série de logs irão aparecer no terminal, semelhante à figura abaixo. **Deve ser aberta a interface gráfica de depuração:**



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
Master receiving msg
```

- Se a descrição do modelo especificada em testcase_example.yaml for 'scmod' ou 'vhdl' a comando irá chamar o simulador Questa, o qual irá carregar uma waveform com os principais sinais da Memphis e iniciar a simulação do sistema.

TERMINANDO A SIMULAÇÃO ANTES DO TEMPO ESPECIFICADO:

Para terminar uma simulação em 'sc' basta digitar em qualquer terminal o seguinte comando:

```
killall my_scenario
```

Para terminar uma simulação no Questa basta clicar no ícone de 'stop' da própria ferramenta

PARTE 5: DEPURAÇÃO

Introdução: Nesta parte 5 do tutorial vamos depurar o cenário simulado utilizando uma ferramenta gráfica desenvolvida especialmente para depuração de MPSoCs. Ela foi desenvolvida em Java. Ela também tem o costume de consumir muita memória do seu computador.

VIDEO TUTORIAL

Criei um vídeo tutorial mostrando as principais funcionalidades da ferramenta, o vídeo está no YouTube, neste link:

<https://youtu.be/nvgtvFcCc60>

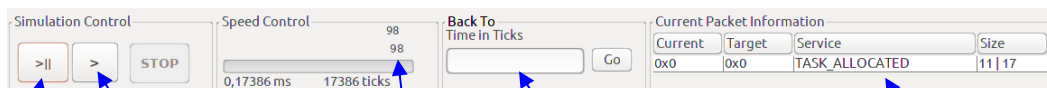
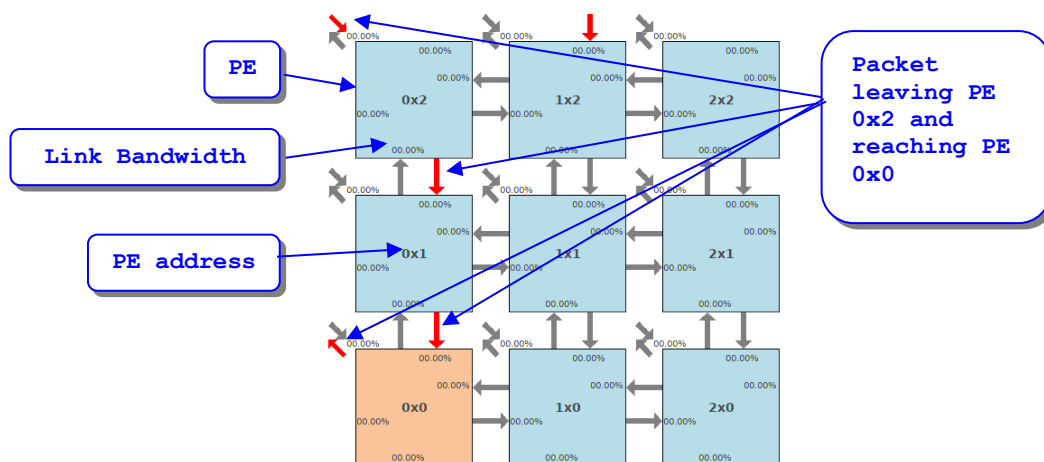
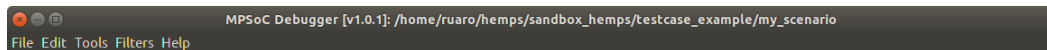
A ferramenta de depuração é aberta automaticamente após a simulação iniciar. Caso deseje abrir a ferramenta manualmente, basta digitar o seguinte comando de qualquer lugar do sistema

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

O argumento esperado pelo comando memphis-debugger é o diretório do cenário, criando dentro do *testcase*.

JANELA PRINCIPAL

A janela principal fornece uma visão geral dos pacotes que estão trafegando no sistema, bem como avaliar se houve um erro durante a execução e algum serviço. A figura abaixo detalha os principais componentes da janela principal



Advances simulation step by step for each packet hop

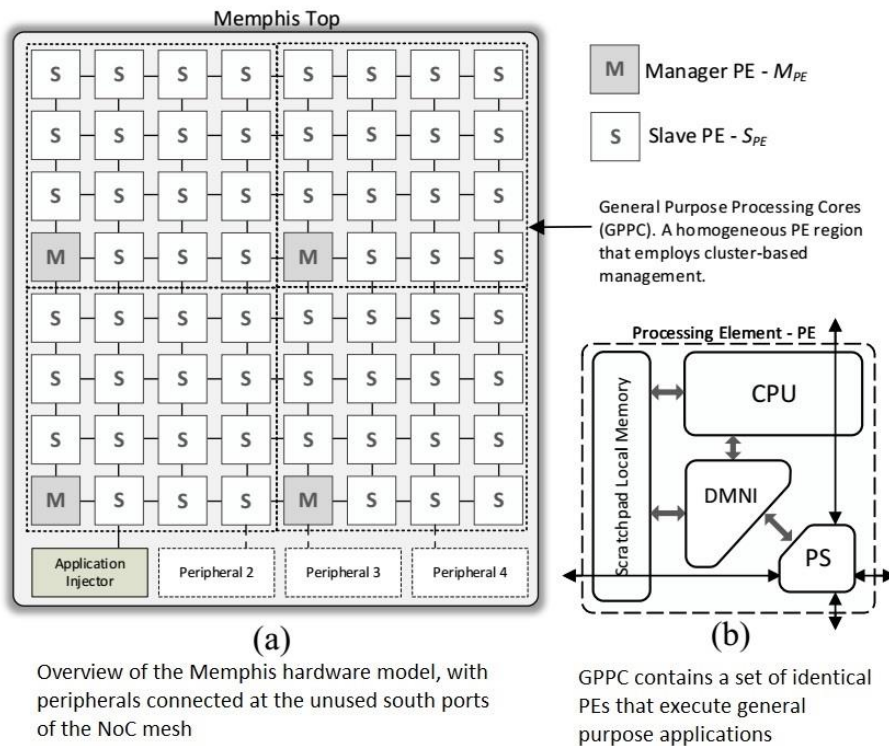
Advances simulation automatically according to the speed control bar

Used to return to a specified time at simulation

Shows packet's information

ANEXO I – INSTANCIAÇÃO DE NOVOS PERIFÉRICOS

Introdução: Este tutorial irá explicar como instanciar novos periféricos na Memphis, ele irá abordar as duas descrições de hardware (SystemC e VHDL). Periféricos são componentes de hardware que podem ser conectados nas bordas do MPSoC. A arquitetura da Memphis é dividida em duas regiões (vide Figura abaixo):



- **GPPC:** *General Purpose Processing Cores*, que consiste na região interna do chip, composta por PE homogêneos e dedicada a execução de aplicações.
- **Peripherals:** Região nas bordas do chip, que permite a implementação de periféricos que implementam serviços de aceleração de hardware e interface I/O.

1. O **primeiro** passo para instanciar um novo periférico e fazer com que ter sua implementação descrita em SystemC (Atualmente somente SystemC é suportado). A interface do periférico é uma interface padrão Hermes, com os seguintes sinais:

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)
    
```

Essa interface implementa o protocolo de controle de fluxo baseado em créditos, mais detalhes podem ser obtidos nos materiais que explicam a NoC Hermes.

VHDL

- Com o periférico já implementado e com sua interface de acordo com um roteador da NoC Hermes, o próximo passo consiste em abrir o arquivo Testbench e instanciar o periférico lá:

- Editar o arquivo **hardware/vhdl/test_bench.vhd** a fim de instanciar o periférico e conectá-lo ao GPPC.
- Criar os sinais que irão ligar o periférico ao GPPC (Memphis)

```

43     signal clock          : std_logic := '0';
44     signal reset          : std_logic;
45
46     -- IO signals connecting App Injector and Memphis
47     signal memphis_injector_tx      : std_logic;
48     signal memphis_injector_credit_i : std_logic;
49     signal memphis_injector_data_out : regflit;
50
51     signal memphis_injector_rx      : std_logic;
52     signal memphis_injector_credit_o : std_logic;
53     signal memphis_injector_data_in  : regflit;
54
55     -- Create the signals of your IO component here:
56

```

- Instanciar o Periférico

```

59     -- Peripheral 1 - Instantiation of App Injector
60     App_Injector : entity work.app_injector
61     port map(
62         clock      => clock,
63         reset      => reset,
64
65         rx          => memphis_injector_tx,
66         data_in     => memphis_injector_data_out,
67         credit_out  => memphis_injector_credit_i,
68
69         tx          => memphis_injector_rx,
70         data_out    => memphis_injector_data_in,
71         credit_in   => memphis_injector_credit_o
72     );
73
74     -- Peripheral 2 - Instantiate your IO component here:
75
76

```

- Conectar o Periférico ao GPPC da Memphis

```

80     Memphis : entity work.Memphis
81     port map(
82         clock      => clock,
83         reset      => reset,
84
85         -- Peripheral 1 - App Injector
86         memphis_app_injector_tx      => memphis_injector_tx,
87         memphis_app_injector_credit_i => memphis_injector_credit_i,
88         memphis_app_injector_data_out => memphis_injector_data_out,
89
90         memphis_app_injector_rx      => memphis_injector_rx,
91         memphis_app_injector_credit_o => memphis_injector_credit_o,
92         memphis_app_injector_data_in  => memphis_injector_data_in
93
94         -- Peripheral 2 - Connect your IO component to Memphis here:
95
96     );
97

```

3. Editar o arquivo **hardware/vhdl/memphis.vhd** a fim de conectar a interface do periférico com o roteador

- Inserir no *portmap* a interface com o periférico

```

22 entity Memphis is
23 port(
24     clock           : in  std_logic;
25     reset           : in  std_logic;
26
27     -- IO interface - App Injector
28     memphis_app_injector_tx      : out std_logic;
29     memphis_app_injector_credit_i : in  std_logic;
30     memphis_app_injector_data_out : out regflit;
31
32     memphis_app_injector_rx      : in  std_logic;
33     memphis_app_injector_credit_o : out std_logic;
34     memphis_app_injector_data_in  : in  regflit
35
36     -- IO interface - Create the IO interface for your component here:
37 );
38
39 end;
40

```

- Conectar a interface do periférico com o roteador

```

152 --IO App Injector connection
153 memphis_app_injector_tx <= tx(APP_INJECTOR)(io_port(i));
154 memphis_app_injector_data_out <= data_out(APP_INJECTOR)(io_port(i));
155 credit_i(APP_INJECTOR)(io_port(i)) <= memphis_app_injector_credit_i;
156
157 rx(APP_INJECTOR)(io_port(i)) <= memphis_app_injector_rx ;
158 memphis_app_injector_credit_o <= credit_o(APP_INJECTOR)(io_port(i));
159 data_in(APP_INJECTOR)(io_port(i)) <= memphis_app_injector_data_in;
160
161 end generate;
162
163 --Insert the IO wiring for your component here:
164

```

4. Editar o arquivo de makefil: **build_env/makes/make_vhdl**

- Adicionar o nome do arquivo .cpp que possui a implementação do seu periférico

```

11 MEMPHIS_PKG =memphis_pkg
12 STAND      =standards
13 TOP        =memphis test_bench
14 IO         =app_injector meu_periferico
15 PE         =pe
16 DMNI       =dmni
17 MEMORY     =mem_testbench_mem_block

```

SYSTEMC

5. Com o periférico já implementado e com sua interface de acordo com um roteador da NoC Hermes, o próximo passo consiste em abrir o arquivo Testbench e instanciar o periférico lá:

- Editar o arquivo **hardware/sc/test_bench.h** a fim de instanciar o periférico e conectá-lo ao GPPC.
- Criar os sinais que irão ligar o periférico ao GPPC (Memphis)
- Instanciar o Periférico

```

46  app_injector * io_app;
47
48  char aux[255];
49  FILE *fp;
50
51  SC_HAS_PROCESS(test_bench);
52  test_bench(sc_module_name name_, char *filename_ = "output_master.txt") :
53  sc_module(name_), filename(filename_)
54  {
55      fp = 0;
56
57      MPSoC = new memphis("Memphis");
58      MPSoC->clock(clock);
59      MPSoC->reset(reset);
60      MPSoC->memphis_app_injector_tx(memphis_injector_tx);
61      MPSoC->memphis_app_injector_credit_i(memphis_injector_credit_i);
62      MPSoC->memphis_app_injector_data_out(memphis_injector_data_out);
63      MPSoC->memphis_app_injector_rx(memphis_injector_rx);
64      MPSoC->memphis_app_injector_credit_o(memphis_injector_credit_o);
65      MPSoC->memphis_app_injector_data_in(memphis_injector_data_in);
66
67
68      io_app = new app_injector("App_Injector");
69      io_app->clock(clock);
70      io_app->reset(reset);
71      io_app->rx(memphis_injector_tx);
72      io_app->data_in(memphis_injector_data_out);
73      io_app->credit_out(memphis_injector_credit_i);
74      io_app->tx(memphis_injector_rx);
75      io_app->data_out(memphis_injector_data_in);
76      io_app->credit_in(memphis_injector_credit_o);
77
78      //Instantiate your IO component here
79      //...
80
81      SC_THREAD(ClockGenerator);

```

- Conectar o Periférico ao GPPC da Memphis

```

60  MPSoC->memphis_app_injector_tx(memphis_injector_tx);
61  MPSoC->memphis_app_injector_credit_i(memphis_injector_credit_i);
62  MPSoC->memphis_app_injector_data_out(memphis_injector_data_out);
63  MPSoC->memphis_app_injector_rx(memphis_injector_rx);
64  MPSoC->memphis_app_injector_credit_o(memphis_injector_credit_o);
65  MPSoC->memphis_app_injector_data_in(memphis_injector_data_in);
66
67
68  io_app = new app_injector("App_Injector");
69  io_app->clock(clock);

```

6. Editar o arquivo hardware/sc/memphis.h a fim de conectar a interface do periférico com o roteador

- Inserir no *portmap* a interface com o periférico

```

36  //IO interface - App Injector
37  sc_out< bool >      memphis_app_injector_tx;
38  sc_in< bool >       memphis_app_injector_credit_i;
39  sc_out< regflit >   memphis_app_injector_data_out;
40
41  sc_in< bool >       memphis_app_injector_rx;
42  sc_out< bool >      memphis_app_injector_credit_o;
43  sc_in< regflit >    memphis_app_injector_data_in;
44
45  //IO interface - Create the IO interface for your component here:
46
47

```

- Conectar a interface do periférico com o roteador

```

94 SC METHOD(pes_interconnection);
95     sensitive << memphis_app_injector_tx;
96     sensitive << memphis_app_injector_credit_i;
97     sensitive << memphis_app_injector_data_out;
98     sensitive << memphis_app_injector_rx;
99     sensitive << memphis_app_injector_credit_o;
100    sensitive << memphis_app_injector_data_in;
101    for (j = 0; j < N_PE; j++) {
102        for (i = 0; i < NPORT - 1; i++) {
103            sensitive << tx[j][i];
104            sensitive << data_out[j][i];
105            sensitive << credit_i[j][i];
106            sensitive << data_in[j][i];
107            sensitive << rx[j][i];
108            sensitive << credit_o[j][i];
109        }
110    }
111 }
112 };
113

```

- Abrir o arquivo memphis.cpp para implementar a conexão da interface do periférico com o roteador

```

156 //IO Wiring (Memphis <-> IO) -----
157 if (i == APP_INJECTOR && io_port[i] != NPORT) {
158     p = io_port[i];
159     memphis_app_injector_tx.write(tx[APP_INJECTOR][p].read());
160     memphis_app_injector_data_out.write(data_out[APP_INJECTOR][p].read());
161     credit_i[APP_INJECTOR][p].write(memphis_app_injector_credit_i.read());
162
163     rx[APP_INJECTOR][p].write(memphis_app_injector_rx.read());
164     memphis_app_injector_credit_o.write(credit_o[APP_INJECTOR][p].read());
165     data_in[APP_INJECTOR][p].write(memphis_app_injector_data_in.read());
166 }
167 //Insert the IO wiring for your component here

```

7. Editar o arquivo de makefil: build_env/makes/make_systemc e build_env/makes/make_systemc_mod

- Adicionar o nome do arquivo .cpp que possui a implementação do seu periférico

```

12 #SystemC files
13 TOP      =memphis test_bench
14 IO       =app_injector
15 PE       =pe
16 DMNI     =dmni
17 MEMORY   =ram
18 PROCESSOR =mlite_cpu
19 ROUTER   =queue switchcontrol router_cc
20

```

- Os próximos passos são independentes de VHDL ou SystemC. Uma vez que as modificações nos arquivos de código fonte estejam concluídas, o próximo passo é especificar no arquivo YAML do testcase o nome do periférico e a posição onde ele será conectado ao GPPC.

ATENÇÃO: O nome do periférico deve ser o mesmo nome usado no código fonte pra referenciá-lo, no caso do AppInjector, o hardware está utilizando a macro APP_INJECTOR, logo o testcase desse especificar o nome APP_INJECTOR no espaço do testcase destinado a descrição do periférico.

```

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

```


Ao criar um periférico no testcase, tal como feito para o APP_INJECTOR a macro APP_INJECTOR fica visível para os arquivos de hardware.

Essa macro também fica visível no kernel slave e mestre. Estes kernels devem utilizar a macro APP_INJECTOR para preencher qualquer cabeçalho de pacote que o kernel deseja enviar para o AppInjector. Veja este exemplo do kernel_master.c onde ele enviar um pacote de APP_ALLOCATION_REQUEST para o periférico AppInjector, na linha 158 o kernel utiliza a macro APP_INJECTOR para preencher o conteúdo do campo p->header (ou seja, o cabeçalho do pacote):

```
150  /** Assembles and sends a APP_ALLOCATION_REQUEST packet to the global master
151      *  \param app The Application instance
152      *  \param task_info An array containing relevant task informations
153      */
154  void send_app_allocation_request(Application * app, unsigned int * task_info){
155
156      ServiceHeader *p;
157
158      p = get_service_header_slot();
159
160      p->header = APP_INJECTOR;
161
162      p->service = APP_ALLOCATION_REQUEST;
```