Intel® Simics® Simulator Internals Training

Lab 01-08

Reacting to Model Activity and Inspecting Models

1 Introduction

In the following labs, you will learn how to use model interfaces to register callbacks, trigger custom notifiers from within these callbacks and have script branches react to it and how to use probes to inspect model activity.

2 Using Instrumentation and Notifiers

Some interfaces allow you to register callbacks to implement more complex reactions. The exact set of interfaces you can use in such a way always depends on the target used, the packages installed, and their versions. When a callback has done their job and found something interesting, it might want to inform a script branch so that the branch can then take actions (like stopping the simulation, saving a checkpoint, or whatever). This lab exercise will show one way of doing that.

The scenario we have for this lab is the following:

Assume, for some reason, we consider writing the same value twice to any control register in our processor to be an error that we want to catch. We do not know if such writes will happen at all, and if they do, which register will be written and with which value. This is difficult to accomplish using scripting with breakpoints.

Further, when such a write happens, you want to take a checkpoint. This is accomplished by having a script branch waiting for the condition to happen and then using the **`write-configuration`** command.

To inform the script branch that something has happened we need to have the script branch wait for something that will only be triggered from the instrumentation callback. For such cases, using a custom notifier is a good solution.

2.1 Creating and Using a Custom Notifier

Notifiers must be associated with a class when created and with an object (of the class) when triggered. You can either create a custom class for this or reuse a class that you have in your simulation. In our case, we reuse our top-level component class **`internals_training_system_comp`** and we will trigger notifiers on the **`ribit`** object. This choice is arbitrary. Any class and object would have done, but it makes sense to use the training system top-level object, as we want to inform the script branch something "bad" happened when software was executing in the system.

1. We will do a fair amount of scripting and things can easily go wrong there, and to avoid having to repeat a ton of steps all the time, we rather use a script.

   Create a file named **`08_my_notifier.py`** in the Intel Simics simulator project and open it in an editor.

2. Add the following lines to the file:

   ```
   my_notifier=SIM_notifier_type("ribit-double-cr-write")

   SIM_register_notifier(SIM_object_class(conf.ribit), my_notifier,
   "Triggered when a processor write the same value twice to a CR.")
   ```

   The first line gets the ID of the notifier type with the name **`ribit-double-cr-write`**. If the notifier type had not been seen before, it will be created. Any subsequent call with the same argument will return the same ID.

The second line registers the notifier with the class of the top-level object named **ribit** and adds some documentation.

3. Create a file named **08-double-cr-writes.simics** in the project and open it in an editor.

4. Add the following lines to the script.

```
load-target simics-internals-training/ribit-hello-world system:num_subsystems=1
run-script 08_my_notifier.py
script-branch "Checkpoint creator" {
  while TRUE {
    bp.notifier.wait-for ribit "ribit-double-cr-write"
    if (python "SIM_simics_is_running()") { stop }
    echo "This is where a checkpoint would be saved"
  }
}
```

   This will create a **ribit** system with only one subsystem, using the hello-world program.

   It runs the Python script we created above.

   Finally, it creates a script branch that waits for all occurrences of the notifier (from the **ribit** system object, to be as precise as possible). When the notifier is notified, the scripts tell us that it would create a checkpoint (we don't really do this in this lab). Since checkpoints can only be saved when the simulation is stopped, it stops the simulation in case it was running.

5. **Start** a new simulation session with your script:

```
[./]simics[.bat] 08-double-cr-writes.simics
```

6. List the notifiers in our session that contain the string "**ribit**" in their name.

```
simics> list-notifiers substr = ribit
```

   You see the newly created notifier and that it is available on the the **ribit** object.

7. List the script branches in your simulation session.

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

   You'll see your script branch telling you that it is waiting for our notifier on object **ribit**.

8. Trigger the notifier manually: the object and the notifier type must be specified.

```
simics> @SIM_notify(conf.ribit, SIM_notifier_type("ribit-double-cr-
write"))
```

   This triggers our notifier on the **ribit** object. You will immediately see the message from the script branch. Repeating the above command will always give you the response from the script branch. Note that the API allows any code to trigger any notifier on any object – it does not have to be code inside the object itself.

   Note: In a script you would rather call **SIM_notifier_type** only once, store the result in a variable and then use that one instead of always calling **SIM_notifier_type**.

9. Try to trigger the notification on another object:

   ```
   simics> @SIM_notify(conf.ribit.counter, SIM_notifier_type("ribit-
   double-cr-write"))
   ```

   This does not cause the script branch to detect the notifier, since it is coming from the wrong object. Notifier subscriptions are tied to an object.

10. **Quit** the simulation.

    ```
    simics> quit
    ```

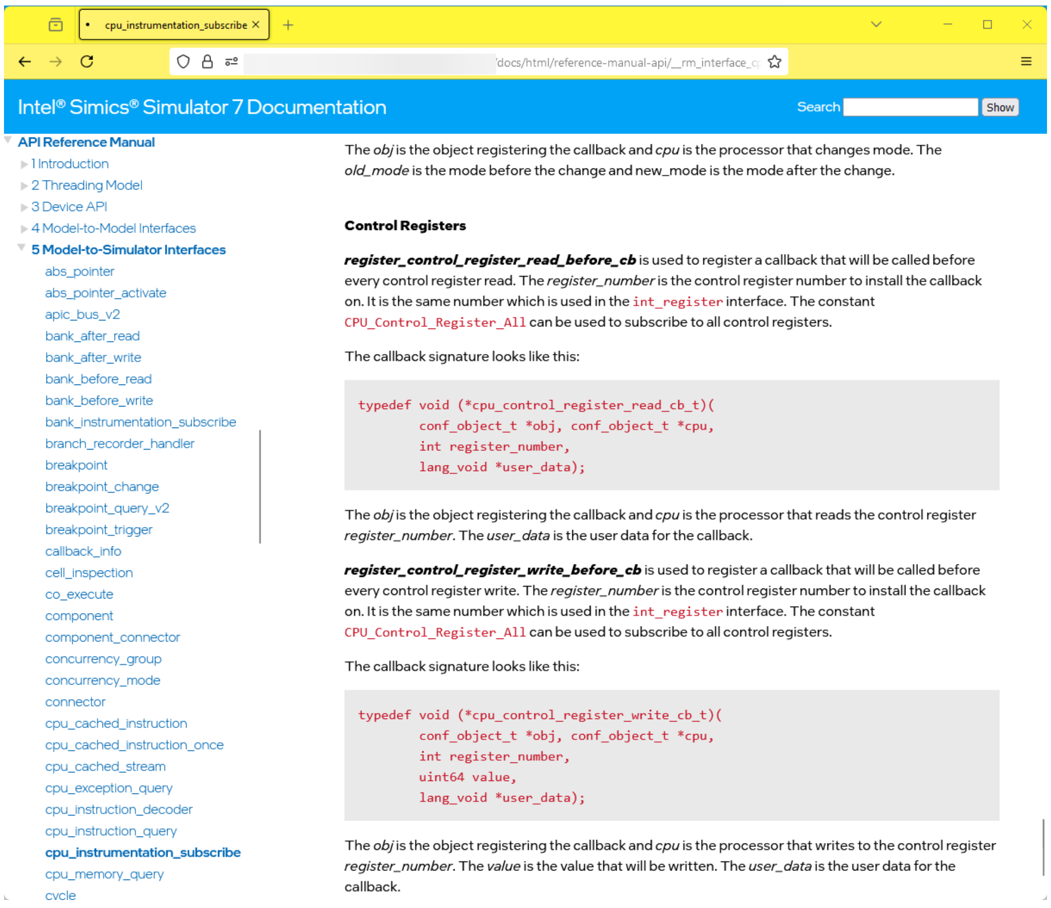
2.2 Creating a Instrumentation Callback

Having tested the notifier creation, triggering, and scripting, the next step is to create the actual code that will watch the execution and notify the notifier when the double-write happens. This will be done using the instrumentation API, creating a custom instrumentation callback.

To find the available callbacks, you have to inspect the object from which you expect a callback. Check its interfaces and read the documentation for the interfaces. In this example, assume we have looked at the processor and found that it has the **`cpu_instrumentation_subscribe`** interface. This interface offers a **`register_control_register_write_before_cb`** function that allows us to register a callback that is called whenever a control register is to be written.

11. Open the Intel Simics simulator documentation and search for **`cpu_instrumentation_subscribe`**. Open the API reference manual section on the

interface, and scroll down to find the documentation of the control register callbacks.



The *obj* is the object registering the callback and *cpu* is the processor that changes mode. The *old_mode* is the mode before the change and new_mode is the mode after the change.

Control Registers

register_control_register_read_before_cb is used to register a callback that will be called before every control register read. The *register_number* is the control register number to install the callback on. It is the same number which is used in the `int_register` interface. The constant `CPU_Control_Register_All` can be used to subscribe to all control registers.

The callback signature looks like this:

```
typedef void (*cpu_control_register_read_cb_t)(
        conf_object_t *obj, conf_object_t *cpu,
        int register_number,
        lang_void *user_data);
```

The *obj* is the object registering the callback and *cpu* is the processor that reads the control register *register_number*. The *user_data* is the user data for the callback.

register_control_register_write_before_cb is used to register a callback that will be called before every control register write. The *register_number* is the control register number to install the callback on. It is the same number which is used in the `int_register` interface. The constant `CPU_Control_Register_All` can be used to subscribe to all control registers.

The callback signature looks like this:

```
typedef void (*cpu_control_register_write_cb_t)(
        conf_object_t *obj, conf_object_t *cpu,
        int register_number,
        uint64 value,
        lang_void *user_data);
```

The *obj* is the object registering the callback and *cpu* is the processor that writes to the control register *register_number*. The *value* is the value that will be written. The *user_data* is the user data for the callback.

12. **Create a file** named **`08_cr_write_callback.py`** in your project.

13. Add the following code to it.

```
def ctrl_reg_before_cb(obj, cpu, num, value, user_data):
  notifier_obj, notifier_type, value_map = user_data
  if value_map.get(num, None)!=None:
    if value == value_map[num]:
      SIM_log_info(1, notifier_obj, 0,
                   f'CR number {num} got double write with value {hex(value)}')
      SIM_notify(notifier_obj, notifier_type)
  value_map[num]=value

cb_arg=[conf.ribit, SIM_notifier_type("ribit-double-cr-write"), dict()]
instrumentation_interface = conf.ribit.unit[0].hart.iface.cpu_instrumentation_subscribe
instrumentation_interface.register_control_register_write_before_cb(
    None,
    CPU_Control_Register_All,
    ctrl_reg_before_cb,
    cb_arg)
```

This code first defines the callback function **`ctrl_reg_before_cb`**. The signature is taken from the definition of the **`cpu_instrumentation_subscribe`** interface.

Argument **`num`** is the control register number that gets the write with the value **`value`**.

The **`user_data`** is passed from the registration of the callback. It provides a reference to the object that shall notify the **`notifier_type`** and the user data also provides a value map. If this is not the first write to the register (i.e., there is a value in the map) we check if the value is the same. If it is, we log and notify. The value written is stored.

After the callback, we construct a list **`cb_arg`** that contains a reference to the top-level **`ribit`** object, the notifier type for our notifier and an empty Python **`dict`**. This will be used to provide the **`user_data`** argument to the callback.

Finally, the callback is registered on the processor, by retrieving the interface from the processor object and then calling the registration function in the interface. This registration is specific to this processor object – if there were other processors in the system, they would not generate the callback.

2.3 Running the Simulation with the Callback Installed

14. **Start** the simulation with the custom script:

```
[./]simics[.bat] 08-double-cr-writes.simics
```

15. Run the Python file you just created:

```
simics> run-script 08_cr_write_callback.py
```

This will install the callback.

16. The instrumentation system provides a neat way to see the installed callbacks:

```
simics> list-instrumentation-callbacks
```

It will show that there is a callback function installed on the control-register callback in the instrumentation API. The **`provider`** is the object that will call the instrumentation callback. The **`callback`** is the function getting called. The **`data`** is the user data.

17. To verify that the instrumentation callback and, eventually, the notifier triggers when the processor write control registers, trace all control register writes using the breakpoint system:

```
simics> bp.control_register.trace -all -w
```

18. Now run the simulation forward.

```
simics> run
```

The simulation will stop (stopped by the script branch acting after seeing the custom notifier). The log from the double-write detection code in the callback appears before the trace shows the register change, something like this:

```
[bp.control_register trace] [trace:2] ribit.unit[0].hart mstatus <- 0xa00002008
[bp.control_register trace] [trace:2] ribit.unit[0].hart mie <- 0x880
[ribit info] CR number 168 got double write with value 0x880
[bp.control_register trace] [trace:2] ribit.unit[0].hart mie <- 0x880
This is where a checkpoint would be saved
```

The reason is that the callback is a pre-write callback, i.e., it is invoked before the instruction is executed. It correctly detected the double write of the same value to the **`mie`** register.

19. Run the simulation forward again.

```
simics> run
```

We catch another double write of the value **`0`** to the **`mscratch`** register.

20. Run forward a final time.

```
simics> run
```

No further double write will be detected. If you like, scroll up and you'll see that this is correct.

So, our double write detector works and notified the script branch correctly!

21. This concludes the lab exercise. **Quit** the simulation session.

```
simics> quit
```

3 Using Probes to Observe Model Activity

In this lab, you will learn how find interesting probes, how they can help observing what a model does, how to create a hierarchy of custom probes to generate more helpful data while observing the simulation.

3.1 Start the Simulation

22. Start a new simulation session, using the target for this exercise:

    ```
    [./]simics[.bat] simics-internals-training/08-probes
    ```

23. Run the simulation until the software has started and synchronized across the processors:

    ```
    simics> bp.notifier.run-until name = i-synchronizer-release
    ```

    The consoles will not show any output yet.

24. Check that something was indeed executed:

    ```
    simics> list-processors -cycles -steps
    ```

    Note that the processors have run hundreds of thousands of cycles and steps.

3.2 Enabling and Finding Available Probes

25. Enable the probes framework so that probes start capturing information.

    ```
    simics> enable-probes
    ```

    Since enabled probes might have an impact on the simulation performance, the simulator does not enable the probe system by default.

26. Run the simulation for 5 seconds.

    ```
    simics> run 5 s
    ```

27. To make sense of the probe values that we are later inspecting, let's first understand the target SW better. Look at the shared serial console.

    You will see that both units write - in an interleaved manner - a message that gets gradually more colorful. They do that by writing only single characters. Character coloring depends on which output register it is printed to: either one that does the per-unit coloring as seen in previous labs, or a register that does not color the output.

    The program is writing the output string every time it prints; but it uses different output registers for varying parts of the string.

    However, colored characters need more characters to be sent from the shared output device to the console object. Thus, the relationship between number of characters written to the shared output device from software and the number of characters it sends to the console will vary over the program run.

28. Now let's see what probes we have and what they can tell us about the simulation behavior. List all available probe kinds.

```
simics> probes.list-kinds
```

You will see a long list that will not even show all probe kinds. Note the "X/Y" (with X and Y being numbers) in the lower left corner telling you that only the first X probe kinds out of a total of Y are shown. Something like this:

```
…
│# 40/124 │                                 │                          │   │                                          │
│Sum (all)│                                 │                          │211│                                          │
│Sum shown│                                 │                          │ 68│                                          │
└─────────┴─────────────────────────────────┴──────────────────────────┴───┴──────────────────────────────────────────┘
```

The exact set and number of available probes depend on the version of the base package, installed packages, the precise contents of the simulator configuration, etc.

The list is long, but we know that we want to inspect probes that tell us something about devices. Such probe kinds are usually named with a "**`dev.`**" prefix. Note: the probe kind used for a specific probe is selected by the programmer implementing it. The naming is not enforceable by the simulator core, so you might see other naming schemes as well.

29. List probe kinds that have "**`dev.`**" in their name.

```
simics> probes.list-kinds substr = dev.
```

You will see at least three probe kinds. One for I/O accesses to a device, one for memory accesses per ID, and one for bytes written to the console per ID. Note that there are multiple objects implementing each probe kind, as signified by the **`Num`** column.

3.3 Reading Probe Values

30. Inspect the current value of the per-id probes offered by the ports of the shared serial device:

```
simics> probes.read object = ribit.shared_out -recursive substr =
per_id
```

There are eight port objects each with a per-ID-probe for inbound memory accesses and one for characters sent to the serial console, as well as one of each on the top-level object.

Note that the device-level probes were automatically added to the device as the framework found that sub-objects had a probe of that kind. For an array of sub-probes, only index 0 will be forwarded to the device level. We will double-check that soon.

Also note that probes are named as ***`object:probe-kind`***.

31. It is possible to read a single probe kind:

```
simics> probes.read object = ribit.shared_out -recursive probe-kind =
dev.mem_access_count.per_id
```

32. Do a colored access to the console with the color of unit 0.

```
simics> ribit.unit[0].core_mem.write 0x30000000 0x21 1
```

You will see a colored exclamation mark (!) in the shared console.

33. Inspect the memory access count probes again.

```
simics> probes.read object = ribit.shared_out -recursive probe-kind =
dev.mem_access_count.per_id
```

The count for ID 0 increased by one, and so did the value at the device level. Apparently, the device level probe value just follows index 0 of the sub-object array and is not the sum or average of the sub-object probes.

34. Inspect all the probes on the shared device again:

```
simics> probes.read object = ribit.shared_out -recursive substr =
per_id
```

Note that the memory accesses for ID 0 increased by one and note how a single memory access causes many more characters to be sent to the console.

35. Perform an access to the console, targeting the non-colored register:

```
simics> ribit.unit[0].core_mem.write 0x30000004 0x21 1
```

36. Inspect all the probes on the shared device again:

```
simics> probes.read object = ribit.shared_out -recursive substr =
per_id
```

This access caused just one character to be sent to the console, incrementing both counters by one.

37. Look at the registers of the shared serial device:

```
simics> print-device-regs ribit.shared_out
```

Note that there are some addresses that are marked as reserved.

38. Perform an access that hits one of the reserved bytes:

```
simics> ribit.unit[0].core_mem.write 0x30000001 0x21 1
```

This should print a `spec-viol` warning.

39. Inspect all the probes on the shared device again:

```
simics> probes.read object = ribit.shared_out -recursive substr =
per_id
```

Note that the memory access count was incremented, but the output was not. It counts all accesses to the device, regardless of which register it hits.

40. Perform a large access that includes the colored register:

```
simics> ribit.unit[0].core_mem.write 0x30000000 0x30303030 4
```

This results in a `0` being written to the console.

41. Inspect all the probes on the shared device again:

```
simics> probes.read object = ribit.shared_out -recursive substr =
per_id
```

Note that the memory access count only went up by one: it counts discrete accesses, not bytes written. It is important to check the definition of a probe in order to understand the produced values correctly.

42. **Quit** the simulation.

```
simics> quit
```

3.4 Using Attribute Probes

The previous lab found and read probes that were provided explicitly by a well-written model with specific information exposed over probes. This case is not all that common.

The probes framework can also be used to monitor all device state that is exposed in attributes, irrespective of whether the modeler added an actual probe for the information.

Let's assume we want to understand the data flow in the system: how many bytes are going from the processor to the shared output device, and how many bytes are going from the shared output device to the serial console, and the relationship between them.

We have probes (per ID) for memory accesses from the processors to the shared output device and for the data going from the device to the console. What is missing is the total number of bytes for each. Maybe there is an attribute in the device for this? Then we can wrap it into a probe so that we can monitor it.

43. Start the simulation again.

```
[./]simics[.bat] simics-internals-training/08-probes
```

44. And run up to the sync point.

```
simics> bp.notifier.run-until name = i-synchronizer-release
```

45. And once again enable probes.

```
simics> enable-probes
```

46. List the attributes of the shared output device. Feeling particularly lucky today, you immediately filter it down to attributes that have "bytes" in their name.

```
simics> list-attributes object = ribit.shared_out substr = bytes
```

This finds an attribute storing the total bytes written to the console. It is called **`total_console_bytes_written`**. Interesting!

47. Inspect the help of the attribute.

```
simics> help ribit.shared_out->total_console_bytes_written
```

The documentation confirms that this is what we are looking for. The information is not qualified by "per ID".

48. Create a new attribute probe, wrapping this attribute:

```
simics> probes.new-attribute-probe class = i_multi_writer_output
attribute-name = total_console_bytes_written
name="dev.con_bytes_written" display-name = "Total console bytes
written"
```

49. List the probe kinds again.

    ```
    simics> probes.list-kinds substr = dev.
    ```

    This should show the new probe kind **`dev.con_bytes_written`** for the wrapped attribute. This is a probe kind, and every instance of the **`i_multi_write_output`** class offers a probe of this kind. If there were more such objects, each would have its own probe.

50. Read out all the probes that are now available on the shared output device:

    ```
    simics> probes.read substr = bytes_written object = ribit.shared_out -
    recursive
    ```

    This shows the previously seen per-ID probes provided by the device class, as well as the new attribute probe.

3.5 Creating and Reading Derived Probes

Now we have per-ID and total numbers for the data from the device to the console. But we still need the total number of memory accesses across all IDs. This can be solved by defining a derived probe that summarizes the values of multiple existing probes.

Note: Devices written in DML also offer an **`io_access_count`** probe for each bank that shows all memory accesses to the bank, but for educational purposes we ignore that here and create a derived probe.

51. Create a new sum probe that sums up the per-ID probes for memory acceses:

    ```
    simics> probes.new-sum-probe name = "dev.mem_access_count.per_id.sum"
    display-name = "Mem accesses, total for all IDs" probe =
    dev.mem_access_count.per_id objects =
    (ribit.shared_out.port.probes_per_id)
    ```

    This will create a new **global** probe (that means not associated without any specific object) of probe kind **name** which will be described in outputs as **`display-name`** and it will sum all probes of kind **probe** that belong to any of the elements provided in **`objects`**.

52. List the device probe kinds again.

    ```
    simics> probes.list-kinds substr = "dev."
    ```

    Provided you named the sum probe correctly, you should see it now listed here together with the pre-existing device probes.

53. Try to read the newly created sum probe for the console bytes.

    ```
    simics> probes.read probe-kind = dev.mem_access_count.per_id.sum
    ```

    This does not work, because combined probes need to be subscribed to (that is, they need to be activated individually). The reason is that these probes are considered expensive. They both read multiple probes and perform further operations with the values.

54. Subscribe to the sum probe.

```
simics> probes.subscribe probe-kind = dev.mem_access_count.per_id.sum
```

55. Try reading the probe again.

```
simics> probes.read probe-kind = dev.mem_access_count.per_id.sum
```

    Note that the probe is considered to belong to the **sim** object because it is global. Currently the probe value is 0 since nothing has happened yet.

56. Derived probes can form a hierarchy, using derived probes as inputs to other derived probes. This can let the probe system perform complex computations to summarize information.

    Let's create a probe that reports how many bytes more are sent to the console, compared to the writes to the shared output device. This can be done based on the sum of accesses per ID and the total console bytes written (the wrapped attribute).

```
simics> probes.new-fraction-probe name = dev.con_bytes_augmentation
numerator-probe = dev.con_bytes_written denominator-probe =
dev.mem_access_count.per_id.sum display-name = "Console bytes
augmentation factor"
```

    The use of the **mem_access_count** as the denominator makes sense, since each access to the output register is exactly one byte in size. If software starts to make bad accesses, the computation will not make sense.

    Note that there are more types of probes that can be created than what is shown in this lab. Inspect the output of "**list-commands object = probes**" and read the individual help text for the commands to find out more.

57. Subscribe to the newly created probe.

```
simcis> probes.subscribe dev.con_bytes_augmentation
```

    Now we are ready to monitor how the number of bytes "flowing through our system" evolves over time.

3.6 Using Probe Monitors to Observe Model Behavior over Time

58. Create a new probe monitor to observe what happens over virtual time.

```
simcis> new-probe-monitor name = mon0 clock = ribit.clock sampling-mode
= virtual interval = 0.00005
```

    This monitor will print every 50 microseconds (virtual time). This interval is based on observing the software behavior. Usually, some trial and error is needed to find the informative sampling intervals.

59. Now add the following probes to the monitor:

```
simics> mon0.add-probe mode = current probe =
sim:dev.mem_access_count.per_id.sum
simics> mon0.add-probe mode = current probe =
ribit.shared_out:dev.con_bytes_written
simics> mon0.add-probe mode = current probe =
ribit.shared_out:dev.con_bytes_augmentation
simics> mon0.add-probe probe = sim:dev.mem_access_count.per_id.sum
simics> mon0.add-probe probe = ribit.shared_out:dev.con_bytes_written
simics> mon0.add-probe probe =
ribit.shared_out:dev.con_bytes_augmentation
```

This adds the probes of interest to the monitor, both as current values and the delta since the last sample. Delta is the default.

60. Run for 1 millisecond.

```
simics> run 1 ms
```

The output is first grouped by owning object: first the global probes, then the device specific probes.

Two global probes (total value and change per sample) monitor the number of accesses. Then follow the current values for the console bytes output and the augmentation probe, followed by the per-sample data.

The *current* console bytes augmentation factor column indicates the aggregate over the whole monitored run. It is constantly increasing since over time there is more and more colored output.

The *per-sample* augmentation rate is not a delta from the previous sample but computed from the current sample's delta for accesses and bytes output. I.e., it is the current augmentation factor for the sample. Over time, this can go up and down, depending on how the sample aligns with the operation of the software. It can even fall back to 1 (no augmentation), when all characters written during the sample used the plain output register.

With the current code and sample length setting, each sample produces between 70 and 110 accesses. Initially, the console bytes increase at the same rate, so the augmentation is 1.0. In these samples, no colored output had been written at all. Then the number of written bytes starts to increase, and with it, the augmentation rate.

61. Repeat the 1 millisecond run command until the software has stopped printing. At this point, the delta or per-sample counters all show zero, and the per-sample augmentation shows a dash.

    Ultimately you will see that the last valid fraction in a sample was as high as 33, so for every byte written to the device we sent 33 bytes to the console. You can also see that the average augmentation is 16.75. Looking back (scrolling up) you will see that the per-sample augmentation can differ quite a bit, like going from 14 down to 7 up to 12 and 16 again.

    Again, such analysis can give insights into unexpected effects (not the case here) and get a better understanding of changes in system behavior over time.

62. This concludes the lab. **Quit** the simulation session.

    ```
    simics> quit
    ```