Intel® Simics® Simulator Internals Training

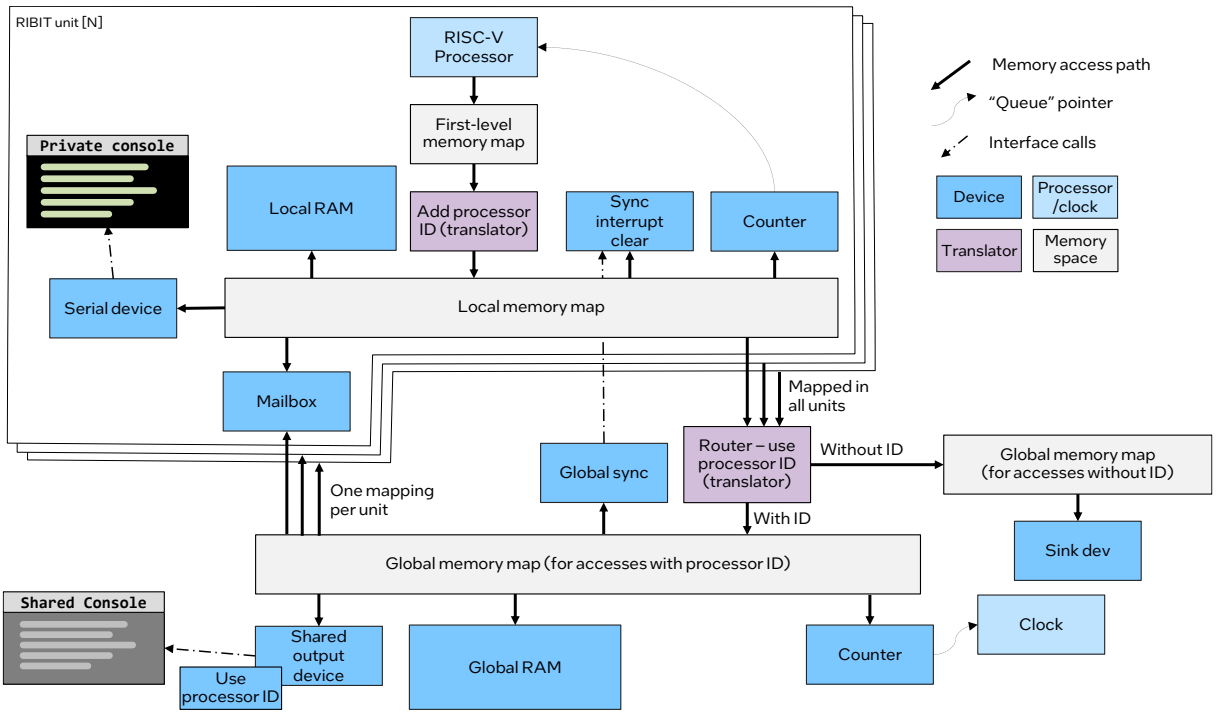
Lab 01-07

Memory Operations in the Intel Simics Simulator

1 Introduction

In the following labs, you will learn how to inspect and manipulate memory mappings, how stalling works for memory accesses, how memory access probing can be performed and what different access options for device registers exist.

The RIBIT system setup contains several examples of mechanisms and patterns borrowed from real systems, including the use of translators and a custom atom to route memory operations. The overall system architecture looks like this, showing the most important connections between the objects in the simulation:



Note the local memory map in each unit mapping the unit's local devices into the memory space accessed by the local processor. Accesses from the units to the shared global devices are handled through a router that inspects the processor ID and sends accesses that do not have an ID to a separate memory map.

It might help to refer back to this system map as you do the labs.

2 Inspecting Memory Maps

In this lab, you will inspect the initial state of memory maps in the internals training setup, see how they can be changed dynamically, and understand how transaction atoms can have an impact on memory accesses.

2.1 Start the simulation

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

```
[./]simics[.bat] simics-internals-training/07-memory-maps
```

2.2 Inspect the flattened memory map of a processor

2. Use the **memory-map** command to display the memory map of the processor of **ribit.unit[0]**:

```
simics> memory-map object = "ribit.unit[0].hart"
```

   You see the memory map from the point of view of the given object (a processor in this case). Note that it is possible (but rare) that translators present in the memory path have effects that the **memory-map** command does not understand, and thus the map might not be 100% correct.

3. Inspect the memory map of the processor in **ribit.unit[1]**:

```
simics> memory-map object = "ribit.unit[1].hart"
```

   At first glance the map looks the same (only that the devices in the map are in the namespace **unit[1]** instead of **unit[0]**). However, if you look closely you'll note that **unit[1]** has a map entry at address **0x10002010** pointing at a device in **unit[0]**, whereas **unit[0]** does not have it. The reason is that **unit[1]**'s mailbox is configured to send to **unit[0]**, while **unit[0]** is only receiving.

   The take-away is that one should always look very carefully at memory maps. The memory maps in different instance of the same subsystem can be different depending on their detailed configuration.

4. The **ribit** units can select the target of their mailbox messages through control registers of the mailbox. Change the destination of mailbox messages of **unit[1]**:

```
simics> write-device-reg "ribit.unit[1].mailbox.bank.c_regs.far_end_id"
3
```

   This will switch the mailbox target from **unit[0]** to **unit[2]**.

5. Inspect the memory map of **unit[1]** again:

```
simics> memory-map object = "ribit.unit[1].hart"
```

   You will notice that the mapping at **0x10002010** has switched from **unit[0]** to **unit[2]**. This illustrates that memory maps are not necessarily static. They can change over time, which makes it important to always double-check the state of a memory map when debugging or analyzing the system behavior.

2.3 Probing specific addresses

6. So far, you only saw the flattened map. That means, the command showed an address and the ultimate destination of accesses to that address. To see the actual path a transaction traverses, use the **probe-address** command.

```
simics> probe-address 0x010002010 obj = "ribit.unit[1].hart"
```

   You see (above the table) that first the address is translated from virtual to physical. This only happens for objects that are processors when address provided to the command is not prefixed with **p:**.

   The translated address is then applied to the **physical memory space** of the processor. All processors have a memory space that essentially represents the memory transactions leaving the processor (as physical addresses).

   You see that at the address the transaction hits is a translator (identified by the "**~**" in the Notes column) called **txn_decorator**. This one adds a few transaction atoms, the processor ID being the interesting one here, and then forwards to the **phys_mem** memory space of **unit[1]**. There, the transaction "travels" further (with its offset changing) towards the **id_router**, which sees a processor ID (as you can see in the column **Inspected Atoms**) and then redirects towards the global memory space for transactions with processor IDs which will then further send the transaction to the targeted mailbox registers.

   To really understand how the transaction flows through the system, let's follow this flow manually.

7. Use the **l2p** command to perform the virtual-to-physical translation. Note that this is evaluated using the current state of the MMU—it will generally not do any page table walks. Thus, the result from **l2p** can be that no translation is found.

```
simics> ribit.unit[1].hart.l2p 0x10002010
```

   You see that we have an identity mapping here.

8. Identify the physical memory space of the processor:

```
simics> ribit.unit[1].hart->physical_memory
```

   It uses **ribit.unit[1].core_mem** as its first-line memory space.

9. Look at the memory map of that memory space:

```
simics> ribit.unit[1].core_mem.map
```

   You will notice that there is nothing mapped. The memory space only has a default target, which is the **txn_decorator** translator. Thus all accesses will hit the **txn_decorator** object, where each transaction gets a processor ID atom added. The transaction address and data content of the transaction is not changed.

10. The **txn_decorator** device will forward any access to its **memory_space** connect. Check the value of that attribute:

```
simics> ribit.unit[1].txn_decorator->memory_space
```

You see that the decorator will forward to the **phys_mem** of **ribit.unit[1]**.

Note: Every translator is unique, and one needs to know how it works to understand what it does to transactions.

11. This particular translator is well-behaved and provides a nice **info** command:

```
simics> ribit.unit[1].txn_decorator.info
```

The output indicates the ID that it adds and where it sends transactions.

12. Inspect the map of **ribit.unit[1].phys_mem**:

```
simics> ribit.unit[1].phys_mem.map
```

This memory map has a bit more content.

Note that the address you are currently interested in (**0x10002010**) hits **ribit.unit[1].mailbox.far_end**. Since the **Offset** is **0x00**, our transaction will hit the **far_end** at (*address – Base + Offset*), where address=**0x10002010**. I.e., address zero in that memory map.

13. The **far_end** is a **memory-space** as well. It is a sub-object of the mailbox object since its mapping are controlled by that object. Inspect the map:

```
simcis> ribit.unit[1].mailbox.far_end.map
```

The mapping at base address **0x0** (which is where the transaction hits) points back to **phys_mem** (transactions can pass the same memory space multiple times in one access). The **Offset** of the mapping is **0x40000010**. Thus, the transaction goes back to **phys_mem** at **0x40000010**.

14. Look at the memory map of **phys_mem** again.

```
simics> ribit.unit[1].phys_mem.map
```

15. Another way to see what is in a certain location in a memory space is to use the **memory-map** command locally:

```
simics> memory-map ribit.unit[1].phys_mem -local
```

In any case, the address **0x40000010** will leave the **ribit[1]** subsystem and hit the system global **ribit.id_router**. This mapping has **Base=Offset**. This means that in the (*address – Base + Offset*) calculation, **Base** and **Offset** cancel out. Thus, the address of the transaction will remain unchanged – it will be sent with an address of **0x40000010**.

The **id_router** is another translator, so you need to know how it works (or use **probe-address** which sends a specific transaction through it to see where it ends up). We "know" that the **id_router** will direct transactions with IDs to **ribit.shared_mem_with_id** and those without to **ribit.shared_mem_without_id**.

16. Assuming we have an ID in the transaction, follow the flow:

```
simics> ribit.shared_mem_with_id.map
```

You see that address **0x40000010** will hit the bank **f_regs** of the **ribit.unit[2].mailbox** object.

Finally, this arrives at the same endpoint as the **probe-address** command. It is clear how much work the **probe-address** command can do for you!

Note: This memory setup may seem convoluted - and it was indeed constructed that way on purpose. However, it is not particularly complex compared to the memory map setups found in many virtual platforms for real hardware.

17. Addresses in any memory space can be probed. Use **probe-address** on **ribit.unit[1].phys_mem**.

```
simcis> probe-address 0x10002010 "ribit.unit[1].phys_mem"
```

The first few steps of the transaction look identical to the case above (starting from **ribit.unit[1].phys_mem**). However, the **id_router** redirects to the global memory space for transactions *without* IDs, eventually ending up in a "sink" device that terminates all memory accesses without doing anything.

Note that the **probe-address** output explicitly shows that the translator tried to read the **training_processor_id** atom but found it missing.

Transaction atoms (in this case the ID atom) can have a big impact on transaction routing. In the training setup, the ID atom is added by the **txn_decorator** and probing a point upstream of it will have it add the atom.

But what if the atom needed to steer the transaction was added by the processor model itself? In this case, any probing of a downstream memory space would not have the atom added.

To solve this problem, integer and enum-based transaction atoms can be provided as arguments to the **probe-address** command.

18. Repeat the previous probe, with a processor ID atom.

```
simics> probe-address 0x10002010 "ribit.unit[1].phys_mem" -add-atoms
ATOM_training_processor_id = 1
```

Note that this time the atom is found, and hence the transaction is routed to the memory space for transactions with IDs.

19. This concludes this part of the lab. Quit the simulation session.

```
simics> quit
```

3 Manipulating Memory Maps

After inspecting memory maps, we can start manipulating them. We will remove devices from maps, add them back and modify the properties of an existing mapping and observe the effects.

3.1 Removing and adding devices

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

```
[./]simics[.bat] simics-internals-training/07-memory-maps
```

2. Probe the address **0x10000000** (on the currently selected front-end processor)

```
simics> probe-address 0x10000000
```

   You see that this hits the serial port (UART) in unit 0.

3. Unmap the UART device from the **ribit.unit[0].phys_mem** memory space:

```
simics> ribit.unit[0].phys_mem.del-map device =
"ribit.unit[0].uart.bank.regs"
```

4. Probe the address again.

```
simics> probe-address 0x10000000
```

   You will notice that the access becomes a miss since there is no device at the given address anymore.

5. Add the device back to the map.

```
simics> ribit.unit[0].phys_mem.add-map device =
"ribit.unit[0].uart.bank.regs" base = 0x10000000 length = 0x11
```

   As you notice, removing a device usually only requires the device as an argument (unless it is mapped multiple times into the same memory space), but adding a device requires at least the device, its base address, and length of the mapping.

6. Finally, double-check that the mapping is working.

```
simics> probe-address 0x10000000
```

   You should see the same result that you got before you removed the mapping.

3.2 Changing properties of existing mappings

7. To change an aspect of an existing mapping, like its base address, length, or priority, you need to manipulate the **map** attribute of the memory space.

8. Look at the **map** attribute of the **ribit.unit[0].phys_mem**:

```
simics> ribit.unit[0].phys_mem->map
```

   You see a list of lists. Each sub-list describes one map entry.

9. To make it easier to digest, look at just one of the sub-lists of the **map** attribute:

```
simics> ribit.unit[0].phys_mem->map[2]
```

10. Use **help** on the attribute to get the details on the attribute format:

```
simics> help ribit.unit[0].phys_mem->map
```

Take some time to read the output to understand the makeup of the individual map entries of a memory space.

You see that the base is at index 0, the mapped object is at index 1, the offset is at index 3, and the length is at index 4.

11. Finding specific devices (and being able to tell at which map list index they are found) is tedious. Python can be used to automate the process:

```
simics> @map=conf.ribit.unit[0].phys_mem.map
simics> @obj=conf.ribit.unit[0].uart.bank.regs
simics> @[map.index(m) for m in map if m[1]==obj]
```

This code puts a reference to the **map** attribute and to the object to search for into Python variables to make the search line shorter. It finds all indices in the map that contain a mapping for the object (an object can be mapped many times).

You should only see a single index there. Note it down for the following exercise.

12. Another way to find the index is to use the output from the list-object-references CLI command. This command finds all references to a certain object, and for objects in memory maps, it points at the list index where they are found:

```
simics> list-object-references ribit.unit[0].uart.bank.regs
```

The output should show a single line, indicating that the register bank is found in the **map** attribute of **ribit.unit[0].phys_mem**. The **Attribute** is shown as **map[*X*][1]**, providing the same index as the Python search.

13. Change the base address of the UART register mapping by changing the contents of the **map** Python variable. As noted, the variable is a reference to the map attribute, not a copy of its contents. Modifying the Python variable will also modify the mapping in the device.

The ***X*** is the index you found in the previous step. The list element at index 0 of a map entry is the base address:

```
simics> @map[X][0]+=1
```

This will move the mapping one byte up in memory.

14. Reduce the length of the mapping by one byte (remember that index 4 is the length):

```
simics> @map[X][4]-=1
```

15. Set the offset to 1 byte:

```
simics> @map[X][3]=1
```

This has moved the mapping up by one byte, with the same end address as before. By setting the offset to 1, the same registers as before will be hit!

Before this change an access to 0x10000003 would arrive at the following address in the registered bank:

address-base+*offset*=**0x10000003**-**0x10000000**+**0**=**3**

With these changes it will still arrive at the same address:

address-base+*offset*=**0x10000003**-**0x10000001**+**1**=**3**.

16. Double-check the resulting map:

```
simics> ribit.unit[0].phys_mem.map
```

You should see the base address for the UART being shown as **0x10000001** with a length of **0x10** and an offset of **0x1**.

17. Run the simulation:

```
simics> run
```

The simulation will stop very soon with a missed access. To no surprise this is the address that we just removed from the UART's mapping by moving its base address (i.e., an access to **0x10000000**).

18. Repair the mapping in a way of your choice. You can either unmap the entire bank and remap it again to the correct base and length, or you undo the changes in the map attribute that you did earlier.

19. Double-check the resulting map:

```
simics> ribit.unit[0].phys_mem.map
```

The UART should now be mapped from **0x1000000**.

20. Run the simulation again:

```
simics> run
```

With a properly repaired mapping, there should be no further errors and all serial consoles should show output.

If you like, scroll upwards in the serial console of **ribit.unit[0]** and you will see that the first character of the output is missing. It was lost to the missed access.

21. This concludes this lab. Quit the simulation session.

```
simics> quit
```

4 Stalling Memory Accesses

Devices can stall the executing processor when they receive a memory access – which is often done as a performance tweak. For example, if code repeatedly reads a timestamp counter in order to wait for it to reach a certain value, then stalling the read can speed up the simulation because there are fewer reads being issued in the same amount of virtual time. The simulator ends spending more time just waiting and less processing code.

Adding stalling to a device is a modeling exercise, but the resulting stall ability can usually be enabled and controlled by end users. In this lab, you will look at how stalling is enabled in a device that has a stalling feature controlled via an attribute, and observe the effect on timing.

4.1 Observe the normal behavior

1. Start the simulation with the target for this lab.

```
[./]simics[.bat] simics-internals-training/07-memory-maps
```

2. Run the simulation forward.

```
simics> run 5 s
```

3. Inspect the **ribit.shared_out** serial console output.

   Scroll to the very top and look at the prints. You see messages in three different colors (one for each unit) and each color will show the same number of characters. The exact number of characters per colored print changes when all three have printed their share of characters, but it is always the same per iteration.

4. Quit the simulation.

```
simics> quit
```

4.2 Observe the effect of stalling

5. **Start** the simulation with the target for this lab:

```
[./]simics[.bat] simics-internals-training/07-memory-maps
```

6. **Inspect** the configuration attributes of the **ribit.shared_out** device:

```
simics> list-attributes ribit.shared_out
```

   There is a **stall_cycles** attribute that is used to set the stall cycles for each processor individually. The processor ID is retrieved from the memory operations, as you looked at in a previous lab.

7. **Set the stall cycles** for accesses from **ribit.unit[1]** to 1000 cycles.

```
simics> ribit.shared_out->stall_cycles[1]=1000
```

8. **Run** the simulator until all units have synchronized. This is just before the demo program starts the loop printing to the shared serial console.

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

9. **Check** the current time across all processors:

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

Note that the steps, cycles, and picosecond values are basically the same across all processors. They have stopped at a point where they have all finished their time quanta and have run the same code up until now. Nothing has been stalled so far.

10. **Run** the simulation forward until a few lines have been printed on the shared console. A simple way to stop at an interesting point is to leverage the output on the local console for unit 0:

```
simics> bp.console_string.run-until object =
"ribit.unit[0].console.con" string = "count 5"
```

Note that unit 1, the second color, prints fewer characters than the other two units. This results from the accesses to the shared serial port taking longer for unit 1, due to the **stall_cycles** setting. With each printout taking more time, fewer characters can be printed in the same amount of virtual time. The other two cores are not stalled when printing, providing a point of comparison.

Also, the output on the local serial console for unit 1 is behind the other units. It shows fewer lines of "**count x**". As it is taking longer to complete each line on the shared serial port.

11. **Check** the current time across all processors:

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

The processor **ribit.unit[0].hart** is slightly ahead in steps, cycles, and picoseconds since the breakpoint hits inside its time quantum.

Note that the processor **ribit.unit[1].hart** has run for the same number of cycles as **ribit.unit[2].hart**, but fewer steps. That is the effect of stalling: instructions, i.e., steps, take more virtual time to run, so that fewer can be run in the same amount of virtual time.

4.3 Single-step across stalling

Take a look at how time evolves as an instruction is stalled (or not), by single-stepping the code that writes to the shared serial port. To do this, it is necessary to find the code address of the write – you will do this without using debug information, just the hardware view of what the code does.

12. **Run** the simulation until the next write to the shared serial device, using a memory breakpoint as seen from the processor:

```
simics> bp.memory.run-until -w address = 0x030000000
```

The simulation will run until any of the units access the serial port. Strictly speaking, it is not known at this point which unit it will be (in the context of this lab with a given build, it will always be the same, but this lab is built to accept changes to the binary without breaking).

13. **Check** which processor hit the breakpoint:

```
simics> pselect
```

    This could be any of the units.

14. **Check** the current code location:

```
simics> da
```

    This is the instruction right *after* the printout to the shared serial device since the breakpoint stopped the execution after the device access was completed. To stop before the device access, i.e., when the instruction doing the shared serial port is about the execute, an instruction execution breakpoint on the store instruction is needed.

    Given how the simulator works, setting a breakpoint on the last byte of the previous instruction is enough. There is no need to figure out its starting address. Note that the last–byte trick works even for variable-length instruction sets.

15. Since we know that all cores execute the same code, regardless of the unit that just hit the breakpoint, we can now **set a breakpoint** on the byte before the current code location, to trigger once for only unit 0. **%pc** is a standard simulator way to read the current instruction or program counter regardless of what it is called in any particular architecture. Subtracting one byte gets to the end of the previous instruction, irrespective of where that starts:

```
simics> bp.memory.break ribit.unit[0].core_mem (%pc-1) -x -once
```

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

```
simics> r
```

17. **Check** the code at the current location in the code:

```
simics> da
```

    The next instruction to execute is a store byte instruction. No new character has appeared on the shared serial console, since the operation has not yet executed.

18. **Save** the address of this instruction, for later use:

```
simics> $i=(%pc)
```

19. **Check** the current time on this processor:

```
simics> print-time
```

20. **Step** a single instruction:

```
simics> step-instruction
```

    The write to the shared serial port completed, and a new character appeared on the console.

21. Check the current time on this processor again:

```
simics> print-time
```

    Both the step counter and cycles counter have incremented by one.

22. Since the code is the same on all processors, put a breakpoint on the same instruction address as before, but this time on unit 1, and run until it hits:

```
simics> bp.memory.run-until ribit.unit[1].core_mem $i -x
```

23. Repeat the print time, step, and print time operation. This can be expressed using the short forms of the commands on a single line:

```
simics> ptime; si; ptime
```

    Stepping across the store instruction causes the step count to increase by one. The cycle count increases by 1001. One cycle for the instruction execution, and 1000 cycles from the stalling.

    Note that when the simulator stops, the other two units have run another time quanta: executing 1001 cycles is enough to finish the current time quantum of the current processor, run the other two processors, and then get back to the current processor.

24. Step another instruction and check the time:

```
simics> si ; ptime
```

    The time progressed with one cycle for one instruction. The stalling affects only instructions accessing the shared serial port.

25. This concludes this section of the lab. Quit the simulation.

```
simics> quit
```

5 Side-effect vs Inquiry vs Attribute Accesses

Register accesses in device models can take different paths:

- **Memory path read/write with side effects**: This is what processors and other parts of the hardware do as part of simulating the hardware functionality. Accessing a register this way should trigger side effects.
- **Memory path inquiry read/write**: Inquiry operations take the same flow through the memory system as regular memory accesses. But they must not trigger side effects. They are effectively "debug accesses", where the currently stored value in memory and devices can be read or changed.
- **Register view inspection**: If a model provides information over the **`register_view`** interface, certain commands use it to get and set the current stored value of a register. These accesses must never have side effects.
- **Attribute get and set**: Attributes are used to save and load checkpoints. They access the current stored state of a register. Attribute accesses must never have side effects. Not all registers will have an attribute; for example, if a register has a value that can be computed from other state in the device, there is no need to checkpoint it on its own and it should not have an attribute.

In this section we will use a device that has log messages inserted to reflect different access paths.

Note: There are different ways how these accesses can be performed. We only show one way for each to show their difference in behavior.

Note 2; A word of caution: Models can be coded in such a way that they violate the distinction between these accesses. The simulator cannot enforce that different types of accesses follow the rules. Well-behaved models should follow the rules.

5.1 Start the simulation

1. Start the simulation with the target for this lab.

```
[./]simics[.bat] simics-internals-training/07-memory-maps
```

2. Set the log level on the shared serial port to 4.

```
simics> log-level ribit.shared_out 4
```

5.2 Memory path access with side effects

3. First, let's "remind" ourselves where the register bank of the shared serial port is mapped.

```
simics> devs ribit.shared_out.bank.regs
```

   You see that it is mapped into the memory space **`ribit.shared_mem_with_id`** at offset **`0x30000000`**.

4. Now print the device registers of the shared serial device, to remind yourself what the offset the character output register has in this bank.

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

As you can see, the register **char_out** is at offset **0x0**. So, an access to **0x30000000** in **ribit.shared_mem_with_id** will hit the shared output register.

5. **Write a value** into the shared serial port through the global memory space.

```
simics> ribit.shared_mem_with_id.write 0x30000000 0x42 size = 1
```

You see that the device reports that it got a transaction. The **write** method gets called, and it in turns calls the register **set** method to save the written value. This is an implementation choice – some **write** methods will modify the internal storage for a register directly without involving the **set** method.

After the stored value is changed, the side effects of the access are executed. In our case, this means that a character is written to the console. The character is seen in the **shared_out** serial console.

6. Check the attribute value of the register:

```
simics> ribit.shared_out.bank.regs->char_out
```

As expected, you see that the value is stored in the register.

7. And inspect it using the register view path:

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

5.3 Memory path inquiry access

8. Use the command-line **set** command on the memory space to invoke an inquiry access:

```
simics> ribit.shared_mem_with_id.set 0x30000000 0xAA size = 1
```

This access still comes in via the transaction path, but it is recognized as an inquiry access, and thus no side effects are executed. Only the **set** method is called. No character is printed on the serial console.

9. Double check the attribute value of the register

```
simics> ribit.shared_out.bank.regs->char_out
```

As expected, the register value stored in the device changed.

10. The register view also changed:

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

11. The get command on memory spaces is used to read values from memory. It is the logical companion to the **set** command:

```
simics> ribit.shared_mem_with_id.get 0x30000000 size = 1
```

12. Another command that uses inquiry accesses is the **examine-memory** or **x** command. Examine the single byte at **0x30000000**:

```
simics> ribit.shared_mem_with_id.x 0x30000000 size = 1
```

This should result in the same log output as the **get** command.

13. Try the **x** command on the whole mapped register area of the shared serial port:

```
simics> ribit.shared_mem_with_id.x 0x30000000 size = 16
```

Each byte is read in turn, using an inquiry access.

5.4 Attribute access

14. Change the register value with an attribute access:

```
simics> ribit.shared_out.bank.regs->char_out = 99
```

You see the log message from the **set** method. This is the only piece of code shared by all access paths. The attribute access does not involve the transaction processing code, instead it goes straight to the **set** method. No character is printed.

15. Check the attribute value and register view of the register:

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

5.5 Register without its own state

There are registers that do not have backing storage in the form of an attribute. Instead, they derive their values from other sources on reads or have side-effects on writes that do not modify their own value. There are endless variations.

One example is the **time_ps** register in the counter devices in the system.

16. **Run** the simulation for a while to get interesting values into the registers.

```
simics> bp.console_string.run-until "ribit.unit[0].console.con" "count
1"
```

A few accesses to the shared serial port will result in logs.

17. Look at the time across the simulation:

```
simics> list-processors -pico-seconds -all
```

Note that there are non-zero picoseconds on all processors.

18. Raise the log level on the counter in unit 0:

```
simics> log-level "ribit.unit[0].counter" 4
```

19. Read the register via the memory map:

```
simics> ribit.unit[0].phys_mem.read 0x10001010 8
```

20. Do an inquiry read via the memory map:

```
simics> ribit.unit[0].phys_mem.get 0x10001010 8
```

21. Print the registers of the device, triggering the register view flow:

```
simics> print-device-regs "ribit.unit[0].counter.bank.regs"
```

This will show a log message indicating that the register's **get** method was called. That is how DML implements register view.

22. Run the simulation one cycle forward:

```
simics> run 1 cycle
```

23. Read the register via the memory map again:

```
simics> ribit.unit[0].phys_mem.read 0x10001010 8
```

This returns an updated value, since the processor has executed and therefore changed the time that the device pulls and reports.

24. List the attributes of the device, including internal-marked registers. Restrict it to attributes called time-something:

```
simics> list-attributes ribit.unit[0].counter.bank.regs -i substr =
time
```

The attribute for the register is marked as **pseudo**, meaning that it is not involved in checkpointing. It can still be read.

25. Try setting the attribute to a new value:

```
simics> ribit.unit[0].counter.bank.regs->time_ps = 0x10
```

26. Read the register via the memory map:

```
simics> ribit.unit[0].phys_mem.read 0x10001010 8
```

It returns the picosecond count from the processor. The attribute set operation was quietly ignored.

27. This concludes the lab. Quit the simulation.

```
simics> quit
```

6 Endianness

Endianness is a question on how a sequence of bytes is interpreted when seen as an integer of a size bigger than one byte.

6.1 Start the simulation

1. Start a new session with the target for this lab:

```
[./]simics[.bat] simics-internals-training/07-endianness
```

2. Make sure the output is set to use hexadecimal, in units of single bytes:

```
simics> output-radix 16 2
```

3. Inspect the memory map of this setup:

```
simics> sys.mem.map
```

There are three mapped objects: RAM, a little-endian register bank (**regs_o_p**) and a big-endian register bank (**regs_o_p_be**). The register banks are identical, except their endianness.

6.2 Little-endian registers

Start with looking at how some little-endian registers function.

4. Check the register set of the little-endian bank:

```
simics> print-device-regs sys.emmd.bank.regs_o_p
```

Note that all default values are symmetrical – the same byte repeated over and over. This is not very useful to demonstrate endianness.

5. Set the value of **regs_o_p.a** to **0x01020304**:

```
simics> set-device-reg sys.emmd.bank.regs_o_p.a 0x01020304
```

6. Check the register values in the bank, again:

```
simics> print-device-regs sys.emmd.bank.regs_o_p
```

Note that the value is shown exactly as it was set.

7. Read the register value through the memory interface, by using the memory space read command:

```
simics> sys.mem.read 0x1000 4
```

This returns the value set into the register – noting that the interpretation is little-endian. The reason it defaults to LE is that the current processor is an LE processor.

```
simics> sys.mem.read 0x1000 4
0x01_02_03_04 (LE)
```

8. Read the first two bytes over the memory interface:

```
simics> sys.mem.read 0x1000 2
```

This returns **0x0304**, since little-endian means the least significant byte is stored first.

9. Set the value of **regs_o_p.b** to **0x05060708**:

```
simics> set-device-reg sys.emmd.bank.regs_o_p.b 0x05060708
```

10. Examine the memory view of both the registers, from the memory space:

```
simics> sys.mem.examine-memory 0x1000 8
```

Note how each of the two 32-bit registers start from their respective least-significant byte.

11. Read the two registers in a single 64-bit little-endian read. The **-l** flag to the command forces the interpretation of the bytes read to be little-endian:

```
simics> sys.mem.read 0x1000 8 -l
```

The resulting value is basically the inverse of the bytes being read with the examine memory command:

```
simics> sys.mem.read 0x1000 8 -l
0x05_06_07_08_01_02_03_04 (LE)
```

12. Write a single byte to the register bank, hitting the first mapped byte of register **c**:

```
simics> sys.mem.write 0x1008 0xff 1
```

13. Check the register values in the bank, again:

```
simics> print-device-regs sys.emmd.bank.regs_o_p
```

Note how the **0xff** byte shows up as the least-significant byte of register **c**.

6.3 Big-endian registers

It gets more interesting with big-endian values.

14. Check the register set of the big-endian bank:

```
simics> print-device-regs sys.emmd.bank.regs_o_p_be
```

Note that all default values are symmetrical – the same byte repeated over and over. This is not very useful to demonstrate endianness.

15. Set the value of register **a** to **0x01020304**:

```
simics> set-device-reg sys.emmd.bank.regs_o_p_be.a 0x01020304
```

16. Set the value of register **b** to **0x05060708**:

```
simics> set-device-reg sys.emmd.bank.regs_o_p_be.b 0x05060708
```

17. Check the register set of the big-endian bank again:

```
simics> print-device-regs sys.emmd.bank.regs_o_p_be
```

This looks exactly like the little-endian bank. The register inspection commands show the value of the register as an integer, not as a sequence of bytes. The endianness only makes itself visible when inspecting the registers over the memory map.

18. Read the value of register **a** through the memory interface, by using the memory space read command:

```
simics> sys.mem.read 0x2000 4
```

The default LE interpretation does not return the integer value that was written to the register:

```
simics> sys.mem.read 0x2000 4
0x04_03_02_01 (LE)
```

19. Read the register again, using a BE interpretation:

```
simics> sys.mem.read 0x2000 4 -b
```

This shows the integer that was written to the register.

20. Read the low two bytes of the register:

```
simics> sys.mem.read 0x2000 2 -b
```

This shows the two most significant bytes of the value:

```
simics> sys.mem.read 0x2000 2 -b
0x01_02 (BE)
```

21. Examine the memory view of both the registers, from the memory space:

```
simics> sys.mem.examine-memory 0x2000 8
```

With big-endian registers, the byte order on each 32-bit word is inverted compared to the little-endian case.

22. Read the two registers in a single 64-bit big-endian read:

```
simics> sys.mem.read 0x2000 8 -b
```

Note that the resulting value shows the bytes in the same order as the **examine-memory** command.

```
simics> sys.mem.examine-memory 0x2000 8
p:0x00002000  0102 0304 0506 0708                      ........
simics> sys.mem.read 0x2000 8 -b
0x01_02_03_04_05_06_07_08 (BE)
```

This is a property of the big-endian way to store values: any larger read from a certain location will result in an integer that matches what you get if you read the memory byte-by-byte.

23. Write a single byte to the register bank, hitting the first mapped byte of register **c**:

```
simics> sys.mem.write 0x2008 0xff 1
```

24. Check the register values in the bank, again:

```
simics> print-device-regs sys.emmd.bank.regs_o_p_be
```

Here, the **0xff** byte shows up as the most-significant byte of register **c**. This once again demonstrates that endianness is a property of the memory interface of the device.

6.4 Plain memory

Plain memory does not have any endianness per se. It is just a sequence of bytes.

25. Set the first bytes of the RAM mapped at **0x0000** to a string:

```
simics> sys.mem.write-string "abcdefgh" 0x0000
```

26. Examine the contents of RAM:

```
simics> sys.mem.examine-memory 0x0000 8
```

The string was put into RAM with the first character at address zero, the next at address one, etc.

27. Memory bytes can be read in any order – it depends on the device or processor doing the reading. The memory space **read** and **write** commands add the interpretation using the **-l** and **-b**. Interpret the first 4 bytes as a little-endian integer:

```
simics> sys.mem.read 0x0000 4 -l
```

28. Remember that it is possible to read memory at any offset for any size, and interpret the result as an integer. For example:

```
simics> sys.mem.read 0x0001 5 -b
```

29. Or little endian:

```
simics> sys.mem.read 0x0001 5 -l
```

30. Program the little-endian register bank mapped at **0x1000** in the same way as the memory:

```
simics> sys.mem.write-string "abcdefgh" 0x1000
simics> print-device-regs sys.emmd.bank.regs_o_p
```

Note how "**abcd**" ended up in the first register, and "**efgh**" in the second. Also, the terminating zero of the string changed the value of the low byte of register **c**.

Note how the value of register **a** is the same as what you got when reading the first four bytes of RAM using a little-endian interpretation.

31. Program the big-endian register bank mapped at **0x2000**:

```
simics> sys.mem.write-string "abcdefgh" 0x2000
simics> print-device-regs sys.emmd.bank.regs_o_p_be
```

Note how "**abcd**" and "**efgh**" ended up in the first and second registers, respectively, but in the opposite order in each register compared to the LE bank. This shows how endianness affects the interpretation of byte sequences written to a register. The terminating zero byte ends up as the highest byte of register **c**.

6.5 Examining memory with grouping

The examine-memory command can interpret the bytes from memories in groups, to produce integers from either a little-endian or big-endian view of the memory.

32. Examine the two big-endian registers as two 32-bit integers:

```
simics> sys.mem.examine-memory 0x2000 8 group-by = 32
```

33. Examine the two little-endian registers in the same way, adding the **-l** switch to tell the command to interpret the bytes using little-endian:

```
simics> sys.mem.examine-memory 0x1000 8 group-by = 32 -l
```

    This gives the same integers as reading each register directly, as each 32-bit word is now interpreted as a whole 32-bit value on its own.

34. Examine the RAM using 32-bit groupings:

```
simics> sys.mem.examine-memory 0x0000 8 group-by = 32
```

35. Or, using a little-endian interpretation:

```
simics> sys.mem.examine-memory 0x0000 8 group-by = 32 -l
```

    Basically, how sequences of bytes are interpreted is up to the beholder. Memory or sequences of bytes in general does not have any inherent endianness.

36. This concludes the lab. Quit the simulation session

```
simics> quit
```