Simics® Simulator Educational Workshop Two

Building an accelerator model

Version 1.0

1 Introduction

This Intel® Simics® simulator educational workshop goes through Simics model building. The workshop is built around a simple Mandelbrot fractal computation engine and shows how this is used in successively more complex system setups. The first step uses just the compute block in isolation, while in the end, a complete accelerator is connected over PCIe to a Simics Quick-Start Platform (QSP) model. The accelerator can be used to perform computations in parallel as seen from the virtual system, as well as running in parallel on the host to optimize simulation performance.

The workshop is intended to be worked through in order – later steps often depend on the results from previous steps.

1.1 Host type

This workshop is written assuming you are working on a Linux host, but all steps should work the same on a Windows host.

1.2 Conventions

The following conventions are used in this workshop instruction:

Actions from the shell on the host (the machine that runs Simics) are indicated by **$**. The commands to type are in **bold**:

```
$ ls
```

Commands for the Simics simulator command line are indicated by **simics>**:

```
simics> help
```

When the Simics simulator is running, the prompt changes to **running>**. Most command-line commands can be used while the simulator is running:

```
running> ptime
```

Actions from the shell on the target (the simulated machine) are also indicated by **$** or **#**. The instructions will indicate that this is to be entered on the target system. The target system prompt is only used while Simics is running, so it should be fairly clear when the target console is used and when the host shell is used.

```
# lspci
```

Output from commands (in any environment) are shown as a box with slightly smaller text, and in regular font. Typically, the command used to generate the output is not shown in the box:

```
Status of sim [class sim]
=========================

Environment:
        Hide Console Windows : No

Simulation Engine:
                Page Sharing : Disabled
      Multithreading enabled : Enabled
                Thread limit : Unlimited
        Worker threads limit : 1
    Simulation threads limit : 0
        CPU module load mode : normal
          Image memory usage : Limited to 22.24 GB
      Image memory limit hit : 0 times
...
```

The Simics simulator scripts used to start simulation runs are all named with a prefix number (for example, **`001-`**). Most of the time, the workshop instructions only refer to the scripts by their number. It is typically possible to tab-complete the full file name once the number has been entered on the console.

1.3 Translating the instructions to Windows hosts

In general, the Simics simulator tries to hide the differences between Windows and Linux host once the simulator is running. However, outside command-line work will be different. Here are the most noticeable differences.

On Windows, use "**`simics.bat`**" to start a new Simics simulator sessions instead of "**`./simics`**":

```
C:\...> simics.bat -v
```

Note that the Simics simulator core allows the use of Linux-style paths on Windows hosts. Thus, you can start Simics on Windows using Linux-style script names that you copy from the instructions in this workshop. For example:

```
C:\...> simics.bat ./simics targets/workshop-02/006-accelerator-in-
qsp.simics
```

Tab completion from CMD will produce **`\`**-separated paths, and these also work. Tab-completing paths from the simulator command line will also produce native Windows paths.

On Windows, **`make`** is present as a script in the project bin directory:

```
C:\...> bin\make
```

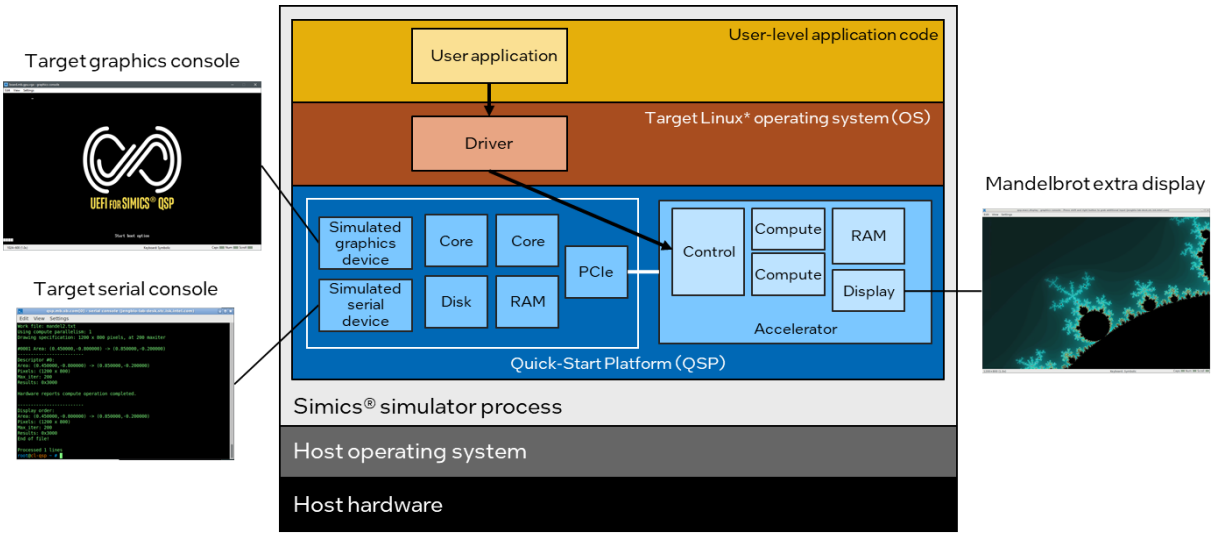
The **`project-setup`** script is in the project **`bin`** on both host types. It takes the same arguments. The path to invoke it contains a backslash instead of forward-slash:

```
C:\...> bin\project-setup
```

1.4 Overview of the target setup

The target system used in this workshop is the Simics Quick-Start Platform (QSP), extended with a custom Mandelbrot accelerator PCI-express (PCIe) add-in card. The QSP runs the standard Clear Linux* image, and the disk image provided contains both a device driver and user-level application that drives the hardware accelerator.

The system overall looks like this:



*Other names and brands may be claimed as the property of others

The standard QSP hardware is on the left, with the accelerator on the right. When running, the accelerator is logically part of the QSP overall platform since it is inserted as a virtual PCIe card in a virtual PCIe slot inside the QSP virtual platform.

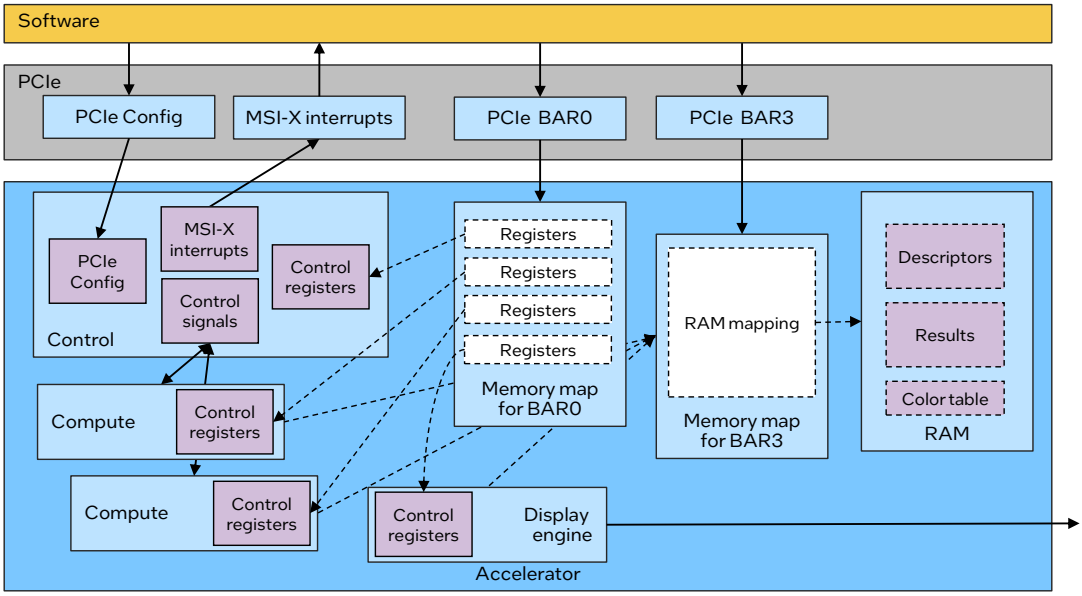
The target graphics console is not used in this workshop.

The target serial console is used to interact with the target Linux.

The Mandelbrot extra display is used to see the results computed by the accelerator.

1.5 Accelerator design

The accelerator looks like this in more detail:



The **control unit** contains the main control registers for the accelerator, as well as the PCIe interface. It manages computation jobs in the compute units via a set of signals to and from the compute units.

Each **compute unit** has a few memory-mapped control registers, as well as control signals towards the control unit. They access the on-accelerator RAM via the "memory" memory map (also available via BAR3) to read descriptors and write results.

The **display unit** displays the results to an external display, using the results and color table in on-accelerator RAM.

When connected to PCIe, the control registers for the control, compute, and display units are mapped through BAR0. The local RAM is mapped as a memory map through BAR3. BAR1 and BAR2 are used to manage the MSI-X interrupts from the accelerator. The complete PCIe interface logic is part of the control unit.

The entire accelerator can also be used without a PCIe connection and without external software, driving it from the Simics simulator command line interface and test scripts.

1.6 Work descriptors

The compute units work from a descriptor held in local memory.

OFFSET	SIZE	MEANING
0x00	4	Bottom (fixed-point)
0x04	4	Left (fixed-point)
0x08	4	Top (fixed-point)
0x0c	4	Right (fixed-point)
0x10	4	Width (pixels)
0x14	4	Height (pixels)
0x18	4	Maximum iterations (really just a 16-bit number)
0x1c	4	Reserved to align the next field
0x20	8	Address of results area

The floating-point values used to encode the drawing area are stored using a 32-bit fixed-point format. The encoding covers the range -2.0 to +2.0, using the formula:

```
Descriptor_value = uint32( (floating_point_value * 0x4000_0000) + 0x8000_0000)
```

The results are stored as an array of 16-bit integers, one per pixel, storing the iteration count for that particular pixel (up to max iterations).

The addresses used by the compute units and other devices are all expressed as offsets in the local memory map (for BAR3). It does not matter where the accelerator memory spaces are mapped in the rest of the system, all addresses are strictly local as used by the hardware.

2 Basic Preparations

2.1 Installing the Simics software and setting up a project

This workshop assumes that you have:

- Installed the public release of the Intel Simics simulator, including all its packages.
- Created a Simics project using the Simics Package Manager. This location will be referred to as **[project]** for the rest of this workshop.
    - *(The default location used by the Simics Package Manager is **~/simics-projects/my-simics-project-1/**, but you can use any location.)*

It is also recommended that you have:

- Activated virtualization for simulation acceleration (VMP), as that makes the simulation run quite a bit faster for many use cases.
- Worked through the getting started tutorial in the Simics simulator documentation to get an idea for basic interaction with Simics.

On a Windows host, make sure to install a MinGW gcc in order to be able to build Simics models.

2.2 Build the compute device model from source code

To ascertain that your project works for model building, copy the source code for the **m_compute** device model to your project and run its unit tests.

1. Go to the host shell, in your Simics project. You can do this from a shell you start yourself or use the Simics Package Manager functionality to start a new shell.
2. Use **project-setup** to copy the **m-compute** module to your project:

   ```
   $ bin/project-setup  --copy-module  m-compute
   ```

   This will copy the given device source code to your project.
3. Check that the module appeared in **[*project*]/modules/**.

   ```
   $ ls modules
   ```

   The output should look like this:

   ```
   $ ls modules/
   m-compute
   ```
4. Build the module:

   ```
   $ make
   ```

   Note that on Windows host, you have to do "**bin\make**" due to how the **make** program is located by the Simics simulator build infrastructure.

The build process will show the build steps. It will look something like this:

```
=== Building module m-compute ===
GEN     module_id.c
DEP     module_id.d
DML-DEP m-compute.dmldep
DEP     m-compute-dml.d
PYC     module_load.pyc
DMLC    m-compute-dml.c
CC      m-compute-dml.o
CC      module_id.o
CCLD    m-compute.so
```

2.3 Find the scripts in the simulator installation

Many exercises in this workshop involve looking at the source code of Simics command-line and Python scripts. These scripts are located in the **Simics simulator installation** and are not found in the Simics project – except for trampoline scripts that call the **`.simics`** files in the installation.

To inspect the files, use an editor to open the files from within the installation. To find the installation location of the **`[simics]/targets/workshop-02/`** directory, use a script that reveals the location of the installation.

5. From the shell in your Simics project, start a new Simics simulation with the script **`000-find-simics-installation.simics`** from your local project:

   ```
   $ ./simics targets/workshop-02/000-find-simics-installation.simics
   ```

   When the script runs, it will print the location, something like this:

   ```
   $ ./simics targets/workshop-02/000-find-simics-installation.simics
   Intel Simics 6 (build 6122 linux64) Copyright 2010-2021 Intel Corporation

   Use of this software is subject to appropriate license.
   Type 'copyright' for details on copyright and 'help' for on-line documentation.

   You can find the script files used in Workshop 02 here:

   /disk1/simics-6-install/simics-training-6.0.pre25/targets/workshop-02
   ```

   This technique works since the **`[project]/targets/workshop-02/000-find-simics-installation.simics`** file in your project is a trampoline that points at the actual installation location. That trampoline was set up when the project was created (and it will be updated each time **`bin/project-setup`** is run).

6. Check the actual sets of scripts that were run when the simulator was started:

   ```
   simics> command-file-history -v
   ```

   To see how the trampoline script in the local project is run first, then it starts the same-name file in the installation, and finally both scripts exit.

2.4 Set up the default number output format

The simulator command line maintains a global preference for how to display numeric values, known as the output radix. The **`output-radix`** command is used to specify the base radix that used for display as well as controlling the grouping of digitals for numbers

7. To set the output to hexadecimal with 4-digit grouping:

```
simics> output-radix 16 4
```

8. Test it by entering some numbers:

```
simics> 0xdeadbeef
```

   Which should result in an output with digit grouping applied:

```
0xdead_beef
```

9. Try a decimal number:

```
simics> 100000
```

   Which should be converted to a hexadecimal number:

```
0x0001_86a0
```

10. Note that the output radix setting doesn't affect output from inline Python.

```
simics> @10000
```

    Should display as:

```
10000
```

11. Not all commands use the default number format for their output. For example, addresses tend to be printed in hexadecimal and time in decimal formats regardless of the default settings. Simics maintains a digit grouping setting for each base.

    Check the digit grouping settings:

```
simics> digit-grouping
```

    The output is something like this (your settings might vary):

Radix	Digits
2	8
8	0
10	3
16	4

12. Test the format of decimal numbers using the **`dec`** command (which prints its argument in decimal):

```
simics> dec 1000000
```

    With a digit grouping of three for base 10, this should look like:

```
"1_000_000"
```

13. Set base 10 as the default format, with a grouping of three:

```
simics> output-radix 10 3
```

14. Test the formatting:

```
simics> 0xdeadbeef
```

Which should result in:

```
3_735_928_559
```

15. To use the same current output base and digit groupings in your next Simics session, use the command **save-preferences**:

```
simics> save-preferences
```

It makes sense to leave the default at 10 for the time being, but if you want to see most numbers in hexadecimal, you can change that.

16. Check the preference values:

```
simics> list-preferences
```

Resulting in:

```
[...]
            output_radix: 10
         output_grouping: [8, 0, 3, 4]
[...]
```

17. Quit this Simics session.

```
simics> quit
```

2.5 Open the Simics simulator documentation

The Simics simulator comes with extensive documentation. It can be opened from inside the Simics Package Manager view of your project. It can also be opened from the host shell in the project.

18. From the shell in your Simics project, bring up the documentation using the **documentation** script (**documentation.bat** on a Windows host):

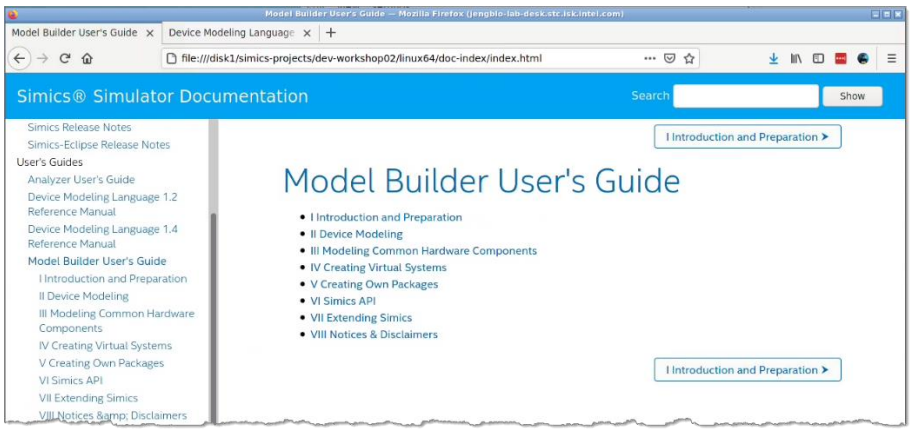
```
$ ./documentation &
```

This will open a web browser with the documentation. Keep this window around.

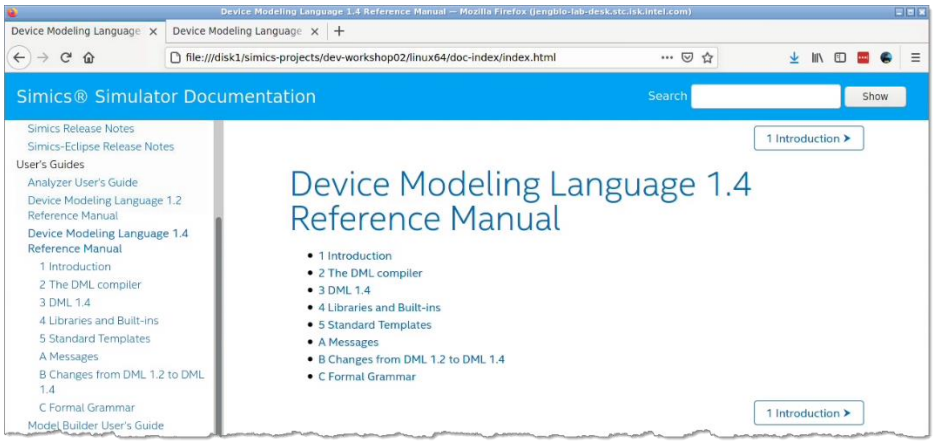
It is a good idea to open interesting parts of the documentation in their own tabs in the browser, that makes it easier to get back to them.

2.6 Important reference points in the documentation

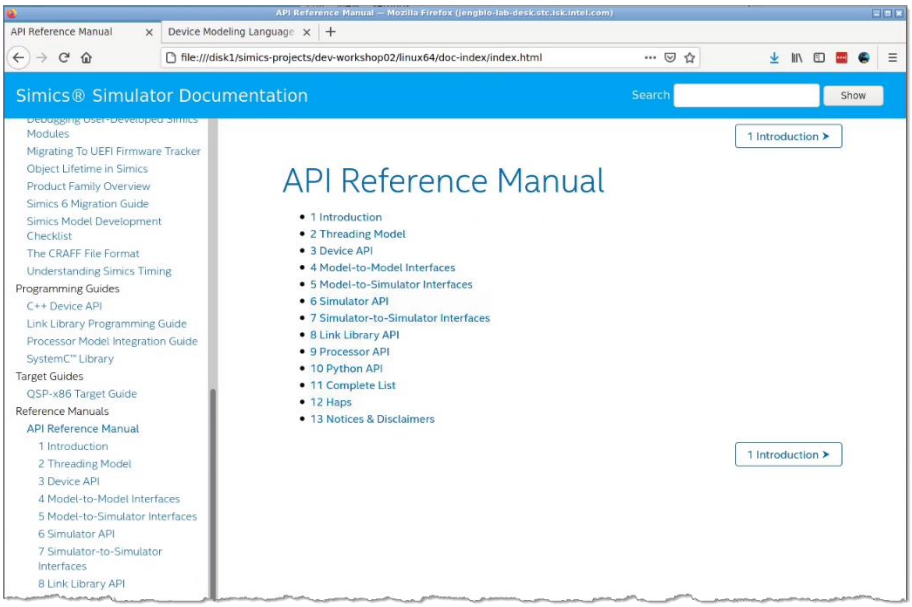
19. The **Model Builder User's Guide.** This document contains introductory and overview information about how to build Simics models.



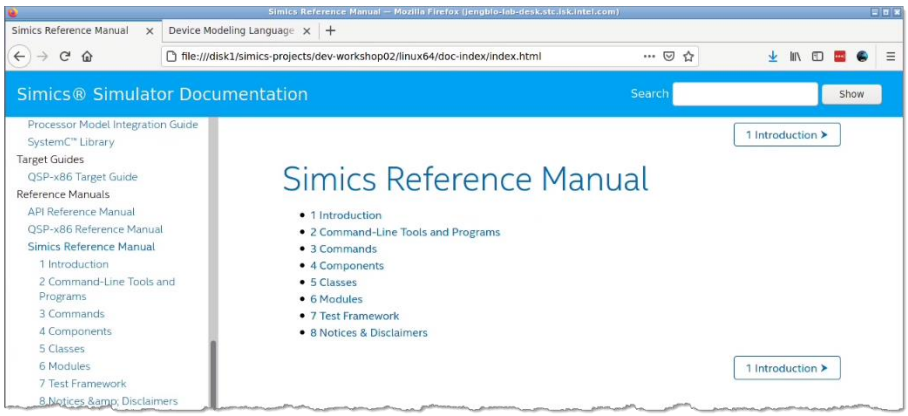
20. The **Device Modeling Language 1.4 Reference Manual**. This is the main source of information about the Device Modeling Language (DML) and the libraries and standard templates provided.



21. The **API Reference Manual**. This manual provides information on the standard interfaces used to build Simics models. Both between the models and the simulator cores and models and other models.



22. The **Simics Reference Manual**. This manual contains information on executable tools used in the model-building process, as well as some frameworks like the model test framework.



3 Quickly Test the Complete Setup

Start by running the complete system with software and everything, to see how it all fits together.

3.1 Start the Simics simulator

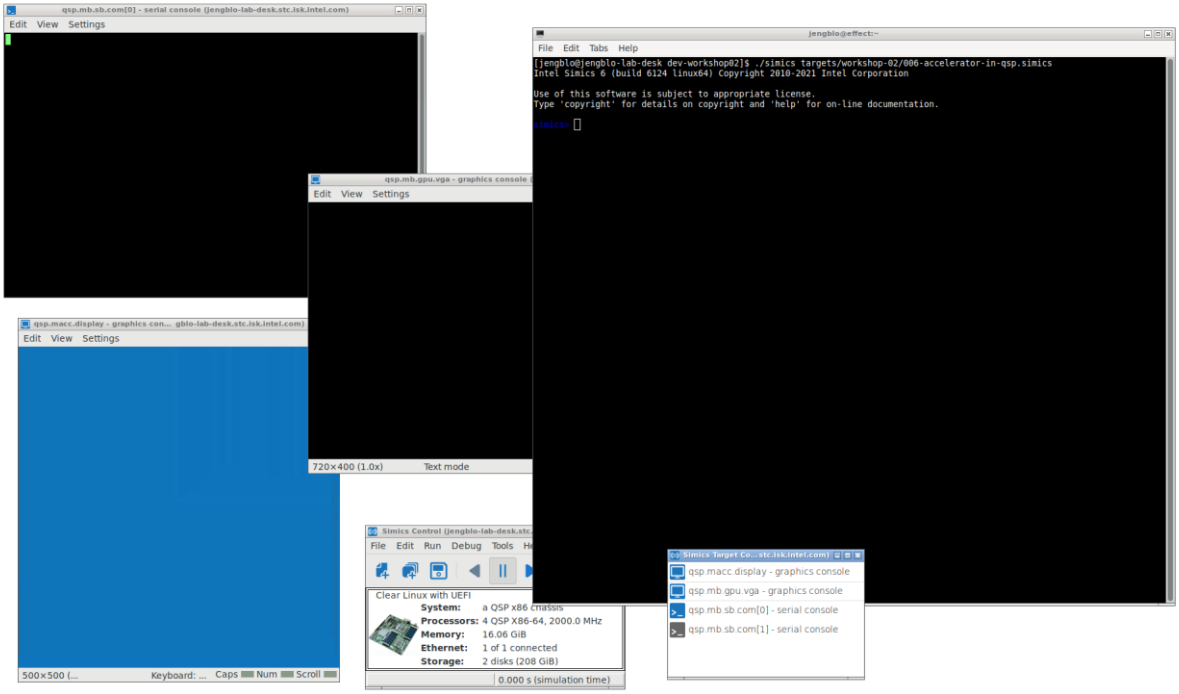
1. Start a new simulation session using the script **006-**:

```
$ ./simics targets/workshop-02/006-accelerator-in-qsp.simics
```

When the Simics simulator has started, you see a number of windows:

- The host shell you started **simics** from.
- The target system serial console (**qsp.mb.sb.com[0]**)
- The target system graphics console (**qsp.mb.gpu.vga**)
- The Mandelbrot accelerator display graphics console (**qsp.macc.display**)
- The target console control window
- The simple Simics Control window

It will look something like this:



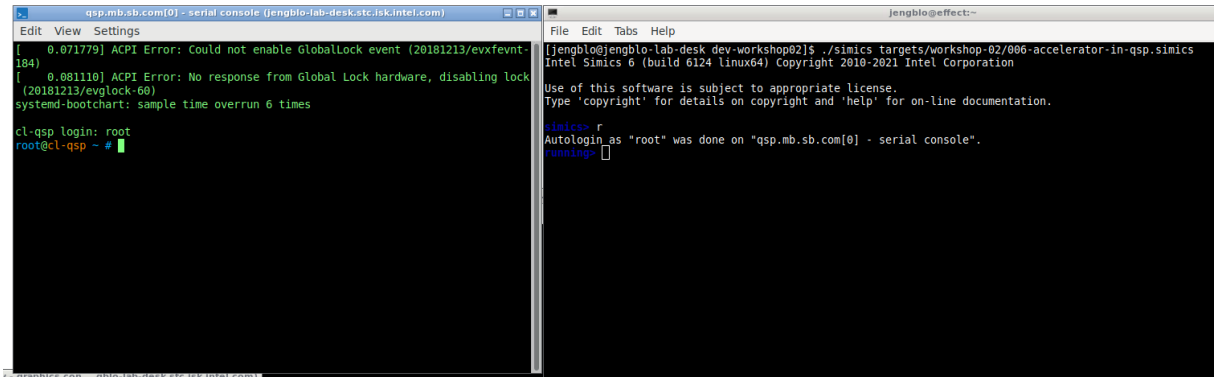
3.2 Boot the target system

2. Run the simulation:

```
simics> r
```

The graphics console will show the UEFI splash screen and then the target Linux login prompt. The serial console will (eventually) show "root" logged in. There will be some messages about ACPI errors during the boot. They are harmless.

3. Wait for the target system serial console to show that root has logged in.



3.3 Activate device driver and display a Mandelbrot

4. Go to the target system serial console, and list the available files:

```
# ls
```

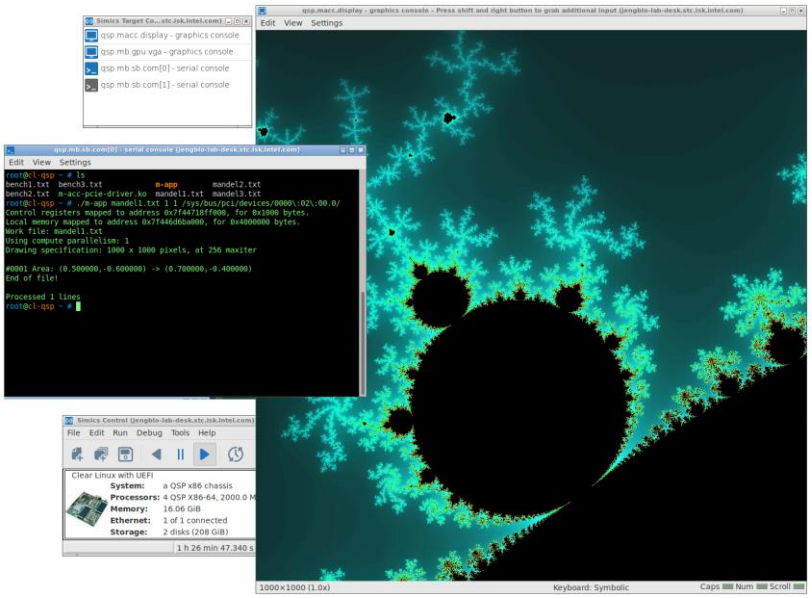
5. Use **insmod** to load the driver file **m-acc-pcie-driver.ko**.

```
# insmod m-acc-pcie-driver.ko
```

6. Run the **m-app** program to display a first Mandelbrot fractal. The first argument to the program is the file to use to specify what to draw, the second argument is the level of parallelism in the rendering, the third the verbosity, and the fourth the path to the device node for the accelerator on the PCI bus. The last argument is crucial as it lets the program memory-map both the register and memory BARs.

```
# ./m-app mandel1.txt 1 1 /sys/bus/pci/devices/0000\:02\:00.0/
```

The result should look something like this:



3.4 Investigate the PCIe setup

7. Check the PCI devices in the system using **lspci**:

```
# lspci
```

Towards the end of the lists there should be listing showing:

```
02:00.0 Processing accelerators: Intel Corporation Device 0d5f (rev 02)
```

The device is not identified by name, since it is not part of the PCI hardware ID list that is part of the target Linux system (see **/usr/share/hwdata/pci.ids**).

8. Check that the accelerator PCI ID is the same. Go to the Simics simulator command line in the shell window, and list the registers of the PCI configuration bank of **qsp.macc.control**. Set the output format to hexadecimal to make the values easier to read:

```
running> output-radix 16
running> print-device-regs qsp.macc.control.bank.pci_config
```

Which shows something like:

```
running> print-device-regs qsp.macc.control.bank.pci_config
Offset  Name                    Size        Value | Offset  Name                    Size   Value
-------------------------------------------------+-----------------------------------------
0x0000  vendor_id                  2       0x8086 | 0x0072  msix_control             2  0x8001
0x0002  device_id                  2       0x0d5f | 0x0074  msix_table               4  0x0001
0x0004  command                    2       0x0407 | 0x0078  msix_pba                 4  0x0002
0x0006  status                     2       0x0010 | 0x0080  exp_capability_header    2  0x0010
0x0008  revision_id                1       0x0002 | 0x0082  exp_capabilities         2  0x0002
…
```

Note the value of the **device_id**, **vendor_id**, and **revision_id** registers matching the output from **lspci** on the target system.

3.5 Look inside the accelerator subsystem

Use the command line to inspect the contents of the accelerator subsystem.

9. The **list-objects** command can be used to show the hierarchical setup of the subsystem. The **-tree** option shows a tree, and **namespace=** shows only a part of the system. Hiding the port objects with **-hide-port-objects** makes the output focus on the most important objects of the simulation

```
running> list-objects namespace=qsp.macc -tree -hide-port-objects
```

The output should look like this:

```
┐
├ compute[0..7]
├ console
├ control
├ display
├ gcon ┐
│      ├ abs_mouse
│      ├ con ┐
│      │     ├ frontend
│      │     ├ tcp
│      │     └ unix_socket
│      ├ device
│      ├ keyboard
│      └ mouse
├ local_memory
├ pci
├ ram ┐
│    └ image
└ register_memory
```

10. To also see the register banks and ports of the objects:

```
running> list-objects -tree namespace = qsp.macc
```

11. To get more information about the type and other properties of a particular object, use the **help** command on the object. For example:

```
running> help qsp.macc.control
```

3.6 Create your own Mandelbrot specification

The target application renders fractals from a "work order" file. To draw a custom Mandelbrot fractal, create your own such file and upload it to the target system, using the Simics Agent back-door system.

12. Check what a file looks like. In the **target serial console**, check the contents of **mandel1.txt** on the target system:

```
# more mandel1.txt
```

13. In a text editor, create a new file in your Simics project. Call it **[*project*]/mandel4.txt**.

14. Copy and paste the contents of **mandel1.txt** from the target system console into **mandel4.txt**.

    The first two numbers are the width and height of the area to plot, in pixels. The last number is the maximum number of iterations to use before considering that a point escapes the set.

    The floating-point numbers on the second line are bottom, left, top, right. You need to make the width/height ratio of the plot area specification match that of the pixel size to get a nice-looking picture.

15. Modify the contents of **mandel4.txt** to indicate a different area. For example:

```
1000 1000 256
0.14 -0.75 0.16 -0.73
```

16. Go to the Simics simulator command line prompt. Start the Simics Agent manager:

```
running> start-agent-manager
```

17. Create a new connection to the target system:

```
running> agent_manager.connect-to-agent
```

    You should see a printout like:

```
matic0 connected to cl-qsp0 (0x1b90f02e4ca081fb)
```

    The connect operations works since the target-side agent software is already running on the target system (it is started on boot in the default QSP setup).

18. Check for the running agent. Go to the target system serial console:

```
# ps -x | grep agent
```

    There should be a **/usr/bin/simics-agent** process running.

19. Go back to the Simics simulator command line and use the Simics Agent **upload** command to upload the **mandel4.txt** file to **/root/** on the target. The connection to the target is called **matic0**, as indicated by the connection message above.

```
running> matic0.upload mandel4.txt /root/
```

20. In the target serial console, check that the file **mandel4.txt** appeared:

```
# ls
```

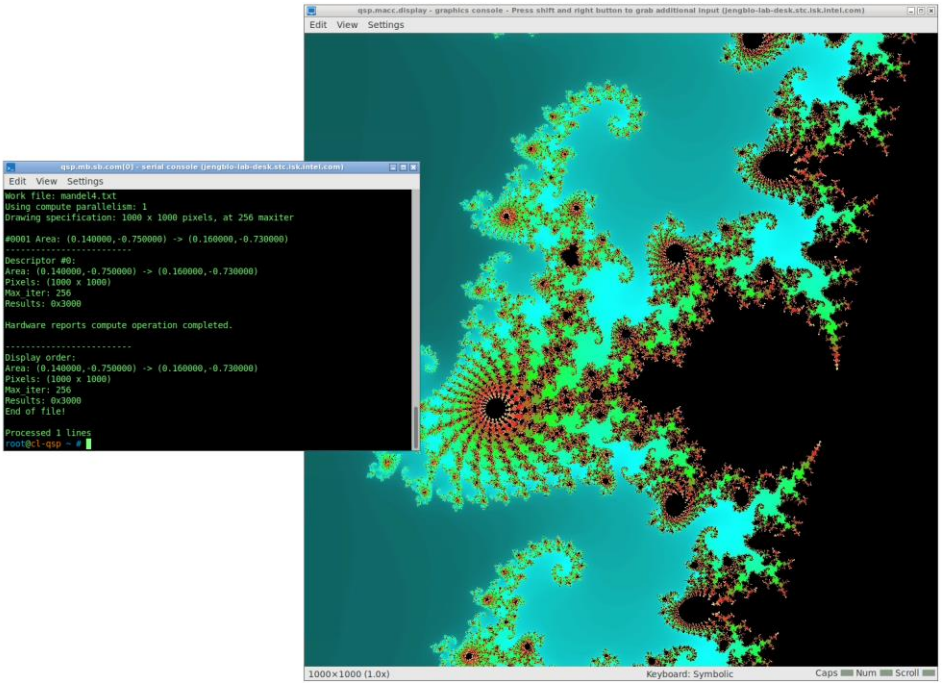
21. On the simulator command line, raise the log level of the **qsp.macc** subsystem to 2 to get basic information about what the model is doing.

```
running> log-level qsp.macc 2 -r
```

22. Run the **m-app** program on the file, using verbosity 2 to see the maximum amount of information from the application:

```
# ./m-app mandel4.txt 1 2 /sys/bus/pci/devices/0000\:02\:00.0/
```

You should see a new fractal displayed in the accelerator display window:



23. On the simulator command line, a number of **`info`** log messages will be printed that show the steps taken in the hardware model. Check that the descriptor seen by the hardware is the same as that specified by the **`mandel4.txt`** file, by looking at the log message printed when the computation is started:

```
…
[qsp.macc.compute[0].port.control_in info] Received request to start compute job
[qsp.macc.compute[0] info] Work descriptor read. Area: (0.140000, -0.750000) -
(0.160000, -0.730000) (1000, 1000) iter: 256 pixels  @ 0x3000
[qsp.macc.compute[0] info] Compute operation time: 0.0100000000 s
…
```

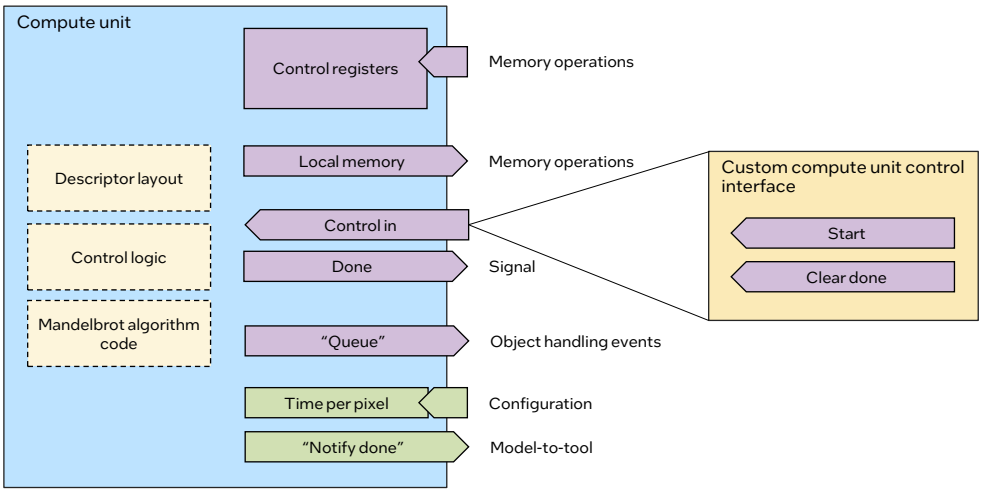
24. Exit this simulation session.

```
running> exit
```

4 Run the Compute Unit on its Own

This section zooms in on a single compute unit. The goal is to understand how to interact with a device model from the Simics simulator command line, to test its basic functionality and interfaces to the outside world.

The overall connectivity and design of the compute unit looks like this:



The external interface of the compute unit contains the following:

- Some control registers, that are accessed via memory operations. One register holds the address of the descriptor to work on, and the other register reports the current operational status of the compute unit.
- An outbound connection to the **local memory** in the accelerator, for reading descriptors and writing results.
- An inbound signal that tells it to **start** computing. This signal is handled not as a Simics simulator signal interface but using a custom interface together with the **clear done** handling.
- An outbound signal that indicates that the operation is done (has completed). This is modeled as a Simics simulator signal interface. It state is supposed to be raised as long as the done flag in the status register is set to 1.
- An inbound signal to **clear the done flag** (and lower the outbound done signal). This is modeled by a function call in the same interface as the start signal.
- An outbound connection to the **queue** used by this object for retrieving time and posting events.
- A Simics simulator **attribute** to control the **compute time per pixel**. In the basic model, this is a fixed time, which might be slightly unrealistic. Overall, it provides the ability to model throughput approximately.
- A Simics simulator **notifier** to tell scripts that the operation is done. This can be used to run the simulation until a compute job is done, for example.

The custom interface used to control the compute unit is defined as its own Simics simulator module. This interface definition has to be used by the control unit and other code when communicating with the compute unit.

Inside the compute unit, there is the core compute code, the control logic for how compute operations are started, and the definition of the memory layout of the descriptor.

4.1 Inspect the custom interface

To inspect the definition of the custom control interface, you need to open its definition.

1. Go to the host shell in the Simics project, and copy the interface module into the local project:

   ```
   $ bin/project-setup --copy-module m-compute-control-interface
   ```

2. Rebuild the code:

   ```
   $ make
   ```

   The interface module build creates a Python wrapper for the interface and provides metadata so that the simulator core knows that it exists.

3. Open the file **[*project*]/ modules/m-compute-control-interface/m-compute-control-interface.dml** in an editor.

4. The definition of the interface is wrapped inside a struct definition. The definition tells us that the device implementing the interface needs to implement two functions:

   ```
   void start_operation(conf_object_t*)
   ```

   And:

   ```
   void clear_done(conf_object_t*)
   ```

   The interface functions can also be called from Python, since the interface is automatically wrapped into Python during compilation.

4.2 Start a Simics simulation session with the compute unit

5. Start a new simulation session using script **001-**:

   ```
   $ ./simics targets/workshop-02/001-try-m-compute.simics
   ```

4.3 Inspect the start-up scripts

6. **Open** the start script in an editor, it is found at **[*simics*]/targets/workshop-02/001-try-m-compute.simics**. See above for how to locate the Simics simulator installation. All it does is call one Python file that is used to set up the experimental setup.

   It also contains a large "commented out" section of code that essentially goes through the commands of this lab.

7. **Open** that Python script file, **[*simics*]/targets/workshop-02/001-m-compute-setup.py**. This file creates a set of **pre_conf_object**s to describe the setup, and then calls **SIM_add_configuration()** to create all the objects at once. This is the

standard way to create small sets of objects in the Simics simulator. In particular for unit testing.

4.4 Inspect the configuration

8. List the objects in the simulation configuration:

```
simics> list-objects
```

The objects you see are a mix of the standard infrastructure objects found in all simulations (**bp**, **breakpoints**, **default_cell0**, **default_sync_domain**, **sim**, **prefs**, **params**), and the small set of objects (**clock**, **compute**, **ram**, **ram_image**, **local_memory**) created from the script.

9. Use **help** on the compute object:

```
simics> help compute
```

The output starts like this:

```
simics> help compute
Class m_compute

   Provided By
      m-compute

   Description
      Compute unit for the mandelbrot hardware accelerator
...
```

This illustrates an important aspect of Simics naming. The object named **compute** is an object in the Simics simulator configuration. It is created from the class **m_compute** (with an underscore). Multiple objects can be created from each class (obviously), with arbitrary names. The class **m_compute** is provided by the module **m-compute** (with a dash – module names are preferably named using dashes). The module is automatically loaded into the simulation session when an object of the class is created. Modules can contain multiple classes, in general.

10. Check the current configuration of the object using its **info** namespace command (such commands can be defined for all classes in the Simics simulator to provide a user-friendly interface to high-level information about objects):

```
simics> compute.info
```

11. Look at the control registers of the **compute** object:

```
simics> print-device-regs compute
```

If the device had had multiple register banks, this would print all the registers from all banks. In this case, there is just a single bank holding two registers. Both are currently zero as no address has been set and no operation has been performed.

12. Check the mappings in the **local_memory** memory space:

```
simics> local_memory.map
```

Memory spaces are how all memory transactions in the Simics simulator reach their destination. In this case, there is one mapping for the local memory ("**ram**") and one for the register bank of the compute unit.

13. Check the configuration attributes of the compute unit:

```
simics> list-attributes compute
```

14. Help can also be used on an individual attribute (in command-line commands, an object attribute is accessed via **->**):

```
simics> help compute->local_memory
```

15. List the interfaces of the compute object – this how other parts of the simulation communicate with the compute unit.

```
simics> list-interfaces compute
```

You will see two "ports" listed: the unified control interface (**control_in**) and the register bank (**ctrl**). A register bank is technically also a port of a model since it receives memory operations.

4.5 Set up a descriptor

To get the compute unit to do anything, it is necessary to set up a compute job descriptor in memory. This can be done manually, setting one value after the other from the command line. To speed things up, there is a Python script available that provides a utility function that creates descriptors and poking them into target system memory.

16. Run the Python script **[*simics*]/targets/workshop-02/001-m-compute-descriptor-generator.py** to define the utility function.

```
simics> run-python-file "%simics%/targets/workshop-02/001-m-compute-descriptor-generator.py"
```

The **%simics%** specifier in the string ask the Simics simulator to look for a file matching the rest of the string in the Simics project, as well as in all installed Simics packages.

17. To see where **%simics%** searches for files, use the **list-simics-search-paths** command:

```
simics> list-simics-search-paths
```

The first entry is your Simics project, and then you see all installed packages in the order in which they are searched.

18. Before building a descriptor, make sure that the target area in local memory is empty. Use the namespaced **x** command on the **local_memory** object. To find the arguments of the command, use help:

```
simics> help local_memory.x
```

19. Display 40 bytes (the size of a descriptor) from offset 0x1000:

```
simics> local_memory.x 0x1000 40
```

20. Call the Python function **`create_m_compute_descriptor()`** with some easily recognizable Mandelbrot parameters. The first argument to the function is the local memory object to write the descriptor to, the second argument is the address of the descriptor, the third argument is the address of the results, and then follow bottom, left, top, right, width, height, and max_iter.

```
simics> @create_m_compute_descriptor(conf.local_memory, 0x1000, 0x2000,
-1.0, -1.0, 1.0, 1.0, 100, 100, 200)
```

"**`@`**" at the beginning of a line makes the Simics simulator interpret the rest of the line as Python code. "**`conf.NNN`**" is the Python reference to the Simics configuration object called **`NNN`**.

21. Display the descriptor area again. Group values in groups of 32 bits, and with little-endian interpretation:

```
simics> local_memory.x 0x1000 40 group-by = 32 -l
```

This should show that the memory now contains a descriptor:

```
p:0x00001000  40000000 40000000 c0000000 c0000000
p:0x00001010  00000064 00000064 000000c8 deadbeef
p:0x00001020  00002000 00000000
```

22. Check the "bottom" value. This indicates the y coordinate of the bottom of the area to plot. Read the value from memory using a **`read`** command:

```
simics> local_memory.read 0x1000 4 -l
```

23. Use this as part of an expression decoding the encoding:

```
simics> ((local_memory.read 0x1000 4 -l) - 0x8000_0000) / 0x4000_0000
```

24. Check that the area used for the results is all zeroes:

```
simics> local_memory.x 0x2000 group-by = 16 -l 64
```

4.6 Start a compute job

Configure the compute unit to use the new descriptor, and instruct it to do the compute job.

25. Raise the log-level of the compute unit to 3 (which includes internal debug messages):

```
simics> log-level compute 3
```

26. Configure logs to display when they happen in virtual picoseconds:

```
simics> log-setup -pico-seconds -group
```

27. Use the **`write-device-reg`** command to write the **`descriptor_addr`** register of the compute unit:

```
simics> write-device-reg compute.bank.ctrl.descriptor_addr 0x1000
```

Note that registers that do not have side effects do normally log anything on accesses; to trace register reads and writes, use the dedicated **`trace-io`** command.

28. To make the compute unit perform the compute operation, it needs to be started using the "start signal". This is part of the custom control interface that is exposed in the **control_in** port.

    Check the metadata on the **control_in** port:

```
simics> help compute.port.control_in
```

    The output indicates that the port implements three interfaces. **control_in** is interface used in the hardware model. **conf_object** and **log_object** are standard framework interfaces that are present on all Simics objects. Note that a port object is a full Simics object from the perspective of the framework.

```
   Description
      control input from the control unit

   Interfaces Implemented
      conf_object, log_object, m_compute_control
```

    The general form of calling an interface in an object in the simulation from the command line using inline Python is this:

```
@conf.<object_name>.iface.<interface_name>.<function_name>(<arguments>)
```

    Since a port object is an object, the same pattern is used, just with the full name of the port object:

```
@conf.<object_name>.port.<port_name>.iface.<interface_name>.<function_name>(<arguments>)
```

29. From the simulator command line, inline Python to call **start_operation()** in the **control_in** port of the compute unit:

```
simics>
@conf.compute.port.control_in.iface.m_compute_control.start_operation()
```

    Note that it is not necessary to have an object to use as the "originator" for an operation like this. The code in the device model will get called just as if the call came from another object, and it has no way to know where the call came from.

    When the signal is raised, several log messages are printed:

```
simics> @conf.compute.port.control_in.iface.m_compute_control.start_operation()
[compute.port.control_in info control] {0 ps} Received request to start compute job
[compute info compute] {0 ps} start_compute_job called
[compute info compute] {0 ps} Work descriptor read. Area: (-1.000000, -1.000000) -
(1.000000, 1.000000) (100, 100) iter: 200 pixels  @ 0x2000
[compute info compute] {0 ps} Compute operation time: 0.0001000000 s
[compute info compute] {0 ps} Starting computation inline in main thread
None
```

    The first part of the log message shows **[*object type group*]** – the *object* that issues the log (in this case either the **compute** object or its **compute.port.control_in** port object). The *type* is **info** in these examples, indicating informational messages. The *group*, finally, indicates the log group for the message. This can be used to turn on and off logging for just certain aspects of a model. They are defined by **loggroup** statements in the device model source code, and some are provided by the standard device templates and modeling libraries.

After that comes the time specification, **{0 ps}**. All the logs here were printed within the same picosecond of virtual time. All happened at once from the perspective of the rest of the simulation, essentially.

30. The logs above indicate that the compute operation is supposed to take 100 microseconds. The delay is implemented by posting an event 100 microseconds into the future – on **queue** assigned to the compute object. In the Simics simulator, event queues are distributed and handled by processors or clock objects, not by the simulation kernel. This means that there are typically several event queues in a single simulation, and each object is configured with a default queue to use for posting events. This configuration is performed using the **queue** attribute present in all objects.

    Read the **queue** attribute of the **compute** object:

    ```
    simics> compute->queue
    ```

    This should indicate the "**clock**" object.

31. In device code, the **SIM_object_clock()** API call is typically used to retrieve the queue for an object. Note that "clock" and "queue" are used interchangeably and not quite consistently in the API.

    Try the API using inline Python on the command line:

    ```
    simics> @SIM_object_clock(conf.compute)
    ```

32. Check the current time of the **clock** object:

    ```
    simics> ptime clock
    ```

    The current time is zero since the simulation has not been run forward yet.

33. Check the event queues of the **clock** object:

    ```
    simics> peq clock
    ```

    This shows a single event posted 100k cycles into the future (since 100 microseconds is 100k cycles at 1GHz):

Cycle	Object	Description
100_000	compute	compute_operation_complete

34. Check the contents of the results memory at this point:

    ```
    simics> local_memory.x 0x2000 group-by = 16 -l 64
    ```

    Note that the results have already been written to memory. They were computed immediately on operation start. However, software is not supposed to look at them until the compute unit indicates that the results are ready.

35. Check the value of the **status** register:

```
simics> print-device-reg-info compute.bank.ctrl.status
```

The "processing" bit it set, indicating that work is in progress, from the perspective of the software using the hardware. Even though the results are already present in the simulated memory.

36. Run the simulation forward for 100 microseconds, to reach the end of the compute operation:

```
simics> run 100 us
```

Log messages will be printed indicating that the operation completes (this is based on rendering 100x100 pixels; if you used other dimensions, the time will be different).

```
[compute info compute] {100000000 ps} Compute operation nominally finished
[compute info control] {100000000 ps} Setting done flag
[compute info control] {100000000 ps} No connected object to signal completion to
```

37. Check the current time in the simulation:

```
simics> ptime clock
```

38. Check the event queue:

```
simics> peq clock
```

It should be empty since the event has triggered and no new events have been posted.

39. Check the value of the **status** register:

```
simics> print-device-reg-info compute.bank.ctrl.status
```

It shows that the operation is done. The number of computed pixels is also reflected in the **count** field.

```
simics> print-device-reg-info compute.bank.ctrl.status
Compute status [compute.bank.ctrl:status]
=========================================


                  Bits : 64
                Offset : 0x8
                 Value : 9_223_372_036_854_785_808 (0x8000_0000_0000_2710)

Bit Fields:
          done[63..63] : 1                                  "Compute completed"
    processing[62..62] : 0                                  "Compute in progress"
        unused[61..32] : 00000000000000000000000000000000   "unused"
          count[31..0] : 00000000000000000010011100010000 "Processed pixel count"
```

40. It is possible access the value of a field using the register name and the field name. For example, to read out the value of the **count** field of the **status** register:

```
simics> read-device-reg compute.bank.ctrl.status count
```

This should return the value 10000.

41. To clear the done flag, write a "1" bit to the done flag, using a whole-register value of **0x8000_0000_0000_0000**:

```
simics> write-device-reg compute.bank.ctrl.status 0x8000_0000_0000_0000
```

The device will complain that zero is being written to the count field – in the form of a **spec_viol** log message. This means that the "software" is not following the documented behavior of the register, by trying to change the value of a read-only field.

4.7 Check the status register implementation

Check the implementation of the status register to see where the **spec-viol** log message comes from.

42. **Open** the file **[*project*]/modules/m-compute/m_compute.dml**.

43. Search the code for "**register status**". The result should look something like this, depending on your editor:

```
    // Status register
    register status @ 0x08 "Compute status" {
        field done        @ [63] is (write)       "Compute completed" ;
        field processing  @ [62] is (read_only)   "Compute in progress" ;
        field unused      @ [61:32] is (reserved) "unused" ;
        field count       @ [31:0] is (read_only) "Processed pixel count" ;
    }
}

//-----------------------------------------------------------------------
//
// Register bank implementation
// - Adding the functionality to the names and offsets declared above
//
//-----------------------------------------------------------------------
bank ctrl {
    register status {
        field done is write {
            method write(uint64 v) {
                if (v==1) {
                    if(this.val==1) {
                        do_clear_done();
                    } else {
                        log spec_viol, 1, control: "Attempt to clear already clear done flag";
                    }
                } else {
                    log spec_viol, 1, control: "Writing zero to done has no effect";
                }
            }
        }
    }
```

The log comes from **read_only** template that has been applied to the **count** field. A field declared as **read_only** will log if a value different from the current value of the field is written to it (and ignore the written value in any case). Given this definition, the "proper" way to clear the register is to write back the value you read from it.

44. The **done** field got cleared despite the warning about **count**. Check the new state of the register:

```
simics> print-device-reg-info compute.bank.ctrl.status
```

This concludes this section. You have seen how to drive an individual device from the simulator command line, and how to inspect the device state and system state. Using this kind of interaction, it is easy to explore the implementation of a device model without a full system context.

4.8 Test some error cases

Investigate how the model deals with some types of bad inputs.

45. Set up a new descriptor in memory, in a different location and with different contents:

```
simics> @create_m_compute_descriptor(conf.local_memory, 0x10000,
0x11000, -0.4, -0.4, 0.4, 0.4, 150, 150, 200)
```

46. Check the contents:

```
simics> local_memory.x 0x10000 group-by = 32 -l 40
```

47. Change the descriptor address register to point at address zero, where the descriptor is *not* located:

```
simics> write-device-reg compute.bank.ctrl.descriptor_addr 0x0000
```

48. Tell the device to start the computation:

```
simics>
@conf.compute.port.control_in.iface.m_compute_control.start_operation()
```

    Luckily, an all zero descriptor results in no work being performed, not a crash. This kind of "likely" mistake might be worth a specific check in the model, along with a spec-violation log message.

49. Check the event queue:

```
simics> peq clock
```

    Zero time is converted to "on the next cycle".

50. Try to fix the error by updating the descriptor address:

```
simics> write-device-reg compute.bank.ctrl.descriptor_addr 0x10000
```

51. ...and starting a new run:

```
simics>
@conf.compute.port.control_in.iface.m_compute_control.start_operation()
```

    The device will complain, since the previous operation has not yet completed.

```
[compute.port.start info control] {100000000 ps} start.signal - Signal raised
[compute.port.start info control] {100000000 ps} Received signal to start compute job
[compute.port.start spec-viol control] {100000000 ps} Operation start request while
operation in progress
```

52. Run 1 cycle to trigger the event and complete the operation:

```
simics> run 1 cycle
```

53. Start a new run:

```
simics>
@conf.compute.port.control_in.iface.m_compute_control.start_operation()
```

    Check the log messages to make sure this run was indeed using the valid descriptor address and valid descriptor contents.

54. Try to write the status register:

```
simics> write-device-reg compute.bank.ctrl.status 0
```

    This time, the **processing** field will complain that a read-only field is being written, and the **done** field will complain that writing zero to it has no effect. Such information can be quite useful to someone writing a device driver for the hardware.

4.9 Detect the end of the operation using notifiers

In the Simics simulator, notifiers are used to signal that something has happened in a device model or other part of the simulated system or simulator itself. They are most commonly used as model-to-simulator channels to implement user-facing features (they can also be used for some instances of model-to-model communication).

55. Run until the job completes, using the notifier built into the model to signal completion. List all available notifiers:

```
simics> list-notifiers
```

56. **m-compute-complete** is triggered by the compute unit when the computation is completed. Run until it triggers:

```
simics> bp.notifier.run-until m-compute-complete
```

    This command uses the Simics simulator breakpoint manager to run the simulation forward until the notifier is triggered.

57. When the simulation stops, the operation has completed. Check the current time:

```
simics> ptime clock
```

58. Clear the done flag using the control interface, imitating how the control unit would do it:

```
simics>
@conf.compute.port.control_in.iface.m_compute_control.clear_done()
```

59. Check the status register contents:

```
simics> print-device-reg-info compute.bank.ctrl.status
```

60. Exit this simulation session.

```
simics> exit
```

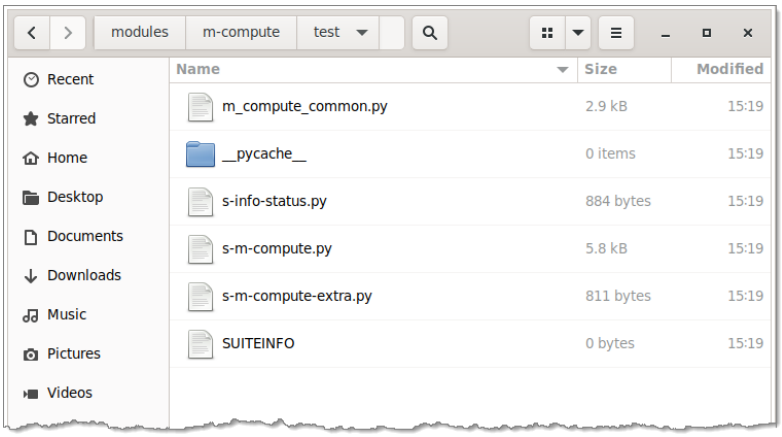
4.10 Run unit tests on the compute unit

The interactive session in the previous section shows how an individual device model can be run on its own inside of a Simics simulator configuration, with a few other objects

around it to provide the necessary context. This ability is also used for unit testing device models. Each device model should come with a dedicated unit test suite that tests the model on its own, without the need for a system context. Unit tests should make use of as few other "real" device classes as possible to keep their dependencies minimal.

The *Simics Model Builder User Guide* contains more information about unit testing in Chapter 16. The *Simics Reference Manual* Chapter 7 describes the overall test framework. Python utilities used in writing tests are described in the *API Reference Manual* in Chapter 10.7.

61. In a file browser, open **[*project*]/modules/m-compute/test/**. This directory contains the test suite for the **m-compute** module.



There are three types of files here:

- **SUITEINFO** declares that this directory contains a test.
- **m_compute_common.py** contains the common setup code used to create the device under test and the stubs devices/fake objects needed to run tests.
- **s-*.py** are the actual tests. Each file contains a set of tests that logically belong together.

When you create a new Simics device model, you get a skeleton implementation of the tests for it.

Open the file **s-m-compute.py**. This is the main test file that tests the basic behavior of the compute unit. There are quite a few steps in the test.

Checks for errors in tests are done using the **stest** Python framework, including all aspects of the execution, from values in registers and memory to log messages printed from device models (or the absence thereof).

62. Go to the Simics project directory in your host shell.
63. Run the tests for the **m-compute** module using the **test-runner** tool.

First, list the tests it knows about:

```
$ bin/test-runner --tests
```

It will show the suite for **m-compute**, and each **s-** Python file is considered its own test:

```
Suite: modules/m-compute/test
    s-info-status
    s-m-compute-extra
    s-m-compute
```

64. Run the tests with verbose output, to see how each test is run in turn:

```
$ bin/test-runner -v
```

The test log file is saved in the project, at

[*project*]/logs/test/[*host-type*]/modules/m-compute/test/test.log

65. For example, on a Linux host, display the contents of the log file:

```
$ more logs/test/linux64/modules/m-compute/test/test.log
```

This shows the log messages printed during the test.

Note that the tests do not quite use the same setup as the interactive session – special test memory is used to make it easier to detect issues like reading from memory that was not previously written to. There is also a fake signal receiver to receive the outbound signal that was not set up at all in the interactive session, as well as a simple receiver for notifications.

4.11 Introduce device misbehavior, fail unit test

To test the unit tests, introduce an intentional error into the device model.

66. If it is not already open, open the file **[project]/modules/m_compute/m_compute.dml** in an editor.
67. Search the code for "**do_clear_done**".
68. Edit the method **do_clear_done()** to return immediately instead of doing the work of actually clearing the done flag:

```
method do_clear_done() {
    return;   // intentional error
    log info, 2, control: "Clearing done flag";
    operation_done.signal_done_clear();
    ctrl.status.done.val = 0;
}
```

69. Build the device model:

```
$ make
```

70. Run the tests:

```
$ bin/test-runner
```

Which will report an error:

```
..f
[...]modules/m_compute/test/test.log:21: test s-m-compute in modules/m_compute/test
failed (*** failed (exit-status 2) ***)
Ran 3 tests in 1 suites in 0.882581 seconds.
Failures: 1  Timeouts: 0
```

71. Check the log file. Either open the log file in an editor (use the full path provided in the test output) or use **`more`** from the shell in the same way as above.

    The error behind the test failure is reported like this:

```
Expected 100 = 0x64
Got      9223372036854775908 = 0x8000000000000064
*** Python script 's-m-compute.py' failed: Python error in PyEval_Evalcode():
…
      "Failed to clear done flag")
…
```

    Basically, a failed test triggers a Python traceback in order to display the point of failure even if tests are run through multiple Python files.

72. Edit the method **`do_clear_done()`** in the model source code to remove the bad return statement.

73. Build the device model and test it in one go, using the **`test`** target:

```
$ make test
```

    This both rebuilds any changed code, and reruns all known tests.

5 Display Results using Python

It makes sense to convert the output of the compute unit to a displayed image early on in the development process – without having to build the model of the display unit and its software interface. Instead, a Python script can be used to visualize the results of the computation, sending pixels to a standard Simics framework graphics console.

The Python code will read the results in simulator memory, convert an iteration value into a color, and then put a colored pixel into the console.

5.1 Compute a result

1. Start a new Simics simulation using script **002-**:

```
$ ./simics targets/workshop-02/002-try-m-compute-with-display.simics
```

2. List all objects in the simulation:

```
simics> list-objects
```

   Note the presence of the graphics console object called **con**.

3. Raise the log level:

```
simics> log-level 2
```

4. Set up a descriptor:

```
simics> @create_m_compute_descriptor(conf.local_memory, 0x1000, 0x2000,
0.5, -0.6, 0.7, -0.4, 1000, 1000, 256)
```

5. Start the operation in the same way as before:

```
simics> write-device-reg compute.bank.ctrl.descriptor_addr 0x1000
simics>
@conf.compute.port.control_in.iface.m_compute_control.start_operation()
```

6. Run until the results are computed:

```
simics> bp.notifier.run-until m-compute-complete
```

7. Display the results in the graphics console, by calling the Python code. It needs to know the console object, the local memory object, the starting address of the results in the local memory, the pixel dimensions of the result, and the maximum iteration count:

```
simics> @display_m_result(conf.con, conf.local_memory, 0x2000, 1000,
1000, 256)
```

   This should result in the console window showing a Mandelbrot fractal.

   In addition, the code will print some basic statistics about the results. The graphics console will emit a few log messages about being resized.

8. Note that the Simics simulator graphics console might show the results as greyed out, as a result of the default behavior of the graphics consoles to grey out their

display when the simulator is stopped. Thus, to see the full color, start the simulation and leave it running.

```
simics> r
```

5.2 Copy the Python display code

The Python display code is currently located in a script in the Simics simulator installation, at **[simics]/targets/workshop-02/002-display-result.py**. To modify it, it should be copied into your Simics project, into **targets/workshop-02/**. The copy can be performed using the simulator command line.

9. Since you have a Simics session running, use the command-line command **lookup-file** to put the path to the script into the CLI variable **$s**:

```
running> $s = (lookup-file "%simics%/targets/workshop-02/002-display-result.py")
```

10. Check the result:

```
running> $s
```

11. The destination in the Simics project can be produced by using **lookup-file** with just the name of the target directory:

```
running> $p = (lookup-file "%simics%/targets/workshop-02")
```

12. Check the result:

```
running> $p
```

13. Use Python **shutil** to copy the file (works on all hosts):

```
running> @import shutil
running> @shutil.copy(simenv.s, simenv.p)
```

simenv.*V* is the way to access a command-line variable named ***V*** from Python.

5.3 Inspect and modify the display code

A core part of the Python code is to determine how to color the fractal. This code is a Python prototype for the code used in the **m-app** application.

14. **Open** the file **[project]/targets/workshop-02/002-display-result.py** in an editor.
15. **Search** for "**def colorize**". This will find the definition of the function that converts an iteration count into a **0x00RRGGBB** 24-bit color value.
16. Rename the existing function to **colorize_2**.

17. Add a new function **colorize**, which implements a simple gradient from black to bright yellow, like this:

```
def colorize(v,min,max):
    # From black (0x000000) to light yellow (0xffff80)
    s1=min
    s2=max
    c1=[0x00,0x00,0x00]
    c2=[0xff,0xff,0x80]
    if(v < s2):
        return rgb_from_list(color_interpolate(c1,c2,(v-s1)/(s2-s1)))
    return 0x000000
```

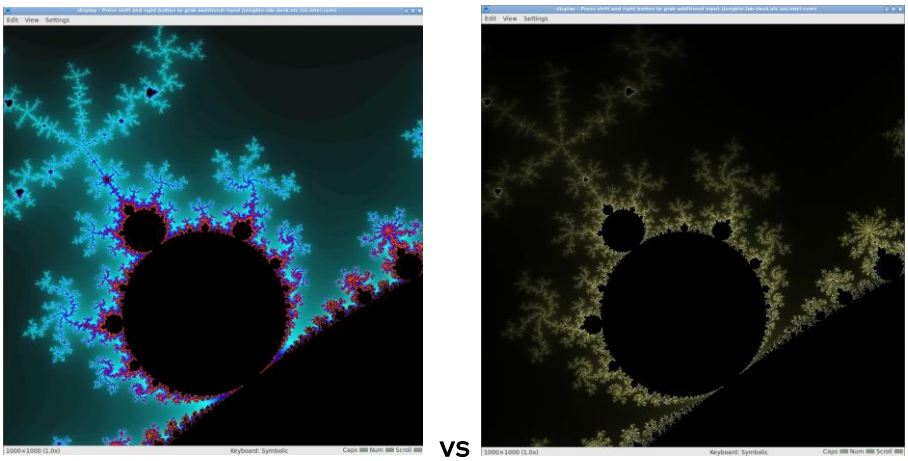
18. Reload the display code from the project:

```
running> run-python-file targets/workshop-02/002-display-result.py
```

19. Redraw the result:

```
running> @display_m_result(conf.con, conf.local_memory, 0x2000, 1000,
1000, 256)
```

The new display is less colorful:



The key point is that it is possible to change and reload Python code during a Simics simulation session, which is very handy for experimenting with parts of the system without having to rebuild anything.

20. To keep experimenting with the color logic, also try some more zoomed-in versions of the fractal. Keep the simulator running; there is no real need to stop to display the result.

For example:

```
running> @create_m_compute_descriptor(conf.local_memory, 0x1000,
0x2000, 0.60032495,-0.55318505,0.60032505,-0.55318495, 1000, 1000, 256)
running>
@conf.compute.port.control_in.iface.m_compute_control.start_operation()
running> @display_m_result(conf.con, conf.local_memory, 0x2000, 1000,
1000, 256)
```

Edit and reload the Python file. Rerun the display:

```
running> run-python-file targets/workshop-02/002-display-result.py
running> @display_m_result(conf.con, conf.local_memory, 0x2000, 1000,
1000, 256)
```

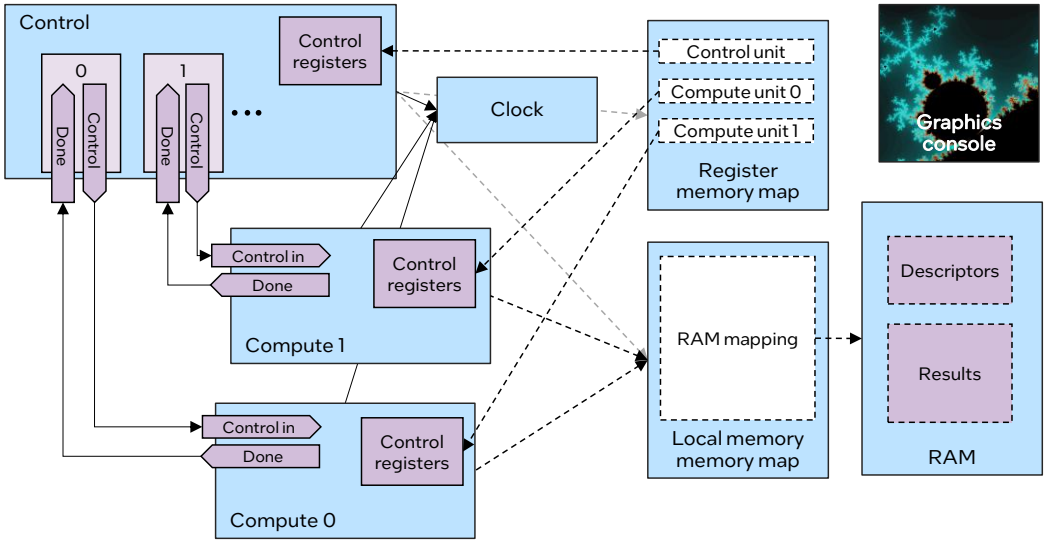
21. Exit this simulation session.

```
simics> exit
```

6 Integrate the Control Unit with the Compute Unit

The control unit is used to start compute operations in multiple compute units. It provides a single interface to the outside that indicates that an operation has completed. The setup shows how device models communicate in the Simics simulator framework.

The configuration looks like this, for the case of two compute units being controlled by a single control unit:



The active devices (control and compute units) use the clock to post events.

All the control registers are mapped in the register memory map. This is unlike the previous setup where a single memory map pointed at both the control registers and the memory.

The compute units access the on-accelerator RAM through the "local memory" memory map. The control unit does not *need* to have references to the memory maps for the simple case, but they are used in the PCIe case and are set up in any case.

The control unit has a connection to control port on each compute unit, to start operations and clear the done flag.

The control unit has a global status register with a done flag reflecting the state of all the compute units. The done flag is set after all compute units signal that they are done. Clearing the global done flag makes the control unit clear the done flag of all connected compute units.

The graphics console is a separate object that is not actually connected to anything else in the configuration. It driven from Python, like in the previous cases.

6.1 Copy the control unit code to the project

1. Use **`project-setup`** to copy the **`m-control`** module to your project:

```
$ bin/project-setup --copy-module m-control
```

2. Build all the modules in the project:

```
$ make
```

6.2 Start the simulation and create objects

3. Start a new simulation session using script **003-**:

```
$ ./simics targets/workshop-02/003-control-unit.simics
```

4. List the objects in the simulation configuration:

```
simics> list-objects
```

There is no model in place (yet), so all that is listed are the standard objects that all Simics simulator sessions contain as part of the simulation framework.

5. **Open** the Python file **[simics]/targets/workshop-02/003-control-unit-setup.py** in an editor. You can find its location using this command-line command:

```
simics> lookup-file "%simics%/targets/workshop-02/003-control-unit-setup.py"
```

The system creation is contained inside a Python function. The first argument to the function is the name of the subsystem, and the second argument is the number of compute units to create as part of the subsystem.

6. Create a new accelerator subsystem using the Python function:

```
simics> @create_N_compute_accelerator("macc",2)
```

7. Inspect the newly created set of models:

```
simics> list-objects namespace = macc
```

There are two compute units, a control unit, plus the local memory and a (new) memory space containing the registers for all the devices.

8. The accelerator objects are now inside the namespace **macc**, instead of being at the top level of the name hierarchy (in the same way that they were in a namespace in Section 3 when using the complete setup).

```
simics> list-objects namespace=macc
```

Class	Object
<clock>	macc.clock
<m_compute>	macc.compute[0]
<m_compute>	macc.compute[1]
<graphcon>	macc.con
<m_control>	macc.control
<memory-space>	macc.local_memory
<ram>	macc.ram
<recorder>	macc.rec
<memory-space>	macc.register_memory

9. Create a second accelerator subsystem called macc8 with 8 units:

```
simics> @create_N_compute_accelerator("macc8",8)
```

The Simics framework can contain an arbitrary set of models, including multiple copies of the same device or multiple subsystems of the same content. As long as names are unique, everything will work nicely.

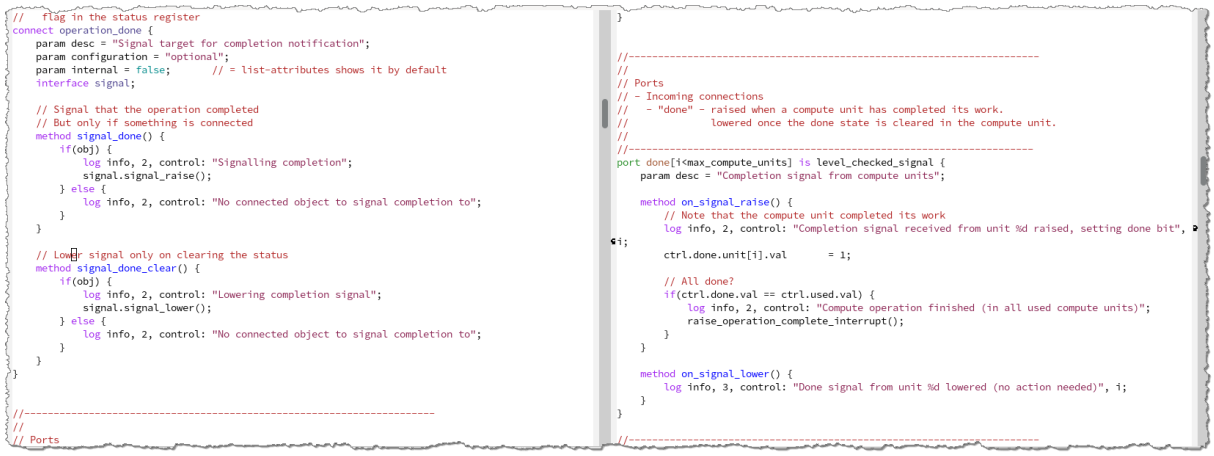
10. Inspect the newly created set of models:

```
simics> list-objects namespace = macc8
```

6.3 Inspecting device-to-device connection in the DML code

The connections between the compute and control units are coded in DML as **`connect`** objects from the side that needs to call into the interface, and as **`port`** objects on the side that is called.

11. **Open** the file **`[project]/modules/m-control/m_control.dml`** in an editor, to get the source code of the control unit.
12. **Search** for "**`port done`**" to find the control unit input **`port`** that receives completion signals from the compute units.
13. **Open** the file **`[project]/modules/m-compute/m_compute.dml`** in an editor, to get the source code of the compute unit.
14. **Search** for "**`connect operation_done`**" to find the compute unit **`connect`** that points at the control unit port.
15. Put the two pieces of code side by side.



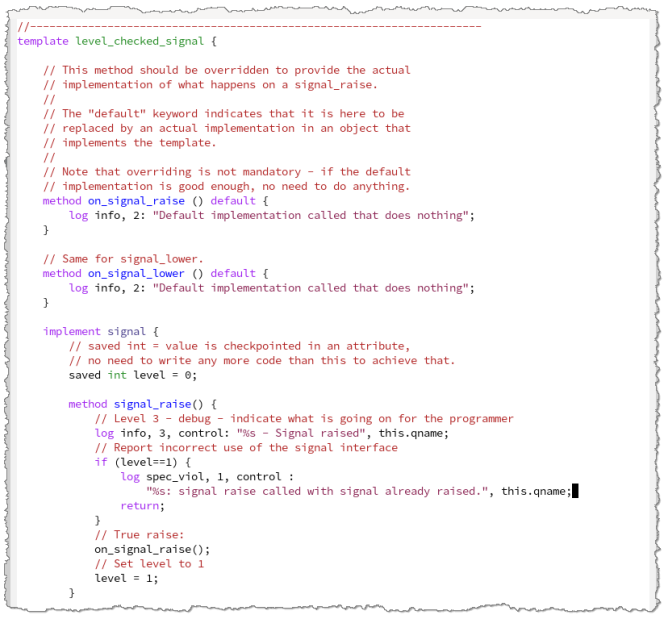
```
//   flag in the status register
connect operation_done {
    param desc = "Signal target for completion notification";
    param configuration = "optional";
    param internal = false;       // = list-attributes shows it by default
    interface signal;

    // Signal that the operation completed
    // But only if something is connected
    method signal_done() {
        if(obj) {
            log info, 2, control: "Signalling completion";
            signal.signal_raise();
        } else {
            log info, 2, control: "No connected object to signal completion to";
        }
    }

    // Lower signal only on clearing the status
    method signal_done_clear() {
        if(obj) {
            log info, 2, control: "Lowering completion signal";
            signal.signal_lower();
        } else {
            log info, 2, control: "No connected object to signal completion to";
        }
    }
}


//---------------------------------------------------------------------
//
// Ports
```

```
}


//---------------------------------------------------------------------
//
// Ports
// - Incoming connections
//   - "done" - raised when a compute unit has completed its work.
//              lowered once the done state is cleared in the compute unit.
//
//---------------------------------------------------------------------
port done[i<max_compute_units] is level_checked_signal {
    param desc = "Completion signal from compute units";

    method on_signal_raise() {
        // Note that the compute unit completed its work
        log info, 2, control: "Completion signal received from unit %d raised, setting done bit", i;
        ctrl.done.unit[i].val       = 1;

        // All done?
        if(ctrl.done.val == ctrl.used.val) {
            log info, 2, control: "Compute operation finished (in all used compute units)";
            raise_operation_complete_interrupt();
        }
    }

    method on_signal_lower() {
        log info, 3, control: "Done signal from unit %d lowered (no action needed)", i;
    }
}

//---------------------------------------------------------------------
```

The connect declaration results in an attribute being added that is used to point at a port or other object. The declaration **`interface signal`** in the connect declaration indicates that the connected object should implement the signal interface.

The declaration **`port done`** creates an array of ports based on a local template called **`level_checked_signal`**. Each connect will point to one of these ports.

16. Scroll back up a bit to find the declaration of **`template level_checked_signal`**:



```
//-----------------------------------------------------------------------
template level_checked_signal {

    // This method should be overridden to provide the actual
    // implementation of what happens on a signal_raise.
    //
    // The "default" keyword indicates that it is here to be
    // replaced by an actual implementation in an object that
    // implements the template.
    //
    // Note that overriding is not mandatory - if the default
    // implementation is good enough, no need to do anything.
    method on_signal_raise () default {
        log info, 2: "Default implementation called that does nothing";
    }

    // Same for signal_lower.
    method on_signal_lower () default {
        log info, 2: "Default implementation called that does nothing";
    }

    implement signal {
        // saved int = value is checkpointed in an attribute,
        // no need to write any more code than this to achieve that.
        saved int level = 0;

        method signal_raise() {
            // Level 3 - debug - indicate what is going on for the programmer
            log info, 3, control: "%s - Signal raised", this.qname;
            // Report incorrect use of the signal interface
            if (level==1) {
                log spec_viol, 1, control :
                    "%s: signal raise called with signal already raised.", this.qname;
                return;
            }
            // True raise:
            on_signal_raise();
            // Set level to 1
            level = 1;
        }
```

The declaration "**`implement signal`**" indicates that the port based on this template implements the signal interface. This aligns with the declaration in the connect.

17. Go to the Simics simulator command line.

    List the attributes of the object **`macc.compute[1]`** (the second compute unit in the first accelerator subsystem created):

```
simics> list-attributes "macc.compute[1]"
```

    Note how the attribute **`operation_done`** is set to point at **`macc.control.port.done[1]`**. This connection is set up by the setup script, but could also be configured and changed interactively from the command line.

    There is nothing on the control unit side that knows about this incoming connection- it just receives signal calls on the port, and such calls can come from other devices or from the command line or from script code.

18. By giving the name of a specific attribute to list-attributes, you get more information about it:

```
simics> list-attributes "macc.compute[1]" operation_done
```

    The help text is derived from the "**`param desc`**" declaration in the connect, plus autogenerated text indicating the required interface(s) of the connect.

19. Test the error checking with an intentionally bad change to the **`operation_done`** attribute. The standard **`sim`** object does not implement the signal interface.

```
simics> macc.compute[1]->operation_done = sim
```

20. Check that the attribute still has the same value as before:

```
simics> macc.compute[1]->operation_done
```

21. To find the inbound ports of the control unit, use **list-objects**:

```
simics> list-objects namespace = macc.control
```

22. To view the ports and banks of the control unit as a tree:

```
simics> list-objects namespace = macc.control -tree
```

23. Use **help** on the **done[1]** port:

```
simics> help macc.control.port.done[1]
```

This shows that the port implements the **signal** interface, as required by the **macc.compute[1].operation_done** attribute you inspected above.

6.4 Check the definition of the signal interface

24. To quickly find the definition of standard Simics simulation interfaces and simulator API calls, the **api-help** command is pretty handy. Note that it currently does not know about interfaces defined outside of the Simics base product. Use tab completion to find the information about the **signal** interface:

```
simics> api-help signal<TAB>
```

This should expand to:

```
simics> api-help signal_interface_t
```

25. Press **<RETURN>**.

```
simics> api-help signal_interface_t
```

This prints a help text explaining how the interface works, as well as the definition of the functions in the interface.

```
...

SHORT DESCRIPTION

#include <simics/devs/signal.h>    // in C/C++
import "simics/devs/signal.dml";   // in DML

typedef struct signal_interface {
    void (*signal_raise)(conf_object_t *NOTNULL obj);
    void (*signal_lower)(conf_object_t *NOTNULL obj);
} signal_interface_t;

// available in Python
```

26. If you do not quite know the name of what you are looking for, use the **api-search** command.

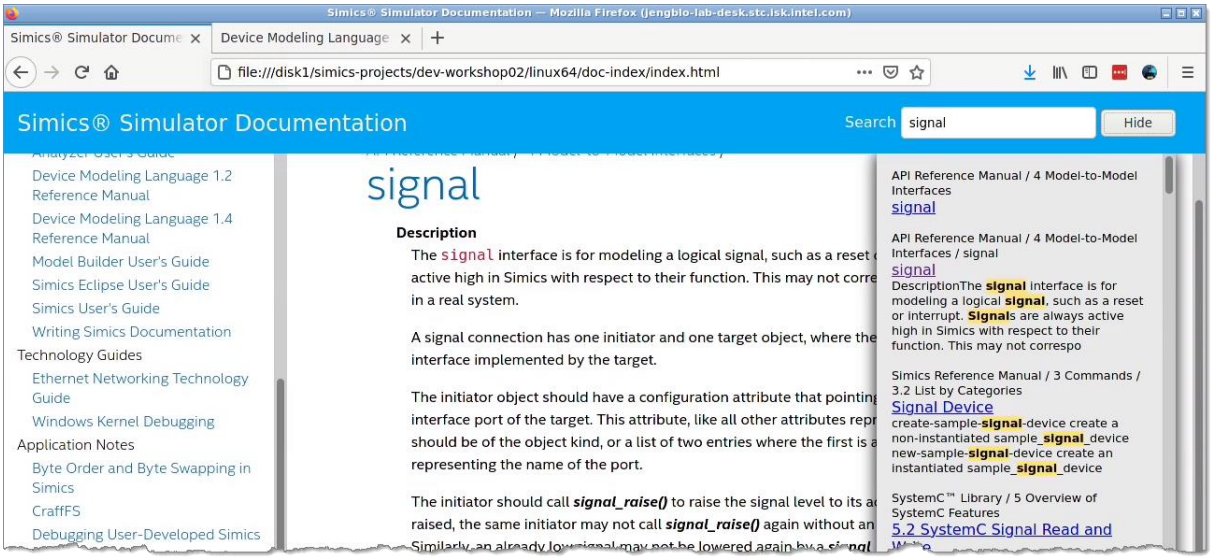
```
simics> api-search signal
```

This will show all **api-help** entries that contain the given string in their name or the help text.

27. The simulator documentation that you opened earlier contains the same information about the signal interface. **Go to** the documentation you have open in a browser window.

28. Enter "**signal**" into the search box.

    A list of results will appear. Select the entry called "**signal**" from the **API Reference Manual**:



29. Go back to the source code and the **template level_checked_signal**.

    Note how it defines implementations for **signal_raise()** and **signal_lower()** inside the **implements signal** block. When something calls the signal interface in the port **macc.control.port.done[N]**, these methods are called.

    The methods check the rules of the signal interface (you are not allowed to call raise or lower multiple times in a row) then call the **on_signal_raise()** and **on_signal_lower()** to perform the actual model actions corresponding to signal raising and lowering. These two methods are declared default, indicating that they can be overridden by objects using this template:

```
    method on_signal_raise () default {
        log info, 2: "Default implementation called that does nothing";
    }

    // Same for signal_lower.
    method on_signal_lower () default {
        log info, 2: "Default implementation called that does nothing";
    }
```

30. Check the code for **port done** again. It includes the template with an **is** statement.

```
port done[i<max_compute_units] is level_checked_signal {
    param desc = "Completion signal from compute units";
    ...
```

    It then implements specific versions of **on_signal_raise()** and **on_signal_lower()** to do the appropriate work in the model.

    Check that **on_signal_raise()** does indeed get called. Use inline Python to call the interface function of the port. To see the log messages, raise the log level to 3.

31. Raise the log level to 3 for **macc.control**:

```
simics> log-level macc.control 3
```

32. Call the method:

```
simics> @conf.macc.control.port.done[1].iface.signal.signal_raise()
```

The output indicates that a bit is being set in the **done** register.

33. Check that this is indeed the case:

```
simics> print-device-reg-info macc.control.bank.ctrl.done
```

It is a bit interesting that this happened when no operation was in progress. On the other hand, is this something the hardware model should check for? It is a judgement call how much checking to include in a device model.

34. Try raising the signal again:

```
simics> @conf.macc.control.port.done[1].iface.signal.signal_raise()
```

In this case, the model complains with a spec-violation.

35. Lower the signal to return the system state to where it was previously.

```
simics> @conf.macc.control.port.done[1].iface.signal.signal_lower()
```

6.5 Inspect the connection from the control unit to the compute units

36. Since the control unit controls multiple compute units, it uses arrays of connections. Go to the editor where **m_control.dml** is open, and search for "**connect compute_unit_control**".

The declaration looks like this:

```
connect compute_unit_control[i<max_compute_units] {
    param desc = "Connection to the compute unit control ports";
    …
    interface m_compute_control;
```

37. This declaration results in an attribute called **compute_unit_start** being added to the control unit model. The attribute value is expected to be a list of objects.

38. Check the current value of the attribute in the current configuration:

```
simics> macc.control->compute_unit_control
```

It is a list of eight elements, the first two of which point at compute units 0 and 1. The rest of the items are NIL, indicating that there is nothing connected to those slots.

39. The control unit also has an attribute indicating the total number of connected compute units. This attribute is used as the master in iterations inside the device model, and the model expects its value to be consistent with the list.

Look at the configuration attributes of the control unit:

```
simics> list-attributes macc.control
```

40. Another way to inspect the configuration of a model is to invoke its **info** command. Ideally, all device models should implement a custom **info** command to allow a quick and easy-to-read inspection of the device configuration. Adding such commands is

good practice, but the framework cannot enforce that. Thus, some models might have empty or misleading info commands.

The control unit model does have a useful **`info`** command:

```
simics> macc.control.info
```

41. Check the setup in the other subsystem, **`macc8`**:

```
simics> macc8.control.info
```

Here, all the slots are filled with references to compute units.

42. The configuration setup code is responsible for making sure that the configuration is consistent. Go to the **`003-control-unit-setup.py`** file (that you already opened above). Scroll down to the loop starting with "**`for i in range(N_units):`**"

For each created compute unit, the code needs to:

- Connect the control unit to the compute unit
- Connect the compute unit to the control unit
- Add the unit to the memory map

It also has to set up the compute unit configuration in the control unit correctly. Getting this right is actually very easy in code, since the number of compute units is a variable.

6.6 Inspect register memory mappings

The integrated setup adds the register memory map to the configuration.

43. Check the memory map of the **`macc`** subsystem, using the namespaced **`map`** command.

```
simics> macc.register_memory.map
```

The output should look like this:

Base	Object	Fn	Offset	Length	Target	Prio	Align	Swap
0x0000	macc.control.bank.ctrl		0x0000	0x0080		0	8	
0x0080	macc.compute[0].bank.ctrl		0x0000	0x0010		0	8	
0x00c0	macc.compute[1].bank.ctrl		0x0000	0x0010		0	8	

Note that the mapped objects are all register banks. The **`Base`** column indicates where it is mapped. **`Offset`** should be zero in most cases. The **`Length`** indicates the length of the mapping – note that there is empty space between the mapped devices. Ignore the rest of the columns for now.

44. Another way to view the memory map is using the **`memory-map`** command:

```
simics> memory-map macc.register_memory
```

45. Check the memory map of the **`macc8`** subsystem:

```
simics> macc8.register_memory.map
```

46. Look at the registers of the **macc.control** device's **ctrl** register bank:

```
simics> print-device-regs macc.control.bank.ctrl
```

At offset zero, there is an eight-byte register holding the number of attached compute units. At offset 32, there is another register holding a bitmap representing the same information.

6.7 An excursion into endianness

47. Read the register at offset zero, using a read operation to the memory map. You need to specify the offset, side of the read, and the interpretation of the bytes read (little endian in this case). Set the output radix to hexadecimal to make it easier to see the individual bytes:

```
simics> output-radix 16
simics> macc.register_memory.read address = 0 8 -l
```

Note about endianness.

Internally in a device model, the value of a register is typically saved as a single integer variable. This could be said to have no inherent endianness.

Endianness is applied when a device register is read from software or from the command line. At that point, an array in the memory operation is filled in with a sequence of byte values corresponding to either a little-endian or big-endian representation of the value, as determined by a param in the DML code. This is typically set for the entire device but can be set per bank if needed.

The **read** command then converts this byte array back to an integer value for presentation. This is where the **-l** and **-b** flags come in. Unlike the named register read commands, the read command only sees a sequence of bytes in memory and needs help to interpret them correctly.

48. Read the register using big-endian byte ordering (intentionally mis-interpreting the byte array):

```
simics> macc.register_memory.read address = 0 8 -b
```

49. Inspect the raw bytes using the **x** command:

```
simics> macc.register_memory.x address = 0 8
```

The result of the above should look something like this (all output in hex):

```
simics> macc.register_memory.read address = 0 8 -l
0x0002 (LE)
simics> macc.register_memory.read address = 0 8 -b
0x0200_0000_0000_0000 (BE)
simics> macc.register_memory.x address = 0 8
p:0x00000000  0200 0000 0000 0000                      ........
```

50. When reading the register using its name, the integer value is returned without any need to specify the endianness since that is given by the register metadata. Try reading the register using its name instead:

```
simics> read-device-reg macc.control.bank.ctrl.compute_units
```

6.8 Run a parallel compute job

Use the two-way parallel subsystem **macc** to compute a fractal, using two compute units to render it. This means setting up two descriptors, one for each compute unit, and having them write the results to two adjacent blocks of memory. By splitting the work vertically, it is trivial to combine the output of multiple compute units.

Consider the previously used descriptor plotting:

- 1000 pixels high
- 1000 pixels wide
- Bottom = 0.5
- Top = 0.7
- Left = -0.6
- Right = -0.4
- Results at 0x2000

The first descriptor would be:

- 500 pixels high
- 1000 pixels wide
- Bottom = 0.6 (this is the top half)
- Top = 0.7
- Left = -0.6, right = -0.4 (same as above)
- Results at 0x2000
- The descriptor can be located at the location used before, 0x1000

And the second:

- 500 pixels high
- 1000 pixels wide
- Bottom = 0.5
- Top = 0.6 (equal to *bottom* above)
- Left = -0.6, right = -0.4 (same as above)
- Results at 0x2000 + the amount of space taken by the above rendering. Which is 500 pixels by 1000 pixels by 16 bits, or exactly 1 million bytes.
- This is a separate descriptor, located at 0x1100

Doing this kind of work from the command line is a way to prototype the design of the software needed to use the accelerator. Later, when eventually writing the target software, the design will have been tested in a quick-turn-around interactive environment with much better error reporting than running actual software on the target system.

51. Raise the log-level on the **macc** subsystem to see what happens behind the scenes.

```
simics> log-level macc 2 -r
```

52. Create the top-half descriptor:

```
simics> @create_m_compute_descriptor(conf.macc.local_memory, 0x1000,
0x2000, 0.6, -0.6, 0.7, -0.4, 1000, 500, 256)
```

53. Create the bottom-half descriptor:

```
simics> @create_m_compute_descriptor(conf.macc.local_memory, 0x1100,
0x2000 + 1_000_000, 0.5, -0.6, 0.6, -0.4, 1000, 500, 256)
```

54. Set the descriptor pointer for compute unit zero, through the register memory map. Find the address locally in the bank:

```
simics> print-device-regs "macc.compute[0].bank.ctrl"
```

55. Write to offset zero, adding in the mapping address of compute unit 0 (which is **0x80**, as seen above from the memory-map command):

```
simics> macc.register_memory.write 0x80 0x1000 8 -l
```

56. Do the same for compute unit one, which is mapped from **0xc0**:

```
simics> macc.register_memory.write 0xc0 0x1100 8 -l
```

57. Check that the descriptor registers have indeed been set:

```
simics> print-device-regs "macc.compute[0].bank.ctrl"
simics> print-device-regs "macc.compute[1].bank.ctrl"
```

58. To start an operation, write the **start** register of the control unit. This will in turn start the work in each compute unit using the control interface. The value to write to the **start** register is the number of units to use for the job, which could be less than the maximum. The **start** register is found at offset **0x08**.

```
simics> macc.register_memory.write 0x08 2 8 -l
```

    Log messages will be printed indicating that the compute units have been activated.

59. Check the **status** register of the control unit:

```
simics> print-device-reg-info macc.control.bank.ctrl.status
```

    The **processing** field is set and the **done** field is not, indicating an operation in progress.

60. Check the events posted:

```
simics> peq
```

    Check that there are two completion events posted, one for each compute unit.

61. Run until the computation has completed, using a notifier. The control unit uses the same notifier name as the compute unit. To wait for it from the control unit specifically, provide an object in addition to the notifier name to the **run-until** command:

```
simics> bp.notifier.run-until object = macc.control name = m-compute-
complete
```

    When the simulation stops, the log messages should indicate that the computation in each compute unit completed, and after that that the control unit received signals indicating that the compute units are in “done” state.

62. Check the current time in the simulation. Note that there is one clock inside each of the accelerator subsystems created. Check both using **-all** to **ptime**:

```
simics> ptime -all
```

Despite having nothing to do, the clock in the **macc8** subsystem has moved forward. This is expected, as the Simics simulation framework makes sure to run all clocks in the system (with a maximum difference equivalent to the current time quantum).

63. Check the contents of the done register in the control unit. This is a bit mask that tracks the completion status of each compute unit used.

```
simics> print-device-reg-info macc.control.bank.ctrl.done
```

64. Check the status register of the control unit:

```
simics> print-device-reg-info macc.control.bank.ctrl.status
```

The **done** bit is set and the processing bit is zeroed.

65. Check the status register of compute unit zero:

```
simics> print-device-reg-info "macc.compute[0].bank.ctrl.status"
```

The register also has its done bit set.

66. Clear the **done** state for the whole accelerator subsystem by writing a 1 to the **done** bit. There is no count value to worry about:

```
simics> macc.register_memory.write 0x10 0x8000_0000_0000_0000 8 -l
```

The log messages indicate that the control unit reaches out to the compute units via the **clear_done** signal to clear all their "done" state. They also lower their respective done signals towards the control unit.

67. The done register should now be all zero. Check it:

```
simics> print-device-reg-info macc.control.bank.ctrl.done
```

68. Also check the status register of compute unit zero:

```
simics> print-device-reg-info "macc.compute[0].bank.ctrl.status"
```

This has also been cleared (as should be obvious from the log messages).

6.9 Display the results

The above focused on the control flow between the control unit and the compute units. Next, check that the results are correct by displaying them. The display code used previously is already loaded by the start scripts.

69. Run the simulation to avoid the greying out of the console that the Simics simulator applies any time the simulation is stopped. To see the full-color version, make sure to run the simulation.

```
simics> r
```

Note that the virtual time proceeds very quickly since there is nothing happening in the simulation. This is harmless.

70. Call **display_m_result()** like above – but note that the objects have changed name since they have been put into the **macc** namespace:

```
running> @display_m_result(conf.macc.con, conf.macc.local_memory,
0x2000, 1000, 1000, 256)
```

71. Check the current time a few times:

```
running> ptime -all
```

6.10 Error handling/specification violations

With the simulation running, test the error handling of the control unit registers. Bad use of registers should result in specification violation log messages from the model.

72. Clear the done state of the status register again (it is already cleared):

```
running> macc.register_memory.write 0x10 0x8000_0000_0000_0000 8 -l
```

73. Write to the read-only compute unit **count** register:

```
running> macc.register_memory.write 0x00 0xffff 8 -l
```

74. Write an invalid value to the **start** register:

```
running> macc.register_memory.write 0x08 0xffff 8 -l
```

75. To see where these log messages come from in the code, go back to the **m_control.dml** file that should still be open in an editor. Search for the last message printed, "**Invalid value**".

    You should find this code:

```
    if ((unitcount==0) ||
        (unitcount > connected_compute_unit_count.val )) {
        log spec_viol, 1, control :
        "Invalid value for compute start: requested %d (expected 1 to %d).",
            unitcount, connected_compute_unit_count.val;
        return;
    }
```

    Such checks for input from the software are good modeling practice. They also help protect the model functionality from bad inputs, making the model more robust.

76. Exit this simulation session.

```
simics> exit
```

7 Package the Accelerator as a Component

To easily create the accelerator subsystem and facilitate its connections to other part of the simulated system, a Simics simulator **component** is used.

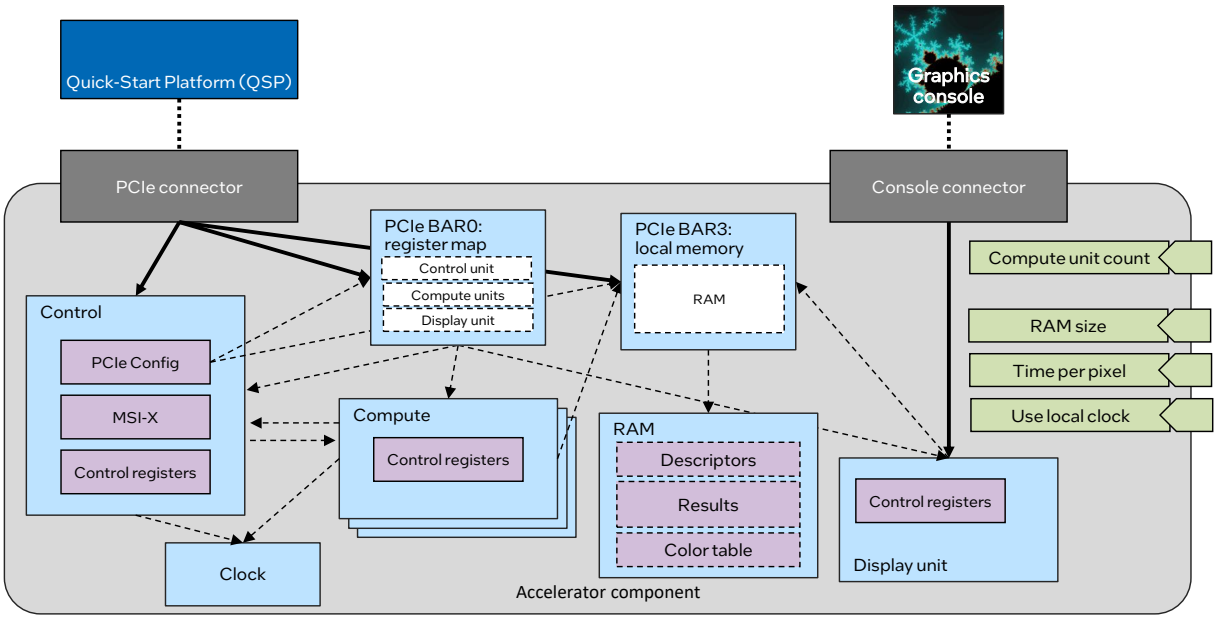
Logically, a **component represents a subsystem** (set of objects) and its high-level connections to the rest of the system. An instance of a component is created using a custom command-line command. It is connected to other components using component-level command-line connect commands. Thus, the component for the accelerator subsystem can be treated as a virtual PCIe card that is connected to a virtual PCIe slot on the virtual quick-start platform. Components are also configurable with component-level configurations, such as the number of compute units in the accelerator.

Physically, the **implementation** of a component is a Simics simulator class written in Python, using the components framework. The code in the component creates objects and the internal connections between them in a way very similar to the Python scripts used previously in this workshop. The component is present in the simulation configuration as an object that provides a **namespace** for the **objects** in the subsystem. There is also code in the components that take action when one component is connected to another component and sets up the references between the objects inside the components. Component connectors are not involved when the connected objects communicate during the simulation.

Components are mostly used when setting up a new target system, but they can also be used at run time to do things like create a new USB disk to connect to running target system. Like everything in the Simics simulator, components can be created and connected during run time, and their connections can be changed (where that makes sense).

Component creation can be done in two ways. Typically, when setting up a system, all components are created in **non-instantiated form** first and connected together. This essentially creates a component-level template for the system to create. Then, once all components are in place, the **`instantiate-components`** command is used to cause all the objects to be created. Essentially, the code in the components creates a set of pre-configuration objects which are then sent to **`SIM_add_configuration()`** when the **`instantiate-components`** command is called. The commands creating non-instantiated components are called **`create-`*`X`***, where ***`X`*** is the name of the component. There are also commands called **`new-`*`X`***, which instantiate the component immediately.

The component for the accelerator subsystem looks like the below, with the internal connections simplified compared to previous illustrations. It has two connectors, one to PCIe to connect to the QSP, and one to a graphics console.



The component also introduces the **display unit** model, driving the graphics console "from hardware" instead of drawing from a script as was done previously. The display unit has a set of control registers and needs a software driver just like all the other hardware units.

7.1 Copy the component source code to the project

To have access to the source code, copy it to the project and open it in an editor.

1. Copy the component source code to the project. It counts as a device.

```
$ bin/project-setup --copy-module m-accelerator-comp
```

2. **Open** the file **[project]/ modules/m-accelerator-comp/m_accelerator_comp.py** in an editor, to get the source code of the component.

3. Build all the modules in the project, including the component.

```
$ make
```

7.2 Test the component stand-alone

4. Start a new simulation from script **004-**:

```
$ ./simics targets/workshop-02/004-use-component.simics
```

5. List the objects in the simulation configuration:

```
simics> list-objects
```

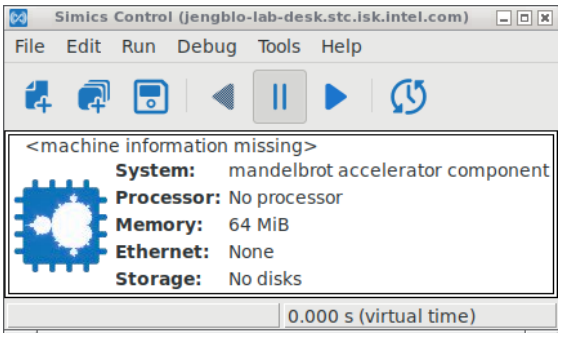
6. Check the help on the component-creation command for the accelerator:

```
simics> help new-m-accelerator-comp
```

7. Create a new accelerator called **macc**, with 8 compute units:

```
simics> new-m-accelerator-comp macc compute_units = 8
```

Note that the simple GUI control window updated to show an icon for the accelerator. The system information is coded into the component.



8. Use the status command to check the configuration attributes of the component:

```
simics> macc.status
```

9. Use the info command to inspect the contents of the component:

```
simics> macc.info
```

This command shows the "slots" of the component, i.e., the objects contained in it.

```
simics> macc.info
Information about macc [class m_accelerator_comp]
================================================

Slots:
                cell : macc.cell
               clock : macc.clock
          compute[0] : macc.compute[0]
          compute[1] : macc.compute[1]
          compute[2] : macc.compute[2]
          compute[3] : macc.compute[3]
          compute[4] : macc.compute[4]
          compute[5] : macc.compute[5]
          compute[6] : macc.compute[6]
          compute[7] : macc.compute[7]
             console : macc.console
             control : macc.control
             display : macc.display
        local_memory : macc.local_memory
                 ram : macc.ram
    register_memory : macc.register_memory

Connectors:
             console : graphics-console     down  hotplug
```

It also shows a single connector – to a graphics console. There is no PCIe connector since no PCIe support was requested when setting up the component.

10. Inspect the objects of the component using the list-objects command. Hide the port objects to keep the output reasonably short:

```
simics> list-objects -tree namespace = macc -hide-port-objects
```

The output looks like this:

```
simics> list-objects -tree namespace = macc -hide-port-objects
┐
├ cell ┐
│      └ ps
├ clock ┐
│       └ vtime ┐
│               ├ cycles
│               └ ps
├ compute[0..7]
├ console
├ control
├ display
├ local_memory
├ ram ┐
│     └ image
└ register_memory
```

Note the object called **`console`** – it is the component connector used to connect from the accelerator subsystem to the graphics console. Each connector has a corresponding object in the object hierarchy.

11. Create a graphics console for the connector to connect to:

```
simics> new-gfx-console-comp gcon
```

12. Check the contents of the **`gcon`** component:

```
simics> gcon.info
```

This has a connector called **`device`**, of type **`graphics-console`**, and direction **`up`**. This matches the console connector of the accelerator, which has the same type but the direction down. Thus, these two connectors are connectable.

13. Connect them together:

```
simics> connect macc.console gcon.device
```

The console will update its contents and size to correspond to the default "empty" state of the display unit. The display unit drives and update to the console object on connection.

7.3 Look at the component source code

Now that you have seen the component in action, have a look at its source code.

14. Go to the file **`[project]/modules/m-accelerator-comp/m_accelerator_comp.py`** that you should have opened in an editor.

15. Look at the component declaration. It is a Python class that inherits from the **`StandardComponent`** class.

```
class m_accelerator_comp(StandardConnectorComponent):
    """Component for the mandelbrot accelerator subsystem."""
    _class_desc = "mandelbrot accelerator component"
    _help_categories = ()
```

16. Scroll down to find the component configuration arguments. They are declared as subclasses inside the component class. For example:

```
    class compute_units(SimpleConfigAttribute(2,"i")):
        """Number of compute units in this accelerator instance."""
        ...
```

17. The core functionality of the component is contained in the **`add_objects()`** method.

```
    def add_objects(self):
        ...
        # Memory
        ram = self.add_pre_obj('ram', 'ram')
        ram_image = self.add_pre_obj('ram.image', 'image')
        ram_image.attr.size = self.ram_size.val
        ram.attr.image = ram_image
        local_memory.attr.map = [[0x0000, ram, 0, 0, self.ram_size.val]]
        ...
```

Note how similar this code is to the code used in scripts to set up ad-hoc Simics simulation configurations. It uses a wrapping around pre-conf objects that is specific to components (**`add_pre_obj`**) and that automatically puts the created object into the component namespace. There is no call to **`SIM_add_configuration()`**, as that is taken care of by the components framework.

There is no assignment to the queue attribute of objects either, as that is also handled by the component system. If the component is set up with a local clock that will be used, otherwise the component system will find a clock to use (typically the first processor of the machine that the accelerator is connected to over PCIe),

18. The connections to other components are set up in the **`setup()`** method, found towards the start of the file.

```
    def setup(self):
        super().setup()
        ...
        ## Add connector to the graphics console
        self.add_connector('console',
                           connectors.GfxDownConnector('display', 'console'))
        ...
```

The connections are added using **`add_connector()`**, using pre-defined connector types (modelers can also define their own connector types). All that is needed is to provide a name for the connector itself (**`'console'`**) and indicate the actual object that is the target of the connection (**`'display'`**).

19. To see some basic documentation on a component connector, use Python **`help`**.

```
simics> @help ( connectors.GfxDownConnector)
```

7.4 Run a compute job

Set up descriptors in the same way as in Section 6.8.

20. Create the top-half descriptor. Note that the name of the **local_memory** object is the same as above, since the component was created with the same name as the subsystem.

```
simics> @create_m_compute_descriptor(conf.macc.local_memory, 0x1000,
0x2000, 0.6, -0.6, 0.7, -0.4, 1000, 500, 256)
```

21. Create the bottom-half descriptor:

```
simics> @create_m_compute_descriptor(conf.macc.local_memory, 0x1100,
0x2000 + 1_000_000, 0.5, -0.6, 0.6, -0.4, 1000, 500, 256)
```

22. Write the descriptor registers using their names (for variety):

```
simics> write-device-reg macc.compute[0].bank.ctrl.descriptor_addr
0x1000
simics> write-device-reg macc.compute[1].bank.ctrl.descriptor_addr
0x1100
```

23. Start the operation:

```
simics> write-device-reg macc.control.bank.ctrl.start 2
```

24. Raise the log level to see what happens:

```
simics> log-level macc 2
```

25. Run the simulation forward:

```
simics> r
```

    The operation will complete very quickly, leave the simulation running.

7.5 Display results using the display unit

To display the results using the display unit, it is necessary to first set up a color table (mapping iteration values computed by the compute units to RGB color values). Then, the display unit needs to be configured with information about the size of the results and where in memory the results are found.

Finally, a redraw request will pick up the results of the compute, convert each pixel to an RGB value, and send it to the console. Technically, the model actually maintains an internal buffer that contains the complete display state, since the graphics console model does not have that responsibility.

26. List the control registers of the display unit:

```
running> print-device-regs macc.display
```

27. Check where it is mapped in the register space of the accelerator subsystem:

```
running> macc.register_memory.map
```

    The control registers for the display are mapped from offset 0x300 and forward. This is useful to write the target software, but for now the interactive exploration will use named register accesses.

28. Set the size of the display:

```
running> write-device-reg macc.display.bank.regs.width 1000
running> write-device-reg macc.display.bank.regs.height 1000
```

29. Set the number of iterations used (256):

```
running> write-device-reg macc.display.bank.regs.max_iter 256
```

30. Set the address of the results (**0x2000**):

```
running> write-device-reg macc.display.bank.regs.iter_data_addr 0x2000
```

31. At this point, a color table is needed. The color table will need to cover 257 values (from zero to the maximum iteration count), each for 4 bytes. Which requires just about 1KiB of RAM. Given that there is a huge memory bank on the accelerator, this can be jammed in between the descriptors and the results area.

    Check where a table containing 257 entries, and starting at **0x1200** would end up:

```
running> hex 0x1200 +  4 * 257
```

    The value is below **0x2000**, so a color table at **0x1200** makes sense.

32. Set the color table pointer register:

```
running> write-device-reg macc.display.bank.regs.color_table_addr
0x1200
```

33. Check the configuration of the display unit using its status command:

```
running> macc.display.status
```

34. To create the color table, use a ready-made Python helper script. Load the Python file (it is located in the installation, you can take a look at it if you want to):

```
running> run-python-file "%simics%/targets/workshop-02/004-build-color-
table.py"
```

35. Use the Python function **create_color_table()** to create a color table. It takes four arguments: the memory space to write the result to, the address of the table, the maximum iteration value, and the Python coloring function to use. There are two coloring functions provided, **colorize_1** and **colorize_2**.

    Use **colorize_1**:

```
running> @create_color_table(conf.macc.local_memory, 0x1200, 256,
colorize_1)
```

36. Check the resulting table in memory (using little-endian display makes it easier to read, as each word will then be rendered as **#00RRGGBB**):

```
running> macc.local_memory.x 0x1200 group-by = 32 -l 1028
```

37. Finally, update the display. Write **1** to the **update** register:

```
running> write-device-reg macc.display.bank.regs.update 1
```

This should result in a fairly subdued picture.

38. Try the other available coloring function:

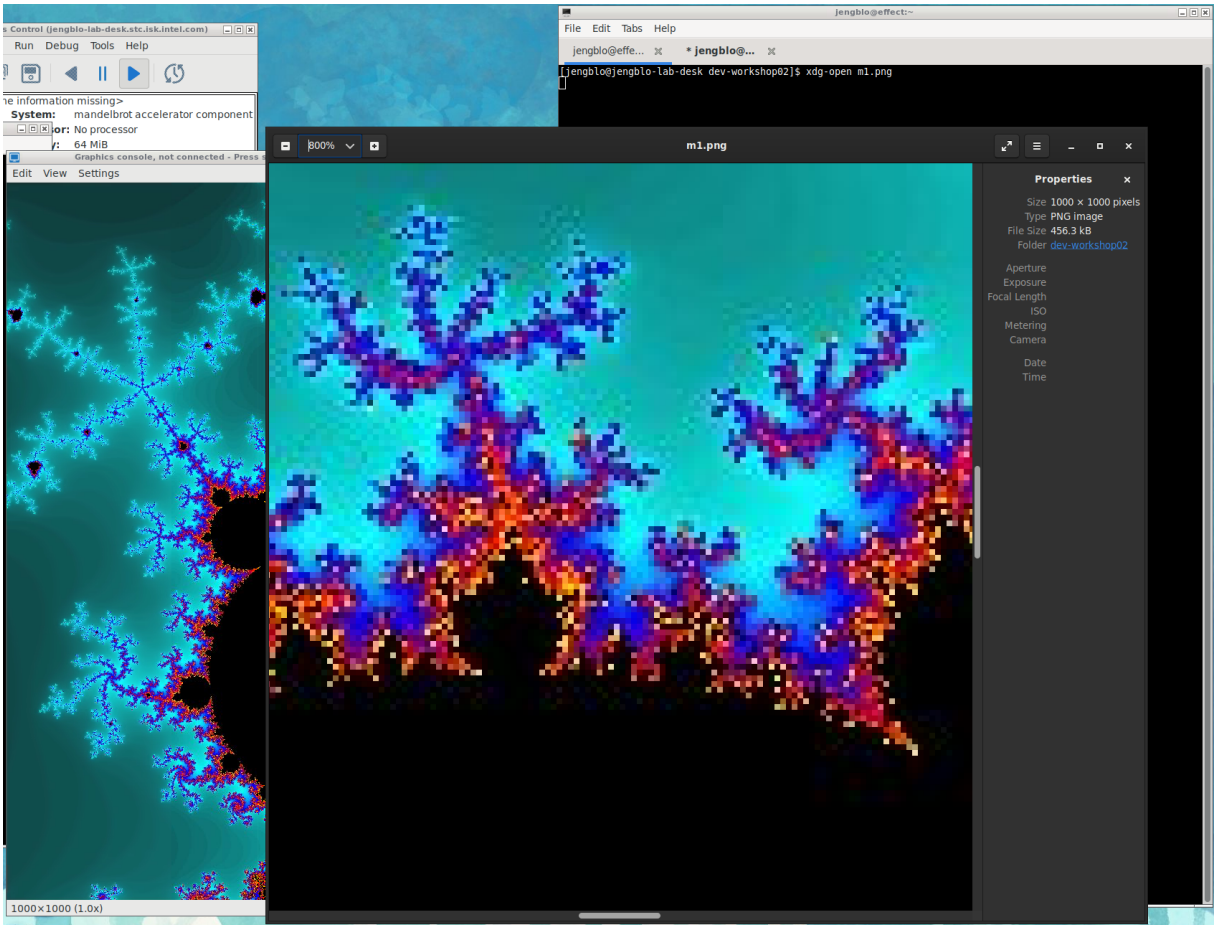
```
running> @create_color_table(conf.macc.local_memory, 0x1200, 256,
colorize_2)
running> write-device-reg macc.display.bank.regs.update 1
```

This is a bit more colorful.

39. The Simics simulator graphics console can save the displayed image as a PNG file, using the screenshot command on the console object. Try it:

```
running> gcon.con.screenshot m1.png
```

40. Go to a file browser, locate your Simics project, and open the image. Exactly how depends on how your Linux or Windows host is configured. Here is one example, including zooming in to see the colored pixels at the edge of the fractal:



41. Exit this simulation session.

```
running> exit
```

7.6 Connect the accelerator using PCIe

The accelerator is designed to be connected to the QSP virtual platform over PCIe. This is all automated in the **006-** script that you tried back in section 3. It can also be done manually.

42. Run the script **006-** with the parameter **add_accelerator** set to **FALSE**. This will skip the accelerator setup and leave you with a standard QSP setup.

```
$ ./simics targets/workshop-02/006-accelerator-in-qsp.simics
add_accelerator=FALSE
```

43. Check that there is no accelerator present:

```
simics> list-objects namespace = qsp
```

44. Create a new uninstantiated accelerator component, with no clock but with PCIe enabled. Use **$system** to get name of the top-level QSP machine, like it is done in typical setup scripts:

```
simics> create-m-accelerator-comp $system.macc compute_units = 8
use_pcie = TRUE use_clock=FALSE
```

45. Connect the accelerator to the QSP using PCIe. This requires some knowledge about the hardware in the QSP. The QSP is a traditional personal computer (PC) where there is a north bridge close to the processors, and a south bridge that holds slower input and output devices. In this case, the PCIe connectors on the north bridge should be used.

    Check the available connections on the "north bridge" component in the QSP machine:

```
simics> $system.mb.nb.info
```

46. The slot used by the defaults of script **006-** is **pcie_slot[1]**. The slot used affects the PCI bus number and thus the **/sys/bus** file system path required by the **m-app** program. Try a different slot instead, **pcie_slot[2]**:

```
simics> connect qsp.macc.pci "qsp.mb.nb.pcie_slot[2]"
```

47. Create a new uninstantiated graphics console component:

```
simics> create-gfx-console-comp $system.macc.gcon
```

48. Connect the accelerator and the graphics console:

```
simics> connect qsp.macc.console qsp.macc.gcon.device
```

49. Instantiate the components:

```
simics> instantiate-components
```

50. In case the new graphics console is not visible, use the show command:

```
simics> qsp.macc.gcon.con.show
```

51. Check the accelerator connections:

```
simics> qsp.macc.status
```

It should be connected to the console and PCIe, like this:

```
simics> qsp.macc.status
Status of qsp.macc [class m_accelerator_comp]

...

Connections:
               console : qsp.macc.gcon:device
                   pci : qsp.mb.nb:pcie_slot[2]
```

52. Check the connections from the north bridge:

```
simics> qsp.mb.nb.status
```

You should see the **pci** connector on the **qsp.macc** component being connected to the **pcie_slot[2]** connector.

53. Run the simulation to boot the target system:

```
simics> r
```

54. **Once the target system has booted**, check where the accelerators ended up from the perspective of the target software. Use the known vendor and device ID as a filter to **lspci**.

Go to the **target system serial console** and enter:

```
# lspci -d 8086:0d5f
```

This should show the device on bus **03**, instead of bus **02** as in the introduction.

55. Stop the simulation.

```
running> stop
```

7.7 Dig deeper into the PCIe modeling

Time to look a bit deeper at how PCIe works in the Simics simulator. Unfortunately, the PCIe slot connector names has no direct relationship to the models of the PCIe ports in the north bridge, or the software-exposed bus numbers. They all follow from the hardware design the model is based on, and that is not one-to-one. Instead of guessing at names it is better to follow the trace from the accelerator through the model.

The control unit in the accelerator subsystem implements the PCIe functionality for the accelerator – configuration bank, mapping of BARs, etc.

56. To find the actual PCIe "bus" that the accelerator is connected to, check the value of the **pci_bus** attribute on the control unit:

```
simics> list-attributes qsp.macc.control substr = pci_bus
```

The value should be **qsp.mb.nb.pcie_p3.downstream_port**.

57. List the objects inside the downstream port:

```
simics> list-objects -tree namespace =
qsp.mb.nb.pcie_p3.downstream_port
```

58. The **cfg_space** holds the configuration banks of the objects on the bus. Use the **map** command to see the devices on the port:

```
simics> qsp.mb.nb.pcie_p3.downstream_port.cfg_space.map
```

The configuration register bank is mapped using a "function number". This is a convention used with PCI and PCIe in the Simics framework.

```
simics> qsp.mb.nb.pcie_p3.downstream_port.cfg_space.map
```

Base	Object	Fn	Offset	Length	Target	Prio	Align	Swap
0x0000	qsp.macc.control	255	0x0000	0x0001_0000		0	8	

…

The length, **0x1_0000** bytes (4192), indicate that this is a PCIe extended configuration space. Old PCI just used 256 bytes.

59. Since the target system is booted, software has set up the memory mappings of the device using the PCIe Base Address Registers (BARs). Check out the mappings in the PCIe port's mem_space (used for PCI "memory" accesses):

```
simics> qsp.mb.nb.pcie_p3.downstream_port.mem_space.map
```

60. Compare this to the values written to the BAR registers:

```
simics> output-radix 16
simics> print-device-regs qsp.macc.control.bank.pci_config pattern =
"base_address*"
```

The mapped addresses align with the addresses in the BARs.

```
simics> qsp.mb.nb.pcie_p3.downstream_port.mem_space.map
```

Base	Object	Fn	Offset	Length	Target	Prio	Align	Swap
0xf000_0000	qsp.macc.local_ memory		0x0000	0x0400_0000		0		
0xf400_0000	qsp.macc.control. bank.dev_msix_pba		0x0000	0x0100		0	8	
0xf400_1000	qsp.macc.control. bank.dev_msix_table		0x0000	0x0100		0	8	
0xf400_2000	qsp.macc.register_ memory		0x0000	0x1000		0		

```
…
simics> print-device-regs qsp.macc.control.bank.pci_config pattern = "base_address*"
Offset  Name            Size        Value
----------------------------------------
0x0010  base_address_0     4  0xf400_2000
0x0014  base_address_1     4  0xf400_1000
0x0018  base_address_2     4  0xf400_0000
0x001c  base_address_3     4  0xf000_0000
0x0020  base_address_4     4       0x0000
0x0024  base_address_5     4       0x0000
```

61. The next question is how these addresses are mapped from the perspective of the processors in the system. All the processors in the simulated system are found using the **`list-processors`** command:

```
simics> list-processors
```

62. Pick the first processor listed, and check its **`info`** command:

```
simics> qsp.mb.cpu0.core[0][0].info
```

The object of interest is the "physical memory" of the processor. This is the memory space where all memory operations from the processor are sent. It uses physical addresses, not the virtual or logical addresses used in software.

63. List the memory map of the processor's physical memory:

```
simics> qsp.mb.cpu0.mem[0][0].map
```

This contains a mapping for the APIC connected to the core, and then a default mapping. Any access not hitting the APIC will go to **`qsp.mb.phys_mem`**, which is common to all the processors in the system.

64. List the memory map of the common memory:

```
simics> qsp.mb.phys_mem.map
```

This shows several RAM mappings, plus a default mapping onwards.

65. Follow the trail into the PCI mapping:

```
simics> qsp.mb.nb.pci_bus.port.mem.map
```

Here there is a mapping for the **pcie_p3.downstream_port.port.mem**:

```
┌──────────┬───────────┬───┬──────────┬────────┬────────┬────┬─────┬────┐
│      Base│Object     │ Fn│    Offset│  Length│Target  │Prio│Align│Swap│
├──────────┼───────────┼───┼──────────┼────────┼────────┼────┼─────┼────┤
…
├──────────┼───────────┼───┼──────────┼────────┼────────┼────┼─────┼────┤
│0xf000_0000│qsp.mb.nb. │   │0xf000_0000│0x0410_ │        │   3│     │    │
│          │pcie_p3.   │   │          │    0000│        │    │     │    │
│          │downstream_│   │          │        │        │    │     │    │
│          │port.port. │   │          │        │        │    │     │    │
│          │mem        │   │          │        │        │    │     │    │
├──────────┼───────────┼───┼──────────┼────────┼────────┼────┼─────┼────┤
…
```

This mapping has **Offset=Base**, which means that memory accesses will retain the full address when passed on. Thus, the memory addresses shown by **qsp.mb.nb.pcie_p3.downstream_port.mem_space.map** map directly to what comes out of the processor.

66. Write to the first mapped address (**0xf000_0000**), from the processor's memory map. This is equivalent to issuing an access to physical address **0xf000_0000** from code in the processor.

```
simics> qsp.mb.cpu0.mem[0][0].write 0xf000_0000 0xcafef00d 4 -l
```

67. Check that the local memory was updated:

```
simics> qsp.macc.local_memory.x 0x00 group-by = 32 -l
```

7.8 Drive the target software using the Simics simulator command line

Next, it is time to test that the accelerator works as intended. Instead of typing the commands on the target console, use **<con>.input** commands from the Simics CLI to direct input to the target system. Note that each line has to end with **\n** to actually press enter on the target system to get the command executed. The serial console is represented by the object **qsp.serconsole.con**.

68. Run the simulation again, so that it can respond to command-line commands:

```
simics> r
```

69. Insert the driver:

```
running> qsp.serconsole.con.input "insmod m-acc-pcie-driver.ko\n"
```

70. Check the results using **dmesg**:

```
running> qsp.serconsole.con.input "dmesg | tail -20\n"
```

71. Set up a memory access breakpoint to accesses to the **local_memory** on the accelerator. Cover the whole RAM, to make no assumptions about the software behavior. This means starting at offset zero and going on for **0x400_0000** bytes. Check the memory map first to see where the offset comes from:

```
running> qsp.macc.local_memory.map
running> bp.memory.break qsp.macc.local_memory 0x0 0x400_0000 -w
```

72. Next, use the application to display a fractal. Remember to use bus **03** instead of **02** in the device argument to the bus! Note that the backslash characters in the path to the device must be escaped so that they come out right:

```
running> qsp.serconsole.con.input "./m-app mandel3.txt 1 1
/sys/bus/pci/devices/0000\\:03\\:00.0/\n"
```

73. The simulator will stop on the first memory access from software. The message should look something like this:

```
[qsp.macc.local_memory] Breakpoint 3: qsp.macc.local_memory 'w' access to p:0x100 len=4
val=0x101010
```

    The number written looks a lot like a color value. Could this be the start of the color table?

74. Check this hunch by removing the breakpoint and instead setting up a trace on the memory instead. This will log memory accesses that hit the defined area, but will not stop the execution.

```
simics> bp.delete -all
simics> bp.memory.trace qsp.macc.local_memory 0x000 0x04000000 -w
```

75. Run the simulation:

```
simics> r
```

    The result is a fairly long trace of memory accesses.

76. Pause the simulation once the fractal is displayed.

```
running> stop
```

77. Scroll back up and look at the memory accesses. There are a long series of 4-byte accesses starting at address **0x100** and going on to **0x8d0**.

    Compute the distance:

```
simics> (0x8d0 - 0x100) / 4
```

    This agrees with the output from the serial console, which indicates that the **maxiter** value for this particular Mandelbrot specification was 500. Having 501 values in the color table makes sense, since the range of iteration values go from zero to **maxiter** (where obviously zero is kind of silly, but it simplifies the indexing code).

78. Next, a set of writes to offset **0x2000** to **0x2020** represents the descriptor. Note the unnecessary write to address **0x201c**, the padding word.

79. Finally, there is a single very large write. This is the accelerator model saving the results from its internal buffers to the simulated memory in one single step.

80. To get a better idea for the software interaction with the devices as well as the memory, add a memory trace on the register memory:

```
simics> bp.memory.trace qsp.macc.register_memory 0x0000 0x0400 -w
```

81. The timing of each memory access can also shed light on the software behavior, and what takes time. Trace messages follow the log-setup settings. Set up basic time-stamping, which will print the following for each message: the current processor or clock at the time the operation happened, its current instruction pointer/program counter, and the current cycle count.

```
simics> log-setup -time-stamp
```

82. Set up some logging in the accelerator:

```
simics> log-level qsp.macc 2
```

83. Run the simulation:

```
simics> r
```

84. Repeat the previous command on the serial console by sending **Up arrow** followed by **Enter**. This can be done using the -e flag to the input command, which provides for Emacs-style keystroke sequences:

```
running> qsp.serconsole.con.input -e "Up Enter"
```

85. After the display unit logs that it is displaying the picture, stop the simulation.

```
simics> stop
```

86. Scroll back up to the color table setup, and note how it the writes happen with an interval of between 200 and 300 cycles. There is probably room for improvement there, but it does not really matter compared to the time spent waiting for the accelerator to complete. For example, here are some cycle numbers from one run:

```
[bp.memory trace] {qsp.mb.cpu0.core[1][0] 0x401208 93347821313132} [trace:4]
qsp.macc.local_memory 'w' access to p:0x818 len=4 val=0xffac5c
[bp.memory trace] {qsp.mb.cpu0.core[1][0] 0x401208 93347821313395} [trace:4]
qsp.macc.local_memory 'w' access to p:0x81c len=4 val=0xffae5d
[bp.memory trace] {qsp.mb.cpu0.core[1][0] 0x401208 93347821313658} [trace:4]
qsp.macc.local_memory 'w' access to p:0x820 len=4 val=0xffb05e
```

87. Find the write that starts the compute operation:

```
[bp.memory trace] {…} […] qsp.macc.register_memory 'w' access to p:0x8 len=4 val=0x1
```

After this write, several log messages have the same clock cycle count, ending with the large write of results to memory. Everything happens at the same instance in virtual time, since that is how the model has been designed.

Using time-stamped logs is a good way to understand the simulation flow and operation timing.

88. Exit the simulation. This concludes the component lab.

```
simics> exit
```

8 Test Performance in Virtual and Real Time

The accelerator performance can be measured in both virtual and real time. Virtual time measures the performance as seen from the target system, where the main variable is the use of parallelism as well as the size of the data to draw. Real time measures how quickly the simulator can complete each simulation job, primarily affected by how the compute unit model is implemented.

8.1 Set up checkpoint

To test performance, it is necessary to run from a booted system with the kernel driver installed and the application available. To avoid having to boot the system each time, save a checkpoint after the boot and use this for further testing.

1. **Start** the Simics simulator using the script **`007-`**:

   ```
   $ ./simics targets/workshop-02/007-prep-system-benchmarking.simics
   ```

2. **Run** the simulation. The script will take care of booting the target system, testing the application, and saving a checkpoint automatically. You can inspect the script in the installation to see what it does.

   ```
   simics> r
   ```

   The script will stop the simulator once the checkpoint has been saved. It will print the name and path of the checkpoint, for reference.

3. List the checkpoints that the Simics simulator knows about – basically, checkpoints in the current project.

   ```
   simics> list-checkpoints
   ```

   There should be a checkpoint with the name printed from the script, and with a comment explaining what it is:

   ```
   simics> list-checkpoints
   …
   ws02-setup-for-benchmarking.ckpt
     Target system booted to prompt, driver installed, ready to run benchmarks for
     mandelbrot accelerator.
   ```

4. Exit this simulation session.

   ```
   simics> exit
   ```

5. From the host shell, check the contents of the checkpoint.

   ```
   $ ls -lh ws02-setup-for-benchmarking.ckpt/
   ```

   The biggest file is the memory image, representing the changes to RAM from the UEFI and Linux boot, as well as operations after the boot like running the **`m-app`** application.

8.2 Run baseline performance test

The baseline test is running with no parallelism in the target software use of the hardware, no threading in the compute unit model, no stall optimization in the control unit (see below), and with all target-visible hardware delays set to defaults.

6. Start the Simics simulator using the script **`008-`**, with default settings:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
```

7. Run the simulation:

```
simics> r
```

This will automatically run the m-app on the target, rendering the **`bench1.txt`** file. The test file contains 100 images, at an iteration level of 200. The test is run three times in a row, to provide a more meaningful average.

After each run of the software run complete, it will print the virtual and real time consumed. Something like this:

```
Run 1/3
  Host time (real time)      (s) : 125.74
  Virtual time (target time) (s) : 0.9721

Run 2/3
  Host time (real time)      (s) : 124.22
  Virtual time (target time) (s) : 0.9615

Run 3/3
  Host time (real time)      (s) : 125.66
  Virtual time (target time) (s) : 0.96082

Averages:

  Host time (real time)      (s) : 125.21
  Virtual time (target time) (s) : 0.9648

Stopping simulation
```

If this test completes in less than a minute of real time, it might be a good idea to switch to a heavier benchmark. To do that, add the argument "**`test_file=bench2.txt`**" or even "**`test_file=bench3.txt`**" to the command line.

8. Note down the virtual and real time execution for this experiment, to have something to compare later runs to. The precise results will vary with the host.

9. Exit this simulation session.

```
simics> exit
```

8.3 Look at the benchmarking script

10. Use an editor to open the script **`008-system-benchmarking.simics`**, as found in the Simics installation (not in your project). See Section 2.3 above for how to locate the installation.

11. Note how the input data file (**test_file**) to use, level of target compute parallelism, and the number of repetitions of the test to run are declared as parameters to the script.

```
decl {
! Start from checkpoint, run benchmarks

    param test_file : string = "bench1.txt"
    ! Test file to use on target system

    param parallelism : int = 1
    ! Parallelism to use in the run (1-8 unless you changed something)

    param repeats : int = 3
    ! Number of times to repeat test

    param checkpoint_name : string = "ws02-setup-for-benchmarking.ckpt"
    ! Name of checkpoint
}
```

12. The software on the target is run from the target serial console. The reason for this is mostly to show that something is happening. The commands could also be run using the Simics agent, but then the run would be very quiet.

    This is done by this code:

```
    foreach $i in (range $repeats) {
        @print(f"\nRun {simenv.i+1}/{simenv.repeats}")

        # Test file, one-way parallel, level 2 verbose, and the pci node
        bp.console_string.wait-then-write $sercon " # " $cmd
```

13. The script picks up the start and end of a software run not by looking at the target serial console, but by using magic instructions. These magic instructions have been compiled into the target software, to signal points of interest. Where possible, this is a good way to only measure interesting parts of a workload. On real hardware, magic instructions are no-ops and thus they can be inserted into code that will run both on a simulator and on real hardware.

```
        # Pick up start of core run
        # Target software has been specially prepared with magic instructions
        bp.magic.wait-for number=3
```

14. The script picks up the current host time using Python, and the current virtual time by running a CLI command from Python:

```
        @start_rt = time.time()
        @start_vt = cli.global_cmds.ptime(_t=True)
```

8.4 Look at the application code hooks

15. Use an editor to open the file **[*simics*]/targets/workshop-02/target-source/m-app/m-app.c**.

16. Search for **MAGIC** in the code to see the hooks for the benchmarking system.

17. Note how **MAGIC(1)** and **MAGIC(2)** are used to bracket the computation and rendering of a single image:

```
void do_one_mandelbrot(work_desc_t *desc,
                       int          parallelism) {

  // Allow Simics to catch the start of each image
  MAGIC(1);
  …
```

18. Note how **MAGIC(3)** and **MAGIC(4)** are used to bracket the entire work of reading a work file:

```
int do_mandelbrot_from_file(FILE *workfile_fp, int parallelism) {
  …
  // Hook for measuring time
  MAGIC(3);

  …
  // Hook for measuring time
  MAGIC(4);
```

19. There is also a **MAGIC(0)** early in the **main()** function. **MAGIC(0)** is used by convention to be the generic "magic breakpoint". If **bp.magic.break** is used without any argument, it will break on magic zero.

8.5 Measure performance when using multiple compute units

After looking at the architecture of the benchmark system, it is time to test some more variants to see how much time they take.

20. Start the Simics simulator using the script **008-** with parallelism set to 8:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=8
```

This will result in a command line command on the target that is set up to tell the application program to run with all eight compute units in use.

21. Run the simulation:

```
simics> r
```

22. When the simulation stops, compare the run time in virtual time and real time to the previous experiment.

Note that the virtual time is much smaller, almost inversely proportional to the parallelism used. The lower virtual time also makes the real time much smaller since there is less virtual time to run through.

While the same amount of compute work is being done, it seems that the target software waiting for the computation to complete is a significant part of the overall simulation time cost.

23. Exit this simulation session.

```
simics> exit
```

24. Start the Simics simulator using the script **008-** with parallelism set to 4, to get a point in between 1 and 8:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=4
```

25. Run the simulation:

```
simics> r
```

26. When the simulation stops, compare the run time in virtual time and real time to the previous experiment.

27. Exit this simulation session.

```
simics> exit
```

8.6 Rebuild compute unit with threading

28. **Open** the source code for compute unit: **[*project*]/modules/m-compute/m_compute.dml**.

29. **Change** the value of the parameter **threaded_compute** to **true**:

```
param threaded_compute = true;
```

30. **Search** the rest of the code for the identifier **threaded_compute** to see what this does. There are quite a few hits, showing how to synchronize between the threaded job and the main simulation thread, as well as how the threaded work is started.

    Read the comments in the code to understand what is done and why; that is intended as the primary source of this information.

31. From the host shell, rebuild the device model and rerun all tests:

```
$ make test
```

    The threaded implementation is very careful to avoid changing the semantics visible to the rest of the system. Thus, the same test should work.

8.7 Measure performance with the threaded model

The threaded model has its biggest effect on runs with parallelism in the workload. To get an idea for the effectiveness of the threading, rerun the 8-way parallel test from above and compare the run time in real time.

32. Start the Simics simulator using the script **008-** with parallelism set to 8:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=8
```

33. Run the simulation:

```
simics> r
```

34. When the simulation stops, compare the run time in virtual time and real time to the previous experiment. The virtual time should be the very close to the previous runs with 8-way parallelism.

The real time execution is expected to be quite a bit smaller, but not eight times. The only part that is threaded is the compute work, and the simulation is doing many other things that are not affected by the parallel computation (Amdahl's law in action). The expectation is that the execution time in real time is reduced by a factor of two to three.

35. Exit this simulation session.

```
simics> exit
```

8.8 Maximize the effect of the threaded model

The effect of threading a workload is always dependent on the relationship between serial and parallel components, and how much work can be run concurrently. The "**bench1.txt**" test is maybe a bit light on the amount of work available for concurrent execution. Simulation performance is almost always dependent on the nature of the workload. Use the much heavier test case "**bench3.txt**" to see if this is case here. The amount of compute work required is roughly 20x bigger (4x the number of images, 5x the number iterations in each image).

36. Start the Simics simulator using the script **008-** with parallelism set to 8 and using the **bench3.txt** test file:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=8 test_file="bench3.txt"
```

37. Check that it is running with the threaded model before continuing! The output should look like this:

```
Accelerator parallelism   : 8
Time per pixel (ps)       : 10000
Model threading           : True
Stall on status poll (ps) : 0
```

38. Run the simulation:

```
simics> r
```

39. When the simulation stops, note that this run took a lot longer. Virtual time increases by approximately 4x, since there are 4 times as many images to render. The virtual rendering time per image is not affected by the increased maximum iteration count, which might be considered an unrealistic hardware model – adding that to the virtual time computation is a possibility.

40. Exit this simulation session.

```
simics> exit
```

41. **Turn of threading:** Open the source code for compute unit: **[*project*]/modules/m-compute/m_compute.dml**.

42. Change the value of the parameter **`threaded_compute`** to **`false`**:

```
param threaded_compute = false;
```

Why not make the threading controllable via a runtime attribute? That does make some sense for experiments like this – but also there is some additional code that is added to the model when compiled for threading that could theoretically disturb the execution. Thus, in to keep things really "clean", it makes sense to rebuild when making changes like this to the model.

In practice, an attribute that affects the nature of the computation would work fine for this code base. You can add that yourself, should not be very hard.

43. From the host shell, rebuild the device model and rerun all tests:

```
$ make test
```

44. Start the Simics simulator using the script **`008-`** with parallelism set to 8 and using the **`bench3.txt`** test file:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=8 test_file="bench3.txt"
```

45. Check that it is running with the model that is not threaded before continuing! The script should print this information:

```
Accelerator parallelism   : 8
Time per pixel (ps)       : 10000
Model threading           : False
Stall on status poll (ps) : 0
```

46. Run the simulation:

```
simics> r
```

47. This run can take a very significant amount of time. The expectation is that the virtual time is the same as for the previous run, but that the real time is four times or more longer. The threading of the device model has a larger effect on the real-time execution time when there is more work that can be done in parallel to the main simulation thread.

48. Exit this simulation session.

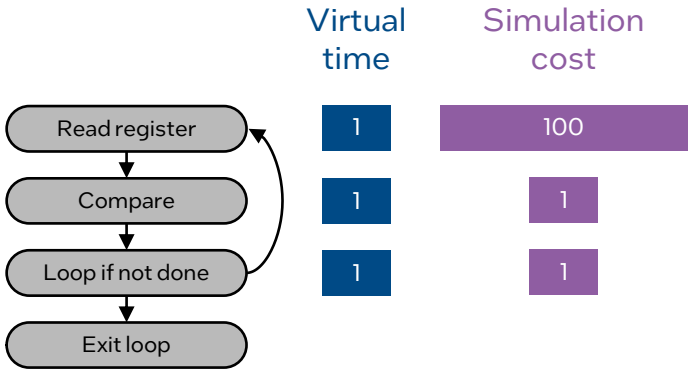
```
simics> exit
```

9 Improve Performance for Software Poll Loops

The **`m-app`** application uses polling to detect the end of a compute run. The code sits in a tight loop reading the status register of the accelerator. Such code is fairly common when driving hardware that is expected to return "soon" – for very long operations, interrupts are more common. However, calling the driver to wait for an interrupt is a little bit complicated and time consuming. Thus, the **`m-app`** program uses polling.

However, such polling can pose a performance issue for a virtual platform. The core of the issue is that doing a memory read to a device model to check the status of the flag is much more expensive than running regular instructions that do not touch devices. Thus, each iteration of the poll loop will only consume a small amount of virtual time but require a large amount of real time.

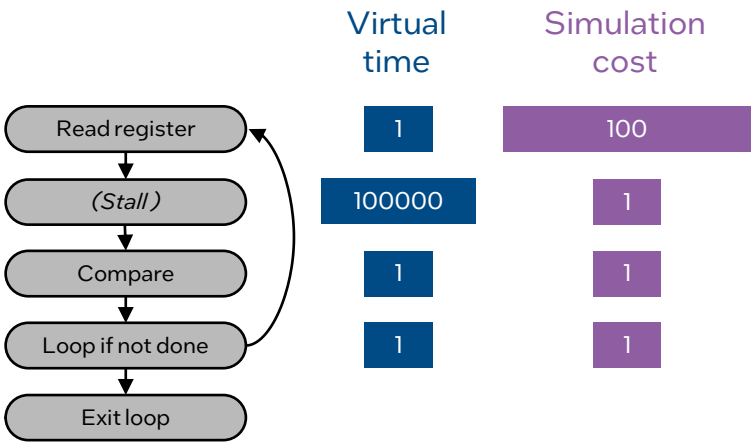
It looks something like this, with the simulation cost of "100" being mostly indicative.



In practice, the simulation time when using **`m-app`** is dominated by the polling due to this effect.

This is a simulation performance problem that can be alleviated in the simulator itself by making reads to the polled register take more virtual time. This will reduce the number of times the loop needs to iterate before detecting the change in the register, thus reducing the amount of simulation time spent running roughly the same virtual time. In the Simics simulator, this is implemented by using the **`SIM_stall()`** call to insert a wait in the

execution of the target code. Using stalling, the above scenario would look like the below when using a 100k virtual processor cycle stall time:



In this section, you will analyze the performance of the simulation to find the register that is being polled using standard performance and target system analysis tools.

9.1 Analyze the simulation performance

1. Start the Simics simulator using the script **`008-`**, using default settings:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
```

2. **Run** the simulation until the **`m-app`** program has started on the target system. This is achieved by looking for magic instruction number 3, which is used to mark the beginning of the actual work of the program (as discussed above in section 8.4):

```
simics> bp.magic.run-until number = 3
```

3. **Wait** until the run stops.

4. Start the **`system-perfmeter`** tool using **`normal`** mode, additionally counting device accesses (using the flag **`-io`**). This provides sufficient information for this analysis, while not producing an overwhelming amount of output:

```
simics> system-perfmeter mode = normal -io -window
```

5. Run the simulation for half a virtual second:

```
simics> r 500 ms
```

   As the simulation runs, the **`system-perfmeter`** tool will print one line every *real-time* second. In particular, note the columns called **`Slowdown`** and **`i I/O`** (meaning instructions per device access).

6. When the simulation stops, a performance summary will be printed by the **`system-perfmeter`** tool. The important lines for this exercise are **`Slowdown`** and **`Steps per I/O`**. They will look something like this (numbers will vary depending on the speed of the host and small differences in the target system state):

```
SystemPerf: Performance summary:
------------------------------
SystemPerf: Target:  4 CPUs in 1 cells [4]
…
SystemPerf: Slowdown:                          134
…
SystemPerf: Steps per I/O:                     15.06
…
```

A slowdown of 134 is not very good, even though the system is doing a lot computation on the host. The steps per IO number indicates that the processor runs about 15 instructions for each device access, on average. For best performance, device accesses should not happen much more often than one in 10k instructions or more. A low number typically indicates that software is running a polling loop somewhere.

7. To figure out which device is being polled, use the **`io-stats`** command:

```
simics> io-stats
```

It indicates that basically all devices accesses are hitting the control device. The output would look something like the below. Note that it counts steps vs device accesses slightly differently from the system perfmeter.

```
simics> io-stats
Total io-accesses   :        59396436
Total steps         :     20500958891 (in average an io access each 345)
Total non-idle steps:     20500958891 (in average an io access each 345)

Most frequently accessed device classes (Total):

Accesses  Class           Percent
59391612  m_control.ctrl   100.0%

Most frequently accessed device objects (Total):

Accesses  Object                      Class           Percent
59391612  qsp.macc.control.bank.ctrl  m_control.ctrl   100.0%
```

8. The command only shows the control unit in this case, since almost all accesses go there. To see more devices, use a 0% cutoff to the command:

```
simics> io-stats cutoff = 0
```

This will show a few more devices, which just makes it even clearer that accesses to the control unit totally dominate the device access count.

9. To find the precise register, you need to use the bank coverage instrumentation tool. To do this, start a new simulation from the same script to get back to the same initial situation.

```
simics> exit
```

9.2 Pin-point polling register

10. Start the Simics simulator using the script **`008-`** with the same settings as above:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
```

11. Run the simulation until the application has started, just like above:

```
simics> bp.magic.run-until number = 3
```

12. Create a new bank coverage tool, targeting only the register bank indicated by **io-stats** in previous experiment:

```
simics> new-bank-coverage-tool banks = qsp.macc.control.bank.ctrl
```

The bank coverage tool collects access count information for one or more banks. It is used for both ascertaining the register use coverage from software, and to investigate how often particular registers are accessed from software.

13. Run the simulation for half a virtual second, just like above:

```
simics> r 500 ms
```

14. When the simulation stops, list the register access counts for the **ctrl** bank in the control unit:

```
simics> coverage_tool0.access-count bank = qsp.macc.control.bank.ctrl
```

The result is should indicate the register that is being accessed all the time:

Row #	Name	Offset	Size	Count
1	start	0x0008	8	52
2	status	0x0010	8	59_391_559
Sum				59_391_611

15. Exit the simulation session.

```
simics> exit
```

9.3 Optimize polling performance

You should have the code for the control unit in your Simics project already. If not, go back to the point where you build it the first time.

16. Open the source code: **[*project*]/modules/m-control/m_control.dml**.
17. Search it for the identifier **stall_on_status_read**.

    This will find a few snippets of implementation that make the device stall any processor that reads from the **ctrl.status** register.

    The attribute **status_reg_stall_time** contains the time to stall after each hardware access, allowing it to be configured at runtime.

18. Change the value of the parameter **stall_on_status_read** to true:

```
// Enable use of stall performance optimization
param stall_on_status_read = true;
```

19. From the host shell, rebuild the device model:

```
$ make m-control
```

20. Rerun the unit tests, to make sure this did not change the behavior of the model in a visible way:

```
$ make test
```

9.4 Test and measure the effect of stalling

Measure the performance after the change. Note that the Simics simulator framework can open a checkpoint taken using a different implementation of the same system as long as the changes are backwards compatible – there is no need to rebuild the checkpoint. The newly compiled version of **`m-control`** will be used along with the state from the checkpoint (this works since no attributes were removed, only a single one added).

21. Start the Simics simulator like above:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
```

Note that the startup printouts indicate a non-zero value for **`Stall on status poll`**, where it previously was always zero:

```
Accelerator parallelism   : 1
Time per pixel (ps)       : 10000
Model threading           : False
Stall on status poll (ps) : 50000000
```

22. Run the simulation until the application has started, just like above:

```
simics> bp.magic.run-until number = 3
```

23. Start the **`system-perfmeter`** tool:

```
simics> system-perfmeter mode = normal -io -window
```

24. Run the simulation for half a virtual second, just like above:

```
simics> r 500 ms
```

25. Wait for the simulation to stop and check the results. Note how **`system-perfmeter`** indicates a significantly lower slow-down, and much higher steps per I/O. Something like this:

```
…
SystemPerf: Slowdown:                          21.81
…
SystemPerf: Steps per I/O:                    458.77
…
```

26. Exit the simulation session.

```
simics> exit
```

9.5 Ascertain the effect on benchmarking runs

The above tests checked the effect using various performance tools. It is also necessary to ascertain the effect of the optimization on an actual complete run without instrumentation attached.

27. Start the Simics simulator using the script **008-** with default settings:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
```

    The script should indicate a non-zero stall time on status reads.

28. Run the simulation:

```
simics> r
```

29. When the simulation stops, compare the run time in virtual time and real time to what you saw previously in the baseline run. You should expect a much lower real time, but essentially the same virtual time.

30. Exit the simulation session.

```
simics> exit
```

31. Start a new session using script **008-** with parallelism set to 8:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=8
```

32. Run the simulation:

```
simics> r
```

33. When the simulation stops, compare the run time in virtual time and real time to the previous experiments.

    The benefit of the stalling optimization is typically much less for the case of parallel computations, since the polling loop already runs for a shorter time on target (thanks to the job being split up into multiple jobs with a total virtual time that is shorter).

34. Exit the simulation session.

```
simics> exit
```

The conclusion of this experiment would seem to be that "stall on poll" optimizations can have a very positive impact on simulation performance with little effect on the target software behavior. At least for this particular accelerator and software stack.

9.6 What is the effect of an extremely large stall time?

However… could stalling have an impact on the software behavior? Test this as well, using an extreme value for the stall time.

35. Start the Simics simulator using the script **008-** with default settings:

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
```

    The script should indicate a non-zero stall time on status reads.

36. Change the stall time by writing the configuration register that appears when stalling is enabled. The value is a floating-point value in seconds, set it to 5 millisecond, or 100 times higher than the default.

```
simics> qsp.macc.control->status_reg_stall_time = 5e-3
```

37. Check that the setting had effect using the **info** command on the control unit:

```
simics> qsp.macc.control.info
```

38. Run the simulation:

```
simics> r
```

39. When the simulation stops, compare the run time in virtual time and real time to what you saw above. The virtual time is expected to increase by about 50%, from around 0.98 to around 1.5. The runtime

40. Exit the simulation session.

```
simics> exit
```

9.7 Full fury movie

End the performance optimization exercises with a full fury run combining all the optimizations to provide a smooth video rendering for the benchmarks.

41. Open the source code for compute unit: **[*project*]/modules/m-compute/m_compute.dml**.

42. Change the value of the parameter **threaded_compute** to **true**, in order to enable threaded computation:

```
param threaded_compute = true;
```

43. From the host shell, rebuild all device models:

```
$ make
```

44. Start a new session using script **008-** with parallelism set to 8 and using the most expensive benchmark, **bench3.txt**. Only repeat the run once.

```
$ ./simics targets/workshop-02/008-system-benchmarking.simics
parallelism=8 test_file=bench3.txt repeats=1
```

45. Show the graphics console for the accelerator:

```
simics> qsp.macc.gcon.con.show
```

46. Run the simulation:

```
simics> r
```

47. Once the run has finished, exit the simulation session.

```
simics> exit
```

10 Create a Custom Command for the Display Unit

This exercise adds a custom command-line command for the display unit. In general, when devices have user-facing features controller with attributes (or interfaces), it is a good idea to create custom CLI commands to invoke them. This makes discovering the features easier. It also makes accessing the features from CLI and Python easier.

For this command, the

10.1 Build the display unit in the project

1. Use **`project-setup`** to copy the **`m-display`** module to your project:

```
$ bin/project-setup --copy-module m-display
```

2. Build the module:

```
$ make
```

10.2 Run the display unit on its own

3. Start Simics using the script **`010-`**:

```
$ ./simics targets/workshop-02/010-try-m-display.simics
```

4. Check the configuration:

```
simics> list-objects
```

Note the display unit object that is called **`dd`**.

5. Raise the log level:

```
simics> log-level dd 3
```

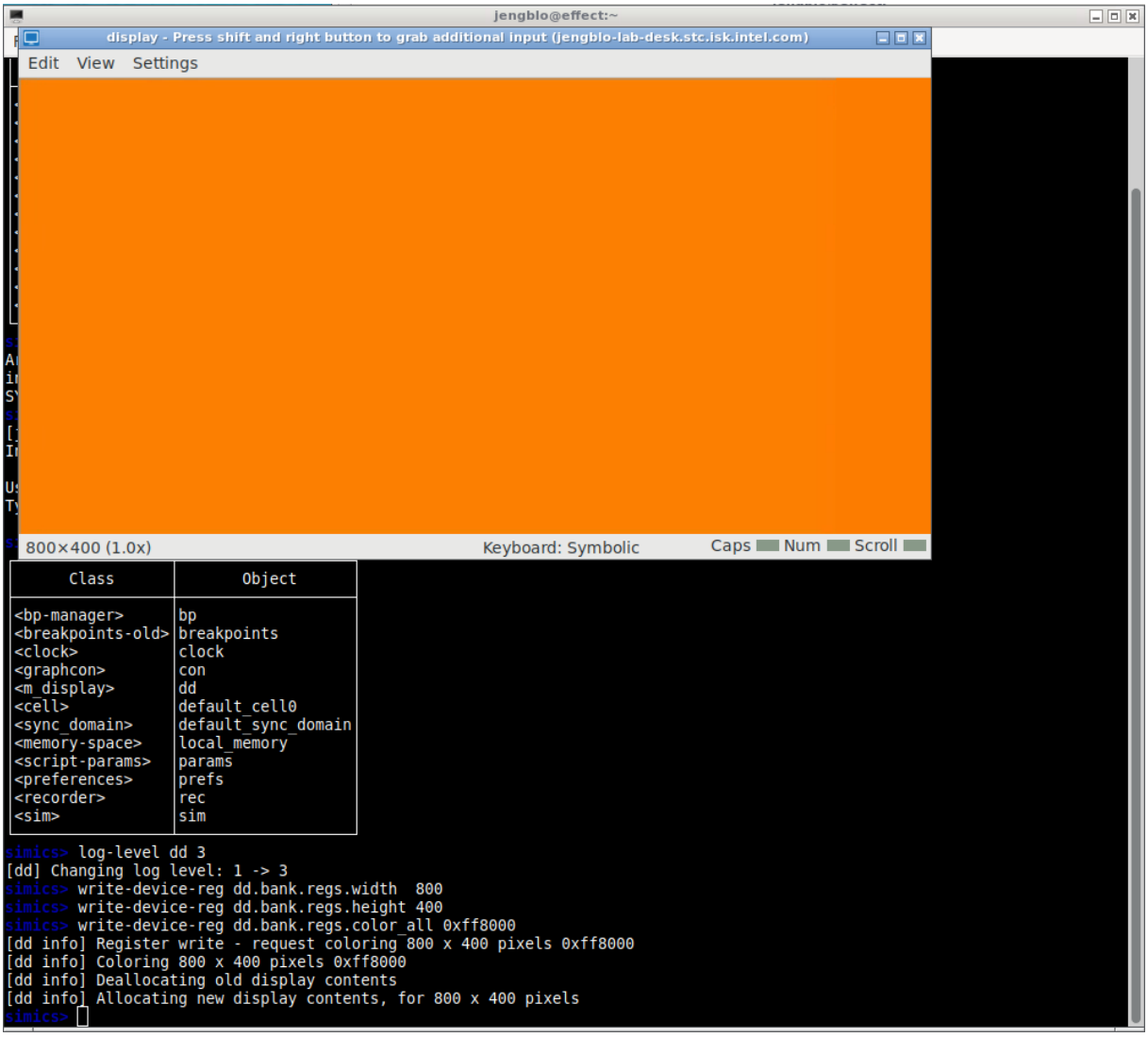
6. Set up the width and height registers in the display unit:

```
simics> write-device-reg dd.bank.regs.width  800
simics> write-device-reg dd.bank.regs.height 400
```

7. Write the "color everything" register, which is designed to test the connection from the display unit to the console:

```
simics> write-device-reg dd.bank.regs.color_all 0xff8000
```

The result should be a nice orange window:



10.3 Create a custom command outside the device code

Custom commands can be created and overridden on the fly in the Simics simulator. This feature can be used to prototype commands within a single Simics simulator session, with no need to rebuild or reload anything.

8. Create a new Python file in your project. Say **`[project]/u10-custom-command.py`**.

9. Open the file and paste in the following initial code:

```
def display_color_all_cmd(object_arg, width, height, color):
    print(f"Command called for {object_arg}, {width}x{height} to 0x{color:08x}")

cli.new_command("color-all",
                display_color_all_cmd,
                args = [cli.arg(cli.obj_t("Display object",
                                          kind="m_display"), "display"),
                        cli.arg(cli.int_t, "width"),
                        cli.arg(cli.int_t, "height"),
                        cli.arg(cli.int_t, "color")],
                short = "Display a solid color",
                doc="""
Set the <arg>display</arg> to size <arg>width</arg> x <arg>height</arg>,
and set the color to <arg>color</arg>.
"""
               )
```

10. Go back to the Simics simulator, and run the file:

```
simics> run-python-file u10-custom-command.py
```

11. Check that the new command was added by doing **`help`** on it:

```
simics> help color-all
```

12. Find the command with **`list-commands`**:

```
simics> list-commands substr = color
```

13. Try to run it:

```
simics> color-all
```

An error will be printed that mandatory arguments are missing.

14. Try again:

```
simics> color-all dd 100 100 0xff00ff
```

This should work and print the message from the command function:

```
simics> color-all dd 100 100 0xff00ff
Command called for <the m_display 'dd'>, 100x100 to 0x00ff00ff
```

This means the command is properly receiving arguments.

15. Next, it is necessary to try out the code to write a register from Python. The simplest way to do this is to use the CLI commands you used above, but from Python. Global CLI commands such as **`write-device-reg`** are available in the **`cli.global_cmds`** namespace in Python.

    Instead of coding this into the Python, try it live. Convert the dashes in the name to underscores, and pass in the command arguments as Python function arguments:

```
simics> @cli.global_cmds.write_device_reg( "dd.regs.width",100)
```

That does not work.

The error says that all arguments must be named.

16. To find the precise name of the arguments, use Python **help()**:

```
simics> @help(cli.global_cmds.write_device_reg)
```

17. With this information, try again:

```
simics> @cli.global_cmds.write_device_reg( register="dd.regs.width",
data=100)
```

18. To use this inside the custom command, it is necessary to convert from the object parameter given to the command to the name string of object. Try that from the command line too. Type in **@conf.dd** to get to the **dd** object from Python, and then tab-complete to check its members.

```
simics> @conf.dd.<TAB>
```

19. There is a "name" member that seems appropriate. Try it:

```
simics> @conf.dd.name
```

    Looks like it returns the expected value.

20. Next, try construct the call to the CLI command in Python, with the object in a variable. Using a Python f-string, it is easy to construct the string value:

```
simics> @o=conf.dd
simics>
@cli.global_cmds.write_device_reg( register=f"{o.name}.regs.width",
data=400)
```

21. Check that the register did change its value:

```
simics> print-device-reg-info dd.regs.width
```

22. With this, the contents of the Python file can be updated. Add three lines of **cli.global_cmds…** to the Python function.

    One solution is found below in *Solution: Color-all command: implementation for the command line*.

23. Reload the Python file:

```
simics> run-python-file u10-custom-command.py
```

24. Test the updated command:

```
simics> color-all dd 600 100 0xff00ff
```

    Which should provide an all-purple display.

25. Exit this simulation session.

```
simics> exit
```

10.4 Add the command to the device class

The command defined above is global, which is not really recommended for a command that only pertains to a single class and at most a few objects at once. Instead, it should be

a namespace command on objects of the class. It should be part of the device model code.

26. Open the file **[*project*]/modules/m-display/module_load.py**. This file is run when the module is loaded, and creates custom commands for the device classes in the module. In this case, the **info** and **status** commands for the class.

    To make the command a namespace command, the **new_command()** function takes an additional "**cls**" argument that specifies the class name. This also means that the "**display**" argument to the command is now implicit and should be removed.

27. Add the code from **u10-custom-command.py** at the end of the **module_load.py** file, modifying it to work with the class. The class name is held in the variable **class_name**.

    One solution is found below in *Solution: Color-all command: implementation for module*_load.py.

28. From the host shell, rebuild the device model:

```
$ make
```

29. Start a new simulation session from script **010-**:

```
$ ./simics targets/workshop-02/010-try-m-display.simics
```

30. Raise the log level:

```
simics> log-level dd 3
```

31. Try the new command:

```
simics> dd.color-all 500 400 0x8080ff
```

32. Check the help on the command:

```
simics> help dd.color-all
```

    Note that the command is documented as provided by the **m_display** class.

10.5 Some bad-case testing

Having a custom command in place makes it easier to make stupid mistakes and enter intentionally bad data.

33. Try setting one dimension to zero:

```
simics> dd.color-all 100 0 0x8080ff
```

    The existing code will block that with a nice error message.

34. However… try a negative number:

```
simics> dd.color-all -1 100 0x8080ff
```

This will cause Simics to assert, as it tries to allocate a very very large block of memory. The number -1 is treated as a 64-bit unsigned integer number, which becomes very large, and there is no check for "unreasonably large" sizes in the device model itself. Instead, the Simics core memory allocators sees an unreasonable request and asserts – since clearly the code requesting the memory is broken.

35. Exit this simulation session – the command line still lives on, so you have to quit Simics even after the fatal error.

```
simics> exit
```

10.6 Add additional checking

36. Open the file **[*project*]/modules/m-display/m_display.dml**.

37. Find the behavior of the **color_all** register, by searching for "**color_all**". This will lead to a function that implements the behavior of writing to the register. Add a check for the display width or height being bigger than 4000 pixels.

One solution is found below in *Solution: Color-all command: implementation of checks in the DML device*.

38. From the host shell, rebuild the device model:

```
$ make
```

39. Start a new simulation session from script **010-**:

```
$ ./simics targets/workshop-02/010-try-m-display.simics
```

40. Raise the log level:

```
simics> log-level dd 3
```

41. Try the command again:

```
simics> dd.color-all -1 100 0x0080ff
simics> dd.color-all 4000 100 0x0080ff
simics> dd.color-all 4001 100 0x0080ff
```

42. Next, try **-1** for the second argument. Note that the Simics CLI parsing makes it necessary to use a parenthesis around "**-1**" when used in here: otherwise, the CLI will interpret the input as "**200-1**" being used as the value for the first argument:

```
simics> dd.color-all 200 (-1) 0x0080ff
```

43. Exit this simulation session.

```
simics> exit
```

10.7 Solution: Color-all command: implementation for the command line

The final contents of **`u10-custom-command.py`**:

```
def display_color_all_cmd(object_arg, width, height, color):
    print(f"Command called for {object_arg}, {width}x{height} to 0x{color:08x}")
    cli.global_cmds.write_device_reg(register=f"{object_arg.name}.bank.regs.width", data=width)
    cli.global_cmds.write_device_reg(register=f"{object_arg.name}.bank.regs.height", data=height)
    cli.global_cmds.write_device_reg(register=f"{object_arg.name}.bank.regs.color_all", data=color)

cli.new_command("color-all",
                display_color_all_cmd,
                args = [cli.arg(cli.obj_t("Display object",
                                          kind="m_display"), "display"),
                        cli.arg(cli.int_t, "width"),
                        cli.arg(cli.int_t, "height"),
                        cli.arg(cli.int_t, "color")],
                short = "Display a solid color",
                doc="""
Set the <arg>display</arg> to size <arg>width</arg> x <arg>height</arg>,
and set the color to <arg>color</arg>.
"""
               )
```

10.8 Solution: Color-all command: implementation for module_load.py

```
#
# ------------------------ color-all -----------------------
#
def display_color_all_cmd(object_arg, width, height, color):
    cli.global_cmds.write_device_reg(
        register=f"{object_arg.name}.bank.regs.width", data=width)
    cli.global_cmds.write_device_reg(
        register=f"{object_arg.name}.bank.regs.height", data=height)
    cli.global_cmds.write_device_reg(
        register=f"{object_arg.name}.bank.regs.color_all", data=color)

cli.new_command("color-all",
                display_color_all_cmd,
                args = [cli.arg(cli.int_t, "width"),
                        cli.arg(cli.int_t, "height"),
                        cli.arg(cli.int_t, "color")],
                cls = class_name,
                short = "Display a solid color",
                doc="""
Set the display to size <arg>width</arg> x <arg>height</arg>,
and set the color to <arg>color</arg>.
"""
               )
```

10.9 Solution: Color-all command: implementation of checks in the DML device

Despite what the comments in the code says, the simplest solution is to check this right in the **`color_it_all()`** function. After the check for zero width or height, add a similar check for >4000 pixels (or some other reasonable number). This will nicely cover the color-all case. The same check should likely be added to the regular display path. Note that adding it to the **`console.set_display_size()`** function becomes more complicated as that function deals with reallocating the internal buffer, and it would have to return an error that would have to be handled… etc. However, adding an assert in that function could be a way to catch an error like this early.

```
    if ( (width==0) || (height==0)) {
        log spec_viol, 1, software: "Display size of %d x %d pixels is non-sensical",
                                    width, height;
        return;
    }

    // Check too-large error
    if ( (width>4000) || (height>4000)) {
        log spec_viol, 1, software: "Display size of %d x %d pixels is too large",
                                    width, height;
        return;
    }

    // Call the general coloring method
    single_color_display(width, height, rgb);
```