Intel® Simics® Simulator New User Training

Lab 004 –Log, trace, and instrumentation

Lab 004 –Log, trace, and instrumentation

The Intel® Simics® Simulator provides a controllable and observable environment that lets you monitor and inspect the operations and state changes in the target system – without disturbing its execution, and without software on the target having any idea that it is being observed. In this lab, you will try several different ways to log and trace execution in a simulation.

A. The log system

The log system in the Intel Simics Simulator lets devices and other modules report what goes on in the simulated hardware and in the simulator itself and report it to the user and scripts. Logs are classified by log level, log group, log type, and emitting objects, allowing for precise filtering and breaking.

1. **Open** the checkpoint you saved in Lab 001, after having done **`insmod`** on the driver (probably called **`AfterDriver.ckpt`**). This will get you to a booted system.

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

2. To keep the rate of logs down a bit, make sure to enable real-time-mode. Go to the command line:

```
simics> enable-real-time-mode
```

3. Execute the following command to display the logging commands. The CLI response describes all the command options available for logging.

```
simics> help Logging
```

Which should result in output like this:

```
Commands available in the "Logging" category:

<conf_object>.log-group   enable/disable a log group
<conf_object>.log-level   set or get the log level
log                       print log entries for all objects
log-filter                suppress log messages for the object
log-group                 enable/disable a log group
log-level                 set or get the log level
log-setup                 configure log behavior
log-size                  set log buffer size
log-type                  set or get the current log types
```

You will use logging to see what goes on inside the training device subsystem.

4. List the set of objects in the **`training_card`** subsystem (excepting port objects by not explicitly listing them):

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

The output is still long:

```
┐
├ button_a ┐
│          └ i2c_ep
├ button_b ┐
│          └ i2c_ep
├ i2c_bus
├ local_memory
├ master_bb ┐
│           └ i2c_ep
├ panel ┐
│       ├ background ┐
│       │            ├ board
│       │            └ led_panel
…
├ rgb_led[62] ┐
│             └ i2c_ep
├ rgb_led[63] ┐
│             └ i2c_ep
├ toggle_t ┐
│          └ i2c_ep
└ toggle_u ┐
           └ i2c_ep
```

5. Dig in a bit on the **`rgb_led`** array, including port objects:

```
simics> list-objects -tree -show-port-objects namespace =
"machine.training_card.rgb_led"
```

Note that each **`rgb_led`** object has a register bank for its I2C -accessible registers, a port for receiving I2C operations, and an I2C endpoint that connects it to the I2C bus in the training card.

```
...
                          └ rgb_led[63] ┐
                                        ├ bank ┐
                                        │      └ i2cregs
                                        ├ i2c_ep
                                        └ port ┐
                                               └ i2c_in
```

Each LED, button, and toggle in the panel is its own separately addressable device on the I2C bus. The endpoint objects are intermediaries between the simulator network links and the connected objects.

6. To find the total number of objects (including bank and port objects) in the **`training_card`**, you can retrieve all the objects using **`list-objects`** and pass the resulting list to **`list-length`**:

```
simics> list-length (list-objects -show-port-objects
namespace=machine.training_card)
```

The result shows several hundred objects. Turning on logging on everything might get overwhelming.

Configuring log output to file, hiding logs on CLI

7. Save all log messages to a file for later review, using the **`log-setup`** command. You want to have time stamps on all messages and save to a file (using **`-overwrite`** if the file already exists). In addition, ask the logging setup to print the log group used for each message, and to not echo all log messages to the CLI:

```
simics> log-setup -time-stamp -group -overwrite -no-console logfile =
lab004.log
```

Note that saving logs to a file does not automatically imply that messages are hidden from the CLI, the default is to save logs to file *in addition to* appearing elsewhere. To avoid overwhelming the console with huge numbers of log messages, this example uses **`-no-console`** to hide the messages.

Set up logging from devices

8. Configure the training card subsystem to emit lots of logs by raising the log level to 3 in the entire subsystem. Log level 3 prints detailed "model debug" information about the internal behavior of the model. With the **`-r`** flag to the **`log-level`** command, you can tell the simulator to apply the same log level recursively in the object tree from a certain point:

```
simics> log-level machine.training_card 3 -r
```

9. The log level settings for a certain object or set of objects can be reviewed using the **`log-level`** command without a level argument. Check the log level of the objects in the **`machine.training_card`** subsystem:

```
simics> log-level machine.training_card
```

All objects at that level of the hierarchy are listed, along with their ports and banks. The log level of each of them should be 3.

```
Current log levels:

Lvl  Object
---------------------------------------------------------
  3  machine.training_card
  3  machine.training_card.button_a
  3  machine.training_card.button_a.bank
  3  machine.training_card.button_a.bank.i2cregs
  3  machine.training_card.button_a.port
  3  machine.training_card.button_a.port.button_in
...
```

10. Check the log level on **machine.mb**:

```
simics> log-level machine.mb
```

Note that the objects in this hierarchy are all set to use log level 1. The **log-level 3** operation only affected the **training_card** subsystem.

```
Current log levels:

Lvl  Object
--------------------------------------------------
  1  machine.mb
  1  machine.mb.apic_bus
  1  machine.mb.bcast
  1  machine.mb.conf
...
```

11. Run the simulation:

```
simics> r
```

12. Go to the target **Serial Console** window, and display a new pattern on the LEDs by sending one of pattern files to the char device:

```
# cat chessboard2.txt > /dev/simics_training_pcie_driver
```

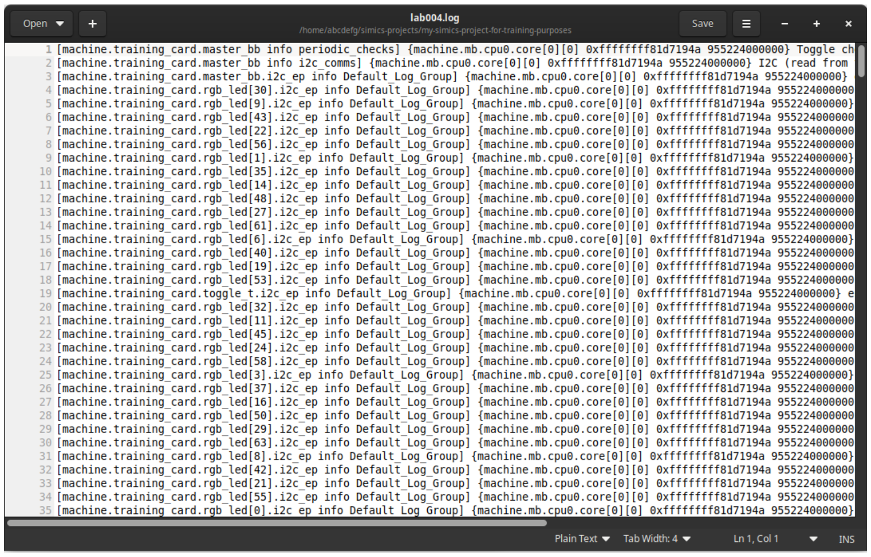
Note that executing this command takes longer than it did before – since the simulator is very busy logging all activities in the training card in the background it is simulating more slowly.

Review the very large log file

13. **Pause** the simulation once the command has finished and the target has returned to shell and the **LED Panel** has been updated.

```
running> stop
```

14. Open the created log file **lab004.log** in an editor of your choice.


Open
lab004.log
/home/abcdefg/simics-projects/my-simics-project-for-training-purposes
Save
Plain Text
Tab Width: 4
Ln 1, Col 1
INS

```
[machine.training_card.master_bb info periodic_checks] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} Toggle ch
[machine.training_card.master_bb info i2c_comms] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} I2C (read from
[machine.training_card.master_bb.i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
[machine.training_card.rgb_led[30].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[9].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
[machine.training_card.rgb_led[43].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[22].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[56].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[1].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
[machine.training_card.rgb_led[35].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[14].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[48].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[27].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[61].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[6].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
[machine.training_card.rgb_led[40].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[19].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[53].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.toggle_t.i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} e
[machine.training_card.rgb_led[32].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[11].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[45].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[24].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[58].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[3].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
[machine.training_card.rgb_led[37].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[16].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[50].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[29].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[63].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[8].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
[machine.training_card.rgb_led[42].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[21].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[55].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000
[machine.training_card.rgb_led[0].i2c_ep info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000}
```

If your editor chokes on opening it, find another editor that can open huge files or use a shell command (like **`cat`**) to inspect the file.

15. Look at the log messages in the log file.

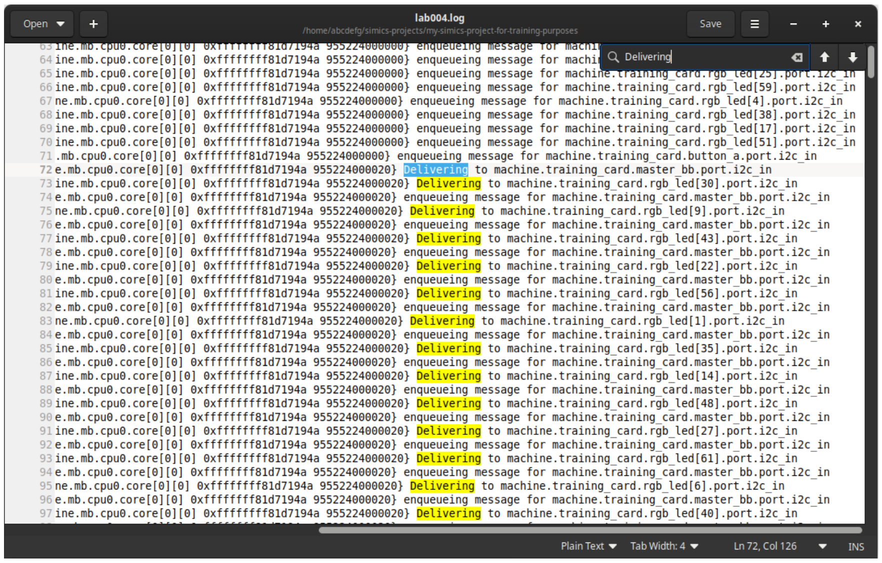
    Note how each message prints the object logging, the type of the log (most are **`info`**), the log group (mostly **`Default_Log_Group`**), the current processor, instruction pointer, time when the log was emitted, and finally the actual message.

    The time for each message is shown in the **`{}`** block that also shows the current processor and its current instruction pointer. For example:

```
[machine.training_card.master_bb.i2c_ep info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 4641800800000} enqueueing message for
machine.training_card.master_bb.port.i2c_in
```

16. To see how the I2C message protocol plays out over time, find the first point where a message is delivered from the link to devices (before that point, messages are enqueueing in the link).

    Search for the word "**`Delivering`**". This should get you to a point where one step of the protocol ends and another begins, like this:


Open
Save
lab004.log
/home/abcdefg/simics-projects/my-simics-project-for-training-purposes
Delivering
Plain Text
Tab Width: 4
Ln 72, Col 126
INS

```
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machi
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machi
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.rgb_led[25].port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.rgb_led[59].port.i2c_in
ne.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.rgb_led[4].port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.rgb_led[38].port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.rgb_led[17].port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.rgb_led[51].port.i2c_in
.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000000} enqueueing message for machine.training_card.button_a.port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[30].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ne.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[9].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[43].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[22].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[56].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ne.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[1].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[35].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[14].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[48].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[27].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[61].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ne.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[6].port.i2c_in
e.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} enqueueing message for machine.training_card.master_bb.port.i2c_in
ine.mb.cpu0.core[0][0] 0xffffffff81d7194a 955224000020} Delivering to machine.training_card.rgb_led[40].port.i2c_in
```

17. You should see something like this:

```
[machine.training_card.rgb_led[0].i2c_ep info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 4641800800000} enqueueing message for
machine.training_card.rgb_led[0].port.i2c_in
[machine.training_card.master_bb.i2c_ep info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 4641800800020} delivering to
machine.training_card.master_bb.port.i2c_in
```

    Note how time steps forward by 20 cycles between the log messages – this is because the latency on the I2C bus is set to be 10ns, which works out to 20 cycles at the processor clock frequency of 2000 MHz.

    Also note that there are many operations taking place on the same cycle – in the Intel Simics Simulator I2C bus model used in this setup, all I2C devices get their messages on the same cycle (since the bus is a broadcast bus).

Reducing the log to a manageable size

This is too much logging to make sense of, at least for a human doing manual inspection.

18. Clear the log file by re-issuing the log-setup command, allowing output to the console this time:

```
simics> log-setup -time-stamp -group -overwrite logfile = lab004.log
```

19. Turn off logging on all the endpoint objects. To do this, use **`list-objects`** to find all **`i2c-link-endpoint`** objects inside the training card. Tell the simulator to look inside of the **`training_card`** and find all i2c endpoints anywhere in the hierarchy beneath it.

```
simics> $l = (list-objects namespace=machine.training_card class=i2c-
link-endpoint)
```

20. Check the set of objects:

```
simics> $l
```

21. And how many there are:

```
simics> list-length $l
```

22. Use a **foreach** loop in CLI to set the log level of all the objects:

```
simics> foreach $o in $l { ($o.log-level 1) }
```

Note that the CLI prints a notification about a single log-level change rather than changes to all objects. That is artifact of CLI, and the change will have been applied to all objects in the list.

23. Do a spot check to see that this worked:

```
simics> log-level machine.training_card.button_a.i2c_ep
```

It should show "1".

24. Retrieving a set of objects like this and iterating over them is a general technique that can be used with any queries to the list-objects command. For the specific case of setting the log level for objects of the same class, there is a class argument to the **log-level** command.

```
simics> log-level class = i2c-link-endpoint 2
```

25. Do a spot check again:

```
simics> log-level machine.training_card.button_a.i2c_ep
```

Which should show "2".

26. Set the log level to 1 again:

```
simics> log-level class = i2c-link-endpoint 1
```

27. Double-check the log level on all devices in the training card again:

```
simics> log-level machine.training_card
```

Check that objects called ...**i2c_ep** have log-level 1 and the other 3:

```
Current log levels:

Lvl  Object                                              | Lvl  Object
-----------------------------------------------------------+---------------------------------------------------
  3  machine.training_card                               |   3  machine.training_card.rgb_led[60].port
...
  3  machine.training_card.button_a.bank.i2cregs         |   3  machine.training_card.rgb_led[62].bank
  1  machine.training_card.button_a.i2c_ep               |   3  machine.training_card.rgb_led[62].bank.i2cregs
  3  machine.training_card.button_a.port                 |   3  machine.training_card.rgb_led[62].port
```

28. Next, turn off all logging for the log group **i2c_comms** (which is used to mark logs related to I2C communications in the objects in the **training_card**). This log group is specific to the models inside **training_card**, and does not exist in all objects in the simulation. However, the Intel Simics Simulator is smart enough to turn off a log group only in objects that actually use the log group. Thus, you can do this globally:

```
simics> log-group -disable i2c_comms
```

The CLI should print a message that the group was disabled in a number of objects:

```
simics> log-group -disable i2c_comms
Log group 'i2c_comms' was disabled in 218 object(s).
```

29. Double-check the effect by looking at the master black-box device in the training card:

```
simics> log-group machine.training_card.master_bb
```

Note that there are quite a few log groups available in this object, and that all are enabled except the **i2c_comms** group. You see the information for all the ports and banks of the object as well, since logging can be independently configured on each port and bank.

```
machine.training_card.master_bb:
 enabled log groups: "periodic_checks" "MSIX_interrupt" "PCI_config" "PCI_DMA" "PCI_IRQ"
"Register_Read" "Register_Write" "Default_Log_Group"
 disabled log groups: "i2c_comms"
machine.training_card.master_bb.bank:
 enabled log groups: "Default_Log_Group"
 disabled log groups:
machine.training_card.master_bb.bank.dev_msix_pba:
 enabled log groups: "periodic_checks" "MSIX_interrupt" "PCI_config" "PCI_DMA" "PCI_IRQ"
"Register_Read" "Register_Write" "Default_Log_Group"
...
```

Using **log-group** like this is a good way to discover available log groups on a particular object (and its ports and banks).

30. **Run** the simulation forward again.

```
simics> r
```

31. Go to the target console in the Serial **Console Window** and enter a new command that shows a new pattern.

```
# cat chessboard1.txt > /dev/simics_training_pcie_driver
```

Note that the execution is faster than before – even if there is still a lot of logging going on.

32. **Pause** simulation:

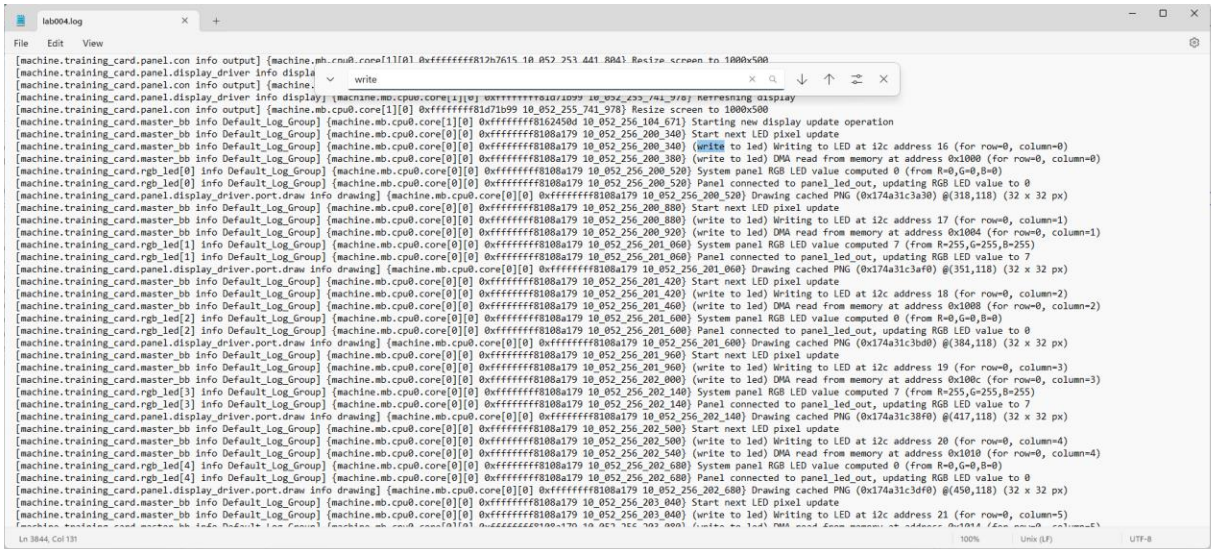
```
running> stop
```

33. Open **lab004.log** again. It starts with periodic I2C reads of the toggle objects, which is how the training card controller determines the state of the toggles:

```
[machine.training_card.master_bb info periodic_checks] {machine.mb.cpu0.core[0][0]
0xffffffff81d7194a 19388566200000} Toggle check event for toggle index 0
[machine.training_card.toggle_t.port.i2c_in info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 19388566200060} I2C read of toggle value:
replying with 0
[machine.training_card.master_bb info periodic_checks] {machine.mb.cpu0.core[0][0]
0xffffffff81d7194a 19388606200000} Toggle check event for toggle index 1
[machine.training_card.toggle_u.port.i2c_in info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 19388606200060} I2C read of toggle value:
replying with 0
```

Note that I2C activity is logged on the I2C port objects of the toggles (**`....toggle_u.port.i2c_in`**).

34. To find the logs that correspond to the update of the LEDs, search for "**`write`**":



```
[machine.training_card.panel.con info output] {machine.mb.cpu0.core[1][0] 0xffffffff812b7615 10_052_253_441_804} Resize screen to 1000x500
[machine.training_card.panel.display_driver info displa
[machine.training_card.panel.con info output] {machine.
[machine.training_card.panel.display_driver info display] {machine.mb.cpu0.core[1][0] 0xffffffff81d71b99 10_052_255_741_978} Refreshing display
[machine.training_card.panel.con info output] {machine.mb.cpu0.core[1][0] 0xffffffff81d71b99 10_052_255_741_978} Resize screen to 1000x500
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff8162450d 10_052_256_104_671} Starting new display update operation
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_340} Start next LED pixel update
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_340} (write to led) Writing to LED at i2c address 16 (for row=0, column=0)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_380} (write to led) DMA read from memory at address 0x1000 (for row=0, column=0)
[machine.training_card.rgb_led[0] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_520} System panel RGB LED value computed 0 (from R=0,G=0,B=0)
[machine.training_card.rgb_led[0] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_520} Panel connected to panel_led_out, updating RGB LED value to 0
[machine.training_card.panel.display_driver.port.draw info drawing] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_520} Drawing cached PNG (0x174a31c3a30) @(318,118) (32 x 32 px)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_880} Start next LED pixel update
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_880} (write to led) Writing to LED at i2c address 17 (for row=0, column=1)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_200_920} (write to led) DMA read from memory at address 0x1004 (for row=0, column=1)
[machine.training_card.rgb_led[1] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_060} System panel RGB LED value computed 7 (from R=255,G=255,B=255)
[machine.training_card.rgb_led[1] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_060} Panel connected to panel_led_out, updating RGB LED value to 7
[machine.training_card.panel.display_driver.port.draw info drawing] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_060} Drawing cached PNG (0x174a31c3af0) @(351,118) (32 x 32 px)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_420} Start next LED pixel update
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_420} (write to led) Writing to LED at i2c address 18 (for row=0, column=2)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_460} (write to led) DMA read from memory at address 0x1008 (for row=0, column=2)
[machine.training_card.rgb_led[2] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_600} System panel RGB LED value computed 0 (from R=0,G=0,B=0)
[machine.training_card.rgb_led[2] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_600} Panel connected to panel_led_out, updating RGB LED value to 0
[machine.training_card.panel.display_driver.port.draw info drawing] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_600} Drawing cached PNG (0x174a31c3bd0) @(384,118) (32 x 32 px)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_960} Start next LED pixel update
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_201_960} (write to led) Writing to LED at i2c address 19 (for row=0, column=3)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_000} (write to led) DMA read from memory at address 0x100c (for row=0, column=3)
[machine.training_card.rgb_led[3] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_140} System panel RGB LED value computed 7 (from R=255,G=255,B=255)
[machine.training_card.rgb_led[3] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_140} Panel connected to panel_led_out, updating RGB LED value to 7
[machine.training_card.panel.display_driver.port.draw info drawing] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_140} Drawing cached PNG (0x174a31c38f0) @(417,118) (32 x 32 px)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_500} Start next LED pixel update
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_500} (write to led) Writing to LED at i2c address 20 (for row=0, column=4)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_540} (write to led) DMA read from memory at address 0x1010 (for row=0, column=4)
[machine.training_card.rgb_led[4] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_680} System panel RGB LED value computed 0 (from R=0,G=0,B=0)
[machine.training_card.rgb_led[4] info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_680} Panel connected to panel_led_out, updating RGB LED value to 0
[machine.training_card.panel.display_driver.port.draw info drawing] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_202_680} Drawing cached PNG (0x174a31c3df0) @(450,118) (32 x 32 px)
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_203_040} Start next LED pixel update
[machine.training_card.master_bb info Default_Log_Group] {machine.mb.cpu0.core[0][0] 0xffffffff8108a179 10_052_256_203_040} (write to led) Writing to LED at i2c address 21 (for row=0, column=5)
```

This shows a reduced trace of the steps involved in updating the LEDs. Configuring logging based on objects, levels, and groups makes it possible to focus on what is interesting for a particular investigation.

35. Turn down the log noise:

```
simics> log-level 1
```

36. **Do not quit** this simulation session, it will be used in the next section!

B. Trace commands

Trace commands trace specific activities in the system. In this section, you will look at some examples of such command.

1. Trace commands generate output that follows **`log-setup`** settings. To ensure output is visible on the command-line, use **`-console`**. Also, turn off logging to file:

```
simics> log-setup -console -no-log-file
```

2. Check the current log settings:

```
simics> log-setup
```

The output should be like this:

```
Time stamp      : enabled
Picoseconds     : disabled
Real time       : disabled
Disassemble     : disabled
Log to console  : enabled
Include group   : enabled
Include level   : disabled
Log file        : disabled
Format          : "default"
```

3. The primary source of "trace" commands is the breakpoint manager. List the set of available breakpoint types:

```
simics> bp.list-types
```

This list will expand with new simulator releases. All breakpoints in the breakpoint manager have the ability to print trace messages when they occur.

Tracing processor exceptions

4. To trace processor exceptions, use the **`exception`** category in the breakpoint manager. The command **`trace`** is used to trace all occurrences of exceptions. It is necessary to specify the set of processors to trace. With several processors in the system, the quickest way to do this is to apply the breakpoint to all objects in the **`machine.mb.cpu0`** hierarchy:

```
simics> bp.exception.trace -all object = "machine.mb.cpu0" -recursive
```

5. Check that the trace applies to all processors:

```
simics> bp.list
```

The trace configuration will appear as a breakpoint:

```
┌──┬──────────────────────────────────────────────────┬───────┬───────┬────────────┬─────────┐
│ID│                   Description                    │Enabled│Oneshot│Ignore count│Hit count│
├──┼──────────────────────────────────────────────────┼───────┼───────┼────────────┼─────────┤
│ 2│Break on any exception on the following objects:  │ true  │ false │           0│        0│
│  │machine.mb.cpu0.core[0][0]                        │       │       │            │         │
│  │machine.mb.cpu0.core[1][0]                        │       │       │            │         │
└──┴──────────────────────────────────────────────────┴───────┴───────┴────────────┴─────────┘
```

6. Compare the set of processors listed to the list of all processors in the system:

```
simics> list-processors
```

7. **Run** the simulation.

```
simics> r
```

You will see a steady flow of exceptions being traced.

```
simics> run
[bp.exception trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff81fb1eef
911024060376} [trace:1] machine.mb.cpu0.core[1][0] Interrupt_35(547) exception triggered
[bp.exception trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff81fb1eef
911028060376} [trace:1] machine.mb.cpu0.core[1][0] Interrupt_35(547) exception triggered
...
```

8. **Pause** the simulation:

```
running> stop
```

Find the panel button exception

9. The next goal is to find the exception that is generated when buttons are clicked in the panel. To do this, it is necessary to find out when the training card controller raises interrupts, and then match that to the processor exceptions. To find out when the controller generates interrupts on its side, raise the log level of the controller along with its I2C endpoint. To do this, use the **-r** "recursive" option to the log-level command.

```
simics> log-level machine.training_card.master_bb 3 -r
```

10. Enable logging of I2C communications again:

```
simics> log-group -enable i2c_comms
```

11. Inject a click on button A in the panel from the command line. Injections like this work whether the panel is visible or not, or actually even if there is no panel in the model at all. The command targets the input of the button object in the training card.

Find all interfaces of the **button_a** object in the training card:

```
simics> list-interfaces machine.training_card.button_a
```

The output indicates that **port.button_in** receives inputs from the panel.

```
simics> list-interfaces machine.training_card.button_a
...
┌──────────────────────────────────────────────────────────────────────────┐
│                              Port objects                                │
├───────────┬──────────────────────────────┬───────────────────────────────┤
│Portname   │Interfaces                    │Description                    │
├───────────┼──────────────────────────────┼───────────────────────────────┤
...
├───────────┼──────────────────────────────┼───────────────────────────────┤
│port.button_in│signal                     │Button presses from the panel  │
├───────────┼──────────────────────────────┼───────────────────────────────┤
...
```

12. Raise the log level on the **button_a** object:

```
simics> log-level machine.training_card.button_a 3
```

13. Fake a click by raising and lowering the signal. Use inline Python to call the methods in the interface. First, **signal_raise()**:

```
simics>
@conf.machine.training_card.button_a.port.button_in.iface.signal.signal_
raise()
```

The CLI will immediately log the effect, with the button signaling back to the master.

```
[machine.training_card.button_a.port.button_in info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 19422533600000} button pressed (signal
raised)
[machine.training_card.button_a info i2c_comms] {machine.mb.cpu0.core[0][0]
0xffffffff81d7194a 19422533600000} Starting to send notification over i2c bus
[machine.training_card.button_a info i2c_comms] {machine.mb.cpu0.core[0][0]
0xffffffff81d7194a 19422533600000} Bus is free, send start(200) to get going
[machine.training_card.master_bb.i2c_ep info Default_Log_Group]
{machine.mb.cpu0.core[0][0] 0xffffffff81d7194a 19422533600000} enqueueing message for
machine.training_card.master_bb.port.i2c_in
```

14. Lower the signal, as that is what the button expects immediately after a raise (the button is driven by changes to the signal, not by its level):

```
simics>
@conf.machine.training_card.button_a.port.button_in.iface.signal.signal_
lower()
```

15. Lower the log level on the **button_a** object, as it will produce a ton of noise for the next step. In the next phase, only the **master_bb** is interesting.

```
simics> log-level machine.training_card.button_a 1
```

16. Delivering messages on the I2C bus takes a small amount of simulated time as you saw in the previous lab. Instead of estimating or guessing the time needed for the notification to reach all the way to the processor, it is better to simply tell the simulator to stop when the next exception triggers.

Use **bp.exception.break** in the same way as **bp.exception.trace** above:

```
simics> bp.exception.break -all object = "machine.mb.cpu0" -recursive
```

17. **Run** until the breakpoint hits:

```
simics> r
```

This will print a sequence of logs from the master device as it receives an I2C write from the button device. It shows the sequence of messages on the I2C bus, and how it plays out with small delays between the steps. It will end with a message that device has indeed sent an MSI-X interrupt over the PCIe system back towards the processor.

The output will look something like this:

```
simics> run
...
[machine.training_card.master_bb.port.i2c_in info i2c_comms] {machine.mb.cpu0.core[0][0]
0xffffffff81fb2ad6 911028200140} I2C stop incoming - noting bus as free
[machine.training_card.master_bb info MSIX_interrupt] {machine.mb.cpu0.core[0][0]
0xffffffff81fb2ad6 911028200140} Raising interrupt for MSI-X for button press.
[bp.exception trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff81fb1eef
911028226304} [trace:1] machine.mb.cpu0.core[1][0] Interrupt_38(550) exception triggered
[machine.mb.cpu0.core[1][0]] Breakpoint 2: machine.mb.cpu0.core[1][0] Interrupt_38(550)
exception triggered
simics>
```

The two messages about the exception triggering are due to there being both a breakpoint and a trace point active.

What this shows is that interrupts from the training card are received as interrupt number 38 by the processor cores.

18. Turn off exception tracing and breakpoints:

```
simics> bp.delete -all
```

19. Lower the log level on the master device and all its subordinate devices in the object tree:

```
simics> log-level machine.training_card.master_bb 1 -r
```

Tracing memory-mapped operations to devices

The simulator can trace all memory-mapped accesses to register banks. This is accomplished using the **bp.bank** breakpoint type in the breakpoint manager.

20. Trace memory-mapped accesses for the master device in the training card. Typically, these operations are issued from the processor that runs the device driver that controls the card. Tracing a device implicitly traces accesses to all banks in the device.

```
simics> bp.bank.trace machine.training_card.master_bb
```

21. Check the trace setup:

```
simics> bp.list
```

The breakpoint is set on all banks of the **machine.training_card.master_bb** device.

22. **Run** the simulation forward.

```
simics> r
```

23. Go to the serial console, and start a read from the training device driver:

```
# cat /dev/simics_training_pcie_driver
```

24. Go to the **System Panel**, and click on **button A**. Alternatively, use command-line operations to drive the button signals as used above.

Each button press produces a couple of read and write operations, as the device driver reads from the status registers in the device to find out which button was pressed and then writes to a register to clear the status.

Pressing button A should result in a read from offset 0x204 followed by a write:

```
running> @conf.machine.training_card.button_a.port.button_in.iface.signal.signal_raise()

[bp.bank trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff8162444b
19720133848540} [trace:3] machine.training_card.master_bb.bank.regs read at offset=0x204
size=0x4 value=0x1 ini=machine.mb.cpu0.core[1][0]
[bp.bank trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff8162450d
19720133848557} [trace:3] machine.training_card.master_bb.bank.regs write at
offset=0x204 size=0x4 value=0x1 ini=machine.mb.cpu0.core[1][0]
```

25. Check the names of the accessed registers:

```
running> output-radix 16
running> print-device-regs machine.training_card.master_bb.bank.regs
pattern = "button*"
```

The list of registers contains the register at offset **0x204**, which is **button_a_status**:

```
Offset  Name                       Size   Value | Offset  Name                        Size   Value
------------------------------------------------+--------------------------------------------
0x0200  button_interrupt_control      4  0x0001 | 0x0810  button_a_i2c_address           4  0x005c
0x0204  button_a_status               4  0x0000 | 0x0814  button_b_i2c_address           4  0x005d
0x0208  button_b_status               4  0x0000 | 0x0818  button_n_i2c_address[0]        4  0x0000
...
```

26. Click on **button B.** This produces an additional read as the software reads each button status in turn to find the one that was activated:

```
[bp.bank trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff8162444b
20960733848416} [trace:3] machine.training_card.master_bb.bank.regs read at offset=0x204
size=0x4 value=0x0 ini=machine.mb.cpu0.core[1][0]
[bp.bank trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff8162444b
20960733848436} [trace:3] machine.training_card.master_bb.bank.regs read at offset=0x208
size=0x4 value=0x1 ini=machine.mb.cpu0.core[1][0]
[bp.bank trace Default_Log_Group] {machine.mb.cpu0.core[1][0] 0xffffffff8162450d
20960733848453} [trace:3] machine.training_card.master_bb.bank.regs write at
offset=0x208 size=0x4 value=0x1 ini=machine.mb.cpu0.core[1][0]
```

If you plan to do a lot of register access tracing, it is highly recommended to redirect output to a file using **`log-setup`**.

27. **Quit** the simulation session.

28. Close the **`lab004.log`** file if still open.

C. Instrumentation

The instrumentation system in the Intel Simics Simulator provides a set of tools to inspect, trace, and collect statistics on the execution of the simulator and the software running on it. There are multiple instrumentation tools available, and more are added over time.

1. Start a new empty simulation session.

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

2. List the available instrumentation tools:

```
simics> list-instrumentation-tools
```

   In the next section, you will be using the bank coverage tool.

Set up bank coverage

The device register or bank coverage tool is an example of an instrumentation-based tool. It tracks the device registers accessed from software and produces reports that describe which registers have been accessed (coverage) as well as how many times each register has been read and written (access count). In this lab section, you will use this feature to inspect the software actions of the device driver for the training card.

3. Load the checkpoint saved in Lab 001, before the **insmod** command has been executed on the target. Probably called "**MyFirstCheckpoint.ckpt**". It is important to start before the driver has been activated, since you want to study which registers the driver uses during startup. List the checkpoints that the simulator knows about:

```
simics> list-checkpoints
```

4. Open the first checkpoint:

```
simics> read-configuration MyFirstCheckpoint.ckpt
```

5. Create a new register coverage tool, using the **new-bank-coverage-tool** command. Use the **parent** parameter to request coverage for all register banks inside the **training_card** (the "parent" mechanism applies to all instrumentation tools as well as some other CLI commands, and is handy to apply some particular functionality to a subset of a system).

```
simics> new-bank-coverage-tool devregcov parent = machine.training_card
```

   The argument **devregcov** is there to provide a name for the created tool (if you do not provide a name, the simulator will generate a generic unique name for you).

6. Use the **coverage** command on the **devregcov** tool to check the current coverage (which is all zero). The **max** argument overrides the default number of lines shown, to make sure all the devices in the **training_card** are shown:

```
simics> devregcov.coverage max=70
```

This lists the current coverage (all zero) and register count for all the devices in the subsystem that are being instrumented. None of the devices have very many registers except the **`master_bb`** device. The register count for a device includes all its register banks, including PCI configuration banks, MSI-X interrupt tables, etc., as well as the non-software-accessible i2c register banks found in all the training card device models.

Collect bank coverage information

7. **Run** the simulation.

```
simics> r
```

8. Go to the target **Serial Console** window and issue the command (just like several times before):

```
# insmod simics-training-pcie-driver.ko
```

9. Once the **`insmod`** command has completed, **pause** the simulation again. Note that using device register coverage does not affect the simulation speed all that much.

```
running> stop
```

10. Check the register coverage achieved by the activation of the driver:

```
simics> devregcov.coverage
```

The coverage is not very complete. Only the **`master_bb`** device has been accessed at all, and most of its registers are yet to be used. It should look something like this:

```
┌─────┬─────────────────────────────────┬────────────────────────┬─────────────┬─────────┐
│Row #│             Device              │No. accessed registers│No. registers│Coverage%│
├─────┼─────────────────────────────────┼────────────────────────┼─────────────┼─────────┤
│    1│machine.training_card.master_bb  │                      21│          160│   13.12%│
│    2│machine.training_card.button_a   │                       0│            3│    0.00%│
│    3│machine.training_card.button_b   │                       0│            3│    0.00%│
│    4│machine.training_card.rgb_led[0] │                       0│            4│    0.00%│
│    5│machine.training_card.rgb_led[10]│                       0│            4│    0.00%│
│    6│machine.training_card.rgb_led[11]│                       0│            4│    0.00%│
│    7│machine.training_card.rgb_led[12]│                       0│            4│    0.00%│
│    8│machine.training_card.rgb_led[13]│                       0│            4│    0.00%│
│    9│machine.training_card.rgb_led[14]│                       0│            4│    0.00%│
│   10│machine.training_card.rgb_led[15]│                       0│            4│    0.00%│
│   11│machine.training_card.rgb_led[16]│                       0│            4│    0.00%│
...
```

11. The coverage tool offers several ways to format and select the data that is being displayed. For example, sort the output on how many registers there are in the banks, and add more unnecessary decimals to the floating-point output:

```
simics> devregcov.coverage sort-on-column = "No. registers" float-
decimals = 4
```

12. Coverage tracks reads and writes separately, but presents them as an aggregate by default. To see the coverage from just write operations. Since the default sort on

"Coverage %" does not provide useful operation, sort on the number of accessed registers instead.

```
simics> devregcov.coverage -write max=10 sort-on-column = "No. accessed
registers"
```

Resulting in:

```
Excluding 293 read-only registers.
┌────────┬────────────────────┬────────────────────┬────────────┬────────────┬─────────┐
│ Row #  │       Device       │No. accessed registers│No. accessed│No. registers│Coverage%│
│        │                    │                    │ registers% │            │         │
├────────┼────────────────────┼────────────────────┼────────────┼────────────┼─────────┤
│       1│machine.training_card.│                  18│     100.00%│         133│   13.53%│
│        │master_bb           │                    │            │            │         │
├────────┼────────────────────┼────────────────────┼────────────┼────────────┼─────────┤
│       2│machine.training_card.│                   0│       0.00%│           0│  100.00%│
│        │button_a            │                    │            │            │         │
├────────┼────────────────────┼────────────────────┼────────────┼────────────┼─────────┤
│       3│machine.training_card.│                   0│       0.00%│           0│  100.00%│
│        │button_b            │                    │            │            │         │
├────────┼────────────────────┼────────────────────┼────────────┼────────────┼─────────┤
...
```

13. The coverage tool can also show you exactly which registers have been accessed. Zoom in on the main control registers bank (**`regs`**) of the **`master_bb`** device:

```
simics> devregcov.access-count bank =
machine.training_card.master_bb.bank.regs
```

Each control register appears to have been written once:

```
┌─────┬───────────────────────┬──────┬────┬─────┐
│Row #│         Name          │Offset│Size│Count│
├─────┼───────────────────────┼──────┼────┼─────┤
│    1│enable                 │0x0000│   4│    1│
│    2│framebuffer_base_address│0x001c│   4│    1│
│    3│button_interrupt_control│0x0200│   4│    1│
│    4│display_i2c_base       │0x0800│   4│    1│
│    5│display_width          │0x0804│   4│    1│
│    6│display_height         │0x0808│   4│    1│
│    7│button_a_i2c_address   │0x0810│   4│    1│
│    8│button_b_i2c_address   │0x0814│   4│    1│
│    9│toggle_i2c_address[0]  │0x0850│   4│    1│
│   10│toggle_i2c_address[1]  │0x0854│   4│    1│
├─────┼───────────────────────┼──────┼────┼─────┤
│Sum  │                       │      │    │   10│
└─────┴───────────────────────┴──────┴────┴─────┘
```

14. **Run** the simulation.

```
simics> r
```

15. Display an example image, to see how that affects coverage. Go to the target **serial console**, and **`cat`** one of the example text files to the device:

```
# cat six.txt > /dev/simics_training_pcie_driver
```

16. Once the command completes, **pause** the simulation.

```
running> stop
```

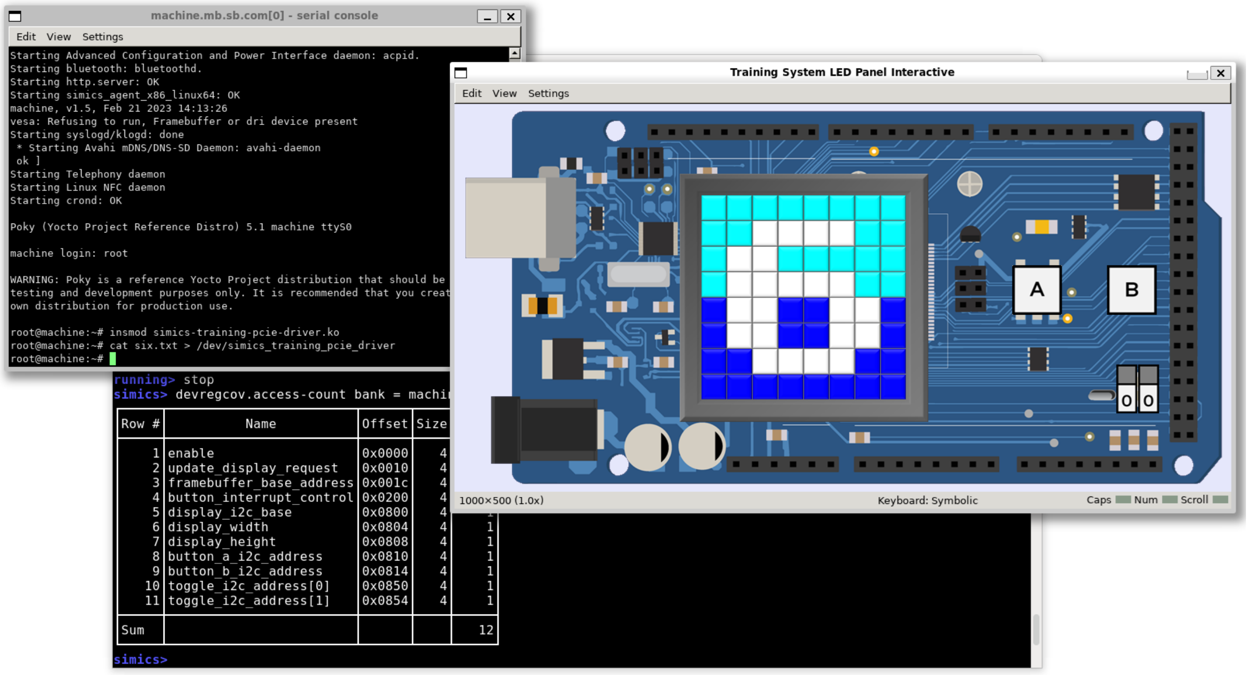
17. Check the current access count to the main control register bank:

```
simics> devregcov.access-count bank =
machine.training_card.master_bb.bank.regs
```

You should see one additional register having been accessed, **`update_display_request`**:

```
┌─────┬─────────────────────────┬──────┬────┬─────┐
│Row #│          Name           │Offset│Size│Count│
├─────┼─────────────────────────┼──────┼────┼─────┤
│    1│enable                   │   0x0│   4│    1│
│    2│update_display_request   │  0x10│   4│    1│
│    3│framebuffer_base_address │  0x1c│   4│    2│
│    4│button_interrupt_control │ 0x200│   4│    1│
│    5│display_i2c_base         │ 0x800│   4│    1│
│    6│display_width            │ 0x804│   4│    1│
│    7│display_height           │ 0x808│   4│    1│
│    8│button_a_i2c_address     │ 0x810│   4│    1│
│    9│button_b_i2c_address     │ 0x814│   4│    1│
│   10│toggle_i2c_address[0]    │ 0x850│   4│    1│
│   11│toggle_i2c_address[1]    │ 0x854│   4│    1│
├─────┼─────────────────────────┼──────┼────┼─────┤
│Sum  │                         │      │    │   12│
└─────┴─────────────────────────┴──────┴────┴─────┘
```

The simulator should look like this:



```
Starting Advanced Configuration and Power Interface daemon: acpid.
Starting bluetooth: bluetoothd.
Starting http.server: OK
Starting simics_agent_x86_linux64: OK
machine, v1.5, Feb 21 2023 14:13:26
vesa: Refusing to run, Framebuffer or dri device present
Starting syslogd/klogd: done
 * Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon
 ok ]
Starting Telephony daemon
Starting Linux NFC daemon
Starting crond: OK

Poky (Yocto Project Reference Distro) 5.1 machine ttyS0

machine login: root

WARNING: Poky is a reference Yocto Project distribution that should be
testing and development purposes only. It is recommended that you creat
own distribution for production use.

root@machine:~# insmod simics-training-pcie-driver.ko
root@machine:~# cat six.txt > /dev/simics_training_pcie_driver
root@machine:~#
```

```
running> stop
simics> devregcov.access-count bank = machi
┌─────┬────────────────────────┬──────┬────┬─────┐
│Row #│          Name          │Offset│Size│
├─────┼────────────────────────┼──────┼────┼─────┤
│    1│enable                  │0x0000│   4│
│    2│update_display_request  │0x0010│   4│
│    3│framebuffer_base_address│0x001c│   4│
│    4│button_interrupt_control│0x0200│   4│
│    5│display_i2c_base        │0x0800│   4│
│    6│display_width           │0x0804│   4│    1│
│    7│display_height          │0x0808│   4│    1│
│    8│button_a_i2c_address    │0x0810│   4│    1│
│    9│button_b_i2c_address    │0x0814│   4│    1│
│   10│toggle_i2c_address[0]   │0x0850│   4│    1│
│   11│toggle_i2c_address[1]   │0x0854│   4│    1│
├─────┼────────────────────────┼──────┼────┼─────┤
│Sum  │                        │      │    │   12│
└─────┴────────────────────────┴──────┴────┴─────┘
simics>
```

18. Next, try the button functionality to see if that can get a few more registers used, since it does use interrupts. **Run** the simulation.

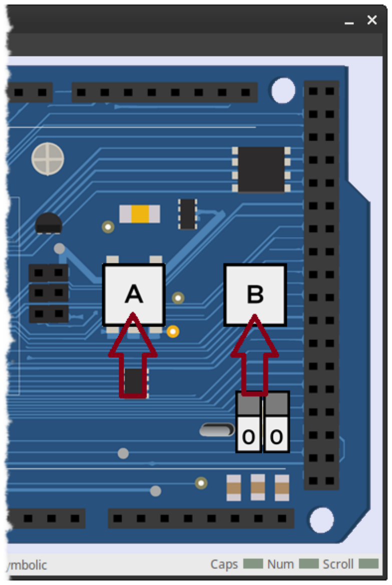
```
simics> r
```

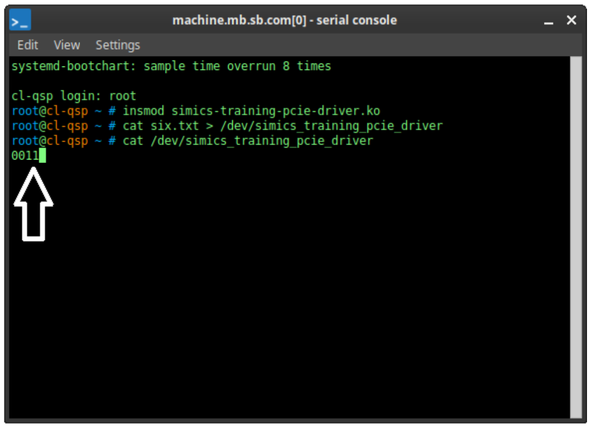
19. Go to the target **serial console**, and **cat** from the driver:

```
# cat /dev/simics_training_pcie_driver
```

20. Go to the **System Panel**. Click on button **A** twice and then on button **B** twice.

    This should print "**0011**" on the target **serial console**:





```
systemd-bootchart: sample time overrun 8 times

cl-qsp login: root
root@cl-qsp ~ # insmod simics-training-pcie-driver.ko
root@cl-qsp ~ # cat six.txt > /dev/simics_training_pcie_driver
root@cl-qsp ~ # cat /dev/simics_training_pcie_driver
0011
```

Alternatively, send signals to the button from the CLI:

```
running> @conf.machine.training_card.button_a.port.button_in.iface.signal.signal_raise()
running> @conf.machine.training_card.button_a.port.button_in.iface.signal.signal_lower()
running> @conf.machine.training_card.button_a.port.button_in.iface.signal.signal_raise()
running> @conf.machine.training_card.button_a.port.button_in.iface.signal.signal_lower()
running> @conf.machine.training_card.button_b.port.button_in.iface.signal.signal_raise()
running> @conf.machine.training_card.button_b.port.button_in.iface.signal.signal_lower()
running> @conf.machine.training_card.button_b.port.button_in.iface.signal.signal_raise()
running> @conf.machine.training_card.button_b.port.button_in.iface.signal.signal_lower()
```

21. **Pause** the simulation.

```
running> stop
```

22. Check if any of the other devices in the **training_card** have been accessed:

```
simics> devregcov.coverage
```

None of the other devices have seen any device accesses. This happens since the register banks in the I2C devices are used by the devices themselves for storage but are not accessible via memory operations from the processor. Device register coverage is designed to measure the hardware/software interaction, not how devices modify their own registers, or register accessed indirectly via channels like I2C.

23. Check the current access counts for the main control registers again:

```
simics> devregcov.access-count bank =
machine.training_card.master_bb.bank.regs
```

Note that two new registers have been accessed – the status registers for the buttons. This makes sense, since the driver must read these registers in order to report the pressed button:

```
┌─────┬─────────────────────────┬──────┬────┬─────┐
│Row #│          Name           │Offset│Size│Count│
├─────┼─────────────────────────┼──────┼────┼─────┤
│    1│enable                   │   0x0│   4│    1│
│    2│update_display_request   │  0x10│   4│    1│
│    3│framebuffer_base_address │  0x1c│   4│    2│
│    4│button_interrupt_control │ 0x200│   4│    1│
│    5│button_a_status          │ 0x204│   4│    6│
│    6│button_b_status          │ 0x208│   4│    4│
│    7│display_i2c_base         │ 0x800│   4│    1│
│    8│display_width            │ 0x804│   4│    1│
│    9│display_height           │ 0x808│   4│    1│
│   10│button_a_i2c_address     │ 0x810│   4│    1│
│   11│button_b_i2c_address     │ 0x814│   4│    1│
│   12│toggle_i2c_address[0]    │ 0x850│   4│    1│
│   13│toggle_i2c_address[1]    │ 0x854│   4│    1│
├─────┼─────────────────────────┼──────┼────┼─────┤
│Sum  │                         │      │    │   22│
└─────┴─────────────────────────┴──────┴────┴─────┘
```

24. Device register coverage can also separate read and write accesses. Check which registers have been read in the **regs** bank, using the **-read** flag:

```
simics> devregcov.access-count bank =
machine.training_card.master_bb.bank.regs -read
```

This shows that only two registers have been read:

```
┌─────┬───────────────┬──────┬────┬─────┐
│Row #│     Name      │Offset│Size│Count│
├─────┼───────────────┼──────┼────┼─────┤
│    1│button_a_status│0x0204│   4│    4│
│    2│button_b_status│0x0208│   4│    2│
├─────┼───────────────┼──────┼────┼─────┤
│Sum  │               │      │    │    6│
└─────┴───────────────┴──────┴────┴─────┘
```

25. Device accesses and tests can also be performed from the command line (as previously shown in Lab 003). Test how command-line accesses interact with device register coverage, by looking at the **toggle_status** registers that have not yet been accessed according to the coverage information collected.

The register is mapped at address **0x10000244** in the local memory space of the **training_card**. Check this using the **probe-address** command:

```
simics> probe-address obj = machine.training_card.local_memory
0x10000244
```

26. Read the current contents of the register (with a side-effect-causing read operation):

```
simics> machine.training_card.local_memory.read 0x10000244 4
```

27. Check the access count of the register bank again, specifically for reads:

```
simics> devregcov.access-count bank =
machine.training_card.master_bb.bank.regs -read
```

The register **toggle_status[0]** should now show up having been accessed:

```
┌─────┬─────────────────┬──────┬────┬─────┐
│Row #│      Name       │Offset│Size│Count│
├─────┼─────────────────┼──────┼────┼─────┤
│    1│button_a_status  │ 0x204│   4│    4│
│    2│button_b_status  │ 0x208│   4│    2│
│    3│toggle_status[0] │ 0x244│   4│    1│
├─────┼─────────────────┼──────┼────┼─────┤
│Sum  │                 │      │    │    7│
└─────┴─────────────────┴──────┴────┴─────┘
```

This indicates that device register coverage can also be used to evaluate Intel® Simics® model test benches that uses Intel® Simics® scripts to send memory accesses to device models.

28. The results of register coverage can also be saved for offline analysis and archiving. To do this, use the "**to-file**" option of the **coverage** command. This will save the data as a comma-separated values (CSV) file, which is easy to open and analyze in other tools like Microsoft Excel.

Save the coverage results from this lab to a csv file in your project:

```
simics> devregcov.coverage to-file = lab004-devregcov.csv
```

29. Open the resulting file in an editor of your choice check its contents.

30. This concludes the device register coverage lab. **Quit** this simulation session.

```
simics> quit
```

D. Optional. Serial port output capture ("logging")

Simulated serial consoles and graphics consoles in text mode support capturing output to file as well as for use in scripts. Note that this is often referred to as "logging" – but logging in the Intel Simics Simulator really refers to messages printed using the simulation log system. The correct term is **serial console capture**.

1. Start a new simulation session from the **`simics-user-training/001-qsp-training-setup`** `target`.

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

Capturing serial output to file

2. Once the simulation is set up and the command line available, enter the following command:

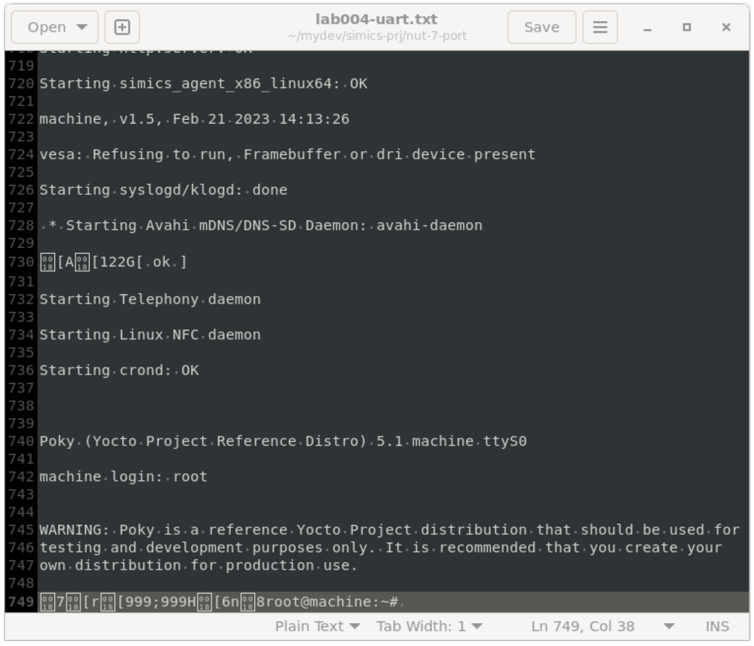
```
simics> machine.serconsole.con.capture-start lab004-uart.txt
```

   Note that if you do this lab multiple times, you can use **`-overwrite`** to get a new clean capture file each time.

3. **Run** the simulation until booted (and some more):

```
simics> run 20 s
```

4. Open the file **lab004-uart.txt** in an editor of your choice. Check that the capture output matches what is shown in the target serial console window. Note that the ANSI escape sequence characters are present in the output:



```

Starting simics_agent_x86_linux64: OK

machine, v1.5, Feb 21 2023 14:13:26

vesa: Refusing to run, Framebuffer or dri device present

Starting syslogd/klogd: done

 * Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon

␛[A␛[122G[ ok ]

Starting Telephony daemon

Starting Linux NFC daemon

Starting crond: OK



Poky (Yocto Project Reference Distro) 5.1 machine ttyS0

machine login: root


WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.

␛7␛[r␛[999;999H␛[6n␛8root@machine:~# 
```

Capturing serial output in command-line scripts

Serial port output can be retrieved by command line commands and from scripts and script branches. Test this from the command line, interactively rather than in a script file.

5. **Run** the simulation:

```
simics> r
```

6. Start collecting (recording) all output on the serial console for later retrieval, using the **`record-start`** command on the serial console.

```
running> machine.serconsole.con.record-start
```

7. Next, send an input to the target system serial console. Since we want to demonstrate control character filtering, the input is a bit complicated. So, you run a script to send it.

```
running> run-script "%simics%/targets/simics-user-training/004-
input.include"
```

   The serial console will show that you just input a printf command with some color control sequences.

8. Once the command has completed, **pause** the simulation.

```
simics> stop
```

9. Retrieve the recorded output.

```
simics> $x = (machine.serconsole.con.record-stop )
```

10. Check the output:

```
simics> $x
```

   The output will look like this:

```
"printf \"\\033[0;31mSome text in red.\\033[0;37m\\n\"\r\n\x1b[0;31mSome text in
red.\x1b[0;37m\r\nroot@machine:~# "
```

   Note how the recorded text includes all characters printed by the target operating system, including both newline (**`\n`**) and carriage return (**`\r`**) characters, as well as VT control sequences (starts with **`\x1b`**).

Using the log system to inspect serial port output

Output to serial consoles can be directed to the simulator command-line as well. This can be handy for debugging target system behavior, and when interacting with a serial console without looking at the window. One log message is printed for each completed line of output.

11. **Run** the simulation:

```
simics> r
```

12. Raise the log level of the serial console to 3.

```
simics> log-level machine.serconsole.con 3
```

13. Send an input to the target system serial console:

```
running> run-script "%simics%/targets/simics-user-training/004-
input.include"
```

The console will log the resulting serial output on the command line:

```
machine.serconsole.con info] root@machine:~# printf "\\033[0;31mSome text in
red.\\033[0;37m\\n"\r\n
[machine.serconsole.con info] \033[0;31mSome text in red.\033[0;37m\r\n
```

14. It is possible to remove the escape codes from the log output. Check the help on the attribute **filter_esc** on the console:

```
running> help machine.serconsole.con->filter_esc
```

15. Set the third item of the list to **TRUE**, to turn on filtering for log output.

```
running> machine.serconsole.con->filter_esc = [TRUE, TRUE, TRUE, FALSE]
```

16. Check the current filtering state using the status command on the console object:

```
running> machine.serconsole.con.status
```

There is a block of output indicating the current setup:

```
…
ANSI escape sequence filtering:
      Breakpoint matching : True
               CLI output : True
               Log output : True
                Elsewhere : False
…
```

17. Enter the command on the command-line again:

```
running> run-script "%simics%/targets/simics-user-training/004-input.
include"
```

The output is filtered

```
[machine.serconsole.con info] root@machine:~# printf "\\033[0;31mSome text in
red.\\033[0;37m\\n"\r\n
[machine.serconsole.con info] Some text in red.\r\n
```

Note that carriage return and newlines and similar control characters are still included. They are not ANSI escape sequences, but a core part of the output and input.

18. This concludes the serial port capture. **Quit** this simulation session.

```
running> quit
```

E. Optional: Collect execution statistics with instrumentation

The Intel Simics Simulator instrumentation feature provides a high-performance way to gather statistics on the code that is executed. You will experiment with instrumentation using one of the standard tools shipping with the Intel® Simics® Simulator. Note that users can also build their own tools, using the instrumentation API.

1. Start a new simulation session from the first checkpoint saved in Lab 001, before the **`insmod`** command has been executed on the target. Probably called **`MyFirstCheckpoint.ckpt`**.
2. Check out available Instrumentation commands using **`help`**:

   ```
   simics> help Instrumentation
   ```

3. To see the available instrumentation tools, use command **`list-instrumentation-tools`**.

   ```
   simics> list-instrumentation-tools
   ```

   One of the tools listed is the instruction histogram tool.

4. Create two new instruction histogram tools, and attach them to all processors in the system as part of their creation:

   ```
   simics> new-instruction-histogram -connect-all
   simics> new-instruction-histogram -connect-all
   ```

   This should give you two instrumentation tools, called **`ihist0`** and **`ihist1`**.

   Note that the simulator turns off VMP, since it is not possible to collect this detailed information while running using VMP. Instead, the simulator will use its JIT compiler to still run fast while providing instrumentation information.

5. Create two new instruction filters, to let us separate OS code and user code.

   ```
   simics> new-cpu-mode-filter name = os mode = supervisor
   simics> new-cpu-mode-filter name = user mode = user
   ```

6. Attach one filter to each tool:

   ```
   simics> ihist0.add-filter os
   simics> ihist1.add-filter user
   ```

7. Check the setup:

   ```
   simics> ihist0.status
   ```

   You should see **`ihist0`** being attached to both cores, and both connections having the "**`os`**" filter in place.

The above should result in a CLI session looking like this:

```
simics> new-instruction-histogram -connect-all
[machine.mb.cpu0.core[0][0] info] VMP not engaged. Reason: instrumentation enabled.
Created ihist0 (connected to 2 processors)
simics> new-instruction-histogram -connect-all
Created ihist1 (connected to 2 processors)
simics> new-cpu-mode-filter name = os mode = supervisor
Created filter os with mode supervisor
simics> new-cpu-mode-filter name = user mode = user
Created filter user with mode user
simics> ihist0.add-filter os
Added filter to 2 connections
simics> ihist1.add-filter user
Added filter to 2 connections
```

8. **Run** the simulation forward.
9. Go to the target **Serial Console** window and issue the command (just like several times before):

```
# insmod simics-training-pcie-driver.ko
```

10. Once the **`insmod`** command has completed, **pause** the simulation again. It will take a little longer than before due to the instrumentation, but not all that much.
11. Look at the instruction mix from the parts of the execution done in the operating system:

```
simics> ihist0.histogram
```

You should see a table showing the most common instructions executed by the operating system kernel. The details will vary depending on how quickly commands were entered, etc., since the input and run control was not scripted. It will look something like this (exact numbers will differ):

```
simics> ihist0.histogram
┌───────┬────────────┬──────────┬───────┬───────────┐
│ Row # │  mnemonic  │  Count   │Count% │Accumulated│
│       │            │          │       │  Count%   │
├───────┼────────────┼──────────┼───────┼───────────┤
│      1│mov         │29_728_962│ 30.58%│     30.58%│
│      2│push        │ 6_222_162│  6.40%│     36.99%│
│      3│pop         │ 5_835_156│  6.00%│     42.99%│
│      4│test        │ 5_719_931│  5.88%│     48.87%│
│      5│cmp         │ 5_449_929│  5.61%│     54.48%│
│      6│je          │ 4_382_485│  4.51%│     58.99%│
│      7│jne         │ 4_361_733│  4.49%│     63.48%│
│      8│add         │ 3_857_882│  3.97%│     67.45%│
│      9│hintnop(1f) │ 3_148_312│  3.24%│     70.68%│
│     10│call        │ 2_867_237│  2.95%│     73.63%│
│     11│ret         │ 2_804_859│  2.89%│     76.52%│
│     12│xor         │ 2_758_453│  2.84%│     79.36%│
…
```

12. Look at the output per instruction, in alphabetical order:

```
simics> ihist0.histogram sort-on-column = mnemonic
```

Which will result in something like this:

```
simics> ihist0.histogram sort-on-column = mnemonic
┌─────────┬───────────┬──────────┐
│  Row #  │  mnemonic │  Count   │
├─────────┼───────────┼──────────┤
│        1│adc        │     8_545│
│        2│add        │ 3_857_882│
│        3│and        │ 1_684_355│
│        4│bsf        │    19_908│
│        5│bsr        │    32_906│
│        6│bswap      │    10_323│
│        7│bt         │    94_962│
│        8│btr        │     8_331│
│        9│bts        │    11_007│
│       10│call       │ 2_867_237│
│       11│cdq        │        82│
...
```

13. Find the least commonly used instructions, by sorting on **Count** but in ascending order:

```
simics> ihist0.histogram sort-on-column = Count sort-order = ascending
```

You should see something like this:

```
simics> ihist0.histogram sort-on-column = Count sort-order = ascending
┌─────────┬──────────┬──────────┬───────┬───────────┐
│  Row #  │ mnemonic │  Count   │Count% │Accumulated│
│         │          │          │       │  Count%   │
├─────────┼──────────┼──────────┼───────┼───────────┤
│        1│setge     │         2│  0.00%│      0.00%│
│        2│clc       │         2│  0.00%│      0.00%│
│        3│cwde      │         4│  0.00%│      0.00%│
│        4│sets      │         4│  0.00%│      0.00%│
│        5│wait      │        11│  0.00%│      0.00%│
│        6│popcnt    │        55│  0.00%│      0.00%│
│        7│setle     │        75│  0.00%│      0.00%│
│        8│cdq       │        82│  0.00%│      0.00%│
...
```

14. Check out the table for user-mode instructions. It is typically a bit different in the specific instructions being the most common (except that **MOV** is almost invariably the most popular instruction on x86 no matter what code base is being run):

```
simics> ihist1.histogram
```

It could look something like this:

```
simics> ihist1.histogram
┌──────────┬───────────┬─────────┬───────┬───────────┐
│  Row #   │ mnemonic  │  Count  │Count% │Accumulated│
│          │           │         │       │  Count%   │
├──────────┼───────────┼─────────┼───────┼───────────┤
│         1│mov        │1_576_615│ 23.79%│     23.79%│
│         2│cmp        │  527_787│  7.96%│     31.75%│
│         3│test       │  423_854│  6.40%│     38.15%│
│         4│je         │  418_131│  6.31%│     44.46%│
│         5│add        │  415_592│  6.27%│     50.73%│
│         6│push       │  304_912│  4.60%│     55.33%│
│         7│jne        │  289_129│  4.36%│     59.69%│
│         8│pop        │  285_745│  4.31%│     64.00%│
│         9│sub        │  226_991│  3.43%│     67.43%│
│        10│lea        │  216_666│  3.27%│     70.70%│
│        11│jmp        │  141_957│  2.14%│     72.84%│
...
```

15. This concludes the instrumentation lab section.

    Quit the simulation session.

```
simics> quit
```

F. Optional. Intel Simics Simulator performance monitoring

Understanding the basics of simulator performance analysis is really helpful when things "seem to be slow". The key to performance improvement is to start by measuring what is going on, before moving on to analysis and adjustments. The simulator comes with several built-in performance analysis and debug tools, and this lab shows some of them.

1. **Start** a new simulation session from the **`simics-user-training/001-qsp-training`** `target`.

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

Check simulation setup

2. Check the overall execution settings for the simulator using the **`info`** and **`status`** command on the **`sim`** object:

```
simics> sim.info
```

   The info command provides information about the host, current module search path, and similar information that is static during a simulation session.

3. The status command shows more information related to how the simulator runs, in particular settings that can change during a session and as a result of user actions:

```
simics> sim.status
```

   It indicates whether particular technologies like multithreading and page sharing are enabled, and how many threads the simulator can use to run this particular setup.

Run system-perfmeter

4. On the command line, enter the following command to activate the **`system-perfmeter`**. Use the **`detailed`** mode and send the output to a file for later processing. In general, if you want to compare performance across runs or analyze it offline, it is recommended to send the **`system-perfmeter`** data to a file.

```
simics> system-perfmeter mode = detailed file = lab004-perfmeter.txt
```

5. **Run** through the boot and a bit more:

```
simics> run 25 s
```

6. **Wait** for the simulation to stop.

7. When the simulation has stopped, **open the performance log** (**`lab004-perfmeter.txt`**) in a text editor.

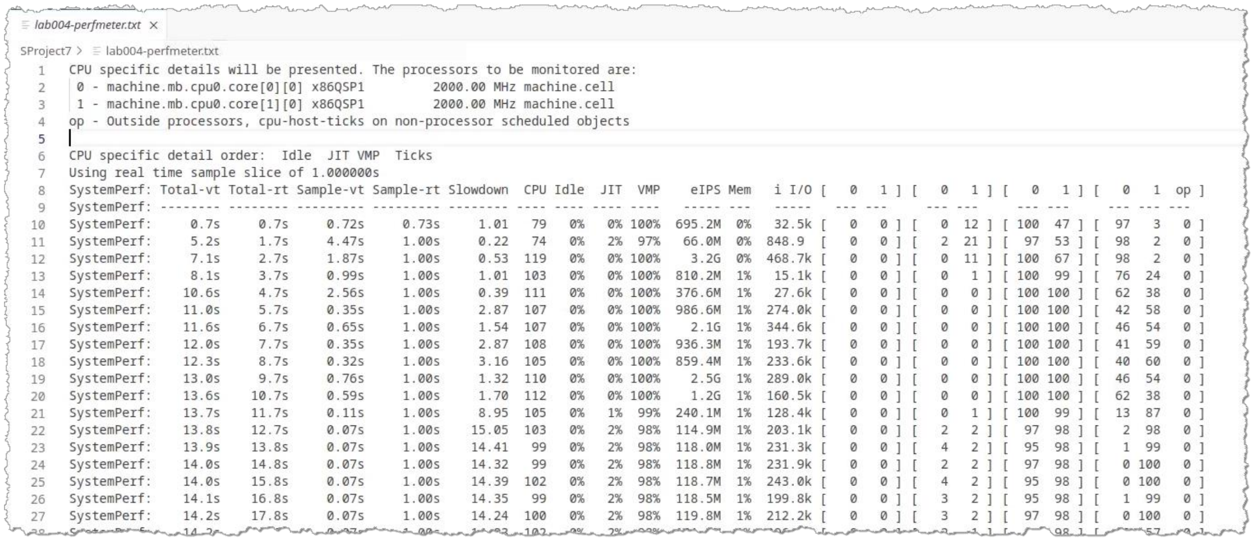
   There is a lot of information in the output.

8. At the very top, the file details the target system processors that it reports on:

```
CPU specific details will be presented. The processors to be monitored are:
 0 - machine.mb.cpu0.core[0][0] x86QSP1         2000.00 MHz machine.cell
 1 - machine.mb.cpu0.core[1][0] x86QSP1         2000.00 MHz machine.cell
CPU specific detail order:  Idle  JIT VMP  Ticks
Using real time sample slice of 1.000000s
```

This information is used to make sense of the samples that come later.

9. Next, there is a long list of sample points.



lab004-perfmeter.txt

SProject7 > lab004-perfmeter.txt

```
CPU specific details will be presented. The processors to be monitored are:
 0 - machine.mb.cpu0.core[0][0] x86QSP1         2000.00 MHz machine.cell
 1 - machine.mb.cpu0.core[1][0] x86QSP1         2000.00 MHz machine.cell
op - Outside processors, cpu-host-ticks on non-processor scheduled objects

CPU specific detail order:  Idle  JIT VMP  Ticks
Using real time sample slice of 1.000000s
SystemPerf: Total-vt Total-rt Sample-vt Sample-rt Slowdown  CPU Idle  JIT  VMP    eIPS Mem   i I/O [   0   1 ] [   0   1 ] [   0   1 ] [   0   1  op ]
SystemPerf: -------- -------- --------- --------- -------- ---- ---- ---- ----   ----- ---   -----   --- ---     --- ---     --- ---     --- --- ---
SystemPerf:     0.7s     0.7s     0.72s     0.73s     1.01   79   0%   0% 100%  695.2M  0%   32.5k [   0   0 ] [   0  12 ] [ 100  47 ] [  97   3   0 ]
SystemPerf:     5.2s     1.7s     4.47s     1.00s     0.22   74   0%   2%  97%   66.0M  0%  848.9  [   0   0 ] [   2  21 ] [  97  53 ] [  98   2   0 ]
SystemPerf:     7.1s     2.7s     1.87s     1.00s     0.53  119   0%   0% 100%    3.2G  0%  468.7k [   0   0 ] [   0  11 ] [ 100  67 ] [  98   2   0 ]
SystemPerf:     8.1s     3.7s     0.99s     1.00s     1.01  103   0%   0% 100%  810.2M  1%   15.1k [   0   0 ] [   0   1 ] [ 100  99 ] [  76  24   0 ]
SystemPerf:    10.6s     4.7s     2.56s     1.00s     0.39  111   0%   0% 100%  376.6M  1%   27.6k [   0   0 ] [   0   0 ] [ 100 100 ] [  62  38   0 ]
SystemPerf:    11.0s     5.7s     0.35s     1.00s     2.87  107   0%   0% 100%  986.6M  1%  274.0k [   0   0 ] [   0   0 ] [ 100 100 ] [  42  58   0 ]
SystemPerf:    11.6s     6.7s     0.65s     1.00s     1.54  107   0%   0% 100%    2.1G  1%  344.6k [   0   0 ] [   0   0 ] [ 100 100 ] [  46  54   0 ]
SystemPerf:    12.0s     7.7s     0.35s     1.00s     2.87  108   0%   0% 100%  936.3M  1%  193.7k [   0   0 ] [   0   0 ] [ 100 100 ] [  41  59   0 ]
SystemPerf:    12.3s     8.7s     0.32s     1.00s     3.16  105   0%   0% 100%  859.4M  1%  233.6k [   0   0 ] [   0   0 ] [ 100 100 ] [  40  60   0 ]
SystemPerf:    13.0s     9.7s     0.76s     1.00s     1.32  110   0%   0% 100%    2.5G  1%  289.0k [   0   0 ] [   0   0 ] [ 100 100 ] [  46  54   0 ]
SystemPerf:    13.6s    10.7s     0.59s     1.00s     1.70  112   0%   0% 100%    1.2G  1%  160.5k [   0   0 ] [   0   0 ] [ 100 100 ] [  62  38   0 ]
SystemPerf:    13.7s    11.7s     0.11s     1.00s     8.95  105   0%   1%  99%  240.1M  1%  128.4k [   0   0 ] [   0   1 ] [ 100  99 ] [  13  87   0 ]
SystemPerf:    13.8s    12.7s     0.07s     1.00s    15.05  103   0%   2%  98%  114.9M  1%  203.1k [   0   0 ] [   2   2 ] [  97  98 ] [   2  98   0 ]
SystemPerf:    13.9s    13.8s     0.07s     1.00s    14.41   99   0%   2%  98%  118.0M  1%  231.3k [   0   0 ] [   4   2 ] [  95  98 ] [   1  99   0 ]
SystemPerf:    14.0s    14.8s     0.07s     1.00s    14.32   99   0%   2%  98%  118.8M  1%  231.9k [   0   0 ] [   2   2 ] [  97  98 ] [   0 100   0 ]
SystemPerf:    14.0s    15.8s     0.07s     1.00s    14.39  102   0%   2%  98%  118.7M  1%  243.0k [   0   0 ] [   4   2 ] [  95  98 ] [   0 100   0 ]
SystemPerf:    14.1s    16.8s     0.07s     1.00s    14.35   99   0%   2%  98%  118.5M  1%  199.8k [   0   0 ] [   3   2 ] [  95  98 ] [   1  99   0 ]
SystemPerf:    14.2s    17.8s     0.07s     1.00s    14.24  100   0%   2%  98%  119.8M  1%  212.2k [   0   0 ] [   3   2 ] [  97  98 ] [   0 100   0 ]
```

The rate of sampling can either be set in virtual time or in real time (the standard modes use real time, as can be seen from the header above and the values in the columns).

Some key information in the output:

- The **Total-vt** and **Total-rt** columns show the total virtual time and real time so far for the run. The **Sample-vt** and **Sample-rt** show the total amount of virtual and real time for this particular sample.
- The **`Slowdown`** column shows the slowdown in this particular sample and serves as the primary indicator for Intel® Simics® performance overall. Note how performance varies during the boot – some parts execute way faster than real time (for example **`0.20`**, which means the simulation running at 5x real-time speed) while other parts are rather slow (**`3`** or higher). It depends on how hard it is for the Intel® Simics® simulator to simulate a particular part of the workload.
- The **`CPU`** columns shows how much host processing power the simulator is using. When it is above 100% the simulator is using more than one core, which can result from running multithreaded or by using additional threads to speed up the JIT compiler.
- The **`Idle`**, **`JIT`**, and **`VMP`** columns show the execution modes of the Intel® Simics® processors models during the sample. For Intel target processors, a higher

percentage in VMP rather than JIT generally indicates better performance. Note that the idle data is no longer generally reliable. This is a limitation in the reporting from the processors. In general, for Intel Architecture targets, a high VMP proportion is good (even if just means idle).

- The **`Mem`** column shows how much of the allocated memory for memory and disk images that the simulator is using. See lab 007 for more on Intel Simics images.
- The **`i I/O`** column shows the number of target instructions executed per IO operation (device access). The lower the number, the lower the expected performance of the simulation – it is a property of how the target software is written, and it is good to know what is going on when analyzing Intel Simics performance. Note that this measurement is currently (Q4 2024) not entirely accurate as it the simulator is migrating to "**`transaction`**" instead of "**`io_memory`**" for memory accesses.
- The three column groups following show the idle, JIT, and VMP percentages for each core. It is typically the case that only a few processors at a time are working, and that can be easily seen from the per-processor idle percentage (unless it is obscured by VMP).
- The last column group shows the percentage of host ticks spent on simulating each target processor. This indicates the load balance of the simulation run and can be useful to estimate the effectiveness of parallelization of a certain workload.

10. Scroll down until you find the **Performance Summary**. Typically, this is where you start your analysis of a workload. It aggregates the data over the entire run until the simulation is stopped.

```
SystemPerf: Performance summary:
--------------------------------
SystemPerf: Target:  2 CPUs in 1 cells [2]
SystemPerf: Running on: Simics build-id 7042 linux64 on 12 CPUs with 31801 MiB RAM
SystemPerf: Threads: 1 execution threads, 5 compilation threads
SystemPerf: Degrades performance: Real-time mode active
SystemPerf: Virtual (target) time elapsed:     25.00
SystemPerf: Real (host) time elapsed:          32.68
SystemPerf: Slowdown:                           1.31
SystemPerf: System execution idle:              0.00%
SystemPerf: System execution JIT:               0.25%
SystemPerf: System execution VMP:              99.63%
SystemPerf: System execution interpreter:       0.11%
SystemPerf: Host CPU utilization:              94.39%
SystemPerf: IPSe (without idle instr.):       484.59M
SystemPerf: Steps per I/O:                     72.93k
SystemPerf: Image memory limit hit (times):        0
SystemPerf: 0 - machine.mb.cpu0.core[0][0] (2000.00 MHz) Idle:   0% JIT:   0% VMP: 100%
Host ticks:      1651 ( 59.8%)
SystemPerf: 1 - machine.mb.cpu0.core[1][0] (2000.00 MHz) Idle:   0% JIT:   0% VMP: 100%
Host ticks:      1108 ( 40.2%)
```

When tuning simulation parameters, this is typically where you start.

11. Finally, the output shows the **Module CPU usage.** This is a rough estimate for where the simulator spent its time in terms of modules. It is common to see quite a bit of time in kernel libraries, as that is how VMP execution shows up.

    To get a better understanding for the profile of the code, apply a dedicated profiler to the Intel Simics Simulator process, such as Intel Vtune. They can provide much better information than the simulator's built in self-profiling.

12. **Quit** this simulation session.

```
simics> quit
```

Compare to no-VMP

The previous session assumed that your host had VMP enabled. If that is the case, it can be quite instructive to compare it to running without VMP. If you did not enable VMP, you should see numbers more like what is seen in this lab.

13. Start a new simulation session from the **`simics-user-training/001-qsp-training`** target.

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

14. Disable VMP, in case it was enabled:

```
simics> disable-vmp
```

15. Start the **`system-perfmeter`** tool, writing output to a different file:

```
simics> system-perfmeter mode = detailed file = lab004-no-vmp-
perfmeter.txt
```

16. **Run** through the boot and a bit more:

```
simics> run 25 s
```

17. **Wait** for the simulation to stop.

18. When the simulation has stopped, **open the performance log (`lab004-no-vmp-perfmeter.txt`)** in a text editor.

    Compare the overall execution time and slowdown to the VMP case – it is a lot slower.

19. **Quit** this simulation session.

```
simics> quit
```