Intel® Simics® Simulator Internals Training

Lab 01-05

Execution Model

1 Execution Scheduling: Temporal Decoupling

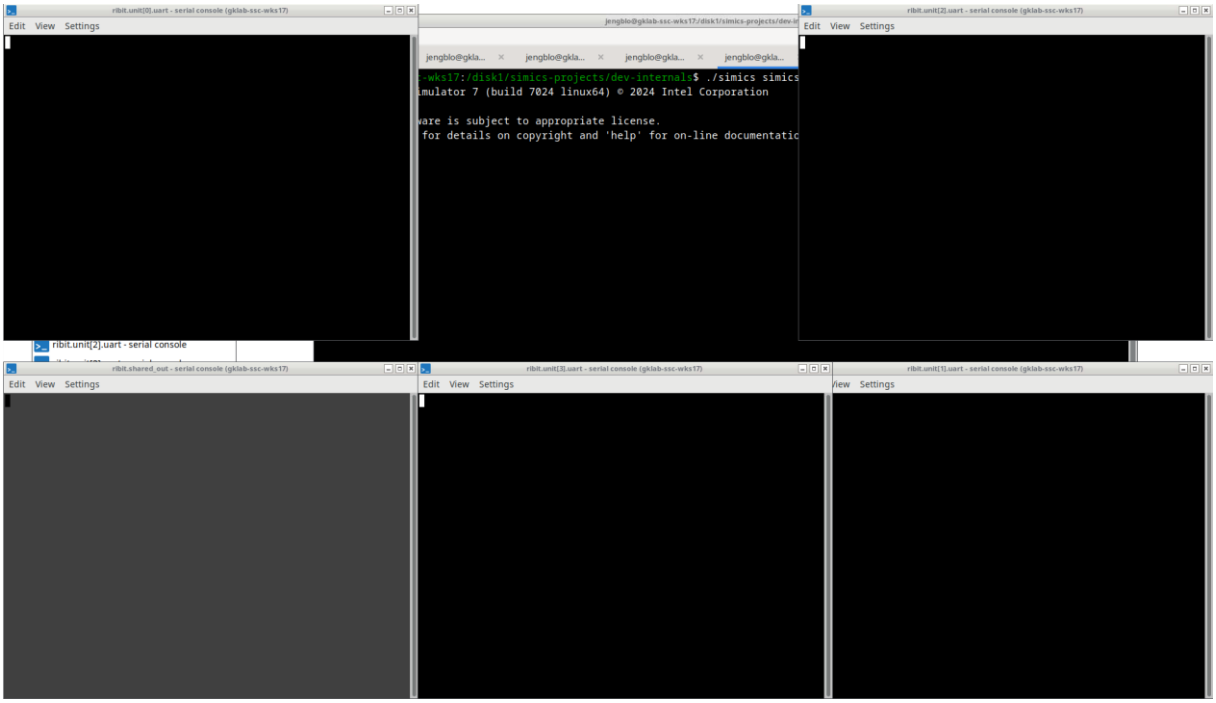
The basic execution model in the Intel® Simics® Simulator is temporal decoupling in a single execution thread. In this mode, each processor model runs a single time slice before yielding to the next processor in the schedule. It is also known as "round robin" execution since the execution rotates between the processors in the system.

1.1 Start a New Simulation Session

1. Start a new simulation session using the target **`simics-internals-training/05-time-quanta`**:

   ```
   $ ./simics simics-internals-training/05-time-quanta
   ```

   When the simulator has started, you should see five serial console windows in addition to the host shell where the simulator command line is active.



   The black consoles belong to the individual subsystems for each processor core. The output on the black consoles is printed from each processor individually.

   The dark gray console is attached to the shared output device where all subsystems can print. The text output to the shared console will be colored differently depending on the processor that printed the text. The console color control codes are added by the device itself – even if a subsystem prints just a single character it will be colored correctly.

   Each processor is running the same program. The program prints its subsystem ID repeatedly as it is running, providing a visualization of the length of a time quantum. The program synchronizes the start of the printout between the subsystem using the global synchronizer device.

1.2 Inspect the Temporal Decoupling Settings

2. **Inspect** the set of processors in the simulation:

```
simics> list-processors -all
```

   There should be four RISC-V processors and a clock. The clock object is used by the system-level shared devices.

3. **Inspect** the current time quantum:

```
simics> set-time-quantum
```

   This shows that the current time quantum is set to 10 microseconds, which works out to 1000 cycles for all processors and the clock since they all have the same simulated clock frequency.

4. **Inspect** the current threading mode:

```
simics> set-threading-mode
```

   This shows a single cell running in serialized mode, and thus all processors will interleave based on the time quantum.

1.3 View the Effect of Temporal Decoupling

The software on each processor will initialize its state and then wait for all processors to reach the same state. This synchronization is performed using the hardware device **`ribit.sync`**. This device maps a "decrement" register in the memory space of all the processors, which is written once by the software. After writing the register, the software stops waiting for an interrupt.

Once the sync device has received writes from all processors, it will issue an interrupt that is broadcast to all processors, releasing them to enter the program's main loop. The device also notifies the **`i-synchronizer-release`**, providing a simple hook for scripting to detect the synchronization of all the devices.

5. To run the simulation forward until all processors have initialized and reached the start of the main loop, set a breakpoint on the notifier:

```
simics> bp.notifier.break name = i-synchronizer-release -once
```

6. Run the simulation:

```
simics> r
```

7. When the simulation stops, check the current time and state of the processors:

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

All the processors should have the same cycle count, since the notifier was triggered using an event posted on the clock in the shared system. Something like this:

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

CPU Name		CPU Class	Freq	Cycles	Disassembly
ribit.clock		clock	100.00 MHz	286000	n/a
ribit.unit[0].hart	*	riscv-rv64	100.00 MHz	286000	v:0x000000000002036e p:0x000000000002036e lui a5,0x1101
ribit.unit[1].hart		riscv-rv64	100.00 MHz	286000	v:0x000000000002036e p:0x000000000002036e lui a5,0x1101
ribit.unit[2].hart		riscv-rv64	100.00 MHz	286000	v:0x000000000002036e p:0x000000000002036e lui a5,0x1101
ribit.unit[3].hart		riscv-rv64	100.00 MHz	286000	v:0x000000000002036e p:0x000000000002036e lui a5,0x1101

```
* = selected CPU
```

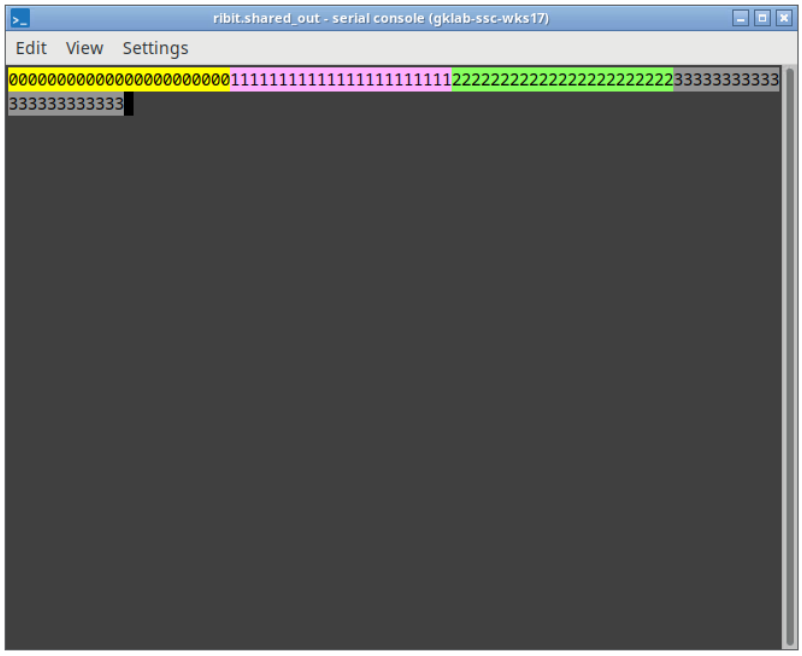
8. Make sure that processor number zero is selected.

```
simics> pselect ribit.unit[0].hart
```

9. **Run 1000 cycles**, exactly:

```
simics> run 1000 cycles
```

The shared console should show how processors 0 to 3 run one time quantum each, printing to the console. The time quantum is long enough for the target software to run several iterations of the main loop (exact pattern might differ in character counts):



10. **Check** the current time:

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

Note how the time is now 1000 cycles ahead on all the processors (but one quantum more on the clock as that runs before the first processor). The processors are also showing the same disassembly since they are all running the exact same software at the exact same speed. This is a highly artificial case.

11. **Run 2000 cycles** to go through two more time quanta.

```
simics> run 2000 cycles
```

The round-robin nature of the simulator scheduler is very obvious.

12. **Run half of a time quantum**:

```
simics> run 500 cycles
```

Note that the simulation stops partway through a sequence of zeroes. The simulator can stop at any point in a time quantum. In serial mode, the other processors will wait for their turn to run.

13. **Check** the current time:

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

Note that the clock is the furthest, followed by the current processor (zero) and the rest are behind since they have not yet started their execution in the current time quantum. The output should look something like this, focus on the cycle count for the first three clocks and processors

```
simics> list-processors -cycles -disassemble -all
┌──────────────────┬──┬─────────────┬──────────┬──────────┬──────────────────────────────────────────────────────────────────────────┐
│     CPU Name     │  │  CPU Class  │   Freq   │  Cycles  │                              Disassembly                                 │
├──────────────────┼──┼─────────────┼──────────┼──────────┼──────────────────────────────────────────────────────────────────────────┤
│ ribit.clock      │  │ clock       │100.00 MHz│ 290000   │n/a                                                                       │
│ ribit.unit[0].hart│* │ riscv-rv64 │100.00 MHz│ 289500   │v:0x00000000000000d6 p:0x00000000000000d6  sd      a5,-24(s0)             │
│ ribit.unit[1].hart│  │ riscv-rv64 │100.00 MHz│ 289000   │v:0x00000000000000e8 p:0x00000000000000e8  ld      a4,-32(s0)             │
│ ribit.unit[2].hart│  │ riscv-rv64 │100.00 MHz│ 289000   │v:0x00000000000000e8 p:0x00000000000000e8  ld      a4,-32(s0)             │
│ ribit.unit[3].hart│  │ riscv-rv64 │100.00 MHz│ 289000   │v:0x00000000000000e8 p:0x00000000000000e8  ld      a4,-32(s0)             │
└──────────────────┴──┴─────────────┴──────────┴──────────┴──────────────────────────────────────────────────────────────────────────┘
* = selected CPU
```

14. **Run 1000 cycles,** still with processor zero being the current processor.

```
simics> run 1000 cycles
```

This moves all processors one quantum forward and stops in the middle of the next time quantum for processor zero. It should be exactly 1000 cycles further on all processors and the clock.

15. **Check** the current time:

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

16. **Select processor two** as the current processor:

```
simics> pselect ribit.unit[2].hart
```

17. **Run 100 cycles** forward and check the time:

```
simics> run 100 cycles
simics> list-processors -cycles -disassemble -all
```

Note how this finishes the time quantum for processor zero, runs through processor one, and then stops in the time quantum of processor two. On the shared console,

the output from processor two is just beginning to be seen (exact pattern might differ in character counts):

That concludes this first demonstration of the Intel Simics Simulator scheduler.

18. **Quit** this simulator session:

```
simics> quit
```

1.4 Use a Shorter Time Quantum

19. **Start** a new simulation session using the target **`simics-internals-training/05-time-quanta`**:

```
$ ./simics simics-internals-training/05-time-quanta
```

20. Before running the simulation, **set the time quantum** to 100 cycles:

```
simics> set-time-quantum 100
```

21. **Run until the notifier** triggers, using **`run-until`** instead of a breakpoint:

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

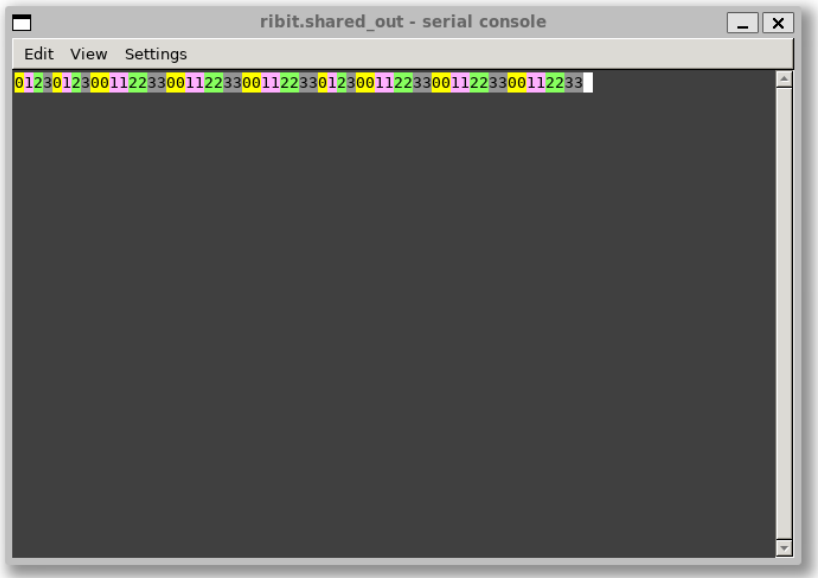
22. When the simulator stops, check the current time on the clock and processors:

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

23. **Run 1000 cycles:**

```
simics> run 1000 cycles
```

Notice how the pattern is very different from the previous run. The number of characters printed is precisely the same, but they are interleaved (exact pattern might differ in character counts):



Note that the number of characters printed by one processor in one quantum varies. This is expected since the software does not take an even divider of 100 cycles to get through an iteration.

24. **Change the time quantum** to 200:

```
simics> set-time-quantum 200
```

The time quantum length can be changed during a run. The changes take effect as soon as possible.

25. **Run 1000 cycles**:

```
simics> run 1000 cycles
```

As would be expected, this results in longer contiguous prints from each processor.

26. **Quit** this simulator session:

```
simics> quit
```

1.5 Different Processor Frequencies

Temporal decoupling respects the relative of processors. So far, all processors were set to the same speed, but what happens their clock frequencies are changed?

27. **Start** a new simulation session using the same target:

```
$ ./simics simics-internals-training/05-time-quanta
```

28. Before starting the simulation, **change the frequencies** of the processors.

```
simics> ribit.unit[0].hart->freq_mhz = 33
simics> ribit.unit[1].hart->freq_mhz = 90
simics> ribit.unit[2].hart->freq_mhz = 200
simics> ribit.unit[3].hart->freq_mhz = 250
```

These numbers have been chosen to have no simple common denominator. There is no requirement in the Intel Simics Simulator to have that – the scheduler will work out a schedule no matter what the frequencies are set to.

29. **Check** that the change has taken effect:

```
simics> list-processors -all
```

Which should show:

```
┌──────────────────┬──┬──────────┬──────────┐
│     CPU Name     │  │ CPU Class│   Freq   │
├──────────────────┼──┼──────────┼──────────┤
│ribit.clock       │  │clock     │100.00 MHz│
│ribit.unit[0].hart│* │riscv-rv64│ 33.00 MHz│
│ribit.unit[1].hart│  │riscv-rv64│ 90.00 MHz│
│ribit.unit[2].hart│  │riscv-rv64│200.00 MHz│
│ribit.unit[3].hart│  │riscv-rv64│250.00 MHz│
└──────────────────┴──┴──────────┴──────────┘
* = selected CPU
```

30. **Check** the length of the time quantum:

```
simics> set-time-quantum
```

The cycles per quantum differs between the processors. The time quantum is set to 1000 cycles on the first processor, which works out to 10 microseconds, and from that the number of cycles on the other processors is computed. Note that numbers are still quite even:

```
Current time quantum: 10.0 µs
┌──────────────┬──────────────────┐
│Cycles/quantum│      Clock       │
├──────────────┼──────────────────┤
│       1000.00│ribit.clock       │
│        330.00│ribit.unit[0].hart│
│        900.00│ribit.unit[1].hart│
│       2000.00│ribit.unit[2].hart│
│       2500.00│ribit.unit[3].hart│
└──────────────┴──────────────────┘
Default time quantum: 1000 cycles
```

31. **Run** until the notifier:

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

32. **Check the time** across the processors, including both steps and actual time:

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

This shows that all the processors have the same time in seconds. They have different cycle counts, since at different clock frequencies, they take different numbers of cycles to get to the same time. It might seem surprising that all the processors running code have the same step count, but they have run through the

exact same software and then gone to sleep waiting for an interrupt (exact numbers might differ).

```
| CPU Name          |   | CPU Class |    Freq    | Steps  | Cycles  | Time (s) |
|-------------------|---|-----------|------------|--------|---------|----------|
| ribit.clock       |   | clock     | 100.00 MHz | n/a    |  862000 |     0.01 |
| ribit.unit[0].hart | * | riscv-rv64 |  33.00 MHz | 284042 |  284460 |     0.01 |
| ribit.unit[1].hart |   | riscv-rv64 |  90.00 MHz | 284042 |  775800 |     0.01 |
| ribit.unit[2].hart |   | riscv-rv64 | 200.00 MHz | 284042 | 1724000 |     0.01 |
| ribit.unit[3].hart |   | riscv-rv64 | 250.00 MHz | 284042 | 2155000 |     0.01 |

* = selected CPU
```

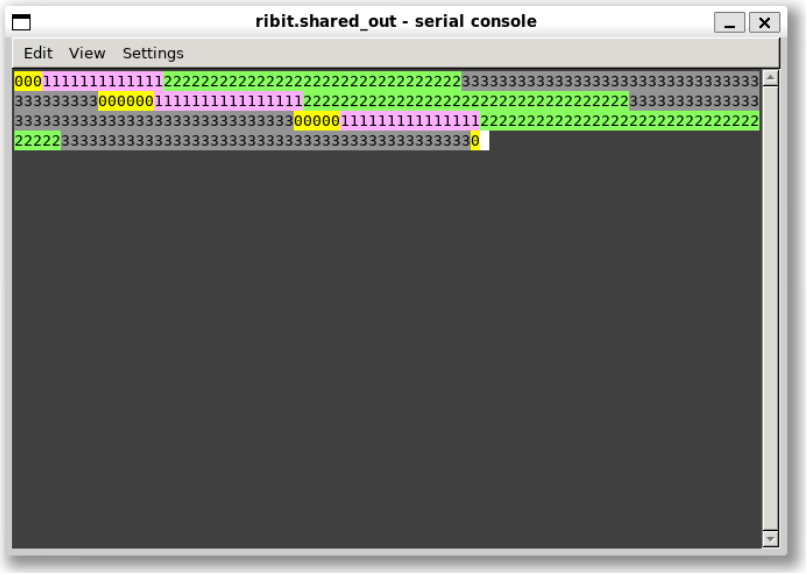
33. Make sure that **`ribit.unit[0].hart`** is the current processor:

```
simics> pselect ribit.unit[0].hart
```

34. Run 1000 cycles:

```
simics> run 1000 cycles
```

This produces a new pattern (exact pattern might differ in character counts):



First of all, many more characters are printed. Running 1000 cycles on processor zero works out to slightly more than three time quanta, which can be seen in the shared console output (three sequences of zeroes). Second, each processor prints a unique number of characters in each time slice, since they are running at unique speeds and thus run the print loop for a different number of iterations.

The key point is that the scheduler ensures that the relative speed of processors is taken into account when running the schedule.

35. Quit this simulator session:

```
simics> quit
```

1.6 Cycles per Instruction

Processor models can also set the number of cycles each instruction takes to run. The default is that every instruction takes one clock cycle, which works well in general. To make processors run at different relative speeds, this setting can be used instead of or in addition to the frequency. Since the frequency is visible to software and should typically follow the hardware design, changing the cycles-per-instruction setting can achieve changes to the relative rate of instruction execution while keeping frequencies unchanged.

36. **Start** a new simulation session using the same target:

```
$ ./simics simics-internals-training/05-time-quanta
```

37. Before starting the simulation, **change the frequencies** of some of the processors.

```
simics> ribit.unit[1].hart->freq_mhz = 50
simics> ribit.unit[2].hart->freq_mhz = 200
```

    This keeps processors zero and three at the default 100 MHz.

38. Set processor zero to use two cycles for each step. I.e., each instruction takes two cycles. This is done using the **set-step-rate** command, which is set in instructions per cycle:

```
simics> ribit.unit[0].hart.set-step-rate "1/2"
```

39. Set processor three to use half a cycle for each step, i.e., two instructions per cycle:

```
simics> ribit.unit[3].hart.set-step-rate 2
```

40. **Check** that the frequency changes have taken effect:

```
simics> list-processors -all
```

    Which should show:

```
┌──────────────────┬──┬─────────────┬──────────┐
│     CPU Name     │  │  CPU Class  │   Freq   │
├──────────────────┼──┼─────────────┼──────────┤
│ribit.clock       │  │clock        │100.00 MHz│
│ribit.unit[0].hart│* │riscv-rv64   │100.00 MHz│
│ribit.unit[1].hart│  │riscv-rv64   │ 50.00 MHz│
│ribit.unit[2].hart│  │riscv-rv64   │200.00 MHz│
│ribit.unit[3].hart│  │riscv-rv64   │100.00 MHz│
└──────────────────┴──┴─────────────┴──────────┘
* = selected CPU
```

41. The info command on the processors will show the cycles per instruction, i.e., the inverse of the step rate:

```
simics> ribit.unit[0].hart.info
simics> ribit.unit[3].hart.info
```

    Check that the quoted CPI is not one.

42. **Check** the length of the time quantum:

```
simics> set-time-quantum
```

The 100 MHz processors all have a quantum length of 1000 cycles, while the slow processor one has 500, and the fast processor two has 2000.

43. **Run** until the notifier:

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

44. **Check the time** across the processors, including both steps and actual time:

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

The result shows that the cycles depend on the clock frequency, while the number of steps is once again the same due to the "wait for everything to synch up" function in the software (exact numbers might differ).

CPU Name		CPU Class	Freq	Steps	Cycles	Time (s)
ribit.clock		clock	100.00 MHz	n/a	570000	0.01
ribit.unit[0].hart	*	riscv-rv64	100.00 MHz	284042	570000	0.01
ribit.unit[1].hart		riscv-rv64	50.00 MHz	284042	285000	0.01
ribit.unit[2].hart		riscv-rv64	200.00 MHz	284042	1140000	0.01
ribit.unit[3].hart		riscv-rv64	100.00 MHz	284042	570000	0.01

* = selected CPU

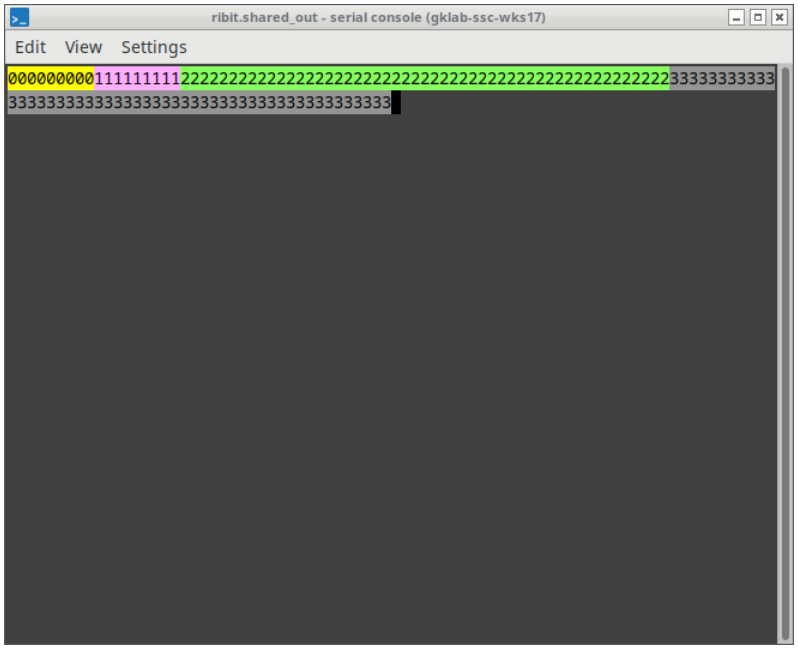
45. Make sure that **ribit.unit[0].hart** is the current processor:

```
simics> pselect ribit.unit[0].hart
```

46. **Run 1000 cycles:**

```
simics> run 1000 cycles
```

This produces a new pattern (exact pattern might differ in character counts):



The first two processors and the last two processors have made exactly the same amount of progress. Despite executing a different number of cycles.

47. **Check the time** across the processors, including both steps and actual time:

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

This shows that the number of steps is the same on each pair of processors, thanks to the step rate being set in such a way as to compensate for the difference in clock frequency.

```
┌──────────────────┬──┬─────────────┬──────────┬────────┬─────────┬──────────┐
│     CPU Name     │  │  CPU Class  │   Freq   │ Steps  │ Cycles  │ Time (s) │
├──────────────────┼──┼─────────────┼──────────┼────────┼─────────┼──────────┤
│ ribit.clock      │  │ clock       │100.00 MHz│ n/a    │  572000 │     0.01 │
│ ribit.unit[0].hart│ *│ riscv-rv64 │100.00 MHz│ 284542 │  571000 │     0.01 │
│ ribit.unit[1].hart│  │ riscv-rv64 │ 50.00 MHz│ 284542 │  285500 │     0.01 │
│ ribit.unit[2].hart│  │ riscv-rv64 │200.00 MHz│ 286042 │ 1142000 │     0.01 │
│ ribit.unit[3].hart│  │ riscv-rv64 │100.00 MHz│ 286042 │  571000 │     0.01 │
└──────────────────┴──┴─────────────┴──────────┴────────┴─────────┴──────────┘
* = selected CPU
```

48. **Run for 100 steps**, not cycles:

```
simics> run 100 steps
```

49. **Repeat** the command until you see all the other processors running through their time quanta to print on the serial console.

    It should take five “run 100 steps” to get out of the current time quantum. Thanks to the step rate of two cycles for every step.

50. **Quit** this simulator session:

```
simics> quit
```

This concludes the lab on how temporal decoupling works in a serial simulation.

2 Event Posting and Temporal Decoupling

This lab shows how event posting across multiple processors and clocks works when running serially using temporal decoupling.

2.1 Start a New Simulation Session

1. Start a new simulation session using the target **`simics-internals-training/05-events`**:

   ```
   $ ./simics simics-internals-training/05-events
   ```

   When the setup is complete, you should see two serial consoles for individual units and the shared console. The shared console will not be used in this lab.

2.2 Investigate the Hardware Setup

The lab makes use of a set of hardware counter devices tied to different clocks in the system. The software will loop, waiting for the counter devices to count up, and print a new message each time they notice a change.

2. Find all the counter devices:

   ```
   simics> list-objects class = i_counter -class-desc
   ```

   This finds three devices. One in each unit, and one global.

3. Check the clocks that the counter devices are connected to, using their helpful **`info`** commands. Start with the global counter in the **`ribit`** namespace:

   ```
   simics> ribit.counter.info
   ```

4. If a device lacks helpful commands, another way to find the associated clock is to use the simulator API:

   ```
   simics> @SIM_object_clock(conf.ribit.counter)
   ```

5. The clock is also pointed to by the standard queue attribute present in all objects:

   ```
   simics> ribit.counter->queue
   ```

6. Every clock also has an associated **`vtime`** hierarchy that breaks out the picosecond and cycle queues and allow posting on picoseconds and inspecting the queues.

   ```
   simics> list-objects namespace = ribit.clock -tree -show-port-objects
   ```

7. The picosecond queue is found using another API call:

   ```
   simics> @SIM_picosecond_clock(conf.ribit.counter)
   ```

   This should agree with the output of the **`info`** command above.

8. Check the other two counters:

   ```
   simics> ribit.unit[0].counter.info
   simics> ribit.unit[1].counter.info
   ```

   They are each connected to the processor in their subsystem.

9. These processors also have **vtime** hierarchies in the same shape as the global clock.

```
simics> list-objects namespace = ribit.unit[0].hart -show-port-objects
-tree
```

And quite a few other ports, since these are functional processors and not just simple clocks.

10. Check the speed and current time of all the clocks and processors in the configuration:

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

To make the lab more logical, the clock is set to the same frequency as the processors. Everything is at time zero, since the simulation has not been run yet.

11. The global counter is mapped to both processors, while the local counter for each is only mapped to one processor. To see this, check the memory map of the two processors:

```
simics> memory-map "ribit.unit[0].hart"
simics> memory-map "ribit.unit[1].hart"
```

This should show a local counter device mapped in the area just beyond **0x1000_0000** and the global counter mapped to both processors at the same local address (above **0x3000_0000**).

2.3 Event Posting

The counter devices are controlled by the software. The software used in this lab programs the counters as it starts.

12. Show the programming registers of the global counter:

```
simics> print-device-regs ribit.counter -description
```

Nothing has been programmed, so all registers are zero.

Note the **time_ps** register that reports the current time in picoseconds, as seen from each device.

13. Get the software stack started. **Run** until the synchronizer notifier hits, just like previously.

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

14. **Check the current time** on all processors. The primary time to use will be cycles, since the counter increments are scheduled in terms of cycles.

```
simics> list-processors  -all -cycles
```

The currently selected processor should be **ribit.unit[0].hart**.

15. Check the current time quantum.

```
simics> set-time-quantum
```

This lab uses a 100000-cycle time quantum.

16. **Run** 10000 cycles, to let the software program the counters.

```
simics> run 10000 cycles
```

17. **Check the current time** on all processors.

```
simics> list-processors -all -cycles
```

The current temporal decoupling state is that the global clock has finished its time quantum, while unit 0 is some distance into its time quantum. This is the same effect as seen above, that the **run** command controls the selected processor.

18. **Check** the status on all counter devices:

```
simics> ribit.unit[0].counter.status
simics> ribit.unit[1].counter.status
simics> ribit.counter.status
```

The counter in **unit[0]** and the global counters have been programmed with an interval of 10000. The counter in **unit[1]** is not programmed, since that processor has yet to run any code after the sync.

The counter in **unit[0]** is close to triggering its first event, while the global counter shows that its next event is in 10000 cycles.

19. The events are posted to the event queues of the clocks and processors in the simulator.

Investigate the state of the event queues. Start with unit 0.

```
simics> print-event-queue "ribit.unit[0].hart"
```

This should show two events on the cycle queue. Something like this:

Cycle	Object	Description
170	ribit.unit[0].counter	callback.tick
7205759403792761503	ribit.unit[0].clint	timeout[0]

The counter device posted the first event. The time shown is relative to the current time in the processor.

20. Investigate the event queue in unit 1:

```
simics> print-event-queue "ribit.unit[1].hart"
```

This does not show any event from its counter device, since the code to start the counter has not yet been run, and thus no event posted.

21. Investigate the event queue of the global clock:

```
simics> print-event-queue "ribit.clock"
```

This shows a single event, posted at 10000 cycles into the future:

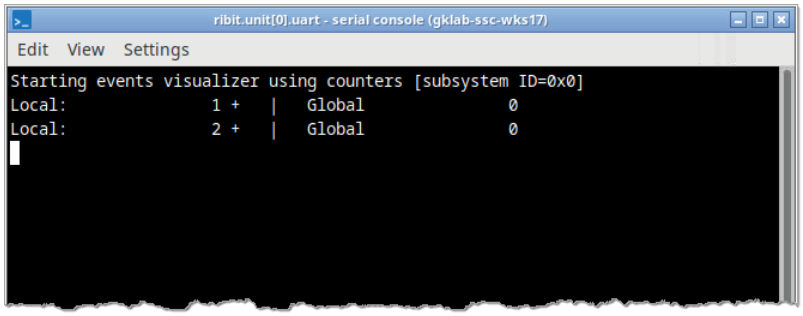
Cycle	Object	Description
10000	ribit.counter	callback.tick

This event will trigger when the global clock runs its time quantum.

22. Run the simulation forward for 20000 cycles.

```
simics> run 20000 cycles
```

This should result in two lines of output on the unit 0 serial console.



The software runs in a polling loop waiting for either the local or global counter to change. When a change is seen, it prints the values of both the local and global counters. The **+** sign marks that the value changed.

23. Check the event queue of the unit 0 processor again:

```
simics> print-event-queue "ribit.unit[0].hart"
```

This shows essentially the same queue state as before: a single event posted at the next instance that the counter will update, plus a far-future timeout. The device model posts a new event each time an event triggers, at 10000 processor cycles into the future.

24. Check the contents of the **ribit.unit[0].hart.vtime.cycles** queue object of the processor:

```
simics> print-event-queue ribit.unit[0].hart.vtime.cycles
```

This is essentially a different name for the same queue as you see when inspecting the processor object directly.

25. Check the contents of the **ribit.unit[0].hart.vtime.ps** queue object:

```
simics> print-event-queue ribit.unit[0].hart.vtime.ps
```

This contains a single event, the end of the time quantum for the processor. It is posted to the picosecond queue since time quanta are ultimately expressed in virtual time, not in cycle counts.

Cycle	Object	Description
700000000	ribit.cell	Time Quantum End

26. The Time Quantum End event is also visible on the processor object, if you tell the command to show **internal** events with the **-i** flag:

```
simics> print-event-queue ribit.unit[0].hart -i
```

This will show two tables: one with the cycle-posted events and one with the ps-posted events.

27. Raise the log level of all the counters, to see internal activity in the devices:

```
simics> log-level class = i_counter 3
```

28. Make sure log messages are time stamped with the local time when the logs are printed:

```
simics> log-setup -time-stamp
```

29. Run the simulation forward, until just before the event should trigger. The point in time can be obtained by looking at the output from the **print-event-queue** command.

```
simics> run NNN-1 cycles
```

It can also be computed in Python by pulling out the contents of the cycle event queue and digging into the returned list, and subtracting one. Like this:

```
simics> run (python
"conf.ribit.unit[0].hart.vtime.cycles.iface.cycle_event.events()[0][2]-
1") cycles
```

30. When the simulation stops, check that it is right before the event triggers:

```
simics> print-event-queue
```

Which should show:

Cycle	Object	Description
1	ribit.unit[0].counter	callback.tick

…

31. Check the status on the counter:

```
simics> ribit.unit[0].counter.status
```

It should also indicate that the next event happens in 1 cycle.

32. Run forward exactly 1 cycle:

```
simics> run 1 cycles
```

The device will print log messages indicating that it ticked, and that it is posting a new event in 10000 cycles from the current time.

33. Check the new event queue contents:

```
simics> print-event-queue
```

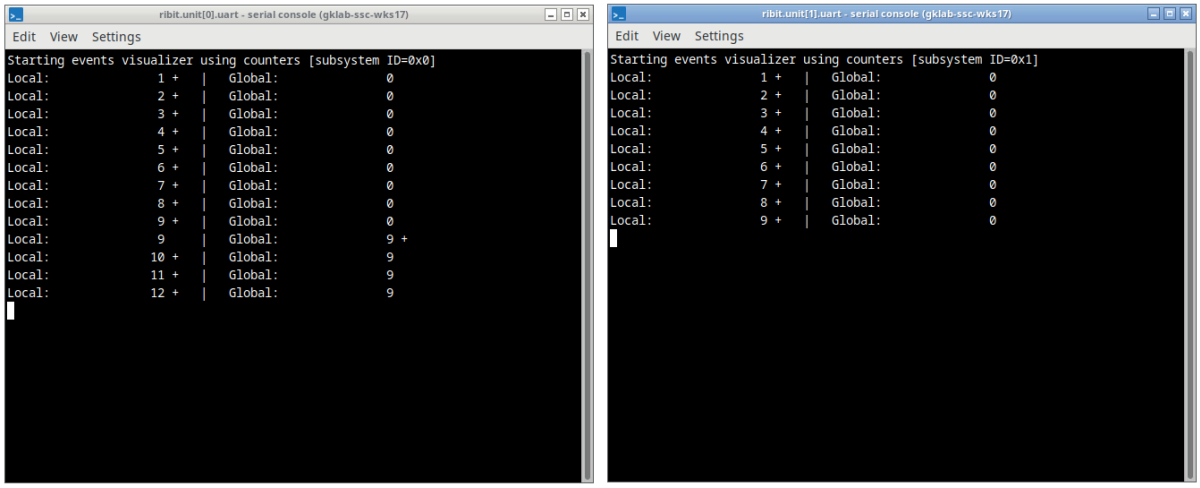
The next counter event is now 10000 cycles into the future.

34. Run the simulation forward for 100000 cycles, i.e., a single time quantum:

```
simics> run 100000 cycles
```

This lets the processor in unit 1 run code as well.

The two serial consoles show similar output (but more lines on unit 0 since the execution stops some distance into its time quantum):



Note the behavior of the global counter. It jumps from 0 to 9, at the point where unit 0 starts its second time quantum. The reason is that all of its counter updates are run in the time quantum of the global clock when no processor is running code that observes the updates.

35. The global counter updates are visible from the log messages printed during the run – **look back** a bit in the simulator output to find lines like this:

```
…
[ribit.counter info] {ribit.clock 470000} Tick (new counter value=7)
[ribit.counter info] {ribit.clock 470000} Posting event in 10000 cycles
[ribit.counter info] {ribit.clock 480000} Tick (new counter value=8)
[ribit.counter info] {ribit.clock 480000} Posting event in 10000 cycles
[ribit.counter info] {ribit.clock 490000} Tick (new counter value=9)
[ribit.counter info] {ribit.clock 490000} Posting event in 10000 cycles
…
```

The number printed next to the name of the processor is the current cycle count. For the processors, there are two numbers: the first is the current instruction pointer at the time of log, the second is the cycle count. Like this:

```
[ribit.unit[0].counter info] {ribit.unit[0].hart 0x22c88 400170} Tick (new counter
value=10)
```

2.4 Changing the Queue

A device's **queue** can be changed to make it post events on and read device time from another clock or processor. This is usually not done at runtime, but it can be.

36. **Check the current time** across the system:

```
simics> list-processors -cycles -all
```

Note the cycle count on the global clock and the unit 0 processor. The global clock should be ahead, since it has finished its time quantum and we are currently running in the quantum of the processor in unit 0.

37. **Check the time** reported by the global counter through its programming registers:

```
simics> output-radix 10 3
simics> print-device-regs ribit.counter
```

This shows a current time agreeing with the time of **`ribit.clock`**.

38. **Change** the **`queue`** attribute of the global counter to point at the processor in unit 0:

```
simics> ribit.counter->queue = ribit.unit[0].hart
```

39. Check that this has the intended effect:

```
simics> ribit.counter.info
```

The output should indicate that the counter now uses **`ribit.unit[0].hart`** as its "standard clock".

40. **Check the time** reported by the global counter through its programming registers again:

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

Now, the **`time_ps`** register reports a different value, corresponding to the time of **`ribit.unit[0].hart`**. The register implementation simply pulls the time from the designated queue, and changing the queue changes the value.

41. The local counter in unit 0 already uses the unit 0 processor. Check its programming registers:

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

The value of **time_ps** is the same as that seen in the global counter.

42. Changing the queue attribute does not affect already posted events (unless the device code takes some very specific actions on such a change, but that is not the common case).

Checking the event queue on the global clock shows that it still has an event scheduled from the global counter:

```
simics> print-event-queue ribit.clock
```

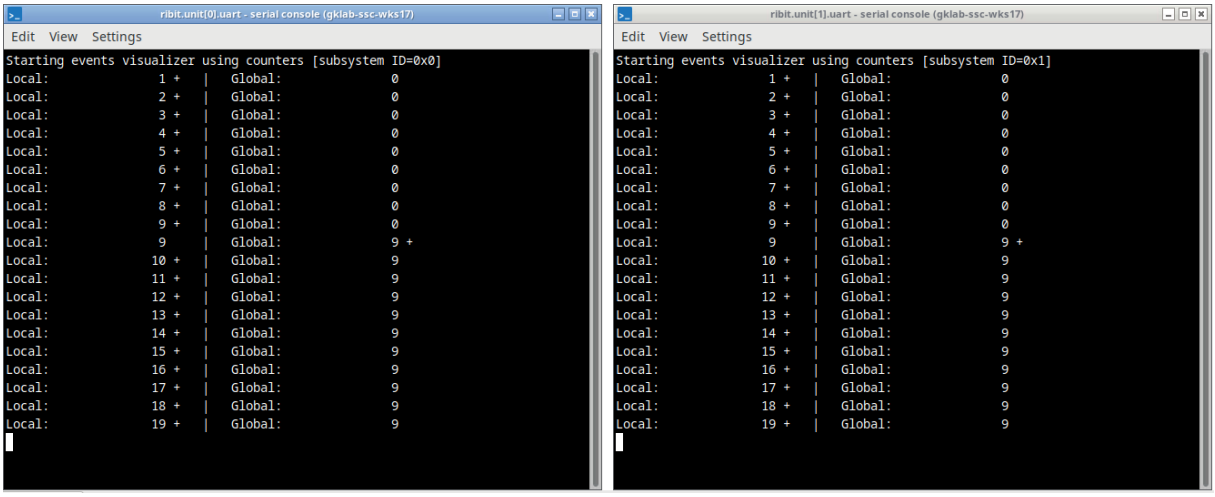
43. The event queue of the processor does not yet have an event posted from the global counter. Check it:

```
simics> print-event-queue "ribit.unit[0].hart"
```

44. **Run** the simulation until the global counter does a tick. This can be done using **cycle_event** breakpoints. These breakpoints use the **vtime** objects to target either plain cycles or picoseconds considered as cycles.

```
simics> bp.cycle_event.run-until event-object = ribit.counter object =
ribit.clock.vtime.cycles
```

When the simulation stops, both the serial ports for the processors show the same state. The global counter has not counted up, while both local counters are up to 19.



Changing the queue attribute does not affect existing events, and indeed a device is free to have events posted on any number of queues.

45. **Check the current time** across the system:

```
simics> list-processors -cycles -all
```

The time across all the processors should be the same. The simulation is at the start of a new round of time quanta.

46. Determine where the next event from the global counter is posted. Check the event queues of the global clock and the processor in unit 0:

```
simics> print-event-queue "ribit.clock"
simics> print-event-queue "ribit.unit[0].hart"
```

This should show that there are now two counter events in the processor, and nothing in the global clock. The events are not quite aligned in time.

47. **Select** the global clock and then **run** 50000 cycles:

```
simics> pselect ribit.clock
simics> run 50000 cycles
simics> list-processors -cycles -all
```

Note that nothing is logged, and no events are triggered. The simulator is working through an empty time quantum on the global clock.

48. **Check** the queue of the processor in unit 0 again:

```
simics> print-event-queue "ribit.unit[0].hart"
```

Nothing has changed, since the processor has not yet run since we last checked on it.

49. **Select** the processor in unit 0 and **run** 50000 cycles:

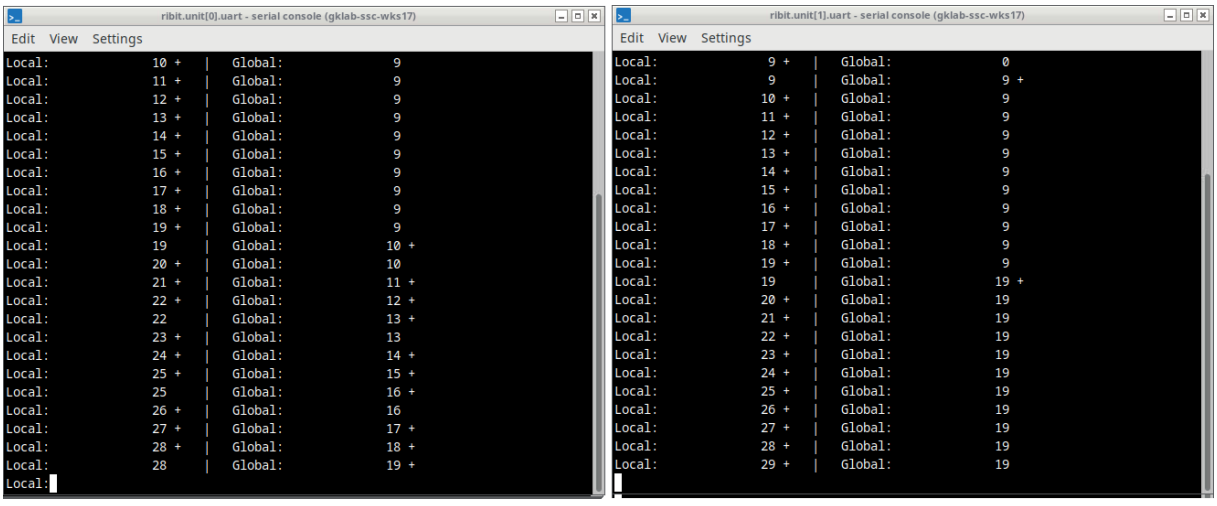
```
simics> pselect "ribit.unit[0].hart"
simics> run 50000 cycles
```

The serial port for unit 0 will show a number of lines where the local and/or global counter is counting up. The way the software is written, it takes a while to print each line, and during that printing either one or both counters will count up.

50. **Run** forward another 50000 cycles, ending the time quantum and letting the processor in unit 1 run:

```
simics> run 50000 cycles
```

The output from unit 1 still shows a constant value from the global counter. Since it is updating in the time quanta of unit 0.



51. **Quit** this simulation session.

```
simics> quit
```

2.5 Using Different Clock Frequencies

The events in this setup are posted on cycles. That means that changing the clock frequency of the clocks will change when the events arrive in virtual time.

52. Start a new simulation session using the target **`simics-internals-training/05-events`**:

```
$ ./simics simics-internals-training/05-events
```

53. Change the clock frequencies on the processors in unit 0 and 1:

```
simics> ribit.unit[0].hart->freq_mhz = 200
simics> ribit.unit[1].hart->freq_mhz = 75
```

54. Get the software stack started. **Run** until the synchronizer notifier hits:

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

55. **Check the current time** on all processors.

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

Note that the time is basically the same across all the clocks (the precision of the time column is a bit low, but the picoseconds column is clear).

56. Check the time quanta per clock:

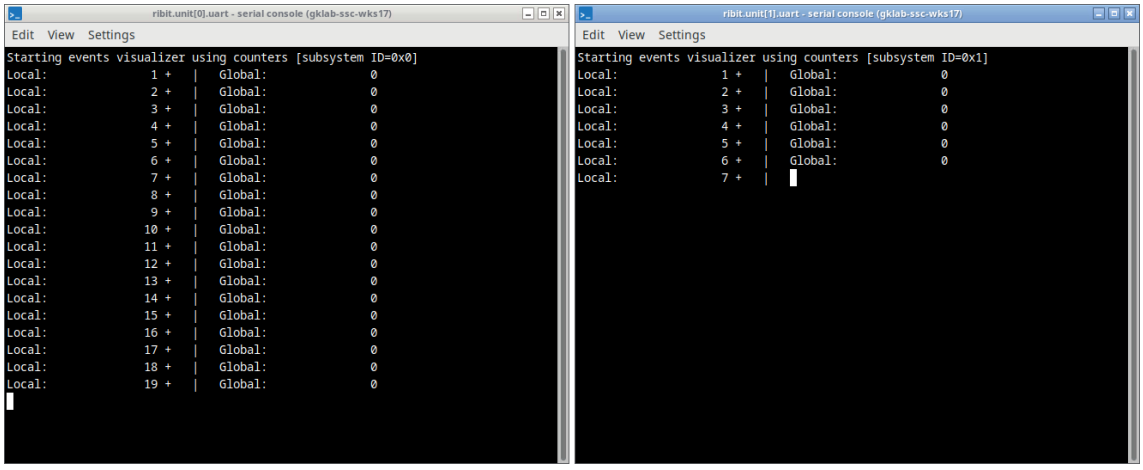
```
simics> set-time-quantum
```

The time quantum is set to 1 millisecond, resulting in a different number of cycles per quantum for each of the clocks in the system.

57. **Run** one time quantum, i.e., 1 millisecond:

```
simics> run 1 ms
```

The result is that more lines are printed from unit 0 than unit 1 since unit 0 goes through more cycles and, therefore, more event count updates. It also has more time to run the printing code, keeping up with the faster counter.



I.e., events posted on cycles happen more frequently (in virtual time) if the clock frequency is increased.

58. **Run** another millisecond:

```
simics> run 1 ms
```

This lets the global clock run a quantum too, and thus the global counter reaches 9.

2.6 Changing the Clock Frequency

The frequency of a processor or clock can be changed during runtime. This affects the contents of the cycle queue.

59. **Check** the current event queue on the processor in unit 0:

```
simics> print-event-queue "ribit.unit[0].hart" -i
```

60. **Change** the clock frequency of the processor to 50 MHz:

```
simics> ribit.unit[0].hart->freq_mhz = 50
```

61. **Check** the event queue again:

```
simics> print-event-queue "ribit.unit[0].hart" -i
```

Note that the cycle number for the events in the cycle queue has changed. The cycle queue is scaled when the frequency changes to maintain events at the same point in virtual time. However, events on the picosecond queue do not change.

62. **Run** the simulation until the counter event triggers:

```
simics> bp.cycle_event.run-until object =
"ribit.unit[0].hart.vtime.cycles" event-object =
"ribit.unit[0].counter"
```

63. **Check** the event queue again:

```
simics> print-event-queue "ribit.unit[0].hart" -i
```

There is a newly posted counter event (**`callback.tick`**) at 10000 cycles. The rescaling of the events in the queue only happened when the frequency changed. Any new events are posted on cycles such as they are at the point of posting.

This is why events should generally be posted on time, not on cycles. The same number of processor or clock cycle counts can mean different amounts of virtual time. Usually, devices define their semantics in terms of virtual time, not in terms of cycles on a processor.

64. **Quit** this simulation session.

```
simics> quit
```

2.7 Switching to Time-Based Posting

There is a backdoor in the counter device that changes it to post on virtual time instead of cycles. This can be used to demonstrate the effectiveness of posting on virtual time instead of processor cycles.

65. Start a new simulation session using the target **simics-internals-training/05-events**:

```
$ ./simics simics-internals-training/05-events
```

66. Change the clock frequencies on the processors in unit 0 and 1, just like above:

```
simics> ribit.unit[0].hart->freq_mhz = 200
simics> ribit.unit[1].hart->freq_mhz = 75
```

67. Get the software stack started. **Run** until the synchronizer notifier hits:

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

68. **Run** one time quantum, i.e., 1 millisecond:

```
simics> run 1 ms
```

This will result in the same as before: different numbers of lines printed for unit 0 and unit 1, since events are counted in cycles and they have different clock frequencies.

69. Engage the time override, by setting the **post_in_picoseconds** attribute on the counters for unit 0 and unit 1. Leave the global counter alone. Set the posting interval to 200 microseconds, one 1/5 of the current time quantum.

```
simics> ribit.unit[0].counter->post_in_picoseconds = 200_000_000
simics> ribit.unit[1].counter->post_in_picoseconds = 200_000_000
```

70. **Run** one time quantum, i.e., 1 millisecond:

```
simics> run 1 ms
```

Note how both the units print notices about five (5) updates to the local counter. The updates are now independent of the clock frequency. The extent of a time quantum is visible from the changes to the global counter: each change to the global counter value marks the beginning of a new time quantum.

71. **Run** the simulation until the next counter event triggers on the processor in unit 0:

```
simics> bp.cycle_event.run-until object =
"ribit.unit[0].hart.vtime.cycles" event-object =
"ribit.unit[0].counter"
```

72. **Check** the event queue:

```
simics> print-event-queue "ribit.unit[0].hart" -i
```

Note that the event is now posted at 40000 cycles instead of 10000 cycles, since that is 200 microseconds for a 200 MHz clock.

73. Slow down the processor for unit 0 to 125 MHz:

```
simics> ribit.unit[0].hart->freq_mhz = 125
```

74. **Check** the event queue:

```
simics> print-event-queue "ribit.unit[0].hart" -i
```

The next counter event is now at 25000 cycles, which makes sense as that maintains the time of 200 microseconds.

75. **Run** one more time quantum, i.e., 1 millisecond:

```
simics> run 1 ms
```

Note how both units show five updates to the local counter.

76. **Quit** this simulation session.

```
simics> quit
```

That concludes the lab on event posting and event queues.

3 Measuring Event Postings

Devices that post events too frequently can impact the simulation performance. The simulator provides tools to measure event postings. This lab demonstrates the performance impact as well as how to measure event postings.

3.1 The Performance Impact of Event Posting - Baseline

1. Start a new simulation session using the target **`simics-internals-training/05-events`**:

```
$ ./simics simics-internals-training/05-events
```

2. **Run** the software through the startup, until the first counter event hits:

```
simics> bp.cycle_event.run-until object =
"ribit.unit[0].hart.vtime.cycles" event-object =
"ribit.unit[0].counter"
```

3. Measure the real time it takes for the simulator to run 500 milliseconds, to get a baseline. The simplest way to measure the real-time execution time is with a small amount of Python:

```
simics> python "t=time.perf_counter()" ; run 500 ms ; python "time.
perf_counter()-t"
```

   Note that the serial consoles will keep printing after the simulator stops; their output is asynchronous to the main execution thread.

4. **Quit** this simulation session.

```
simics> quit
```

3.2 Post Events More Frequently

To see the impact of more frequent event triggering and posting on performance, change the interval between events. This is done by writing the "interval" register behind the back of the software after it has started. The software never touches the register after the initial programming.

5. Start a new simulation session using the target **`simics-internals-training/05-events`**:

```
$ ./simics simics-internals-training/05-events
```

6. **Run** the software through the startup, until the first counter event hits:

```
simics> bp.cycle_event.run-until object =
"ribit.unit[0].hart.vtime.cycles" event-object =
"ribit.unit[0].counter"
```

7. **Check** the current value of the interval register for the counter in unit 0:

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

The current value is 10000, or 0x2710.

8. **Change** the interval to 1 cycle, to make the performance impact as big as possible:

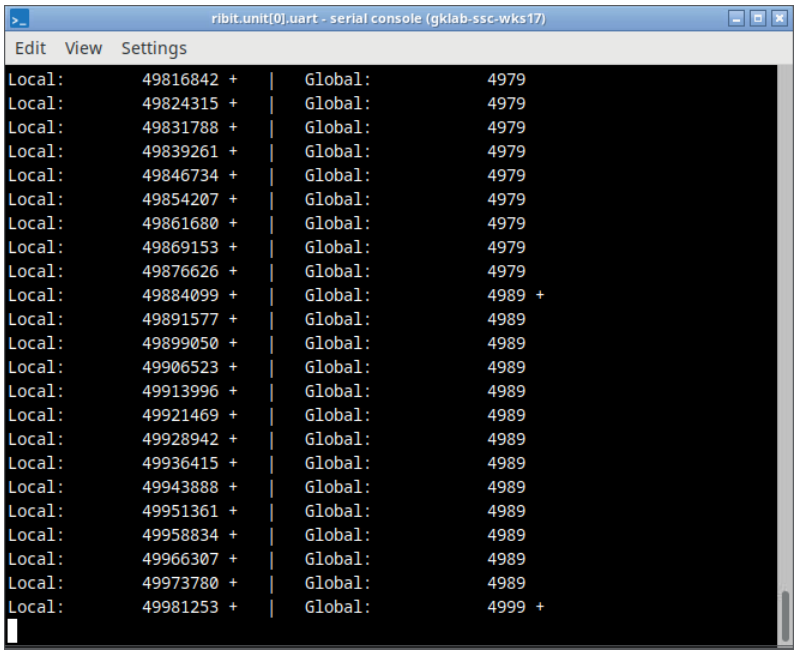
```
simics> write-device-reg
"ribit.unit[0].counter.bank.regs.event_interval" 1
```

9. Measure the real time it takes for the simulator to run 500 milliseconds, with this more frequent event triggering:

```
simics> python "t=time.perf_counter()" ; run 500 ms ; python "time.
perf_counter()-t"
```

You should see a runtime that is significantly higher than before. The effect is not linear, since there are many other things going on in the simulation.

Note the output on the serial port for unit 0: each printout shows an increment in the counter of far more than 1. Basically, while the software is busy printing a line, the event counter counts up many times. These changes are immediately picked up, and the software prints another line as quickly as it can.



Basically, the software is doing nothing more than it was before – it prints about as many lines of output as for a slow counter, but the simulation is still slowed down by the explosion of events. Triggering events too often has a significant performance impact on processors.

10. Add some more event processing by making the global counter also count up 10000x faster:

```
simics> write-device-reg "ribit.counter.bank.regs.event_interval" 1
```

11. Measure the impact with both the global counter and unit 0 counter triggering events on every cycle. Run another 500 ms:

```
simics> python "t=time.perf_counter()" ; run 500 ms ; python "time.
perf_counter()-t"
```

This is expected to take even longer – since the simulator is doing even more work processing events.

12. **Quit** this simulation session.

```
simics> quit
```

3.3 Measure Event Posting

13. Start a new simulation session using the target **simics-internals-training/05-events**:

```
$ ./simics simics-internals-training/05-events
```

14. **Run** the software through the startup, until the first counter event hits:

```
simics> bp.cycle_event.run-until object =
"ribit.unit[0].hart.vtime.cycles" event-object =
"ribit.unit[0].counter"
```

15. Create a new event histogram tool:

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

16. **Run** 100 milliseconds:

```
simics> run 100 ms
```

Wait for the simulation to stop. The collection of statistics will slow down the simulator, that's why you only run it for 100 ms.

17. **Check** the data collected by the event histogram tool:

```
simics> evh0.histogram
```

Perhaps surprisingly, the most common events come from the serial ports. They post an event for every character that is printed. These events are short-lived and thus unlikely to show up when inspecting the queues previously in this lab.

This explains the relatively small impact of making the counter events happen more often – there is already a large number of events being triggered.

18. Redo the event counting experiment with more events on unit 0.

Quit this simulation session.

```
simics> quit
```

19. Start a new simulation session using the target **simics-internals-training/05-events**:

```
$ ./simics simics-internals-training/05-events
```

20. **Run** the software through the startup, until the first counter event hits:

```
simics> bp.cycle_event.run-until object =
"ribit.unit[0].hart.vtime.cycles" event-object =
"ribit.unit[0].counter"
```

21. Create a new event histogram tool:

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

22. Set up the event interval:

```
simics> write-device-reg
"ribit.unit[0].counter.bank.regs.event_interval" 1
```

23. **Run** 100 milliseconds:

```
simics> run 100 ms
```

    Wait for the simulation to stop. To gauge the progress, look at the global counter value – it should be close to 1000 when the run finishes.

    This will take even longer to finish – there are very many events being processed, and each event will now cause the instrumentation tool to activate and count it. This is many times more processing per event than what is being done inside the virtual platform model.

24. Check the results of the histogram again:

```
simics> evh0.histogram
```

    The unit 0 counter should totally dominate the event counts.

    Note the **`Average interval`** for the various events. This is an indication of just how expensive an event is, since triggering more frequently should induce a higher overhead. But keep in mind that different events might carry very different costs.

25. **Quit** this simulation session.

```
simics> quit
```

4 A Closer Look at the Event Queue

This lab uses a set of special registers in the counter devices to demonstrate some details about the simulator event queues. The previous two labs used the whole system and the software. This lab basically only uses CLI interaction, but still with the software running "in the background."

4.1 Start a New Simulation Session

1. Start a new simulation session using the target **`simics-internals-training/05-events`**:

   ```
   $ ./simics simics-internals-training/05-events
   ```

2. **Run** the software through the startup, until the first counter event hits. This puts some interesting events on the cycle queues of the clocks:

   ```
   simics> bp.cycle_event.run-until object =
   "ribit.unit[0].hart.vtime.cycles" event-object =
   "ribit.unit[0].counter"
   ```

4.2 Multiple Events on the Queue

3. Inspect the secondary register bank of the counter for unit 0:

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

   Writing a value to any of these registers will post an event that many cycles into the future. The effect of an event is simply to print a log message; there is no side-effects for the state of the rest of the machine.

4. Raise the log level to 2 on the counter for unit 0:

   ```
   simics> log-level "ribit.unit[0].counter" 2
   ```

5. The processor in unit 0 should be the current processor. Check this and make sure it is the case if it is not:

   ```
   simics> pselect
   ```

   This should show "**`ribit.unit[0].hart`**".

6. **Check** the current event queue on the processor in unit 0 – the command shows the state for the current clock/processor by default:

   ```
   simics> print-event-queue -i
   ```

   This is the same as what you saw before: one event at 10000 cycles from the regular counter counting, and then a clint timeout at way later. Plus a time quantum end in the picosecond queue.

   Add some new events to the queue.

7. Post an event using register **alpha**, in 5 cycles. Check the results:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.alfa 5
simics> print-event-queue -i
```

This should show a new event posted on cycle 5.

8. Post an event using register **bravo**, in 6 cycles. Check the results:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.bravo 6
simics> print-event-queue -i
```

9. Post events using registers **charlie** and **delta**, in 7 cycles. Check the results:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.charlie 7
simics> write-device-reg ribit.unit[0].counter.bank.extras.delta 7
simics> print-event-queue -i
```

10. Post another event using register **alpha**, also in 7 cycles. Check the results:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.alfa 7
```

11. Check the resulting event queue:

```
simics> print-event-queue -i
```

This has multiple events on cycle 7. The order of the listing corresponds to the order of posting:

```
simics> print-event-queue -i
┌─────────────────────┬──────────────────────┬──────────────────────┐
│        Cycle        │        Object        │     Description      │
├─────────────────────┼──────────────────────┼──────────────────────┤
│                   5 │ribit.unit[0].counter │extras.alfa.event     │
│                   6 │ribit.unit[0].counter │extras.bravo.event    │
│                   7 │ribit.unit[0].counter │extras.charlie.event  │
│                   7 │ribit.unit[0].counter │extras.delta.event    │
│                   7 │ribit.unit[0].counter │extras.alfa.event     │
│               10000 │ribit.unit[0].counter │callback.tick         │
…
```

Note that "the same" event can be posted multiple times – that is entirely up to the device model.

12. Run four (4) cycles:

```
simics> run 4 cycles
```

13. Check the event queue:

```
simics> print-event-queue -i
```

All the events have moved 4 cycles closer in time.

14. Run one (1) cycle:

```
simics> run 1 cycle
```

This should trigger a log message indicating that an event callback for **alpha** happened.

15. **Check** the event queue:

```
simics> print-event-queue -i
```

The first event posted is gone, and the next event is in 1 cycle on **bravo**.

16. **Run** one (1) cycle:

```
simics> run 1 cycle
```

The **bravo** callback happens.

17. **Run** one (1) cycle:

```
simics> run 1 cycle
```

This shows three callbacks happening: **charlie**, **delta**, and **alpha** in that order. The order of the callbacks is the same as the order of posting.

18. The same callback can be posted multiple time on the same cycle as well. Test this by **posting multiple events** from the same register, on the same cycle:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.alfa 5
simics> write-device-reg ribit.unit[0].counter.bank.extras.bravo 5
simics> write-device-reg ribit.unit[0].counter.bank.extras.charlie 5
simics> write-device-reg ribit.unit[0].counter.bank.extras.bravo 5
simics> write-device-reg ribit.unit[0].counter.bank.extras.alfa 5
simics> write-device-reg ribit.unit[0].counter.bank.extras.delta 5
simics> print-event-queue -i
```

19. **Run** five (5) cycles:

```
simics> run 5 cycles
```

This should trigger multiple log messages, in the same order as the events were posted.

4.3 Drive the Picosecond Queue (Optional)

Time can be driven using the picosecond queue as well as the cycles. This lab is optionally since normally the simulation is driven in terms of seconds… but it is interesting in any case.

20. **Post** an event using register **papa**, in 1 picosecond. Check the results:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.papa 1
simics> print-event-queue -i
```

Note how this event shows up on the picosecond queue, not the cycle queue. Also note that it is one picosecond into the future, which is rather less than a cycle. Still, it possible to drive the simulator at this level of granularity.

21. **Run** the simulation for 1 picosecond:

```
simics> run 1 ps
```

The event triggers. Just what is the current queue after this?

22. **Check** the event queue:

```
simics> print-event-queue -i
```

Note that the time of the "time quantum end" event has decreased by a single picosecond!

23. Set up a more complex scenario, with events on 10 ps, 20000 ps, and 1 cycle:

```
simics> write-device-reg ribit.unit[0].counter.bank.extras.papa 10
simics> write-device-reg ribit.unit[0].counter.bank.extras.quebec 20000
simics> write-device-reg ribit.unit[0].counter.bank.extras.alfa 1
```

24. **Check** the event queue:

```
simics> print-event-queue -i
```

25. **Run** the simulation for 1 picosecond and check the queue:

```
simics> run 1 ps
simics> print-event-queue -i
```

Note that cycle queue has not counted down, but that all ps-queue events have moved down 1 ps.

26. **Run** the simulation for 1 cycle (which happens to also be 10000 picoseconds):

```
simics> run 1 cycle
```

Note that the events for **papa** and **alfa** trigger – and **papa** is before **alfa** since it is actually somewhat closer in time.

27. **Check** the event queue:

```
simics> print-event-queue -i
```

The counter event on the queue has come 1 cycle closer. The **quebec** event is 10001 picoseconds away. This shows that running one cycle runs until the next cycle has happened, but not necessarily for an equivalent number of picoseconds.

28. **Run** the simulation for one more cycle:

```
simics> run 1 cycle
simics> print-event-queue -i
```

Note that the **quebec** event is now 1 picosecond away.

29. **Run** the simulation for one picosecond:

```
simics> run 1 ps
```

The event triggers.

30. **Quit** this simulation session.

```
simics> quit
```