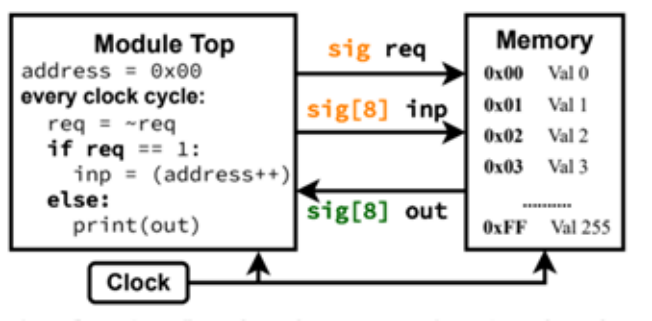
Anvil

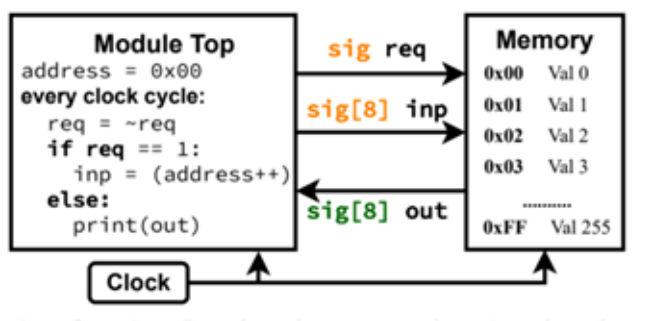
A General-Purpose Timing-Safe Hardware Description Language

Jason Zhijingcheng Yu*, Aditya Ranjan Jha*,
Umang Mathur, Trevor E. Carlson, Prateek Saxena

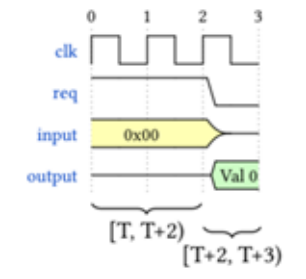
Timing Hazards in Hardware Designs



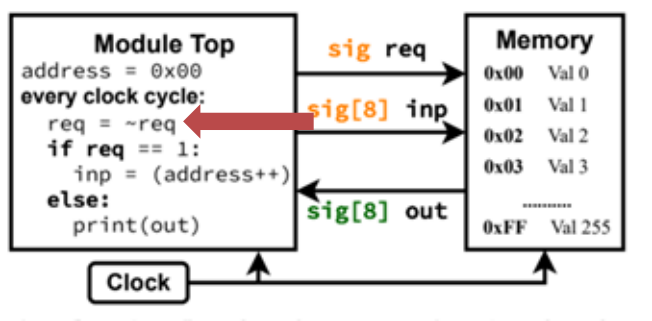
Timing Hazards in Hardware Designs



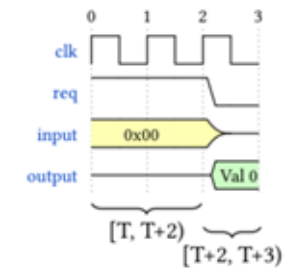
Protocol



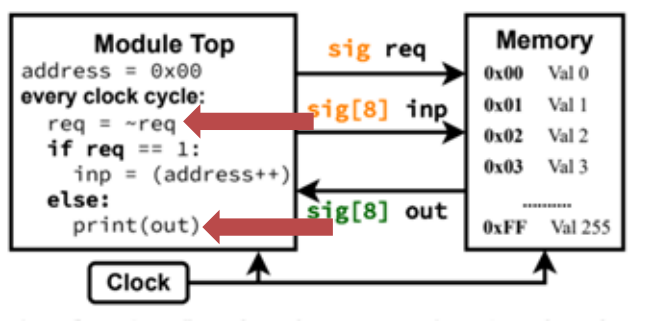
Timing Hazards in Hardware Designs



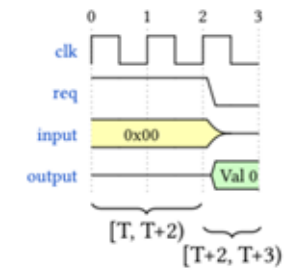
Protocol



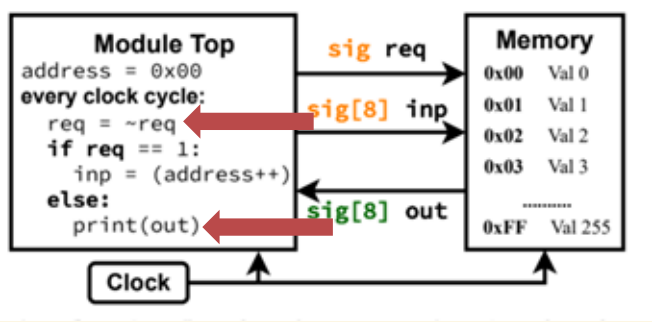
Timing Hazards in Hardware Designs



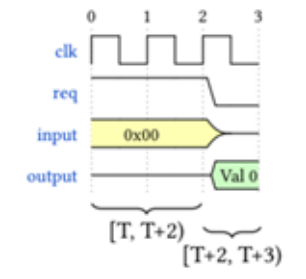
Protocol



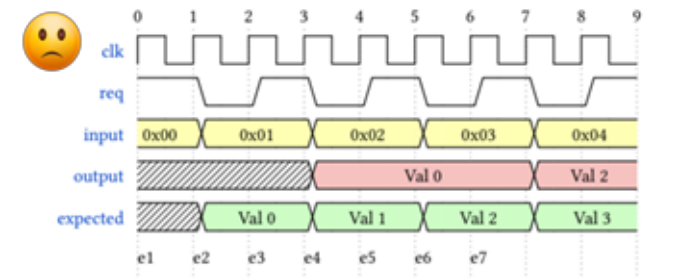
Timing Hazards in Hardware Designs



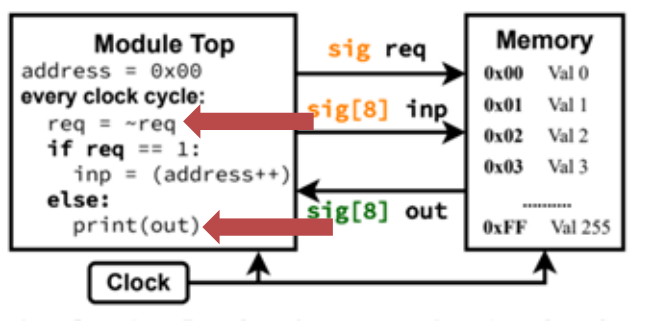
Protocol



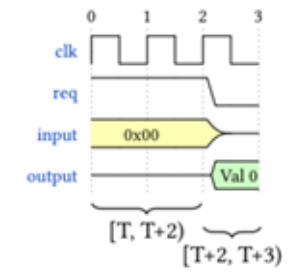
Reality



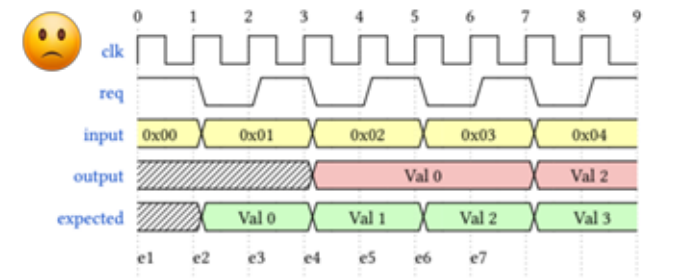
Timing Hazards in Hardware Designs



Protocol

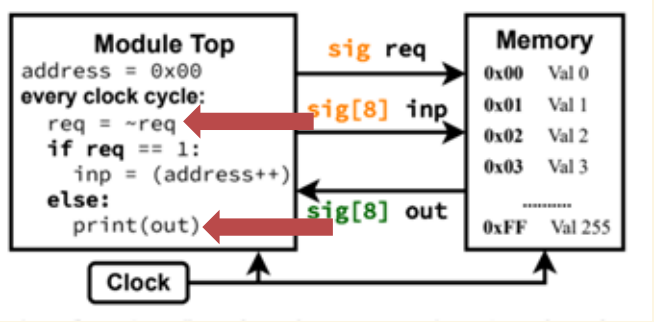


Reality



Timing Hazards

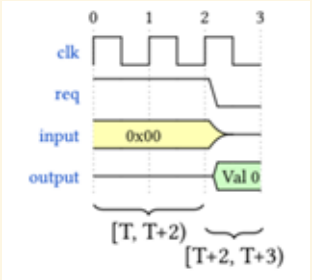
Timing Hazards in Hardware Designs



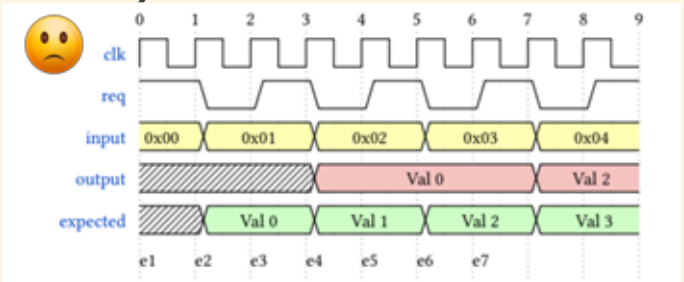
Timing Hazards

Expectation	Reality
Value keeps the same across cycles	Value changes halfway 😕

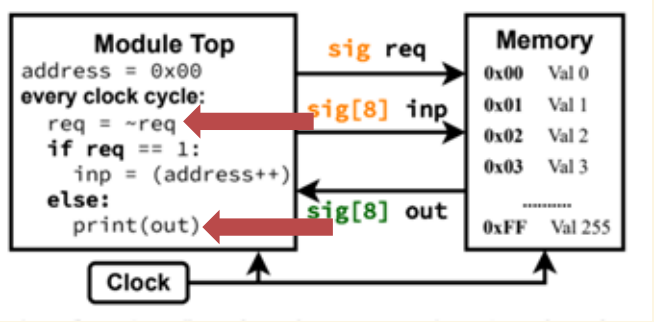
Protocol



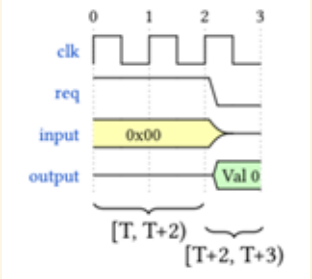
Reality



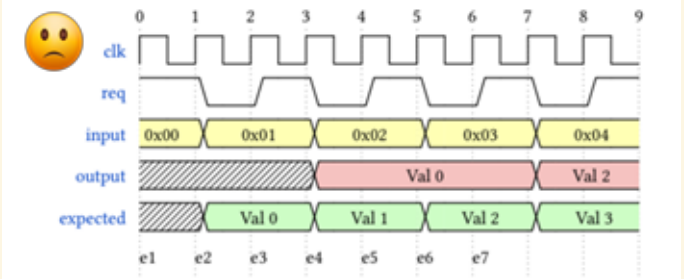
Timing Hazards in Hardware Designs



Protocol



Reality



Timing Hazards

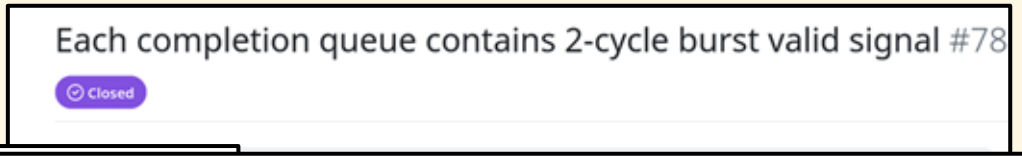
Expectation	Reality
Value keeps the same across cycles	Value changes halfway 😕
Value is meaningful	Value carries garbage 😕

Timing Hazards Are Ubiquitous!

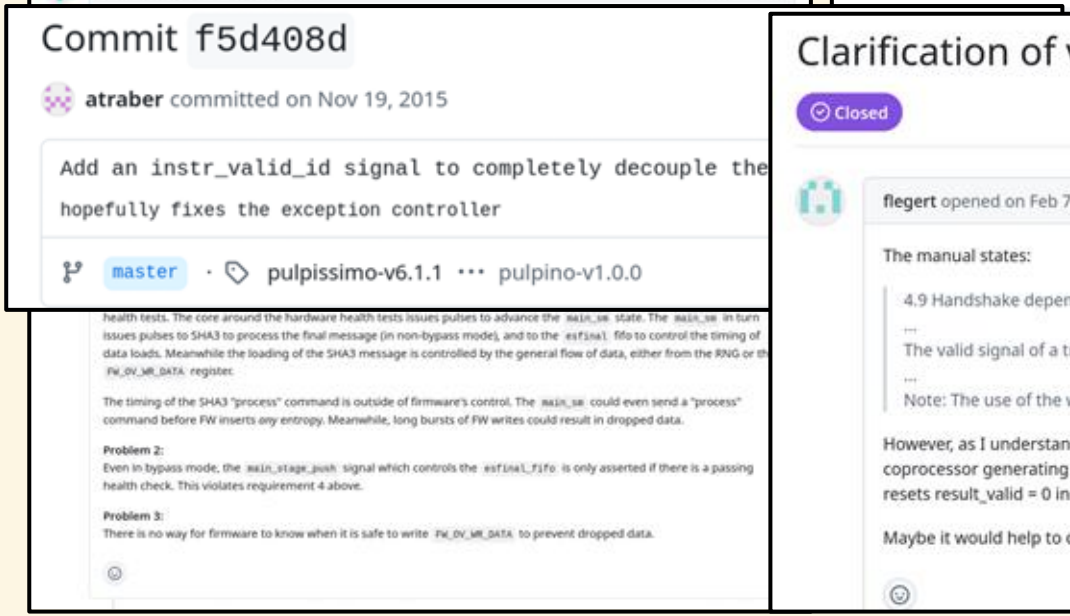

[entropy_src] Timing issues in FW_OV "Insert Entropy" feature

Closed #11003

martin-lueker opened on Feb 20, 2022 Contributor


Each completion queue contains 2-cycle burst valid signal #78

Closed


Commit f5d408d

atraber committed on Nov 19, 2015

Add an instr_valid_id signal to completely decouple the

hopefully fixes the exception controller

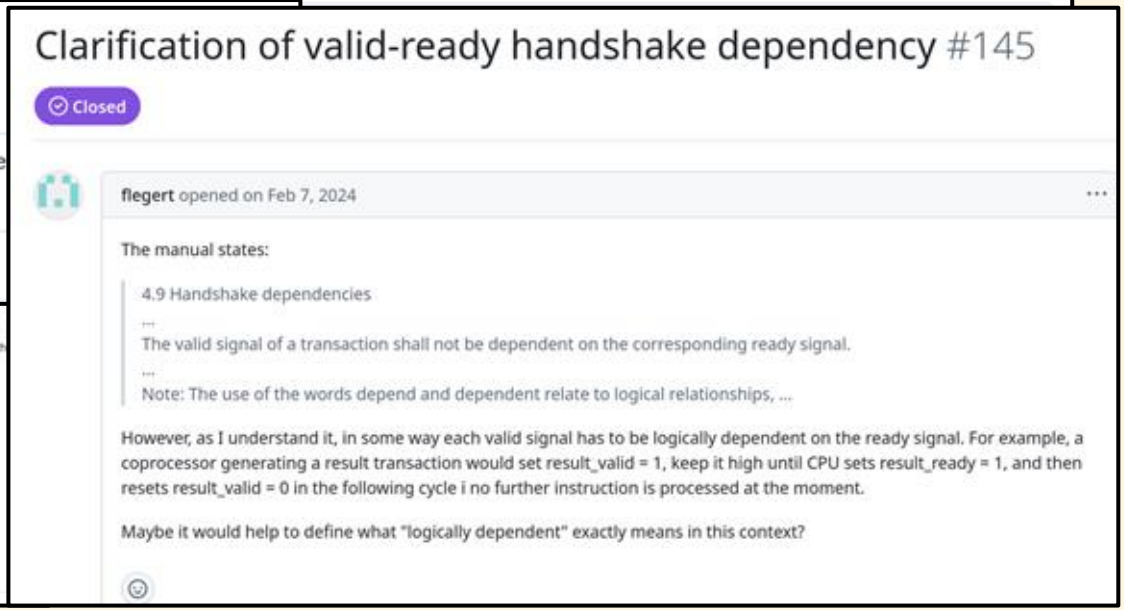
master · pulpissimo-v6.1.1 ··· pulpino-v1.0.0

health tests. The core around the hardware health tests issues pulses to advance the main_sm state. The main_sm in turn issues pulses to SHA3 to process the final message (in non-bypass mode), and to the esfinal fifo to control the timing of data loads. Meanwhile the loading of the SHA3 message is controlled by the general flow of data, either from the RNG or th FW_OV_WR_DATA register.

The timing of the SHA3 "process" command is outside of firmware's control. The main_sm could even send a "process" command before FW inserts any entropy. Meanwhile, long bursts of FW writes could result in dropped data.

Problem 2:
Even in bypass mode, the main_stage_push signal which controls the esfinal_fifo is only asserted if there is a passing health check. This violates requirement 4 above.

Problem 3:
There is no way for firmware to know when it is safe to write FW_OV_WR_DATA to prevent dropped data.


Clarification of valid-ready handshake dependency #145

Closed

flegert opened on Feb 7, 2024 ···

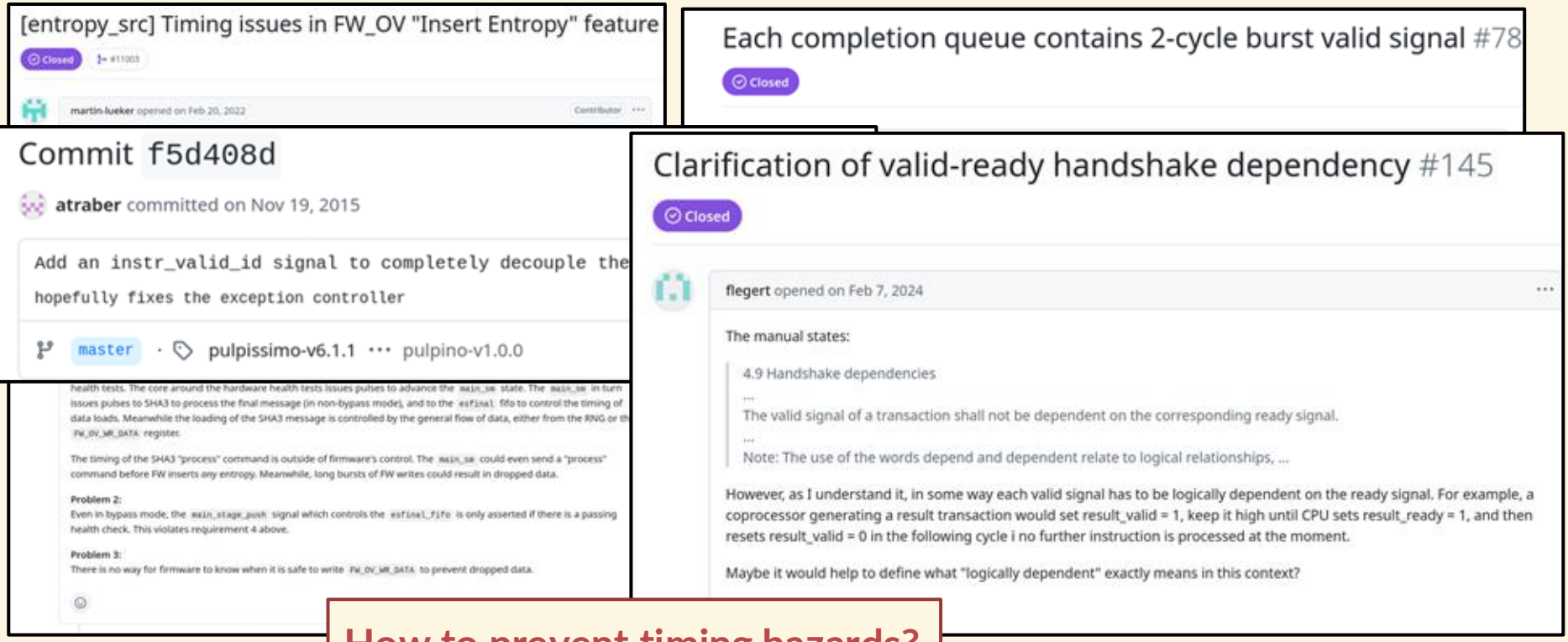
The manual states:

> 4.9 Handshake dependencies
> ...
> The valid signal of a transaction shall not be dependent on the corresponding ready signal.
> ...
> Note: The use of the words depend and dependent relate to logical relationships, ...

However, as I understand it, in some way each valid signal has to be logically dependent on the ready signal. For example, a coprocessor generating a result transaction would set result_valid = 1, keep it high until CPU sets result_ready = 1, and then resets result_valid = 0 in the following cycle i no further instruction is processed at the moment.

Maybe it would help to define what "logically dependent" exactly means in this context?

Timing Hazards Are Ubiquitous!



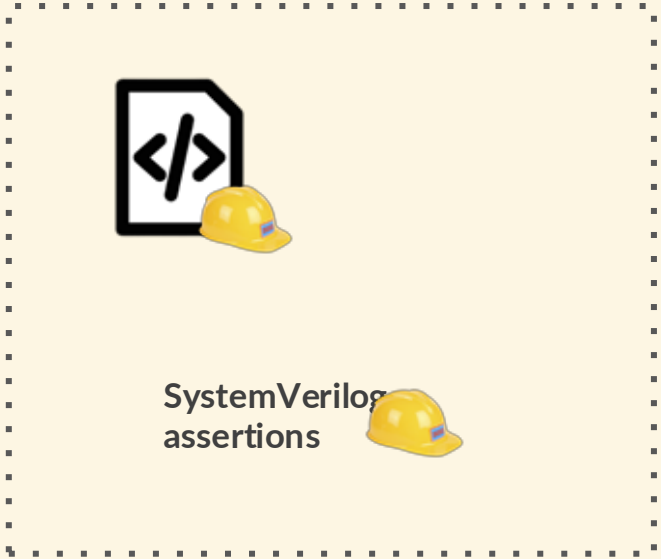
How to prevent timing hazards?

Post-design Verification is Difficult

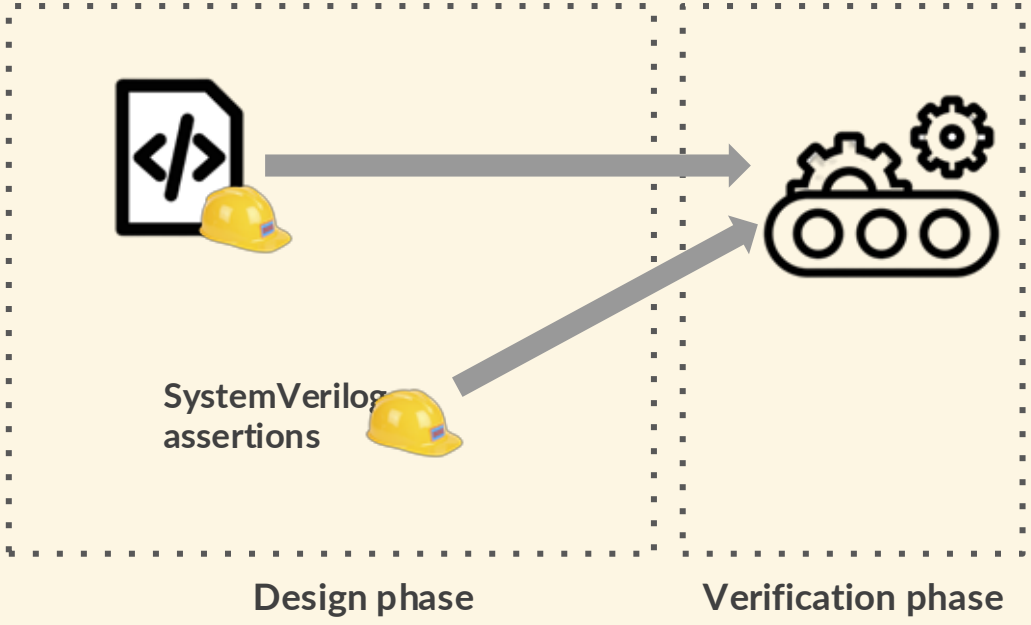
Post-design Verification is Difficult

SystemVerilog
assertions

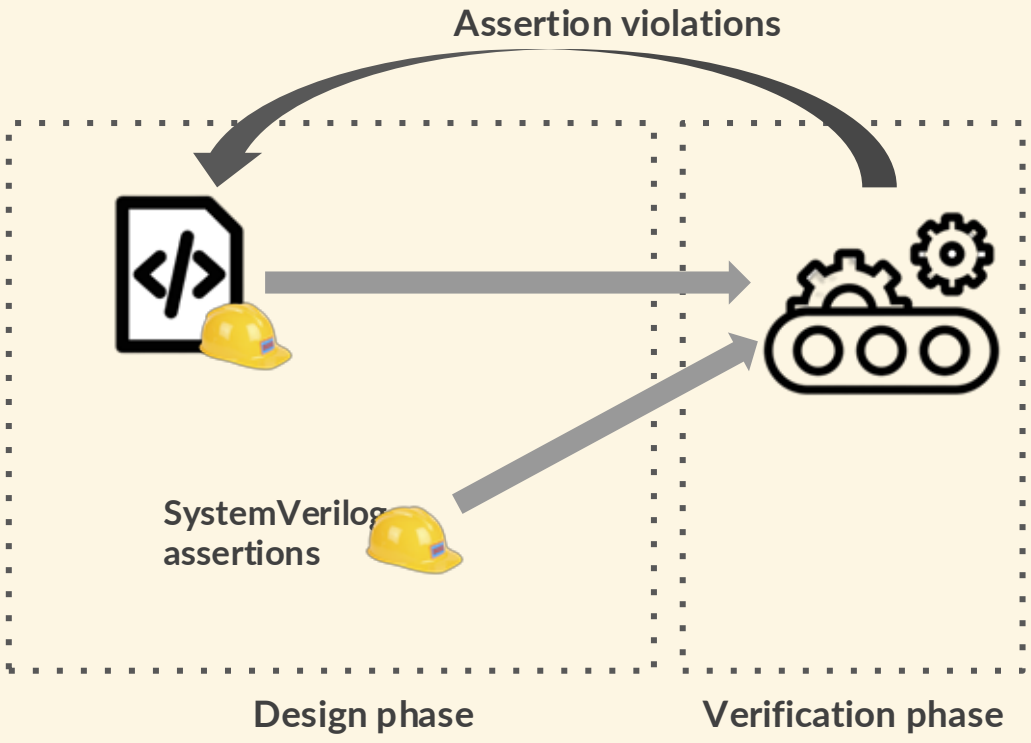
Post-design Verification is Difficult



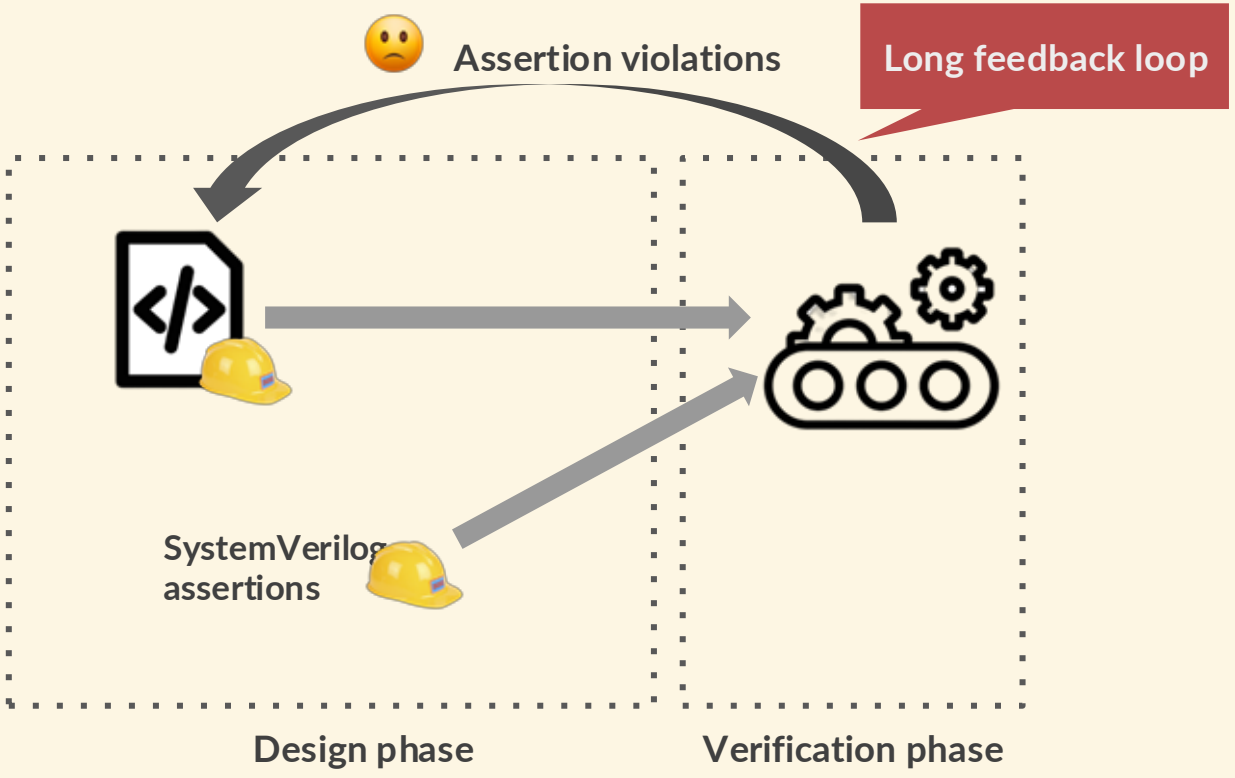
Post-design Verification is Difficult



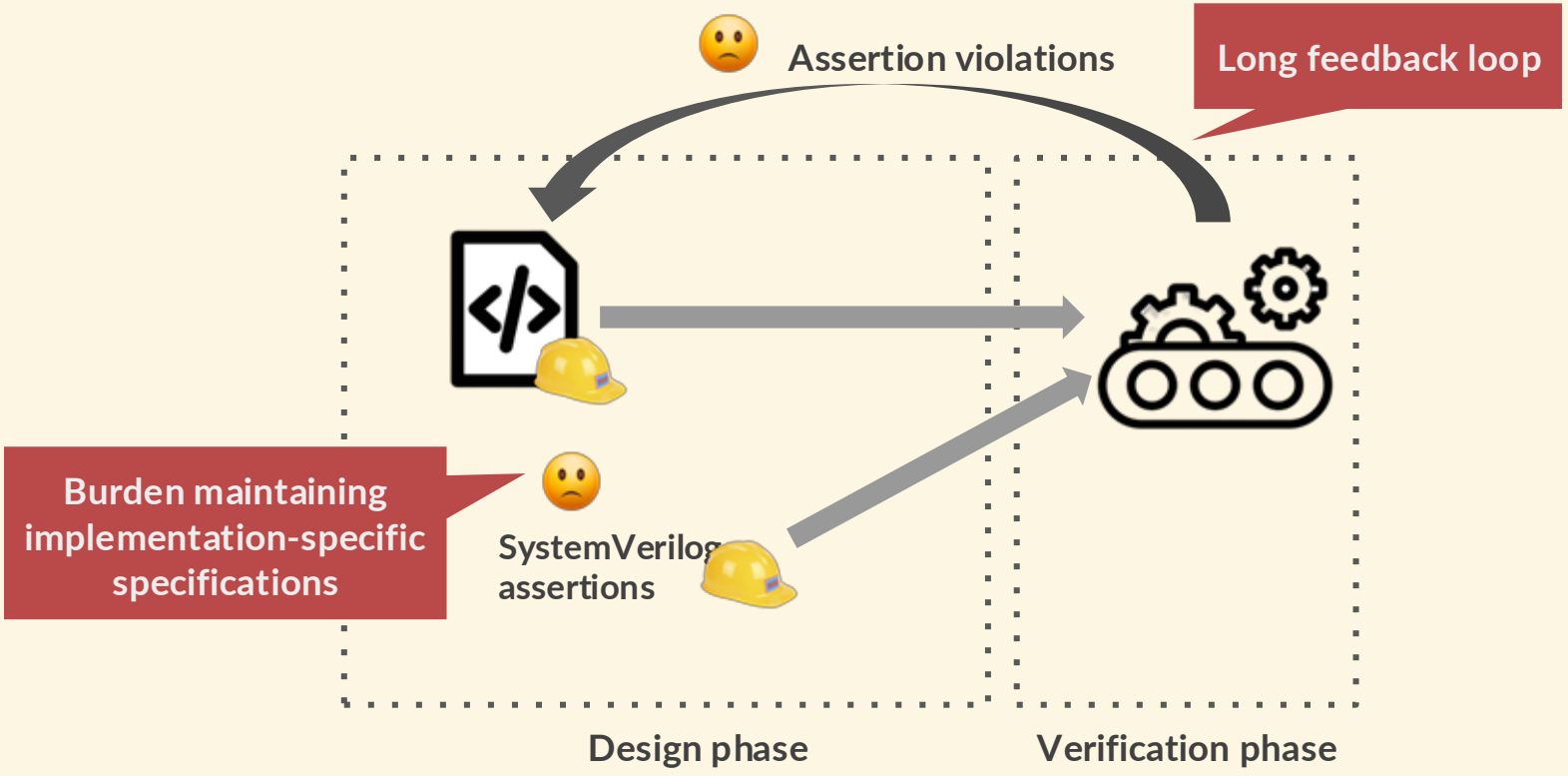
Post-design Verification is Difficult



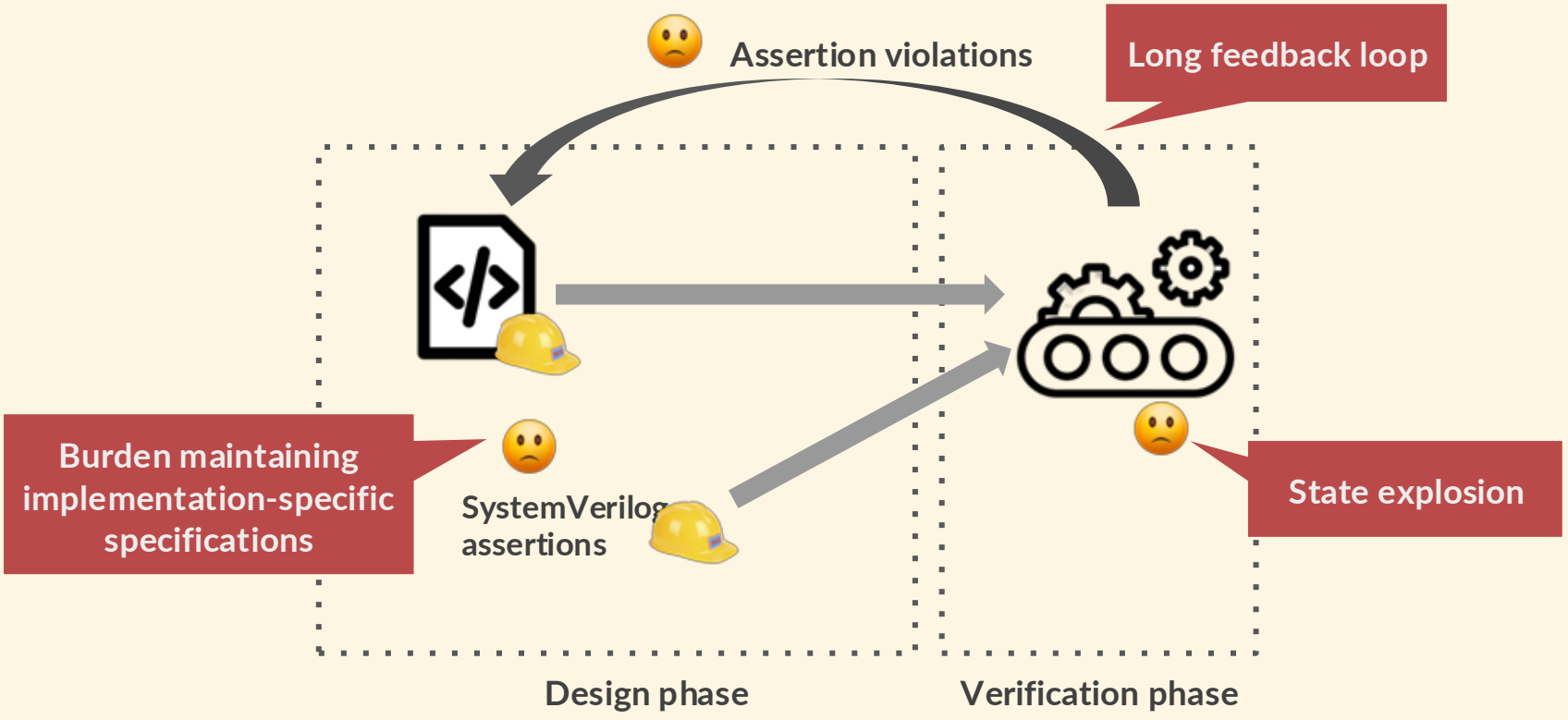
Post-design Verification is Difficult



Post-design Verification is Difficult

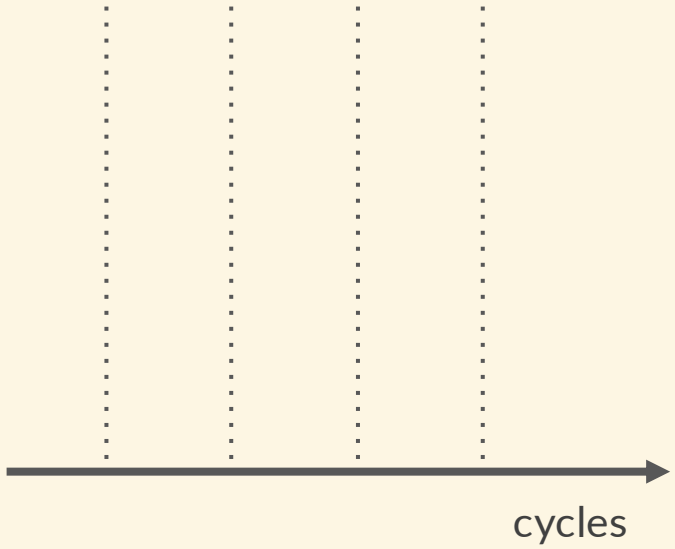


Post-design Verification is Difficult



Root Cause: Lacking Abstractions

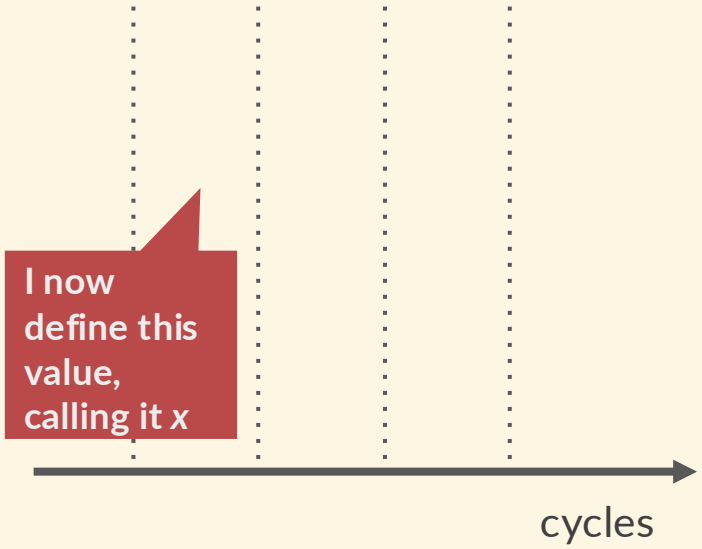
Expectation



Hardware description languages

Root Cause: Lacking Abstractions

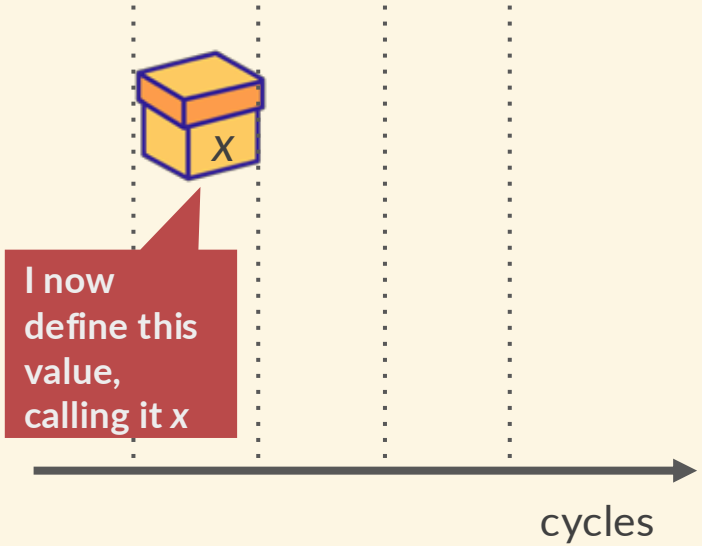
Expectation



Hardware description languages

Root Cause: Lacking Abstractions

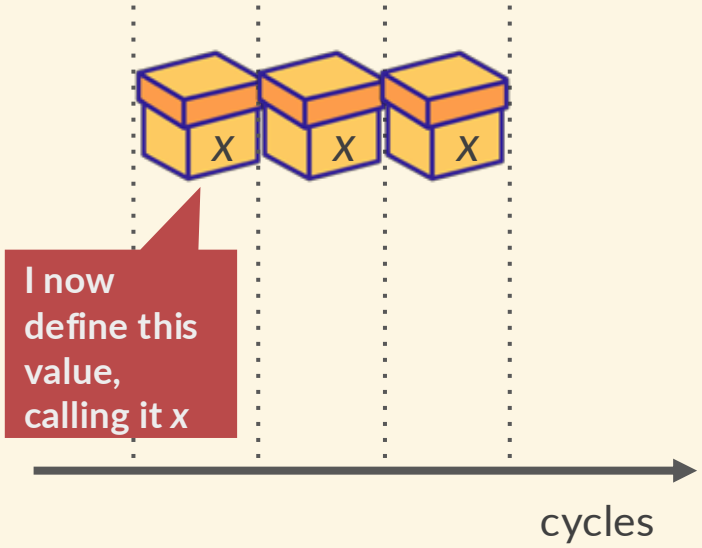
Expectation



Hardware description languages

Root Cause: Lacking Abstractions

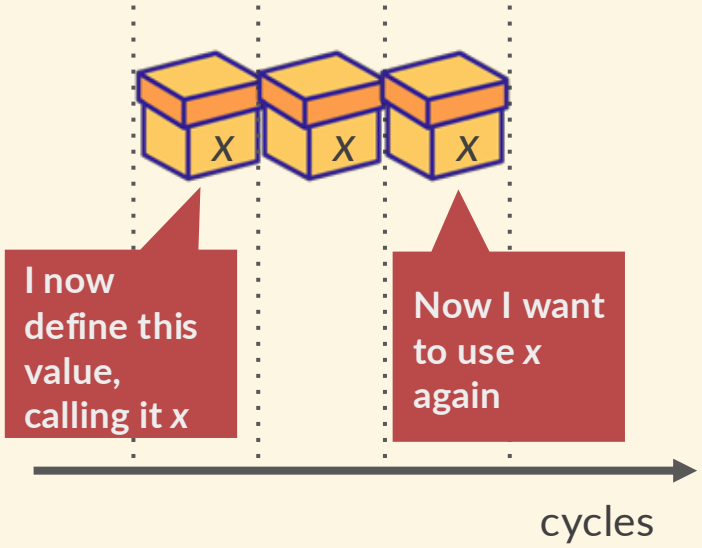
Expectation



Hardware description languages

Root Cause: Lacking Abstractions

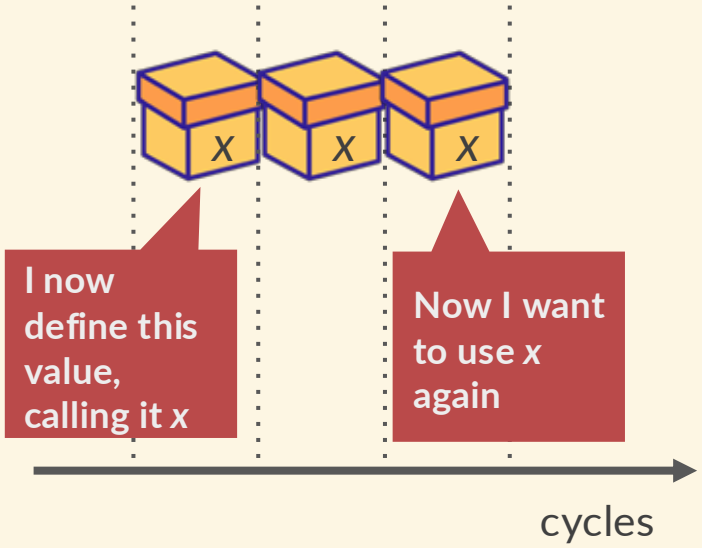
Expectation



Hardware description languages

Root Cause: Lacking Abstractions

Expectation



Hardware description languages

x

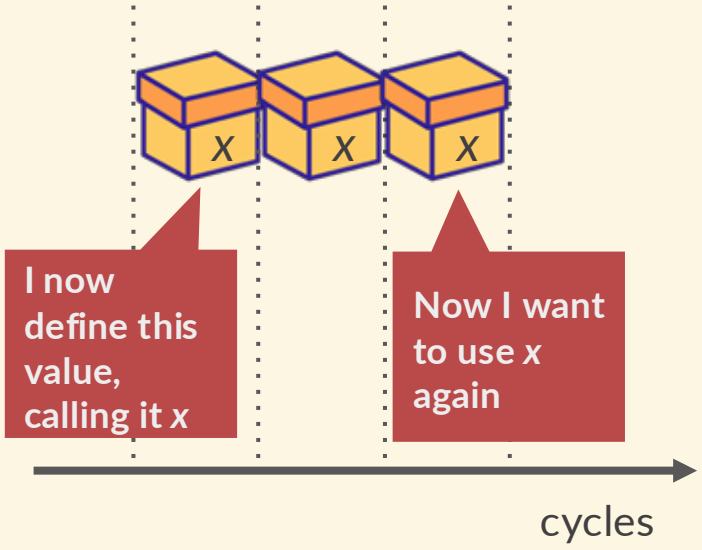
y

z

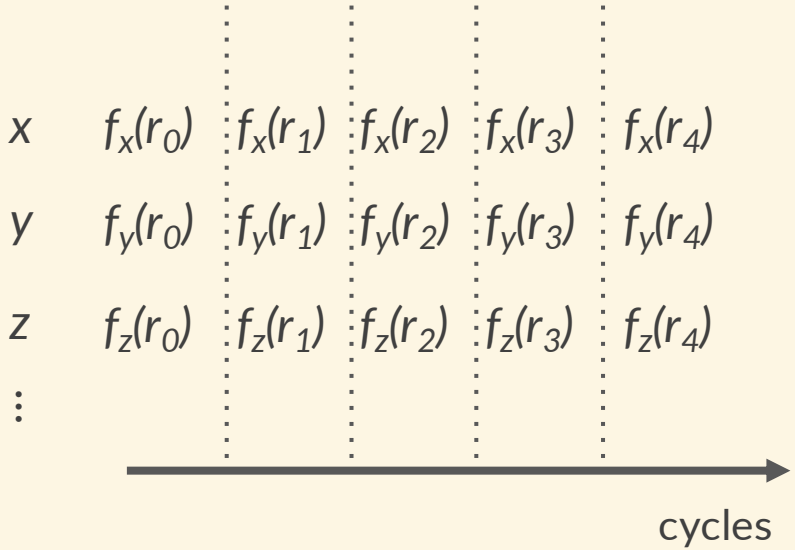
$\vdots$

Root Cause: Lacking Abstractions

Expectation

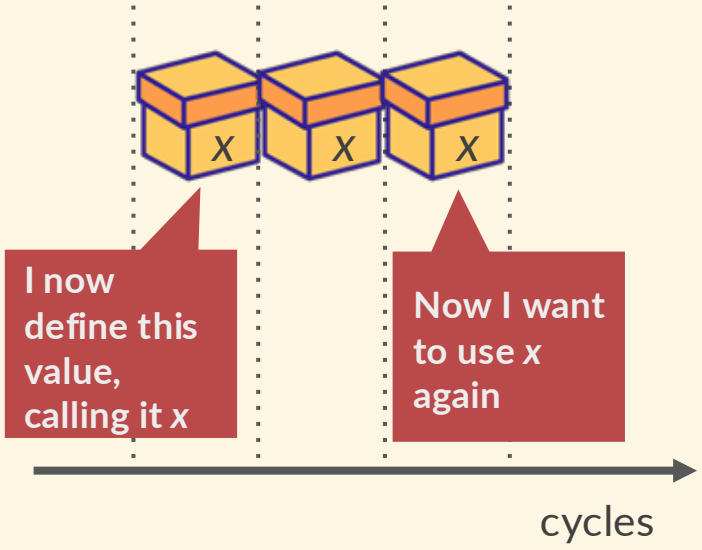


Hardware description languages

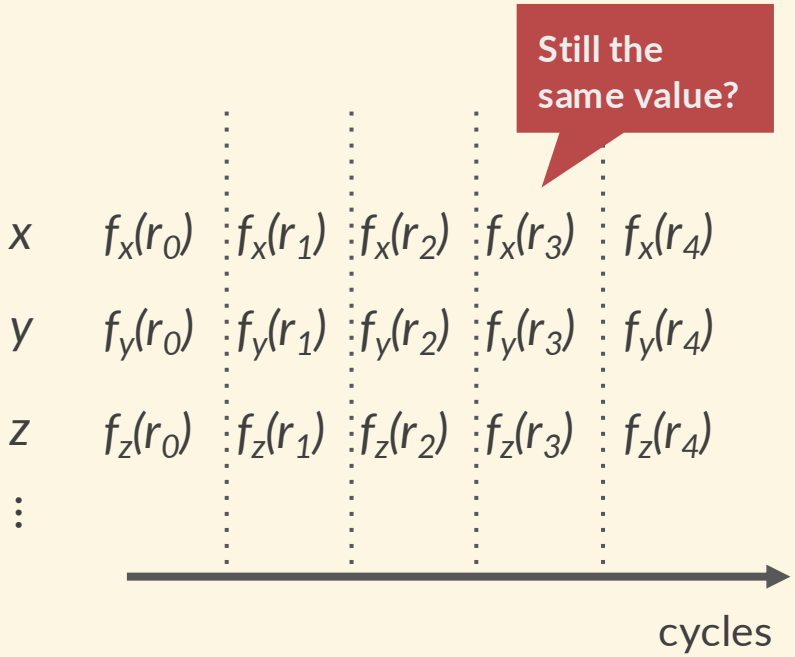


Root Cause: Lacking Abstractions

Expectation

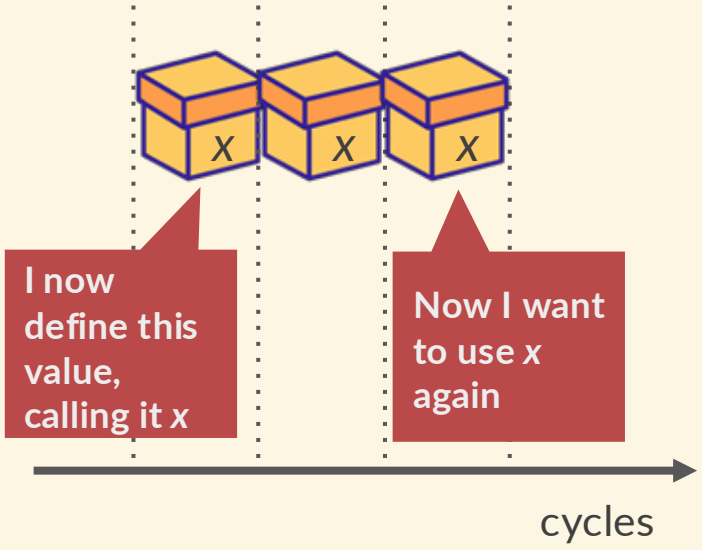


Hardware description languages

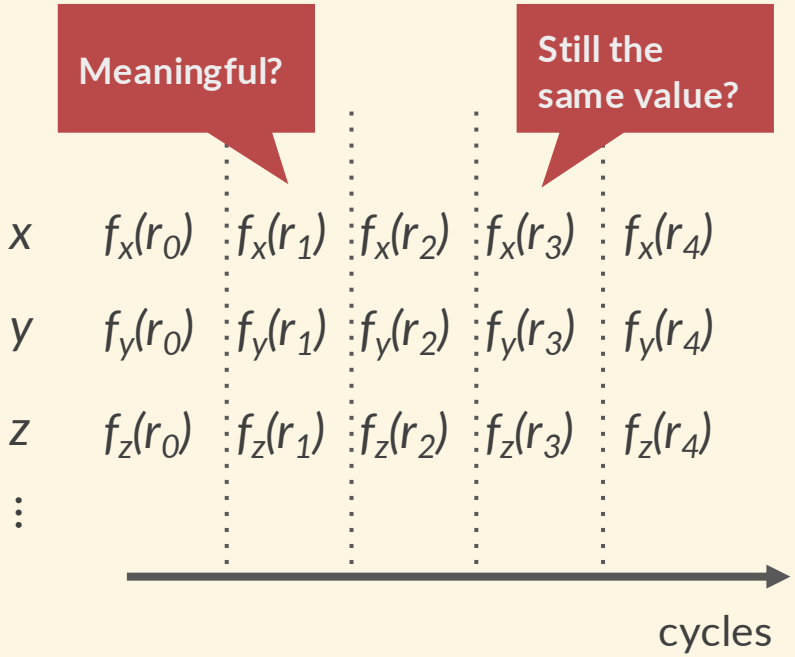


Root Cause: Lacking Abstractions

Expectation

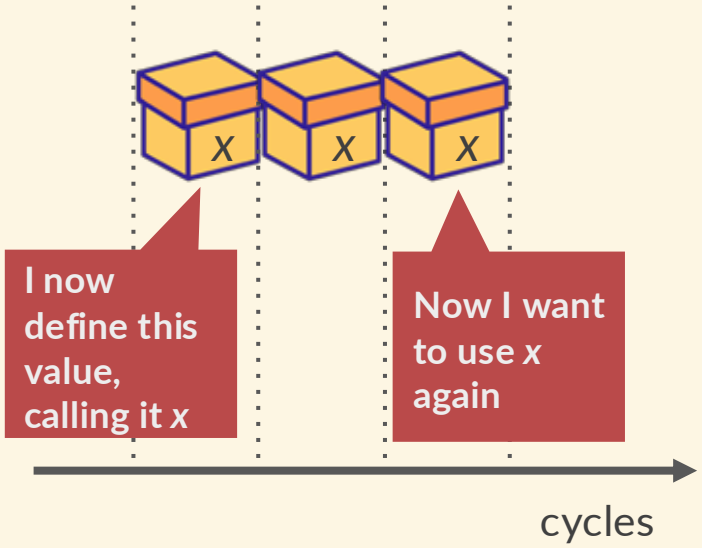


Hardware description languages

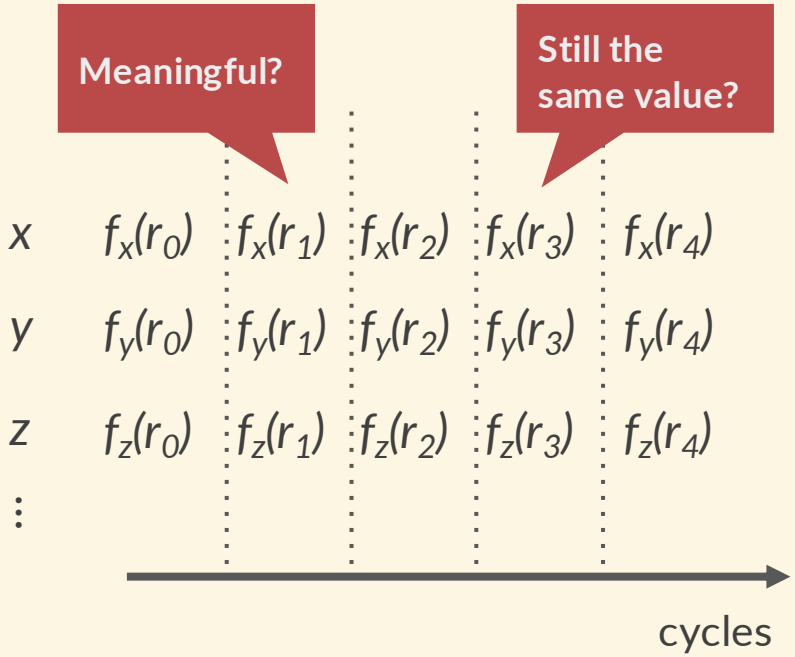


Root Cause: Lacking Abstractions

Expectation

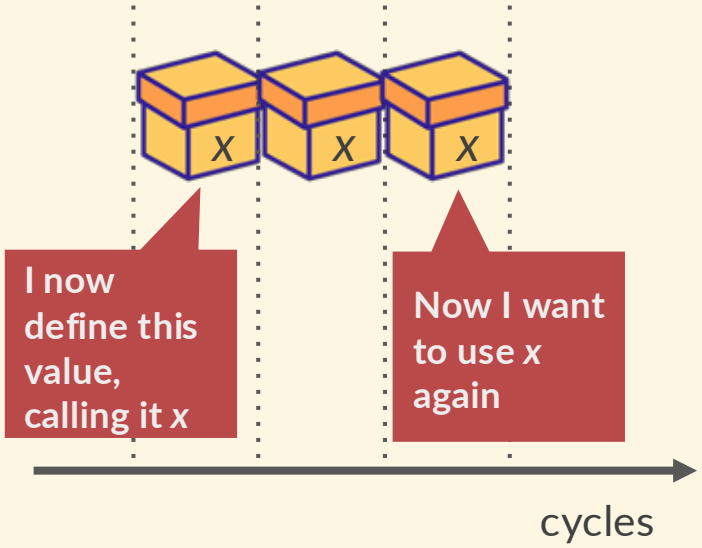


Hardware description languages

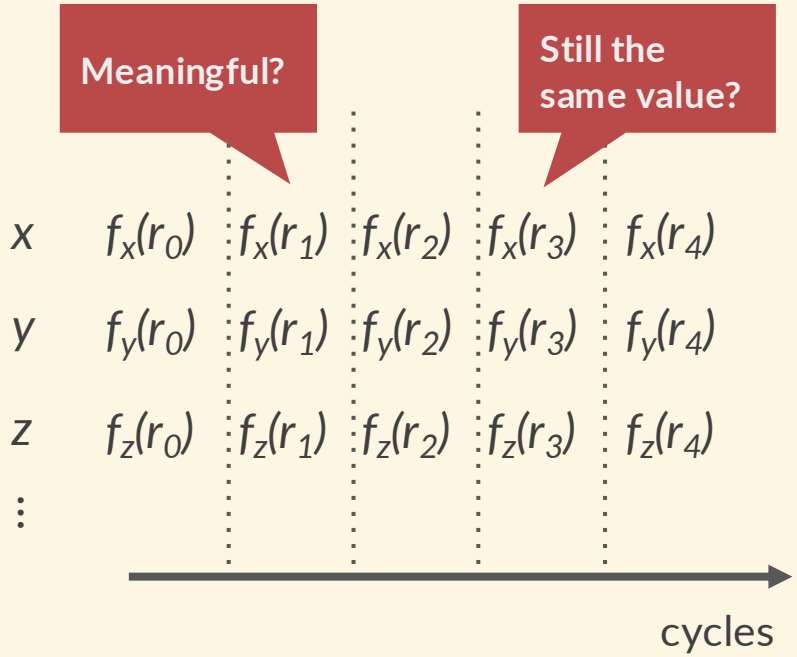


Root Cause: Lacking Abstractions

Expectation



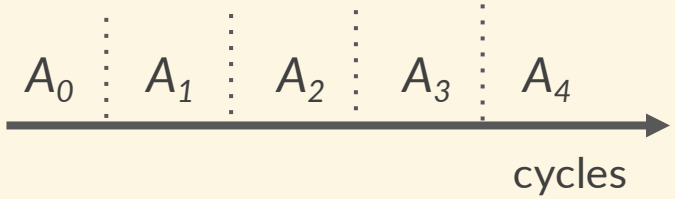
Hardware description languages



What do we want in the abstraction?

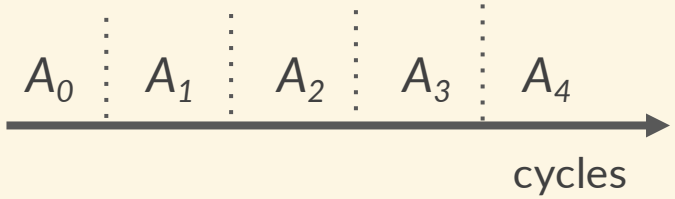
Timing Safety is what we want

Trace:



Timing Safety is what we want

Trace:

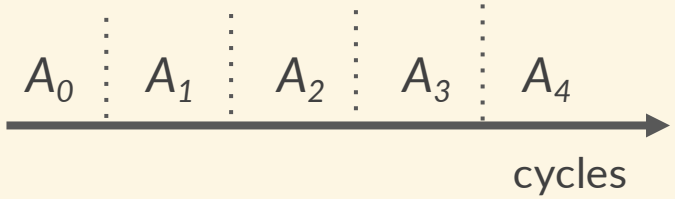


action $\in A_i$

- ValCreate(v, R)
- ValUse(v)
- RegMut(r)

Timing Safety is what we want

Trace:



action $\in A_i$

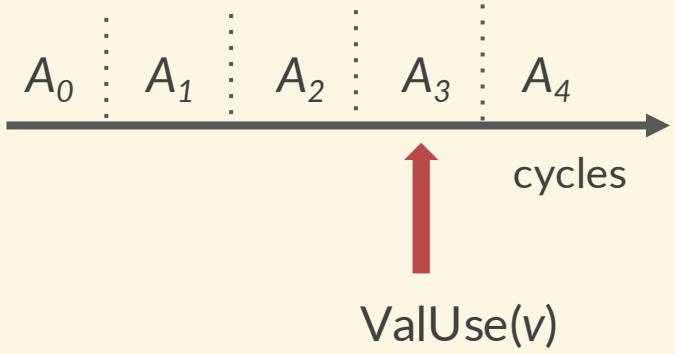
- ValCreate(v, R)
- ValUse(v)
- RegMut(r)

Timing safety:

$$\forall\ v\ i, \text{ValUse}(v) \in A(i) \rightarrow$$
$$\exists\ R\ j, j \leq i \wedge \text{ValCreate}(v, R) \wedge \nexists\ k\ r, j \leq k \leq i \wedge r \in R \wedge \text{RegMut}(r) \in R$$

Timing Safety is what we want

Trace:



action $\in A_i$

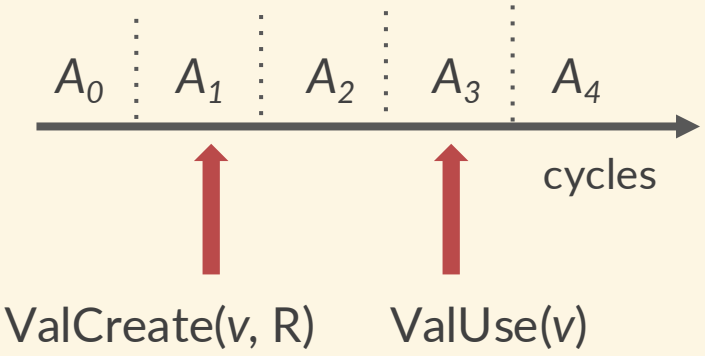
- ValCreate(*v*, R)
- ValUse(*v*)
- RegMut(*r*)

Timing safety:

$$\forall\ v\ i, \text{ValUse}(v) \in A(i) \rightarrow$$
$$\exists\ R\ j, j \leq i \wedge \text{ValCreate}(v, R) \wedge \nexists\ k\ r, j \leq k \leq i \wedge r \in R \wedge \text{RegMut}(r) \in R$$

Timing Safety is what we want

Trace:



action $\in A_i$

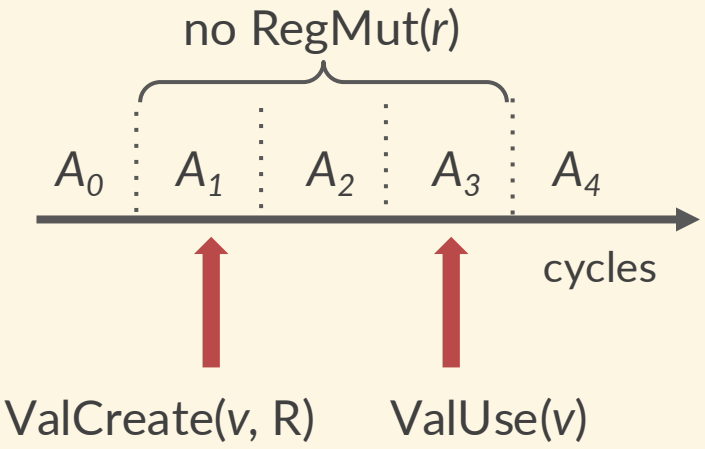
- ValCreate(*v*, R)
- ValUse(*v*)
- RegMut(*r*)

Timing safety:

$$\forall\ v\ i, \text{ValUse}(v) \in \text{A}(i) \rightarrow$$
$$\exists\ \text{R}\ j, j \leq i \wedge \text{ValCreate}(v, \text{R}) \wedge \nexists\ k\ r, j \leq k \leq i \wedge r \in \text{R} \wedge \text{RegMut}(r) \in \text{R}$$

Timing Safety is what we want

Trace:



action $\in A_i$

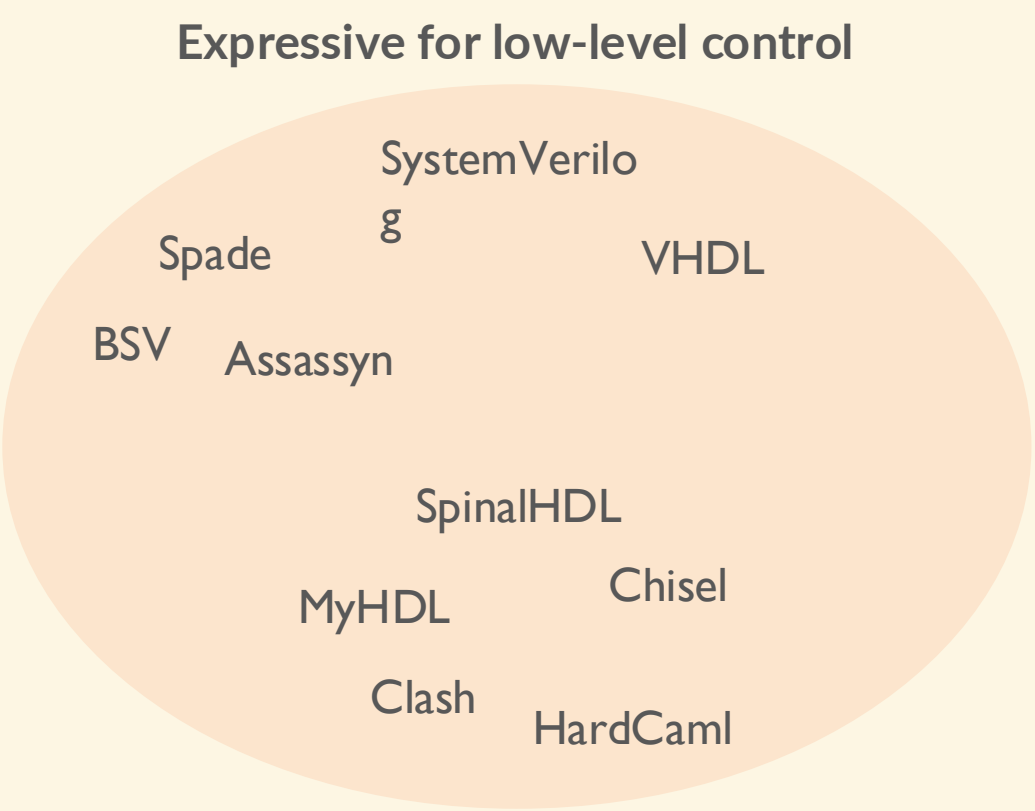
- ValCreate(*v*, R)
- ValUse(*v*)
- RegMut(*r*)

Timing safety:

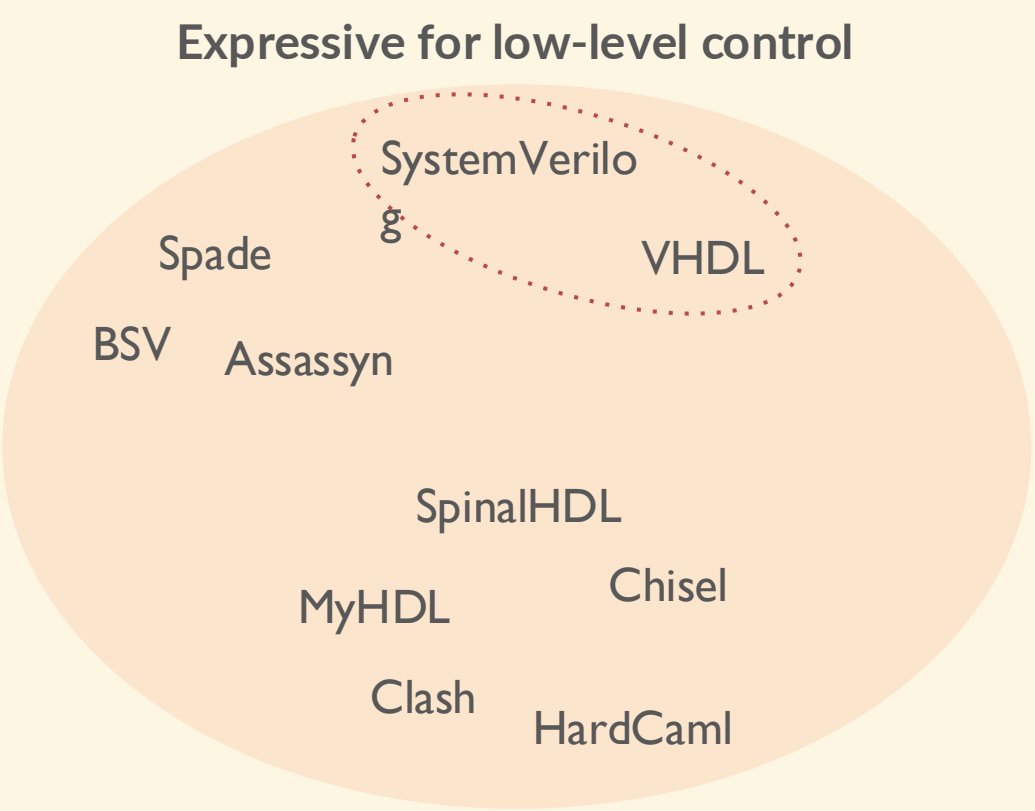
$$\forall\ v\ i, \text{ValUse}(v) \in A(i) \rightarrow$$
$$\exists\ R\ j, j \leq i \wedge \text{ValCreate}(v, R) \wedge \nexists\ k\ r, j \leq k \leq i \wedge r \in R \wedge \text{RegMut}(r) \in R$$

Prior Work: Timing Safety – Expressiveness Dilemma

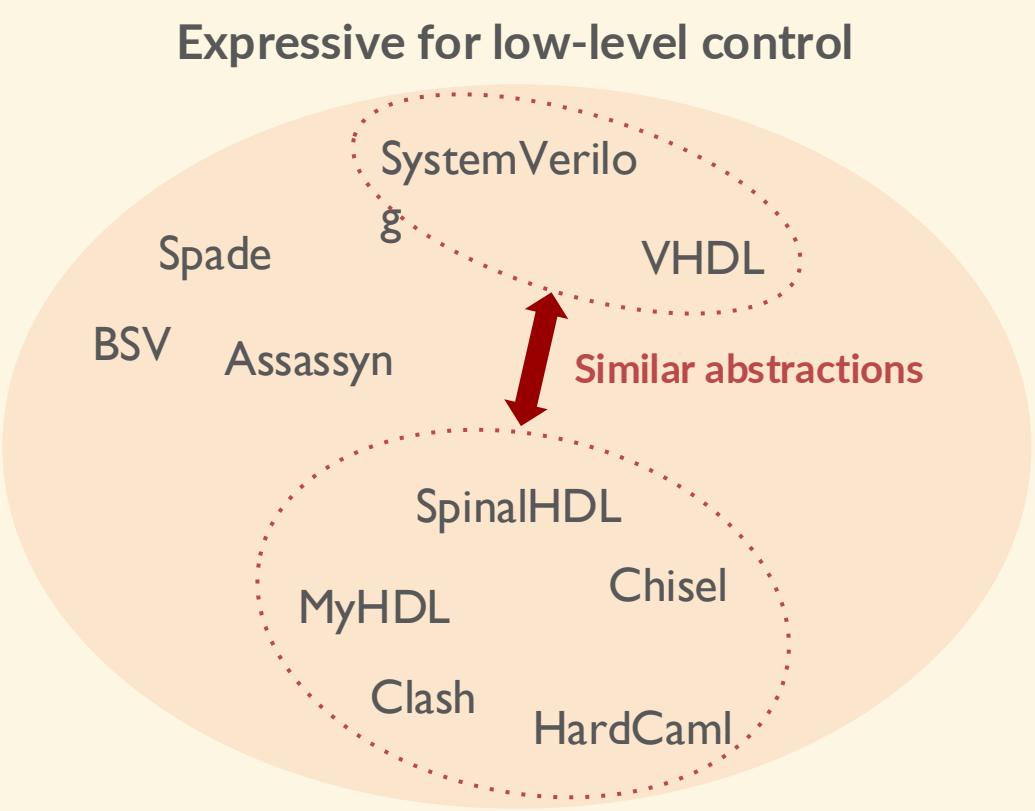
Prior Work: Timing Safety – Expressiveness Dilemma



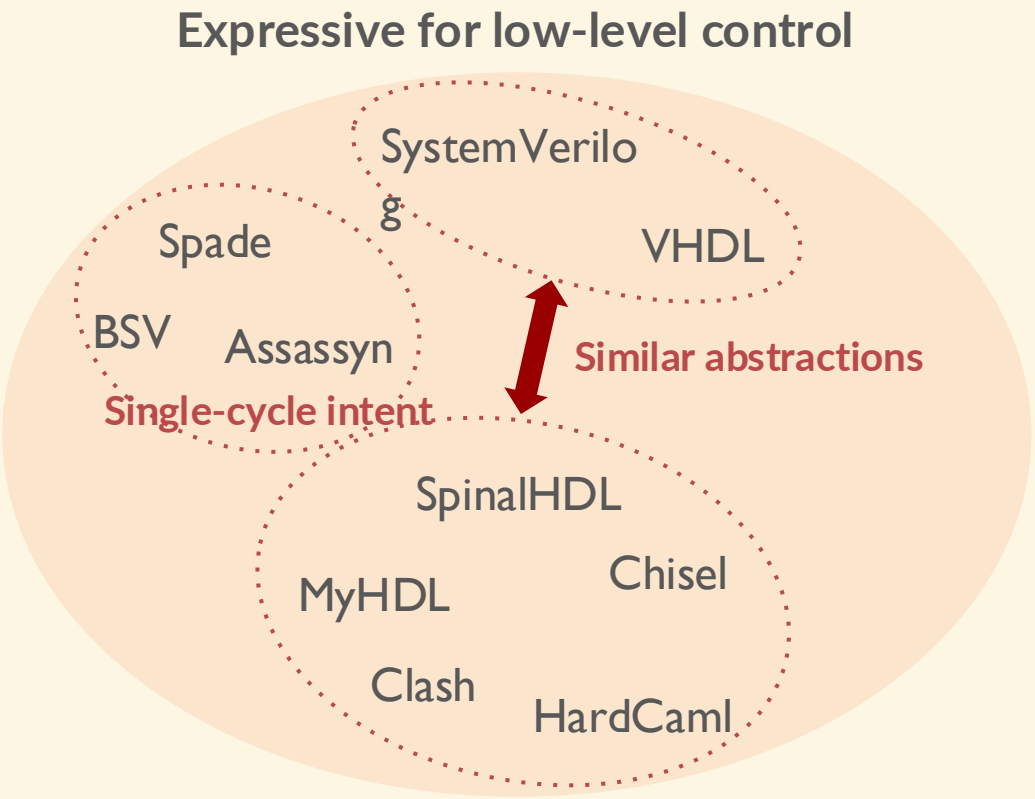
Prior Work: Timing Safety – Expressiveness Dilemma



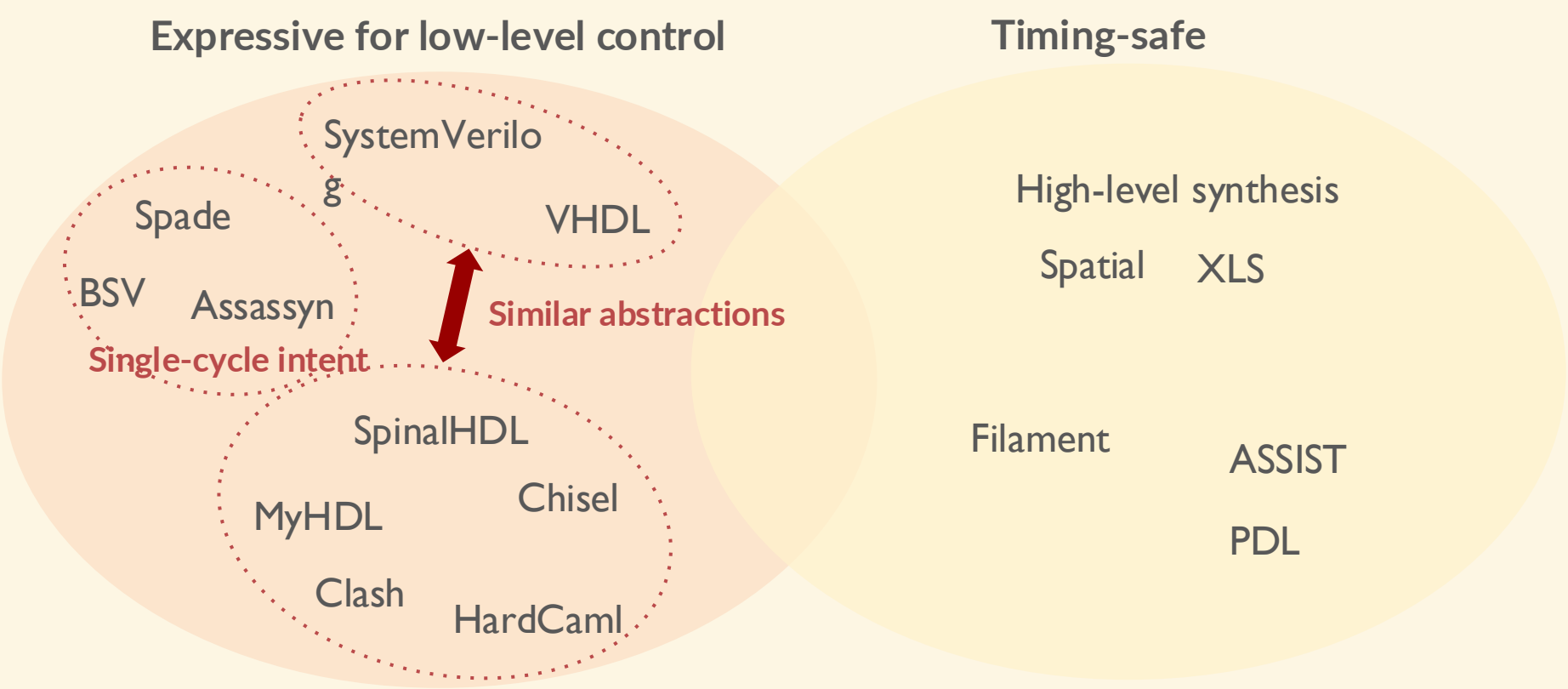
Prior Work: Timing Safety – Expressiveness Dilemma



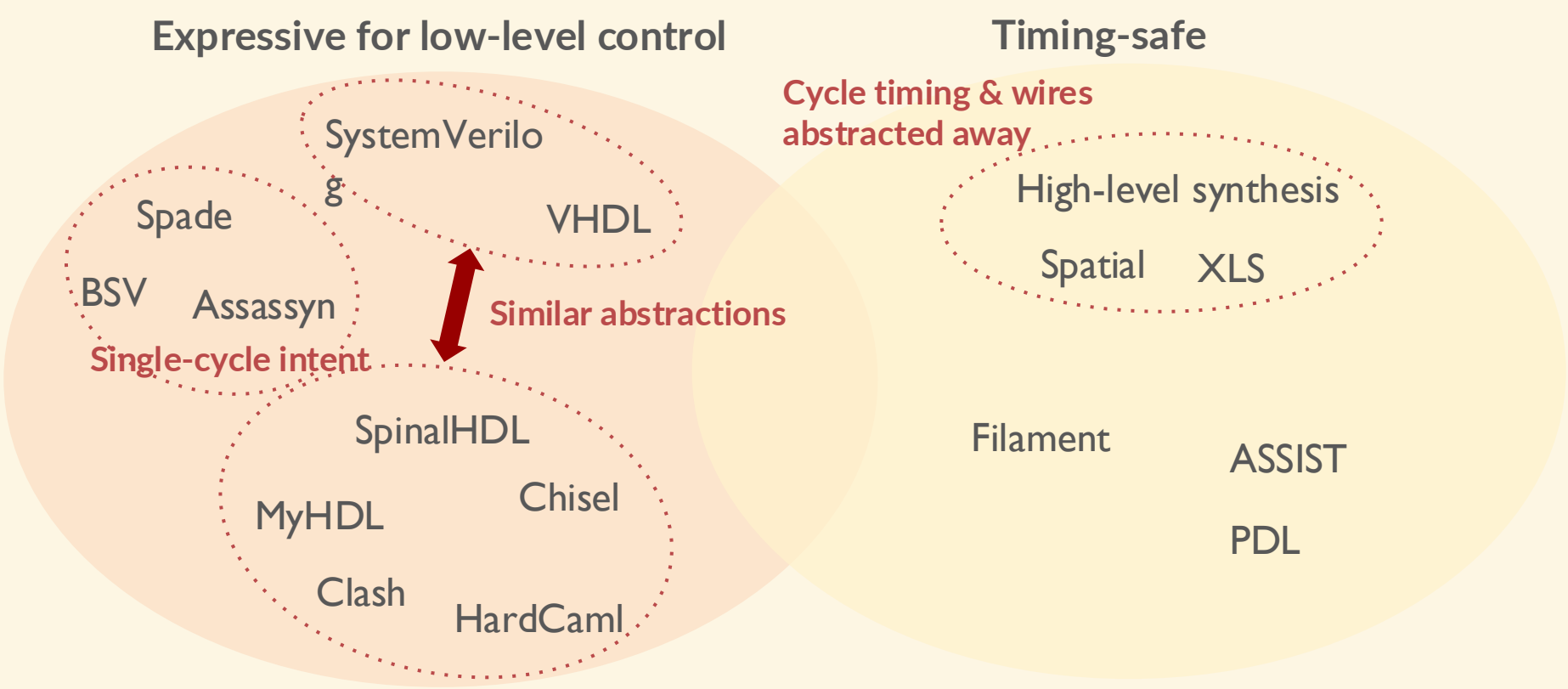
Prior Work: Timing Safety – Expressiveness Dilemma



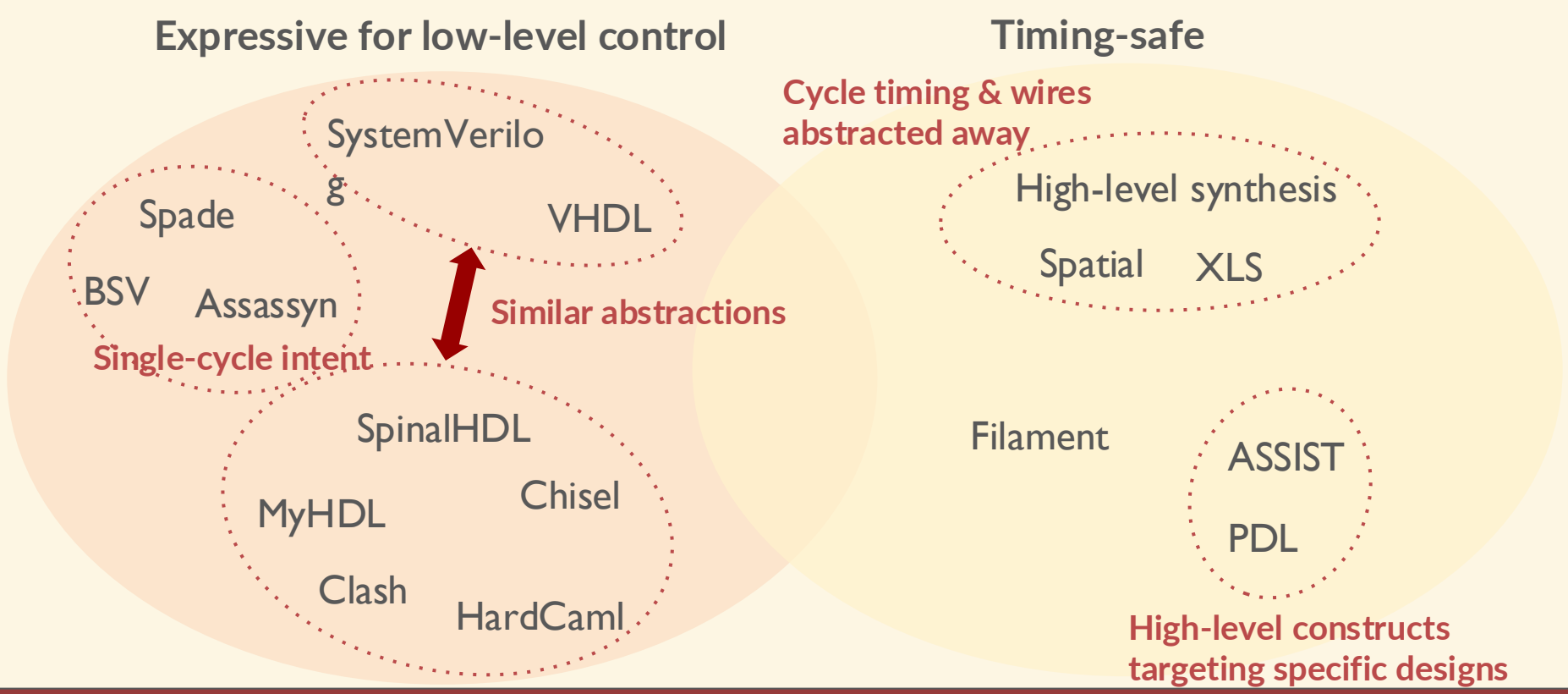
Prior Work: Timing Safety – Expressiveness Dilemma



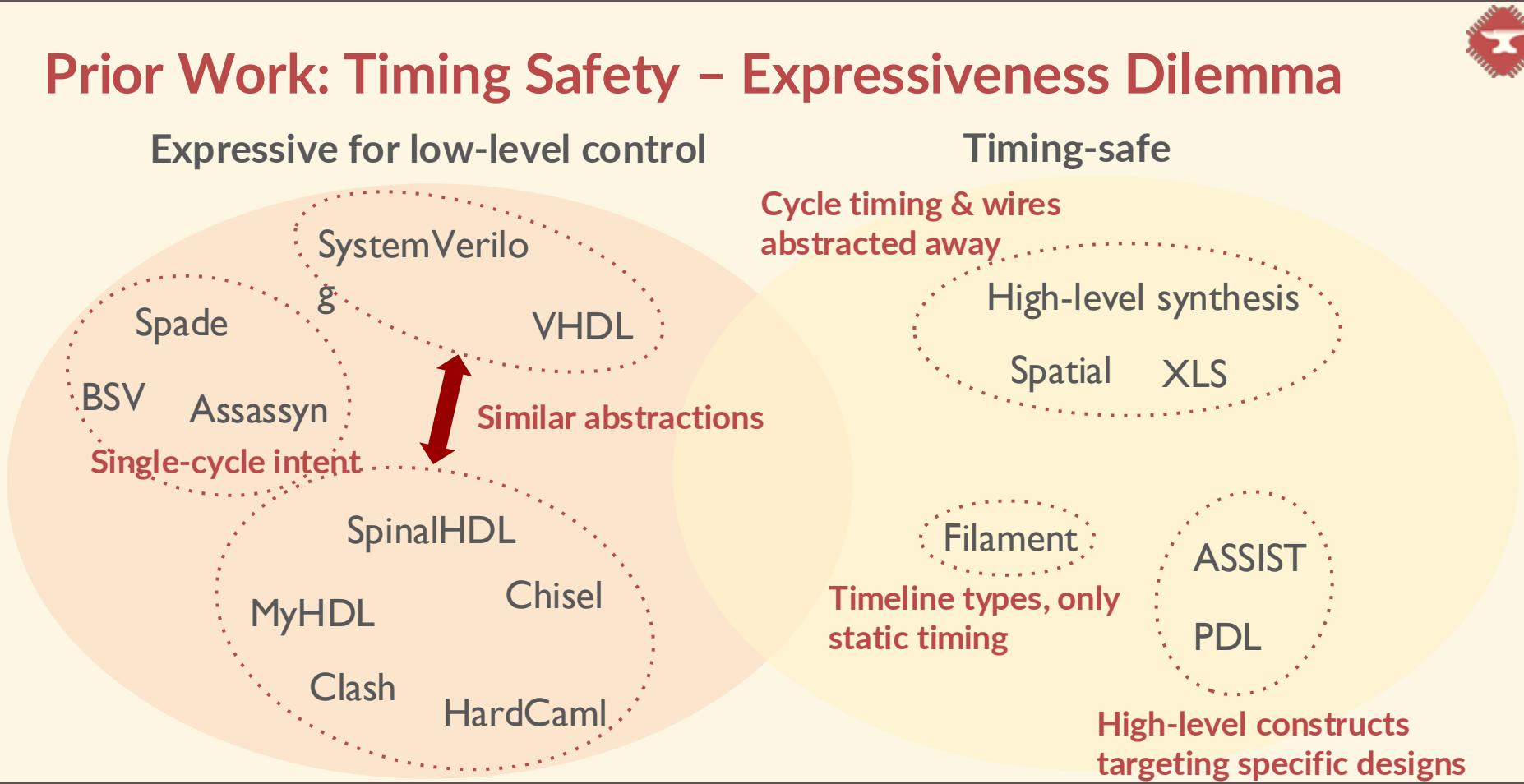
Prior Work: Timing Safety – Expressiveness Dilemma



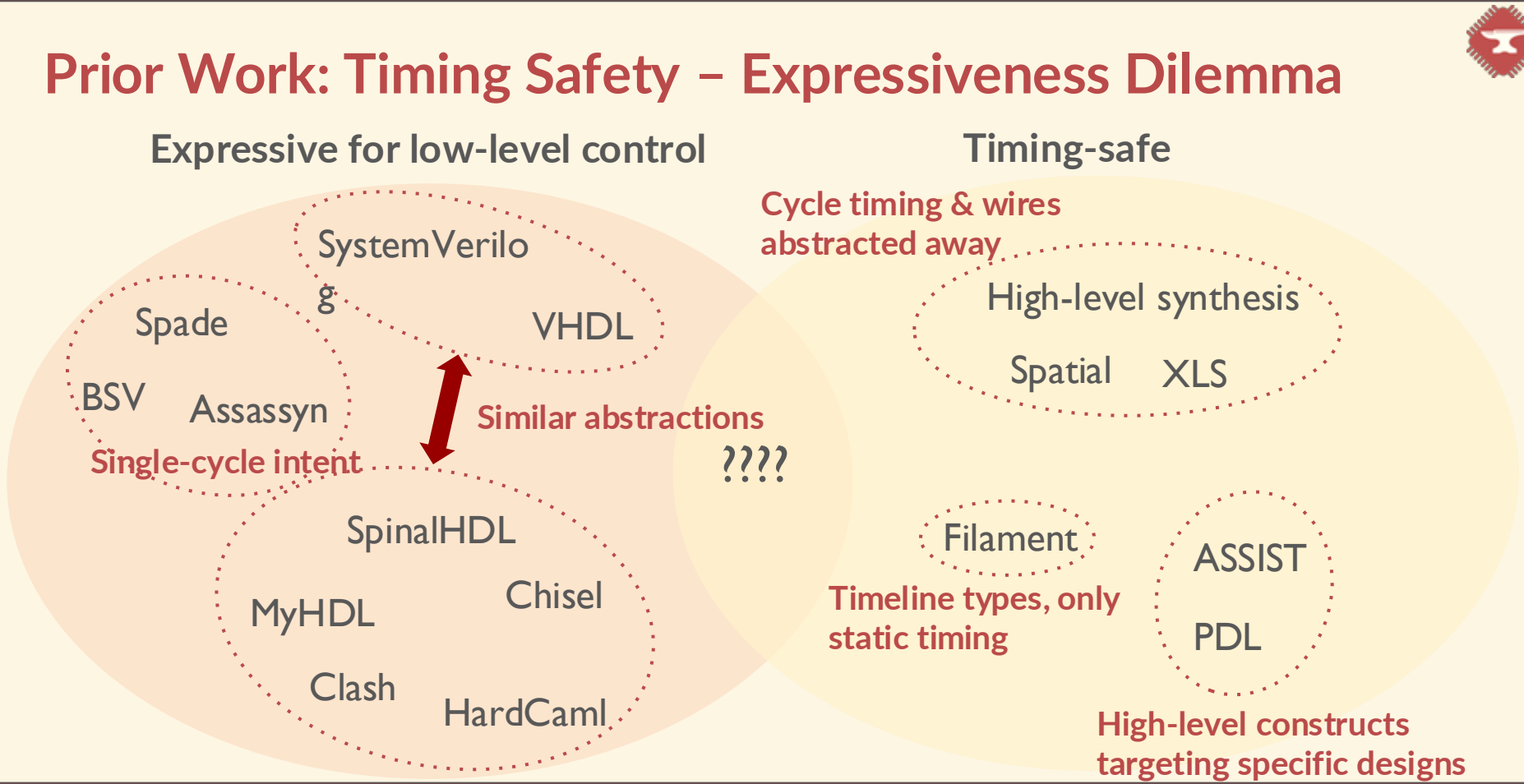
Prior Work: Timing Safety – Expressiveness Dilemma



Prior Work: Timing Safety – Expressiveness Dilemma



Prior Work: Timing Safety – Expressiveness Dilemma



Question: Best of both worlds?

Question: Best of both worlds?

Timing Safety

Meaningful and unchanging values across multiple cycles

Automatically enforced by the compiler during design phase

Question: Best of both worlds?

Timing Safety

Meaningful and unchanging values across multiple cycles

Automatically enforced by the compiler during design phase

Expressiveness

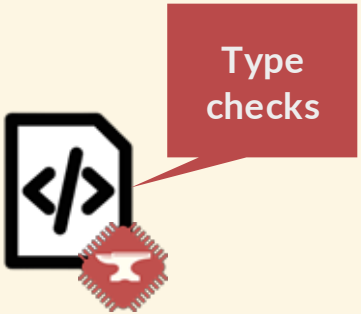
Cycle-level timing control

Dynamic timing

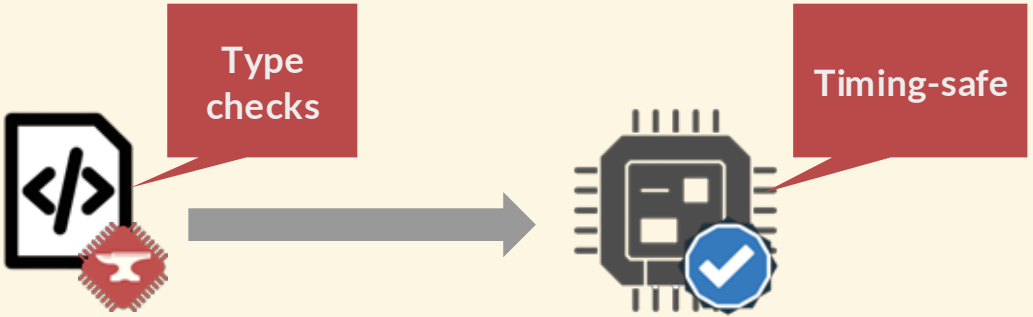
Distinct register/wire abstractions

Anvil: Type System for Timing Safety

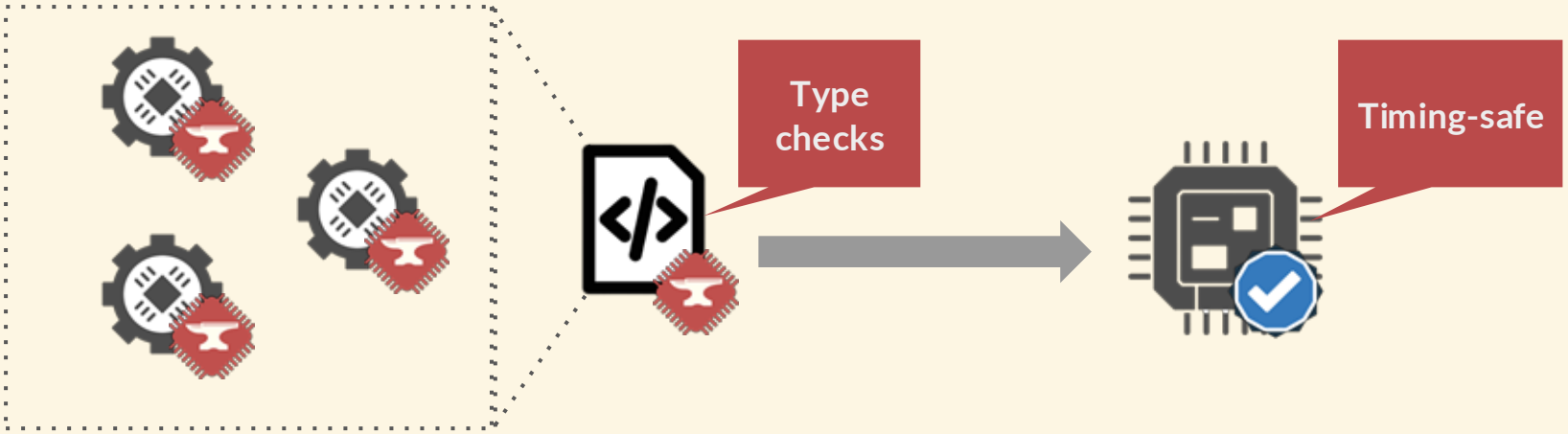
Anvil: Type System for Timing Safety



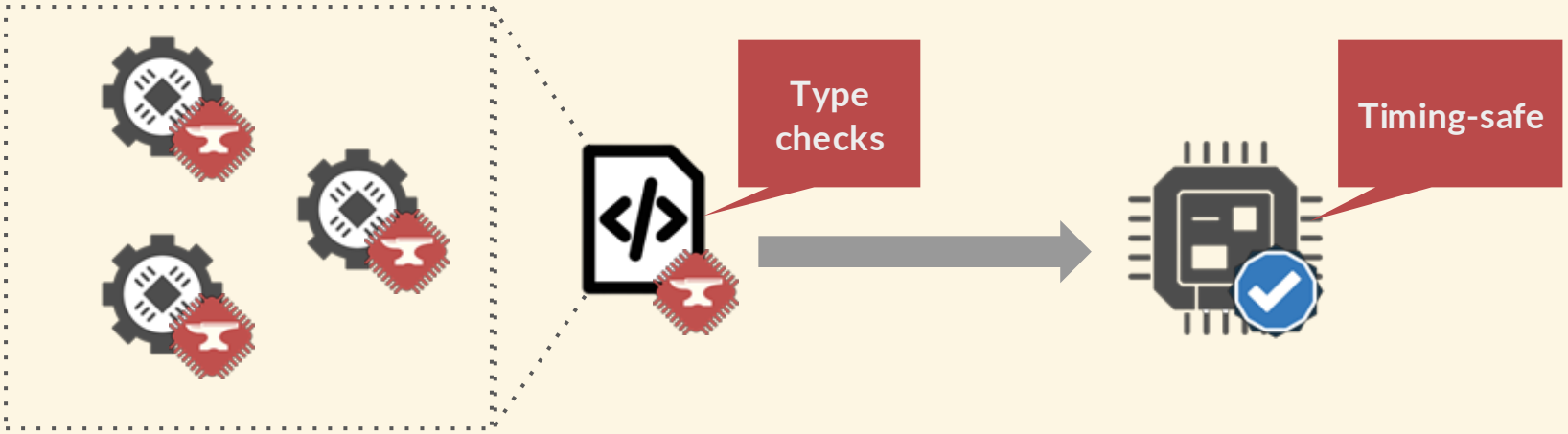
Anvil: Type System for Timing Safety



Anvil: Type System for Timing Safety



Anvil: Type System for Timing Safety



Expressive
Practical:
practical overhead (4.50% area, 3.75% power);
interoperable with other HDLs;
used in our projects

Core Idea behind Anvil: Lifetime and Loan Time

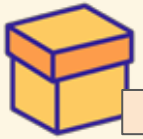
Core Idea behind Anvil: Lifetime and Loan Time

Values 

Registers 

Core Idea behind Anvil: Lifetime and Loan Time

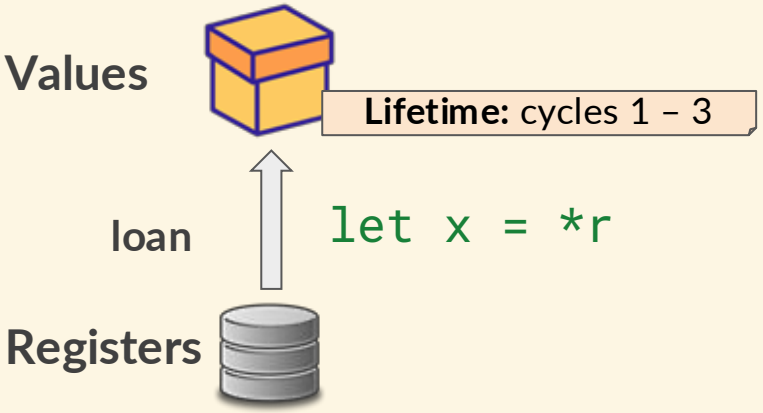
Values



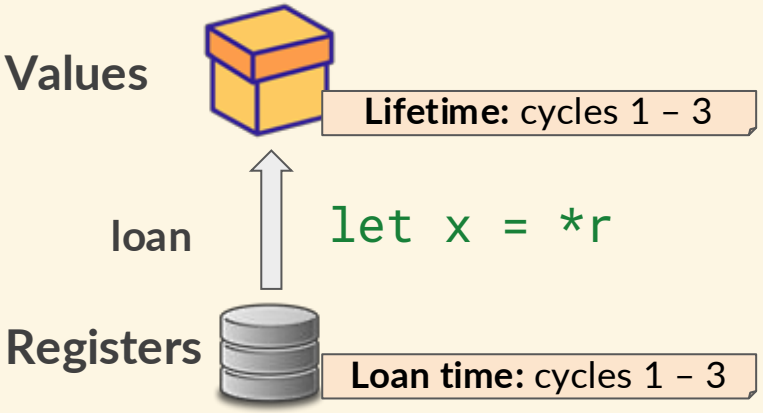
Lifetime: cycles 1 – 3

Registers

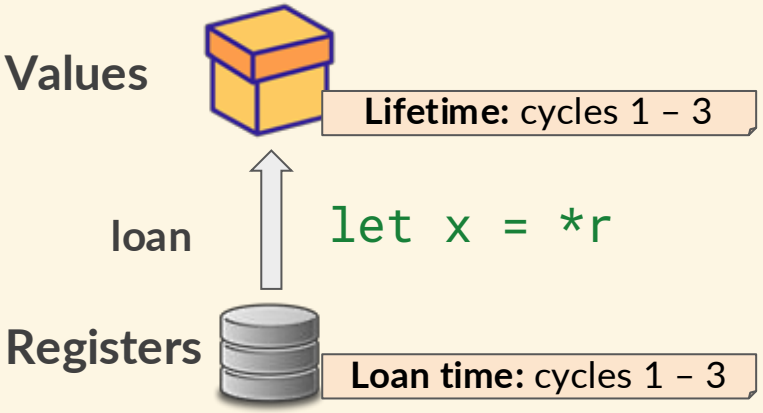
Core Idea behind Anvil: Lifetime and Loan Time



Core Idea behind Anvil: Lifetime and Loan Time

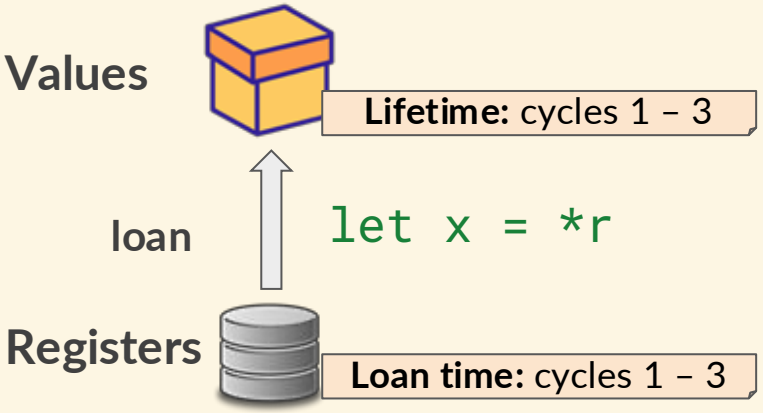


Core Idea behind Anvil: Lifetime and Loan Time



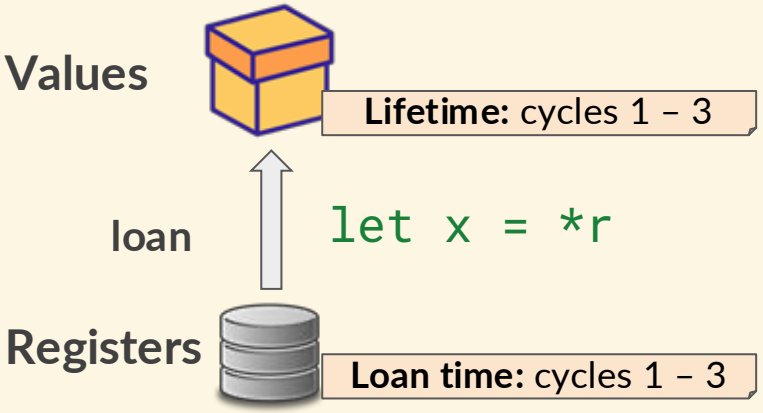
Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

Core Idea behind Anvil: Lifetime and Loan Time



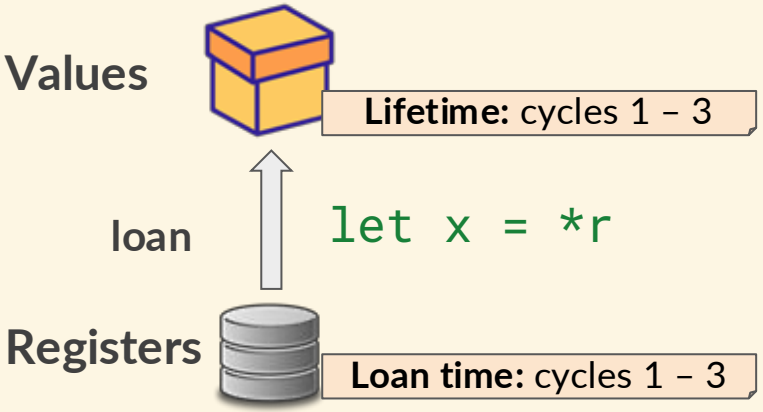
Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

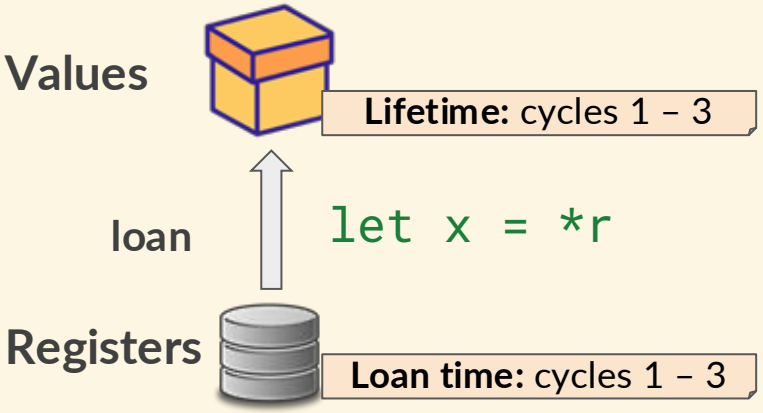
Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

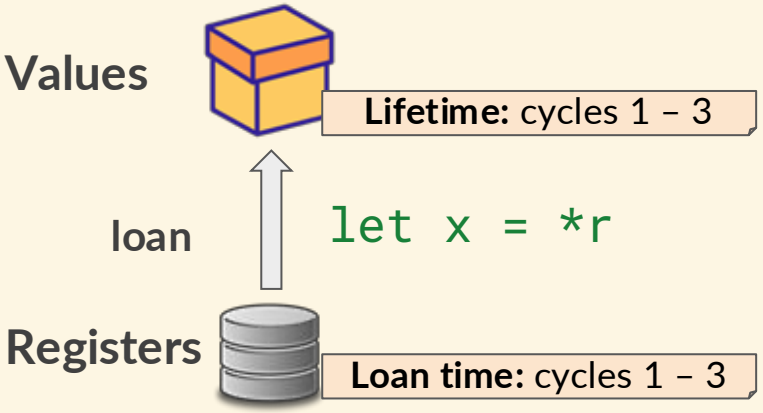
Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

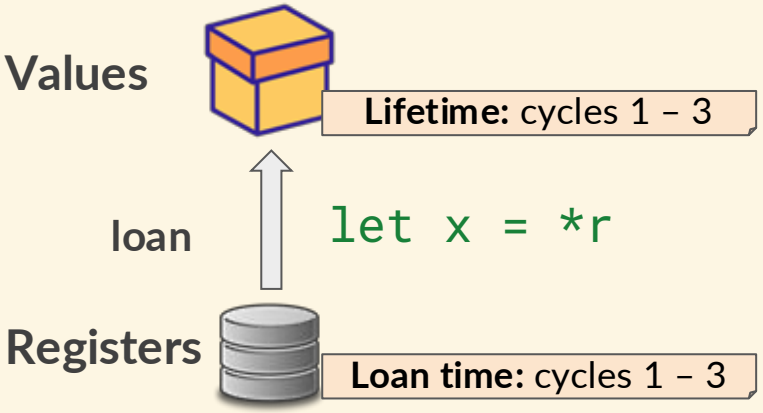
Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

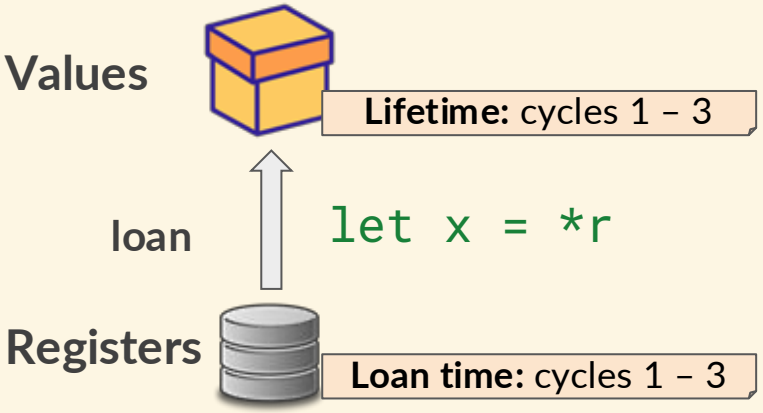
Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

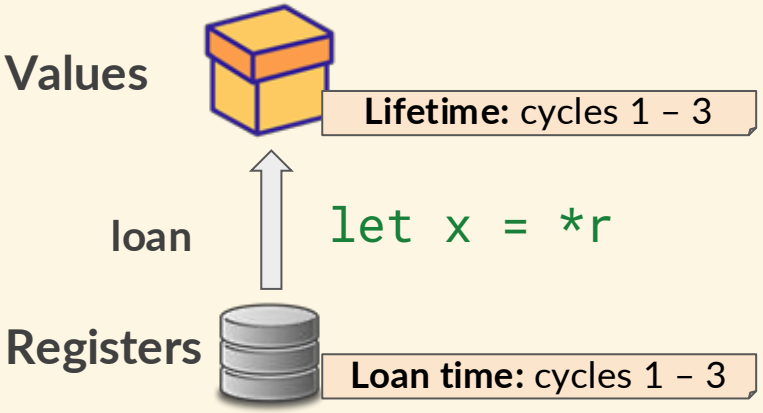
```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

Core Idea behind Anvil: Lifetime and Loan Time




```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

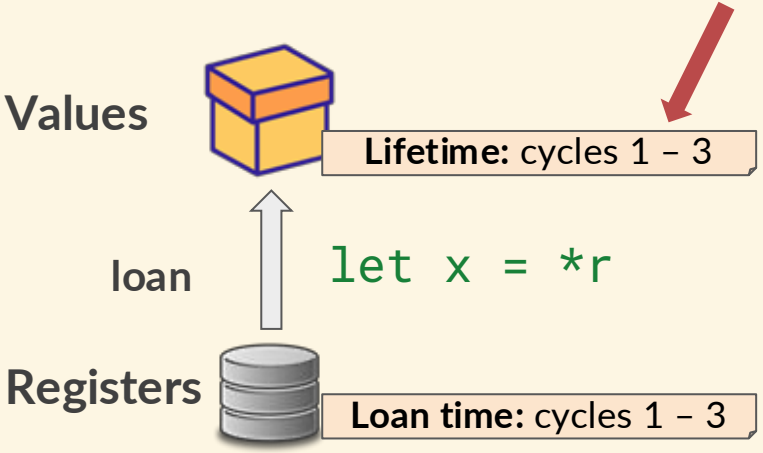
Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

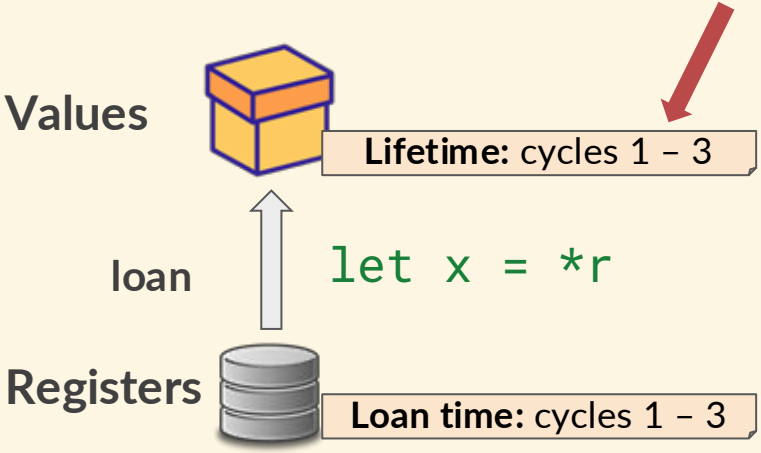
Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

Core Idea behind Anvil: Lifetime and Loan Time



Cycle	1	2	3	4
Use x	✓	✓	✓	✗
Assign to r	✗	✗	✗	✓

```
set r := 1'b1 >>
let v = *r;
dprint "Value of v is %d" (v);
set r := 1'b0 >>
dprint "Value of v is %d" (v);
dfinish
```

What is "*abstract* time"?

Event Graph: Reasoning about Abstract Time

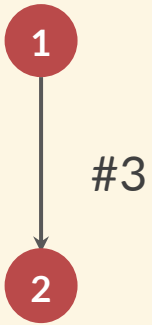
event: abstract clock cycle

define events relative to others

Event Graph: Reasoning about Abstract Time

event: abstract clock cycle

define events relative to others



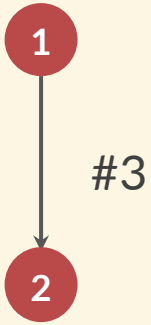
1 2

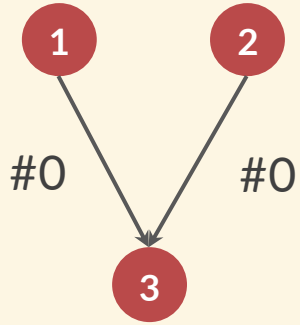
```
cycle 3
```

Event Graph: Reasoning about Abstract Time

event: abstract clock cycle

define events relative to others



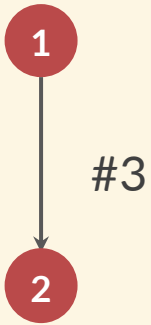


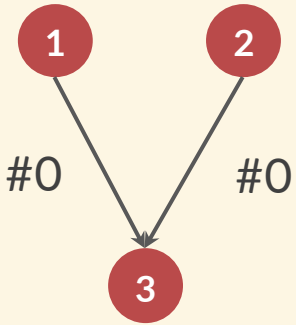

```
1
cycle 3
2
```

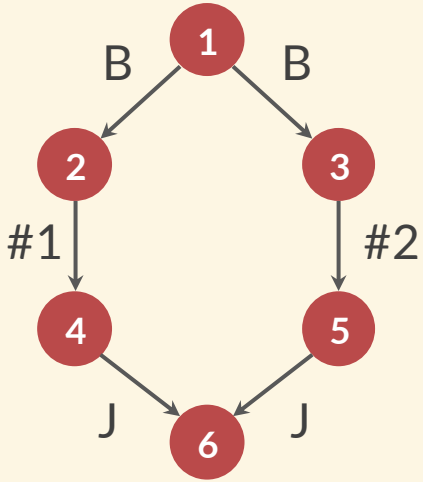
```
1   2
(t1; t2) >> 3
…
```

Event Graph: Reasoning about Abstract Time

event: abstract clock cycle ↓ define events relative to others





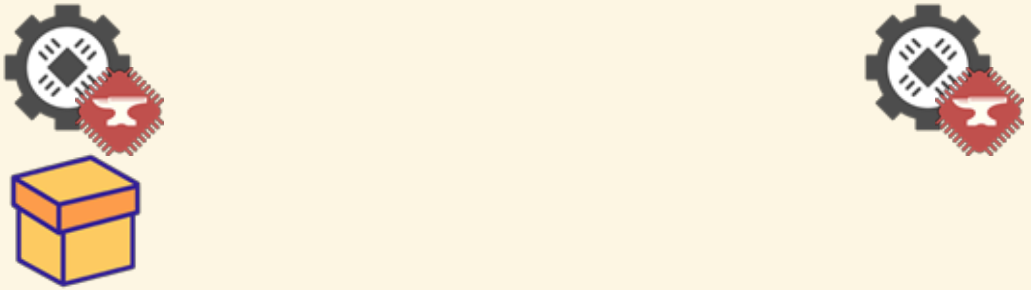



```
1
cycle 3
2
```

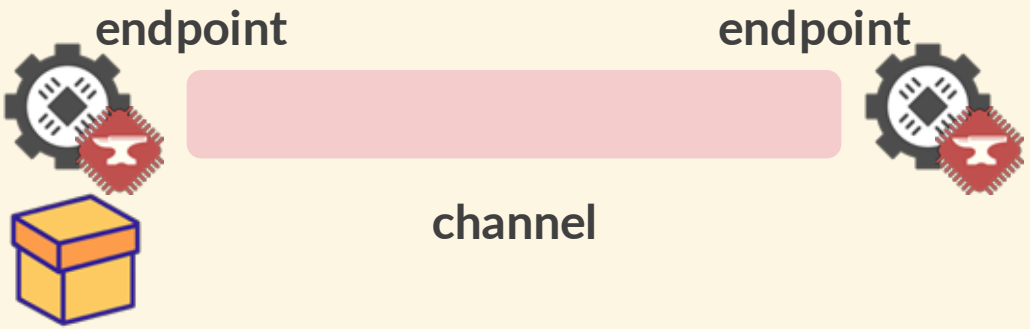
```
1    2
(t1; t2) >> …
3
```

```
1
if x { cycle 1 }
2      4
else { cycle 2 }
3      5  6
```

Synchronous Message Passing Communication

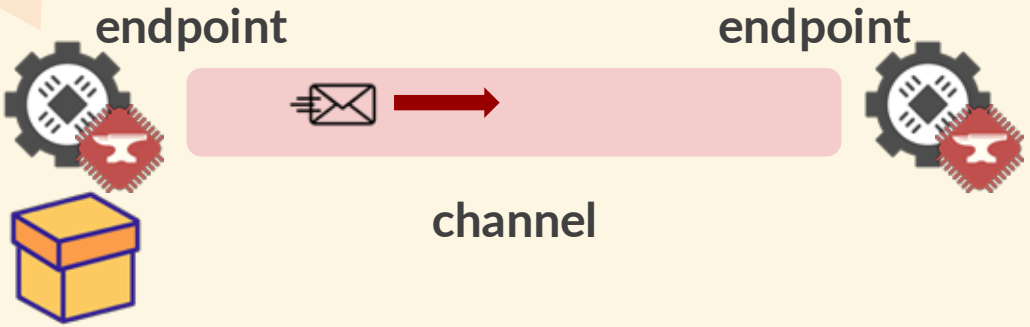


Synchronous Message Passing Communication

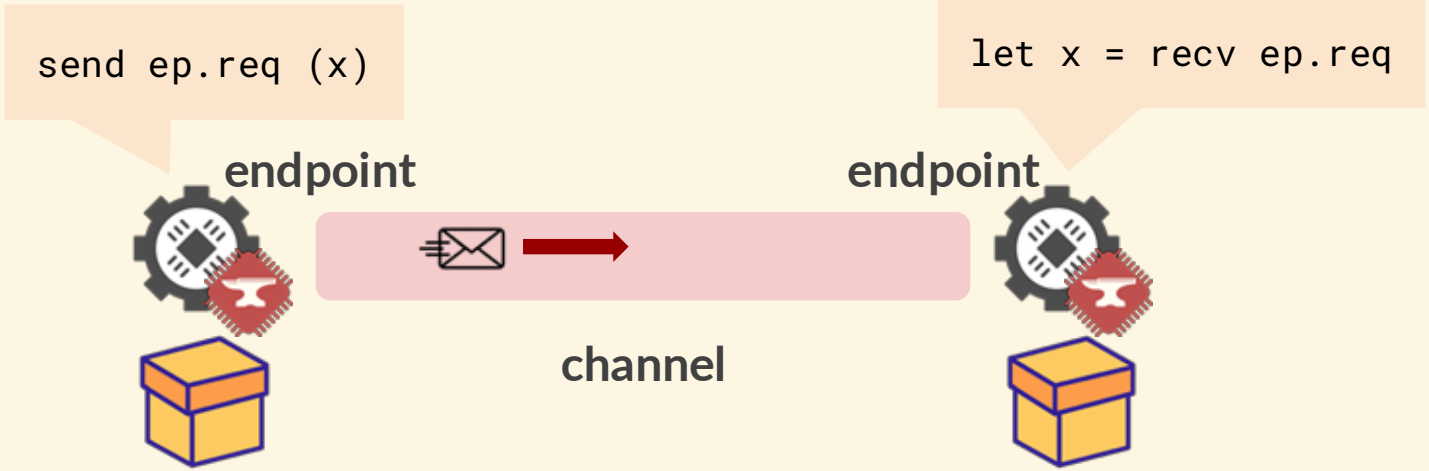


Synchronous Message Passing Communication

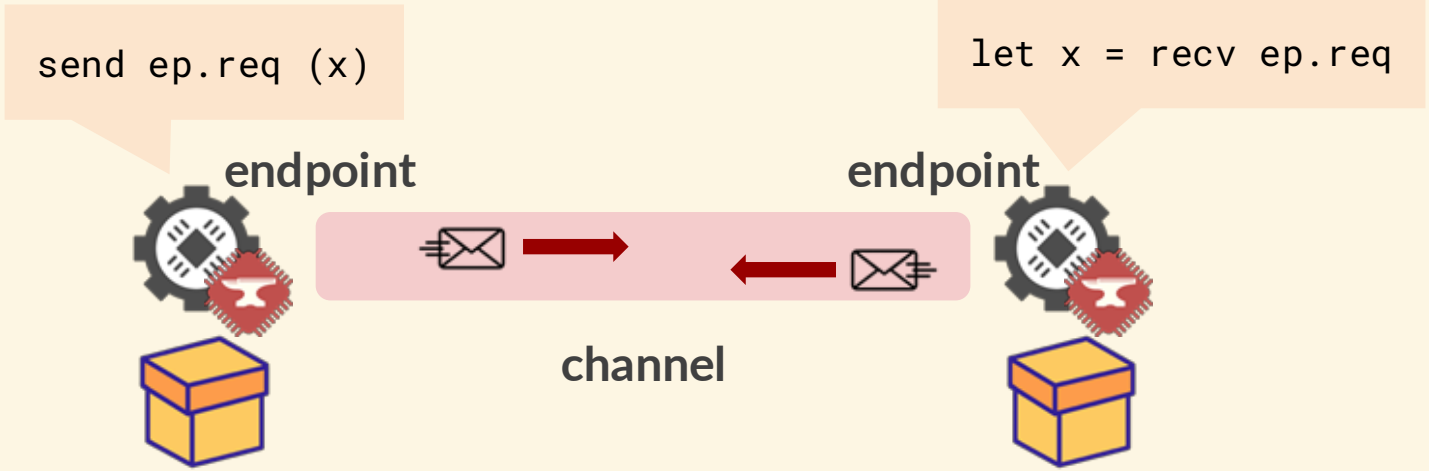
```
send ep.req (x)
```



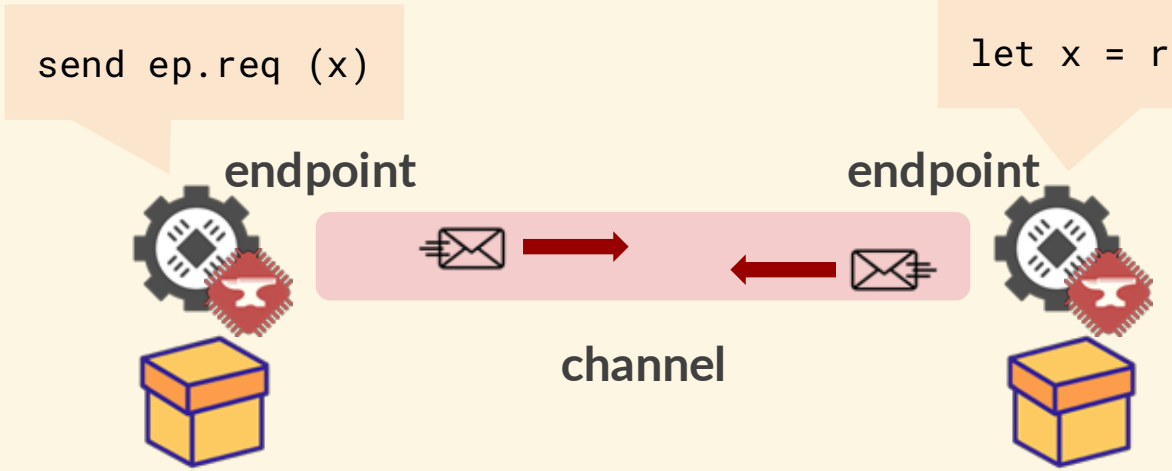
Synchronous Message Passing Communication

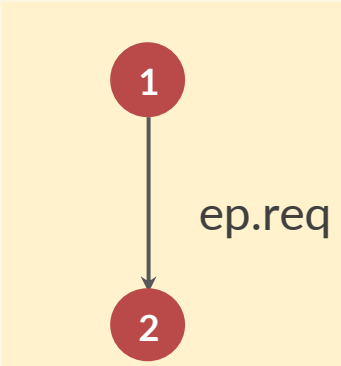


Synchronous Message Passing Communication



Synchronous Message Passing Communication

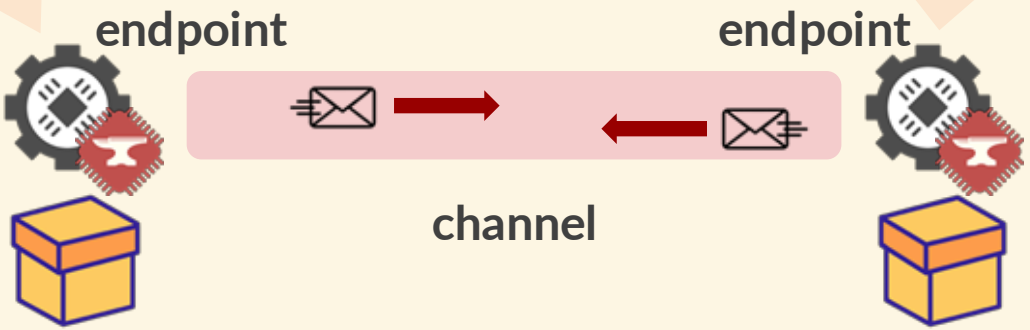


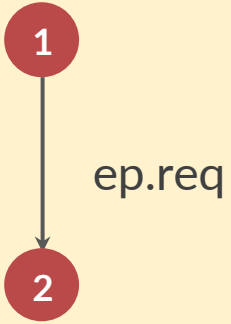


Synchronous Message Passing Communication

```
send ep.req (x)
```

```
let x = recv ep.req
```



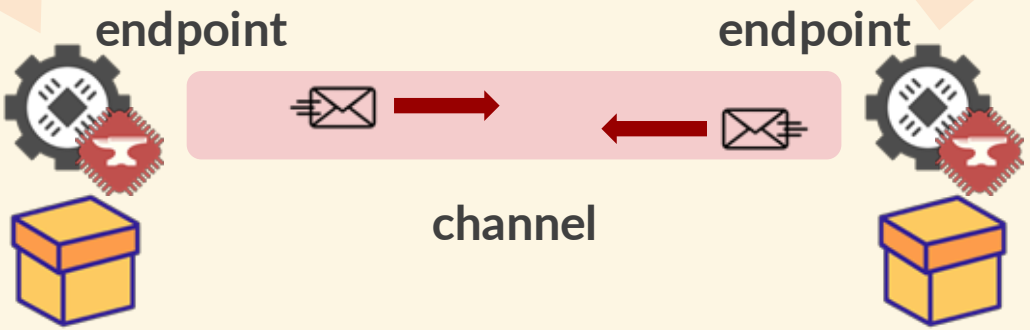


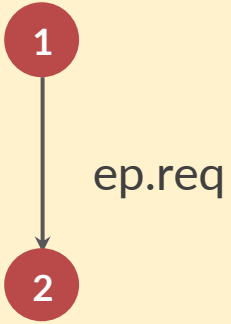
✓ Explicit communication intent

Synchronous Message Passing Communication

```
send ep.req (x)
```

```
let x = recv ep.req
```





- ✓ Explicit communication intent
- ✓ Synchronisation primitive

Example: Message Passing

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#1)
}
proc Top() {
    chan ep_l -- ep_r : ch;
    spawn adder(ep_l);
    loop {
        send ep_r.req(request::{ a = 16'd3425; b = 16'd123 }) >>
        let res = recv ep_r.resp >>
        dprint "result = %d" (res);
        cycle 1 >>
        dfinish
    }
}
```

Example: Message Passing

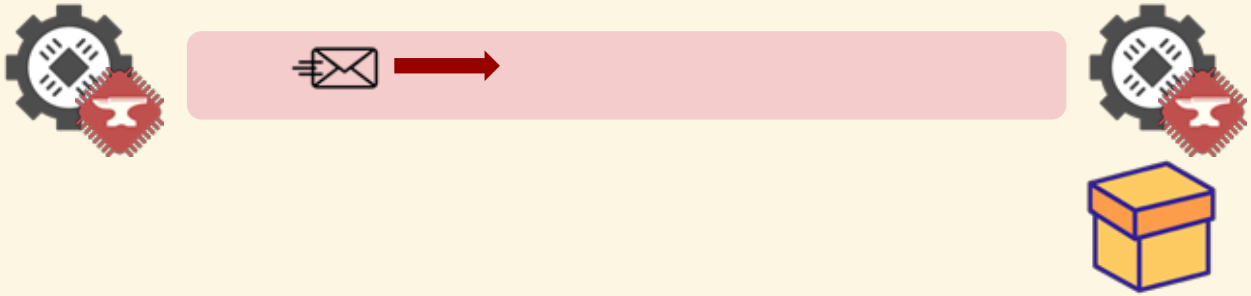
```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#1)
}
proc Top() {
    chan ep_l -- ep_r : ch;
    spawn adder(ep_l);
    loop {
        send ep_r.req(request::{ a = 16'd3425; b = 16'd123 }) >>
        let res = recv ep_r.resp >>
        dprint "result = %d" (res);
        cycle 1 >>
        dfinish
    }
}
```

Example: Message Passing

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#1)
}
proc Top() {
    chan ep_l -- ep_r : ch;
    spawn adder(ep_l);
    loop {
        send ep_r.req(request::{ a = 16'd3425; b = 16'd123 }) >>
        let res = recv ep_r.resp >>
        dprint "result = %d" (res);
        cycle 1 >>
        dfinish
    }
}
```

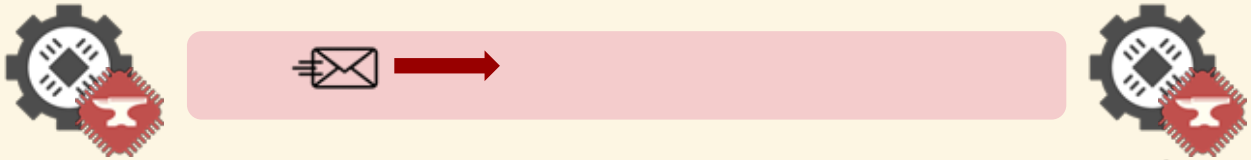
Challenge: Lifetime across Modules

```
let x = recv ep.req
```



Challenge: Lifetime across Modules

```
let x = recv ep.req
```

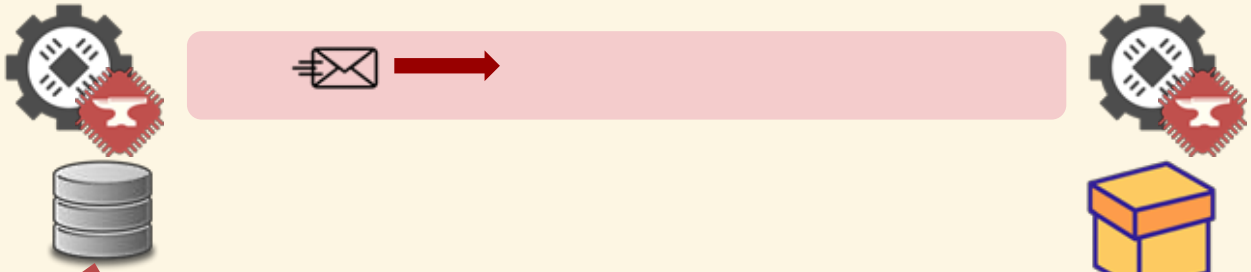


Until when can I use this value?

Challenge: Lifetime across Modules

```
let x = *r;
send ep.req (x)
```

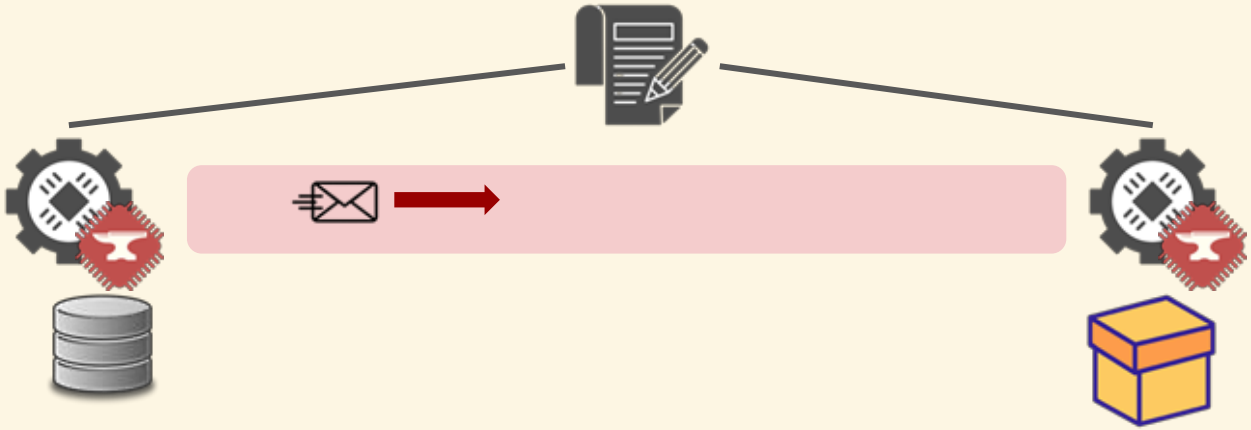
```
let x = recv ep.req
```



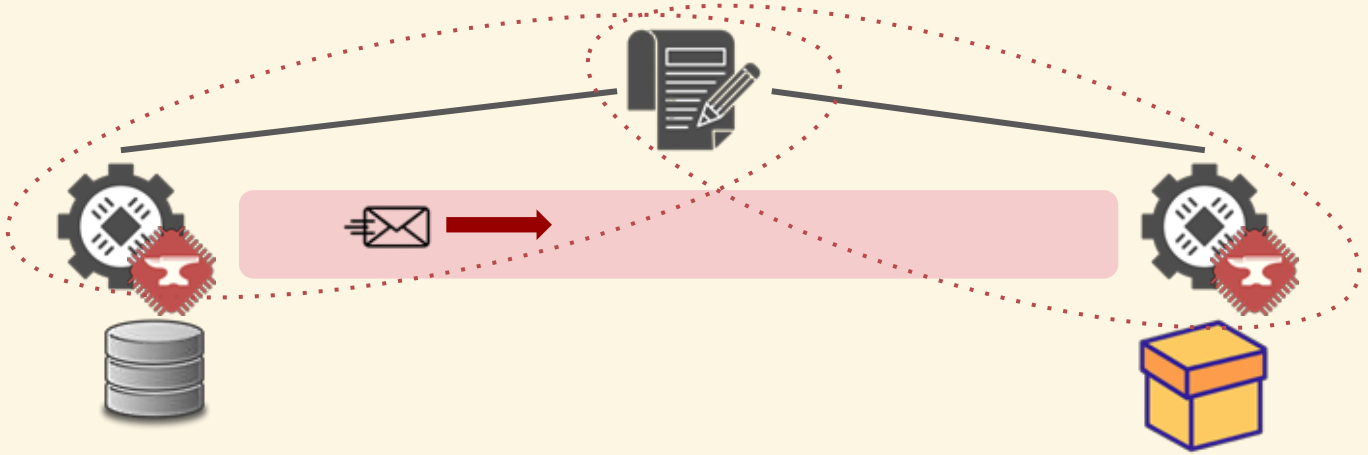
Until when should I avoid changing this register content?

Until when can I use this value?

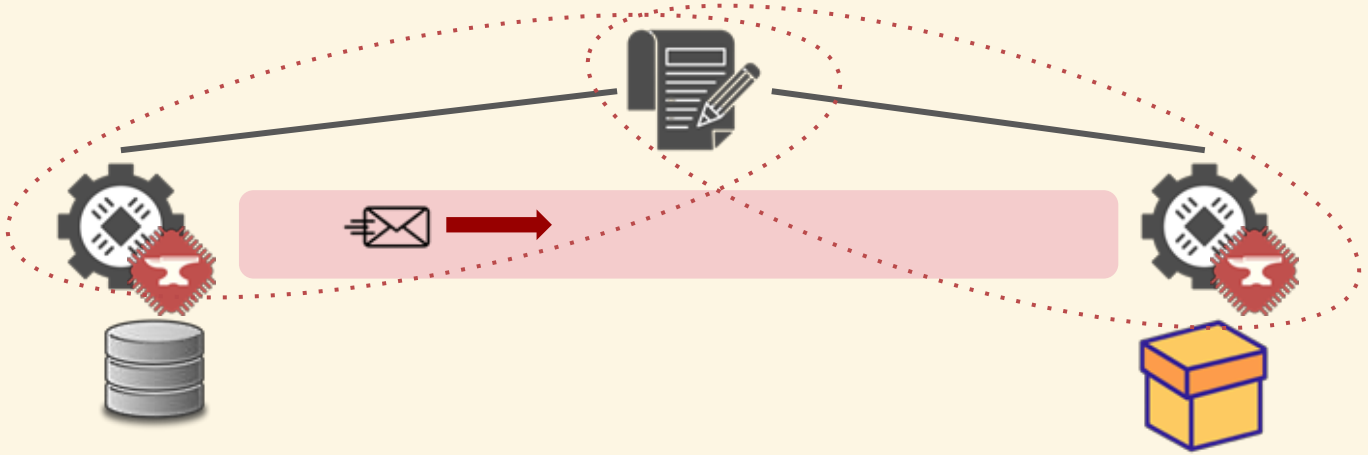
Lifetime Contracts



Lifetime Contracts

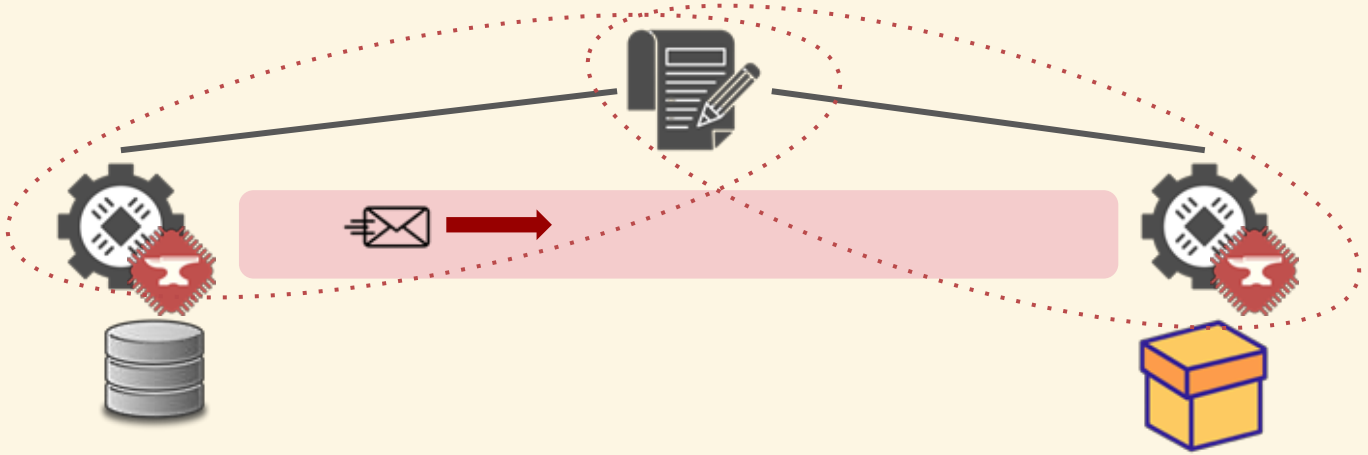


Lifetime Contracts



How to describe time in a contract?

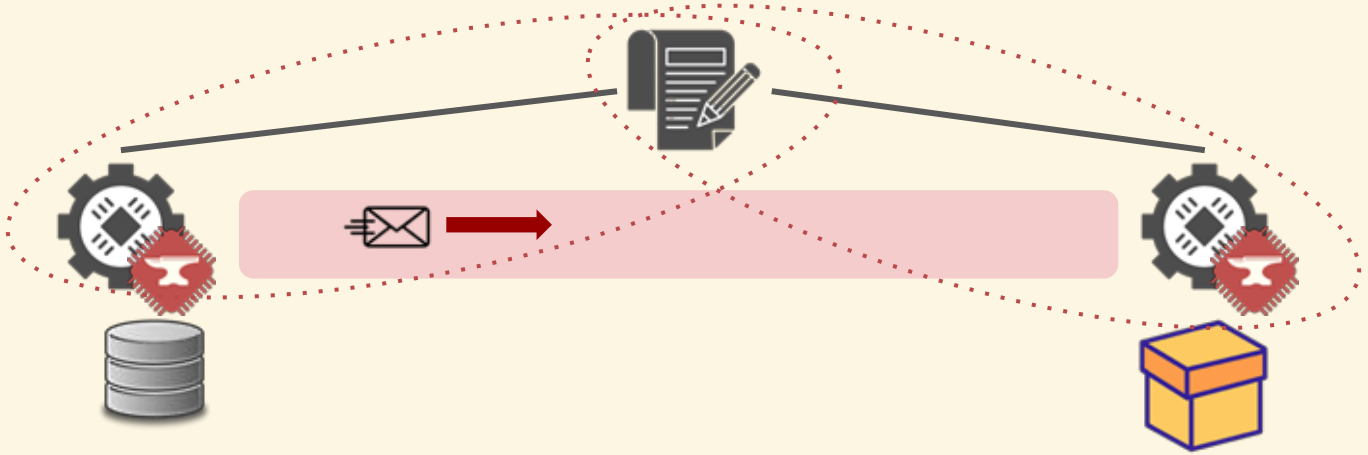
Lifetime Contracts



How to describe time in a contract?

Message passing

Lifetime Contracts

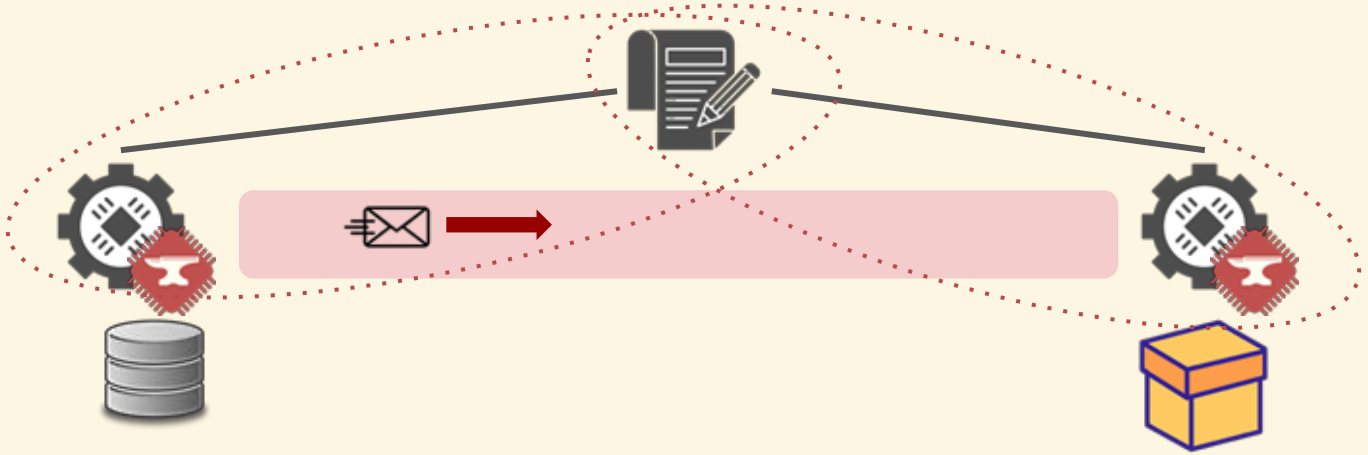


How to describe time in a contract?

Message passing

$t + k$

Lifetime Contracts



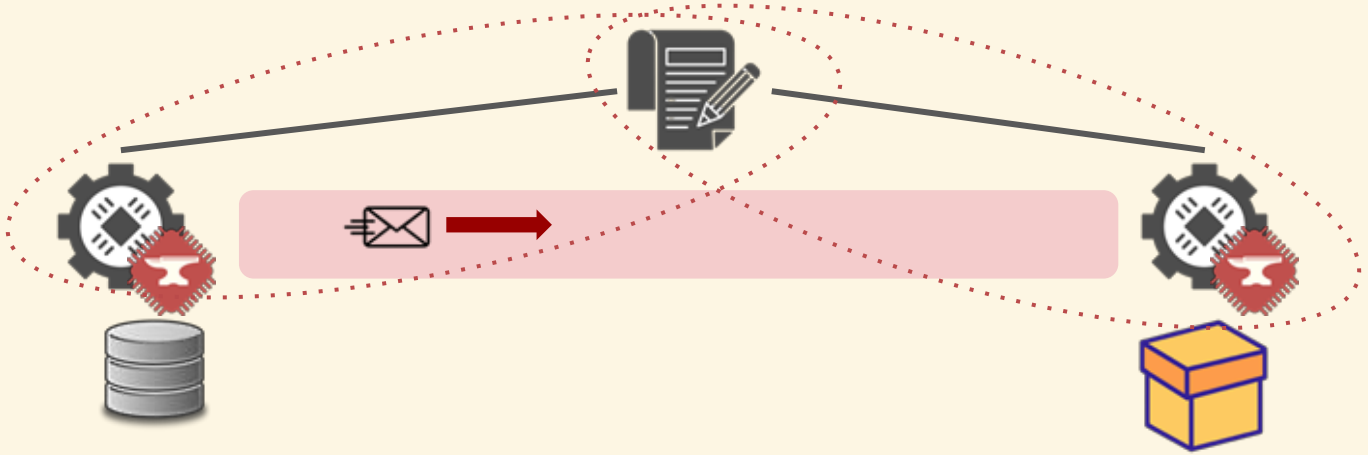
How to describe time in a contract?

Message passing

$t + k$

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#3)
}
```

Lifetime Contracts



How to describe time in a contract?

Message passing

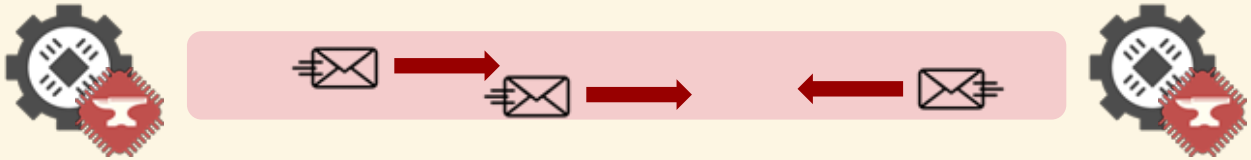
$t + k$

How about dynamic lifetimes?

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#3)
}
```

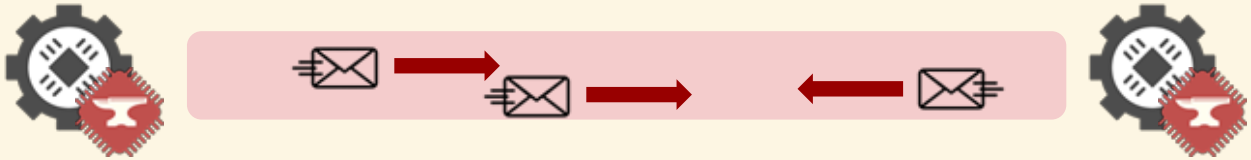
Dynamic Lifetime Contracts

Observation: Multiple message types share the same channel



Dynamic Lifetime Contracts

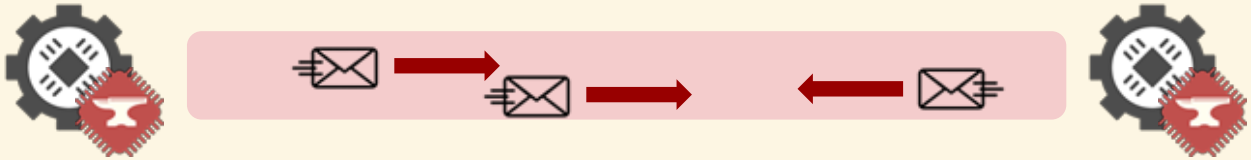
Observation: Multiple message types share the same channel



Agreed on: *t* + *m* (first *m* after *t*)

Dynamic Lifetime Contracts

Observation: Multiple message types share the same channel

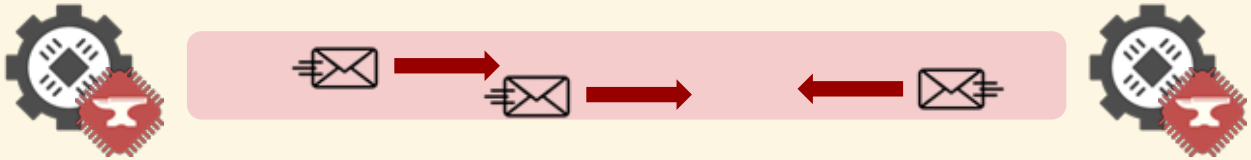


Agreed on: *t* + *m* (first *m* after *t*)

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#3)
}
```

Dynamic Lifetime Contracts

Observation: Multiple message types share the same channel



Agreed on: *t* + *m* (first *m* after *t*)

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@#3)
}
```

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
```

More Interesting Constructs in Anvil

Sync mode: two-way handshake not always needed

Recursive: easy pipelining

External proc: interfacing with foreign modules

More Interesting Constructs in Anvil

Sync mode: two-way handshake not always needed

Recursive: easy pipelining

External proc: interfacing with foreign modules

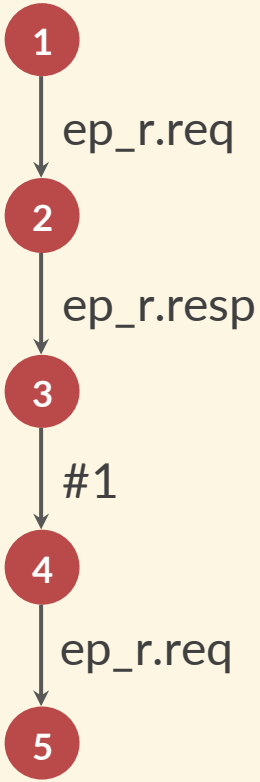
Find them in paper or documentation

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
    send ep_r.req(...) >>
    let res = recv ep_r.resp >>
    dprint "result = %d" (res);
    cycle 1 >>
    send ep_r.req(...) >>
    dprint "result = %d" (res);
    ...
```

Type Checking on Event Graph: Example

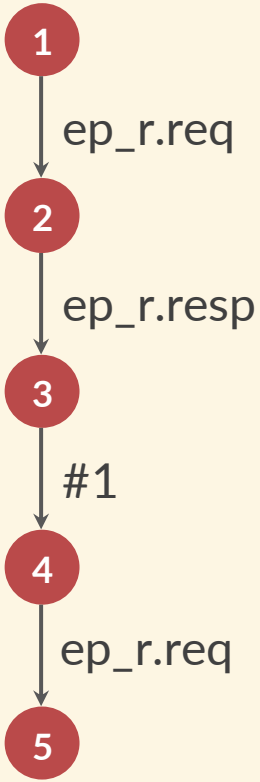
```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
1  send ep_r.req(...) >>
2  let res = recv ep_r.resp >>
3  dprint "result = %d" (res);
3  cycle 1 >>
4  send ep_r.req(...) >>
5  dprint "result = %d" (res);
   ...
```



Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
```

```
1  send ep_r.req(...) >>
2  let res = recv ep_r.resp >>
3  dprint "result = %d" (res);
3  cycle 1 >>
4  send ep_r.req(...) >>
5  dprint "result = %d" (res);
   ...
```



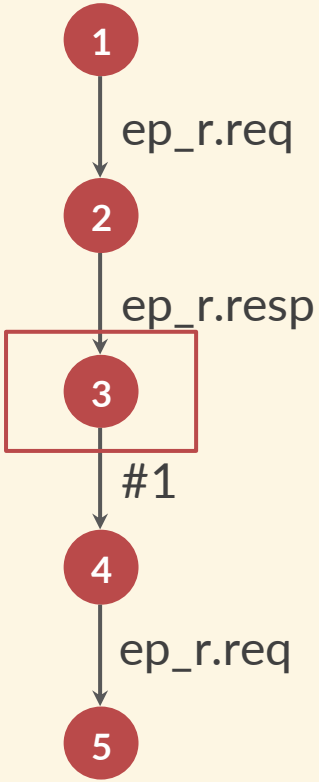
Lifetime of `res`

? — ?

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
```

```
1  send ep_r.req(...) >>
2  let res = recv ep_r.resp >>
3  dprint "result = %d" (res);
3  cycle 1 >>
4  send ep_r.req(...) >>
5  dprint "result = %d" (res);
   ...
```



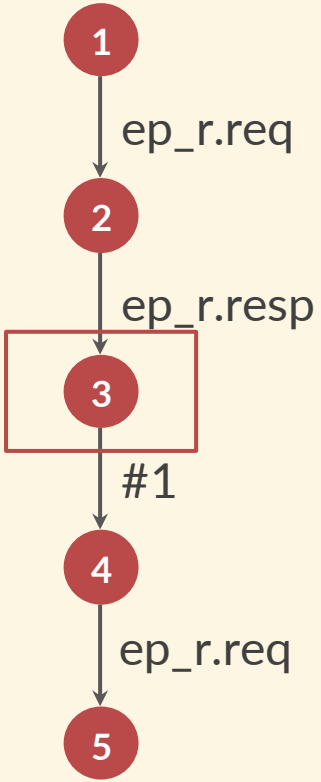
Lifetime of `res`

? — ?

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
```

```
1  send ep_r.req(...) >>
2  let res = recv ep_r.resp >>
3  dprint "result = %d" (res);
3  cycle 1 >>
4  send ep_r.req(...) >>
5  dprint "result = %d" (res);
   ...
```



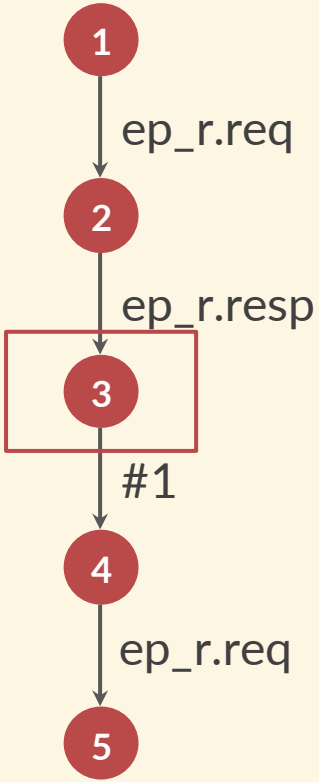
Lifetime of `res`

e3 — ?

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}

1   send ep_r.req(...) >>
2   let res = recv ep_r.resp >>
3   dprint "result = %d" (res);
3   cycle 1 >>
4   send ep_r.req(...) >>
5   dprint "result = %d" (res);
    ...
```

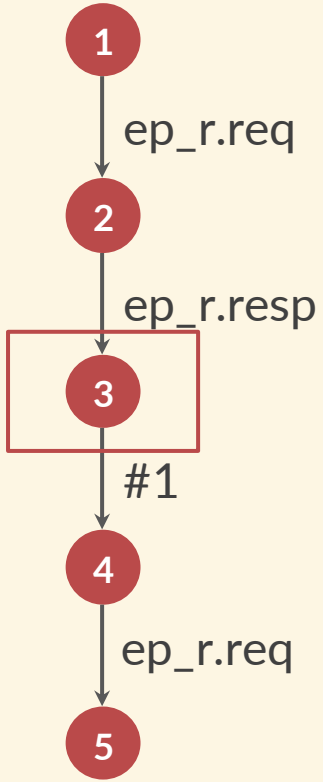


Lifetime of `res`

e3 — ?

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
1 send ep_r.req(...) >>
2 let res = recv ep_r.resp >>
3 dprint "result = %d" (res);
3 cycle 1 >>
4 send ep_r.req(...) >>
5 dprint "result = %d" (res);
  ...
```

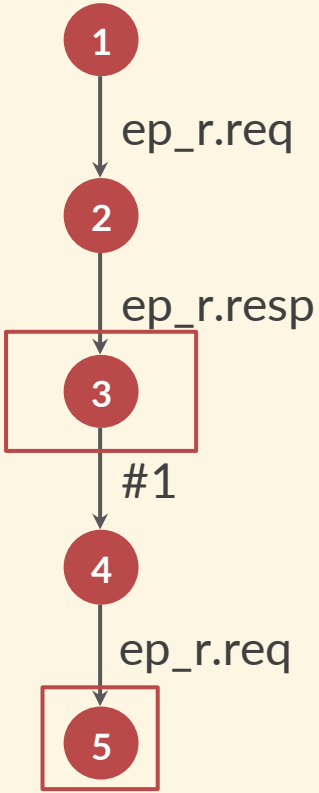


Lifetime of `res`

e3 — e3 + req

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
1 send ep_r.req(...) >>
2 let res = recv ep_r.resp >>
3 dprint "result = %d" (res);
3 cycle 1 >>
4 send ep_r.req(...) >>
5 dprint "result = %d" (res);
  ...
```



Lifetime of `res`

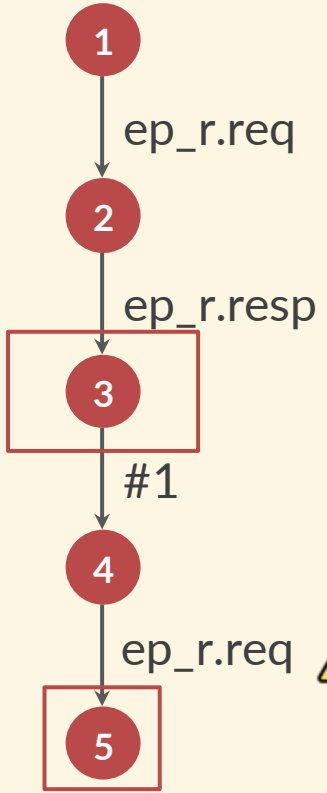
e3 — e3 + req

Use of `res` at e5 requires:
e5 be within the lifetime

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
```

```
1  send ep_r.req(...) >>
2  let res = recv ep_r.resp >>
3  dprint "result = %d" (res);
3  cycle 1 >>
4  send ep_r.req(...) >>
5  dprint "result = %d" (res);
   ...
```



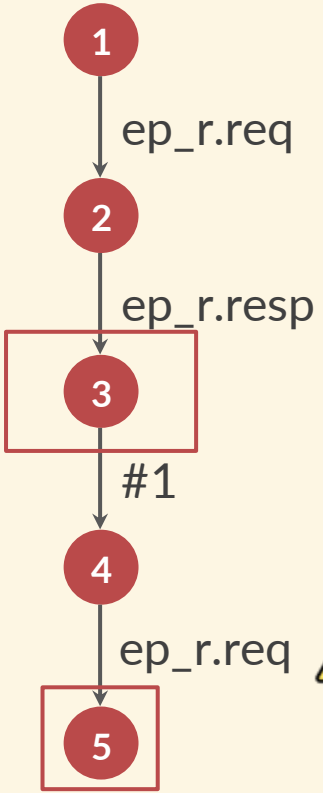
Lifetime of `res`

e3 — e3 + req

Use of `res` at e5 requires:
e5 be within the lifetime

Type Checking on Event Graph: Example

```
chan ch {
    left req : (request@#1),
    right resp : (logic[16]@req)
}
1  send ep_r.req(...) >>
2  let res = recv ep_r.resp >>
3  dprint "result = %d" (res);
3  cycle 1 >>
4  send ep_r.req(...) >>
5  dprint "result = %d" (res);
   ...
```



Lifetime of `res`

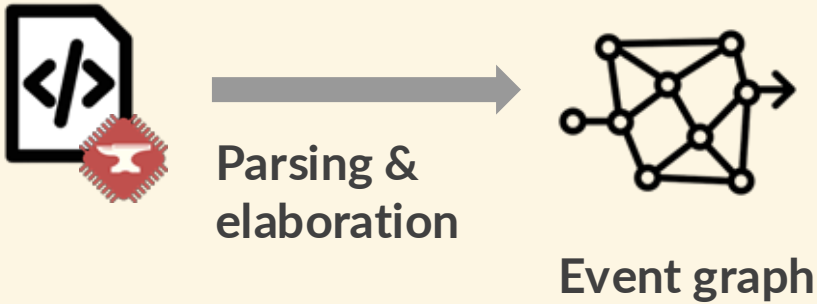
e3 — e3 + req

Use of `res` at e5 requires:
e5 be within the lifetime

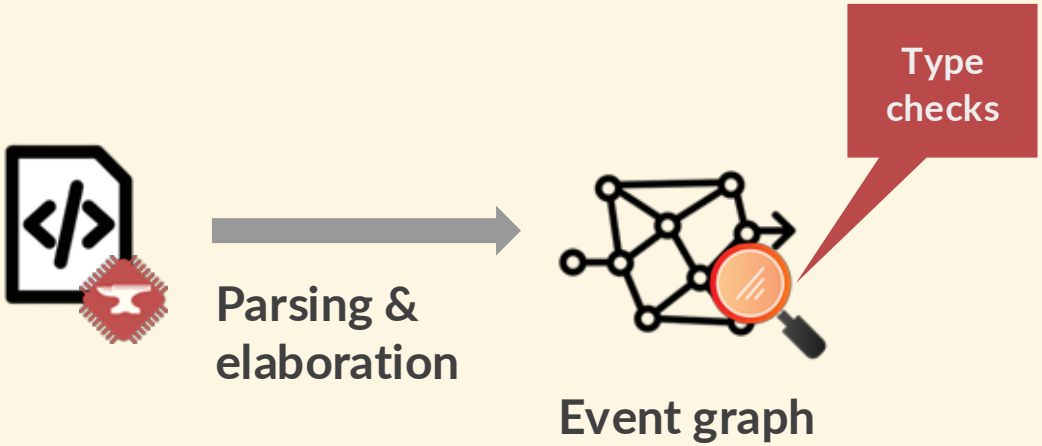
Implementation of the other module not needed

Anvil Compiler: Event Graph as IR

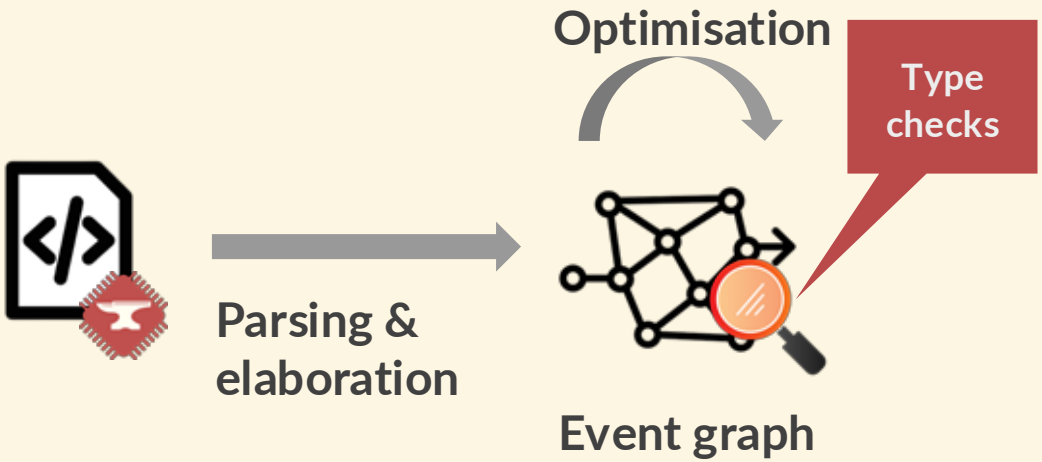
Anvil Compiler: Event Graph as IR



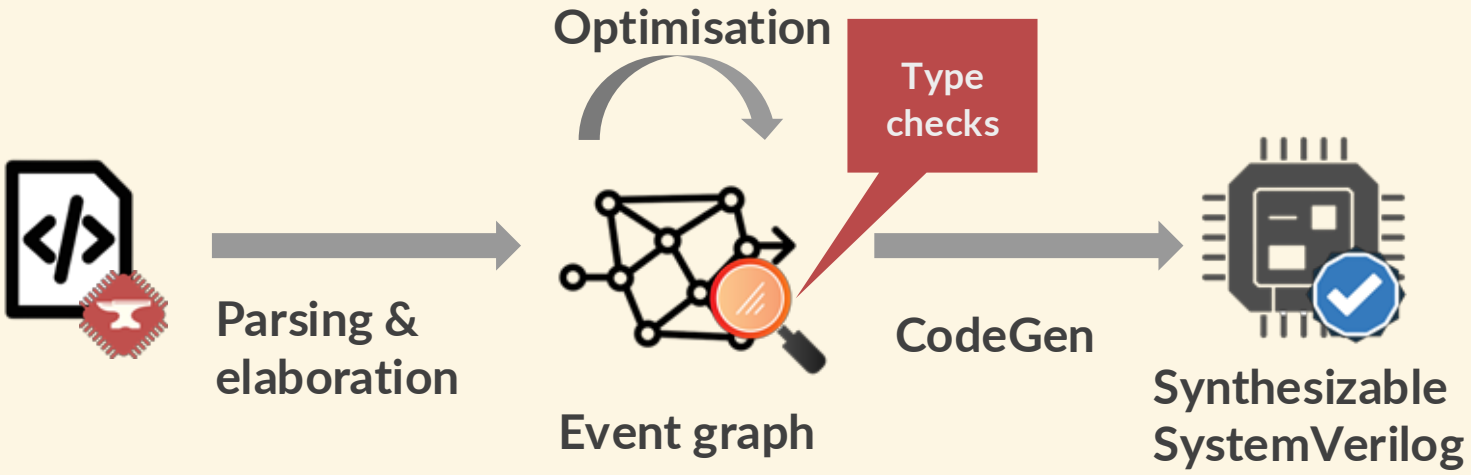
Anvil Compiler: Event Graph as IR



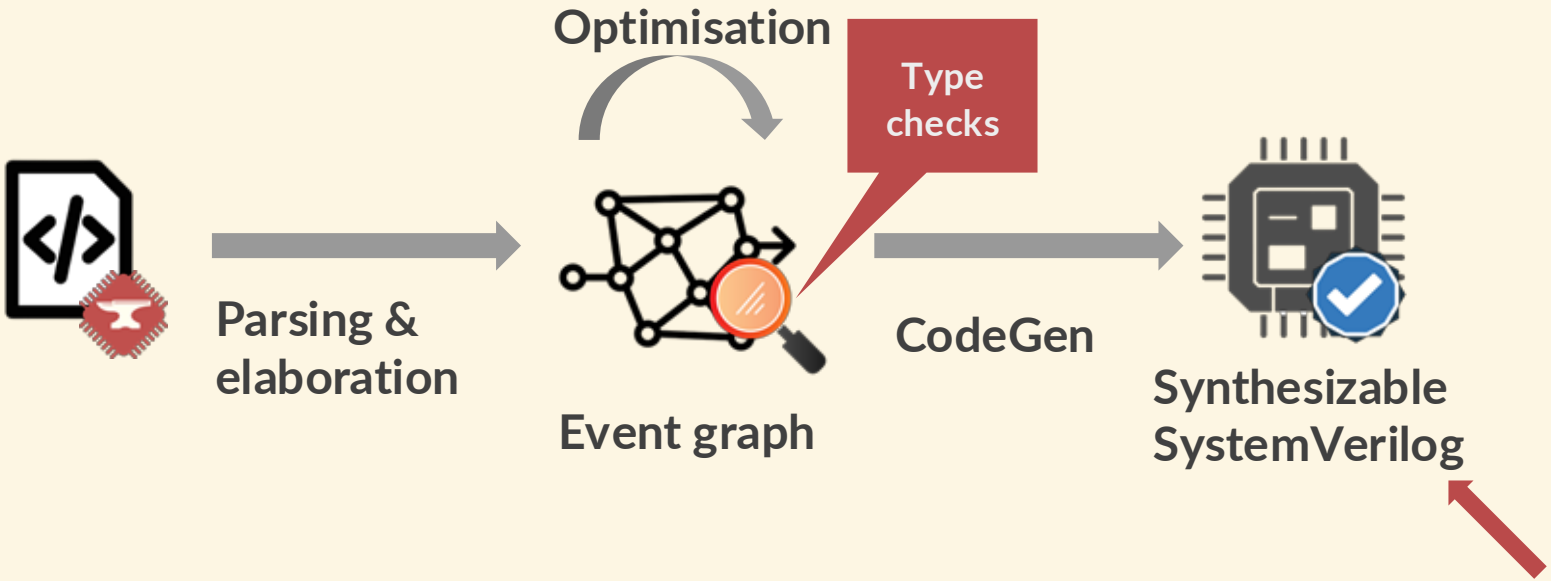
Anvil Compiler: Event Graph as IR



Anvil Compiler: Event Graph as IR



Anvil Compiler: Event Graph as IR



Evaluation Questions

Safety
Timing hazard prevention?

Expressiveness
Expressing cycle-level behaviour of real-world designs?

Practicality
Practical overhead in generated hardware?

Evaluation Questions

Safety
Timing hazard prevention?

Formalisation
Soundness proof:
P type checks ⇒ P is timing-safe

Expressiveness
Expressing cycle-level behaviour of real-world designs?

Practicality
Practical overhead in generated hardware?

Evaluation Questions

Safety
Timing hazard prevention?

Formalisation
Soundness proof:
P type checks ⇒ P is timing-safe

Expressiveness
Expressing cycle-level behaviour of real-world designs?

Experiments

Practicality
Practical overhead in generated hardware?

Evaluation Questions

Safety
Timing hazard prevention?

Formalisation
Soundness proof:
P type checks ⇒ P is timing-safe

Expressiveness
Expressing cycle-level behaviour of real-world designs?

Experiments

Practicality
Practical overhead in generated hardware?

Evaluation Methodology

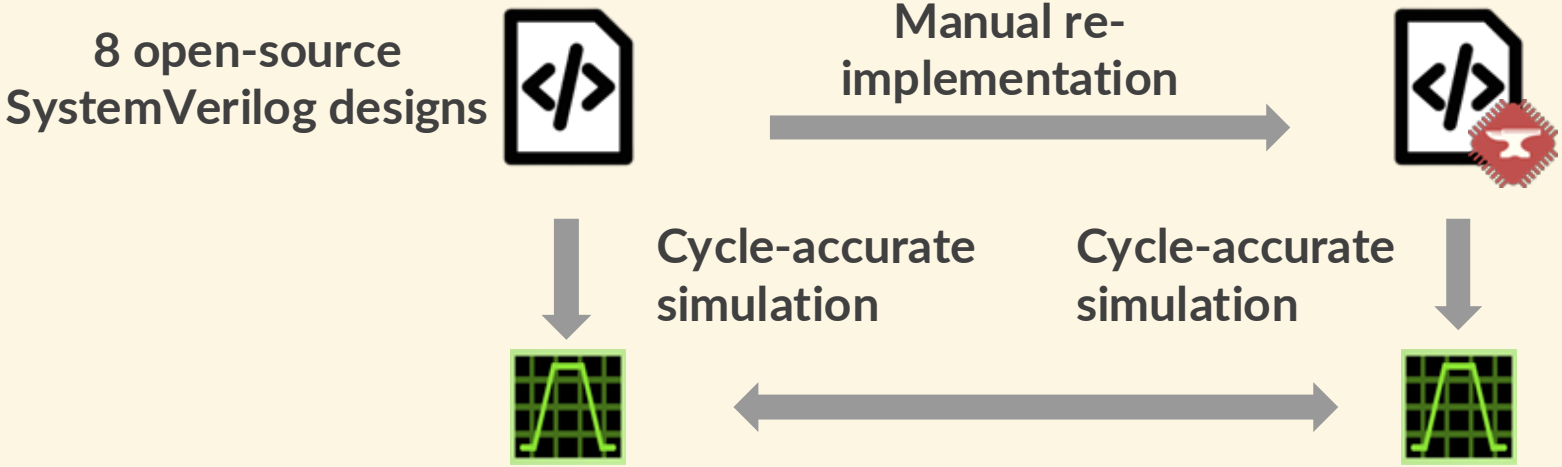
8 open-source
SystemVerilog designs

Evaluation Methodology

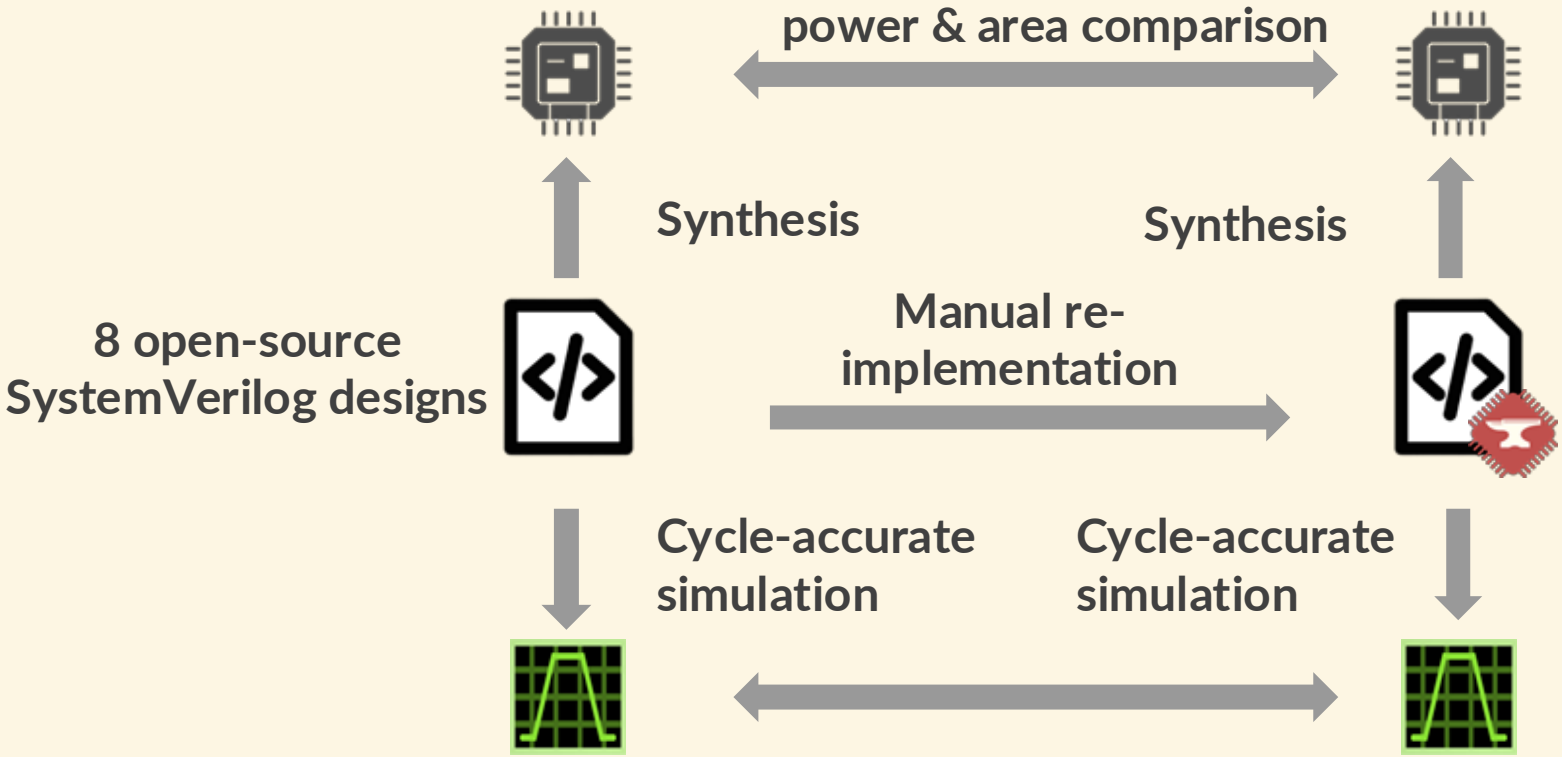
8 open-source SystemVerilog designs

Manual re-implementation

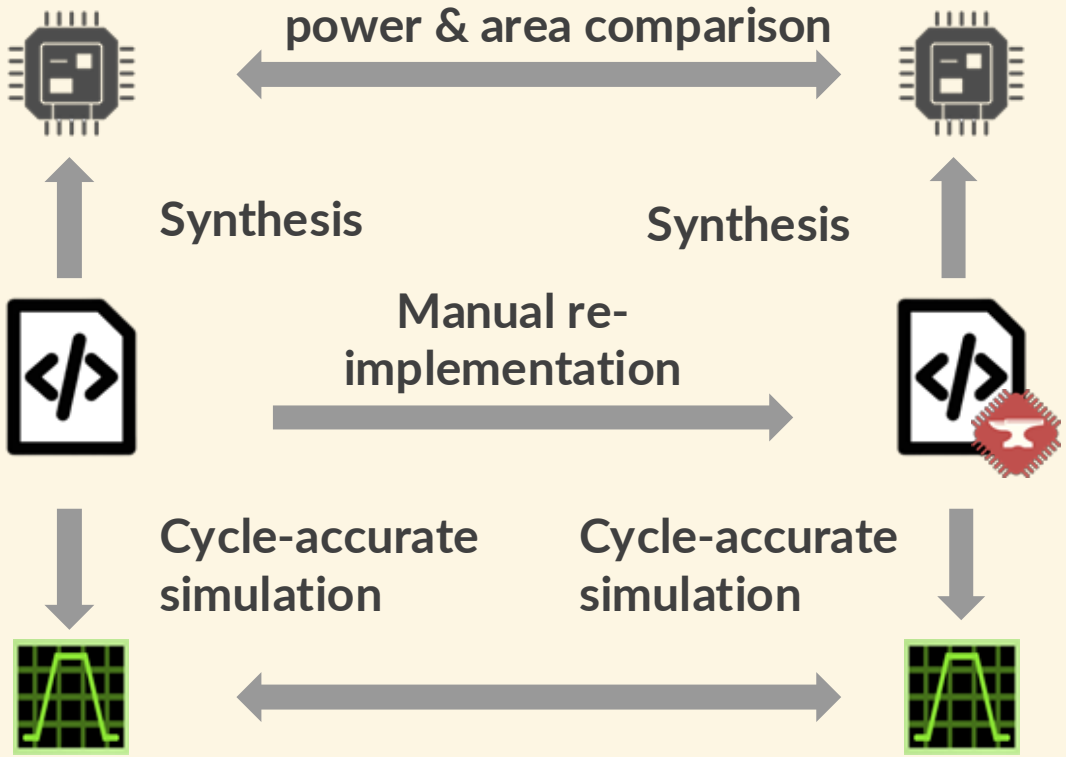
Evaluation Methodology



Evaluation Methodology



Evaluation Methodology



Diverse, Dynamic timing

Data structure
Cryptographic accelerator
Bus router
Memory management unit

Evaluation Result (Anvil vs SystemVerilog)

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
FIFO Buffer (SV)	690	674 (−2%)	1.434	1.403 (−2%)	4062	4156	dyn	0
Spill Register (SV)	165	171 (3%)	0.459	0.469 (2%)	5187	5375	dyn	0
Passthrough Stream FIFO (SV)	679	679 (0%)	1.239	1.264 (2%)	4093	3625	1	0
CVA6 Translation Lookaside Buffer (SV)	5561	5611 (0%)	5.813	5.835 (0%)	2468	2406	dyn	0
CVA6 Page Table Walker (SV)	499	561 (12%)	0.649	0.676 (4%)	3531	3281	dyn	0
AES Cipher Core (SV)	9096	9090 (0%)	0.793	0.972 (22%)	781	1229	dyn	0
AXI Lite Demux Router (SV)	1318	1469 (11%)	1.351	1.385 (2%)	2437	2125	dyn	0
AXI Lite Mux Router (SV)	1448	1633 (12%)	1.336	1.324 (0%)	2406	2187	dyn	0
Average overhead compared with SystemVerilog baselines: Area = 4.50%, **Power** = 3.75%								

Evaluation Result (Anvil vs SystemVerilog)

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
FIFO Buffer (SV)	690	674 (−2%)	1.434	1.403 (−2%)	4062	4156	dyn	0
Spill Register (SV)	165	171 (3%)	0.459	0.469 (2%)	5187	5375	dyn	0
Passthrough Stream FIFO (SV)	679	679 (0%)	1.239	1.264 (2%)	4093	3625	1	0
CVA6 Translation Lookaside Buffer (SV)	5561	5611 (0%)	5.813	5.835 (0%)	2468	2406	dyn	0
CVA6 Page Table Walker (SV)	499	561 (12%)	0.649	0.676 (4%)	3531	3281	dyn	0
AES Cipher Core (SV)	9096	9090 (0%)	0.793	0.972 (22%)	781	1229	dyn	0
AXI Lite Demux Router (SV)	1318	1469 (11%)	1.351	1.385 (2%)	2437	2125	dyn	0
AXI Lite Mux Router (SV)	1448	1633 (12%)	1.336	1.324 (0%)	2406	2187	dyn	0
Average overhead compared with SystemVerilog baselines: Area = 4.50%, **Power** = 3.75%								

Evaluation Result (Anvil vs SystemVerilog)

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
FIFO Buffer (SV)	690	674 (−2%)	1.434	1.403 (−2%)	4062	4156	dyn	0
Spill Register (SV)	165	171 (3%)	0.459	0.469 (2%)	5187	5375	dyn	0
Passthrough Stream FIFO (SV)	679	679 (0%)	1.239	1.264 (2%)	4093	3625	1	0
CVA6 Translation Lookaside Buffer (SV)	5561	5611 (0%)	5.813	5.835 (0%)	2468	2406	dyn	0
CVA6 Page Table Walker (SV)	499	561 (12%)	0.649	0.676 (4%)	3531	3281	dyn	0
AES Cipher Core (SV)	9096	9090 (0%)	0.793	0.972 (22%)	781	1229	dyn	0
AXI Lite Demux Router (SV)	1318	1469 (11%)	1.351	1.385 (2%)	2437	2125	dyn	0
AXI Lite Mux Router (SV)	1448	1633 (12%)	1.336	1.324 (0%)	2406	2187	dyn	0
Average overhead compared with SystemVerilog baselines: Area = 4.50%, **Power** = 3.75%								

Comparison with SystemVerilog designs:

- No additional cycle-level latency overhead
- Average overhead: area 4.50%, power 3.75%

Evaluation Result (Anvil vs SystemVerilog)

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
FIFO Buffer (SV)	690	674 (−2%)	1.434	1.403 (−2%)	4062	4156	dyn	0
Spill Register (SV)	165	171 (3%)	0.459	0.469 (2%)	5187	5375	dyn	0
Passthrough Stream FIFO (SV)	679	679 (0%)	1.239	1.264 (2%)	4093	3625	1	0
CVA6 Translation Lookaside Buffer (SV)	5561	5611 (0%)	5.813	5.835 (0%)	2468	2406	dyn	0
CVA6 Page Table Walker (SV)	499	561 (12%)	0.649	0.676 (4%)	3531	3281	dyn	0
AES Cipher Core (SV)	9096	9090 (0%)	0.793	0.972 (22%)	781	1229	dyn	0
AXI Lite Demux Router (SV)	1318	1469 (11%)	1.351	1.385 (2%)	2437	2125	dyn	0
AXI Lite Mux Router (SV)	1448	1633 (12%)	1.336	1.324 (0%)	2406	2187	dyn	0
Average overhead compared with SystemVerilog baselines: Area = 4.50%, **Power** = 3.75%								

Comparison with SystemVerilog designs:

- No additional cycle-level latency overhead
- Average overhead: area 4.50%, power 3.75%

Comparison with Filament

Filament (Nigam et al., 2023): timeline types, specialized for pipelines with static timing

Comparison with Filament

Filament (Nigam et al., 2023): timeline types, specialized for pipelines with static timing

Comparison with Filament

Filament (Nigam et al., 2023): timeline types, specialized for pipelines with static timing

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
Pipelined ALU (Filament)	498	404 (−18%)	0.606	0.626 (3%)	3062	4675	1	0
Systolic Array (Filament)	2509	2425 (−3%)	2.493	2.764 (10%)	2400	2862	1	0
Average overhead compared with Filament baselines: Area = −10.5%, **Power** = 6.5%								

Comparison with Filament

Filament (Nigam et al., 2023): timeline types, specialized for pipelines with static timing

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
Pipelined ALU (Filament)	498	404 (−18%)	0.606	0.626 (3%)	3062	4675	1	0
Systolic Array (Filament)	2509	2425 (−3%)	2.493	2.764 (10%)	2400	2862	1	0
Average overhead compared with Filament baselines: Area = −10.5%, **Power** = 6.5%								

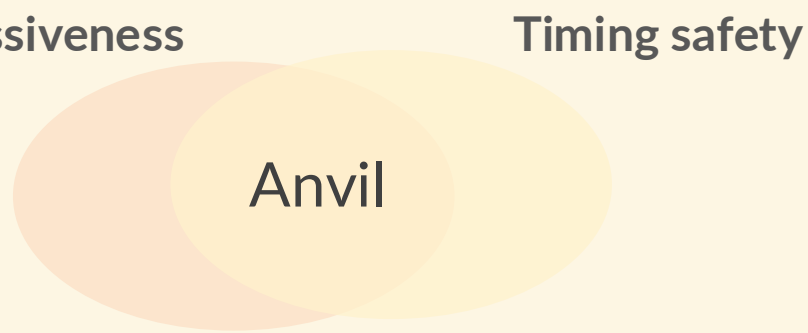
Comparison with Filament designs:

- No additional cycle-level latency overhead
- Average overhead: area -10.50%, power 6.5%

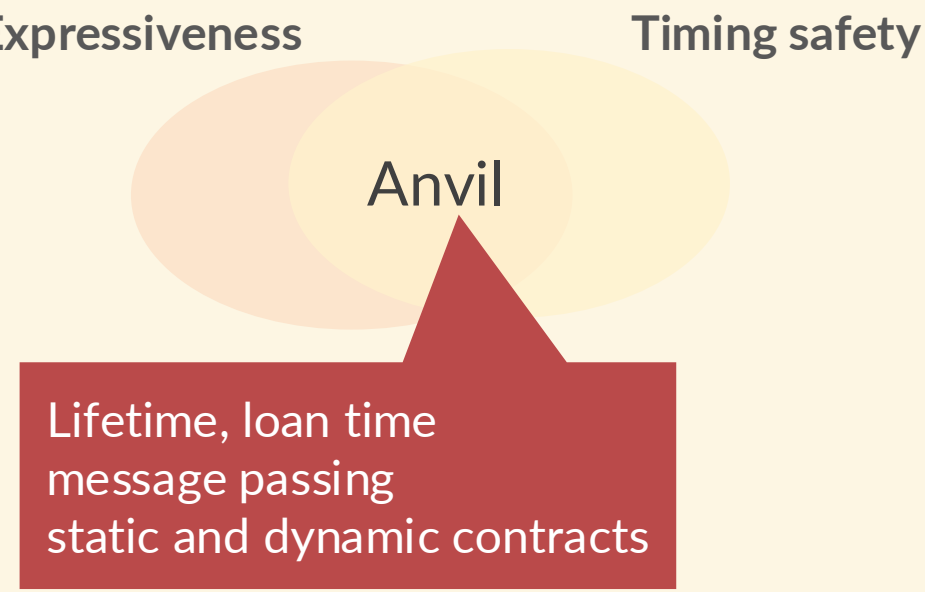
Conclusion

Timing safety

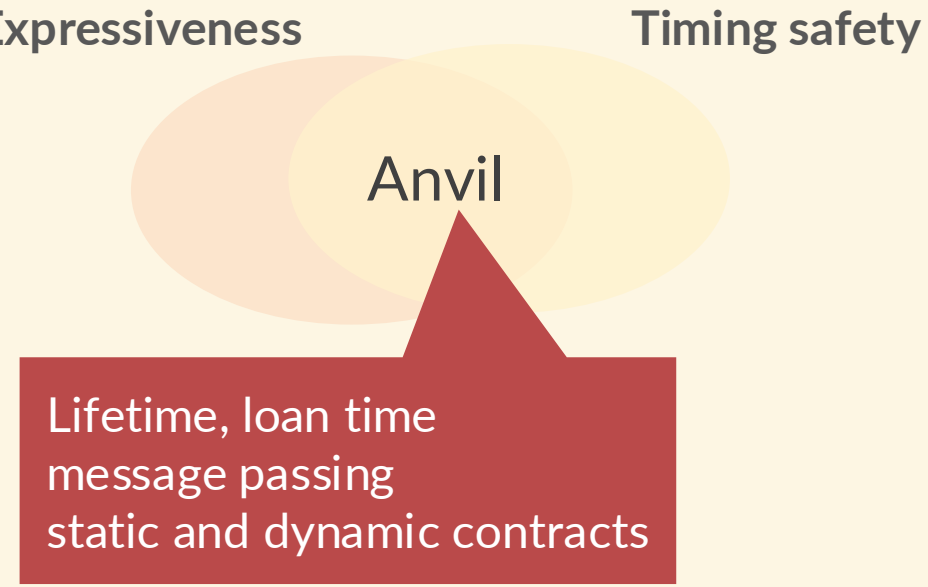
Conclusion



Conclusion


Expressiveness
Timing safety
Anvil
Lifetime, loan time
message passing
static and dynamic contracts

Conclusion



Practical

practical overhead (4.50% area, 3.75% power)

Conclusion

Expressiveness **Timing safety**

Lifetime, loan time
message passing
static and dynamic contracts

Anvil Playground

Practical

practical overhead (4.50% area, 3.75% power)