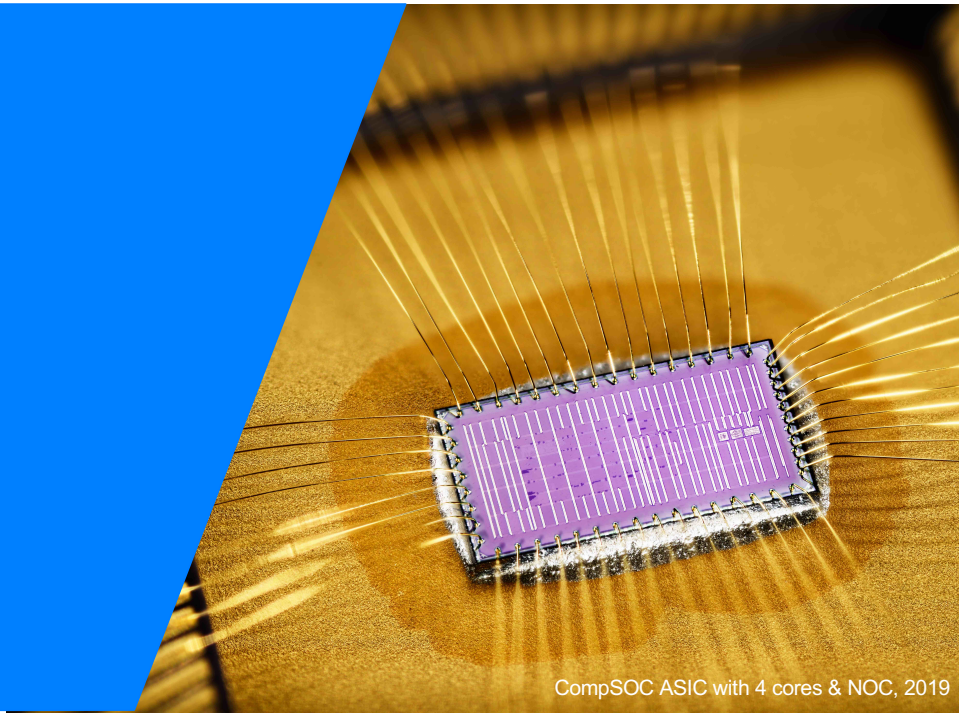
***DO NOT PRINT***

***HIDDEN SOLUTION SLIDES***



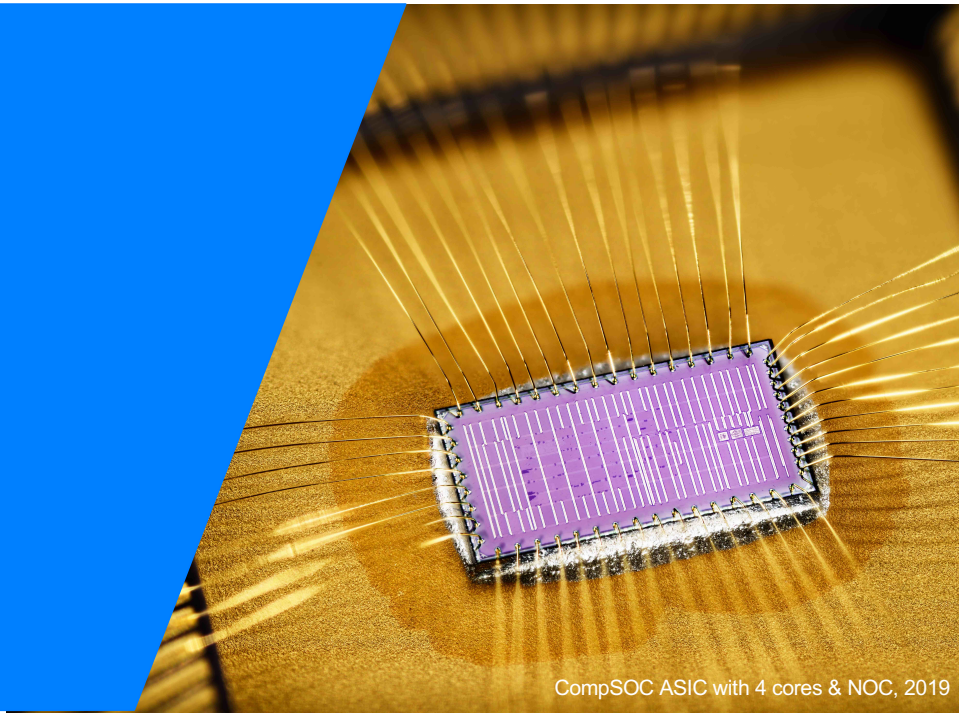
CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# *Getting started tutorial – Part I*

Kees Goossens



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 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, and between RISC-V and ARM cores
- and
  - implement a one-place buffer to safely communicate between cores
  - port & parallelise a fractal application

# using gitlab, PYNQ, and getting the tutorial



- see the CANVAS documentation
  - How to set up your git repository (gitlab, PYNQ)
  - Connect to the PYNQ board using MobaXterm (Windows)
  - Connect to the PYNQ board using Terminal (Linux, Mac)
  - Starting with the tutorial
- see next page

- it is not recommended to use Microsoft Visual Studio (Windows, Mac)
  since it seems to crash boards due to a buggy Ubuntu networking driver on the PYNQ boards

# Example output

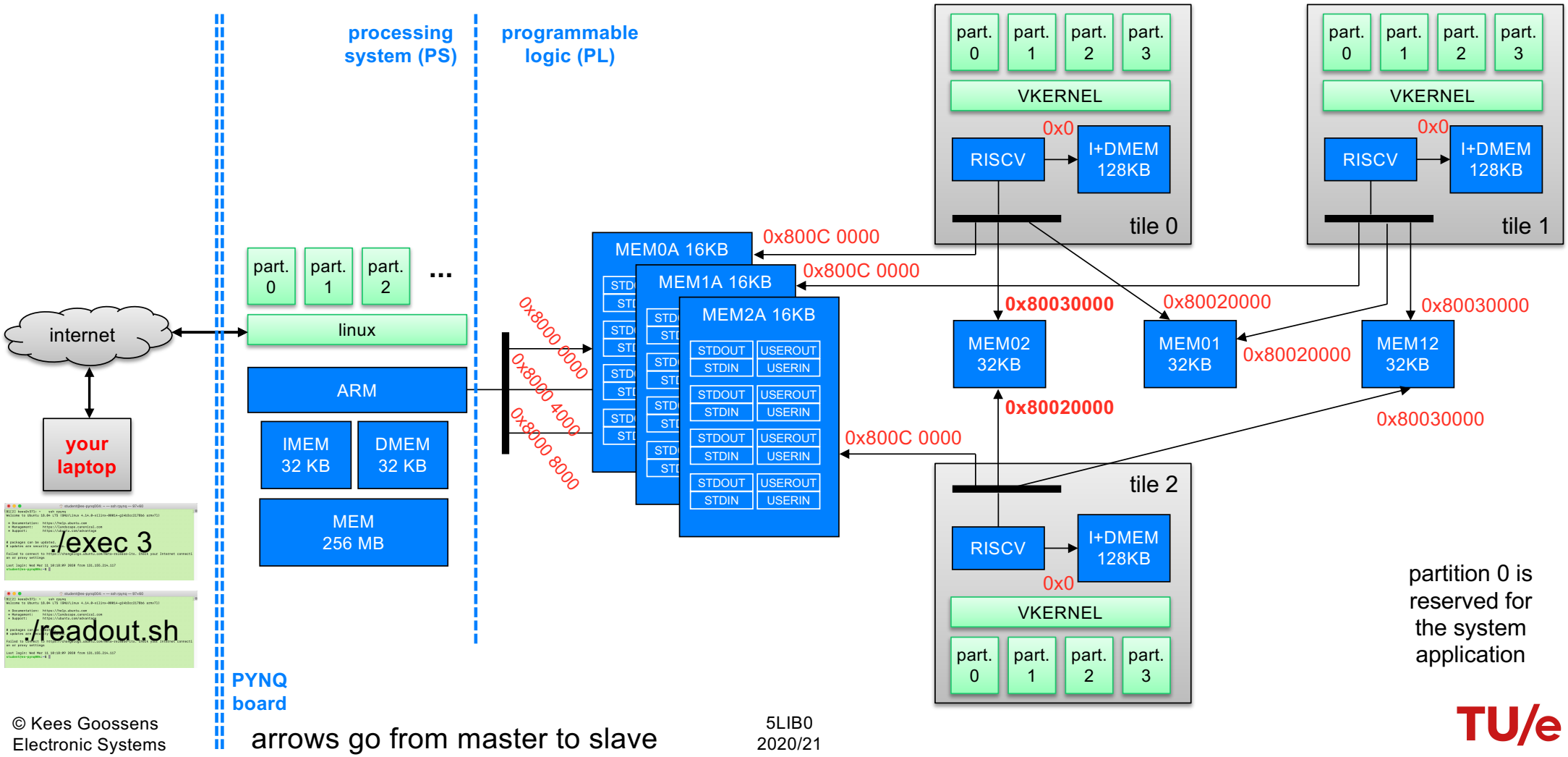
```
student@es-pynq004:~$ git clone git@gitlab.tue.nl:5lib0/5lib0-2020/group-50.git
Cloning into 'group-50'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
student@es-pynq004:~$ ls
group-50
student@es-pynq004:~$ cd group-50/
student@es-pynq004:~/group-50$ ls
README.md
student@es-pynq004:~/group-50$ mkdir tutorial
student@es-pynq004:~/group-50$ ls
README.md  tutorial
student@es-pynq004:~/group-50$ cd tutorial
student@es-pynq004:~/group-50/tutorial$ ls
student@es-pynq004:~/group-50/tutorial$ touch README.txt
student@es-pynq004:~/group-50/tutorial$ ls
README.txt
student@es-pynq004:~/group-50/tutorial$ git add README.txt
student@es-pynq004:~/group-50/tutorial$ git status
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
new file:   README.txt
student@es-pynq004:~/group-50/tutorial$
```

tip: “git status” will always give hints on what to do next

```
student@es-pynq004:~/group-50/tutorial$ git commit -m "added tutorial directory"
[master ec04156] added tutorial directory
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tutorial/README.txt
student@es-pynq004:~/group-50/tutorial$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
student@es-pynq004:~/group-50/tutorial$ git push
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 331 bytes | 331.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To gitlab.tue.nl:5lib0/2019-2020/group-50.git
   4b1e19c..ec04156  master -> master
student@es-pynq004:~/group-50/tutorial$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
student@es-pynq004:~/group-50/tutorial$ cd
student@es-pynq004:~$ git clone https://gitlab.tue.nl/kgoossens/2020-2021-tutorial.git ~/tutorial-stud1
Cloning into '/home/student/tutorial-stud1'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
student@es-pynq004:~$ ls
group-50  tutorial-stud1
student@es-pynq004:~$
```

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






partition 0 is reserved for the system application

arrows go from master to slave





TU/e

# file structure



| | |
|---|---|
| `Makefile` | • clean or compile all VEPs |
| `vep_0` | • vep_0 contains the system application and all libraries – do not modify |
| `vep_1` | • vep_1 contains all the files for the application in VEP 1 |
| `vep_2` | • similarly for VEPs 2, 3, 4 |
| `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` | |
| `tools` | • system files – do not modify |

- in most of the tutorial we will work with one VEP only (see next slide)

# file structure



```
vep_1
        Makefile
        app_arm_echo
                Makefile
                *.[ch]
        partition_0_1
                Makefile
                *.[ch]
        shared

        shared_memories
                vep_private_memory.h
                vep_public_memory.h
                vep_shared_memories.c
        vep-config.txt
```

- vep_1 contains all the files for the application in VEP 1
  - clean or compile only this VEP
  - files for ARM application (here, the echo application)
    - clean or compile only this application
    - source files
  - files for partition 1 on RISC-V 0
    - clean or compile only this partition
    - source files
  - your files shared between all partitions in this VEP
    - useful for common typedefs
  - files shared between all partitions in this VEP
    - your def. of VEP's private data structures in shared memories
    - your def. of VEP's public data structures in shared memories
    - system file, do not modify
  - reserve (shared) memory and TDM slots for this VEP

# terminology



- partition: one executable on one core, with resources:
  - one region in instruction/data memory
  - zero or more TDM slots
  - stdin/out & user-in/out FIFOs ARM-RISC-V memory (arm0/1/2a)
- virtual execution platform (VEP) consists of one or more partitions on one or more tiles, with resources:
  - at most one private region per shared memory (mem01, 02, 12)
  - at most one public region per shared memory (mem01, 02, 12)
- one application per VEP
  - an application could consist of multiple VEPs, but we won't consider that possibility here
- system application
- microkernel / VKERNEL
- global timer
- partition timer

# compiling and running



- `make clean`
  - remove all generated files (executables, etc.)
  - `make veryclean` also removed compiled system libraries
- `make`
  - to compile your programs into a executables that will be uploaded
  - calls the `Makefile` for all `app/partion_*` 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, is otherwise the same
- `stop.sh`
  - stop all running RISC-V programs
- `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` otherwise they won't work

# compiling and running



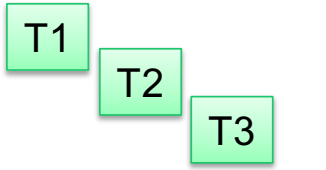
- `run.sh`
  - to compile, upload executables to the RISC-V cores, program TDM of VKERNEL, and run the programs
  - `rerun.sh` is same but doesn't recompile
- `stop.sh`
  - stop all running RISC-V programs
- `exec.sh s` is equal to: `run.sh; sleep s; stop.sh`
  - useful if there's lots of output and you may not be able to run `stop.sh`
- execute these scripts by typing `./run.sh ./rerun.sh ./stop.sh ./exec.sh ./readout.sh` on the command
- `sudo reboot`
  - reload the Verintec platform on the FPGA by rebooting the board
  - this realigns the TDM tables if required

- advanced, for later:
  - `make tdm memmap memmap-physical`
    - show TDM table / logical memory map / physical memory map for all VEPs

# tutorial overview

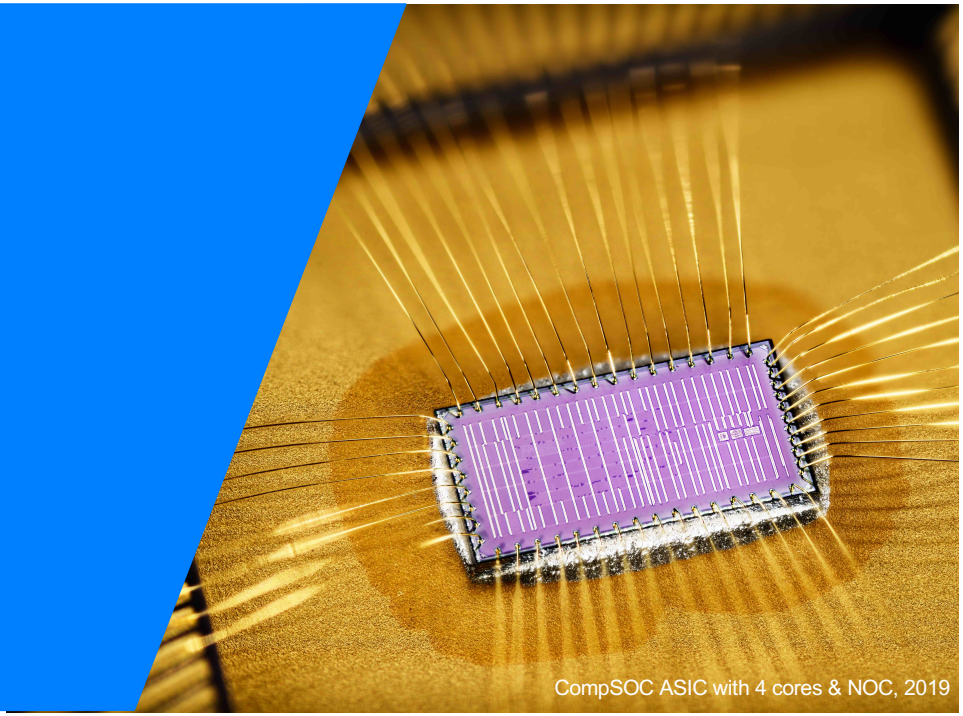


1. running the example platform
2. global timer
3. TDM schedule
4. partition timer
5. RISC-V memories
6. time-based synchronisation
7. data-based synchronisation
8. 1-place buffer
9. ARM-RISC-V communication
10. fractal



# *Tutorial*

## *1. getting started*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# how to run applications on the PYNQ

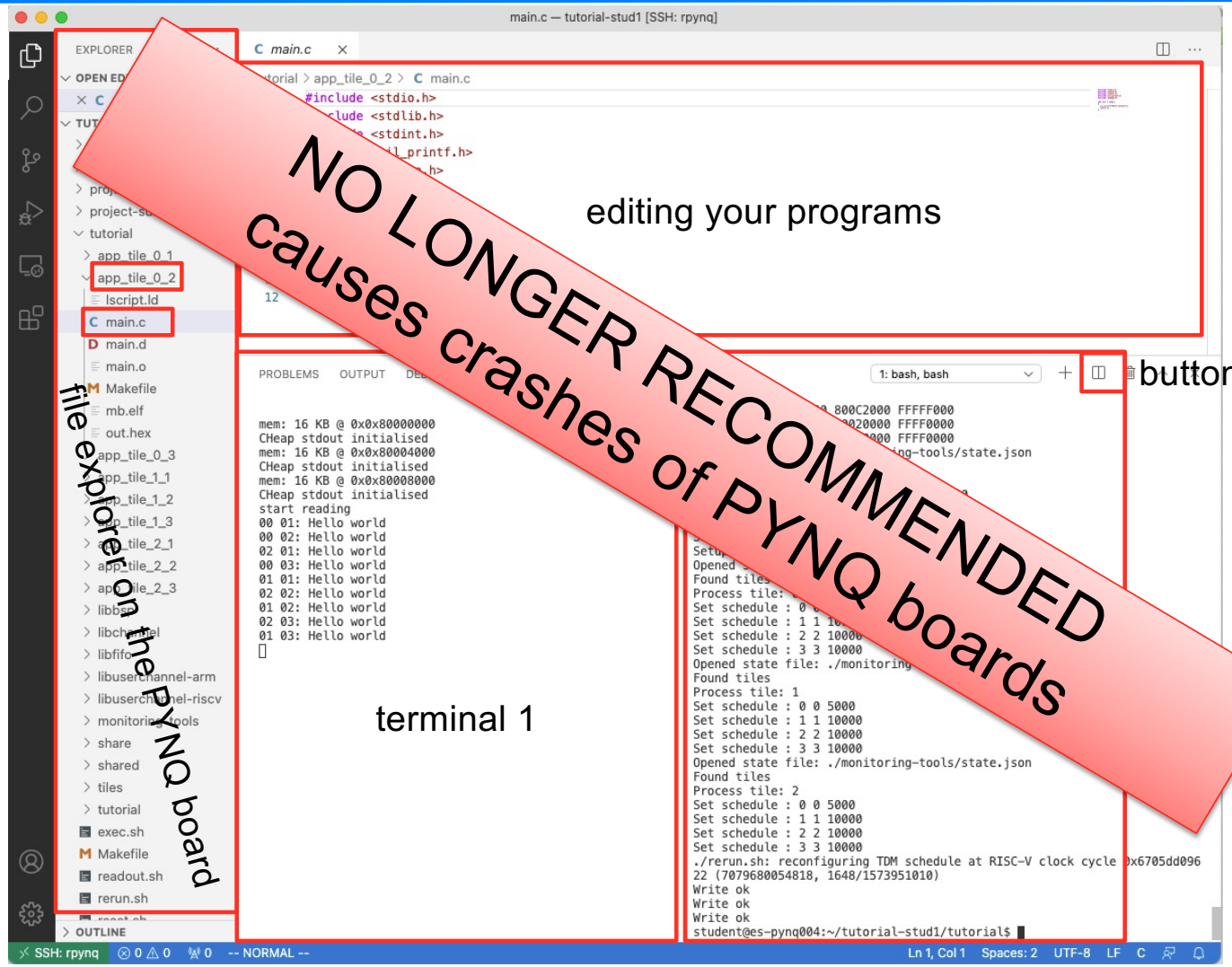


- you need to access the PYNQ board
- you can use
  - MobaXterm (Windows)
  - Terminal (MacOS, Linux)

- in all cases you need to setup the ssh connection and then use terminals on the PYNQ board
- in the following I will show output using Terminal on a Mac

# Visual Studio on a Mac








TU/e

# 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, then tutorial again (`cd tutorial-stud1; cd tutorial`)
- wait until the `make` (5) has finished before typing `./readout.sh` (6)
- finally, type `./exec.sh 10`

# how to run applications on the PYNQ

T1



screenshots to be updated

after typing `./readout.sh` in this terminal,
and `./exec.sh 10` in the other terminal
this should be the output of running the example

```
student@es-pynq004:~/tutorial-stud1/tutorial$ ls
Makefile       app_tile_1_3/  libchannel/            rerun.sh*         stop.sh*
app_tile_0_1/  app_tile_2_1/  libfifo/              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/            vep-config.txt
app_tile_1_1/  exec.sh*       monitoring-tools/      shared/
app_tile_1_2/  libbsp/        readout.sh*            sort-on-time.sh*
student@es-pynq004:~/tutorial-stud1/tutorial$ ./readout.sh
mem: 16 KB @ 0x0x80000000
CHeap stdout initialised
mem: 16 KB @ 0x0x80004000
CHeap stdout initialised
mem: 16 KB @ 0x0x80008000
CHeap stdout initialised
start reading
00 01: Hello world
01 01: Hello world
02 01: Hello world
00 02: Hello world
02 02: Hello world
01 02: Hello world
00 03: Hello world
02 03: Hello world
01 03: Hello world
```

initialise the FIFOs in
`mem0a, mem1a, mem2a`

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

```
Setup: 3 2 80020000 80020000 FFFF00
Setup: 3 3 80030000 80030000 FFFF0000
Opened state file: ./monitoring-tools/state
Found tiles
Process tile: 0
Set schedule : 0 0 5000
Set schedule : 1 1 10000
Set schedule : 2 2 10000
Set schedule : 3 3 10000
Opened state file: ./monitoring-tools/state.json
Found tiles
Process tile: 1
Set schedule : 0 0 5000
Set schedule : 1 1 10000
Set schedule : 2 2 10000
Set schedule : 3 3 10000
Opened state file: ./monitoring-
Found tiles
Process tile: 2
Set schedule : 0 0 5000
Set schedule : 1 1 10000
Set schedule : 2 2 10000
Set schedule : 3 3 10000
./rerun.sh: reconfiguring TDM schedule at RISC-V clock cycle 0x627e2a63
3391)
Write ok
Write ok
Write ok
Stopping all VPs
4 active entries
Stopping all VPs
4 active entries
Stopping all VPs
4 active entries
student@es-pynq004:~/tutorial-stud1/tutorial$
```

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

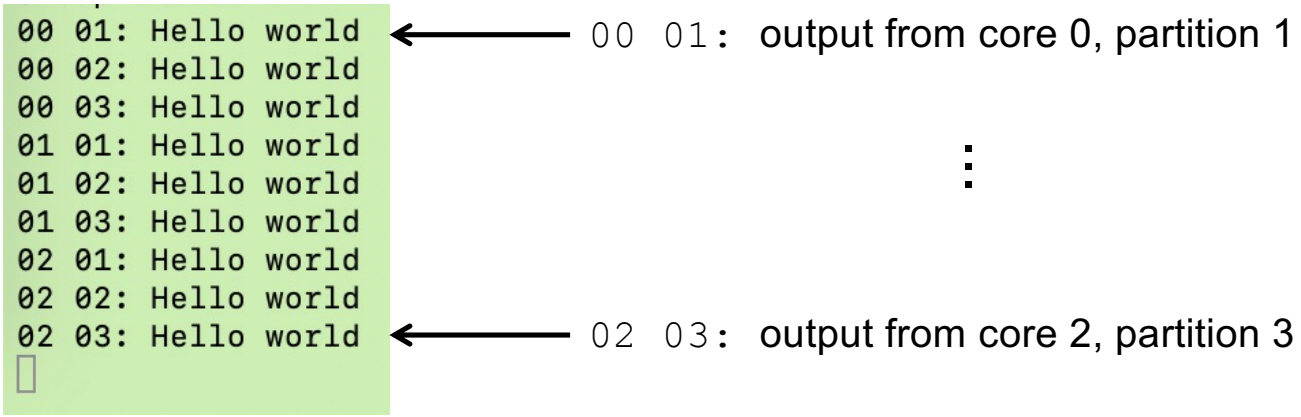
stop all partitions on tiles 0, 1, 2 after 5 seconds

# how to run applications on the PYNQ

T1



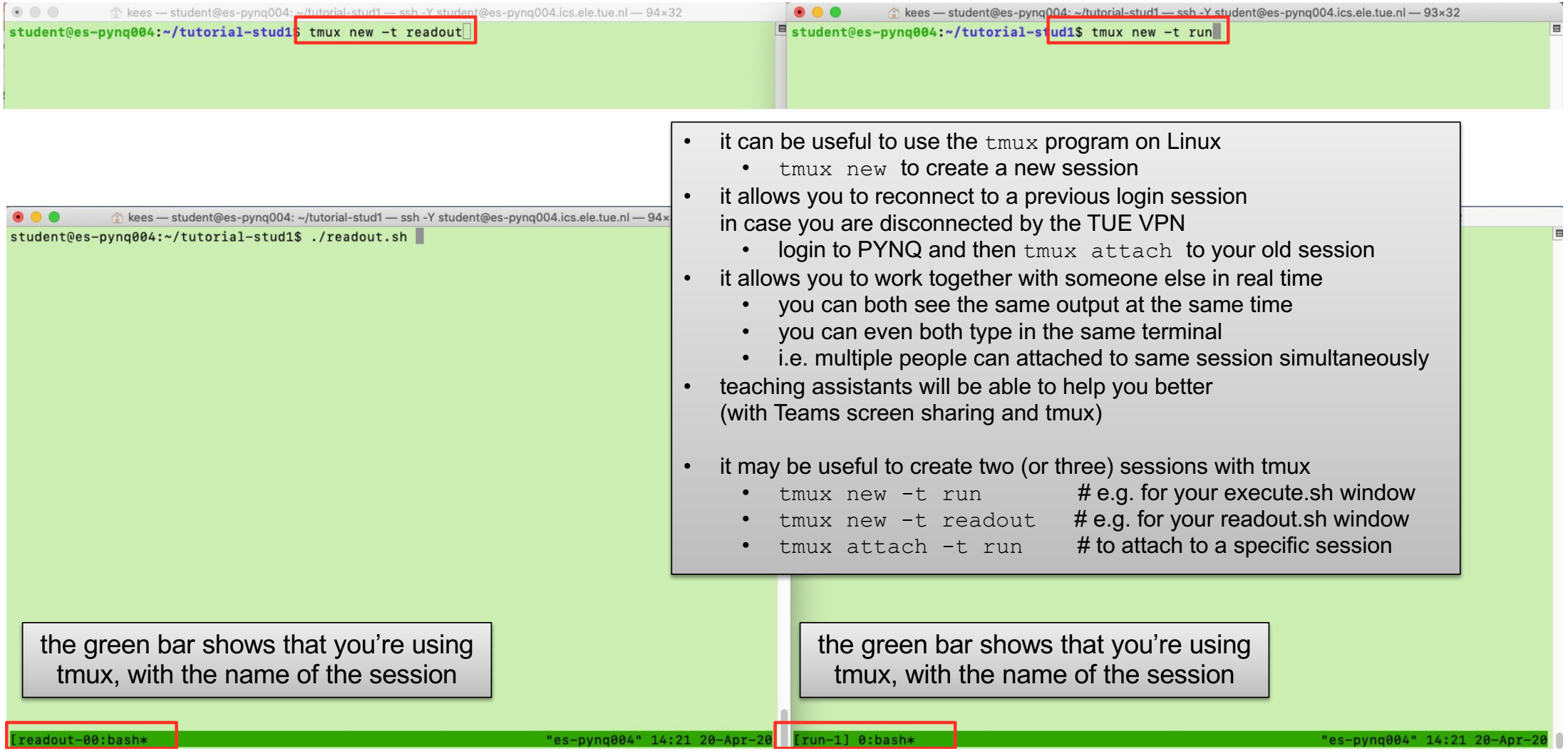
- output of the RISC-V cores




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

# may be useful: tmux (terminal multiplexer)






- it can be useful to use the `tmux` program on Linux
  - `tmux new` to create a new session
- it allows you to reconnect to a previous login session in case you are disconnected by the TUE VPN
  - login to PYNQ and then `tmux attach` to your old session
- it allows you to work together with someone else in real time
  - you can both see the same output at the same time
  - you can even both type in the same terminal
  - i.e. multiple people can attached to same session simultaneously
- teaching assistants will be able to help you better (with Teams screen sharing and tmux)

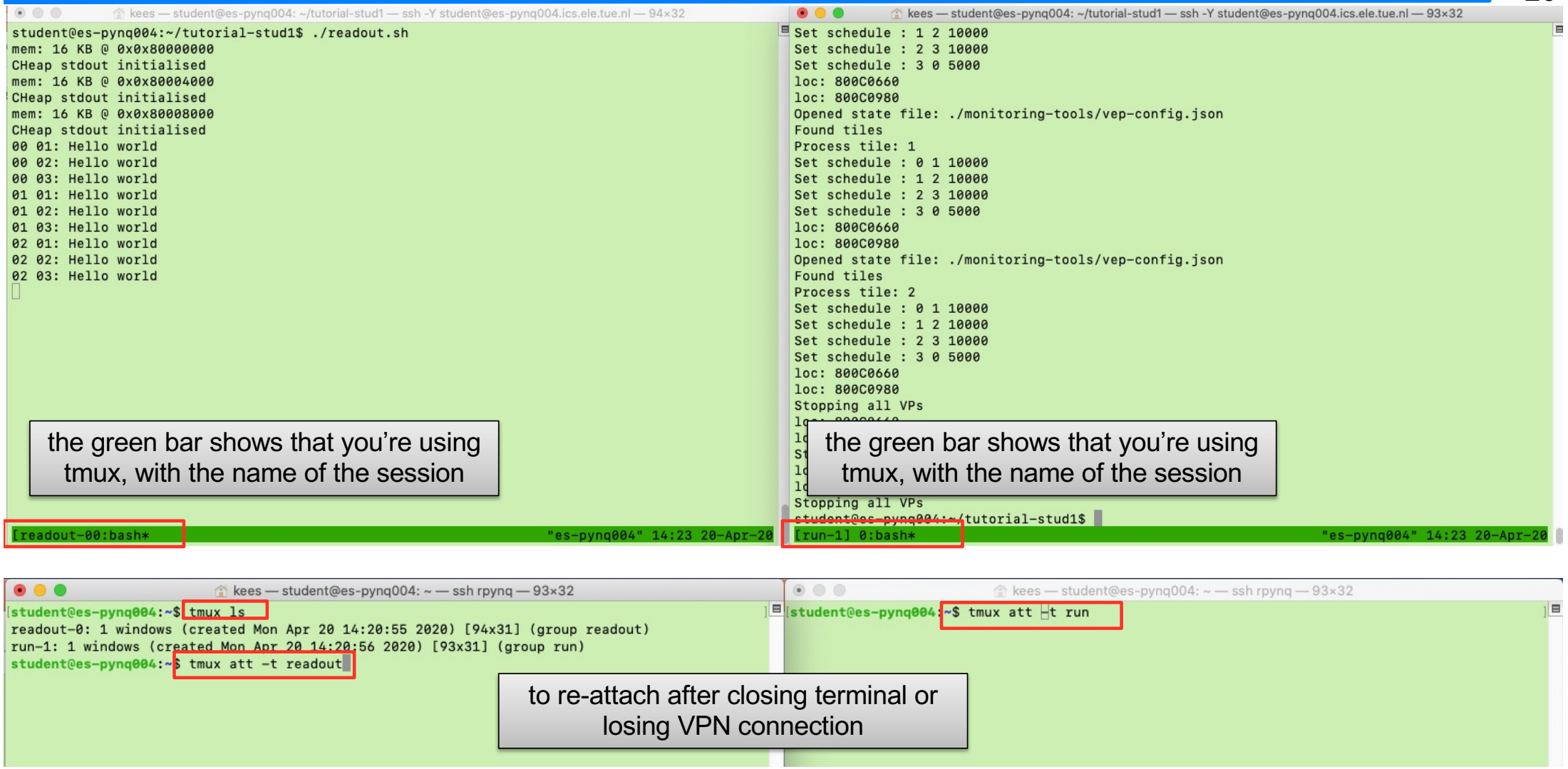
- it may be useful to create two (or three) sessions with tmux
  - `tmux new -t run` # e.g. for your execute.sh window
  - `tmux new -t readout` # e.g. for your readout.sh window
  - `tmux attach -t run` # to attach to a specific session

the green bar shows that you're using tmux, with the name of the session

the green bar shows that you're using tmux, with the name of the session

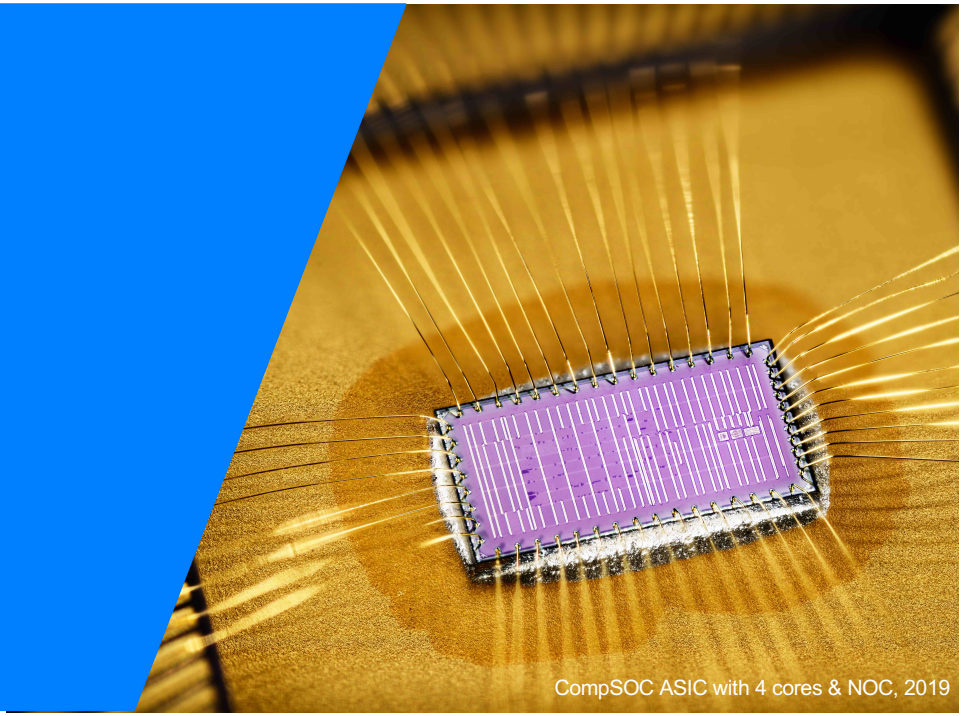
# may be useful: tmux (terminal multiplexer)





# *Tutorial*

# *2. global timer*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# timers

T2



- each RISC-V tile has 2 timers

- global timer
  - `volatile uint64_t *g_timer = (uint64_t*)TILE0_GLOBAL_TIMER;`
  - always running
  - can be used to compute the precise elapsed time (from start to finish) – i.e. the response time (RT)
  - global 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*)TILE0_PARTITION_TIMER;`
  - 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 partition has executed – i.e. the execution time (ET)

# timers

T2



- modify the `main.c` program of tile 0, partition 1 (`partition_0_1/main.c`)
- to add the declaration of the two timers and a temporary variable t

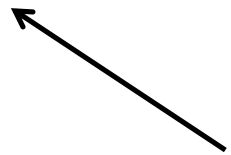
```
volatile const uint64_t * const g_timer = (uint64_t*)TILE0_GLOBAL_TIMER;
volatile const uint64_t * const p_timer = (uint64_t*)TILE0_PARTITION_TIMER;
```

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 see the following slides for more information

# what happened here?



- a discontinuity in the time?

```
00 01: 0011/4275576835: Hello world
01 01: 0011/4277317835: Hello world
02 01: 0011/4279124835: Hello world
00 02: 0011/4279146835: Hello world
01 02: 0011/4279146835: Hello world
02 02: 0011/4279146835: Hello world
00 03: 0011/4279168835: Hello world
01 03: 0011/4279168835: Hello world
02 03: 0011/4279168835: Hello world
00 01: 0012/0003544539: Hello world
01 01: 0012/0005285539: Hello world
02 01: 0012/0007092539: Hello world
00 02: 0012/0007114539: Hello world
01 02: 0012/0007114539: Hello world
02 02: 0012/0007114539: Hello world
02 03: 0012/0007136539: Hello world
00 03: 0012/0007136539: Hello world
01 03: 0012/0007136539: Hello world
```

# what happened here?

T2



- what appears to be a discontinuity in the time
  is just the wrapping of the lower 32 bits of the 64 bit counter

```
00 01: 0011/4275576835: Hello world
01 01: 0011/4277317835: Hello world
02 01: 0011/4279124835: Hello world
00 02: 0011/4279146835: Hello world
01 02: 0011/4279146835: Hello world
02 02: 0011/4279146835: Hello world
00 03: 0011/4279168835: Hello world
01 03: 0011/4279168835: Hello world
02 03: 0011/4279168835: Hello world
00 01: 0012/0003544539: Hello world
01 01: 0012/0005285539: Hello world
02 01: 0012/0007092539: Hello world
00 02: 0012/0007114539: Hello world
01 02: 0012/0007114539: Hello world
02 02: 0012/0007114539: Hello world
02 03: 0012/0007136539: Hello world
00 03: 0012/0007136539: Hello world
01 03: 0012/0007136539: Hello world
```

# timers – this is reference info, can be skipped now

T2



- this does work not if the partition 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



```
inline uint64_t read_global_timer() {
  volatile const uint64_t * const timer = (uint64_t *) TILE0_GLOBAL_TIMER;
  volatile const uint32_t * const timer_ls = (uint32_t *) TILE0_GLOBAL_TIMER;
  uint64_t t1;
  uint32_t t2_ls;
  while (1) {
    t1 = *timer;
    t2_ls = *timer_ls;
    if (t2_ls >= ((uint32_t) t1) + 12 && t2_ls <= ((uint32_t) t1) + 18) return t1;
  }
}
inline uint64_t read_partition_timer() {
  volatile const uint64_t * const timer = (uint64_t *) TILE0_PARTITION_TIMER;
  volatile const uint32_t * const timer_ls = (uint32_t *) TILE0_PARTITION_TIMER;
  uint64_t t1;
  uint32_t t2_ls;
  while (1) {
    t1 = *timer;
    t2_ls = *timer_ls;
    if (t2_ls >= ((uint32_t) t1) + 12 && t2_ls <= ((uint32_t) t1) + 18) return t1;
  }
}
```

this is part of `timers.h` in `libbsp` which you can use by `#include <timers.h>` including in your code

to be checked – may be updated in new hardware

# 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

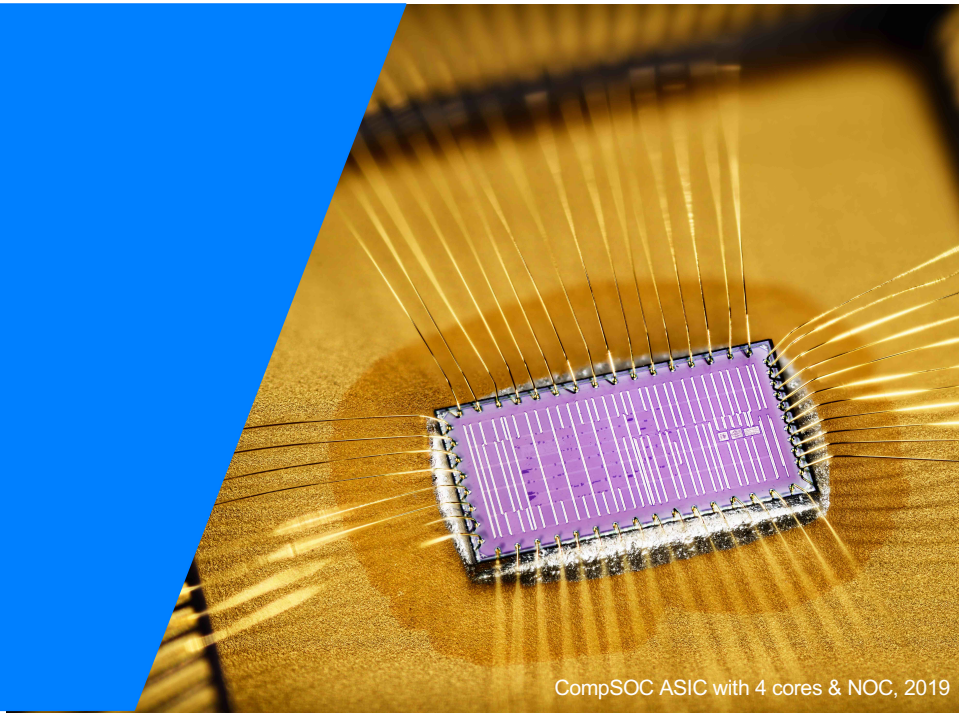
- when the ARM does not accept the stdout data then the RISC-V cores will stall
  - this process is called flow control or back-pressure
- you will see this in the time stamps because the RISC-V programs will run slower

- also, notice that `xil_print` is slow!
- for example, `xil_printf("%d\n",1234567890);`
- takes 3657 clock cycles: 10 divisions of 30 cycles each, modulo (mult, div), formatting, inserting each byte into the FIFO to the ARM, etc.

now commit your T2 solution to your gitlab repo

# *Tutorial*

# *3. TDM schedule*


CompSOC ASIC with 4 cores & NOC, 2019

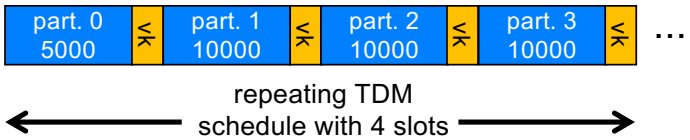
**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# TDM schedule



- the `vep-config.txt` file defines the time-division multiplexing (TDM) slot table
  - i.e. which partitions run and how much time they get
  - contains 1 to 32 slots
  - the system application (0) requires at least one slot
  - the VKERNEL runs between slots for 2000 cycles
  - a partition may have more than 1 slots
  - you are allowed to have gaps between slots; they will be allocated to the system partition




- as shown on the right, on every tile
  - every partition has 1 slot of 10000 clock cycles
  - part. 0 (system application) is automatically added and always runs in the first slot 0 for 5000 clock cycles
  - thus the first user slot is starts at 7000

original vep-config.txt

```
on tile 0 partition 1 has 10000 cycles of 43000 starting at  7000
on tile 0 partition 2 has 10000 cycles of 43000 starting at 19000
on tile 0 partition 3 has 10000 cycles of 43000 starting at 31000
on tile 1 partition 1 has 10000 cycles of 43000 starting at  7000
on tile 1 partition 2 has 10000 cycles of 43000 starting at 19000
on tile 1 partition 3 has 10000 cycles of 43000 starting at 31000
on tile 2 partition 1 has 10000 cycles of 43000 starting at  7000
on tile 2 partition 2 has 10000 cycles of 43000 starting at 19000
on tile 2 partition 3 has 10000 cycles of 43000 starting at 31000
```

output of run.sh

```
Process tile: 0
Set schedule : 0 0 5000
Set schedule : 1 1 10000
Set schedule : 2 2 10000
Set schedule : 3 3 10000
Opened state file: ./tools/state.json
Found tiles

Process tile: 1
Set schedule : 0 0 5000
Set schedule : 1 1 10000
Set schedule : 2 2 10000
Set schedule : 3 3 10000
Opened state file: ./tools/state.json
Found tiles

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

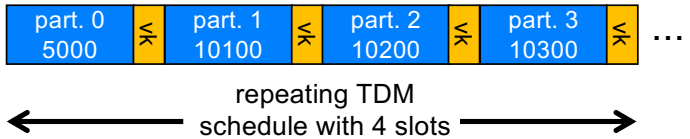
the slots are listed in order

# TDM schedule

T3



- modify the TDM schedule to see what happens
- as shown on the right, on tile 0
  - part. 1 (partition_0_1/main.c)
    runs in slot 1 for 10100 clock cycles
  - ...




modified vep-config.txt

```
on tile 0 partition 1 has 10100 cycles of 43000 starting at  7000
on tile 0 partition 2 has 10200 cycles of 43000 starting at 19000
on tile 0 partition 3 has 10300 cycles of 43000 starting at 31000
on tile 1 partition 1 has 10600 cycles of 43000 starting at  7000
on tile 1 partition 2 has 10000 cycles of 43000 starting at 19000
on tile 1 partition 3 has 10000 cycles of 43000 starting at 31000
on tile 2 partition 1 has 10600 cycles of 43000 starting at  7000
on tile 2 partition 2 has 10000 cycles of 43000 starting at 19000
on tile 2 partition 3 has 10000 cycles of 43000 starting at 31000
```

note: you need to change more!

output of run.sh

```
Process tile: 0
Set schedule : 0 0 5000
Set schedule : 1 1 10100
Set schedule : 2 2 10200
Set schedule : 3 3 10300
Opened state file: ./tools/state.json
Found tiles

Process tile: 1
Set schedule : 0 0 5000
Set schedule : 1 1 10600
Set schedule : 2 2 10000
Set schedule : 3 3 10000
Opened state file: ./tools/state.json
Found tiles

Process tile: 2
Set schedule : 0 0 5000
Set schedule : 1 1 10600
Set schedule : 2 2 10000
Set schedule : 3 3 10000
```





TU/e

# TDM schedule

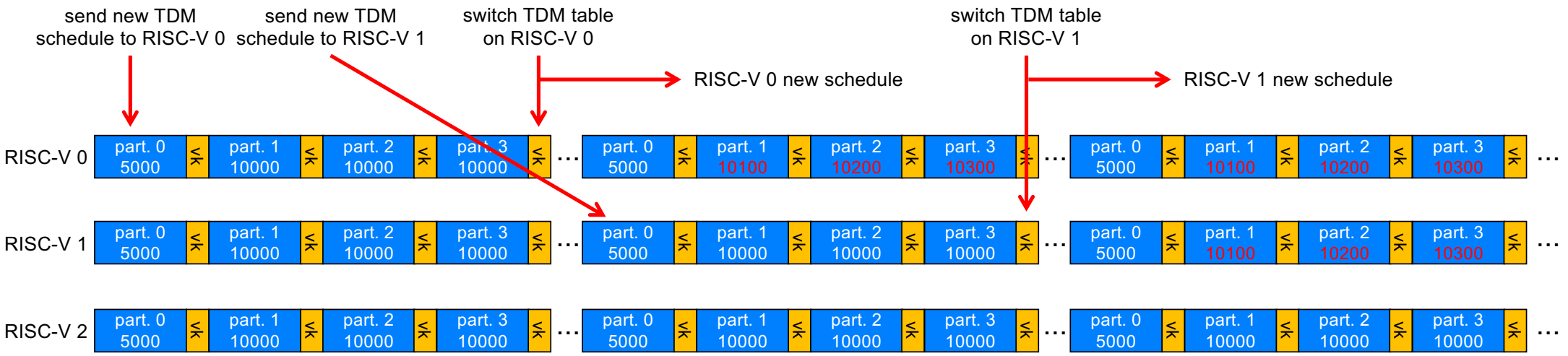


- to (re)start applications the ARM
  - stops all partitions on all RISC-V
  - then sends the partition binaries to the system application on each RISC-V
  - then zeroes out the shared memories
  - then sends a new TDM schedule to each RISC-V
- all this takes some time
- applications are started & stopped at different points in time on the different RISC-V cores
- examples of doing it wrong:
  - zero shared memories before stopping partitions
  - sending new binaries before stopping partitions
  - updating RISC-V 0 before RISC-V 1 & 2
- the really correct way:
  1. stop all partitions on all tiles by sending a new TDM schedule
     in which the TDM slots of the partitions are allocated to the system partition
  2. send new binaries to all tiles
  3. zero out all shared memories
  4. send new TDM schedules to all tiles

# TDM schedule switching



- this still does not solve all problems!
- we also need to ensure that the TDM tables are switched at the same point on all RISC-V cores

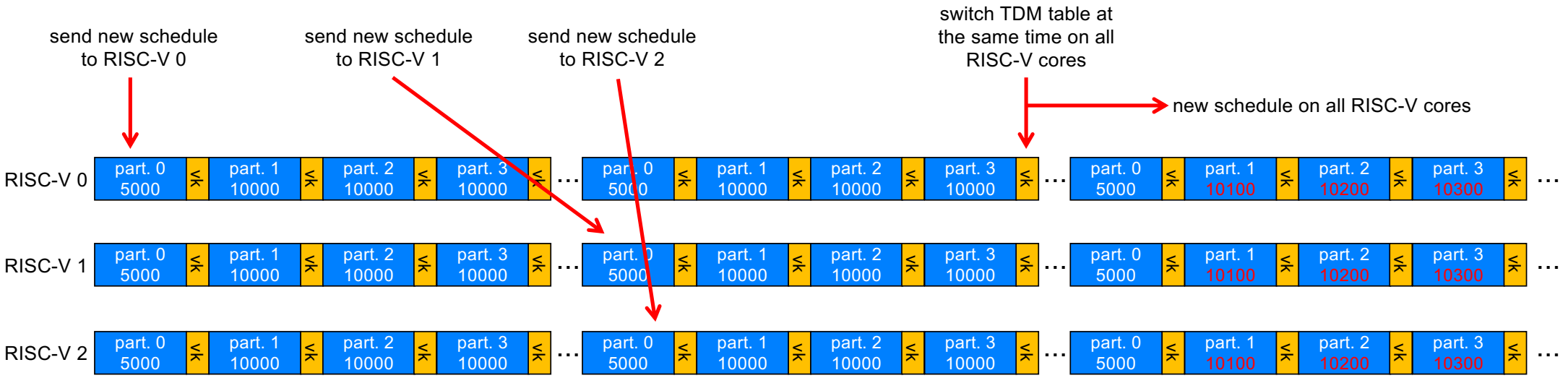



example of non-aligned TDM schedule switching

# TDM schedule switching



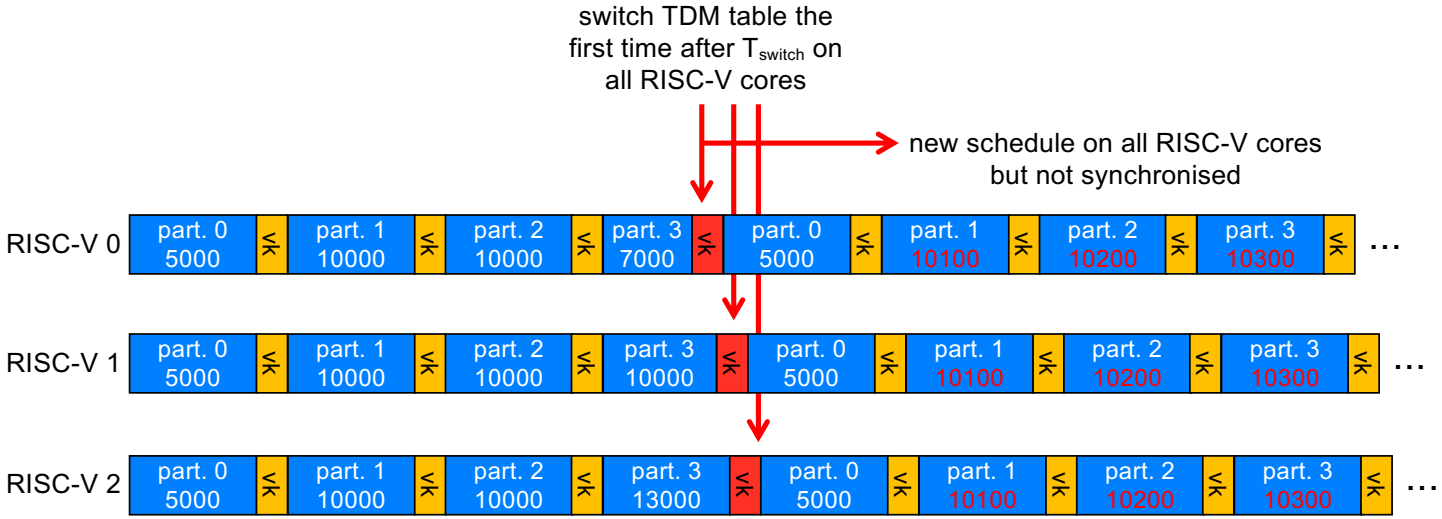
- therefore the ARM sends to the system application on each RISC-V
  - a new TDM schedule
  - the global RISC-V time $T_{switch}$ at which to switch to the new schedule
- the VKERNEL switches to the new schedule at the last VKERNEL slot in the TDM table
- at the earliest global RISC-V time after $T_{switch}$
- $T_{switch}$ must sufficiently far in the future (default is 2 seconds)



# TDM schedule (re)alignment



- all the previous techniques only work if the point in time at which the VKERNEL changes to the new schedules is the same on all RISC-V
- in other words, the TDM schedules should be synchronised to stay synchronised
- if, for some reason, you end up with unsynchronised TDM tables and you want to resynchronise, then you can either
  - `sudo reboot` the PYNQ board
  - or manually resynchronise (how would you do this?)



# TDM schedule switching



- the (re)run.sh scripts print the total TDM lengths
- you can use this to check if you kept them aligned

```
./rerun.sh: info: stop all partitions
./rerun.sh: info: (re)loading all partitions
info: slot table lengths are: 43600 43600 43600 cycles
```

- they also print Tswitch in three formats
- the actual switching will take place at the first end of the TDM table on or after that time

```
Process tile: 2
Set schedule : 0 0 5000
Set schedule : 1 1 10600
Set schedule : 2 2 10000
Set schedule : 3 3 10000
./rerun.sh: reconfiguring TDM schedule at RISC-V clock cycle 0x6343bbdf80b (6821410371595, 1588/1002305547)
```

MSB | 32 LSB

64 bit hexadecimal | 64 bit decimal | 32 MSB decimal | 32 LSB decimal

# memory & stack



- at some point you will need to increase the memory or stack size of programs on the RISC-V
- see the `vep-config.txt` file, e.g.

```
##
## memory and stack allocation for each partition
## - per tile the system application with 32K is always added
## - 8K, 16K, 32K, or 64K per partition
## - max 128K per tile
##
on tile 0 partition 1 has 4K stack in 32K memory starting at 32K
on tile 0 partition 2 has 4K stack in 32K memory starting at 64K
on tile 0 partition 3 has 4K stack in 32K memory starting at 96K
on tile 1 partition 1 has 4K stack in 32K memory starting at 32K
on tile 1 partition 2 has 4K stack in 32K memory starting at 64K
on tile 1 partition 3 has 4K stack in 32K memory starting at 96K
on tile 2 partition 1 has 4K stack in 32K memory starting at 32K
on tile 2 partition 2 has 4K stack in 32K memory starting at 64K
on tile 2 partition 3 has 4K stack in 32K memory starting at 96K
```

- memory must be 8K, 16K, 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
- first 32K are reserved for partition 0
- memory must be aligned (128 % size == 0)

# memory & stack



- you'll probably run out of memory at some point:

```
/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 | partition_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



- restore the vep-config.txt to its default (10000 cycles for all applications)
- 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;
}
```

- explain what happens to the timer values
- hint: have a look at the slot table lengths reported by run.sh and the timer difference

answer:

# sample output



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
00 01: global timer diff:           0000129000
```

- can you explain the value 129000?

answer:

# sample output



t03a.c

```
mem: 16 KB @ 0x0x80000000
CHeap stdout initialised
mem: 16 KB @ 0x
CHeap stdout in
mem: 16 KB @ 0x
CHeap stdout in
00 01: Hello wo
00 02: Hello wo
00 01: global t
00 01: global t
00 03: Hello wo
00 01: global t
00 01: global t
00 01: global t
00 01: global t
00 01: global t
00 01: global t
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
```

answer:
129000 = 3 slot revolutions of 43000 cycles each

- 1 * 5000 cycles system partition 0
- 3 * 10000 cycles partitions 1, 2, 3
- 4 * 2000 cycles VKERNEL

therefore it takes < 129000 cycles to execute one while loop iteration
then the `asm("wfi")` sleeps until the start of the next slot

use the output of rerun.sh:

```
./rerun.sh: info: stop all partitions
./rerun.sh: info: (re)loading all partitions
info: slot table lengths are: 43000 43000 43000 cycles
```

```
0000129000
0000129000
0000129000
0000129000
0000129000
0000129000
0000129000
0000129000
0000129000
0000129000
0000129000
```

# accurate timing

T3



- printing with `xil_printf` is quite slow (that's why it takes 3 TDM slots to print out the 3 lines of text!)
- to avoid slowing down your program while doing measurements, try the following:
  - 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 with length of 1 TDM table revolution

# 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
```

```
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
```

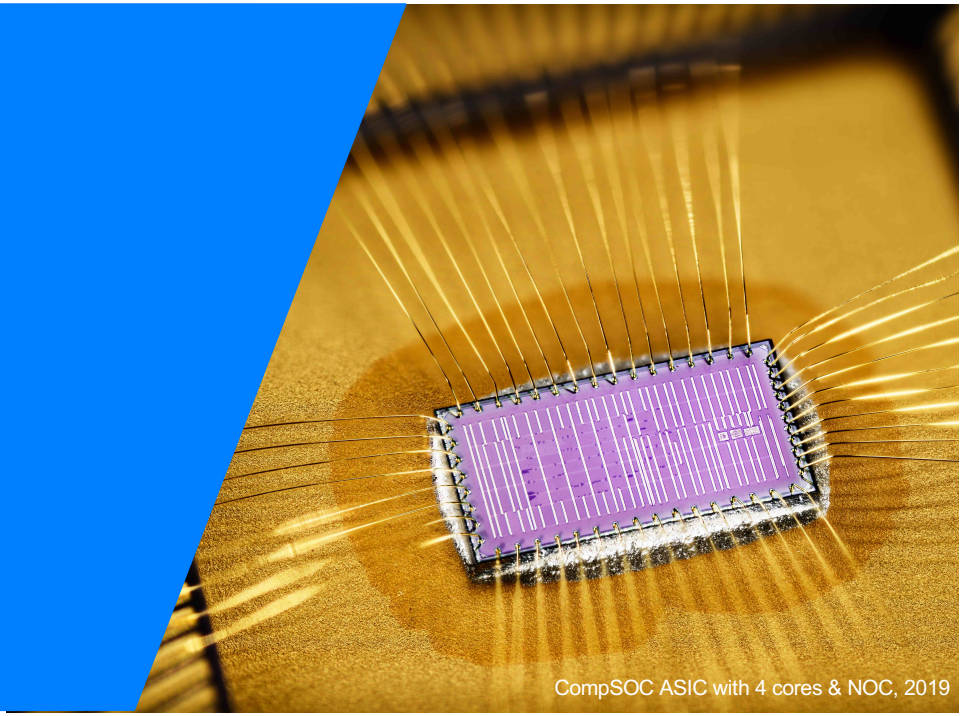
now it takes only takes 1 TDM revolution for each time stamp

43000 = 1 TDM table revolution

- 3 * 10000 cycles partitions 1, 2, 3
- 1 * 5000 cycles system partition 0
- 4 * 2000 cycles VKERNEL

# *Tutorial*

# *4. partition timer*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# partition timer

T4



- restore `vep-config.txt` to its original state (all partitions have a slot of 10000 cycles)
  - or you can use `git checkout -- vep-config.txt`
- on tiles 0 & 1 part. 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 partitions the same?

answer:

# partition timer

T4



- restore `vep-config.txt` to its original state (all partitions have a slot of 10000 cycles)
  - or you can use `git checkout -- vep-config.txt`
- on tiles 0 & 1 part. 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 partitions the same?

answer:
- the partition timer only runs when the partition is active
- we run the same program in two independent partitions and hence the output should be the same

# sample output

T4



t04a.c

- the output shown by `readout.sh` is always in the right order per RISC-V
- but the output of different RISC-V cores may be arbitrarily interleaved
- with later output of core x shown before earlier output of core y
- it is therefore a good idea to print the global time with every message

- 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

```
00 01: global timer top 32 bits:       0000001659
01 01: global timer top 32 bits:       0000001659
00 01: global timer bottom 32 bits:    2698388024
00 01: global timer diff:              0000043000
01 01: global timer bottom 32 bits:    2698412424
01 01: global timer diff:              0000043000
00 01: partition timer top 32 bits:    0000000000
01 01: partition timer top 32 bits:    0000000000
00 01: partition timer bottom 32 bits: 0000989626
01 01: partition timer bottom 32 bits: 0000989626
00 01: partition timer diff            0000009996
01 01: partition timer diff            0000009996
00 01: global timer top 32 bits:       0000001659
01 01: global timer top 32 bits:       0000001659
00 01: global timer bottom 32 bits:    2698431024
01 01: global timer bottom 32 bits:    2698455424
00 01: global timer diff:              0000043000
01 01: global timer diff:              0000043000
00 01: partition timer top 32 bits:    0000000000
01 01: partition timer top 32 bits:    0000000000
00 01: partition timer bottom 32 bits: 0000999622
```

# save & restore before next tutorial steps



now commit your T4 solution to your gitlab repo

- save the main.c of partitions 0_1 and 1_1 somewhere
- copy partition 0_2 (`partition_0_2/main.c`) to 0_1 and 1_1
  - or use `gitlab checkout -- partition_?_1/main.c`
- all applications should now print Hello world again

# timing your program(s)



- on each RISC-V
  - execution time → partition timer
  - response time → global timer

  - per function (incl. its subfunctions)
  - # function calls

- on ARM
  - use monotonic clock (`CLOCK_MONOTONIC`) of `sys/timer.h`, see https://man7.org/linux/man-pages/man2/clock_gettime.2.html
  - execution time → per process
  - response time → system wide

# timing your program(s)



- receive a structure with the number of seconds and nano seconds

```
struct timespec {
    time_t   tv_sec;         /* seconds */
    long     tv_nsec;        /* nanoseconds */
};
```

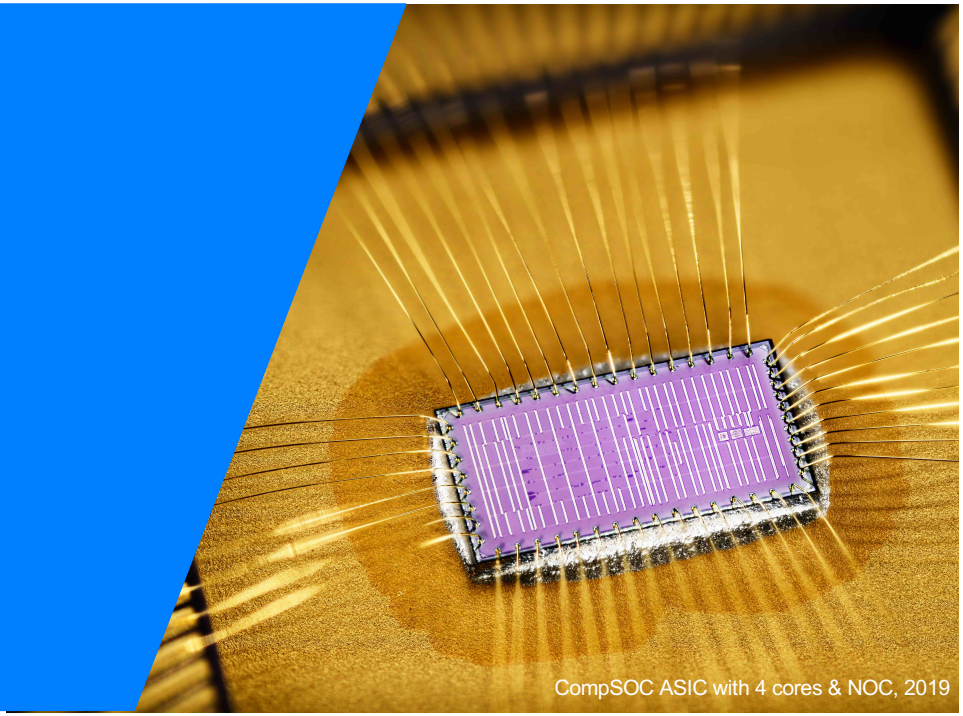
- you can then use a macro like this to calculate the difference between two structures.

```
#define timersub(a,b,result)                                    \
   do {                                                         \
      (result)->tv_sec = (a)->tv_sec - (b)->tv_sec;             \
      (result)->tv_nsec = (a)->tv_nsec - (b)->tv_nsec;          \
      if ((result)->tv_nsec < 0) {                              \
        --(result)->tv_sec;                                     \
        (result)->tv_nsec += 1000000000;                        \
      }                                                         \
    } while (0)
```

- add `-lrt` to the `LDLIBS` and `-D_POSIX_C_SOURCE=199309L` to `CFLAGS` in the ARM Makefile

# *Tutorial*

# *5. RISC-V memories*



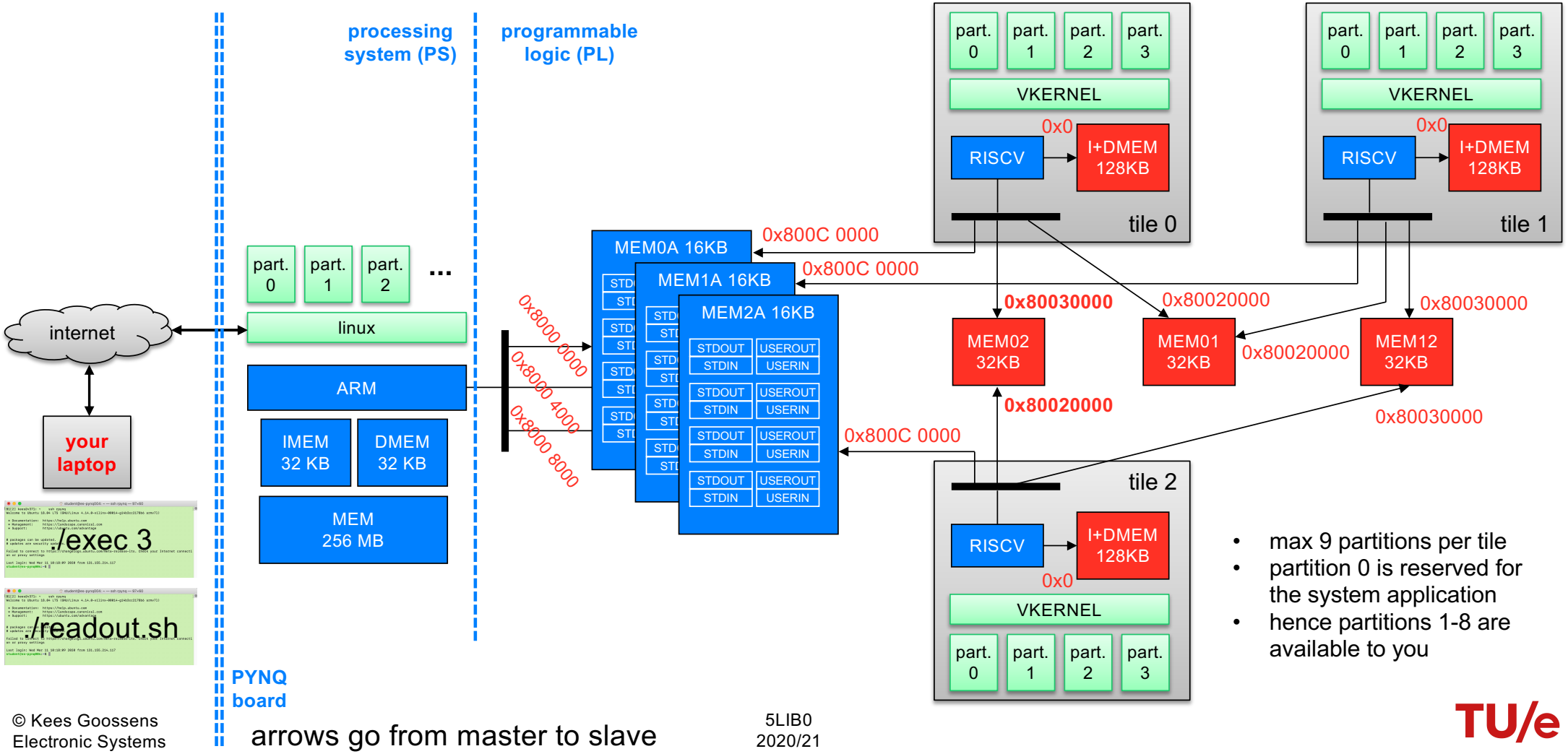
CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# ARM processor, 3 RISC-V cores, up to 9 partitions per tile






- max 9 partitions per tile
- partition 0 is reserved for the system application
- hence partitions 1-8 are available to you

arrows go from master to slave





TU/e

# RISC-V memories

T5



- in partition 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 partition 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:

- the addresses may also be different
- the reason for this is that 3+4 and 30+40 are computed at compile time
- the result 7 can be stored in a half-word instruction
- but the result 70 cannot, and therefore results in a different memory layout

# RISC-V memories

T5



- in partition 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 partition 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?

t05a-*.c

```
00 01: Hello world
00 02: 000009/0795862389: &a=0x00001BE0 a=3
00 02: 000009/0795862389: &b=0x00001BE4 b=4
00 03: 000009/0795917389: &a=0x00001BE0 a=30
00 02: 000009/0795862389: &c=0x00001BE8 c=7
00 03: 000009/0795917389: &b=0x00001BE4 b=40
00 03: 000009/0795917389: &c=0x00001BE8 c=70
01 01: Hello world
01 02: Hello world
01 03: Hello world
02 01: Hello world
02 02: Hello world
02 03: Hello world
```

answer:
- the instruction and data memory on the RISC-V is virtualised
- all partitions therefore see addresses starting from 0x0 (and e.g. up to 0x8000 for 32K memory)
- the values are different because the virtualisation ensures that the same virtual addresses are mapped to different physical addresses

# RISC-V shared memories – using the private region

T5



- to use the shared memories (mem01/02/12)
  you need to tell the compiler what data you want to map into the memory
  in `shared_memories/vep_private_memory.h`
- by default it contains, for each memory

```
typedef volatile struct {
  uint32_t initialized;
  // more fields
} vep_private_mem01_t;
// #define USE_VEP_PRIVATE_MEM01
```

- change it to

```
typedef volatile struct {
  uint32_t d;
} vep_private_mem01_t;
#define USE_VEP_PRIVATE_MEM01 /* uncomment this line */
```

- the private region in shared memories of a VEP are initialised to zero when the VEP is loaded

# virtualised memory – private memory regions



- declare the data layout of the shared memory regions in `shared_memories/vep_private_memory.h`

```
typedef volatile struct {
  // your data in mem01
} vep_private_mem01_t;
#define USE_VEP_PRIVATE_MEM01

typedef volatile struct {
  // your data in mem02
} vep_private_mem02_t;
#define USE_VEP_PRIVATE_MEM02

typedef volatile struct {
  // your data in mem12
} vep_private_mem12_t;
#define USE_VEP_PRIVATE_MEM12
```

```
// IMPORTANT: to use a private memory region you must:
// 1- typedef the struct containing all data to be placed in the region
// 2- uncomment the relevant #define USE_VEP_PRIVATE_MEM* below
//    this will declare & initialise the vep_private_memXY external variables
// 3- declare the memory region in the vep-config.txt file
//    without this, the region will not be declared in the memory map
//
// all fields are set to 0 when the vep is loaded
// (it may therefore be used to have an 'initialized' as first field in the struct,
// this allows other veps can check if this vep is running and initialized or not)
//
// the private data of this vep on tile T in memory M is mapped to
// VEP_PRIVATE_REGION_TILE<T>_MEM<M>_START
```

# RISC-V shared memories

T5



- in partition 2 on tile 0 declare an integer d in the private region of the shared memory mem01

```
typedef volatile struct {
  int32_t d;                    // the variables that you want to map in the shared mem
} vep_private_mem01_t;
#define USE_VEP_PRIVATE_MEM01 /* uncomment this line */
```

- include the private memory declaration in the main.c of partitions that want to use the shared variables

```
#include "vep_private_memory.h"
```

- the system will now automatically declare the variable `vep_private_mem01` which is a pointer to mem01
- you can use like it this in all partitions of VEP 1 that can access mem01

```
xil_printf("variable d is mapped in private memory at address 0x%08X with value %d\n",
           &(vep_private_mem01->d), vep_private_mem01->d);
```

```
00 02: variable d is mapped in private memory at address 0x80020000 with value 0
00 03: variable d is mapped in private memory at address 0x80020000 with value 0
```

- similarly for all shared memories mem01/02/12

# RISC-V shared memories

T5



- in partition 2 on tile 0 declare an integer d in the private region of the shared memory 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 partition 3 on tile 0

```
typedef volatile struct {
  int32_t d;
} vep_private_mem01_t;
#define USE_VEP_PRIVATE_MEM01
```

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

answer:

now commit your T5 solution to your gitlab repo

# RISC-V shared memories

T5



- in partition 2 on tile 0 declare an integer d in the private region of the shared memory 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 partition 3 on tile 0

t5b.c

1st TDM slot

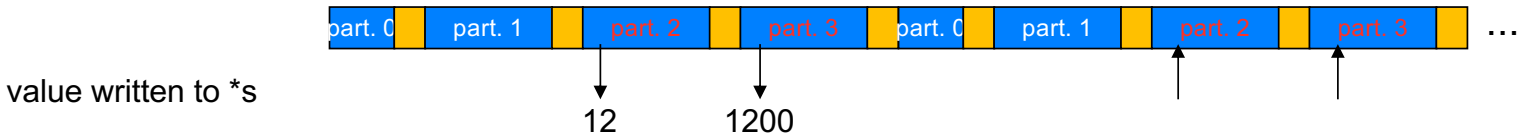
2nd TDM slot

```
00 02: 000011/0768388793:  d=0x80020000 *d=12
00 03: 000011/0768400793:  d=0x80020000 *d=1200
00 02: 000011/0768388793: &a=0x00001C50 a=3
00 03: 000011/0768400793: &a=0x00001C90 a=30
00 02: 000011/0768388793: &b=0x00001C54 b=4
00 03: 000011/0768400793: &b=0x00001C94 b=40
00 02: 000011/0768388793: &c=0x00001C58 c=7
00 03: 000011/0768400793: &c=0x00001C98 c=70
00 02: 000011/0768603752:  d=0x80020000 *d=1200
00 03: 000011/0768615752:  d=0x80020000 *d=1200
00 02: 000011/0768603752: &a=0x00001C50 a=3
00 03: 000011/0768615752: &a=0x00001C90 a=30
00 02: 000011/0768603752: &b=0x00001C54 b=4
00 03: 000011/0768615752: &b=0x00001C94 b=40
00 02: 000011/0768603752: &c=0x00001C58 c=7
00 03: 000011/0768615752: &c=0x00001C98 c=70
```

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

answer:

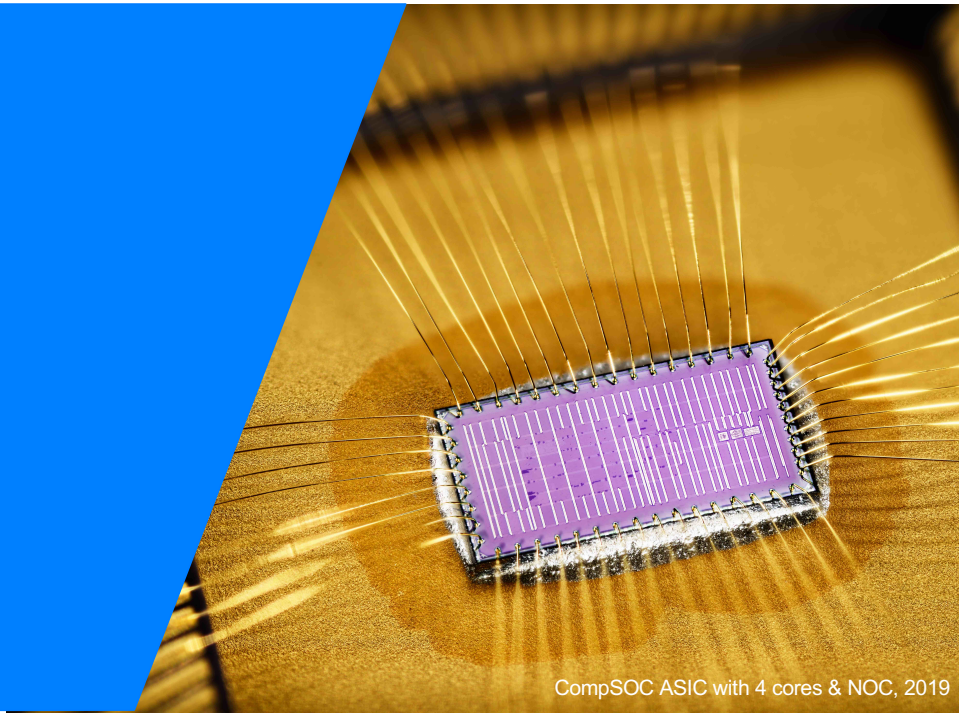
- specified in vep-config.txt, part. 0_2 is scheduled to run before part. 0_3 and puts the value 12 in address 0x0 (mem01)
- later partition_0_3 runs and overwrites the value 12 with the value 1200
- both partitions wake up in their second slot and see the same value 1200
- note that d is automatically volatile




TU/e

# *Tutorial*

# *6. Time-based synchronisation*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# RISC-V shared memories

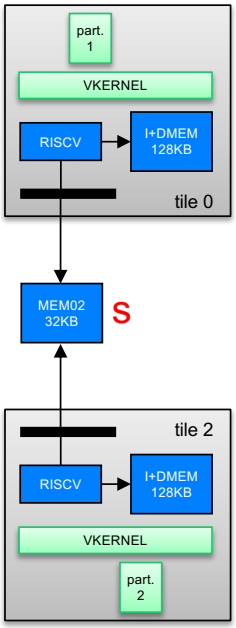
T6



- set the TDM slots for all partitions to 20000 cycles in vep-config.txt
- let's see what happens when two partitions on different tiles access shared memory
- in part. 1 on tile 0 declare an integer `s` initialised in the private region of mem02

t06a-01.c

```
volatile const uint64_t * const g_timer = (uint64_t*)TILE0_GLOBAL_TIMER;
uint64_t t = *g_timer;
xil_printf("%06u/%010u:  &s=0x%08X s=%d\n", ... (uint32_t) &vep_private_mem02->s, vep_private_mem02->s);
vep_private_mem02->s = 01;
t = *g_timer;
xil_printf("%06u/%010u:  &s=0x%08X s=%d\n", ... (uint32_t) &vep_private_mem02->s, vep_private_mem02->s);
asm("wfi");
t = *g_timer;
xil_printf("%06u/%010u:  &s=0x%08X s=%d\n", ... (uint32_t) &vep_private_mem02->s, vep_private_mem02->s);
```




- copy partition 1 on tile 0 to partition 2 on tile 2

t06a-22.c

– replace `vep_private_mem02->s = 01;` by `vep_private_mem02->s = 22;`

- run the system
- why are the addresses of `s` different?
- why are the values for `s` the same and/or different?

answer:





TU/e

# RISC-V shared memories

T6

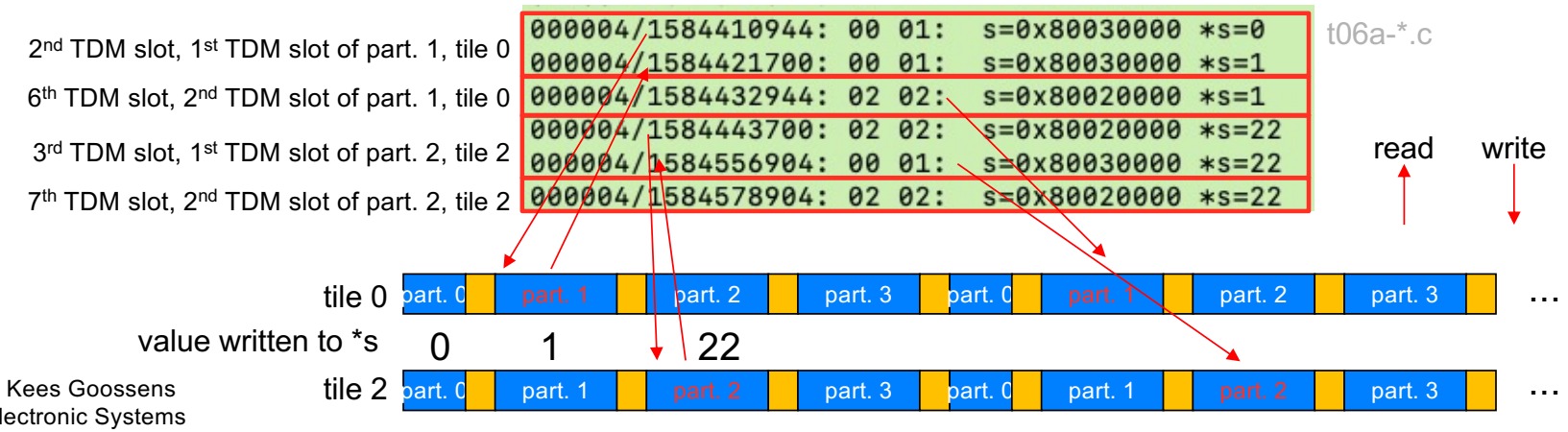


- run the system
- why are the addresses for `s` different?
- why are the values the same and/or different?

```
#define TILE0_MEM01_START 0x00020000 /* mem01 in core 0's mem map */
#define TILE0_MEM01_START 0x00030000 /* mem02 in core 0's mem map */
#define TILE1_MEM01_START 0x00020000 /* mem01 in core 1's mem map */
#define TILE1_MEM12_START 0x00030000 /* mem12 in core 1's mem map */
#define TILE2_MEM02_START 0x00020000 /* mem02 in core 2's mem map */
#define TILE2_MEM12_START 0x00030000 /* mem12 in core 2's mem map */
```

answer:

- according to platform.h the same physical memory location 0x0 in mem02
  has address 0x80030000 on tile 0
  and address 0x80020000 on tile 1
- according to vep-config.txt all tiles have the same schedule
- we can analyse the sequence of reads & writes on the different cores, see below






TU/e

# time vs. time synchronisation



- although it is possible to synchronise between partitions based on time (as we've seen above)
- it is better to use data synchronisation

- optional task:
  - how would you synchronise the start of all partitions on all tiles based on time?
  - the worst-case offset between slot 0 on two different cores is 1 TDM table
  - hints:
    - you need to wait for a common time in the future
    - you cannot compare two 64-bit numbers; split it up in two comparisons

now commit your T6 solution to your gitlab repo

# time-based synchronisation



- set an absolute point in time in the future
- worst-case difference is a TDM table revolution

t06b.c

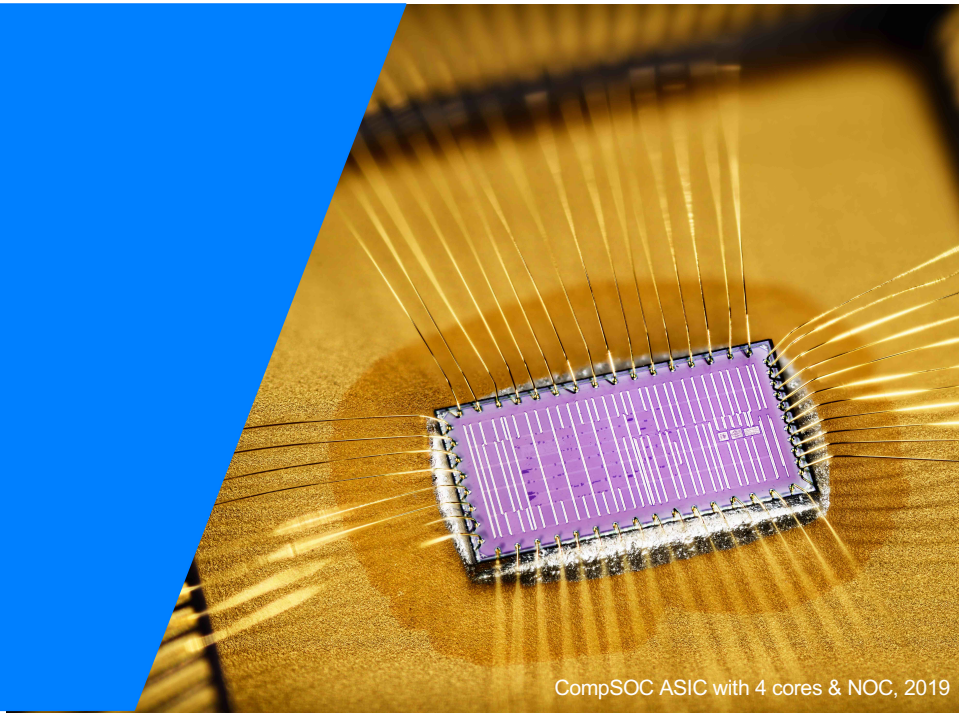
```
volatile const uint64_t * const g_timer = (uint64_t*)TILE0_GLOBAL_TIMER;
// round up to second-nearest 1e8 cycles
uint64_t w = ((t / (uint64_t) 1e8) +2) * (uint64_t) 1e8;
xil_printf("%06u/%010u: wait until %06u/%010u\n",
    (uint32_t)(t>>32), (uint32_t)t, (uint32_t)(w>>32), (uint32_t)w);
// wait until deadline
do { t = *g_timer; }
while ((t >> 32) <= (w >> 32) && (uint32_t) t <= (uint32_t) w);
asm("wfi"); // keep if you want to start at slot boundaries
/***** synchronised start from here *****/
t = *g_timer;
xil_printf("%06u/%010u: hello world\n", (uint32_t)(t>>32), (uint32_t)t);
```

difference 39,000 cycles after immediately resetting the board
(hence all the timer is reset too)

```
00 03: 000000/0854428047: wait until 000000/1000000000
00 01: 000000/0854447047: wait until 000000/1000000000
00 02: 000000/0854459047: wait until 000000/1000000000
01 01: 000000/0857528047: wait until 000000/1000000000
01 02: 000000/0857540047: wait until 000000/1000000000
01 03: 000000/0857552047: wait until 000000/1000000000
02 03: 000000/0860628047: wait until 000000/1000000000
02 01: 000000/0860702047: wait until 000000/1000000000
02 02: 000000/0860714047: wait until 000000/1000000000
02 02: 000000/1000034009: hello world
01 02: 000000/1000042009: hello world
02 03: 000000/1000046009: hello world
00 01: 000000/1000045009: hello world
01 03: 000000/1000054009: hello world
00 02: 000000/1000057009: hello world
02 01: 000000/1000065009: hello world
00 03: 000000/1000069009: hello world
01 01: 000000/1000073009: hello world
```

# *Tutorial*

# *Recapitulation*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# recap

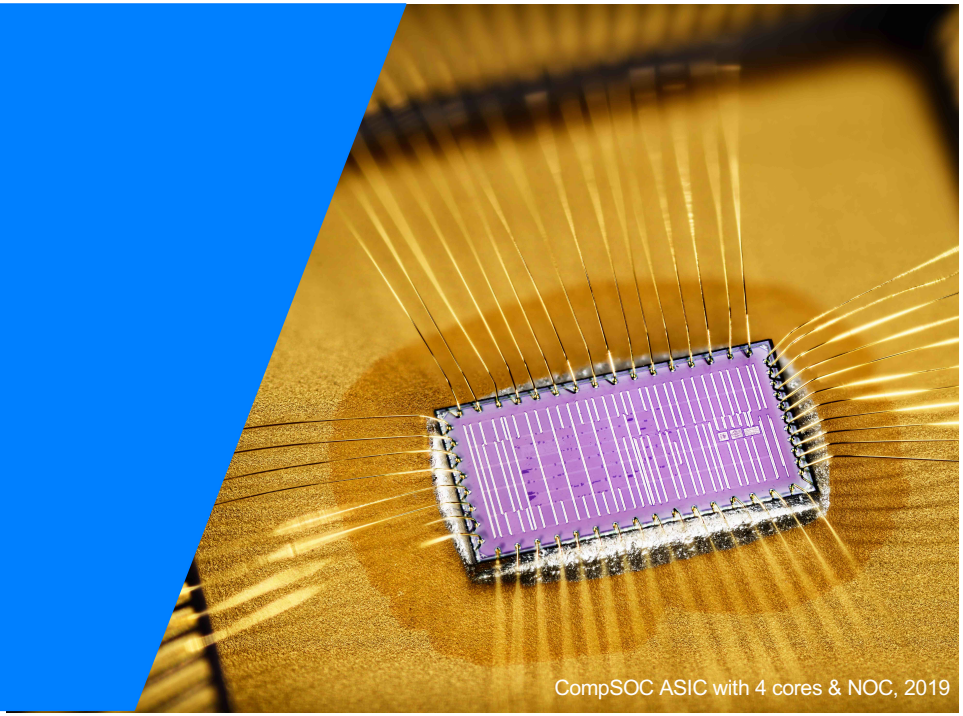


- space:
  - addresses of instruction & data memories mem*i* are virtualised
  - addresses of shared memories mem*ij* are virtualised
    - and the same memory location may have different physical addresses associated with it
  - vep-config.txt specifies the stack & memory sizes of each partition on each core

- time
  - vep-config.txt specifies the start & duration of time slots of partitions on each core
  - tip: allocate unused slots to the system partition 0, rather than deleting them
    that helps to keep the slot tables aligned

# *Tutorial*

# *7. Data-based synchronisation*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# spin lock

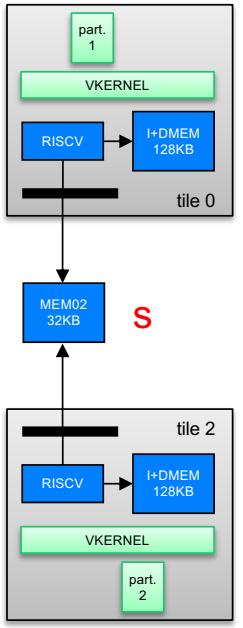
T7



- reset the platform to its initial state (by now you know how to do this?)
- we want to start partition 1 on tile 0 only when partition 2 on tile 2 has given it a signal
- you can do this by making partition 1 wait for the value 22 of the shared variable that partition 2 writes
- in partition 1 on tile 0 declare an integer `s` in mem02
- this is called a spin lock, because the processors "spins" on the lock
- the shared variable s has to be declared volatile because it may be asynchronously modified by another core, and the program must re-read it on every use (e.g. in the while loop) [this is the same as for the timers]

t07a-*.c

```
volatile const uint64_t * const g_timer = (uint64_t*)TILE0_GLOBAL_TIMER;
uint64_t t = *g_timer;
xil_printf("%06u/%010u:  &s=0x%08X s=%d; waiting for flag\n", ...
           &vep_private_mem02->s, vep_private_mem02->s);
while (*vep_private_mem02->s != 22) /* do nothing */ ;
t = *g_timer;
xil_printf("%06u/%010u:  &s=0x%08X s=%d; flag unlocked\n", ...
           &vep_private_mem02->s, vep_private_mem02->s);
```




- run the system

# spin lock



- your output should be similar to this

```
01 01: Hello world
02 01: Hello world
00 02: Hello world
01 02: Hello world
02 03: Hello world
00 03: Hello world
01 03: Hello world
00 01: 000011/4171191871:   s=0x80030000 *s=0
02 02: 000011/4171203872:   s=0x80020000 *s=0; unlock flag
00 01: 000011/4171235717:   s=0x80030000 *s=1; waiting for flag
02 02: 000011/4171250400:   s=0x80020000 *s=22; flag unlocked
00 01: 000011/4171283183:   s=0x80030000 *s=22; flag unlocked
```

update screenshot

- would all embedded systems behave like this or
  is there a special property of the (memory in the) CompSOC platform that make this work?

# spin lock – initialising memories



- your output should be similar to this

```
00 01: 000011/4171191871:   s=0x80030000 *s=0
02 02: 000011/4171203872:   s=0x80020000 *s=0; unlock flag
00 01: 000011/4171235717:   s=0x80030000 *s=1; waiting for flag
00 01: 000011/4171250400:   s=0x80020000 *s=22; flag unlocked
00 01: 000011/4171283183:   s=0x80030000 *s=22; flag unlocked
```

update screenshot

- would all embedded systems behave like this or
  is there a special property of the (memory in the) CompSOC platform that make this work?

- in CompSOC (and not in most other systems)
  the shared memories are initialised to zero before you program starts
  - variables in instruction/data memories are initialised according to the rules of C

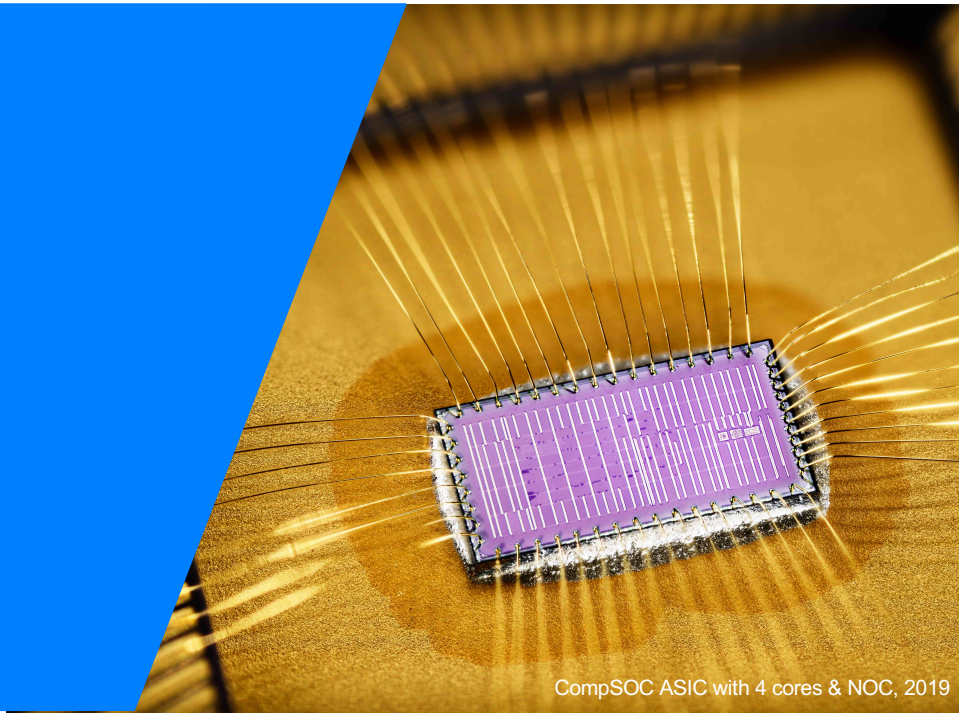
```
Clearing memory: 0x80020000 - 0x80024000.
       0
Clearing memory: 0x80030000 - 0x80034000.
       0
Clearing memory: 0x80020000 - 0x80024000.
       0
Clearing memory: 0x80030000 - 0x80034000.
       0
```

- this makes booting the system and inter-process communication much easier
  - how would you implement the spin lock if the memory contained random data?

now commit your T7 solution to your gitlab repo

# *Tutorial*

# *8. 1-place buffer*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

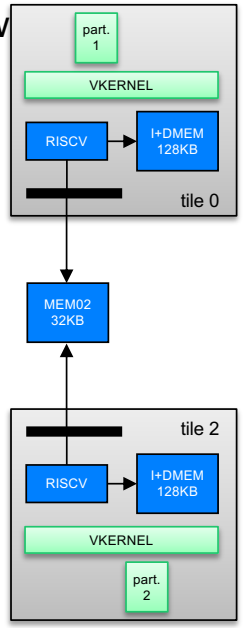
TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# 1-place buffer

T8



- a 1-place buffer is a shared memory region that is
  alternately written by a producer (partition 2) and read by a consumer (partition 1)
  - part. 1 writes
  - part. 2 reads
  - part. 1 writes
  - part. 2 reads
  - etc.
- the consumer cannot see from the data if it is ready to be read, e.g. the producer may send the same data tw
- the buffer can contain any data, and can therefore be larger than 1 word
  - how to ensure atomicity, i.e. only reading the buffer when it has been entirely written?




- implement a 1-place buffer that safely transfers the following data structure from partition 1 to partittion 2

t08a-*.c

```
typedef struct {
  int32_t data[5];      // the data
  uint64_t timestamp;   // indicates when the data was produced
} token_t;
```

- the i-th token contains integers 10*i, 10*i+1, 10*i+2, 10*i+3, 10*i+4
- the consumer must give an error message if the data is not consistent
  (i.e. doesn't contain the expected numbers)

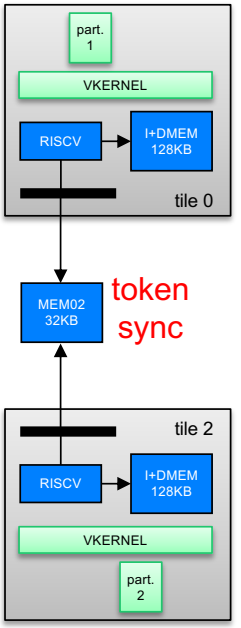
# 1-place buffer, hints

T8



- you need to declare a single token that both partitions can access
- you need to synchronise access to the token (e.g. who is the “owner” of the token)
- wait (spin lock) until you’re owner before reading or writing the token
- recall that shared memory is initialised with zeros, giving you a well-defined initial state (the producer should be the first owner)
- since you need to declare multiple variables in the same shared memory, you need to take care of your memory map

```
#define PRODUCER_IS_OWNER ...
#define CONSUMER_IS_OWNER ...
typedef struct {
  int32_t data[5];     // the data
  uint64_t timestamp; // indicates when the data was produced
} token_t;
typedef volatile struct {
  token_t token = ...;
  uint32_t sync = ...;
} vep_private_mem??_t;
#define USE_VEP_PRIVATE_MEM??
```



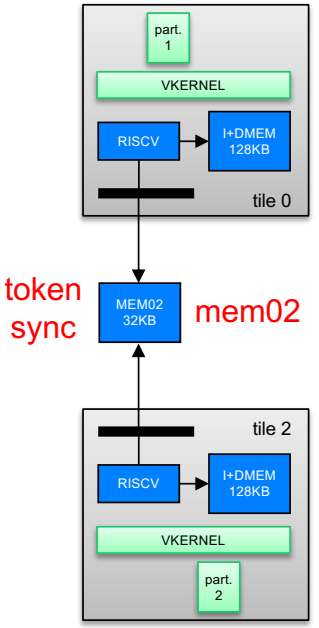
# 1-place buffer, step 1

T8



- first define the token type, and declare the token
- then declare the synchronisation variable
- print out the addresses and values (time stamp and data) to check that
  - they do not use the same locations
  - they are the same on both processors

```
00 01: 000005/3767533581:  token=0x80030000 ts=0000000000, data=0,0,0,0,0,0
00 01: 000005/3767533581:  sync=0x80030020 *sync=0
02 02: 000005/3774058581:  token=0x80020000 ts=0000000000, data=0,0,0,0,0,0
02 02: 000005/3774058581:  sync=0x80020020 *sync=0
```



# 1-place buffer, step 2

T8

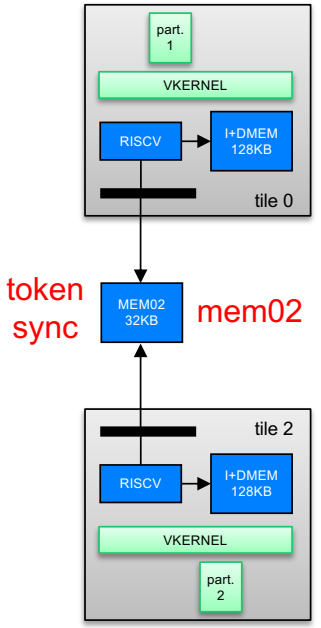


- producer (part. 2)
  - fills the first token (data + time stamp)
  - flips the sync variable
- consumer (part. 1)
  - spin locks on sync
- modify the printfs to this format:

t08b-*.c

```
02 02: 000006/3976977356: wrote token ts=3976977323, data=0000,0001,0002,0003,0004
00 01: 000006/3976988240: read token  ts=3976977323, data=0000,0001,0002,0003,0004, dts=0000010901
```

- note that the order of printing is not always time order!
- use `sort-on-time.sh` to ensure this
- dts is duration from the time the token was written
  to the time the token was read (*g_timer - token.timestamp)



# 1-place buffer, step 3

T8



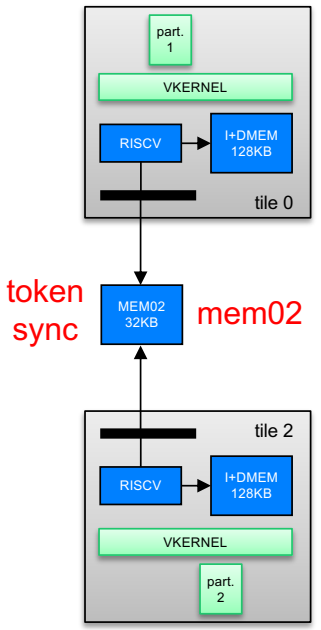
- add a loop around the token production & consumption
- implement the check at the consumer that data is right

```
000007/3209816079: 02 02: wrote token ts=3209816045, data=0000,0001,0002,0003,0004
000007/3209826946: 00 01: read token  ts=3209816045, data=0000,0001,0002,0003,0004, dts=0000010901
000007/3209945019: 02 02: wrote token ts=3209944985, data=0010,0011,0012,0013,0014
000007/3209955943: 00 01: read token  ts=3209944985, data=0010,0011,0012,0013,0014, dts=0000010958
000007/3210074016: 02 02: wrote token ts=3210073982, data=0020,0021,0022,0023,0024
000007/3210084943: 00 01: read token  ts=3210073982, data=0020,0021,0022,0023,0024, dts=0000010961
000007/3210203016: 02 02: wrote token ts=3210202982, data=0030,0031,0032,0033,0034
000007/3210213943: 00 01: read token  ts=3210202982, data=0030,0031,0032,0033,0034, dts=0000010961
000007/3210332016: 02 02: wrote token ts=3210331982, data=0040,0041,0042,0043,0044
000007/3210342943: 00 01: read token  ts=3210331982, data=0040,0041,0042,0043,0044, dts=0000010961
```

t08c-*.c

- explain the value of dts
- how could you reduce it?

answer:




now commit your T8 solution to your gitlab repo

# 1-place buffer, step 3

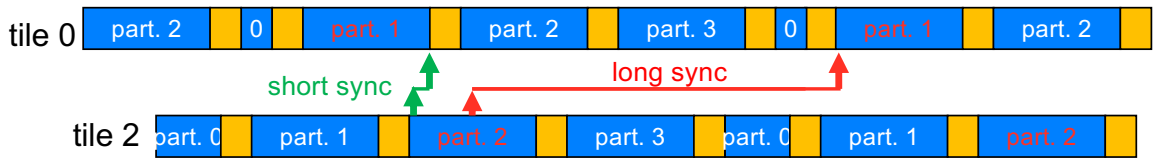


```
004255/4169572184: 02 02: wrote token ts=4169572142, data=0000,0001,0002,0003,0004
004255/4169580935: 00 01: read token  ts=4169572142, data=0000,0001,0002,0003,0004, dts=0000008793
```

- explain the value of dts

answer:
- you may see multiple values for dts in the same run or between runs
  - very short – when the time slots of the partitions are aligned
  - long – when the time slots are misaligned by one or multiple intervening slots

```
grep dts output.txt | sed 's/.*dts=//g' | sort -u
0000000036
0000000037
0000000038
...
0000034364
0000036280
0000041710
0000041798
0000041817
0000042937
```

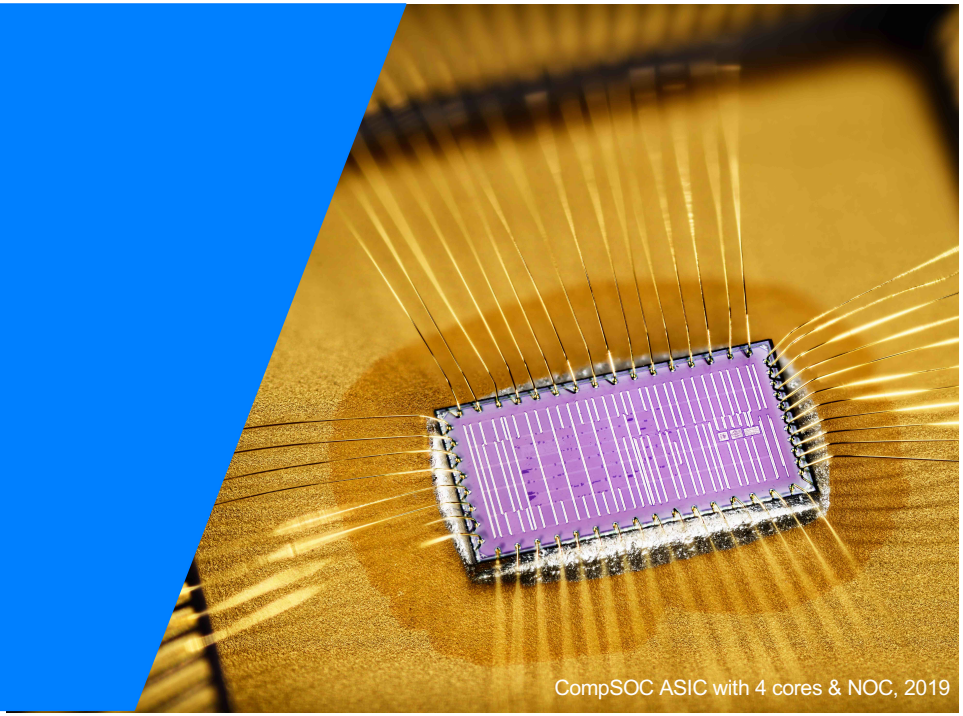



```
Process tile: 0
Set schedule : 0 1 10000
Set schedule : 1 2 10000
Set schedule : 2 3 10000
Set schedule : 3 0 5000
Set schedule : 0 1 10000
Set schedule : 1 2 10011
Set schedule : 2 3 10000
Set schedule : 3 0 5000
```

if we make the TDM slot lengths relative primes then they will move relative to each other

# *Tutorial*

# *9. ARM – RISC-V communication*



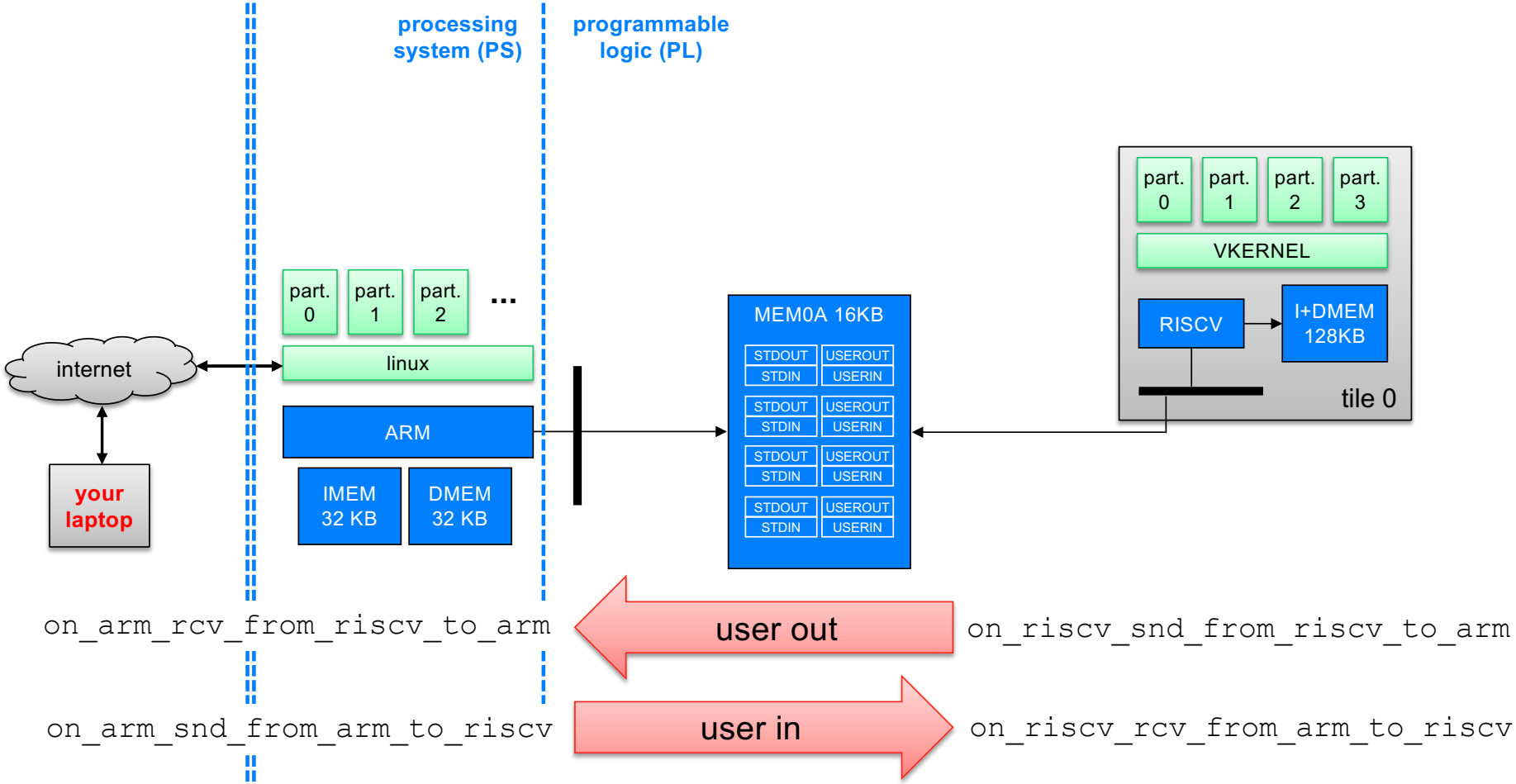
CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

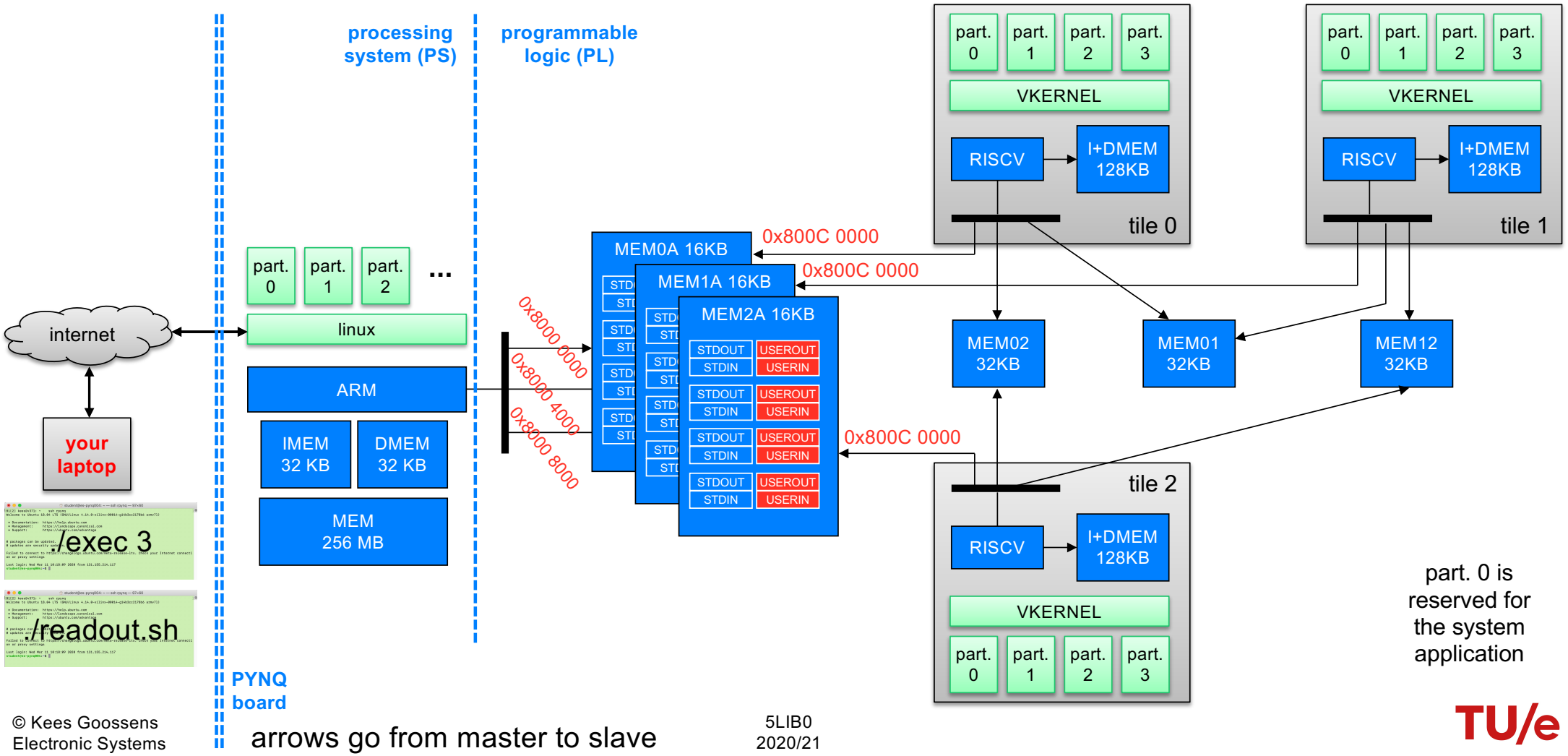
# every partition also has independent pair of user-in & user-out FIFOs





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










TU/e

# ARM – RISC-V communication

T9



- every partition on every RISC-V core has a user-in & user-out FIFO to the ARM
- RISC-V program (`partition_2_2/echo-riscv.c`) uses the FIFOs using the following API

```
#include <libuserchannel-riscv.h>
/* includes the following declarations:
extern uint32_t on_riscv_poll_from_arm_to_riscv (void);       // is there data from the arm?
extern uint32_t on_riscv_poll_from_riscv_to_arm (void);       // is there space to send to the arm?
extern unsigned char on_riscv_rcv_from_arm_to_riscv (void);   // get char from the ARM
extern void on_riscv_snd_from_riscv_to_arm (unsigned char c); // send char to the ARM
*/

int main (void)
{
  volatile const uint64_t * const g_timer = (uint64_t*)TILE0_GLOBAL_TIMER;
  while(1) {
    // read char from user FIFO from ARM & echo on user FIFO to ARM
    unsigned char c = on_riscv_rcv_from_arm_to_riscv();
    on_riscv_snd_from_riscv_to_arm(c);
    uint64_t t = *g_timer;
    xil_printf("%04u/%010u: echo back \'%c\'\n", (uint32_t)(t>>32), (uint32_t)t, c);
  }
  return 0;
}
```

# ARM – RISC-V communication

T9

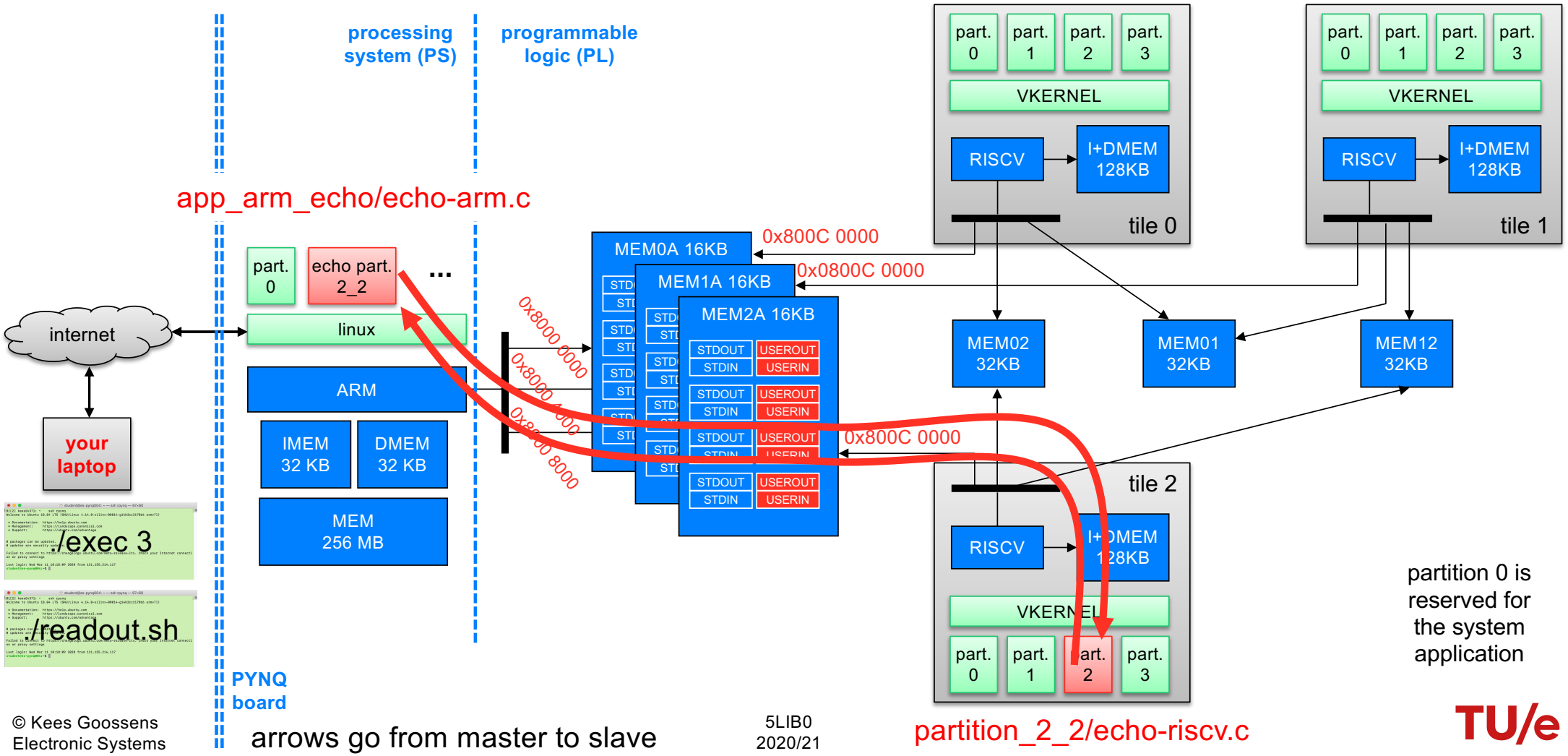


- a program that runs on the ARM must use the following API to communicate with RISC-V partitions
- RISC-V program (`partition_2_2/echo-riscv.c`) uses the FIFOs using the following API

```
#include <libuserchannel-arm.h>
/* includes the following declarations:
extern unsigned int on_arm_initialise (int tileid, int pid);   // initialises both user-in & user-out FIFOs
extern void on_arm_cleanup (int tileid, int pid);
extern int on_arm_poll_from_riscv_to_arm (int tileid, int pid);   // is there data from the riscv?
extern int on_arm_poll_from_arm_to_riscv (int tileid, int pid);   // is there space to send to the riscv?
extern unsigned char on_arm_rcv_from_riscv_to_arm (int tileid, int pid);         // get char from riscv
extern void on_arm_snd_from_arm_to_riscv (int tileid, int pid, unsigned char c); // send char to riscv
extern unsigned long int on_arm_bytes_sent_from_arm_to_riscv (int tileid, int pid);
extern unsigned long int on_arm_bytes_received_from_riscv_to_arm (int tileid, int pid);
*/
int main (void)
{
  int tileid = 2, pid = 2, c;
  on_arm_initialise (tileid,pid);
  while ((c = getchar()) != EOF) {
    on_arm_snd_from_arm_to_riscv (tileid, pid, (unsigned char) c);
    putchar (on_arm_rcv_from_riscv_to_arm (tileid, pid));
  }
  on_arm_cleanup (tileid,pid);
}
```

# example echo application on ARM










TU/e

# example echo application on ARM



- reset the repository to its initial state
- copy the
  - `tutorial/echo/app_arm_echo`
  - `tutorial/echo/partition_2_2`
- to the main directory
- have a look at the .c and Makefiles
  - notice that the Makefiles of the ARM & RISC-V partitions include the userchannel libs
- open three terminals on the PYNQ board
- start the programs in the following order:
  - in the main directory, to `./run.sh` (and later `./stop.sh`) the RISC-V applications
  - in the main directory, to run `./readout.sh`, to get the stdin/out from the PYNQ boards
  - in the `part._arm_echo` directory, to run `sudo ./echo-arm 2 2`
    to type text and see it echoed back (or try: `sudo ./echo-arm 2 2 < echo-arm.c` - what happens?)
- see the following slide

a terminal for each ARM application run with `sudo`

<u>start second</u>

terminal that shows stdout of all RISC-V partitions

<u>start first</u>

terminal to start/stop the real-time RISC-V platform

<u>start last</u>

task:

- run the echo part. & check the output
  - start readout.sh first
  - then start echo-arm
  - then start run.sh
- change the TDM table on tile 3 and give partition 3 4,000,000 cycles
- re-run; explain what happens

after starting,
you type in this terminal
and your input should be echoed back

T9

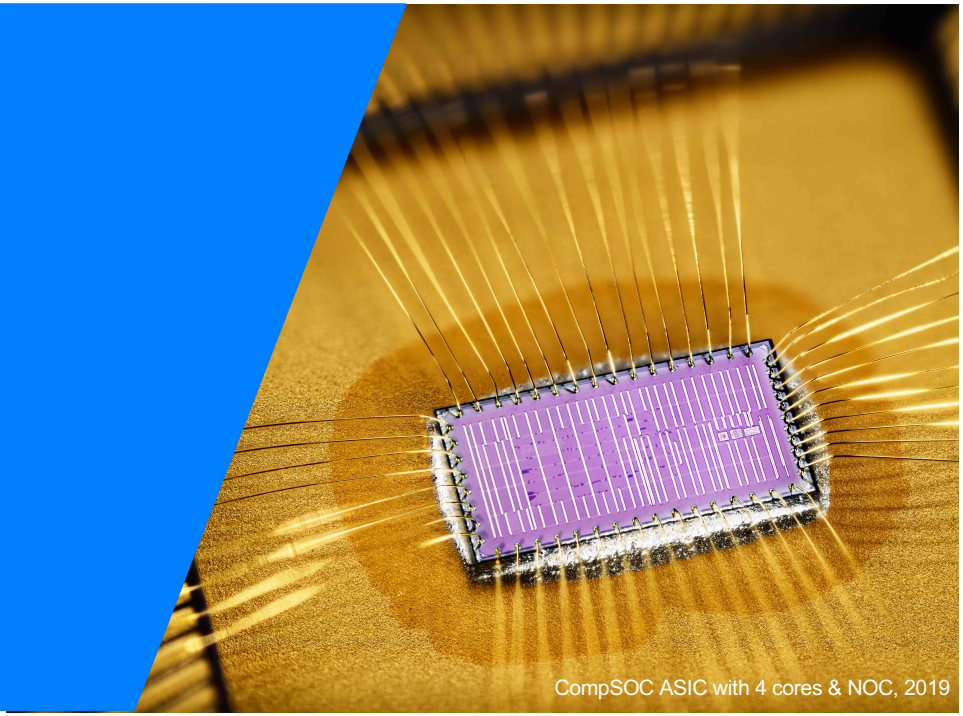
# important hints



- the order of starting readout.sh, fractal-arm, exec.sh is important!
  1. readout.sh initialises the FIFOs between ARM & RISC-Vs, and must always be started first
  2. exec.sh or run.sh starts using the FIFOs from the RISC-V side
  3. echo-arm starts using the FIFOs from the ARM side

- in 2 & 3 ARM & RISC-V cores will block on empty or full FIFOs and can be started in either order

# *Detailed explanation of ARM – RISC-V communication, (re)running, readout, etc.*



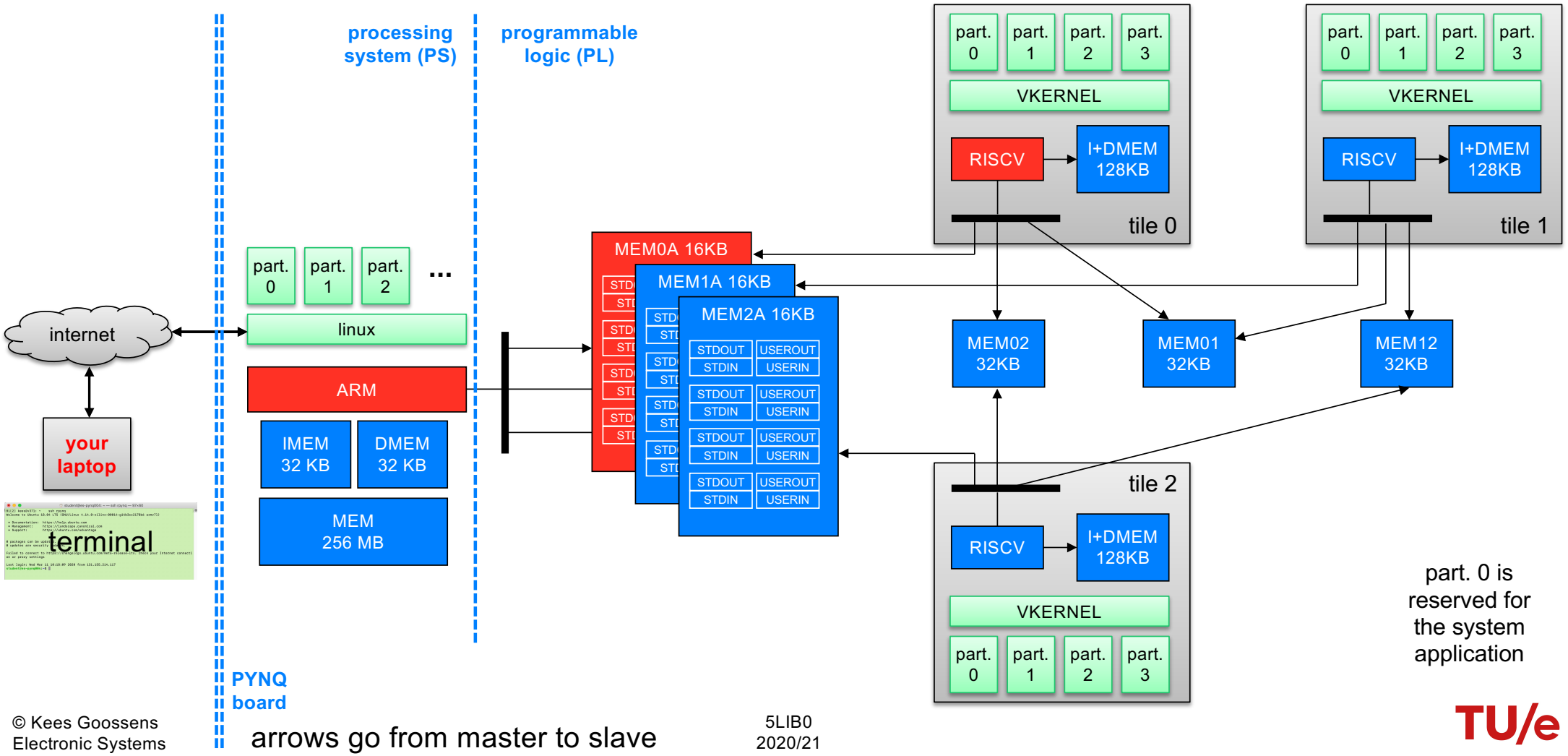
CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# platform overview







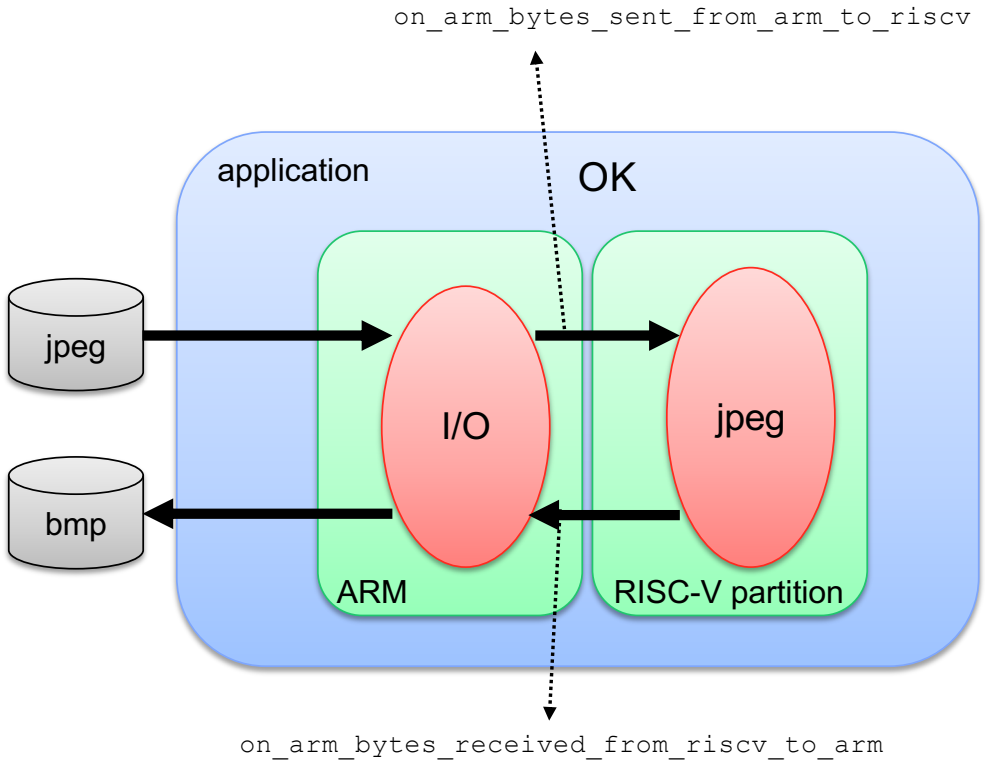


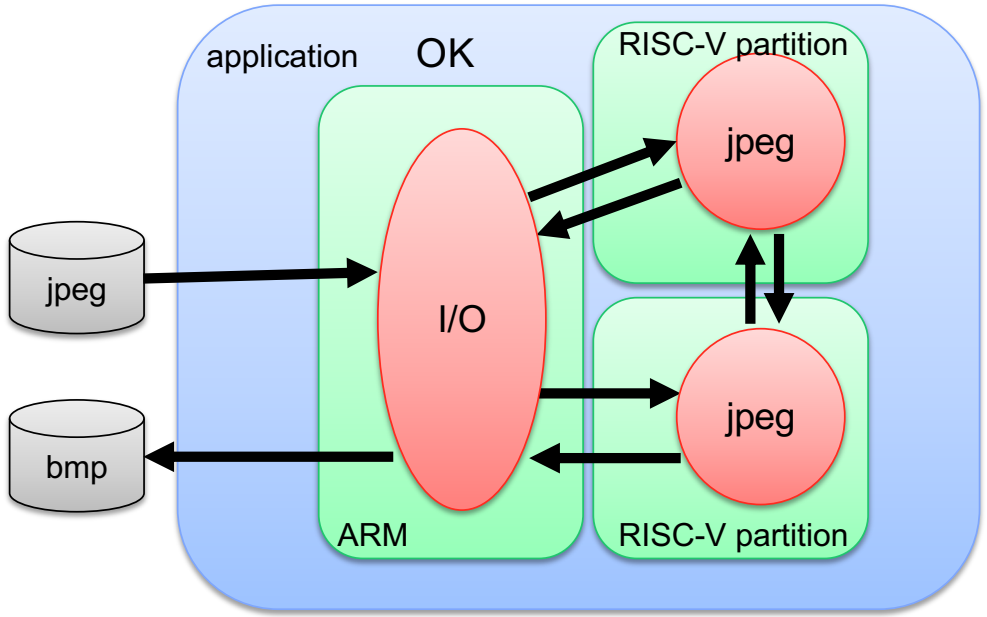

TU/e

# ARM + RISC-V FIFOs



- a single RISC-V partition can only communicate to a single ARM program
- one ARM program can communicate to multiple RISC-V partitions
- hint: use on_arm_poll_* functions in your single ARM program





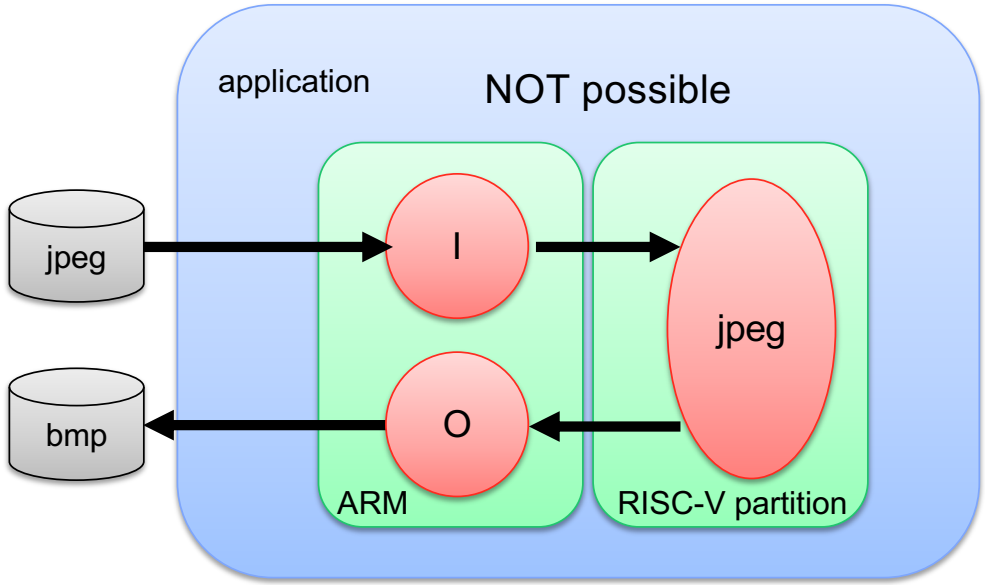



TU/e

# ARM + RISC-V FIFOs



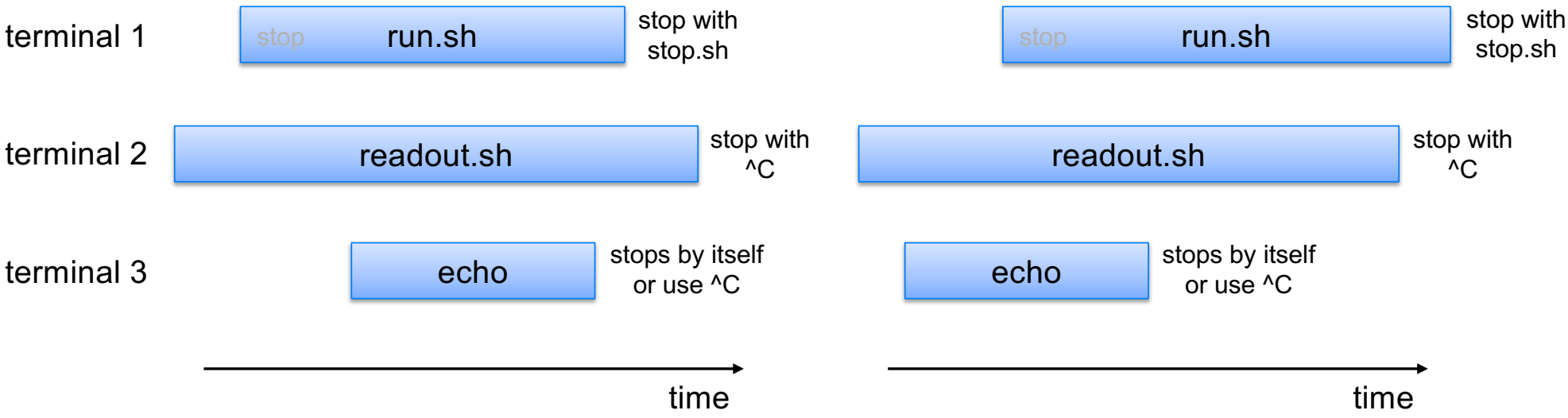
- a single RISC-V partition can only communicate to a single ARM program
- one ARM program can communicate to multiple RISC-V partitions
- hint: use on_arm_poll_* functions in your single ARM program



# echo application



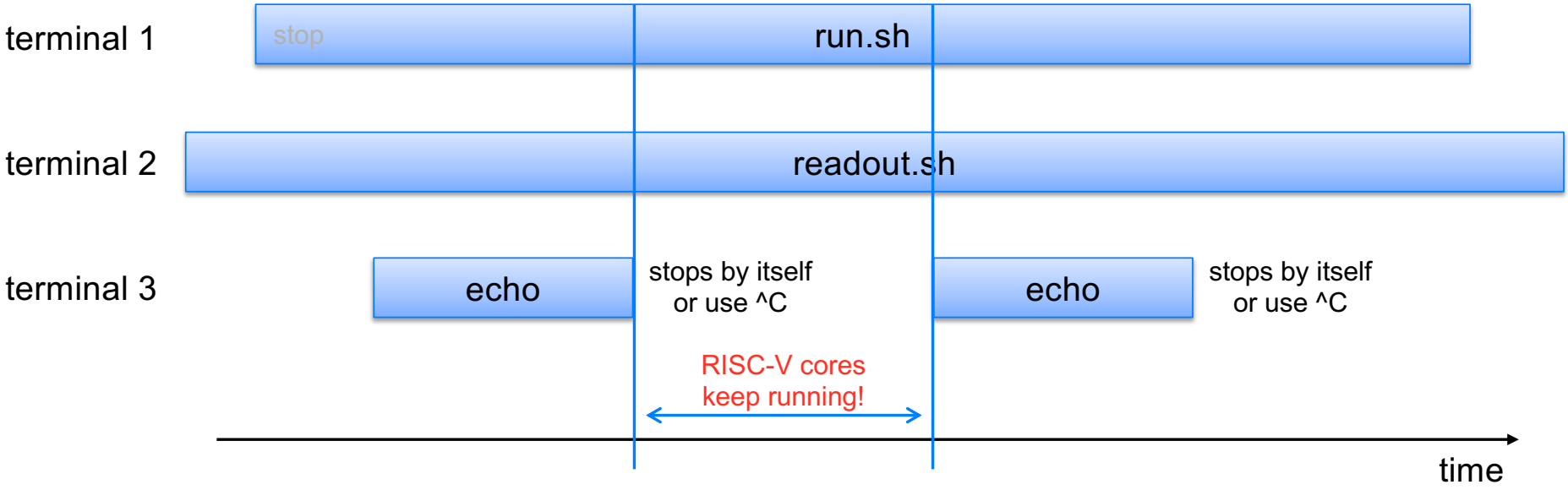
- we consider the echo application in the example, but other applications are the same
- readout.sh should always start first as it initializes the FIFOs between ARM & RISC-V cores
- (re)run.sh and the ARM application can be started in any order, and then stopped in any order



# re-running



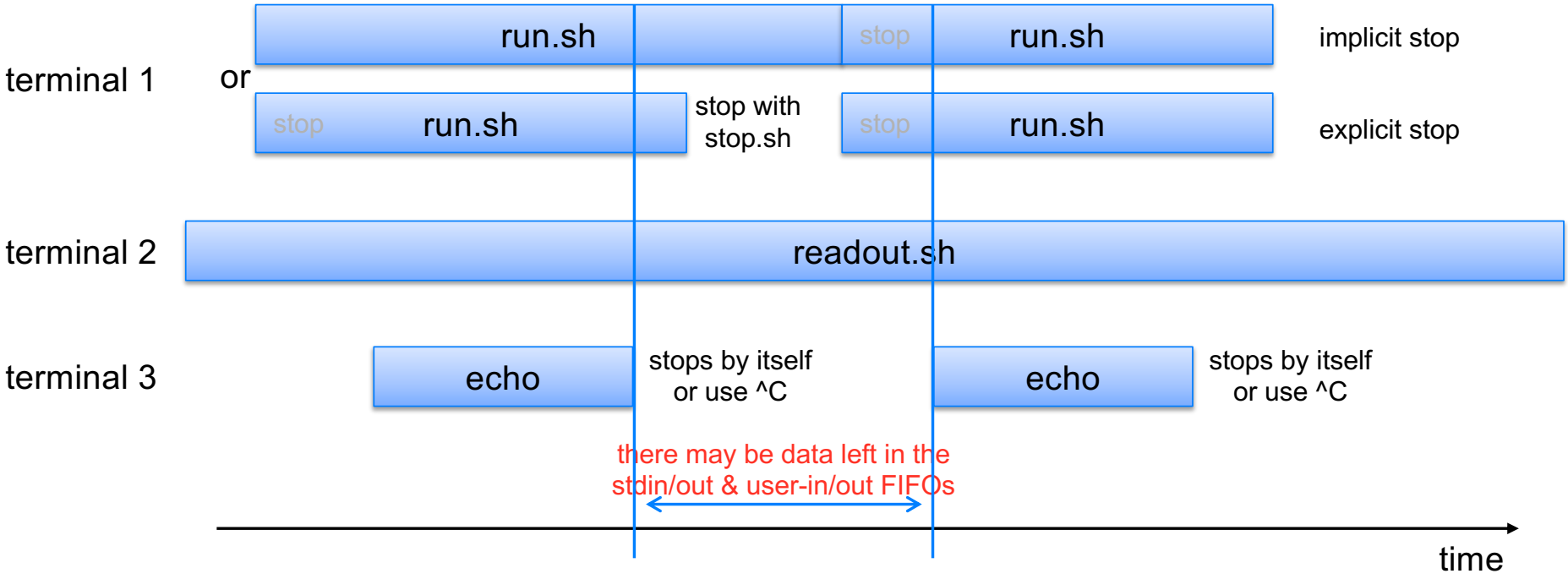
- you can re-run echo
- it will reconnect to the RISC-V cores using the existing ARM – RISC-V FIFOs
- but: there may be some (inconsistent) data left in the user-in/out FIFOs
  - programs on the RISC-V cores may have produced data in the mean time that your echo program should have consumed
  - depending on the RISC-V programs they may have non consumed some data in the user-in FIFOs
- restarting echo may give you unexpected results because of this (“old” xil_printf output, etc.)



# rerunning



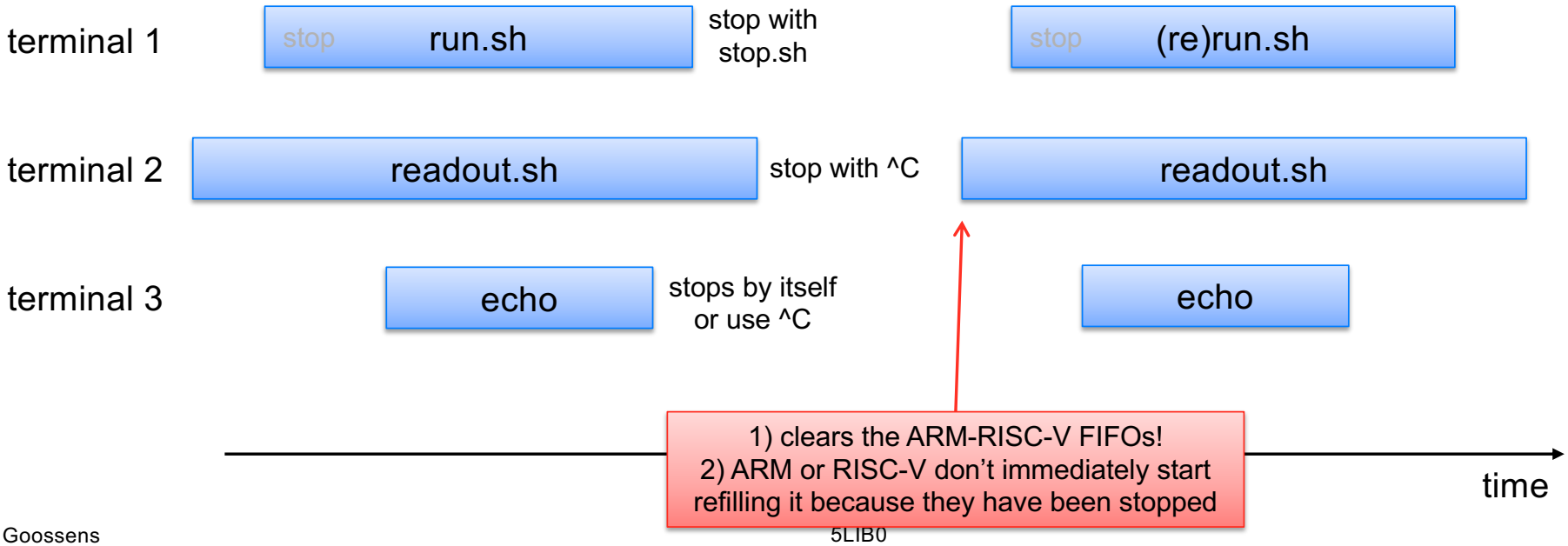
- if readout.sh keeps running, all this can even happen when you stop the RISC-V cores in between
- because readout.sh keeps the stdin/out and user-in/out FIFOs alive
  - old unconsumed data remains there
  - and RISC-V and ARM programs reconnect to them



# safely rerunning, or restarting if you get junk in/output

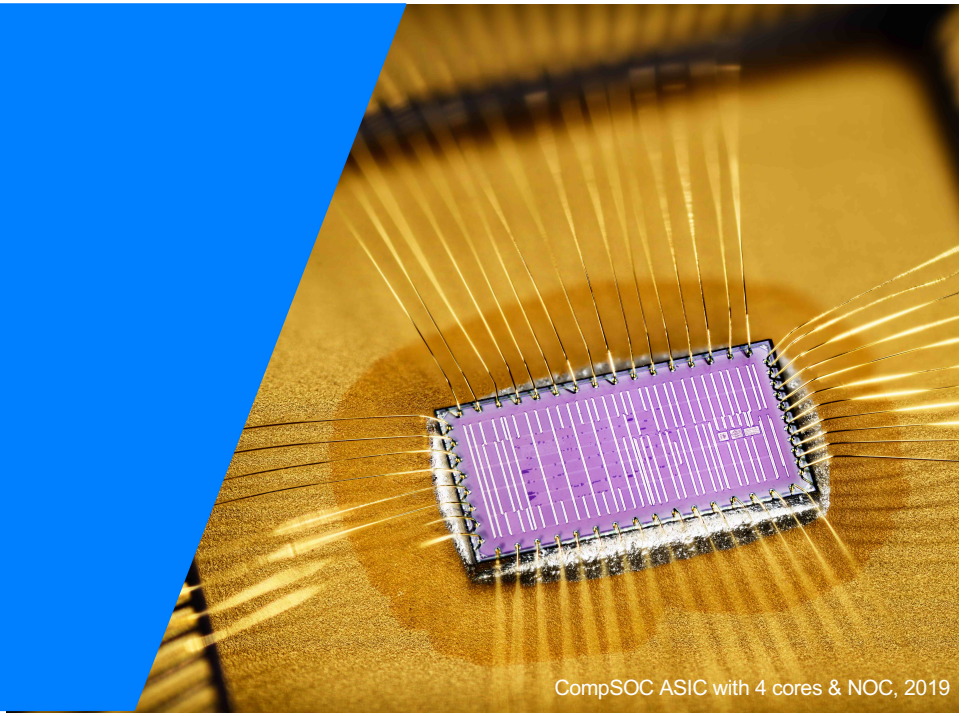


1. ^C your ARM program
2. ./stop.sh # stop the RISC-V programs
3. ^C your ./readout.sh
4. ./readout.sh # this re-initializes the FIFOs and removes any data left in the stdin/out, user-in/out FIFOs
5. ./run.sh (or ./rerun.sh) # start the RISC-V programs
6. restart your ARM program



# *Tutorial*

# *Fractal*



CompSOC ASIC with 4 cores & NOC, 2019

**Kees Goossens <k.g.w.goossens@tue.nl>**
**Electronic Systems Group**
**Electrical Engineering Faculty**

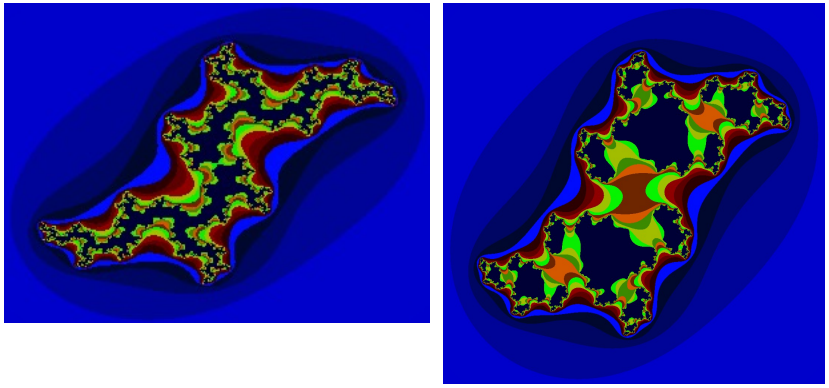
TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# fractal

[tutorial/fractal-arm/fractal.c]

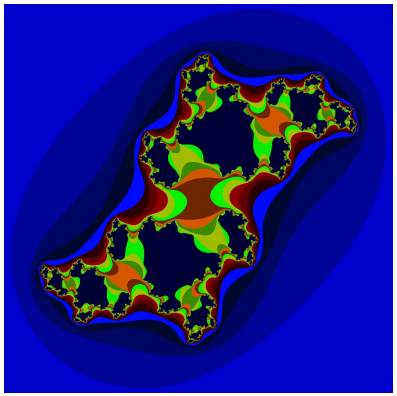


1. run (`make`) the fractal program on the ARM
   - make sure you understand the program structure
   - what & how the fractal computes is not important



# fractal
## [tutorial/fractal-arm/fractal.c]





2. port the fractal program to a RISC-V partition (partition_2_2/fractal-riscv.c) that sends the BMP header & sequence of RGB values from the RISC-V to the ARM
   - create part._arm_fractal/fractal-arm.c to write the stream of received bytes to a fractal.bmp file
     - hints
       - re-use the part._arm_echo directory, incl. the Makefile
       - remove the echo-arm.c file, otherwise you get the error "multiple definition of `main'"
       - make sure that you always use at least one xil_printf in the RISC-V program on each core, otherwise you get the error "_dynmalloc unknown"
     - hint: a BMP of size X*Y pixels contains (y*(3*x+x%4)+54)bytes
     - hint: there is no need to store the FrameBuffer on the RISC-V (in fact, you'll run out memory immediately)
   - run & check that the output is correct
     - use linux `cmp` command
     - display the `fractal.bmp` image by double-clicking the file in MobaXterm or by using: `eog fractal.bmp`
     - you need to install `eog` with the command `sudo apt install eog`
     - make sure that X forwarding is enabled (`ssh -X` on Linux or `ssh -Y` on MacOS)

eog does not work with re-attached tmux sessions

3. extend your application such that it asks for an image size, and then generates a fractal of that size
4. parallelise the fractal: generate half the fractal on tile 0, partition 2 and and the other half on tile 1, partition

now commit your fractal solutions to your gitlab repo