Getting started tutorial

Kees Goossens

Mojtaba Haghi

Kees Goossens
Mojtaba Haghi s.m.haghi@tue.nl
Electronic Systems Group
Electrical Engineering Faculty

Technische Universiteit
Eindhoven
University of Technology

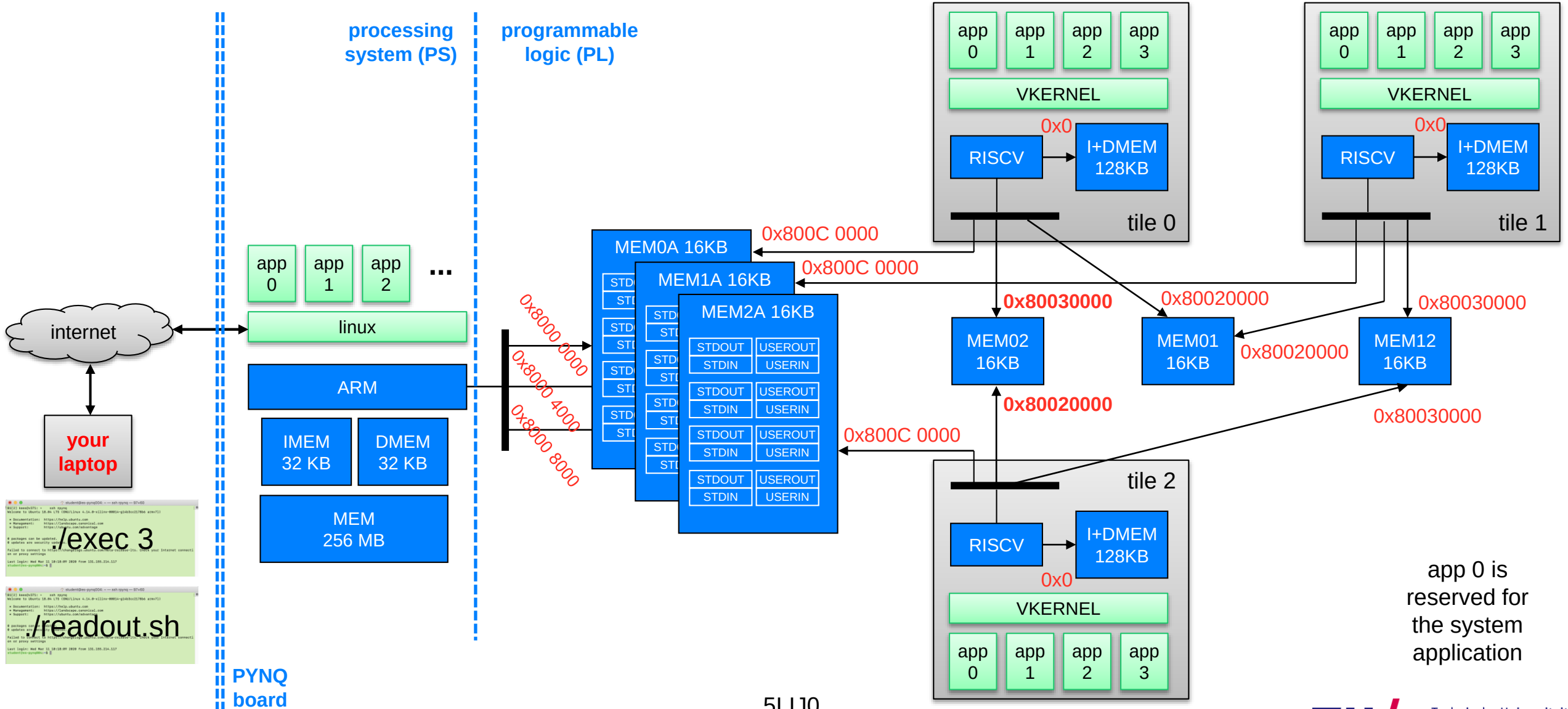
Where innovation starts

overview

- the goal of this tutorial is to familiarize with CompSOC programming, execution, benchmarking and debugging
- we use the Verintec industrial implementation of CompSOC

- you will learn how to:
  - run applications
  - use the timers for benchmarking
  - modify TDM schedules
  - map variables into memories
  - communicate between RISC-V cores

ARM processor, 3 RISC-V cores, 4 partitions per tile



app 0 is reserved for the system application

arrows go from master to slave

PYNQ, and getting the tutorial

- Connect to the PYNQ board using MobaXterm (Windows)
- Go to the directory “Tutorial_SDK” → Use command: “cd Tutorial_SDK”
- The contents of the SDK is shown below:

/home/student/Tutorial_SDK/

Name	Size (KB)	Last modified	Owner	Group	Access
..					
app_tile_0_1		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_0_2		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_0_3		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_1_1		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_1_2		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_1_3		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_2_1		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_2_2		2020-08-11 ...	student	student	drwxrwxr-x
app_tile_2_3		2020-08-11 ...	student	student	drwxrwxr-x
final-project-submission		2020-08-11 ...	student	student	drwxrwxr-x
libbsp		2020-08-11 ...	student	student	drwxrwxr-x
libchannel		2020-08-11 ...	student	student	drwxrwxr-x
libexample		2020-08-11 ...	student	student	drwxrwxr-x
libuserchannel-arm		2020-08-11 ...	student	student	drwxrwxr-x
libuserchannel-riscv		2020-08-11 ...	student	student	drwxrwxr-x
monitoring-tools		2020-08-11 ...	student	student	drwxrwxr-x
project		2020-08-11 ...	student	student	drwxrwxr-x
share		2020-08-11 ...	student	student	drwxrwxr-x
shared		2020-08-11 ...	student	student	drwxrwxr-x
tiles		2020-08-11 ...	student	student	drwxrwxr-x
tutorial		2020-08-11 ...	student	student	drwxrwxr-x
exec.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
Makefile	1	2020-08-11 ...	student	student	-rw-rw-r--
readout.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
rerun.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
reset.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
run.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
sort-on-time.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
stop.sh	1	2020-08-11 ...	student	student	-rwxrwxr-x
vep-config.txt	1	2020-08-11 ...	student	student	-rw-rw-r--

file structure

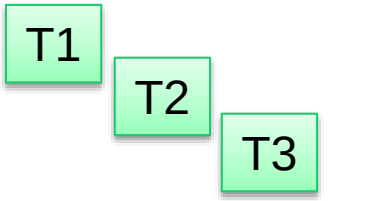
`Makefile`	clean or compile all applications (calls all `app_tile_*/Makefile`)
`app_tile_0_1`	all the files for application in VEP 1 on tile 0
`Makefile`	clean or compile only this application
`main.c`	all *.c files for this application
`...`	...
`app_arm_echo`	all the files for the echo application running on ARM
`app_arm_fractal`	all the files for the fractal application running on ARM
`...`	...
`vep-config.txt`	definition of VEPs on the three RISC-V cores
`readout.sh`	print output of RISC-V cores on the ARM
`run.sh, stop.sh, exec.sh`	start, stop, and execute all applications for x seconds
`sort-on-time.sh`	

Commands for compiling and running

- `make clean`
  - remove all generated files (executables, etc.)
  - `make veryclean` also removed compiled libraries
- `make`
  - to compile your programs into a executables that will be uploaded
  - calls the `Makefile` for all `app_*` directories, i.e .for both RISC-V and ARM applications
- `run.sh`
  - to compile, upload executables to the RISC-V cores, program TDM of VKERNEL, and run the programs
  - `rerun.sh` doesn't recompile
- `stop.sh`
  - stop all running RISC-V programs
- `exec.sh s` is equal to: `run.sh; sleep s; stop.sh`
  - if there's lots of output, you may not be able to run `stop.sh`
- `reset.sh`
  - reload the Verintec platform on the FPGA
- `readout.sh`
  - to receive the output from the RISC-V cores on the terminal running on the ARM Linux
  - execute in a separate terminal from `exec.sh` (otherwise you get output from two processes in the same terminal)
  - you must always run `readout.sh` before `run, start, stop, reset` otherwise they won't work
- execute these scripts by typing `./run.sh ./stop.sh ./exec.sh s ./readout.sh ./reset.sh` on the command line in a terminal

tutorial overview

1. running the example platform
2. global timer
3. TDM schedule
4. partition timer



...

Tutorial

1. getting started

Kees Goossens
Mojtaba Haghi s.m.haghi@tue.nl
Electronic Systems Group
Electrical Engineering Faculty

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

how to run applications on the PYNQ

T1

- login with two separate terminals on the PYNQ (place them side by side on your screen)
- both terminals run on Ubuntu Linux on the ARM
- change to the tutorial directory you just checked out (`cd directory`)

kees — student@es-pynq004: ~/tutorial-stud1 — ssh -Y student@es-pynq004.ics.ele.tue.nl — 94×32

```
kees@TUE16447 ~ % ssh -Y student@es-pynq004.ics.ele.tue.nl
student@es-pynq004.ics.ele.tue.nl's password:
Warning: No xauth data; using fake authentication data for X11 forwarding.
Welcome to Ubuntu 18.04 LTS (GNU/Linux 4.14.0-xilinx-00014-g14b3cc2178b6 armv7l)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage


0 packages can be updated.
0 updates are security updates.

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet conne
ction or proxy settings

Last login: Mon Apr 20 14:16:46 2020 from 131.155.93.21
student@es-pynq004:~$ cd tutorial-stud1/
student@es-pynq004:~/tutorial-stud1$ ls
Makefile      app_tile_1_3  libchannel            rerun.sh         stop.sh
app_tile_0_1  app_tile_2_1  libexample            reset.sh         tiles
app_tile_0_2  app_tile_2_2  libuserchannel-arm    run.sh           tutorial
app_tile_0_3  app_tile_2_3  libuserchannel-riscv  share            tutorial-echo
app_tile_1_1  exec.sh       monitoring-tools      shared           tutorial-fractal
app_tile_1_2  libbsp        readout.sh            sort-on-time.sh  vep-config.txt
student@es-pynq004:~/tutorial-stud1$ ./readout.sh
```

kees — student@es-pynq004: ~/tutorial-stud1 — ssh -Y student@es-pynq004.ics.ele.tue.nl — 93×32

```
kees@TUE16447 ~ % ssh -Y student@es-pynq004.ics.ele.tue.nl
student@es-pynq004.ics.ele.tue.nl's password:
Warning: No xauth data; using fake authentication data for X11 forwarding.
Welcome to Ubuntu 18.04 LTS (GNU/Linux 4.14.0-xilinx-00014-g14b3cc2178b6 armv7l)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage


0 packages can be updated.
0 updates are security updates.

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet conn
ection or proxy settings

Last login: Mon Apr 20 13:58:04 2020 from 131.155.93.21
student@es-pynq004:~$ cd tutorial-stud1/
student@es-pynq004:~/tutorial-stud1$ ls
Makefile      app_tile_1_3  libchannel            rerun.sh         stop.sh
app_tile_0_1  app_tile_2_1  libexample            reset.sh         tiles
app_tile_0_2  app_tile_2_2  libuserchannel-arm    run.sh           tutorial
app_tile_0_3  app_tile_2_3  libuserchannel-riscv  share            tutorial-echo
app_tile_1_1  exec.sh       monitoring-tools      shared           tutorial-fractal
app_tile_1_2  libbsp        readout.sh            sort-on-time.sh  vep-config.txt
student@es-pynq004:~/tutorial-stud1$ ./exec.sh 2
```

- start the monitoring (`./readout.sh`) to get the output on the ARM
- then start the execution for 2 seconds on the real-time platform (`./exec.sh 2`)

how to run applications on the PYNQ

T1

kees — student@es-pynq004: ~/tutorial-stud1 — ssh -Y student@es-pynq004.ics.ele.tue.nl — 94×32

```
0 packages can be updated.
0 updates are security updates.

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet conne
ction or proxy settings

Last login:
student@es-p
student@es-p
Makefile                                                    top.sh
app_tile_0_1  app_tile_2_1  libexample            reset.sh          tiles
app_tile_0_2  app_tile_2_2  libuserchannel-arm    run.sh            tutorial
app_tile_0_3  app_tile_2_3  libuserchannel-riscv  share             tutorial-echo
app_tile_1_1  exec.sh       monitoring-tools      shared            tutorial-fractal
app_tile_1_2  libbsp        readout.sh            sort-on-time.sh   vep-config.txt
student@es-pynq004:~/tutorial-stud1$ ./readout.sh
mem: 16 KB @ 0x0x80000000
CHeap stdout initialised
mem: 16 KB @ 0x0x80004000
CHeap stdout initialised
mem: 16 KB @ 0x0x80008000
CHeap stdout initialised
00 01: Hello world
00 02: Hello world
00 03: Hello world
01 01: Hello world
01 02: Hello world
01 03: Hello world
02 01: Hello world
02 02: Hello world
02 03: Hello world
```

after typing `./readout.sh` in this terminal, and `./exec.sh 5` in the other terminal this should be the output of running the example

initialise the FIFOs in `mem0a, mem1a, mem2a`

- output from all the RISC-V cores
- type ^C (control-C) to stop ./readout.sh

kees — student@es-pynq004: ~/tutorial-stud1 — ssh -Y

```
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
loc: 800C0660
loc: 800C0980
Opened state file: ./monitoring-tools/vep-config.jsor
Found tiles
Process tile: 1
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
loc: 800C0660
loc: 800C0980
Opened state file: ./monitoring-tools/vep-config.jsor
Found tiles
Process tile: 2
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
loc: 800C0660
loc: 800C0980
Stopping all VPs
loc: 800C0660
loc: 800C0980
Stopping all VPs
loc: 800C0660
loc: 800C0980
Stopping all VPs
student@es-pynq004:~/tutorial-stud1$
```

load the TDM tables in the VKERNELs on tiles 0, 1, 2

stop all partitions on tiles 0, 1, 2 after 5 seconds

how to run applications on the PYNQ

T1

- output of the RISC-V cores

```
00 01: Hello world
00 02: Hello world
00 03: Hello world
01 01: Hello world
01 02: Hello world
01 03: Hello world
02 01: Hello world
02 02: Hello world
02 03: Hello world
```

`00 01:` output from core 0, partition 1

⋮

`02 03:` output from core 2, partition 3

- three cores
- three partitions
  - partition 0 (system application) is not shown

Tutorial

2. global timer

Kees Goossens
Mojtaba Haghi s.m.haghi@tue.nl
Electronic Systems Group
Electrical Engineering Faculty

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

timers

T2

- each RISC-V tile has 2 timers

- global timer
  - `volatile uint64_t *g_timer = (uint64_t*)TILE2_TIMER_0_S_AXI;`
  - always running
  - can be used to compute the precise elapsed time (from start to finish) – i.e. the response time (RT)
  - wall timers on all tiles are identical (it is as if it is a single timer for all tiles)

- partition timer (virtualised per partition)
  - `volatile uint64_t *p_timer = (uint64_t*)TILE2_TIMER_1_S_AXI;`
  - runs only when the partition is running
  - i.e. in all TDM slots allocated to this application on this core
  - the precise number of clock cycles that the application has executed – i.e. the execution time (ET)

timers

T2

- modify the `main.c` program of tile 0, partition 1 (`app_tile_0_1/main.c`)
- to add the declaration of the two timers and a temporary variable t

```
volatile const uint64_t * const g_timer = (uint64_t*)TILE0_TIMER_0_S_AXI;
volatile const uint64_t * const p_timer = (uint64_t*)TILE0_TIMER_1_S_AXI;
```

read the declaration backwards: g_timer is a const pointer to a 64-bit unsigned integer constant that is volatile

- read the value of the timer and print the most-significant 32 bits and the least-significant 32 bits
- use the `xil_printf` function, which is a simple version of `printf`, e.g.

```
uint64_t gnow = *g_timer; // get 64-bits timer value by reading the value at address g_timer
xil_printf("timer top 32 bits: %010x\n", (uint32_t) (gnow>>32)); // print hex
xil_printf("timer top 32 bits: %010u\n", (uint32_t) (gnow>>32)); // print dec
```

- `xil_printf` cannot print long integers (64 bits), therefore must print top & bottom 32 bits separately
  - (it will compile, but gives wrong output)

must cast a 64-bit integer to a 32-bit unsigned int

timers

T2

- run your program
- you should see something like this:

t02a.c

```
mem: 16 KB @ 0x0x80000000
CHeap stdout initialised
mem: 16 KB @ 0x0x80004000
CHeap stdout initialised
mem: 16 KB @ 0x0x80008000
CHeap stdout initialised
00 01: global timer top 32 bits:    0000008518
00 02: Hello world
00 03: Hello world
00 01: global timer bottom 32 bits: 0513494160
01 01: Hello world
01 02: Hello world
01 03: Hello world
02 01: Hello world
02 02: Hello world
02 03: Hello world
```

timers

T2

- put the printing of the timer in a `while(1)` loop and re-run your program

t02b.c

```
uint64_t gprev = 0;
while (1) {
  uint64_t gnow = *g_timer;
  xil_printf("global timer top 32 bits:    %010u\n", (uint32_t)(gnow>>32));
  xil_printf("global timer bottom 32 bits: %010u\n", (uint32_t)gnow);
  xil_printf("global timer diff:           %010u\n", (uint32_t)(gnow-gprev));
  gprev = gnow;
}
```

- the bottom 32 bits of the timer change every time you read the timer, but the top 32 bits don't
- run long enough such that the top 32 bits of the timer also change

timers

T2

- how long do you need to run to see bit 32 (LSB of top 32 bits) change?
- give a formula for this duration (in seconds)
  - hint: 40 MHz, 32 bits, overflow

answer:

- the timer is 64 bits but the RISC-V and the peripheral bus are 32 bits
- so it takes 2 reads and multiple clock cycles to get the complete value of the timer
- the timer hardware ensures consistency between the two reads
  - a copy is made of all 64 bits when the timer is read
  - the next time the timer is read the value of the copy is returned
  - this does work not if the application is swapped out between the two reads, which is very unlikely

timers – this is reference info, can be skipped now

T2

- this does not work if the application is swapped out between the two reads

- when the partition is swapped out between the first and second read on the bus, when the second read transaction occurs, the second word is no longer consistent with the first word
- even then, most of the time this is not a problem because the top and bottom 32 bits are consistent as long as the bottom 32 bits do not overflow in between being swapped out and being swapped in
- you're ok as long as this does not happen in time interval that you're swapped out (e.g. ~35000 cycles for 3 other slots out of 2^32 cycles would lead to ~8e-6 probability of error)

- to be 100% safe, you should therefore check for this in software, and reread the timer (both words) if the second word was not consistent with the first word
- given that you will do this at the start of your TDM slot (since you were just swapped out) and TDM slots are long enough the second read of the timer will not have the problem

timers – this is reference info, can be skipped now

T2


```
uint64_t read_gtimer()
{
  // we assume all tiles have the timer at the same memory location
  volatile const uint64_t *const timer    = (uint64_t *)TILE0_TIMER_0_S_AXI;
  volatile const uint32_t *const timer_ls = (uint32_t *)TILE0_TIMER_0_S_AXI;
  uint64_t t1;
  uint32_t t2_ls;
  while (1) {
    t1 = *timer;
    t2_ls = *timer_ls;
    const uint32_t t1_ls = t1;
    // it takes 12 or 18 cycles to read the LSW of timer
    // first condition is required to deal with possible overflow in second condition
    if (t2_ls > t1_ls && t2_ls >= t1_ls + 12 && t2_ls <= t1_ls + 18) return t1;
    const int32_t diff = t2_ls - t1_ls; // can wrap                                  // optional code
    xil_printf("warning: %s; re-read timer; t1=%08X%08X t2_ls=%08X diff=%d\n",      // optional code
               (t2_ls <= t1_ls ?  "timer wrapped" : "swapped out"),                 // optional code
               (uint32_t) (t1 >> 32), t1_ls, t2_ls, diff);                          // optional code
  }
}
```

ARM – RISC-V flow control

T2

- if the RISC-V cores produce a lot of output with xil_printf and the ARM is busy running other programs then the ARM may not be fast enough to accept all stdout data

- the stdin/out, user-in/FIFOs never lose data

- when the ARM does not accept the stdout data then the RISC-V cores will stall
  - this process is called flow control or back-pressure
- you will see this in the time stamps because the RISC-V programs will run slower

Tutorial

3. TDM schedule

Kees Goossens
Mojtaba Haghi s.m.haghi@tue.nl
Electronic Systems Group
Electrical Engineering Faculty

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

TDM schedule

T3

- the `vep-config.txt` file defines the time-division multiplexing (TDM) slot table for the VKERNEL
  - i.e. which applications run and how much time they get
  - always contains 4 slots of which you can use up to 3
  - the system application gets all un-assigned slots and always has 5000 cycles total in the slot table
  - the VKERNEL runs between slots for 2000 cycles
  - an application may have more than 1 slots

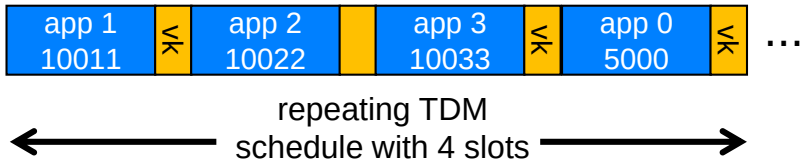
vep-config.txt

```
# this is a comment
on tile 0 next slot is for partition 1 with 10011 cycles
on tile 0 next slot is for partition 2 with 10022 cycles
on tile 0 next slot is for partition 3 with 10033 cycles
on tile 1 next slot is for partition 1 with 10000 cycles
on tile 1 next slot is for partition 2 with 10000 cycles
on tile 1 next slot is for partition 3 with 10000 cycles
on tile 2 next slot is for partition 1 with 10000 cycles
on tile 2 next slot is for partition 2 with 10000 cycles
on tile 2 next slot is for partition 3 with 10000 cycles
```

- as shown on the right, on tile 0
  - app 1 (tile_app_0_1/main.c) runs in slot 1 for 10011 clock cycles
  - ...
  - app 0 (system app) is automatically added and always runs in the last slots 0 for 5000 clock cycles total

output of run.sh

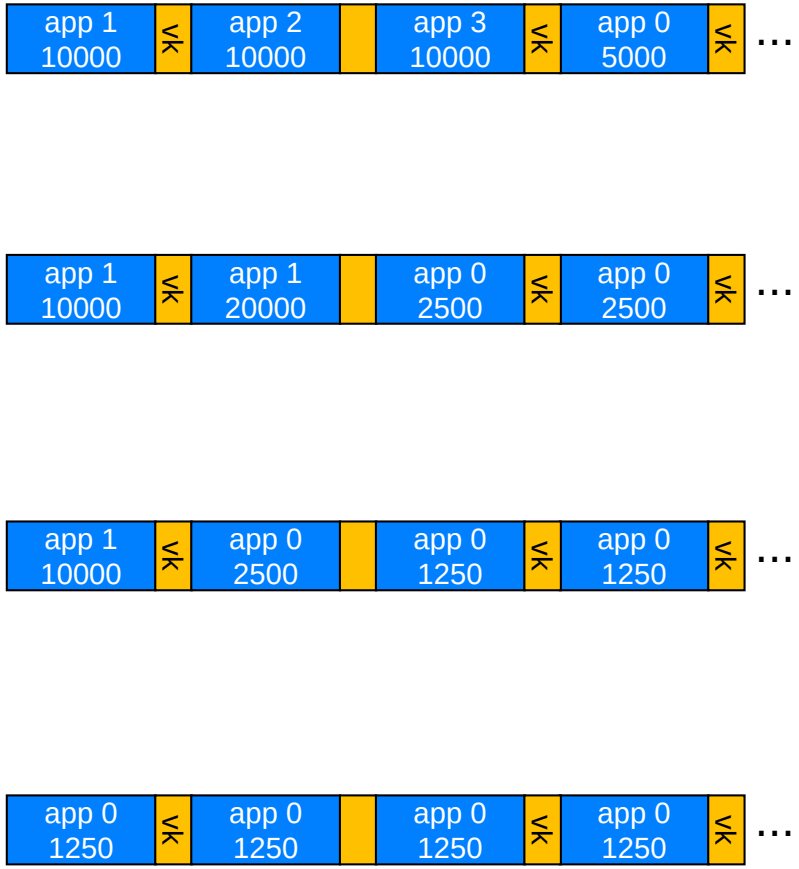
```
Process tile: 0
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
Process tile: 1
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
Process tile: 2
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
```



TDM schedule

T3

vep-config.txt
bold = your input
italic = automatically added




```
on tile 0 next slot is for partition 1 with 10000 cycles
on tile 0 next slot is for partition 2 with 10000 cycles
on tile 0 next slot is for partition 3 with 10000 cycles
on tile 0 next slot is for partition 0 with 5000 cycles
```

```
on tile 0 next slot is for partition 1 with 10000 cycles
on tile 0 next slot is for partition 2 with 10000 cycles
on tile 0 next slot is for partition 0 with 2500 cycles
on tile 0 next slot is for partition 0 with 2500 cycles
```

```
on tile 0 next slot is for partition 1 with 10000 cycles
on tile 0 next slot is for partition 0 with 2500 cycles
on tile 0 next slot is for partition 0 with 1250 cycles
on tile 0 next slot is for partition 0 with 1250 cycles
```

```
on tile 0 next slot is for partition 0 with 1250 cycles
on tile 0 next slot is for partition 0 with 1250 cycles
on tile 0 next slot is for partition 0 with 1250 cycles
on tile 0 next slot is for partition 0 with 1250 cycles
```

how to run applications on the PYNQ

- `vep-config.txt`
  - defines the time slots for applications running in vep v on tile t
  - only schedule applications that exist in app_tile_t_v (tile t, vep v)
  - even if the application code exists you can decide not to schedule it, which is often useful to debug a single application at a time

- running consists of
  - stopping running programs
  - clearing the shared memories
  - loading the executables and TDM tables
- you can do this per core (see `./run.sh` for the detailed commands)

- to re-align TDM slots to their original state you can `./reset.sh` the platform
- this isn't necessary in any way, but can help to make your TDM timing experiments reproducible

memory & stack

- see the `vep-config.txt` file, e.g.

```
## - partitions can only have a memory size of 32K or 64K
## - the size of all partitions on a tile must be at most 96K
##   (i.e. 128K including the system application)
## - default memory is 32K, default stack is 4K
##
## use /opt/riscv/bin/riscv32-unknown-elf-size [-A] *.elf
## to analyse a partition's memory usage
##
on tile 0 partition 1 has 32K memory and 4K stack
on tile 0 partition 2 has 32K memory and 4K stack
on tile 0 partition 3 has 32K memory and 4K stack
on tile 1 partition 1 has 32K memory and 4K stack
on tile 1 partition 2 has 32K memory and 4K stack
on tile 1 partition 3 has 32K memory and 4K stack
on tile 2 partition 1 has 32K memory and 4K stack
on tile 2 partition 2 has 32K memory and 4K stack
on tile 2 partition 3 has 32K memory and 4K stack
```

- memory must be 32K or 64K
- stack must be less than memory
- default memory is 32K
- default stack is 4K
- per core, sum of user-partition memories must be at most 96K

- in other words,
  - 1-4 partitions of 32K, or
  - 1-2 partitions of 32K and 1 of 64K

memory & stack

- You might run out of memory:

```
/opt/riscv/.../bin/ld: mb.elf section `.bss.large' will not fit in region `mem
/opt/riscv/.../bin/ld: region `mem' overflowed by 26060 bytes
collect2: error: ld returned 1 exit status
../share/make_tile.mk:74: recipe for target 'mb.elf' failed
make: *** [mb.elf] Error 1
```

- to understand where your functions & data are placed in memory
  - `/opt/riscv/bin/riscv32-unknown-elf-size *.elf`
  - `/opt/riscv/bin/riscv32-unknown-elf-size -A *.elf`
- in the example, the total memory used by the program is 13580 + 4096
  - this must be less than the memory allocated to the partition
  - ignore everything after the stack
- remember that you also have shared memories, where you can place your data (manually)
- it's recommended not to use malloc, since then it's harder to keep track of what happens to memory

text	data	bss	dec	hex	filename
11924	184	9614	21722	54da	app_tile_0_1/mb.elf

program fits since 0x54da < 0x8000

section	size	addr
.text.init	76	0
.text	926	76
.text.get_next_MK	68	1002
...		
.sbss.window	1	13564
.sbss.__malloc_free_list	4	13568
.sbss.__malloc_sbrk_start	4	13572
.sbss.heap_end.1820	4	13576
.stack	**4096**	**13580**
.comment	34	0
....		
debug_frame	232	0
Total	84427	

TDM schedule

T3

- add the red line to your code, to put the RISC-V core to sleep until the start of the next TDM slot

```
uint64_t gprev = 0;
while (1) {
  asm("wfi");
  uint64_t gnow = *g_timer;
  xil_printf("global timer top 32 bits:    %010u\n", (uint32_t)(gnow>>32));
  xil_printf("global timer bottom 32 bits: %010u\n", (uint32_t)gnow);
  xil_printf("global timer diff:           %010u\n", (uint32_t)(gnow-gprev));
  gprev = gnow;
}
```

- what happens to the timer values?

sample output

T3

t03a.c

```
mem: 16 KB @ 0x0x80000000
CHeap stdout initialised
mem: 16 KB @ 0x0x80004000
CHeap stdout initialised
mem: 16 KB @ 0x0x80008000
CHeap stdout initialised
00 01: Hello world
00 02: Hello world
00 01: global timer top 32 bits:    0000000035
00 01: global timer bottom 32 bits: 1949482647
00 03: Hello world
00 01: global timer diff:           1949482647
00 01: global timer top 32 bits:    0000000035
00 01: global timer bottom 32 bits: 1949611647
00 01: global timer diff:           0000129000
00 01: global timer top 32 bits:    0000000035
00 01: global timer bottom 32 bits: 1949740647
00 01: global timer diff:           0000129000
00 01: global timer top 32 bits:    0000000035
00 01: global timer bottom 32 bits: 1949869647
00 01: global timer diff:           0000129000
```

- use `./readout.sh | tee output.txt`
- to save the output to a file,
  while also printing it on the screen
- you can then filter out e.g. tile 0 application 1
  `grep '^00 01' output.txt`
- or all lines containing diff:
  `grep diff output.txt`

```
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
00 01: global timer diff:           0000129000
```

- can you explain the value 129000?

answer:

accurate timing

T3

- make sure that you get accurate timings in the following manner
  - change the `while(1)` loop to a `for` loop with 100 iterations
  - save 100 timer values in an array of `uint64_t`
  - after the loop, print out the 100 timer values and differences

```
for (int i=0; i < 100; i++) {
  xil_printf("global timer top 32 bits:    %010u\n", (uint32_t)(gnow[i]>>32));
  xil_printf("global timer bottom 32 bits: %010u\n", (uint32_t)gnow[i]);
  xil_printf("global timer diff:           %010u\n", (uint32_t)(gnow[i]-gnow[i-1]));
  ...
}
```

  - (don’t forget to fix the bug in the example code!)

- you should now have 100 identical global timer diffs

sample output

T3

t03b.c

```
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3767899999
00 01: global timer diff:               0000000000
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3767942999
00 01: global timer diff:               0000043000
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3767985999
00 01: global timer diff:               0000043000
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3768028999
00 01: global timer diff:               0000043000
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3768071999
00 01: global timer diff:               0000043000
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3768114999
00 01: global timer diff:               0000043000
00 01: global timer top 32 bits:        0000000098
00 01: global timer bottom 32 bits:     3768157999
00 01: global timer diff:               0000043000
```

perfectly periodic

```
00 01: global timer diff:               0000000000
00 01: global timer diff:               0000043000
00 01: global timer diff:               0000043000
00 01: global timer diff:               0000043000
00 01: global timer diff:               0000043000
00 01: global timer diff:               0000043000
00 01: global timer diff:               0000043000
00 01: global timer diff:               0000043000
```

now it takes only takes 1 TDM revolution for each time stamp

43000 = 1 slot revolutions, each

- 3 * 10000 cycles applications 1, 2, 3
- 1 * 5000 cycles system application 0
- 4 * 2000 cycles VKERNEL

TDM schedule

T3

- copy your tile 0 application 1 `main.c` program to tile 1 application 1
- run the system and check that both programs behave the same

- modify the TDM schedule such that tile 1 application 1 has a slot of 400,000 cycles
- run the system

- why do tile 0 app 1 and tile 1 app 1 have different outputs?
- why does the application on tile 1 produce it's output slower than on tile 0 even though it has more time?
- what happens if you remove the wfi?

answers:

Tutorial

4. partition timer

Kees Goossens
Mojtaba Haghi s.m.haghi@tue.nl
Electronic Systems Group
Electrical Engineering Faculty

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

partition timer

T4

- restore `vep-config.txt` to its original state (tile 0 app 1 has a slot of 10000 cycles)
  - or you can use `git checkout -- vep-config.txt`
- on tiles 0 & 1 app 1 duplicate your code for reading the global timer, but for the partition timer instead

```
xil_printf("global timer top 32 bits:       %010u\n", (uint32_t)(gnow[i]>>32));
xil_printf("global timer bottom 32 bits:    %010u\n", (uint32_t)gnow[i]);
xil_printf("global timer diff:              %010u\n", (uint32_t)(gnow[i]-gnow[i-1]));
xil_printf("partition timer top 32 bits:    %010u\n", (uint32_t)(pnow[i]>>32));
xil_printf("partition timer bottom 32 bits: %010u\n", (uint32_t)pnow[i]);
xil_printf("partition timer diff            %010u\n", (uint32_t)(pnow[i]-pnow[i-1]));
```

- run the system
- why does the partition timer run slower than the global timer?
- why are the partition timers of the two applications the same?

answer:

sample output

T4

- note that the programs on tiles 1 and 2 start later than tile 0
- the run/exec.sh script that runs on the ARM tells the system application to update the TDM schedule
- the scripts take some time and the ARM does not have a predictable timing, which cause a staggered delay between the schedule changes on the cores

```
01 01: global timer diff:                0000043000
00 01: global timer diff:                0000043000
01 01: partition timer diff              0000010005
00 01: partition timer diff              0000010005
01 01: global timer diff:                0000043000
00 01: global timer diff:                0000043000
01 01: partition timer diff              0000010005
00 01: partition timer diff              0000010005
```

t04a.c

- moreover, output shown by `readout.sh` is not always in the order of time stamps
  - an earlier line may in fact have been printed later
- if you print information on the RISC-V cores like this

```
uint64_t t = *g_timer;
xil_printf("%06u/%010u: your message with args\n", (uint32_t)(t>>32), (uint32_t)t, ...);
```

- you can then use the `sort-on-time.sh` script to ensure that the output is shown in order of time stamps

save & restore before next tutorial steps

- save the main.c of applications 0_1 and 1_1 somewhere
- copy application 0_2 (`app_tile_0_2/main.c`) to 0_1 and 1_1
  - or use `gitlab checkout -- app_tile_?_1/main.c`
- all application should now print Hello world again

Tutorial

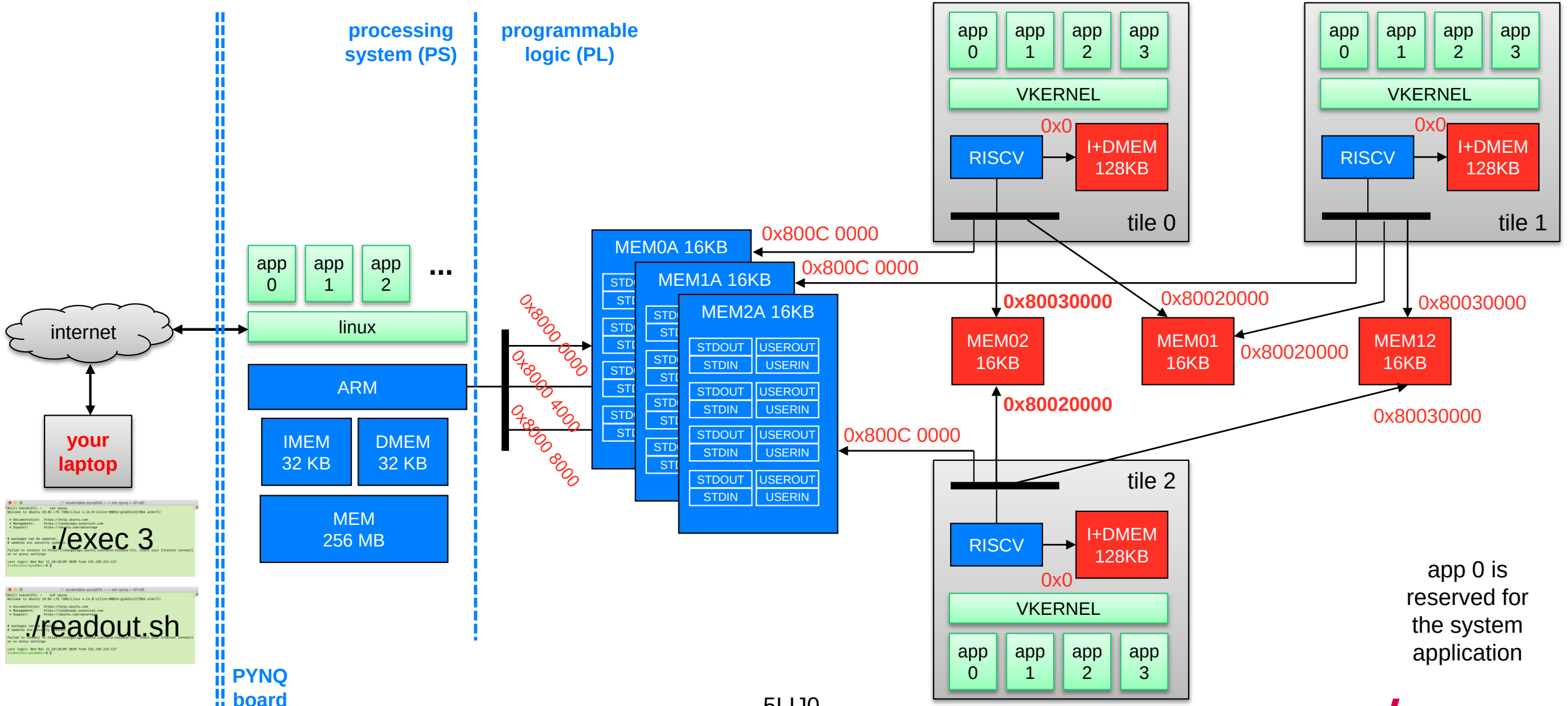
5. RISC-V memories

Kees Goossens
Mojtaba Haghi s.m.haghi@tue.nl
Electronic Systems Group
Electrical Engineering Faculty

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

ARM processor, 3 RISC-V cores, 4 partitions per tile



app 0 is reserved for the system application

arrows go from master to slave

RISC-V memories

T5

- in app 2 on tile 0 declare integers (int32_t) a, b, c
- set a to 3, b to 4, and c equal to their sum
- print their addresses and their values

- in app 3 on tile 0 declare integers (int32_t) a, b, c
- set a to 30, b to 40, and c equal to their sum
- print their addresses and their values

- run the system
- why are the addresses the same
- why are the values different?

answer:

RISC-V shared memories

T5

- in app 2 on tile 0 declare a pointer to an integer (int32_t *) d initialised to address 0x0 in mem01
- store the product of a and b in d
- print the address and the value of d, and then print out the values of a, b, c
- sleep (`asm("wfi");`), then print the values again
- do the same for app 3 on tile 0

- run the system
- why are the addresses for d the same
- why are the values the same and/or different?

answer:

hint:
- use the `tiles/memmap.h` file for the address of mem01
- use the symbolic name tileX_commY rather than the hex address
- `int32_t *f = (int32_t *) the right address;`

```
#define tile0_comm1 0x00020000 /* mem01 in core 0's mem map */
#define tile0_comm2 0x00030000 /* mem02 in core 0's mem map */
#define tile1_comm0 0x00020000 /* mem01 in core 1's mem map */
#define tile1_comm2 0x00030000 /* mem12 in core 1's mem map */
#define tile2_comm0 0x00020000 /* mem02 in core 2's mem map */
#define tile2_comm1 0x00030000 /* mem12 in core 2's mem map */
```