Intel® Simics® Simulator New User Training

Lab 003 – Target system setup and inspecting the configuration

Lab 003 – Target system setup and inspecting the configuration

This lab covers Intel® Simics® simulator target setup files, how to create a custom setup script, and how to inspect the structure and state of the system model.

A. Investigate target setup scripts

First of all, you will look at the setup scripts that are used to launch a new simulation session. A simulated system is set up by a script that calls other scripts that call other scripts... and so on. In this first section, you will inspect some of that nested script structure.

1. Start a new session using the target **`simics-user-training/001-qsp-training`** (just like you did in Lab 001).

   ```
   [$|C:>] simics[.bat] simics-user-training/001-qsp-training
   ```

   Note that from this point on, command-line launches of the Intel Simics Simulator will be written in the above host-independent style. Paths will be Linux-style, as that works correctly on both types of hosts.

2. Identify the YML file that defines the target you just started.

   ```
   simics> list-targets -verbose substr = user-training
   ```

   Which should show something like this:

   ```
   ┌─────────────────┬───────────────┬───────────────┬───────────────┐
   │     Target      │    Package    │    Script     │  Description  │
   ├─────────────────┼───────────────┼───────────────┼───────────────┤
   │simics-user-     │Simics Training│/path/to/simics│Script that    │
   │training/001-qsp-│               │-training-6.0.p│brings up the  │
   │training         │               │reXXX/targets/s│New User       │
   │                 │               │imics-user-trai│Training setup.│
   │                 │               │ning/001-qsp-tr│               │
   │                 │               │aining.target.y│               │
   │                 │               │ml             │               │
   ...
   ```

3. Open the ***`/path/to/`*****`simics-training-6.0.pre`*****`XXX`*****`/targets/simics-user-training/001-qsp-tr aining.target.yml`** script with a text editor. Take the exact path from the output you got in the previous step. Files ending with **`.target.yml`** are considered to define *targets* that can be used to start new simulator sessions from the host shell or using the **`load-target`** command from the command line.

The file looks like this. It is a YAML-format file using keywords defined by the simulator parameter system.

```
%YAML 1.2
---
description: Script that brings up the New User Training setup.
params:
  platform:
    import: "%simics%/targets/qsp-x86/clear-linux.target.yml"
    defaults:
      machine:
        system_info: QSP x86 with Linux - Simics Training Setup
        hardware:
          name: machine
          processor:
            num_cores: 2
        software:
          linux:
            auto_login_root: true
  training_card:
    import: "%simics%/targets/simics-user-training/training-card.yml"
    defaults:
      owner: ^platform:output:system
      pcie_slot: .mb.nb.pcie_slot[1]
[…]
script: "%script%/001-qsp-training.target.yml.include"
...
```

The target file imports other target files that it depends on, provides specific values for parameters, and defines its own parameters. At the end, the **`script`** declaration points onwards to an actual script file that will create the target. The script file can be written using command-line scripting or in Python.

4. **Open** the file (**`001-qsp-training.target.yml.include`**) whose name you found at the end of the YML file. Note that the path to the file in the YAML file starts with the "**`%script%`**" marker, which means that the file is located in the same directory as the **`.target.yml`** file. The content of the file looks like this:

```
## Invoke the base Quick-Start Platform setup
##   Note that this path is the same as that listed in the .target.yml file
run-script "%simics%/targets/qsp-x86/clear-linux.target.yml" namespace = platform

## Same for the training-card setup script
run-script "%simics%/targets/simics-user-training/training-card.yml" namespace = training_card
```

As you can see, the script first runs the YML scripts from which it imported parameters and then does some additional setup steps.

Note the use of the top-level **`params`** object – this object is always present in a simulation configuration that uses targets and holds all the parameters seen so far.

5. Optional: If your editor supports it, select syntax highlighting for Bash or Sh shell. This does a pretty decent job for Intel Simics Simulator scripts.

6. To see where the simulator finds the **`training-card.yml`** script, run this command:

```
simics> lookup-file "%simics%/targets/simics-user-training/training-card.yml"
```

   The precise path depends on where you Intel Simics simulator installation is located.

```
"/this/path/depends/on/your/installation/simics-training-NNN/targets/simics-user-training/training-card.yml"
```

7. The **`training-card.yml`** script is used in the training setup to add the PCIe training card to the system. Open the file in the editor. Again, you can see the parameters that it defines and the name of the "**`.yml.include`**" file that will perform the actual actions to set things up:

```
%YAML 1.2
---
  name:
    type: str
    description: Name of the subsystem to create.
    default: training_card
  owner:
    type: str
    description: The name of the system inside of which we create the card.
    required: false
  pcie_slot:
    type: str
    description: The PCIe slot to which to connect the card.
    required: false
  use_behavioral_box:
    type: bool
    description: >-
                 Use the behavioral model of the training device.
                 At the time, there is no firmware box model available.
    default: true
    advanced: 2
[…]
  output:
    training_card:
      type: str
      output: true

script: "%script%/training-card.yml.include"
...
```

8. Open the **`training-card.yml.include`** file. By now you should know where to find it, based on the "**`%script%`**" marker in the **`.yml`** file. Look at how it creates and attaches the PCIe card to the system:

```
##
## Create behavioral or firmware version of device
## (right now, there is no firmware version available)
##
if (params.get use_behavioral_box) {
    load-module led-system-comp

    # Create an uninstantiated version of the component
    $training_card = (create-led-system-comp
      $name
      clock_freq            = (params.get clock_frequency)
      pixel_update_time     = (params.get bb_pixel_update_time)
      toggle_check_interval = (params.get bb_toggle_check_interval))

    # Instantiate the objects created by the create- command
    instantiate-components
    $training_card.panel.con.show
    $training_card.panel.con.dimming value = FALSE
    params.setdefault output:training_card $training_card
} else {
    interrupt-script "No firmware box model is available (yet)"
}
```

This is what Intel Simics simulator system setup scripts do: they load modules, create components, connect them to other components, and eventually instantiate the components to create the actual simulation configuration.

9. Go back to **`001-qsp-training.target.yml`** and see how it refers to **`yocto-linux.target.yml`**. Locate the actual path of the file as we did before and open it in an editor. Note that this file is in the Quick-Start Platform target package and not in the Training package.

10. In **`yocto-linux.target.yml`** you will see a reference to **`%script%/yocto-linux-setup.yml`**. Open that file as well. This script is an example of a setup script that creates a hardware system and loads a software stack. You can see that the hardware is defined in the aptly named file **`hardware.yml`** in the same directory.

11. Open **`hardware.yml`**. It creates the standard Quick-Start Platform hardware (to which the **`001-qsp-training`** adds the training card):

```
%YAML 1.2
---
description: >-
  Sets up the hardware for a QSP-x86 virtual machine, exposing many
  configuration options as Simics parameters. This include file is used by
  other scripts to setup a complete virtual machine.
  Adds two disks to the machine. If disk image and disk size are NIL, no disk
  will be added. Also adds a CD-ROM to the machine.

params:
  name:
    type: str
    description: Name of the machine. Used for the top level component.
[…]
```

12. Look at the parameters in the "**processor**" namespace:

```
  processor:
    description: Processor settings
    class:
      type: str
      default: x86QSP1
      description: >-
        It should be specified as processor component class without
        the "processor_" prefix and with underscores replaced with hyphens,
        e.g. x86-intel64. Tip: to obtain the list of processor component classes
        please use the "list-classes substr = processor_" Simics command.
    num_cores:
      type: int
      default: 1
      description: >-
        Number of cores per physical processor or slot in the machine.
        Allowed values are 1 - 128.
    freq_mhz:
      type: int
      default: 2000
      description: Processor frequency in MHz.
[...]
```

Getting help on parameters and targets

The above was a peek behind the scenes. Normally, you do not dig into the files in the installation to understand the parameters of a system. Instead, you use the help functionality of the parameter system. Target parameters and target scripts are self-documenting.

13. To find all available targets, use the **`list-targets`** command without any arguments:

```
simics> list-targets
```

This shows targets from the training package and the Intel Simics Quick-Start Platform and any other targets from packages you have installed.

14. To see the parameters of the **`qsp-x86/yocto-linux`** target, use the **`params.help`** command:

```
simics> params.help target = "qsp-x86/yocto-linux"
```

This results in a fairly long list of parameters. The list can be narrowed down using various arguments to the command.

15. Check the definition of the parameters in the processor namespace of the **`qsp-x86/yocto-linux`** target. They reflect the YAML description that you looked at above, in the **`hardware.yml`** file:

```
simics> params.help target = "qsp-x86/yocto-linux" namespace =
machine:hardware:processor
```

The output will show a selection of the parameters:

```
┌────────────────────────────────────┬────┬─────────────────────────────────────────┬───────┐
│Name                                │Type│Description                              │Default│
├────────────────────────────────────┼────┼─────────────────────────────────────────┼───────┤
│machine:hardware:processor:class    │str │It should be specified as processor      │x86QSP1│
│                                    │    │component class without                  │       │
│                                    │    │the "processor_" prefix and with         │       │
│                                    │    │underscores replaced with hyphens,       │       │
│                                    │    │e.g. x86-intel64. Tip: to obtain the list│       │
│                                    │    │of processor component classes           │       │
│                                    │    │please use the "list-classes substr =    │       │
│                                    │    │processor_" Simics command.              │       │
├────────────────────────────────────┼────┼─────────────────────────────────────────┼───────┤
│machine:hardware:processor:freq_mhz │int │Processor frequency in MHz.              │2_000  │
├────────────────────────────────────┼────┼─────────────────────────────────────────┼───────┤
│machine:hardware:processor:num_cores│int │Number of cores per physical processor or│1      │
│                                    │    │slot in the machine.                     │       │
│                                    │    │Allowed values are 1 - 128.              │       │
└────────────────────────────────────┴────┴─────────────────────────────────────────┴───────┘
```

16. However, not all the parameters declared in the YAML file are seen. The reason is that some parameters have an "advanced" level set to more than one and thus are not shown by default. Retry the above, but also showing advanced parameters up to level 2:

```
simics> params.help target = "qsp-x86/yocto-linux" namespace =
machine:hardware:processor advanced = 2
```

A few more show up.

17. Next, try advanced level 3:

```
simics> params.help target = "qsp-x86/yocto-linux" namespace =
machine:hardware:processor advanced = 3
```

This adds a couple more parameters.

18. Next, try advanced level 4:

```
simics> params.help target = "qsp-x86/yocto-linux" namespace =
machine:hardware:processor advanced = 4
```

This should not result in any new parameters being shown. If it does, the target scripts changed since this lab was last updated.

19. Another way to search is to use the **`substr`** argument. For example, to locate parameters whose name contains "memory":

```
simics> params.help target = "qsp-x86/yocto-linux" substr = memory
```

This will find the **`machine:hardware:memory_megs`** parameter you used in Lab 001.

20. **Quit** this simulation session.

```
simics> quit
```

Loading targets from the simulator command line

Simulator sessions can be started from the simulator command line as well. Thanks to the CLI, the help and tab-completion abilities are better than when loading a target directly from the outside shell.

21. **Start** a new empty simulator session:

    ```
    [$|C:>] simics[.bat]
    ```

22. **Load** the standard training setup using tab completion on the name:

    ```
    simics> load-target simics-u <TAB>
    ```

    This will expand to:

    ```
    simics> load-target "simics-user-training/00
    ```

23. Complete the name using tab-completion. Type "**1**" and press tab again.

    ```
    simics> load-target "simics-user-training/001<TAB>
    ```

    Which should result in a full target name:

    ```
    simics> load-target "simics-user-training/001-qsp-training"
    ```

24. Press **Enter** to load the target.

    The simulator will print the name of the namespace where the target parameters for this target load are stored. This is not the same as the name of the top-level object created for the target.

Inspecting the parameters of the current session

The target parameter system keeps track of the parameters used to set up the current session, allowing inspection of the parameters after the setup process has been completed.

25. To see all the parameters defined so far in the simulation session, use the **`params.list`** command:

    ```
    simics> params.list
    ```

    You will see that all the parameters are put into a namespace "**`001_qsp_training`**". This namespace is generated by default by the **`load-target`** command. Each target being loaded requires its own namespace to avoid parameter name collisions.

    The **`params.list`** command uses the advanced specifications to filter output.

26. To see the parameter hierarchy as a tree, use the **`-tree`** option:

    ```
    simics> params.list -tree
    ```

27. Look for the value of the **`memory_megs`** parameter by searching for something that includes the string "**`memory`**":

```
simics> params.list substr = memory
```

This should show the parameter along with its default value:

```
┌──────────────────────────────────────────────────┬────┬─────┐
│Name                                              │Type│Value│
├──────────────────────────────────────────────────┼────┼─────┤
│001_qsp_training:platform:machine:hardware:memory_megs│int │8_192│
└──────────────────────────────────────────────────┴────┴─────┘
```

28. The parameter system can show you more information about the parameter, including which file it is defined in and whether the current value is considered a default or something configured by the user (check the "**`state`**" field):

```
simics> params.describe
001_qsp_training:platform:machine:hardware:memory_megs
```

29. **Quit** this simulation session.

```
simics> quit
```

B. Creating and using presets

Setting parameters on the command line does not scale well when customizing many settings and sharing such a set of customized settings with others. The command lines can get very long. Thus, best practice is to put custom sets of parameter values into so-called **preset files** and share those. A preset is a YML-format file that provides values for some target parameters.

Launch a target with some custom arguments

You can write preset files from scratch, but often you have a running setup from which you want to create a preset file that configures the parameters of the platform in the same way that you currently have.

1. Start the training target from shell, with some modified parameters. You can use tab completion to build up the command line:

```
[$|C:>] simics[.bat] simics-user-training/001-qsp-training
platform:machine:hardware:memory_megs=16Ki
platform:machine:hardware:processor:num_cores=8
platform:machine:system_info="My custom training setup"
```

2. This should set up the standard training system, with 16GiB of memory and two processor cores. Check the current setup:

```
simics> print-target-info
```

Which should show a setup with two cores and 16 gibibytes of memory:

```
simics> print-target-info
My custom training setup

    ┌──────────┬──────────────────────────┐
    │Namespace │machine                   │
    │System    │a QSP x86 chassis         │
    │Processors│8 QSP X86-64, 2000.0 MHz  │
    │Memory    │16 GiB                    │
    │Ethernet  │1 of 1 connected          │
    │Storage   │2 disks (300 GiB)         │
    └──────────┴──────────────────────────┘
```

3. List the parameters that you changed. The parameter system tracks this, as mentioned above:

```
simics> params.list -only-changed
```

Which should show:

```
┌─────────────────────────────────────────────────────────┬────┬────────────────────────┐
│Name                                                     │Type│Value                   │
├─────────────────────────────────────────────────────────┼────┼────────────────────────┤
│001_qsp_training:platform:machine:hardware:memory_megs   │int │16_384                  │
│001_qsp_training:platform:machine:hardware:processor:num_cores│int │8              │
│001_qsp_training:platform:machine:system_info            │str │My custom training setup│
└─────────────────────────────────────────────────────────┴────┴────────────────────────┘
```

Creating a preset file

4. Save the changed parameters to a preset file using the **save** command of the **params** object. This command takes a namespace argument, indicating the node in the parameter tree whose content is to be saved. Typically, this would be the parameter namespace of the platform you just loaded, as generated by the target parameter system (**001_qsp_training** in this specific case). By convention, preset files have the extension **.preset.yml**.

```
simics> params.save file = my-custom-training-preset.preset.yml
namespace = 001_qsp_training
```

5. Check the contents of the generated file. Either open the file in an editor or invoke a shell command to print it to the simulator command line. The **!** operator tells the simulator to run the rest of the line as a command in the host shell.

   In a Windows environment:

```
simics> !type my-custom-training-preset.preset.yml
```

   In a Linux environment:

```
simics> !cat my-custom-training-preset.preset.yml
```

   The file is expected to look like this. Make sure that the top level after the initial "**args:**" is "**platform**", and not "**001_qsp_training**":

```
%YAML 1.2
---

args:
  platform:
    machine:
      system_info: My custom training setup
      hardware:
        memory_megs: 16384
        processor:
          num_cores: 8
...
```

6. **Quit** the Simics session.

```
simics> quit
```

Start a simulation session with parameters from a preset file

7. Now you can start the training target and pass the preset file to set the parameters:

```
[$|C:>] simics[.bat] simics-user-training/001-qsp-training --preset
my-custom-training-preset.preset.yml
```

   If you get an error saying "**Arguments provided for non-declared parameters … ['001_qsp_training:platform:machine:system_info'**…" you missed the step of removing the top-level namespace when saving the file.

8. Once the session is up, verify that all your settings have been applied.

```
simics> print-target-info
```

Which should show:

```
My custom training setup
   ┌─────────┬──────────────────────┐
   │System   │a QSP x86 chassis     │
   │Processors│8 QSP X86-64, 2000.0 MHz│
   │Memory   │16 GiB                │
   │Ethernet │1 of 1 connected      │
   │Storage  │2 disks (300 GiB)     │
   └─────────┴──────────────────────┘
```

9. **Quit** the simulation session.

```
simics> quit
```

Create a launchable preset

The preset you created only provides a set of values for particular named parameters. To use it, you needed to provide both the target and the preset on the command line. The preset could theoretically be used with another script that uses the same-named parameters. It is also possible to define a preset that directly launches a particular target and that effectively works as a target in its own right. Such files are called **launchable presets**.

10. Copy **`my-custom-training-preset.preset.yml`** to **`my-custom-training-setup.target.yml`**. I.e., using "**`target`**" instead of "**`preset`**" in the filename extension, and changing from "**`preset`**" to "**`setup`**" in the name.

11. Add a **`target:`** line to **`my-custom-training-setup.target.yml`** as shown below. This line tells the simulator to apply the provided parameter values to a given target, creating a self-contained target.

```
%YAML 1.2
---
args:
  platform:
    machine:
      system_info: My custom training setup
      hardware:
        memory_megs: 16384
        processor:
          num_cores: 8
target: simics-user-training/001-qsp-training
...
```

12. Now, you can run this launchable preset directly as an argument to the simulator, giving the name of the file:

```
[$|C:>] simics[.bat] my-custom-training-setup.target.yml
```

13. Once the session is up, verify that all your settings have been applied.

```
simics> print-target-info
```

14. **Quit** the simulation session.

```
simics> quit
```

Listing presets and targets

15. So far, you passed both the preset as well as the launchable preset (target) as files to the simulator. The original target was started by using its name. Let's check if the Intel Simics simulator can find your custom preset and target in such a way that they can be used by name (i.e., without the file extensions or path to a file). Invoke the two commands below.

    ```
    [$|C:>] simics[.bat] --list-targets
    ```

    And:

    ```
    [$|C:>] simics[.bat] --list-presets
    ```

    You will see targets and presets from the installed packages but not from the project. The reason is that the simulator only searches for targets and presets in specific directories of your project and the installed packages.

16. Move **`my-custom-training-preset.preset.yml`** and **`my-custom-training-setup.target.yml`** into **`<project>/targets/simics-user-training`**.

17. Now repeat the list commands. The result should be something like the one below. Note that the "package" for the target and preset are indicated as **`<project>`**:

    ```
    C:\... > simics.bat --list-targets
    +--------------------------------------------+--------------------+
    |                   Preset                   |      Package       |
    +--------------------------------------------+--------------------+
    […]
    |simics-user-training/my-custom-training-setup|<project>          |
    +--------------------------------------------+--------------------+
    ```

    And:

    ```
    C:\... > simics.bat --list-presets
    +---------------------------------------------+--------------------+
    |                   Preset                    |      Package       |
    +---------------------------------------------+--------------------+
    […]
    |simics-user-training/003-12-core             |Simics Training     |
    |simics-user-training/003-12-GiB              |Simics Training     |
    |simics-user-training/my-custom-training-preset|<project>          |
    +---------------------------------------------+--------------------+
    ```

    After this, your custom files can be used by name and not just by filename.

18. Start the simulator using your launchable preset by using it as a named target (and not using the filename):

    ```
    [$|C:>] simics[.bat] simics-user-training/my-custom-training-setup
    ```

19. Verify the target info:

    ```
    simics> print-target-info
    ```

20. **Quit** this simulation session:

```
simics> quit
```

Launching with a named preset

21. The plain preset file can now be used by name when parametrizing the standard setup:

    ```
    [$|C:>] simics[.bat] simics-user-training/001-qsp-training --preset
    simics-user-training/my-custom-training-preset
    ```

    The same method is used to apply standard presets provided by an Intel Simics package. Multiple `--preset` arguments can be provided to a single target.

22. Verify the target info:

    ```
    simics> print-target-info
    ```

23. **Quit** this simulation session:

    ```
    simics> quit
    ```

Use load-target with a named preset

You can also apply presets when loading a target from the simulator command line inside a running simulator session.

24. Start a new empty simulation session:

    ```
    [$|C:>] simics[.bat]
    ```

25. Load the standard training target with the custom named preset you created above:

    ```
    simics> load-target simics-user-training/001-qsp-training
    preset=simics-user-training/my-custom-training-preset
    ```

26. Verify the target info:

    ```
    simics> print-target-info
    ```

27. **Quit** this simulation session:

    ```
    simics> quit
    ```

Launching with multiple named presets

Intel Simics Simulator target packages can provide standard presets to control different aspects of the target setup. If these presets control orthogonal aspects of the target system, they can be stacked and used together. The training setup comes with two standard presets intended to demonstrate this ability.

28. From the shell, list the available presets again:

    ```
    [$|C:>] simics[.bat] --list-presets
    ```

29. Build a launch command that uses both the presets called "**`003-`**…" (copying the names from the `--list-presets` output is an easy way to get the names right). Note that each preset file is preceded by a `--preset` option.

```
[$|C:>] simics[.bat] simics-user-training/001-qsp-training --preset
simics-user-training/003-12-core --preset simics-user-training/003-12-
GiB
```

30. Once the simulator has started, check the target configuration:

```
simics> print-target-info
```

It should indicate that the system has 12 processor cores and 12GiB of memory.

31. Check the changed parameters:

```
simics> params.list -only-changed
```

Which should show two changed parameters:

```
┌──────────────────────────────────────────────────────────────┬────┬──────┐
│Name                                                          │Type│Value │
├──────────────────────────────────────────────────────────────┼────┼──────┤
│001_qsp_training:platform:machine:hardware:memory_megs        │int │12_288│
│001_qsp_training:platform:machine:hardware:processor:num_cores│int │12    │
└──────────────────────────────────────────────────────────────┴────┴──────┘
```

In general, each preset will contain more than one parameter. This is just intended as a simple example.

32. Check which file provided the value of one of the changed parameters:

```
simics> params.describe
001_qsp_training:platform:machine:hardware:memory_megs
```

The output should show that the parameter comes from a file in the installation.

```
[…]
state: user
file: […]/targets/simics-user-training/003-12-GiB.preset.yml
```

33. **Quit** this simulation session:

```
simics> quit
```

Use load-target with multiple named presets

Multiple presets can also be used when launching a new session from the simulator command line.

34. Start a new empty simulation session:

```
[$|C:>] simics[.bat]
```

35. List the presets, including their description:

```
simics> list-presets -verbose
```

Note that verbose output is not currently available when listing presets from the host shell.

36. Launch a simulation using the standard training target, and the two standard presets. Unfortunately, you cannot use the **`preset=`** argument multiple times. Instead, you use the **`presets=`** argument that takes a list of presets.

```
simics> load-target "simics-user-training/001-qsp-training" presets =
["simics-user-training/003-12-GiB","simics-user-training/003-12-core"]
```

37. Once the simulator has started, check the target configuration:

    ```
    simics> print-target-info
    ```

    It should indicate that the system has 12 processor cores and 12GiB of memory.

38. **Quit** this simulation session:

    ```
    simics> quit
    ```

Adding pre-load and post-load actions to a target

Customizing a target by changing the values of parameters is sometimes not enough to achieve what you want. For example, you might want to start with a given platform, and then instantiate additional network elements, add PCIe cards, or add custom script branches for custom actions.

One way is to build a complete custom target with a **`.target.yml`** and a script (like the **`001-qsp-training`** target that you inspected above). Creating such a fully-fledged target makes sense when you want to package and ship your script to other users. In most cases, you just want to automate things for yourself, and there is no need to expose parameters and configurability to further users. In that case, creating an Intel Simics simulator script file is simpler and sufficient.

39. **Create** a file called **`<project>/targets/simics-user-training/lab003-target-automation.simics`** in your favorite editor. The file could also be put at the project top level, depending on if you think it is neater to keep all files inside the target they belong to or just globally in the project.

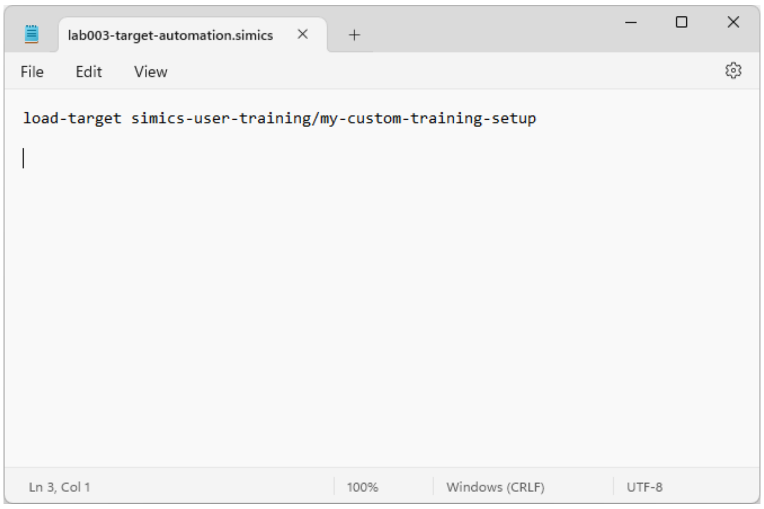
40. Add the below line to the file to load your launchable preset.

    ```
    load-target simics-user-training/my-custom-training-setup
    ```

    Note that you could pass parameters to the **`load-target`** command to provide additional customizations. Any of the uses of **`load-target`** shown above would still apply.

41. **Save** the file.

    The file should look like this:



```
load-target simics-user-training/my-custom-training-setup
```

42. **Run** your newly created Simics script.

```
[$|C:>] simics[.bat] targets/simics-user-training/lab003-target-
automation.simics
```

The simulation will start with the custom setup you have seen several times already.

43. **Verify** the target info:

```
simics> print-target-info
```

44. **Quit** the simulation session:

```
simics> quit
```

45. Add an admittedly trivial pre-load action to the script:

```
echo "------------------------------"
echo "About to load the custom target"
echo "------------------------------"

load-target simics-user-training/my-custom-training-setup
```

46. Also, add a post-load action: Add a script branch that waits for the automatic login and then lists the files there. Plus, a command to reduce the log output. In order to retrieve the name of the serial console object, you need to know the namespace that the parameters were put into. This is retrieved as the return value from the **`load-`**

target command, and then used in **params.get** to construct the full name of the parameter.

```
echo "------------------------------"
echo "About to load the custom target"
echo "------------------------------"

$ns = ( load-target simics-user-training/my-custom-training-setup )

script-branch "List files after boot" {
    local $sercon = (params.get $ns+":output:system").serconsole.con
    bp.console_string.wait-then-write $sercon "root@machine:~#" "ls\n"
    log-level 1  ## restore log level after the boot!
}
log-level 0
```

47. Now run your customized target with pre-load and post-load actions.

```
[$|C:>] simics[.bat] targets/simics-user-training/lab003-target-
automation.simics
```

You should see the pre-load message stating that the target is about to be loaded when the simulator has started.

48. Check that the script branch was created:

```
simics> list-script-branches
```

There should be a script branch listed, coming from your script.

49. **Run** the simulation:

```
simics> r
```

50. **Wait** for the boot process to complete.

Once the target reaches the prompt on the serial console, you should see the script branch that you added as a post-load action to change the user to root and change the directory.

51. **Quit** the simulation session.

```
simics> quit
```

C. Start simulations using old-style scripts

The old style of starting Intel Simics Simulations is to use **`.simics`** files trampolines in the Intel Simics project. The **project setup** script generates a **trampoline script** in the project for each **`.simics`** file in the **`targets/`** directory structure in a package. The trampolines simply run the corresponding script in the package. The **`simics-user-training`** target contains a script called **`003-qsp-training-old-style.simics`** that demonstrates this functionality.

Find the trampoline

1. List the files **`<project>/targets/simics-user-training`** (i.e., the targets directory in your project).

   In a Windows CMD environment:

   ```
   C:…> dir targets\simics-user-training\
   ```

   In a Linux environment:

   ```
   $ ls targets/simics-user-training/
   ```

   There should be a file called **`targets/simics-user-training/003-qsp-training-old-style.simics`**, among other files you created in the labs up until now.

2. Open the file **`003-qsp-training-old-style.simics`** in an editor. You will see content similar to what is shown below.

   ```
   # Auto-generated file. Any changes will be overwritten!
   run-script "/home/user/simics/pkgs/simics-7.x.y/../simics-training-
   7.a.b/targets/simics-user-training/003-qsp-training-old-style.simics"
   ```

   Note how the file contains a single **`run-script`** command that points at a fixed path. The path depends on the location of your current local Intel Simics Simulator installation and the specific versions of packages installed.

Launching using a trampoline script

3. To start a new simulation session using a trampoline, you provide the name of the script as an argument to the simulator program. The name can be tab completed by the host shell.

   ```
   [$|C:>] simics[.bat] targets/simics-user-training/003-qsp-training-
   old-style.simics
   ```

4. Use the **`command-file-history`** command to see the invocations following from the trampoline.

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

   The top of the history looks like this:

```
┌─────────────────────────────────┬───────┐
│Command File                     │Action │
├─────────────────────────────────┼───────┤
│003-qsp-training-old-style.simics │started│
│003-qsp-training-old-style.simics │started│
│001-qsp-training.target.yml.include│started│
…
```

The two same-name scripts on the top are the trampoline and the "real" script in the package that it runs.

5. Check the full path to each file that is run using the **-v** flag:

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

The second line shows the same path as used in the trampoline.

6. Check the target information:

```
simics> print-target-info
```

The output should look like this:

```
Old-fashioned setup with a theme of three
    ┌──────────┬─────────────────────────┐
    │Namespace │machine                  │
    │System    │a QSP x86 chassis        │
    │Processors│3 QSP X86-64, 3333.0 MHz │
    │Memory    │9 GiB                    │
    │Ethernet  │1 of 1 connected         │
    │Storage   │2 disks (300 GiB)        │
    └──────────┴─────────────────────────┘
```

You will get back to these settings.

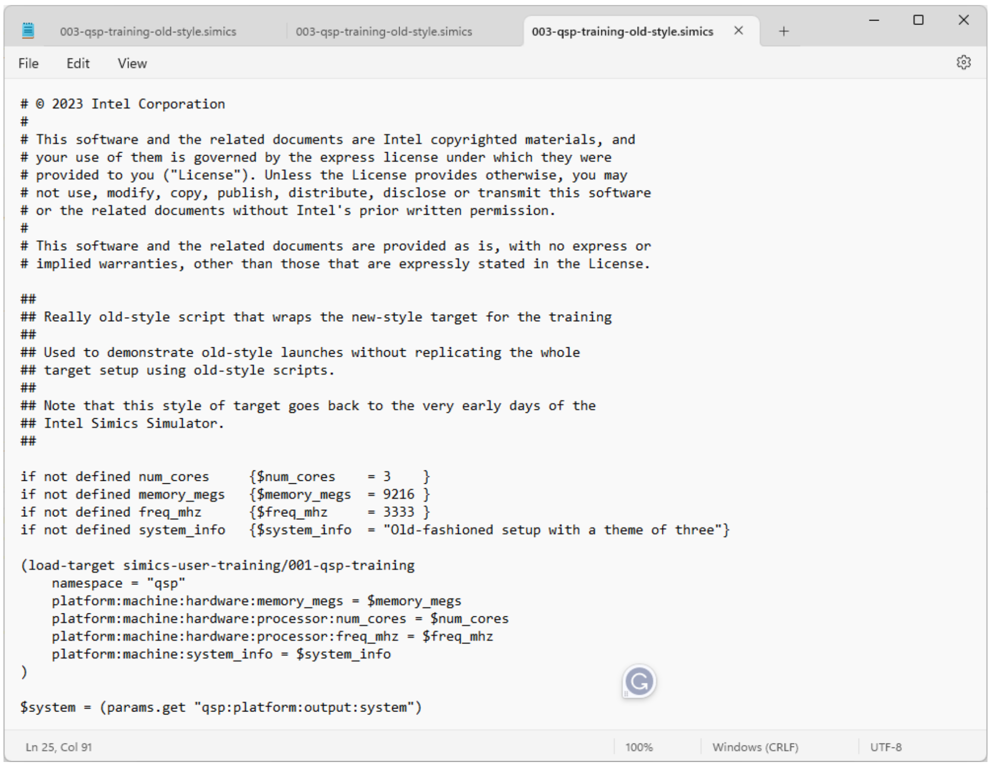
7. **Quit** the simulation session.

```
simics> quit
```

Set parameters to an old-style script

Old-style scripts do not have a formal parameter system like new-style targets. Instead, they rely on setting command-line variables (CLI variables) to certain values before running the script. Default values are created for any relevant variables that lack a value.

8. Open the underlying script file in your Intel Simics Simulator installation. The path is found both in the trampoline script and in the **`command-file-history -v`** output.


003-qsp-training-old-style.simics
003-qsp-training-old-style.simics
003-qsp-training-old-style.simics
File
Edit
View
Ln 25, Col 91
100%
Windows (CRLF)
UTF-8

```
# © 2023 Intel Corporation
#
# This software and the related documents are Intel copyrighted materials, and
# your use of them is governed by the express license under which they were
# provided to you ("License"). Unless the License provides otherwise, you may
# not use, modify, copy, publish, distribute, disclose or transmit this software
# or the related documents without Intel's prior written permission.
#
# This software and the related documents are provided as is, with no express or
# implied warranties, other than those that are expressly stated in the License.

##
## Really old-style script that wraps the new-style target for the training
##
## Used to demonstrate old-style launches without replicating the whole
## target setup using old-style scripts.
##
## Note that this style of target goes back to the very early days of the
## Intel Simics Simulator.
##

if not defined num_cores     {$num_cores    = 3    }
if not defined memory_megs   {$memory_megs  = 9216 }
if not defined freq_mhz      {$freq_mhz     = 3333 }
if not defined system_info   {$system_info  = "Old-fashioned setup with a theme of three"}

(load-target simics-user-training/001-qsp-training
    namespace = "qsp"
    platform:machine:hardware:memory_megs = $memory_megs
    platform:machine:hardware:processor:num_cores = $num_cores
    platform:machine:hardware:processor:freq_mhz = $freq_mhz
    platform:machine:system_info = $system_info
)

$system = (params.get "qsp:platform:output:system")
```

The "parameters" to this file are "declared" by the "**`if not defined…`**" lines.

9. When starting a new Intel Simics Simulator session from the host shell, you can provide values for CLI variables as **`name=value`** pairs in the command starting the simulator.

   Set the number of cores to 5 and the frequency to 5555.

```
[$|C:>] simics[.bat] targets/simics-user-training/003-qsp-training-
old-style.simics num_cores=5 freq_mhz=5555
```

10. Check the target setup:

```
simics> print-target-info
```

   The output should look like this:

```
Old-fashioned setup with a theme of three
    ┌──────────┬─────────────────────────┐
    │Namespace │machine                  │
    │System    │a QSP x86 chassis        │
    │Processors│5 QSP X86-64, 5555.0 MHz │
    │Memory    │9 GiB                    │
    │Ethernet  │1 of 1 connected         │
    │Storage   │2 disks (300 GiB)        │
    └──────────┴─────────────────────────┘
```

11. The variables used to set this up are still present after the script has finished:

```
simics> list-variables
```

This shows the variables from the script, plus a **`$system`** variable that returns the name of the top-level namespace/component of the created machine for use in further scripting. This is a common pattern in old-style scripts.

12. Test the **`$system`** variable by checking the info command on the serial console object:

```
simics> $system.serconsole.con.info
```

13. Quit the simulation session.

```
simics> quit
```

D. Inspect the target system

In this section, you will look at the contents of the target system from the CLI.

A simulation model consists of objects, which usually represent hardware elements (but some are simulation artifacts, like the breakpoint manager). The simulation objects form a hierarchy, and well-designed models make use of the hierarchy to group related objects together and put subordinate objects underneath the objects that they support.

A special class of objects are the **components** that create other objects and provide the large-scale structure of the simulated system. They are used to hierarchically and logically group other objects (including other components), connect them internally, and offer connectors between components that hide the complexity of connecting many individual objects into simple "connect component A to component B" statements. Examples of components are processor clusters, systems on chips, or entire boards. Components only give structure, not functionality. Functionality is implemented by other objects (often referred to as "devices").

Note that every component is an object, but not every object is a component.

Component structure

14. Start a new simulation session.

```
[$|C:>] simics[.bat] simics-user-training/001-qsp-training
```

15. You can see the top-level components using the list-components command.

```
simics> list-components
```

This should show three top-level components:

```
┌───────────────┬─────────────────┐
│Component      │Class            │
├───────────────┼─────────────────┤
│ethernet_switch0│ethernet_switch  │
│machine        │chassis_qsp_x86  │
│service_node_cmp0│service_node_comp│
└───────────────┴─────────────────┘
```

There can be multiple components at the top level. In this case, the top level contains an Ethernet network with a service node (networking will come up in later labs) and the machine called "**`machine`**."

16. Look inside the machine to see how it is split into components:

```
simics> list-components machine
```

The top level of the machine contains **`mb`**, the motherboard where most of the hardware is located. Next to this, there are standard components like disks, consoles, and input devices. There is also the training card.

17. Components have **connectors** that typically correspond to end-user-visible connections in the hardware. Typical examples are PCIe, Ethernet, USB, serial, and

SATA. The connectors of a component are shown using its **status** command. Look at the connections to the training card:

```
simics> machine.training_card.status
```

It shows that its PCI connector is connected to the PCIe slot on the northbridge chip of the platform (**machine.mb.nb**).

18. To see all the connectors in the system and what they connect to, use the **-v** flag to **list-components**.

```
simics> list-components machine -v
```

Note that most of the connections go between different levels of the hierarchy of components. For example, look at the console (the graphics console) and what it connects to:

```
Component            Class
...

console        gfx_console_comp
abs_mouse      abs-mouse              up     machine.tablet:abs_mouse
device         graphics-console       up     machine.mb.gpu:console
keyboard       keyboard               up     machine.mb.sb:kbd_console
mouse          mouse                  up     machine.mb.sb:mse_console
...
```

19. Look at the components and connectors one level down in the **machine.mb**:

```
simics> list-components machine.mb -v
```

This shows the counterpart to many of the connections listed in the previous command. For example, the **machine.mb.sb** component shows the connectors to the graphics console (**machine.console**) (and more):

```
...
sb             southbridge_ich10
eth_slot       ethernet-link         down   ethernet_switch0:device0
ide_slot[0]    ide-slot              down
ide_slot[1]    ide-slot              down
ide_slot[2]    ide-slot              down
ide_slot[3]    ide-slot              down
ide_slot[4]    ide-slot              down
ide_slot[5]    ide-slot              down
kbd_console    keyboard              down   machine.console:keyboard
lpc_bus        lpc-bus               down
mse_console    mouse                 down   machine.console:mouse
pci_slot[0]    pci-bus               down
pci_slot[1]    pci-bus               down
pci_slot[2]    pci-bus               down
pci_slot[3]    pci-bus               down
sata_slot[0]   sata-slot             down   machine.disk0:sata_slot
sata_slot[1]   sata-slot             down   machine.cdrom:sata_slot
sata_slot[2]   sata-slot             down   machine.disk1:sata_slot
...
```

The object hierarchy

The object hierarchy is the more general concept in the simulator. Over time, more and more structure is expressed using the hierarchy without necessarily putting in explicit components at each level. To inspect the object hierarchy, use the very flexible **`list-objects`** command.

20. Try the **`list-objects`** command. It will show all objects in the configuration, except port and bank objects, with components separated out in their own table at the beginning of the output.

    ```
    simics> list-objects
    ```

21. More useful output is usually achieved by using options to limit the scope.

    Inspect the top level of the target machine with the **`-local`** option to avoid showing the whole object hierarchy:

    ```
    simics> list-objects -local namespace = machine
    ```

    The result is much more manageable in size:

    ```
    ┌─────────────────────┬──────────────────────────┐
    │       Object        │     Component Class      │
    ├─────────────────────┼──────────────────────────┤
    │machine.cdrom        │<sata_cdrom_comp>         │
    │machine.console      │<gfx_console_comp>        │
    │machine.disk0        │<sata_disk_comp>          │
    │machine.disk1        │<sata_disk_comp>          │
    │machine.mb           │<motherboard_x58_ich10>   │
    │machine.serconsole   │<txt_console_comp>        │
    │machine.tablet       │<usb_tablet_component>    │
    │machine.training_card│<led_system_comp>         │
    └─────────────────────┴──────────────────────────┘
    ┌────────────────────┬──────────────┐
    │       Object       │    Class     │
    ├────────────────────┼──────────────┤
    │machine.cell        │<cell>        │
    │machine.cell_context│<context>     │
    │machine.cell_rec0   │<recorder>    │
    │machine.software    │<os_awareness>│
    └────────────────────┴──────────────┘
    ```

22. The **`list-objects`** command can present the set of objects as a tree. This often provides a better overview of the system structure than a simple table of objects with hierarchical names.

    ```
    simics> list-objects -tree namespace = machine
    ```

    The output is fairly extensive but manageable for a simple platform like the Quick-Start Platform used in the training. For more complex platforms, it is likely to be overwhelming.

23. Focus the tree on **`machine.mb`**, and only show the top two levels.

    ```
    simics> list-objects namespace = machine.mb -tree max-depth = 2
    ```

    This shows that most of the model device objects are found in the **`cpu0`**, **`sb`**, and **`nb`** sub-trees.

24. Dig into **`machine.mb.sb`**, and list all the objects in table form.

    ```
    simics> list-objects namespace = machine.mb.sb
    ```

    This table is short enough to be perused.

25. Use the tree view and show the classes of the objects. Note that some objects are of the class **`connector`**, which are the component connectors. Component connectors are also objects.

    ```
    simics> list-objects namespace = machine.mb.sb -with-class-name -tree
    ```

26. Register banks and interface ports of objects are collectively known as **port objects**. These are considered helper objects and are hidden by default to keep the **`list-objects`** output reasonable. However, there are cases where it is important to see the ports and banks of an object, which is supported by the **`-show-port-objects`** flag.

    Note: older versions of the Intel Simics Simulator used the opposite logic for port objects: they were shown by default and had to be explicitly hidden. Keep this in mind when using older versions of the simulator and reading examples that have not been updated to the new behavior.

    Show the port objects in the tree. Leave the class names hidden to keep the display clean:

    ```
    simics> list-objects namespace = machine.mb.sb -show-port-objects -
    tree
    ```

    Which looks something like this:

    ```
    ┐
    ├ bridge ┐
    │        ├ bank ┐
    │        │      ├ outbound_io1
    │        │      ├ outbound_io2
    │        │      ├ outbound_mem1
    │        │      ├ outbound_mem2
    │        │      ├ outbound_mem3
    │        │      └ pci_config
    │        └ port ┐
    │               └ pltrst
    │
    ...
    ├ uhci[5] ┐
    │         └ bank ┐
    │                ├ pci_config
    │                ├ usb_ctrl
    │                └ usb_regs
    └ usb_port[0..11]
    ```

27. To see the structure of just a single device, use **`list-objects`** with the tree view on the **`lan`** device in **`machine.mb.sb`**:

```
simics> list-objects namespace = machine.mb.sb.lan -tree -show-port-
objects
```

This shows the register banks and port objects of the LAN controller in a rather small tree.

```
┐
├ bank ┐
│      ├ csr
│      ├ io_mapped
│      └ pci_config
└ port ┐
       ├ HRESET
       └ SRESET
```

28. Another way to limit the output of **`list-objects`** to something specific is to use the **`substr`** argument to search for matching objects. Locate the LAN controller in the platform, by looking for an object called **`lan`**.

```
simics> list-objects substr = lan
```

The result shows the **`machine.mb.sb.lan`** object without port objects:

```
┌──────────────┬─────────────────┐
│    Class     │     Object      │
├──────────────┼─────────────────┤
│<ich10_lan_v2>│machine.mb.sb.lan│
└──────────────┴─────────────────┘
```

29. To see the hierarchy of leading down the device, use the -**`tree`** option along with **`substr`**:

```
simics> list-objects -tree substr = lan -show-port-objects
```

This should result in output like shown below, visualizing the hierarchy on the CLI:

```
┐
└ machine ┐
          └ mb ┐
               └ sb ┐
                    └ lan ┐
                          ├ bank ┐
                          │      ├ csr
                          │      ├ io_mapped
                          │      └ pci_config
                          └ port ┐
                                 ├ HRESET
                                 └ SRESET
```

30. It is also possible to find objects based on their class. To find all the old-style serial ports, list all objects of class **`NS16550`**:

```
simics> list-objects class = NS16550
```

This will find four serial port device objects.

Iterating over objects

The list-objects command can return a list of the objects found. Assign the value of the command to a CLI variable to get a list that CLI can then iterate over.

31. Retrieve the list of serial ports into a variable **`$coms`**:

```
simics> $coms = (list-objects class=NS16550)
```

32. Check the returned set of objects:

```
simics> $coms
```

This should show a list containing the names of the four serial ports:

```
["machine.mb.sb.com[0]", "machine.mb.sb.com[1]", "machine.mb.sb.com[2]",
"machine.mb.sb.com[3]"]
```

33. Write a small loop in CLI that iterates over all the serial ports and runs their info commands:

```
simics> foreach $c in $coms { $c.info }
```

This will print out the information for each device, including, in particular, whether they are connected to serial consoles (i.e., serial console windows).

Object attributes

34. You can list and inspect attribute values from the command line. To see all attributes of an object, use the **`list-attributes`** command. Apply it to the LAN controller object (**`machine.mb.sb.lan`**):

```
Simics> list-attributes machine.mb.sb.lan
```

The output shows a list of the attributes, the type of CLI values that can be assigned to them, their attribute kind, and current value. Note that the default output only shows attributes that are considered to be useful for users or the configuration.

```
┌─────────────────────────┬────────┬────────┬─────┬───────────────────────────────────────────────────────────┐
│Attribute                │Type    │Attr    │Flags│Value                                                      │
├─────────────────────────┼────────┼────────┼─────┼───────────────────────────────────────────────────────────┤
│aes_gcm                  │o|[os]|n│Optional│     │NIL                                                        │
├─────────────────────────┼────────┼────────┼─────┼───────────────────────────────────────────────────────────┤
│chipset_config           │o|[os]  │Required│     │[machine.mb.sb.lpc, "cs_conf"]                             │
├─────────────────────────┼────────┼────────┼─────┼───────────────────────────────────────────────────────────┤
│config_registers         │[i*]    │Pseudo  │     │[281_903_238, 1_048_576, 33_554_432, 0, 0, 0, 1, 0, 0, 0,  │
│                         │        │        │     │0, 32_902, 0, 200, 0, 256, 0, 0, 0, 0, 0, ...              │
├─────────────────────────┼────────┼────────┼─────┼───────────────────────────────────────────────────────────┤
...
```

Required attributes must be set during simulation setup for the configuration to be valid. **Optional** attributes typically hold the current state of the object and are saved in checkpoints. **Pseudo** attributes report something about the nature or state of configuration of the object, but are not saved and restored in checkpoints.

35. To see all the attributes of an object, use the **`-i`** flag:

```
simics> list-attributes machine.mb.sb.lan -i
```

This list is much longer and includes a huge set of pseudo attributes that correspond to register values in the banks of the device. These are present for backwardcompatibility reasons and will be removed at some point. To see the attributes for registers in the banks, the correct way is to look at the bank itself.

36. The list-attributes command can be used to search for attributes that match a particular substring. For example, to find all attributes containing the string “mac”:

```
simics> list-attributes machine.mb.sb.lan substr = mac
```

Which will show a single attribute in the same format as above.

37. To just read the value of an attribute, use the **`obj->attribute`** syntax. Use this to read the **`mac_address`** attribute:

```
simics> machine.mb.sb.lan->mac_address
```

This displays or returns the value:

```
simics> machine.mb.sb.lan->mac_address
"00:17:A0:00:00:00"
```

38. Additional information about the attribute can be retrieved using the **`attribute-name`** argument to the **`list-attributes`** command:

```
simics> list-attributes machine.mb.sb.lan attribute-name = mac_address
```

This includes any help strings that have been defined for the attribute. In this case, it documents the attribute format. The current value is also shown.

```
Attribute <ich10_lan_v2>.mac_address

   Optional attribute; read/write access; type: string.

   MAC address ('XX:XX:XX:XX:XX:XX' string)

   Value: "00:17:A0:00:00:00"
```

Info and Status commands

39. Many devices in a simulation (but not all) feature **`info`** and **`status`** commands that provide a quick overview of their configuration and current operational status. The output of these commands can be seen from CLI. Try it on the **`lan`** device:

```
simics> machine.mb.sb.lan.info
```

Note that this also shows the MAC address.

40. And then for **`status`**:

```
simics> machine.mb.sb.lan.status
```

As you can see, the idea for these commands is to provide a high-level summary of the device status and configuration.

E. Explore memory maps and object connections

In this section, you will explore memory maps and how the simulated system is put together.

Exploring the initial memory maps

1. The first step is to find the processors in the system.

```
simics> list-processors
```

This should show the two processors:

```
┌──────────────────────────┬─┬─────────┬────────┐
│        CPU Name          │ │CPU Class│  Freq  │
├──────────────────────────┼─┼─────────┼────────┤
│machine.mb.cpu0.core[0][0]│*│x86QSP1  │2.00 GHz│
│machine.mb.cpu0.core[1][0]│ │x86QSP1  │2.00 GHz│
└──────────────────────────┴─┴─────────┴────────┘
* = selected CPU
```

2. Each processor in a simulation normally has a specific physical memory space that represents its addressable memory. To find it, use the **info** command on a processor. Use the first processor listed by the previous command:

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

This shows the name of the physical memory of the processor:

```
simics> machine.mb.cpu0.core[0][0].info
Information about machine.mb.cpu0.core[0][0] [class x86QSP1]
============================================================

...
                Physical memory : machine.mb.cpu0.mem[0][0]
...
```

3. The **map** command on the memory space shows the mappings in the space:

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

With output like this:

```
┌───────────┬──────────────────────────┬──┬──────┬──────┬──────┬────┬─────┬────┐
│       Base│Object                    │Fn│Offset│Length│Target│Prio│Align│Swap│
├───────────┼──────────────────────────┼──┼──────┼──────┼──────┼────┼─────┼────┤
│0xfee0_0000│machine.mb.cpu0.apic[0][0]│  │0x0000│0x1000│      │   0│    8│    │
│  -default-│machine.mb.phys_mem       │  │0x0000│      │      │    │     │    │
└───────────┴──────────────────────────┴──┴──────┴──────┴──────┴────┴─────┴────┘
```

You can see that the core has its local APIC mapped to memory, and a default mapping is used to pass on all other accesses to the **machine.mb.phys_mem** memory space. In this particular system, all processor cores use the same **phys_mem** and thus see the same devices and memory mapped in the same physical location (except for their private APICs).

4. Check the memory map of **machine.mb.phys_mem**:

```
simics> machine.mb.phys_mem.map
```

Which should show:

```
|           Base|Object          |Fn|         Offset|         Length|Target|Prio|Align|Swap|
|         0x0000|machine.mb.dram_|  |         0x0000|    0x000a_0000|      |   0|     |    |
|               |space           |  |               |               |      |    |     |    |
|    0x0010_0000|machine.mb.dram_|  |    0x0010_0000|    0xdff0_0000|      |   0|     |    |
|               |space           |  |               |               |      |    |     |    |
|0x0001_0000_0000|machine.mb.dram_|  |0x0001_0000_0000|0x0005_0000_0000|      |   0|     |    |
|               |space           |  |               |               |      |    |     |    |
|      -default-|machine.mb.nb.  |  |         0x0000|               |      |    |     |    |
|               |pci_bus.port.mem|  |               |               |      |    |     |    |
```

5. All accesses that miss RAM are sent to the PCI memory of the chipset. Check the map of that memory space:

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

The map looks like this:

```
|       Base|Object                |Fn|Offset|     Length|Target               |Prio|Align|Swap|
|0x000a_0000|machine.mb.gpu.dmap_  |  |0x0000|0x0002_0000|                     |   0|     |    |
|           |space                 |  |      |           |                     |    |     |    |
|0x000c_0000|machine.mb.shadow     |  |0x0000|0x0004_0000|machine.mb.shadow_mem|   1|     |    |
|0xfec0_0000|machine.mb.sb.ioapic  |  |0x0000|     0x0020|                     |  -1|    8|    |
|0xffe0_0000|machine.mb.rom        |  |0x0000|0x0020_0000|                     |   0|     |    |
|  -default-|machine.mb.dram_space |  |0x0000|           |                     |    |     |    |
```

The memory map is almost empty because the system has not yet booted, and the PCI(e) mappings have not yet been configured by the software.

Boot the system to configure PCIe mappings

6. Boot the system: Go to the command line, and issue:

```
simics> run 20 s
```

7. Wait until the simulation stops.

Investigate memory mappings after the boot

8. Check the contents of **`machine.mb.nb.pci_bus.port.mem`** again. It now contains a lot of devices mapped into PCI memory-mapped IO (MMIO) space. Each mapping corresponds to one PCI BAR, or mappable range of addresses.

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

The memory space map now looks like this:

```
┌──────────┬────────────────────┬──┬──────────┬──────────┬─────────────────┬────┬─────┬────┐
│      Base│Object              │Fn│    Offset│    Length│Target           │Prio│Align│Swap│
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
...
│0xf100_0000│machine.mb.nb.pcie_ │  │0xf100_0000│0x0110_0000│                 │   2│     │    │
│          │p2.downstream_port. │  │          │          │                 │    │     │    │
│          │port.mem            │  │          │          │                 │    │     │    │
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
│0xf210_0000│machine.mb.sb.lan.  │  │    0x0000│0x0002_0000│                 │   0│    8│    │
│          │bank.csr            │  │          │          │                 │    │     │    │
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
│0xf212_0000│machine.mb.sb.sata2.│  │    0x0000│    0x0800│                 │   0│    8│    │
│          │bank.ahci           │  │          │          │                 │    │     │    │
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
│0xf212_0800│machine.mb.sb.smbus.│  │    0x0000│    0x0020│                 │   0│    8│    │
│          │bank.smbus_func     │  │          │          │                 │    │     │    │
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
│0xf212_1000│machine.mb.sb.      │  │    0x0000│    0x0400│                 │   0│    8│    │
│          │ehci[0].bank.usb_   │  │          │          │                 │    │     │    │
│          │regs                │  │          │          │                 │    │     │    │
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
│0xf212_2000│machine.mb.sb.      │  │    0x0000│    0x0400│                 │   0│    8│    │
│          │ehci[1].bank.usb_   │  │          │          │                 │    │     │    │
│          │regs                │  │          │          │                 │    │     │    │
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
...
├──────────┼────────────────────┼──┼──────────┼──────────┼─────────────────┼────┼─────┼────┤
│ -default-│machine.mb.dram_    │  │    0x0000│          │                 │    │     │    │
│          │space               │  │          │          │                 │    │     │    │
└──────────┴────────────────────┴──┴──────────┴──────────┴─────────────────┴────┴─────┴────┘
```

Note how mappings for the chipset devices have been added. This is an example of how the memory mappings in a simulation are dynamic and will often change during a simulation run.

The object called **`machine.mb.nb.pcie_p2.downstream_port.port.mem`** provides access to the memory-mapped devices on PCIe port 2 – which is where the training device is mapped. The port objects of the **`downstream_port`** represent various PCIe access types.

The mapping of **`downstream_port.port.mem`** has a non-zero offset of **`0xf100_0000`**, equal to the base address of the mapping. This means that mappings in **`downstream_port.port.mem`** will show the same address as if they had been mapped directly into the **`pci_bus`** map.

9. Look at the objects underneath the **`downstream_port`** object:

```
simics> list-objects -tree namespace =
machine.mb.nb.pcie_p2.downstream_port -show-port-objects
```

Which should result in output like this:

```
┐
├ cfg_space
├ conf_space
├ impl ┐
│      ├ cfg_to_conf
│      ├ conf_to_cfg
│      ├ port ┐
│      │      └ msg_routing
│      └ transaction_to_pci_express
├ io_space
├ mem_space
├ msg_space
└ port ┐
       ├ HOT_RESET
       ├ broadcast
       ├ cfg
       ├ downstream
       ├ ecam
       ├ io
       ├ mem
       ├ msg
       ├ phy
       └ upstream
```

The top-level objects are sub-objects that represent the configuration space (in two variants), IO space, memory space, and message space of the PCIe port. It also has a set of port objects which are *translators* that pass accesses on to the appropriate sub-objects.

10. Run **help** on the port object that is mapped into the PCI memory space, to get an understanding for what it is:

```
simics> help machine.mb.nb.pcie_p2.downstream_port.port.mem
```

11. Follow the trail onwards. Check the memory map of the **downstream_port.port.mem**:

```
simics> machine.mb.nb.pcie_p2.downstream_port.port.mem.map
```

The training device has four mappings in place: three register banks and one memory space.

Base	Object	Fn	Offset	Length	Target	Prio	Align	Swap
0xf100_0000	machine.training_card. local_memory	4	0x0000	0x0100_0000		0		
0xf200_0000	machine.training_card. master_bb.bank.dev_msix_ pba		0x0000	0x0100		0	8	
0xf200_1000	machine.training_card. master_bb.bank.dev_msix_ table		0x0000	0x0100		0	8	
0xf200_2000	machine.training_card. master_bb.bank.regs		0x0000	0x1000		0	8	

Finding where an object is mapped

There are some command-line commands that can be used to find where an object is mapped and what other objects have references to it.

12. Check where the **machine.training_card.master_bb.bank.regs** bank is mapped using the **devs** command:

```
simics> devs machine.training_card.master_bb.bank.regs
```

The result is a table showing all the places where the bank is mapped. Note that banks can be mapped in multiple places with different addresses in each mapping. This is the case here, since the control registers are mapped both via PCIe and as part of the local memory map inside the card:

```
┌─────┬────────────────────┬────────────────────────────┬─────────────────────────┬──┐
│Count│      Device        │           Space            │          Range          │Fn│
├─────┼────────────────────┼────────────────────────────┼─────────────────────────┼──┤
│    0│machine.training_card.│machine.mb.nb.pcie_p2.    │0xf200_2000 - 0xf200_2fff│ 0│
│     │master_bb.bank.regs │downstream_port.mem_space   │                         │  │
│     │                    │machine.training_card.local_│0x1000_0000 - 0x1000_0fff│ 0│
│     │                    │memory                      │                         │  │
└─────┴────────────────────┴────────────────────────────┴─────────────────────────┴──┘
```

The first column shows the number of times the object has been accessed in the current simulation session. In this case it is zero, since the driver for the training card has not yet been loaded.

13. Check where the control registers (**bank.csr**) of the **machine.mb.sb.lan** device are mapped:

```
simics> devs machine.mb.sb.lan.bank.csr
```

The result shows a mapping that has been accessed quite a few times:

```
┌──────┬────────────────────────┬────────────────────────┬─────────────────────────┬──┐
│Count │         Device         │         Space          │          Range          │Fn│
├──────┼────────────────────────┼────────────────────────┼─────────────────────────┼──┤
│19_631│machine.mb.sb.lan.bank.csr│machine.mb.nb.pci_bus.mem_│0xf210_0000 - 0xf211_ffff│ 0│
│      │                        │space                   │                         │  │
└──────┴────────────────────────┴────────────────────────┴─────────────────────────┴──┘
```

Probing what is behind an address

Another aspect of memory mapping is figuring out what is mapped at a certain address. The CLI **probe-address** command is used to do that. It starts from a memory space or processor and shows where an access to a particular address ends up.

14. Try the **probe-address** command from the perspective of the current processor:

```
simics> probe-address  address = 0xf210_0000
```

This should give an error since virtual address **0xf2100000** is not mapped.

15. Check the precise processor that is currently selected, to understand where the previous probe came from:

```
simics> pselect
```

It is most likely processor core [0][0].

16. Try a physical address instead, using the **p:** prefix in front of the address:

```
simics> probe-address address = p:0xf210_0000
```

Note that **probe-address** will show the register that a probe hits, if such information is available. Probing address **0xf210_000** should hit the LAN device:

```
┌────────────────────────┬───────────┬─────┬─────────────────────────┬────────────┐
│         Target         │  Offset   │Notes│       Added Atoms       │Missed Atoms│
├────────────────────────┼───────────┼─────┼─────────────────────────┼────────────┤
│machine.mb.cpu0.        │0xf210_0000│     │                         │            │
│mem[0][0]               │           │     │                         │            │
├────────────────────────┼───────────┼─────┼─────────────────────────┼────────────┤
│machine.mb.phys_mem     │0xf210_0000│     │                         │            │
├────────────────────────┼───────────┼─────┼─────────────────────────┼────────────┤
│machine.mb.nb.pci_bus.  │0xf210_0000│~    │completion               │pcie_type   │
│port.mem                │           │     │owner=machine.mb.nb.pci_ │            │
│                        │           │     │bus                      │            │
│                        │           │     │pcie_type=pcie_type_t.   │            │
│                        │           │     │PCIE_Type_Mem            │            │
│                        │           │     │                         │            │
├────────────────────────┼───────────┼─────┼─────────────────────────┼────────────┤
│machine.mb.nb.pci_bus.  │0xf210_0000│     │                         │            │
│mem_space               │           │     │                         │            │
├────────────────────────┼───────────┼─────┼─────────────────────────┼────────────┤
│machine.mb.sb.lan.bank. │0x0000_0000│     │                         │            │
│csr                     │           │     │                         │            │
└────────────────────────┴───────────┴─────┴─────────────────────────┴────────────┘
'~' - Translator implementing 'transaction_translator' interface
Destination: machine.mb.sb.lan.bank.csr offset 0x0
Register:    ctrl @ 0x0 (4 bytes) + 0
```

17. **Probe-address** can start the probe from any memory space in the system. This provides a way to investigate what a particular subsystem or processor would actually hit with a particular memory operation.

For memory spaces, all addresses are physical so there is no need to prefix them with **p:**

Check what address **0xfee0_0000** maps to, when seen from processor [1][0], by using **probe-address** on its private physical memory map:

```
simics> probe-address obj = "machine.mb.cpu0.mem[1][0]" address =
0xfee0_0000
```

18. From another processor object:

```
simics> probe-address obj = "machine.mb.cpu0.mem[0][0]" address =
0xfee0_0000
```

Note that the accesses hit different APICs, since each processor has its own local APIC mapped to its local physical memory.

Inspecting memory contents

19. From the command-line, use the **x** command on a memory space to inspect its content. The command takes an address, and optionally the number of bytes to examine: For example:

```
simics> machine.mb.phys_mem.x 0xfffe0 32
```

20. The **x** command has quite a few formatting options. To see the memory bytes grouped into specific word sizes, use the group-by argument. For example:

```
simics> machine.mb.phys_mem.x 0xfffe0 32 group-by = 64
```

Note that the bytes are interpreted in big-endian order by default. In case the values were written from little-endian software (such as that running on X86 processors), use the **-l** flag to interpret the values as little-endian.

21. Use the training card on-board memory as a safe place to experiment:

```
simics> machine.training_card.local_memory.map
```

Note that there is RAM from address zero and up.

22. Write a 64-bit value to memory, using little-endian byte ordering (as if a processor wrote it). Write to address 0x1000:

```
simics> machine.training_card.local_memory.write 0x1000
0x0102030405060708 8 -l
```

23. Check the contents using the **x** commands with default settings:

```
simics> machine.training_card.local_memory.x 0x1000
```

This prints the memory contents as a sequence of big-endian 16-bit values, which conveniently also shows the order of the bytes in memory. Little-endian stores the least significant byte first, which is clearly seen:

```
p:0x00001000  0807 0605 0403 0201 0000 0000 0000 0000  ................
```

24. Grouping the **x** output by 64 bits implicitly shows the big-endian interpretation of the bytes as a 64-bit integer:

```
simics> machine.training_card.local_memory.x 0x1000 group-by = 64
```

This shows the same byte sequence, just grouped differently.

```
p:0x00001000  0807060504030201 0000000000000000        ................
```

25. Group by 64 bits, and interpret the value as little endian:

```
simics> machine.training_card.local_memory.x 0x1000 group-by = 64 -l
```

The output now reflects the value written, but the command no longer displays the character interpretation of the bytes since the order of the bytes would make that confusing.

```
p:0x00001000  0102030405060708 0000000000000000
```

26. Values can also be read from memory using the read and get commands on memory spaces. By default, these commands interpret the values using the byte order of the current processor, but they can also be forced to use a specific byte order.

Read the 8-byte value from 0x1000, and format as hex:

```
simics> hex (machine.training_card.local_memory.read 0x1000 8)
```

27. To force an interpretation as big-endian despite the current processor being a little-endian Intel processor:

```
simics> hex (machine.training_card.local_memory.read 0x1000 8 -b)
```

Inspecting device registers

Device models in Intel Simics virtual platforms often provide metadata about the memory-mapped registers of the devices in the configuration. Some models might be missing the information if it has been explicitly removed for confidentiality reasons. Register information is generated automatically for models written using the **Device Modeling Language (DML)**, the standard way to build models for the Intel Simics Simulator. Models written in other languages might implement the appropriate interfaces to provide register information, but there are no guarantees.

28. Use the command **`print-device-regs`** to list all banks and registers in a device.

```
simics> print-device-regs machine.training_card.master_bb
```

This shows all the registers in all five register banks of the device:

```
Bank: dev_msix_pba
Offset  Name             Size  Value
------------------------------------
     0  pending_bits[0]     8      0

Bank: dev_msix_table
Offset  Name                Size  Value | Offset  Name                Size  Value
----------------------------------------+----------------------------------------
     0  message_address[0]     8      0 |     16  message_address[1]     8      0
     8  message_data[0]        4      0 |     24  message_data[1]        4      0
    12  vector_control[0]      4      1 |     28  vector_control[1]      4      1

Bank: i2cregs
...
Bank: pci_config
...
Bank: regs
Offset  Name                           Size  Value
-----------------------------------------------------
     0  enable                            4      0
     4  reset                             4      0
    16  update_display_request            4      0
    20  update_display_status             4      0
...
```

Such output is often overwhelming.

29. To only inspect a single register bank, specify it in the argument:

```
simics> print-device-regs machine.training_card.master_bb.bank.regs
```

This shows the registers of just that bank.

30. The output radix setting affects the format of the output. Set the **`output-radix`** to 16 and list the registers again:

```
simics> output-radix 16
simics> print-device-regs machine.training_card.master_bb.bank.regs
```

The output should now use hexadecimal for the offset and value:

```
Offset  Name                              Size   Value
------------------------------------------------------
0x0000  enable                               4  0x0000
0x0004  reset                                4  0x0000
0x0010  update_display_request               4  0x0000
0x0014  update_display_status                4  0x0000
0x0018  update_display_interrupt_control     4  0x0000
0x001c  framebuffer_base_address             4  0x1000
0x0020  color_all                            4  0x0000
0x0024  dprog_base_addr                      4  0x0000
...
```

31. To look at the details of a specific register, use the **`print-device-reg-info`** command:

```
simics> print-device-reg-info
machine.training_card.master_bb.bank.regs.enable
```

32. To look for a register with a specific name or part of its name, use the **`pattern`** argument. This argument takes a file-style pattern using **`*`** to represent arbitrary sequences of characters. Look for all registers having "**`status`**" in their name, across all banks of the **`master_bb`** device:

```
simics> print-device-regs machine.training_card.master_bb pattern =
"*status*"
```

This shows fewer registers, but still quite a few:

```
Bank: pci_config
Offset  Name             Size   Value | Offset  Name              Size   Value
-------------------------------------+-------------------------------------
0x0006  status              2  0x0010 | 0x00a0  exp_root_status      4  0x0000
0x008a  exp_dev_status      2  0x0000 | 0x00aa  exp_dev_status2      2  0x0000
0x0092  exp_link_status     2  0x0041 | 0x00b2  exp_link_status2     2  0x0000
0x009a  exp_slot_status     2  0x0000 | 0x00ba  exp_slot_status2     2  0x0000

Bank: regs
Offset  Name                   Size   Value
-------------------------------------------
0x0014  update_display_status     4  0x0000
0x0028  dprog_status              4  0x0000
0x0204  button_a_status           4  0x0000
0x0208  button_b_status           4  0x0000
0x020c  button_n_status[0]        4  0x0000
...
```

It is possible to apply pattern to just a single bank to reduce the number of hits, or to use more specific patterns.

33. Look for registers including the string "**`control`**" in the **`regs`** bank:

```
simics> print-device-regs machine.training_card.master_bb.bank.regs
pattern = "*control*"
```

This shows only a few register matches:

```
Offset  Name                              Size   Value
------------------------------------------------------
0x0018  update_display_interrupt_control     4  0x0000
0x002c  dprog_control                        4  0x0000
0x0200  button_interrupt_control             4  0x0000
```

34. The simulator can also search for a certain register name in the complete simulation. This is a much more expensive operation than searching in a single device or bank, and the output is simply a list of matching registers. Search for all registers called something with "enable":

```
simics> list-device-regs pattern = "*enable*"
```

The command finds a few registers:

```
machine.mb.sb.uhci[0].bank.usb_regs.usb_interrupt_enable
machine.mb.sb.uhci[1].bank.usb_regs.usb_interrupt_enable
machine.mb.sb.uhci[2].bank.usb_regs.usb_interrupt_enable
machine.mb.sb.uhci[3].bank.usb_regs.usb_interrupt_enable
machine.mb.sb.uhci[4].bank.usb_regs.usb_interrupt_enable
machine.mb.sb.uhci[5].bank.usb_regs.usb_interrupt_enable
machine.training_card.master_bb.bank.regs.enable
```

Even in the current quite small model, this search can take some time.

Finding references to an object

The simulator uses unidirectional references from object to object to implement connections between objects. An object is thus typically not aware of all the other objects that have references to it. References from one object to another should be held in object attributes of the type "o". To debug and investigate connections between objects, the **`list-object-references`** command looks in the complete simulator configuration to find all references to a certain object – and all the objects that it refers to.

35. List references to the **`master_bb`** object in the training card:

```
simics> list-object-references machine.training_card.master_bb
```

This shows two inbound references to the object from the PCI system, and four outbound references.

36. List references to the training card **`master_bb.bank.regs`** bank:

```
simics> list-object-references
machine.training_card.master_bb.bank.regs
```

The result points at the same memory spaces as the **devs** command did previously. However, in this case, what is returned is the precise place in the map attribute of the memory maps where the object is found:

```
┌──────────────────────────────────────────┬─────────┬──────────────────────────────────────────┐
│                  Object                  │Attribute│                Reference                 │
├──────────────────────────────────────────┼─────────┼──────────────────────────────────────────┤
│machine.mb.nb.pcie_p2.downstream_port.    │map[3][1]│machine.training_card.master_bb.bank.regs │
│mem_space                                 │         │                                          │
├──────────────────────────────────────────┼─────────┼──────────────────────────────────────────┤
│machine.training_card.local_memory        │map[1][1]│machine.training_card.master_bb.bank.regs │
└──────────────────────────────────────────┴─────────┴──────────────────────────────────────────┘
```

37. If very many references are found, the command will limit the output length. The default is to show ten hits, but this can be increased:

```
simics> list-object-references machine.mb.apic_bus
```

The output is truncated:

```
┌─────────────────────────────────┬─────────┬──────────────────────────┐
│             Object              │Attribute│        Reference         │
├─────────────────────────────────┼─────────┼──────────────────────────┤
│machine.mb.cpu0.apic[0][0]       │apic_bus │machine.mb.apic_bus       │
│machine.mb.cpu0.apic[1][0]       │apic_bus │machine.mb.apic_bus       │
│machine.mb.nb.core.remap_unit[0] │apic_bus │machine.mb.apic_bus       │
│machine.mb.nb.core.remap_unit[1] │apic_bus │machine.mb.apic_bus       │
│machine.mb.nb.ioxapic.ioapic     │apic_bus │machine.mb.apic_bus       │
│machine.mb.sb.ioapic             │apic_bus │machine.mb.apic_bus       │
│machine.mb.sb.pic                │irq_dev  │machine.mb.apic_bus       │
│machine.mb.apic_bus              │queue    │machine.mb.cpu0.core[0][0]│
│machine.mb.apic_bus              │apics[0] │machine.mb.cpu0.apic[0][0]│
│machine.mb.apic_bus              │apics[1] │machine.mb.cpu0.apic[1][0]│
└─────────────────────────────────┴─────────┴──────────────────────────┘
First 10 references listed.
```

38. Raise the limit to 30:

```
simics> list-object-references machine.mb.apic_bus max-len = 30
```

This output ends without a note about the first references, so this shows all the inbound and outbound references of the object.

Inspecting the processor core state

It is often useful to inspect the state of the processor cores running code in the virtual platform, or simply get an inventory of which processor cores are present in the current simulation session.

39. To list all processors and some basic information about them, use **`list-processors`** command on the CLI, just like you did above. You have already used this in a previous lab.

```
simics> list-processors
```

40. In a configuration with very many processors, the output can get long and hard to interpret. To solve this, the **`list-processors-summary`** command combines all processors of the same class into a single line.

```
simics> list-processors-summary
```

Which should show that all eight processors are of the same type (from the same model class):

```
┌───┬────────┬─────────┬───────────────────┬─────┬────────────┐
│   │Num CPUs│CPU Class│    Description    │Freq │   Cells    │
├───┼────────┼─────────┼───────────────────┼─────┼────────────┤
│   │       2│x86QSP1  │model of QSP X86-64│2 GHz│machine.cell│
├───┼────────┼─────────┼───────────────────┼─────┼────────────┤
│Sum│       2│         │                   │     │            │
└───┴────────┴─────────┴───────────────────┴─────┴────────────┘
```

41. The **`list-processors`** command can also be used to show the current state of the processors, in particular the current instruction, instruction pointer, and time. There are several options available, check them with **`help`**.

```
simics> help list-processors
```

42. As a simple example, show the cycle count and current instruction disassembly:

```
simics> list-processors -cycles -disassemble
```

Which shows something like this (the output from the command can get very wide when more columns are added):

```
┌────────────────┬─┬─────────┬────────┬──────────────┬─────────────────────────┐
│    CPU Name    │ │CPU Class│  Freq  │    Cycles    │       Disassembly       │
├────────────────┼─┼─────────┼────────┼──────────────┼─────────────────────────┤
│machine.mb.cpu0.│*│x86QSP1  │2.00 GHz│40_000_000_000│cs:0xffffffff81fb2ad6 p: │
│core[0][0]      │ │         │        │              │0x0011b2ad6 mov          │
│                │ │         │        │              │rax,qword ptr gs:        │
│                │ │         │        │              │[0x2c800] # MWait state  │
├────────────────┼─┼─────────┼────────┼──────────────┼─────────────────────────┤
│machine.mb.cpu0.│ │x86QSP1  │2.00 GHz│40_000_000_000│cs:0xffffffff81fb2ad6 p: │
│core[1][0]      │ │         │        │              │0x0011b2ad6 mov          │
│                │ │         │        │              │rax,qword ptr gs:        │
│                │ │         │        │              │[0x2c800] # MWait state  │
└────────────────┴─┴─────────┴────────┴──────────────┴─────────────────────────┘
* = selected CPU
```

Finish the lab

43. **Quit** the simulation session.