UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Stream-based Runtime Verification

Hannes Kallwies[1] Martin Leucker[1] César Sánchez[2] Malte Schmitz[1]
Volker Stolz[3] **Daniel Thoma[1]**

[1]Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany

[2]IMDEA Software Institute, Spain

[3]Western Norway University of Applied Sciences

FM'19 – 3rd wolrd congress on formal methods

Agenda

09:00 Practical Demonstration: Processor Tracing with SRV
Daniel Thoma, University of Lübeck

10:00 Coffee Break

10:30 Stream-based Runtime Verification
César Sánchez, IMDEA Software Institute

11:30 TeSSLa – Temporal Stream-based Specification Language
Hannes Kallwies, University of Lübeck

12:30 Lunch Break

14:00 TeSSLa in Practice: Specifying and Monitoring
Malte Schmitz, University of Lübeck

15:30 Coffee Break

16:00 In-depth Applications
Volker Stolz, Høgskulen på Vestlandet

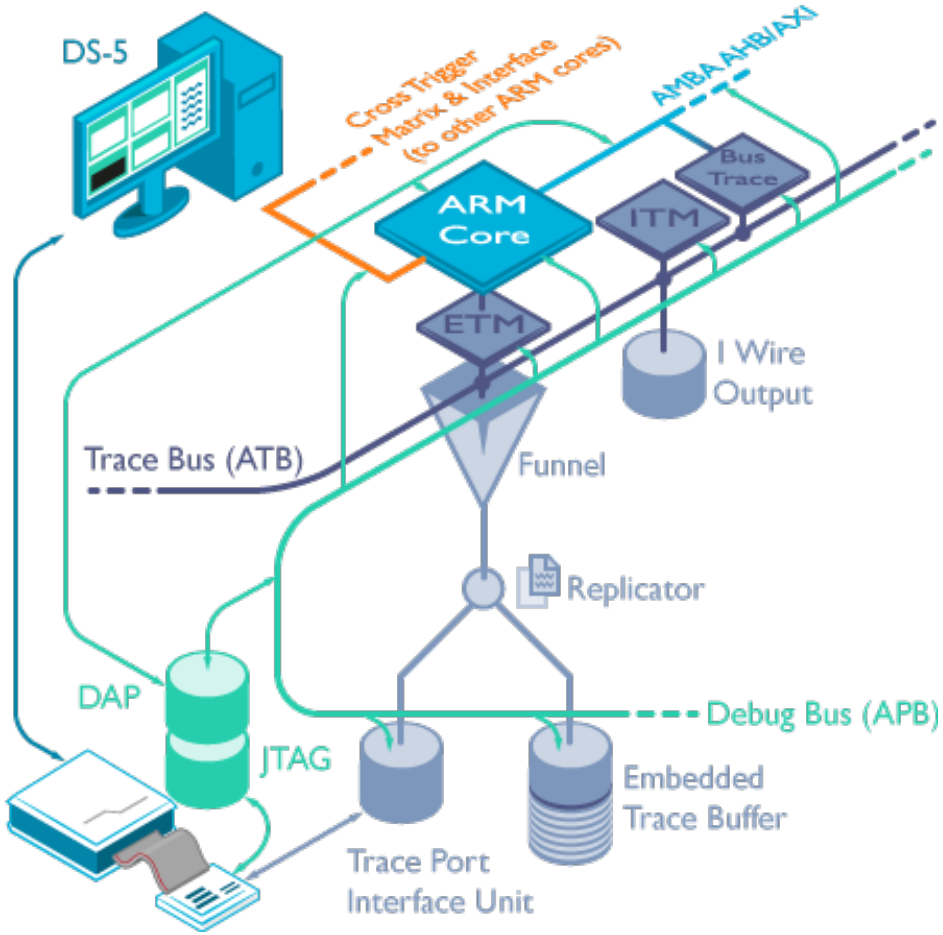
17:00 Closing and Future Prospects
Martin Leucker, University of Lübeck

Outline

State of the Art Hardware Monitoring / Debugging

- Embedded, hybrid and cyber-physical systems have tight time and resource constraints.
- Comprehensive logging output in the software (e.g. via instrumentation) decreases the performance significantly.
- Breakpoint-based debugging features of the processor are slow due to the potentially high number of interruptions.
- Logging and breakpoints are highly intrusive.
  Problematic for concurrent programs or real-time applications.

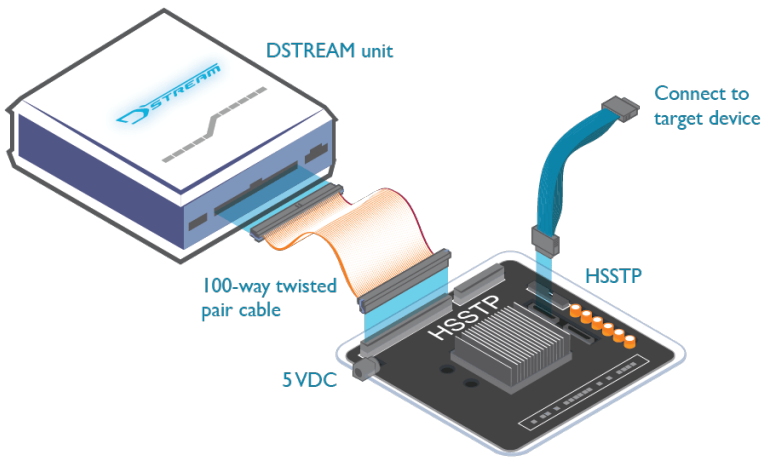
Embedded Tracing Unit: ARM CoreSight

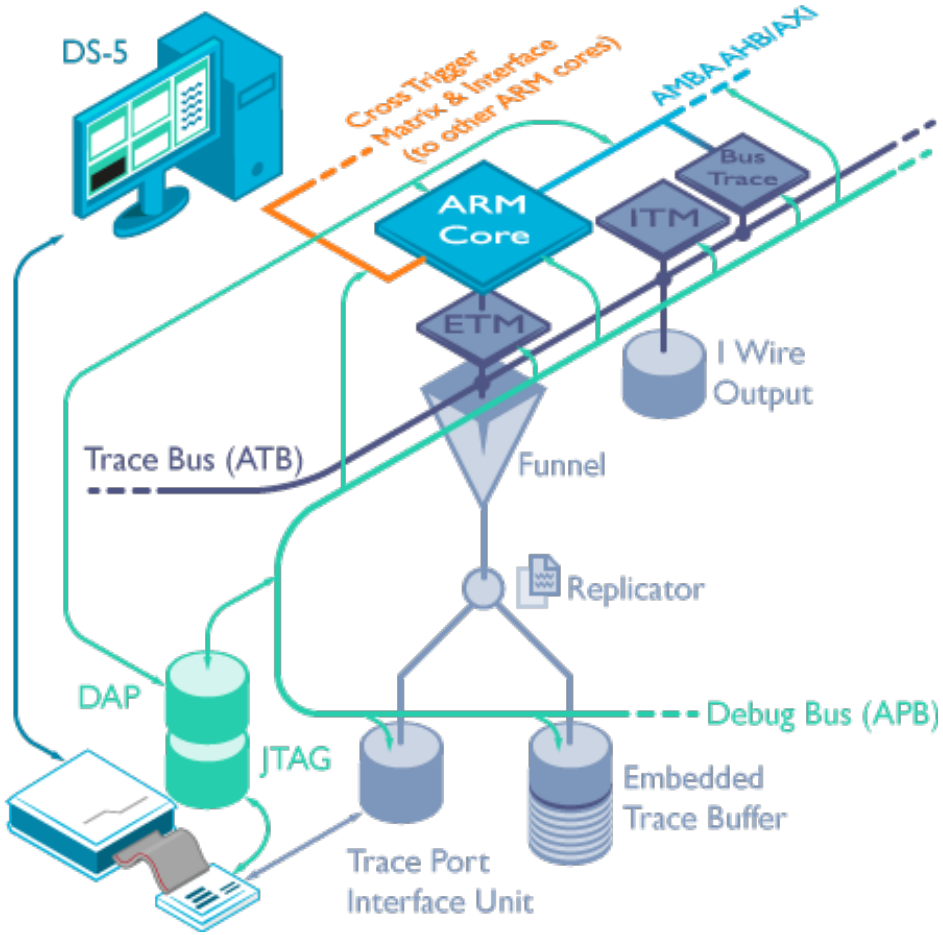


Embedded Tracing Unit: ARM CoreSight

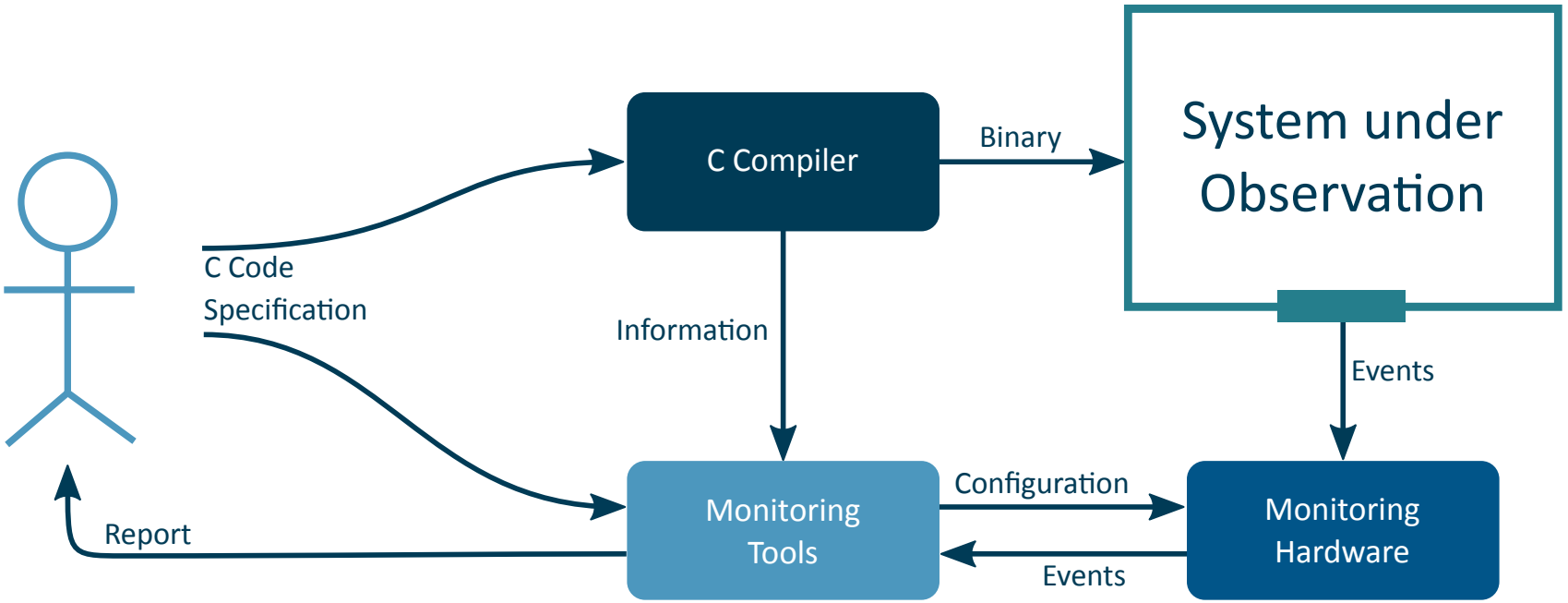
DSTREAM

- Record trace for offline reconstruction and analysis.
- Traces can be recorded for at most a few seconds.
- Trace buffer of 4GB for a recording speed of 10 Gbit/s.

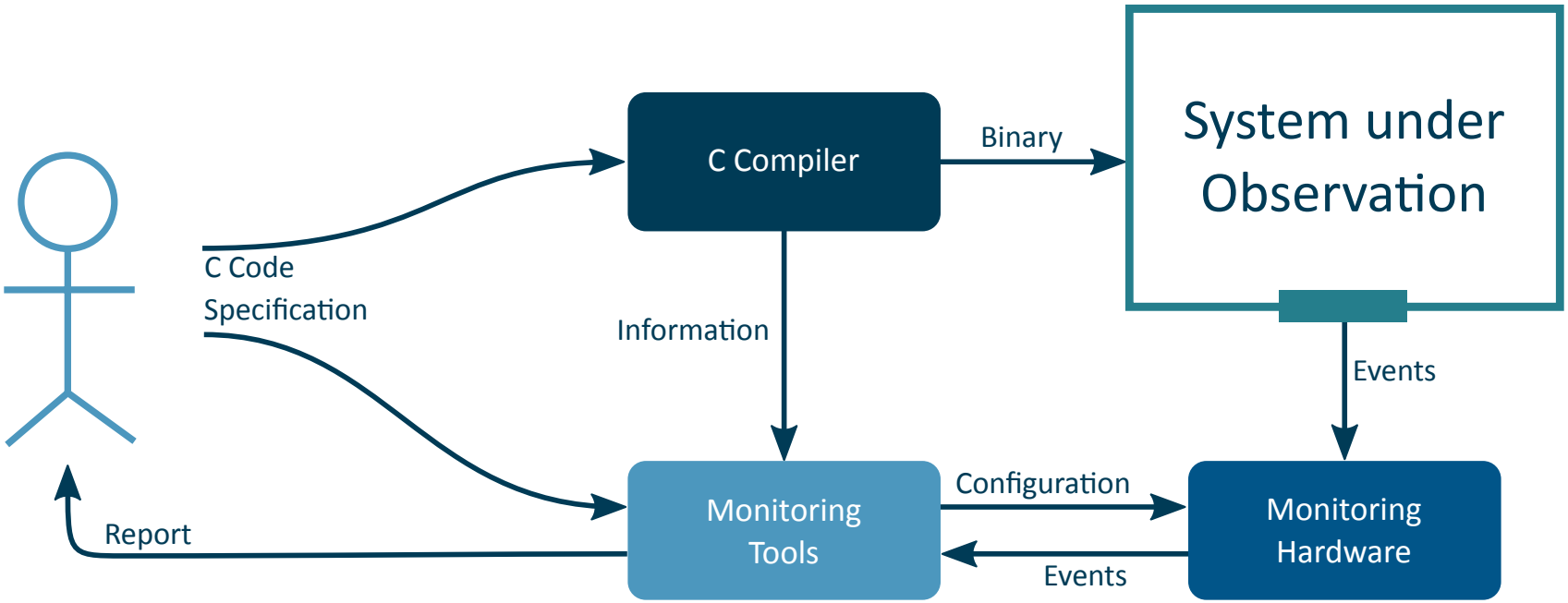




Interactive Hardware Monitoring Workflow



Interactive Hardware Monitoring Workflow



Fast reconfigurablility for interactive debugging sessions

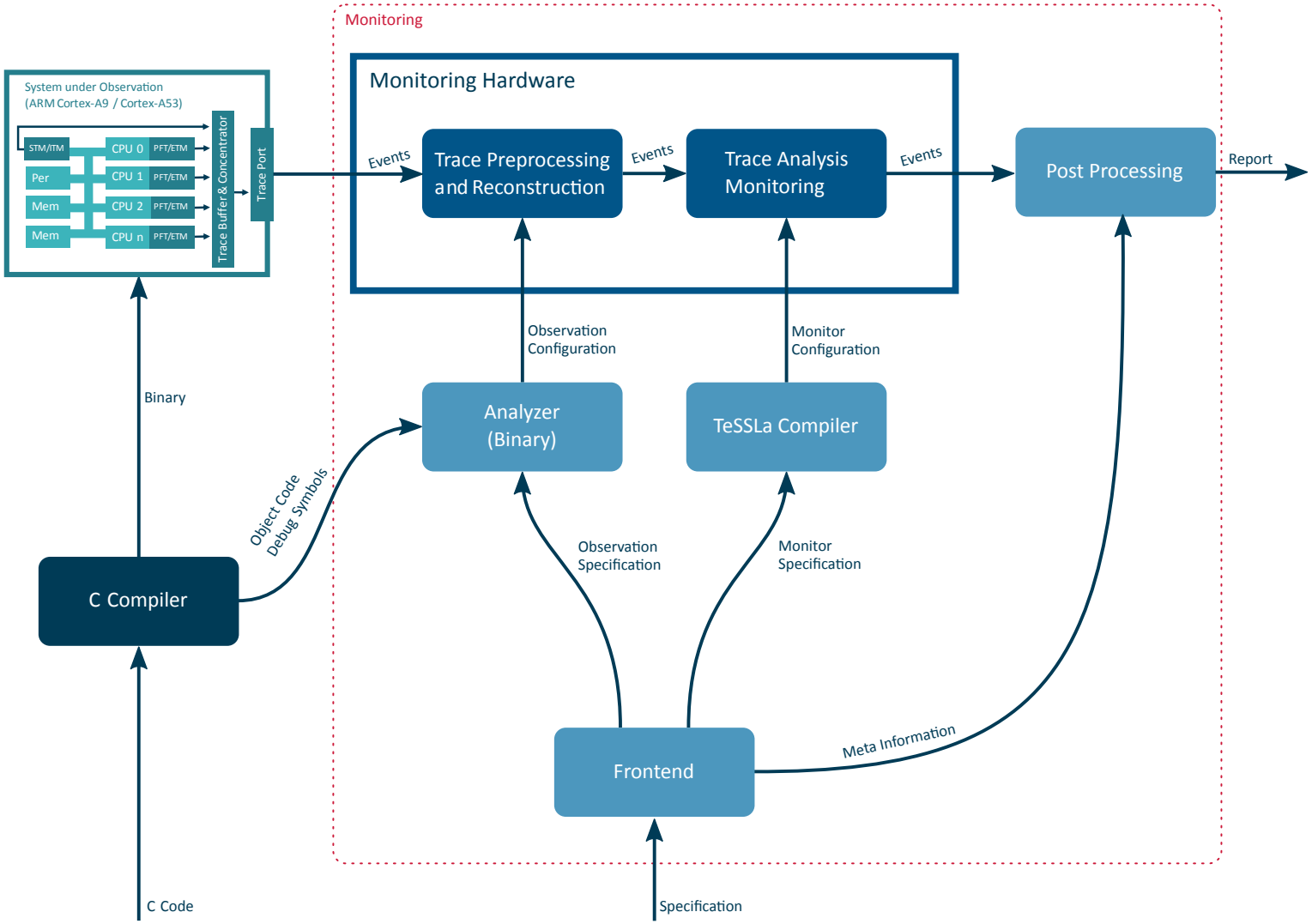
Synthesized FPGA design

- Trace Reconstruction
- Monitoring Engine

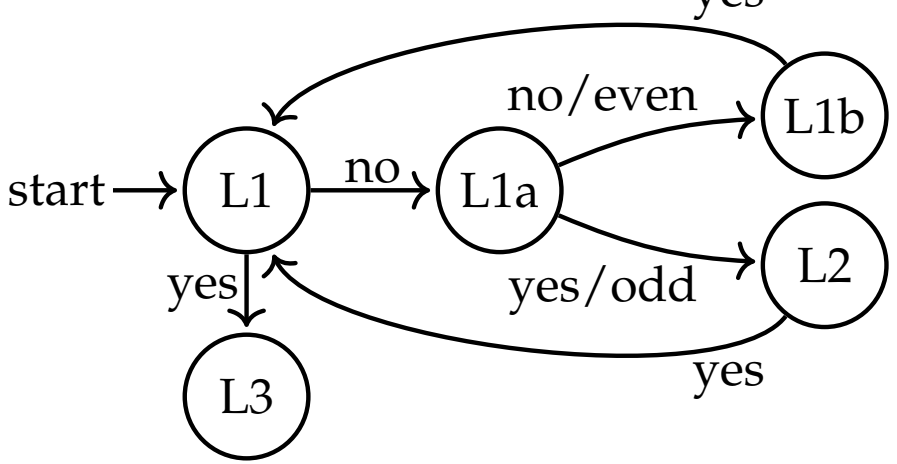
configured through memory with

- program binary (LUT of jumps)
- monitoring specification

Monitoring Program Flow Trace: Tools



Trace Reconstruction



C Code

```
while (n > 1) {


  if (n % 2 == 0) {



    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

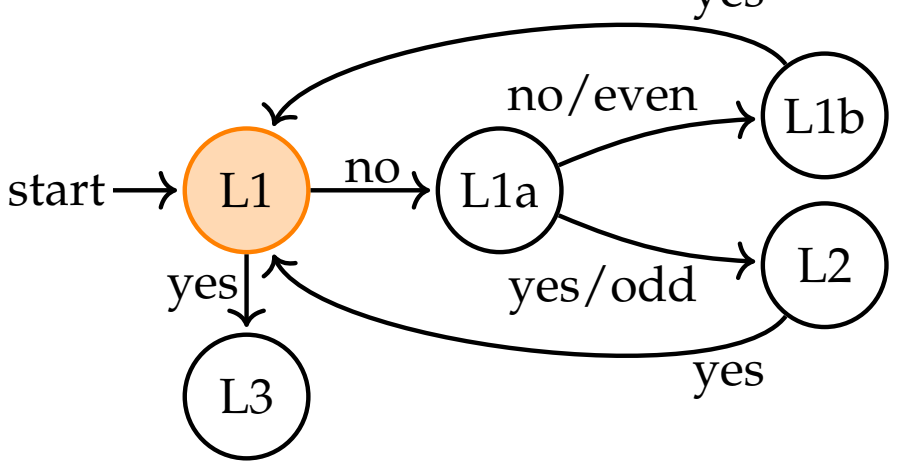
Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

Events

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

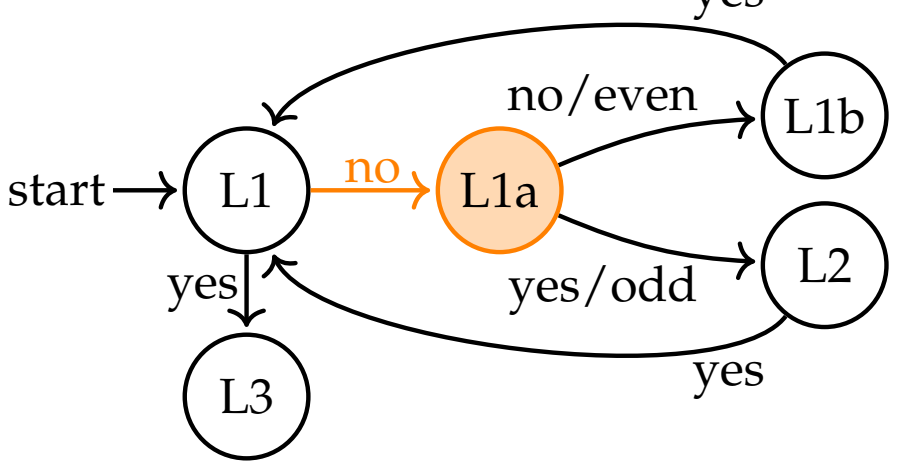
Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

Events

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

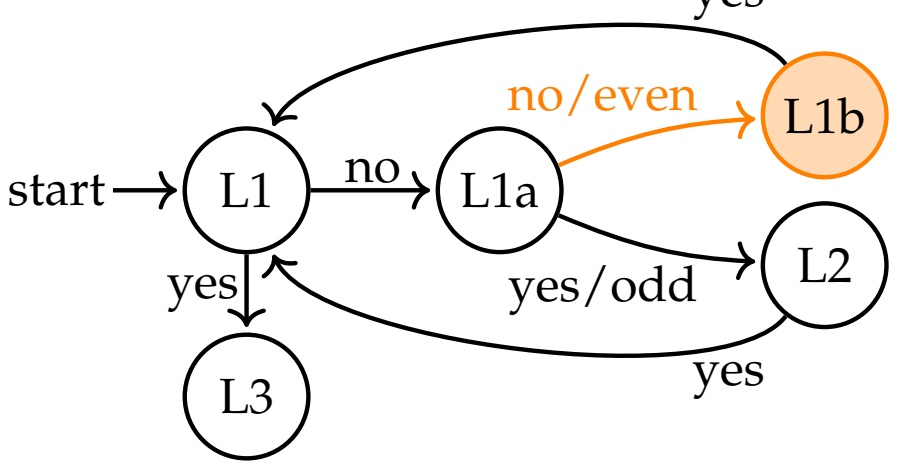
```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

no

Events

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

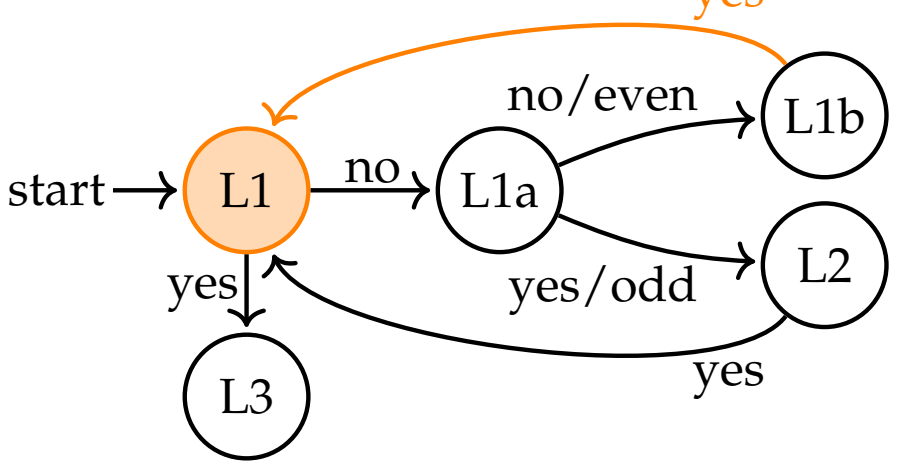
no

no

Events

even

Trace Reconstruction



C Code

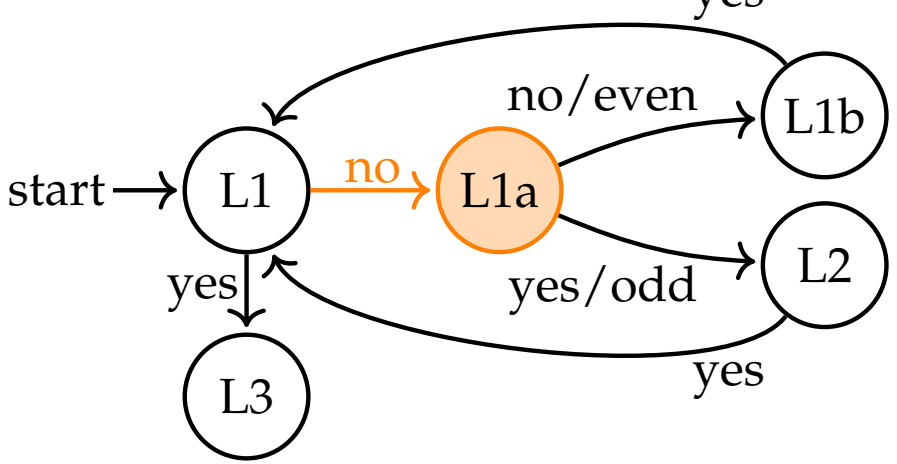
```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	

Trace Reconstruction



C Code

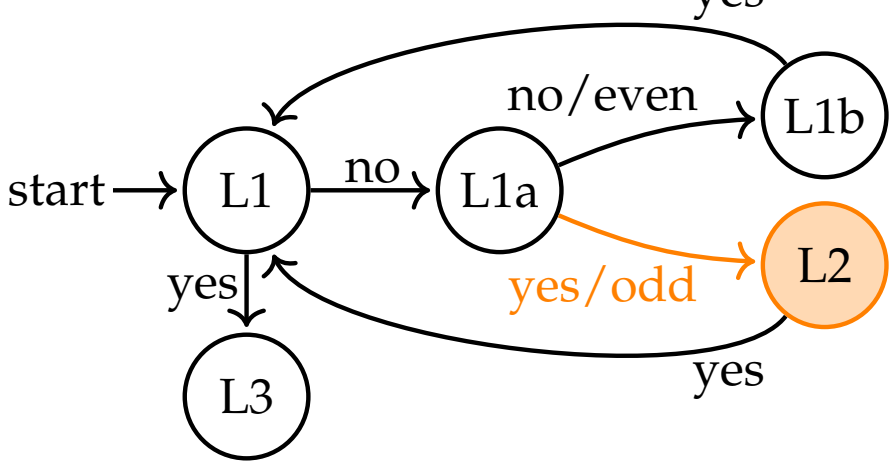
```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	

Trace Reconstruction



C Code

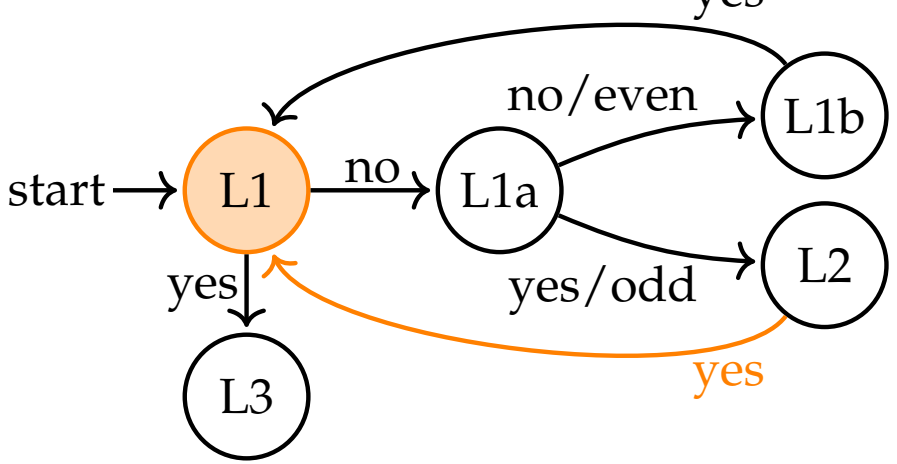
```
while (n > 1) {


  if (n % 2 == 0) {



    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	
yes	odd

Trace Reconstruction



C Code

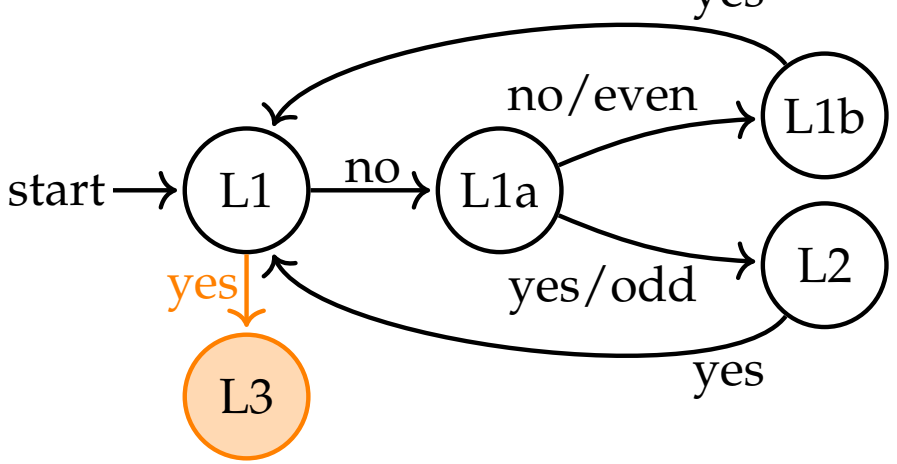
```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	
yes	odd
yes	

Trace Reconstruction



C Code

```
while (n > 1) {


  if (n % 2 == 0) {



    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	
yes	odd
yes	
yes	

Trace Reconstruction

Tracing between jumps

- Instructions between jumps are executed consecutively
- Indirect jumps to arbitrary addresses
- Interrupt handlers can be called at any time
- Lookup-table of all code addresses

Performance

- complete trace is huge
- FPGA significantly slower than target
- Only generate events of interest
- Reconstruct in parallel
- Split at absolute jumps or sync packages

Multi-threading

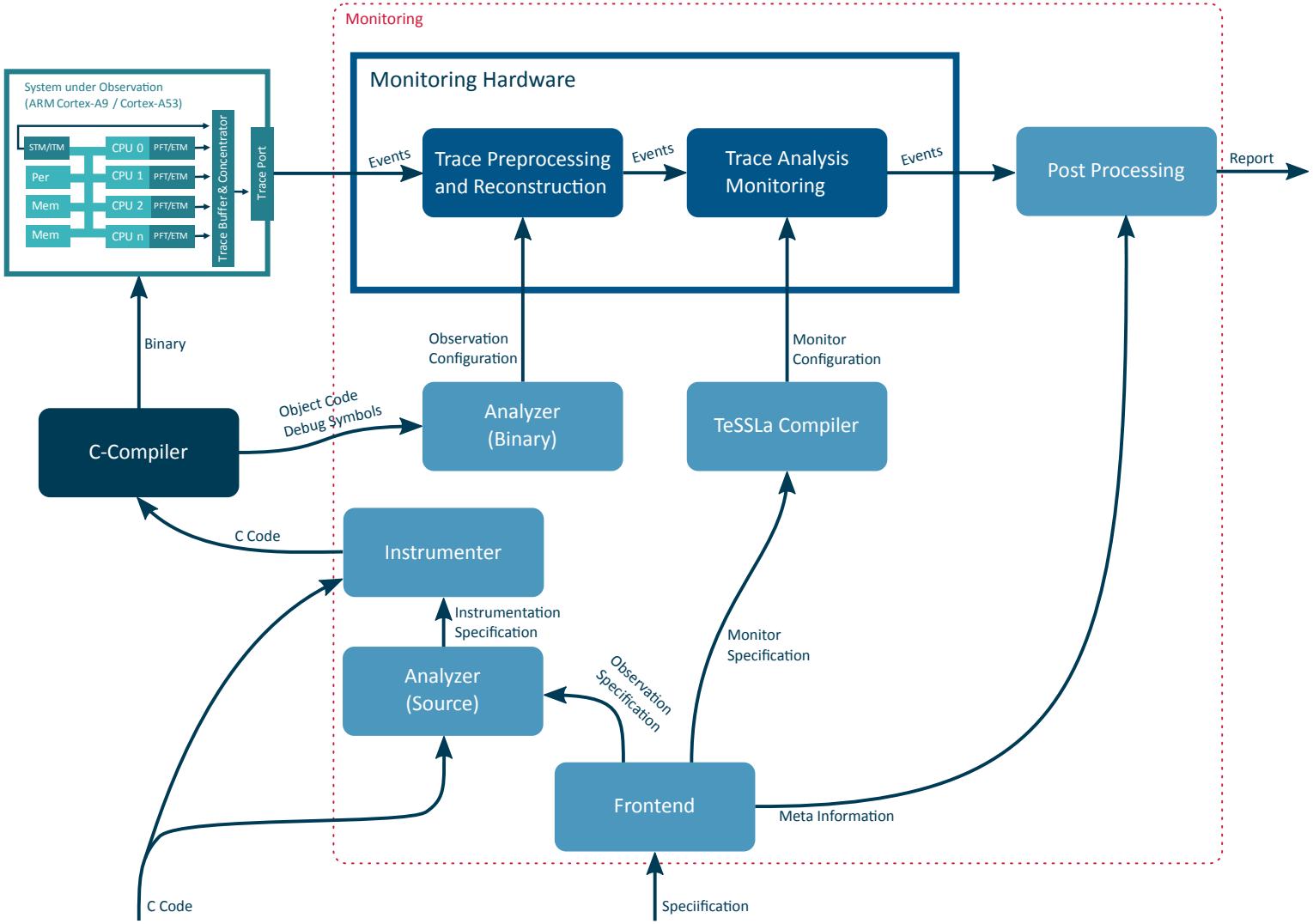
Problem: Distinguish multiple threads

- We can only distinguish instructions traced from the different cores.
- Scheduler can execute multiple threads on the same core.
- Context switch reconfigures MMU
  $\Longrightarrow$ Same logical addresses used in different threads.

Solution: Context ID Register

1. OS writes thread ID to context ID register.
2. Tracing unit generates context ID message.
3. Trace reconstruction changes lookup table
   for the program flow reconstruction information.

Monitoring Data Values



Stream Processing: Technical Prerequisites

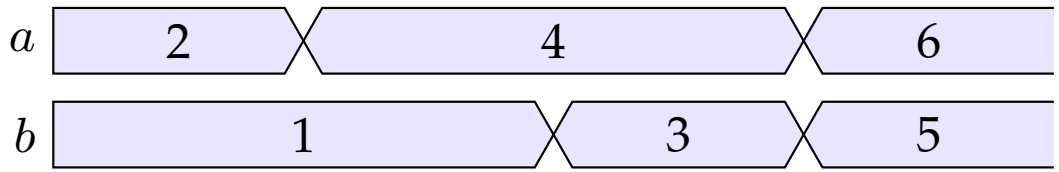
- Multi-core CPUs generate large amounts of trace data.
  $\Longrightarrow$ Perform monitoring in hardware.
- FPGAs have limited amount of memory.
  $\Longrightarrow$ Explicit memory usage. Constant memory usage per operator.
- Properties and analyses might become very complex.
  $\Longrightarrow$ Combined monitoring on hardware and in software.
- Timing is crucial in embedded and cyber-physical systems.
  $\Longrightarrow$ Support time as first-class citizen.
- Properties and analyses might require data.
  $\Longrightarrow$ Support analyses and aggregation of data values.

Stream Processing with TeSSLa

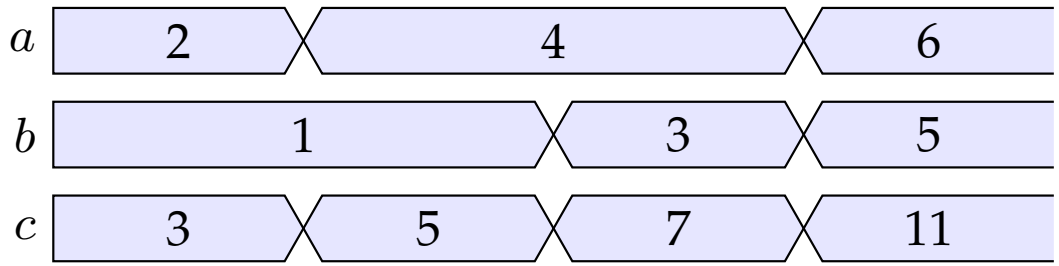
www.tessla.io

Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, Daniel Thoma.
TeSSLa: Temporal Stream-based Specification Language.
SBMF 2018, CoRR abs/1808.10717.

TeSSLa by Example

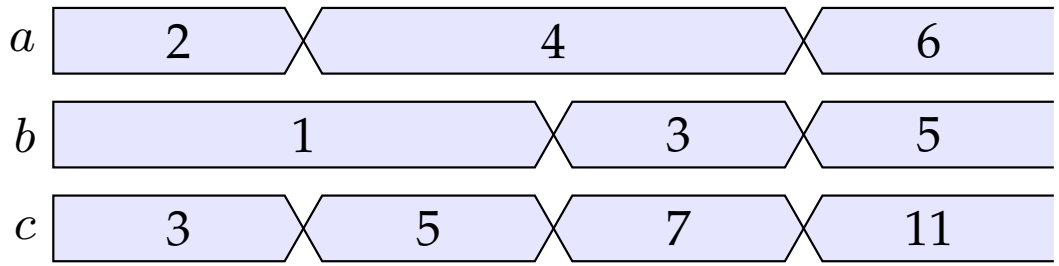


TeSSLa by Example

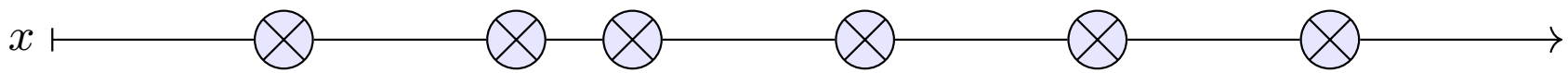



```
def c := a + b
```

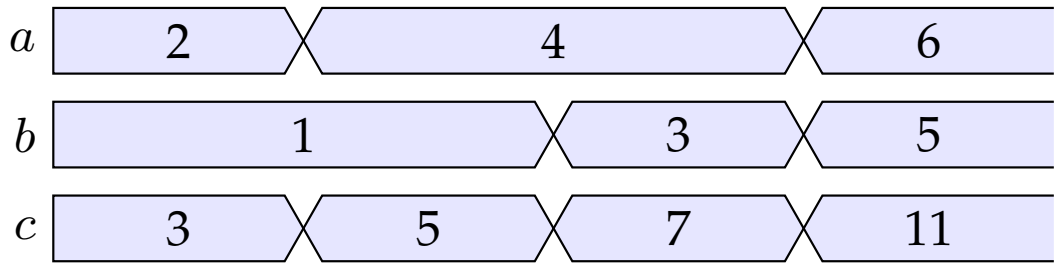
TeSSLa by Example



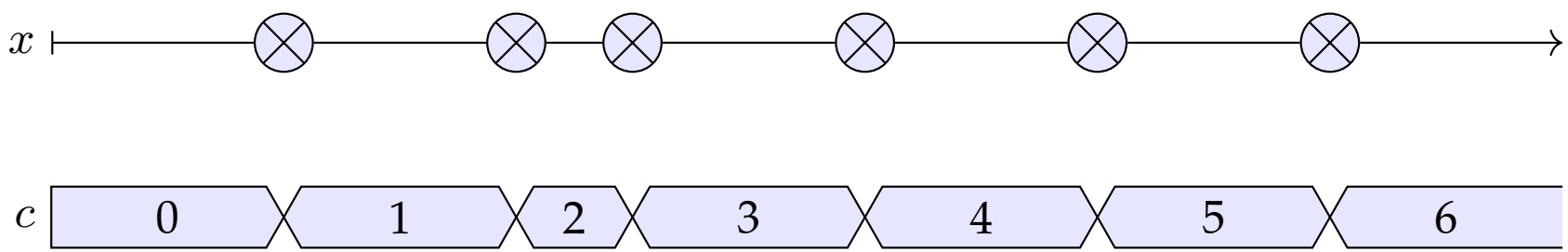

```
def c := a + b
```



TeSSLa by Example

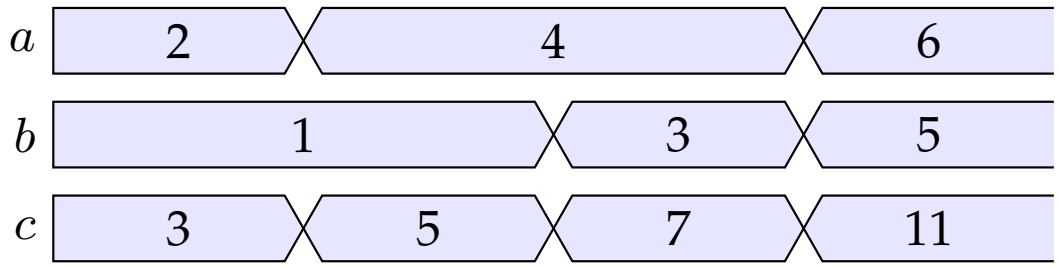



```
def c := a + b
```

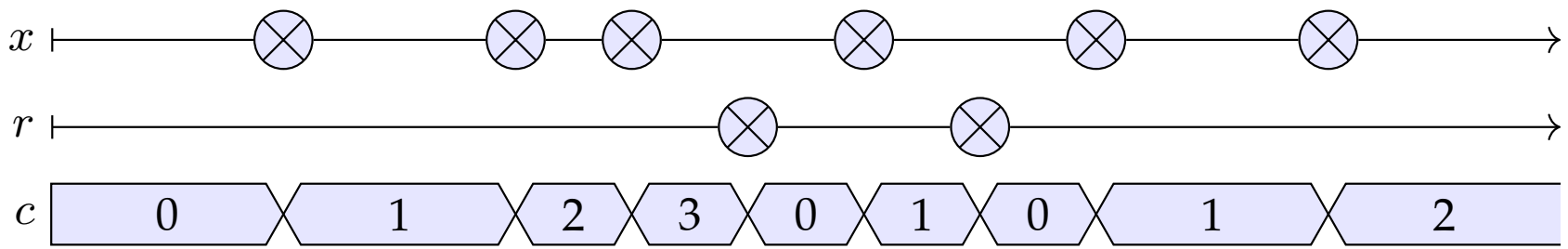



```
def c := eventCount(x)
```

TeSSLa by Example




```
def c := a + b
```




```
def c := eventCount(x, reset = r)
```

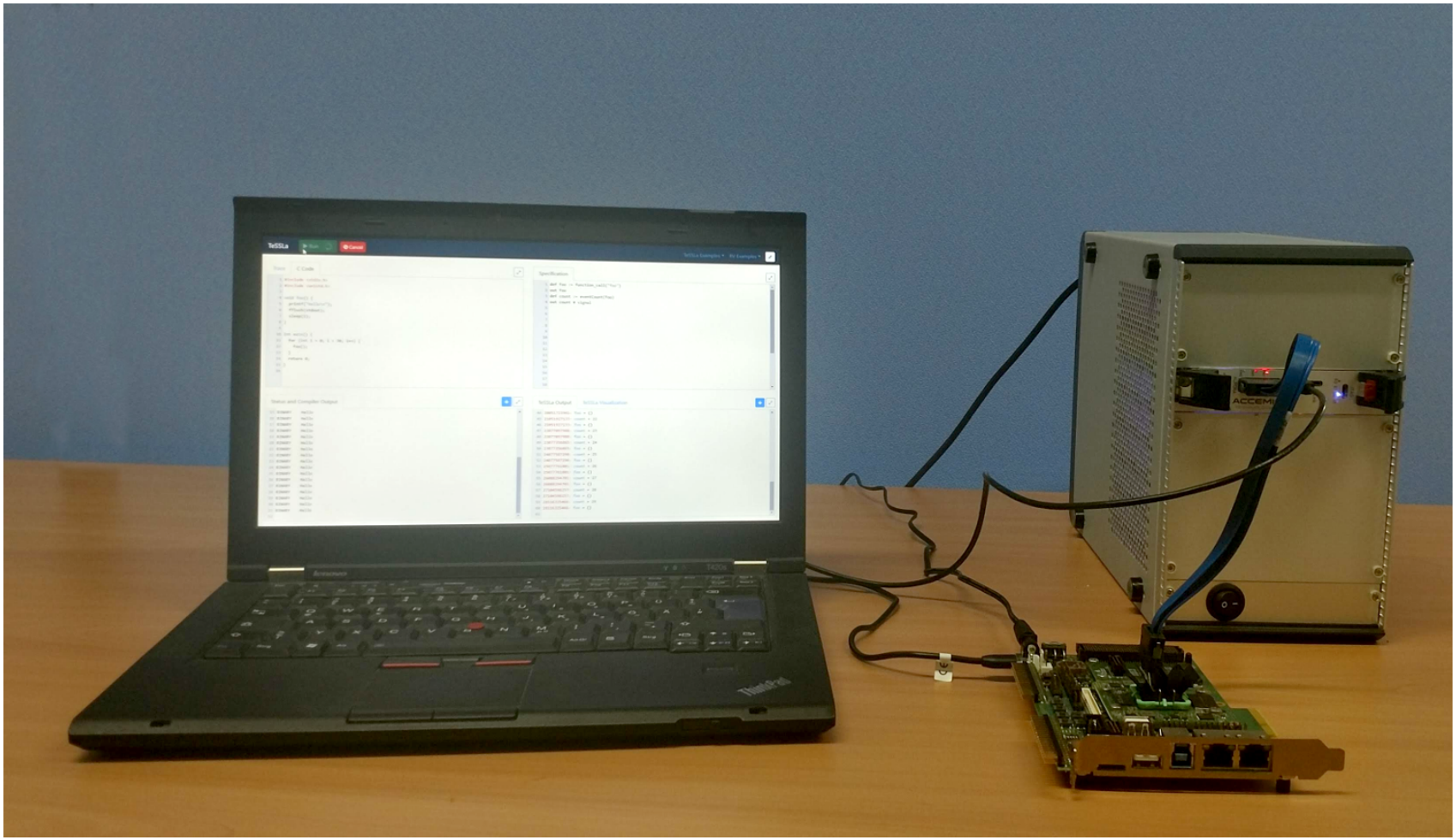
Control Flow Events

```
def @FunctionCall(name: String)

def @FunctionCalled(name: String)

def @FunctionReturn(name: String)

def @FunctionReturned(name: String)

def @LabelReached(name: String)
```

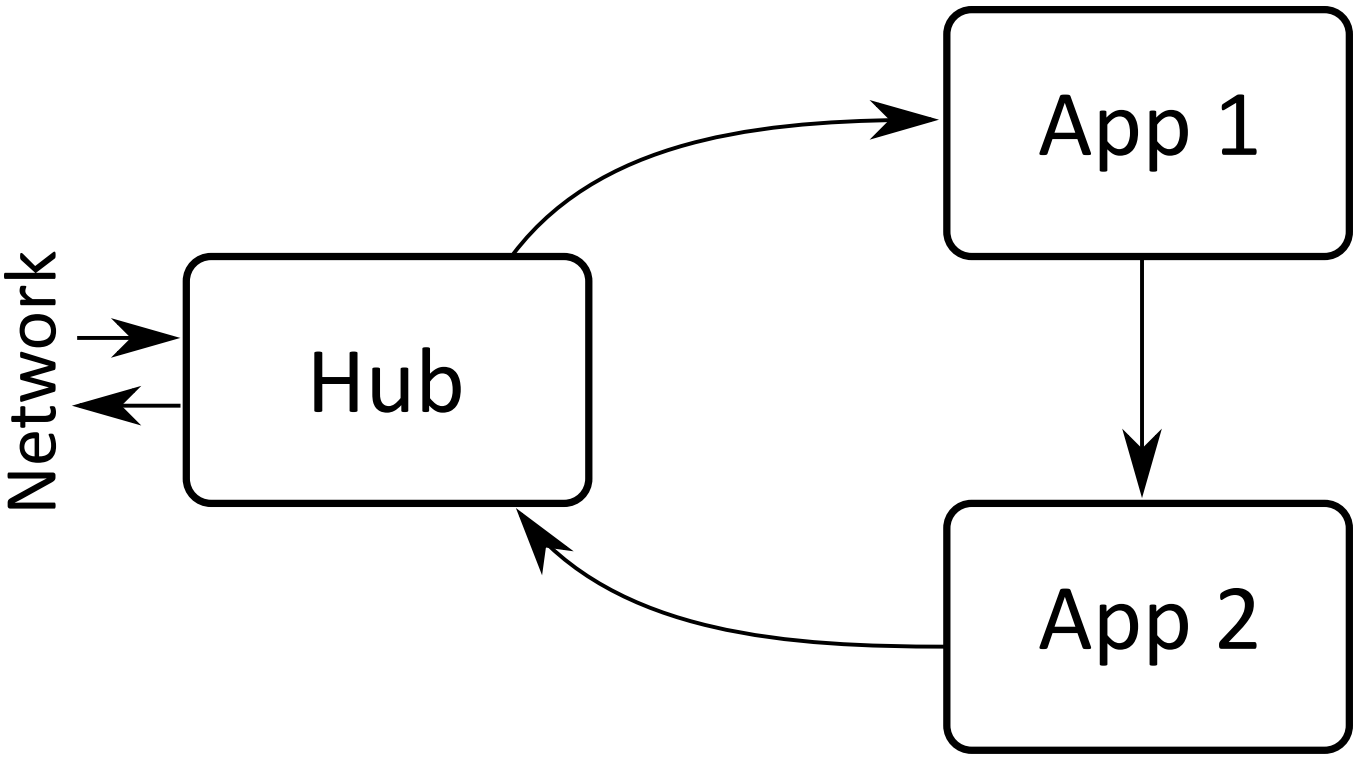
Instrumentation Events

```
def @InstFunctionCallArg(name: String, index: Int)

def @InstFunctionCalled(name: String)
def @InstFunctionCalledArg(name: String, index: Int)

def @InstFunctionReturn(name: String)
def @InstFunctionReturnValue(name: String)

def @InstFunctionReturned(name: String)
def @InstFunctionReturnedValue(name: String)

def @GlobalRead(lvalue: String)
def @LocalRead(lvalue: String, function: String)

def @GlobalWrite(lvalue: String)
def @LocalWrite(lvalue: String, function: String)
```

Practical Hardware Setup and Demonstration



Practical Hardware Setup and Demonstration



Conclusion

1. COEMS provides interactive debugging and continuous monitoring for embedded, hybrid and cyber-physical systems.
2. Continuous monitoring of processor traces provided by embedded tracing units (ETU) requires
   - online trace reconstruction in hardware and
   - monitoring in hardware.
3. Using TeSSLa we can check
   - event ordering constraints,
   - timing constraints and
   - complex event patterns like the burst pattern.
4. TeSSLa can be used to aggregate data and compute statistical data.
5. Monitoring data values is possible with lightweight instrumentation.
6. There is also a software interpreter (see tessla.io).
7. Very small core language (2-3 operators).
8. Can express large classes of stream transformations
9. Open to many other applications