Intel® Simics® Simulator Internals Training

Lab 01-09

System Setup and Components

1 Introduction

In the following labs, you will learn how to create, connect, instantiate and inspect components. You will also learn how to create and connect individual devices in case you don't have a matching component yet or you want to quickly prototype things or create tests setups. This also demonstrates the lower level of the simulator configuration system.

2 Using Components

In this lab, you will find and load the module that provides a selected component. You will then create the component, inspect it, connect it, and instantiate it. Along the way, you will learn the difference between creating and "newing" a component.

2.1 Start the Simulation

1. Start a new simulation session with the target for this lab in your project.

```
[./]simics[.bat] simics-user-training/001-qsp-training
```

   This creates the QSP-based training setup that you may have seen in other labs.

2.2 Finding the Module to Load

2. Check if the creation commands for the i210 NIC component are available:

```
simics> list-commands substr = i210
```

   This will come up empty. Since we want to use the creation commands that every component automatically provides, we need to load the module that provides the component as only after loading the commands become available.

3. Use the **`list-classes`** command to find the module to load:

```
simics> list-classes i210_comp -m
```

   You see that the module to load is **`i210-comp`**. Note that you need to know the class name of your component to find the module. If in doubt, use a substring of your class name.

4. Load the module:

```
simics> load-module i210-comp
```

2.3 Creating a Component

5. Check the available commands again:

```
simics> list-commands substr = i210
```

   You see several commands, the interesting ones being the **`create-i210-comp`** and the **`new-i210-comp`** commands.

6. All commands related to a certain class can be found using the **`class`** argument to **`list-commands`**:

```
simics> list-commands class = i210_comp
```

   This shows a few more commands, notable inspection commands provided as standard by the component system.

7. Try the **new-** command:

```
simics> new-i210-comp name = nic0
```

The simulator will report an error and not create any component or objects, because the **pci_bus** attribute is not set in the **mac** device.

8. Check that nothing was created:

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

9. The required attribute that the **new-** command complained about is set when the component's PCI connector is connected to another component's PCI connector.

Components can only be connected when they exist. In this case, when the connector has to be connected to satisfy the requirements of an object in the component, the solution is to create the component but not instantiate it. This is the standard flow when using components, supported by the **create-** commands.

New- commands only make sense for components that can be created stand-alone and then hot-plugged into another component.

10. Use the **create-** command to create a i210 component:

```
simics> create-i210-comp name = nic0
```

The simulator reports that it created a non-instantiated component.

2.4 Connecting and instantiating components

11. Check the components we have right now.

```
simics> list-components
```

You see the components that were created from the training machine setup, plus the **nic0** component just created. Note that the latter is marked as not instantiated.

12. Check the **info** of the component.

```
simics> nic0.info
```

You see that it has two connectors, one for PCI and one for Ethernet. The slots listed are just the connectors; once the component is instantiated, more slots will be added for the device model objects it creates.

13. List the objects currently inside the component:

```
simics> list-objects namespace = nic0
```

There are two objects, both connectors. The device objects are still in the pre-instantiation state and not part of the simulator configuration (yet). The only (official) way to interact with a pre-instantiated component is through its connectors and attributes.

14. Try to instantiate the component, to see what happens to a partially instantiated component. This is known not to work, but test it anyways:

```
simics> instantiate-components nic0
```

    You see a similar message to earlier when you tried to use the **new-** command. Note the last part of the message: it says, "*Deleting all partially created objects.*" The half-instantiated component will not be rolled back into the pre-instantiated state, but instead deleted in its entirety.

15. Verify the list of existing components.

```
simics> list-components
```

    The **nic0** component is no longer listed!

16. Create a pre-instantiated component again.

```
simics> create-i210-comp name = nic0
```

17. Check the status of the component.

```
simics> nic0.status
```

    Note that the "Connections" section in the status output is empty.

18. Connect the component to the QSP.

```
simics> connect nic0.pci_bus machine.mb.nb.pcie_slot[0]
```

19. Repeat the status check.

```
simics> nic0.status
```

    The connection to the PCIe slot is now shown since it is connected. Note that the Ethernet connector is not connected - and it does not have to be, since it can be hot-plugged later.

20. Finally, instantiate the component.

```
simics> instantiate-components nic0
```

21. List the components again:

```
simics> list-components
```

    The **nic0** component is listed, without the previous "not instantiated" marker.

22. List the objects underneath the **nic0** component.

```
simics> list-objects namespace = nic0
```

    You see that now it does not only have the connector objects but also the device objects, because all objects have moved from their pre-conf-object state into actual simulator objects.

23. Check the **info** of the component.

```
simics> nic0.info
```

This shows a longer list of slots than before, including all the objects created when the component was instantiated. It is the same as the list of objects inside the component.

24. This concludes this exercise. **Quit** the simulation.

```
simics> quit
```

3 Working with Pre-Conf-Objects

The previous section used an Intel Simics simulator component, which was created and connected using the features of the simulator component system. The component system is designed to encapsulate subsystems and components are classes in their own right that have to be coded. There are other cases when users need to create a set of objects, such as unit tests and ad-hoc configurations. In this case, writing new components is clearly overkill.

Intel Simics simulator **pre-conf-objects** provide a way to create sets of objects together. This lab shows how to create a setup using pre-conf-objects directly. Behind the scenes, components also work with pre-conf-objects, and the **`instantiate`** operation basically goes from pre-conf-objects to real objects.

In contrast to other labs, we need to be a bit more verbose here when describing the scenario for the lab to make sense.

3.1 Creating a setup script

You typically want to use a script when "manually" creating a simulation configuration. Writing multiple lines at the interactive command line is just cumbersome, especially if you make a mistake and need to start over. Single lines can be tested interactively, of course, but ultimately, they should go into a script file. This lab goes straight to a script file, but each line can in principle be pasted in and executed on its own in Python in the simulator CLI.

Assume you want to create a minimal setup to develop or run some bare metal driver/firmware for a new device you have modeled. This would constitute a system-level test. Now for this you need something to run your software. You could create a full-blown component, but let's further assume that this is another team's responsibility, and you need something fast and only for internal testing. So, since your device seems to be a rather trivial one 😉, all that is needed is a processor, some memory, and the unit under test.

The unit under test in this setup is the shared console device from the ribit setup.

The lab will walk through how the script is built line by line and explain what each line does. You can also execute each line directly in the CLI in parallel.

1. In your project, **create a file** named **`09_mini_system.py`** (you can put it at the top level of the project or in any subdirectory you want; the rest of the lab assumes it is at the top level) and open it in a text editor of your choice.
2. Add the following line to the script.

   ```
   ns = pre_conf_object('mini_sys','namespace')
   ```

   This creates a pre-conf-object of class **`namespace`** named **`mini_sys`** and stores a reference to this object in the Python variable **`ns`**. The namespace pre-conf-object class has the nice property that you can add new objects to it, and all those objects

will be created when the namespace is created, simplifying the call to **SIM_add_configuration**.

In addition, it will name all its sub-objects after their Python variable names, so we do not need to worry about explicitly passing name strings around. As you can see, the name of the Python variable is **ns**, while the Intel Simics simulator object name is going to be **mini_sys** (the Intel Simics simulator object does not exist yet, just a "template" for it in the form of a pre-conf-object). As you will see later, using the **namespace** class will avoid this dual naming for all further objects.

It is a common "trick" to push all objects that you want to create into a namespace to make naming and later instantiation (as we will see) simpler.

3. Add the next line.

```
ns.mem=pre_conf_object('memory-space')
```

This creates a pre-conf-object of class **memory-space** and pushes it into the (new) member variable **mem** of the previously created namespace pre-conf-object.

The name of the object in your script is **ns.mem**. There is no need to also provide an explicit name string to the **pre_conf_object()** declaration.

But what will the ultimate object name be? Since the pre-conf-object referenced by the Python variable **ns** was called **mini_sys** the **memory-space** will be called **mini_sys.mem** in the simulation. This shows that it would have been much smarter to call the Python variable **mini_sys** instead of **ns** to have everything aligned, but we intentionally did not do that to show that in general the name of the Python variable in the code and the eventual Intel Simics simulator object are entirely separate.

4. The next line:

```
ns.ram=pre_conf_object('ram', self_allocated_image_size = 0x4000, image = None)
```

Again, we create an unnamed pre-conf-object. This time of class **ram** and we push it into a member variable of the same name in our namespace object. Note how we set attributes to make our **ram** object allocate its own internal image object, saving a step or two in the script code.

5. And another line:

```
ns.hart=pre_conf_object('riscv-rv64', freq_mhz=10, physical_memory=ns.mem)
```

This adds a pre-conf-object of class **riscv-rv64** into the member **hart** of the namespace. Again, note how we set some attributes and note that one of them is setting the connect **physical_memory** to point to the **memory-space** we created earlier. It is important to understand that this works with the Python variable of the pre-conf-objects, but that once the objects are created for real, the real simulation objects will be used.

6. All objects in a subsystem should have their queue set to the appropriate clock or processor. In this case, the **ns.mem** was created before the **ns.hart** processor, so we need to set the queue attribute to point at **ns.hart**.

```
ns.mem.attr.queue = ns.hart
```

This cannot be resolved by simply creating **ns.hart** first and referring to it when creating **ns.mem**. **ns.hart** need to refer to **ns.mem** to set its **physical_memory** attribute. Cyclical dependencies between multiple objects are very common, and pre-conf-objects solve that neatly by operating before actual objects are created.

This is particularly important for required attributes.

Using the "***preobj*.attr.*X***" way to reference attributes is recommended; it makes it clear that the operation assigns an attribute. It usually works to assign "***preobj.X***"... unless there is a collision between attribute names and object names.

7. Set the namespace and ram **queue** attributes as well:

```
ns.attr.queue = ns.hart
ns.ram.attr.queue = ns.hart
```

8. Add yet another line to create the shared serial port:

```
ns.uut=pre_conf_object('i_multi_writer_output', queue=ns.hart)
```

All objects should have their queue attribute set, and since this pre-conf-object is created after **ns.hart**, it can be done in the creation line.

9. Instantiate or create all the actual Intel Simics simulator configuration objects by adding the following line:

```
SIM_add_configuration([ns],None)
```

We only need to pass the namespace pre-conf-object to the **SIM_add_configuration** call.

If we had created all the pre-conf-objects at the top-level (like the namespace right now), we would need to list all the objects in the argument to the call. That can be really annoying when adding new objects to an existing setup, as adding an object requires you to also add it to the list argument.

Hence the namespace technique is highly recommended. It also has the benefit of neatly grouping related objects together.

After this line has been executed, we will have fully-fledged configuration objects.

10. After the object instantiation, the Python variable **ns** still points to the pre-conf-object. It cannot be used to refer to the actual simulation objects. Since it is a handy name, change its value to the actual object:

```
ns = SIM_get_object(ns.__object_name__)
```

Using **__object_name__** allows us to avoid using the name explicitly. So, if we ever decide that **mini_sys** is a bad name, we only need to change it in a single line.

ns is now a Python reference to a simulator configuration object. Thanks to the simulator Python wrapping, all members of **ns** are also configuration objects.

11. Do a post-instantiation object attribute change to set the memory map:

```
ns.mem.attr.map = [[0, ns.ram, 0, 0, 0x4000],
                   [0x8000, ns.uut.bank.regs, 0, 0, 1],]
```

This maps the **ram** object (remember, **ns.ram** is a standard simulator configuration object at this time) and the register bank of the unit under test into the memory space.

Setting the map should normally be done in the pre-conf stage, but we wanted to demonstrate how you go from pre-conf-objects to actual simulator configuration objects.

12. **Save** the file.

3.2 Running the setup script and inspecting the result

13. Start a new simulator session with the custom script:

```
[./]simics[.bat] 09_mini_system.py
```

If all lines are exactly as shown above, the script should just work. If not, try to make sense of the error messages and fix them.

If later steps do not show the expected response, double-check that every line is present in the script file.

14. Check the set of existing components.

```
simics> list-components
```

There should be no components. This is expected and meant to underline the fact that the Python script has done the job of a component: create, configure, and connect a set of device objects.

15. Check the objects in the configuration:

```
simics> list-objects
```

Find the **mini_sys** namespace object and its sub-objects. There will also be some port objects that were created when the device objects were created, such as the **extensions** sub-object of the **hart** object or the **own_image** sub-object of the **ram** object.

16. Check the memory map:

```
simics> mini_sys.mem.map
```

As expected, we have the small RAM mapped at base 0x0 and the unit under test at offset 0x8000.

17. Verify queue settings.

```
simics> mini_sys.uut->queue
```

This should show that the unit under test's event queue is the RISC-V processor of the mini system.

3.3 Connecting to a Component

The mini system is up and we can now run (very simple) software on it. Given that the unit under test is a serial device that wants to send characters, we should connect an endpoint that can show what is printed. Creating a console object manually could be a bit painful, instead create a standard console component.

18. Check the info for the unit under test:

```
simics> mini_sys.uut.info
```

    It reports not to be connected to anything.

19. Load the module for our component (remember that we need to do that to get the new/create commands)

```
simics> load-module console-components
```

20. Create an instantiated text console component. The component can be hot plugged, so we can use the "new" command.

```
simics> new-txt-console-comp name = serconsole
```

    Now that we have the component, how do we connect it? The mini system is not a component, and it has no connectors to connect to other components. Just like we mimicked the behavior of a component when creating the devices, we will now manually do what a serial connector would do.

    To do that, one needs to know how such a connector works. For serial connectors, the sender (serial device, typically) has a **`console`** attribute that must point to the receiver (console). The receiver has an attribute **`device`** that points back to the sender. The reality is a bit more complicated, but this information suffices for this lab.

    When two components are connected over a serial connector, there is code behind the scenes that makes sure that the two device objects involved are set up correctly.

21. Find the configuration attributes of the unit under test:

```
simics> list-attributes mini_sys.uut
```

22. Take a closer look at the **`console`** attribute:

```
simics> help mini_sys.uut->console
```

    The text shows the required interface of the counterpart, which happens to be **`serial_device`**.

23. List the interfaces of the **`serconsole.con`** object:

```
simics> list-interfaces serconsole.con
```

    This is the actual console object that does the job of showing characters at runtime. The component is just a wrapper around the actual console, used to create and connect it.

    Note that the object implements the interface that the unit under test requires.

24. Set the **console** attribute of the unit under test to point at the console object:

```
simics> mini_sys.uut->console = serconsole.con
```

25. Set the device attribute of the serial console device.

```
simics> serconsole.con->device = mini_sys.uut
```

26. Check the info from the unit under test again.

```
simics> mini_sys.uut.info
```

    Now it shows it is connected to **serconsole.con**.

27. Inspect the window of the serial console.

    Note that it still says, "**not connected**". The reason for this is that the window title is set by the connector code in the component. Since we bypassed the component connector system and established the connection manually between the device objects, the Window title was not updated.

    The console is still connected, though. This inconsistency is a typical side-effect of manually stitching things together. In some cases, cosmetic or metadata information does not get updated as they require additional actions.

28. If the "not connected" is annoying, fix it:

```
simics> serconsole.con->window_title = "Connected to unit under test"
```

29. Double-check the connection with the **info** command of the serial console object:

```
simics> serconsole.con.info
```

    Here, you can clearly see the connection is established.

3.4 Running Test "Software"

Now we could load software into RAM and test the device from software. Given how trivial our unit under test is, we do not have any fancy test software. Instead, we'll just make the processor execute a single write towards the unit under test so that we see a character appear.

30. Disassemble the current instruction

```
simics> da
```

    You see that we would currently execute an illegal instruction if we would start the simulation.

31. Write a single valid instruction at the current program counter value into memory:

```
simics> set %pc 0x00c58023 4
```

32. Disassemble again.

```
simics> da
```

    You see that we will now do a store-byte (SB) instruction to the address stored in register **a1** writing the content of register **a2**.

33. Set register **a1** to contain the address of the unit under tests character register.

```
simics> %a1 = 0x8000
```

34. Set register **a2** to contain the value 0x21, which corresponds to the '!' character.

```
simics> %a2 = 0x21
```

35. Single step the current instruction.

```
simics> si
```

    You see a message coming from our unit under test complaining about a missing processor ID. This is expected. We did not bother to add the translator needed for this to the mini system. There should still be an exclamation mark (**!**) printed on the console.

    Thus, the mini system is working fine. A more complicated test could be created based on this setup, typically using Python scripting to exercise and check the device behavior.

36. This concludes this part of the lab. **Exit** the simulation session.

```
simics> exit
```

3.5 Scaling Pre-Conf-Objects

The pre-conf-object system can handle very large sets of objects and create them quickly. In this lab you will test that for yourself.

37. In your project, **create a file** named **09_big_system.py** (you can put it at the top level of the project or in any subdirectory you want; the rest of the lab assumes it is at the top level) and open it in a text editor of your choice.

38. Paste the following code into the file, all at once:

```
def buildmany(name, a, b):
    l=[]
    # Build a subsystems with a processor each
    for i in range(a):
        ns = pre_conf_object(f"{name}[{i}]", 'namespace')
        ha = pre_conf_object(f"{name}[{i}].hart",'riscv-rv64', freq_mhz=1000 + 100 * i)
        me = pre_conf_object(f"{name}[{i}].mem", 'memory-space', queue=ha, map=[])
        ha.attr.physical_memory=me
        l+=[ns,ha,me]
        ## b devices for every subsystem
        for j in range(b):
            de = pre_conf_object(f"{name}[{i}].dev[{j}]",'NS16550', queue=ha)
            me.attr.map += [[0x1000+j*0x10, de.bank.regs, 0, 0, 11],]
            l+=[de]
    SIM_add_configuration(l,None)
```

    Note that this code explicitly accumulates the created objects into a list that is sent to **SIM_add_configuration**. This is due to the use an **index-map** object that does not support the automatic gathering of sub-objects.

39. Start a new simulator session with the custom script:

```
[./]simics[.bat] 09_big_system.py
```

This will start a new session, but no objects will have been created as the file only defines a creation function.

40. Count the number of objects in the current session using the simulator API from Python:

```
simics> @len(list(SIM_object_iterator(None)))
```

This way shows all objects in the simulation. The **list-objects** command-line command is a bit selective and will hide certain objects like the bank and port namespaces on all devices. Thus, the number of objects returned from it might be different, hence we will use this somewhat complex Python construct to count the objects now and in later steps.

41. Create a very small system:

```
simics> @buildmany("a",1,1)
```

42. Look at the configuration:

```
simics> list-objects -tree
```

The subsystem "**a[0]**" has been created with a few objects underneath it.

43. Show all the port and bank objects as well:

```
simics> list-objects -tree namespace = a[0] -show-port-objects
```

Quite a few port objects have been created!

44. Count the number of objects in the current session again:

```
simics> @len(list(SIM_object_iterator(None)))
```

The number of objects will have increased by something like 50!

45. Create more objects. Say 10 top-level subsystems, each with 20 devices inside:

```
simics> @buildmany("b",10,40)
```

This should finish quickly.

46. Count the number of objects in the current session again:

```
simics> @len(list(SIM_object_iterator(None)))
```

Note how the simulator configuration grew to a few thousand objects, but it did not take all that much time.

The created setup is not entirely trivial. It does contain some real processor objects, for example:

47. List all the processors:

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

48. Show the contents of the simulation, this is a long list:

```
simics> list-objects -tree -show-port-objects
```

49. Before building a really big configuration, check the memory usage of the simulator:

```
simics> enable-probes
simics> probes.read probe-kind = sim.process.memory.virtual
```

50. Create more objects. Say 25 top-level subsystems, each with 250 devices inside:

```
simics> @buildmany("c",25,250)
```

    This might take a short while to complete.

51. Count the number of objects in the current session again:

```
simics> @len(list(SIM_object_iterator(None)))
```

    The operation added tens of thousands of simulation objects.

52. Check the memory usage:

```
simics> probes.read probe-kind = sim.process.memory.virtual
```

    The memory usage will go up by a few hundred megabytes. Note that this is just the cost of storing the state for the new objects. All their code and static data had already been loaded when the first object of each class was first created.

53. This concludes the lab.

```
simics> quit
```

54. Optionally, you could try giving the create functions very large arguments to see how the simulator core handles very large numbers of objects.

```
simics> @buildmany("d",250,2000)
```

    Something like that will take a while to build, but it will work. The number of objects in this example is some 15 to 20 times larger than the largest seen in practice.