IODINE: Verifying Constant-Time Execution of Hardware

Klaus von Gleissenthall

Rami Gökhan Kıcı

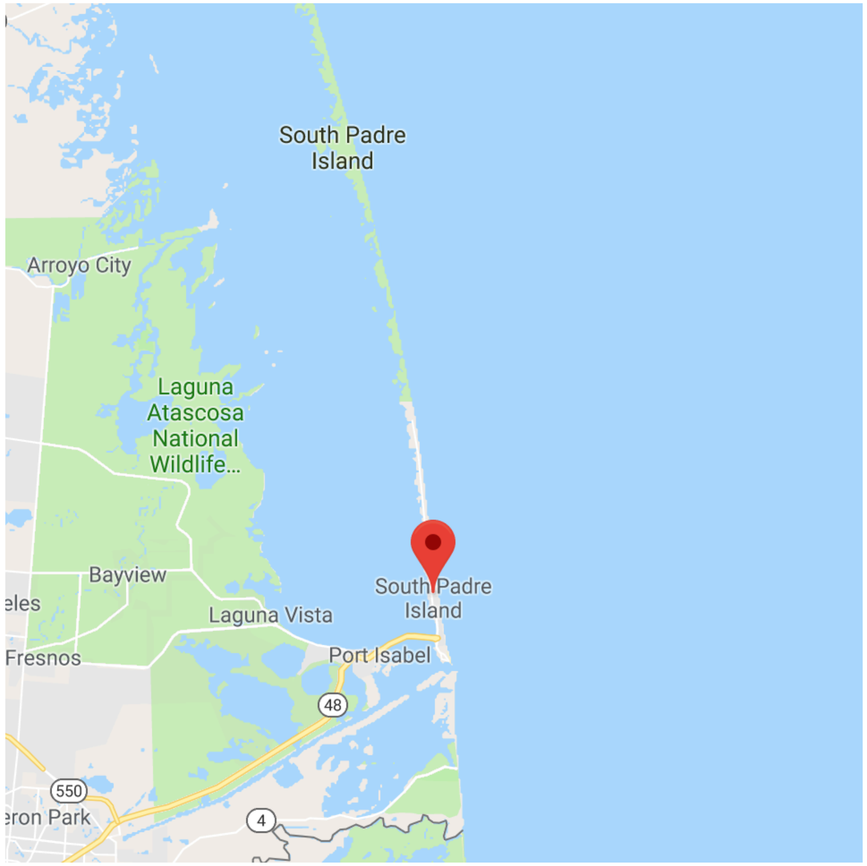
Deian Stefan

Ranjit Jhala

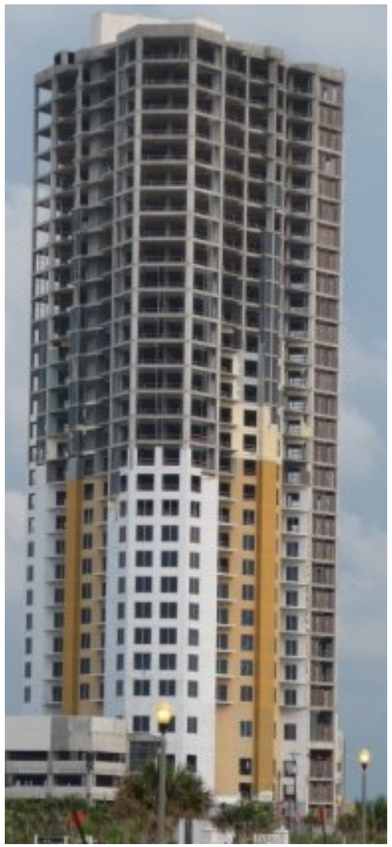
UC San Diego

2006: Investors set out to build luxury tower

$2m per unit, built to withstand extreme winds



Two years later construction stopped



… and the tower was demolished

Why?

It was built on *expandable clay*

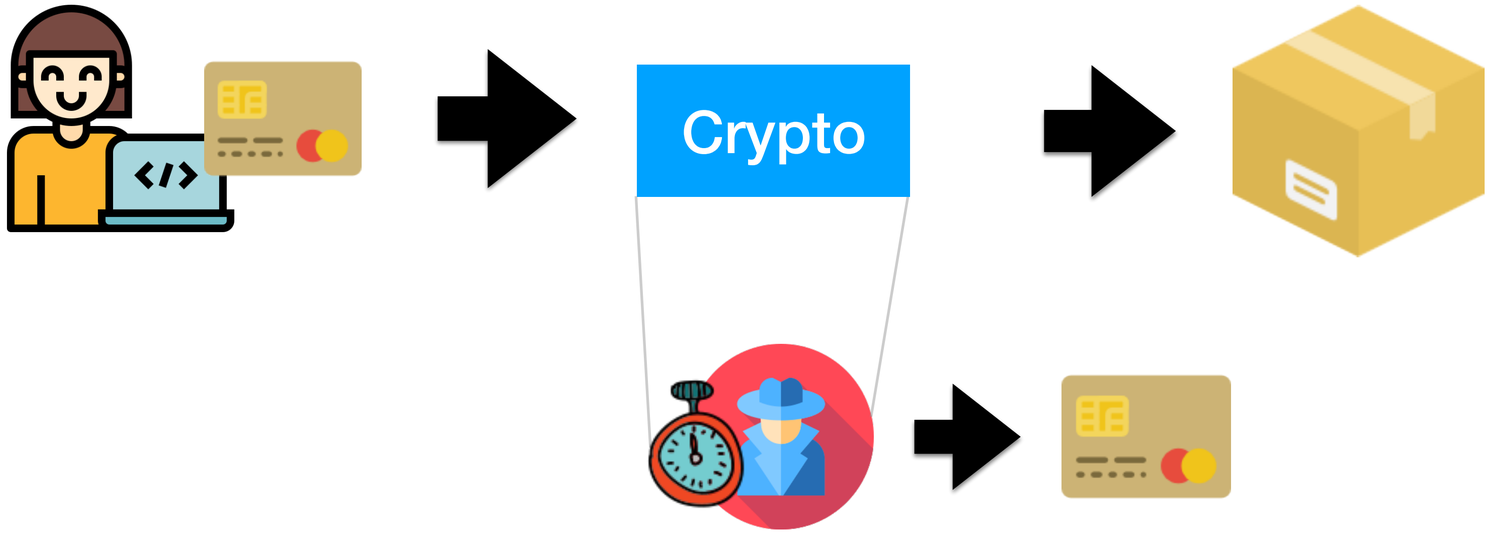
… that compresses under weight,

… causing the tower to sink


02/27/2008

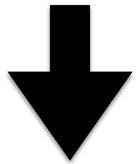
Our towers are *crypto algorithms*

Instead of wind, we worry about *timing side channels*



Like reinforcing, we write constant time code

```
if (secret) x = e
```



```
x = (-secret & e) | (secret - 1) & x
```

We even verify that code really executes in constant time

HACL* is used in Firefox

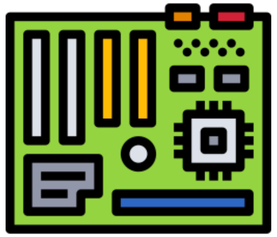
Fiat Crypto is used in Chrome

miTLS influenced the TLS spec

Is this code actually safe?

It depends on what we build on!

We build on hardware!



AND

SUB

XOR

⋮

We trust hardware to be constant time

What if it’s not?

AND	DIV
SUB	FMUL
XOR	FDIV
⋮	⋮

We need to *verify* that hardware is constant time!

But How? There's no hardware definition!

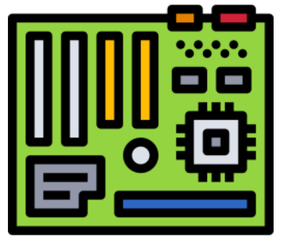
Can we use
Software Methods?

Definitions Don’t Apply: Parallelism

Software

straight line code

sequential



Hardware

never terminates

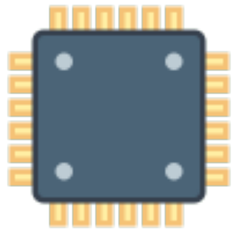
parallel

pipelined

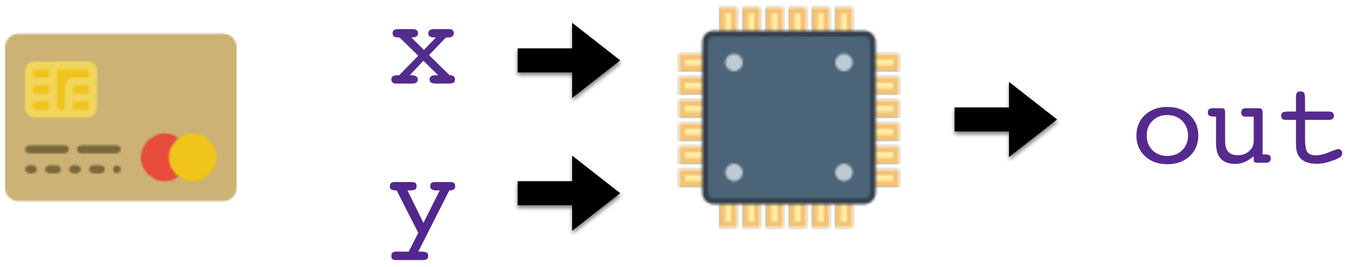
Outline

1. Definition
2. Verification: Iodine 
3. Evaluation

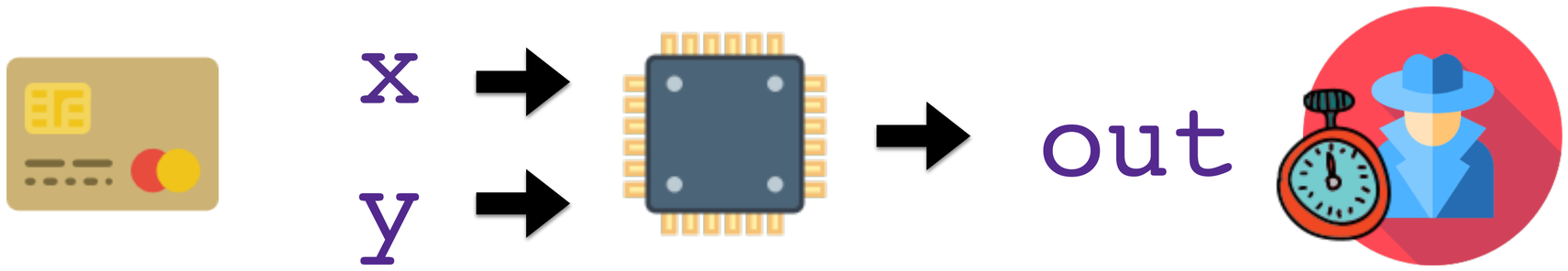
Definition Example: FPU multiplier



MUL x y



Attacker can observe timing of outputs



Multiplies numbers *x and y*

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Set flag `iszero`, if `x` or `y` is zero

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

If `iszero` is set, return zero

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

else, compute `flp_res` along slow path

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

How do we show that the multiplier is not constant time?

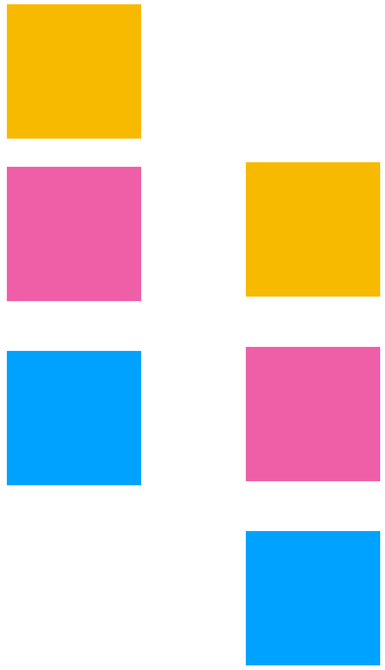
What’s the timing of a pipelined computation ?

Stage 1

Stage 2

Stage 3

What’s the timing of a pipelined computation ?

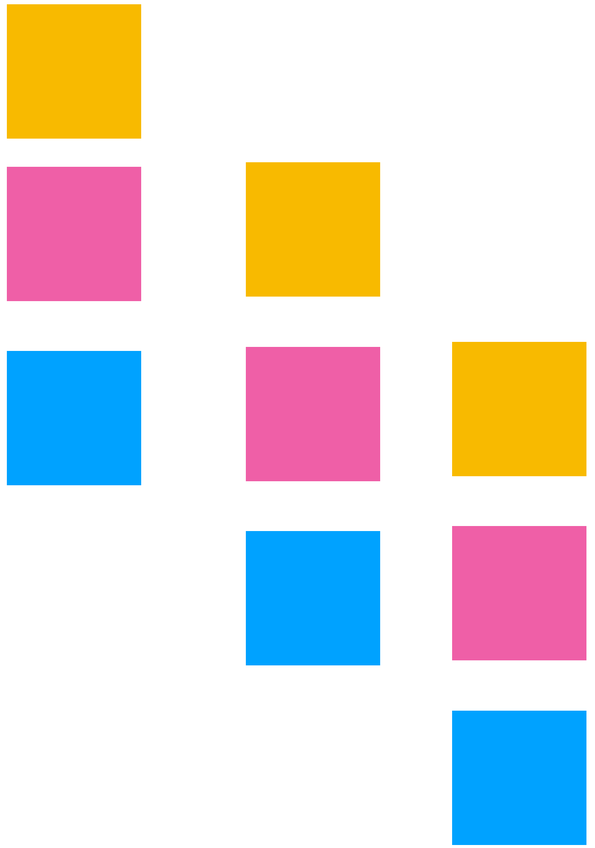


Stage 1

Stage 2

Stage 3

What’s the timing of a pipelined computation ?



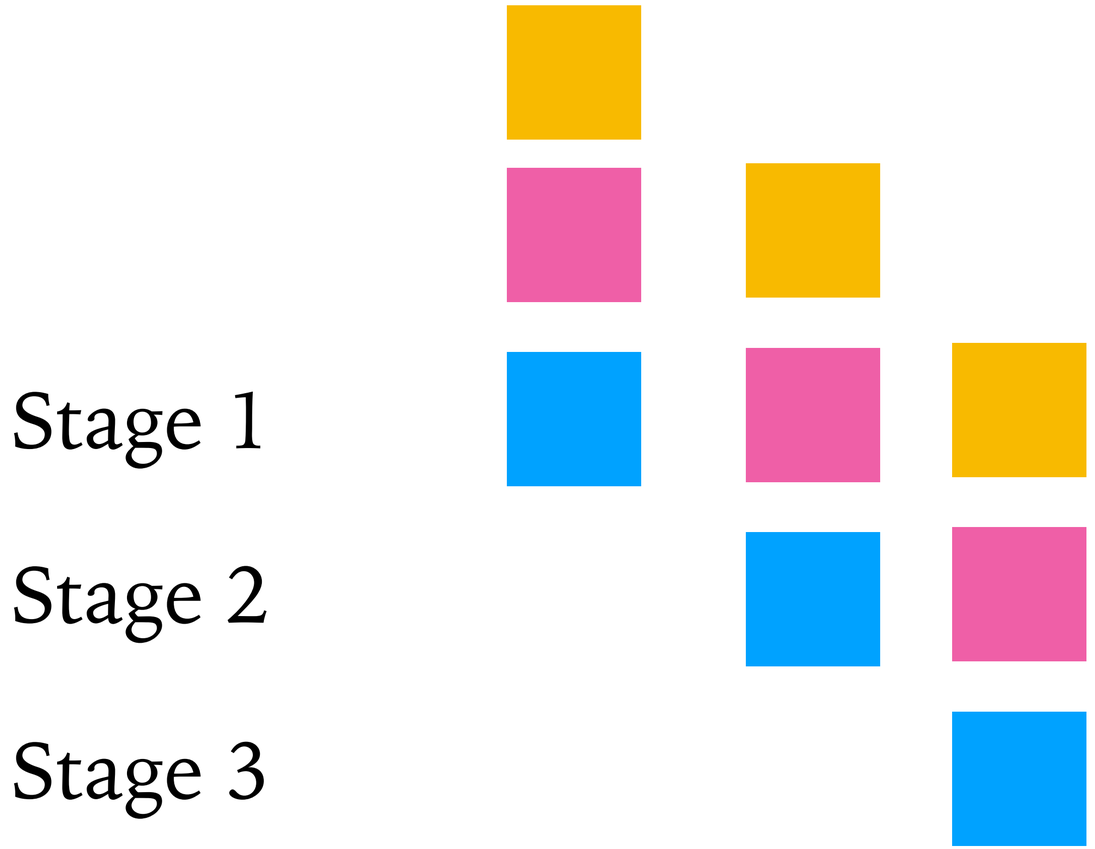
Stage 1

Stage 2

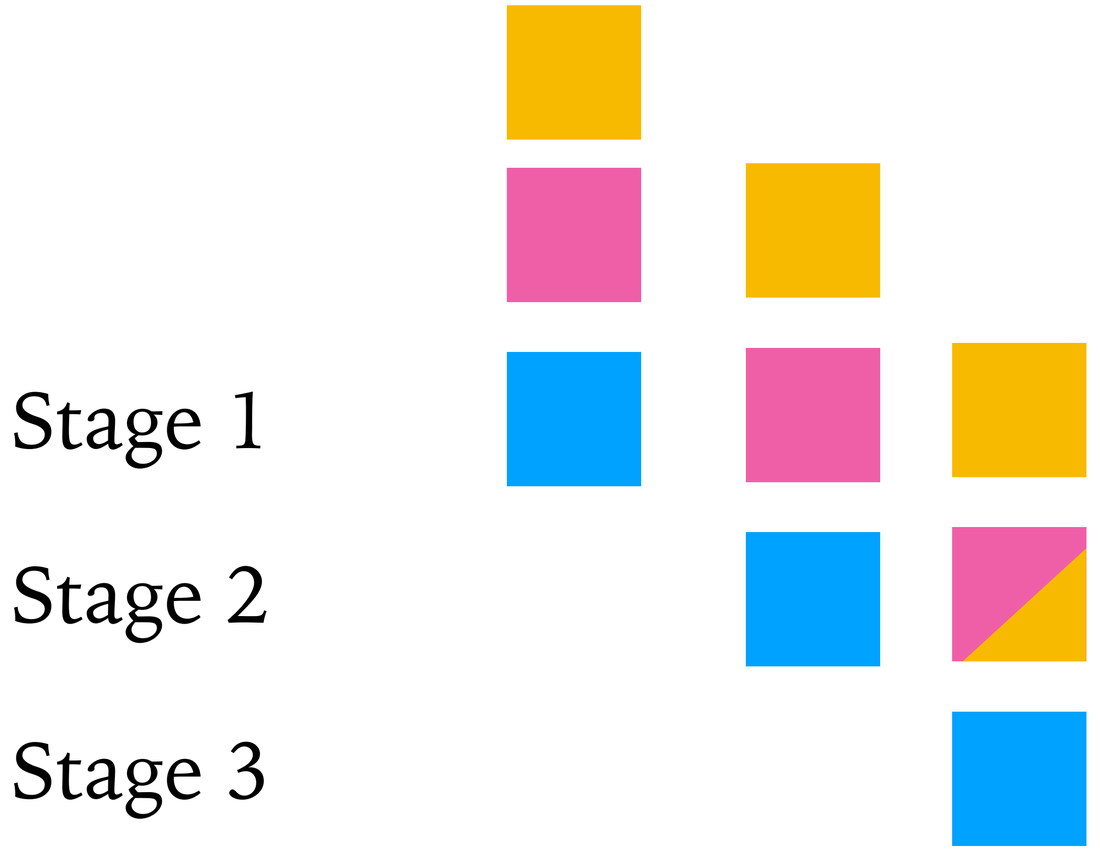
Stage 3

3?

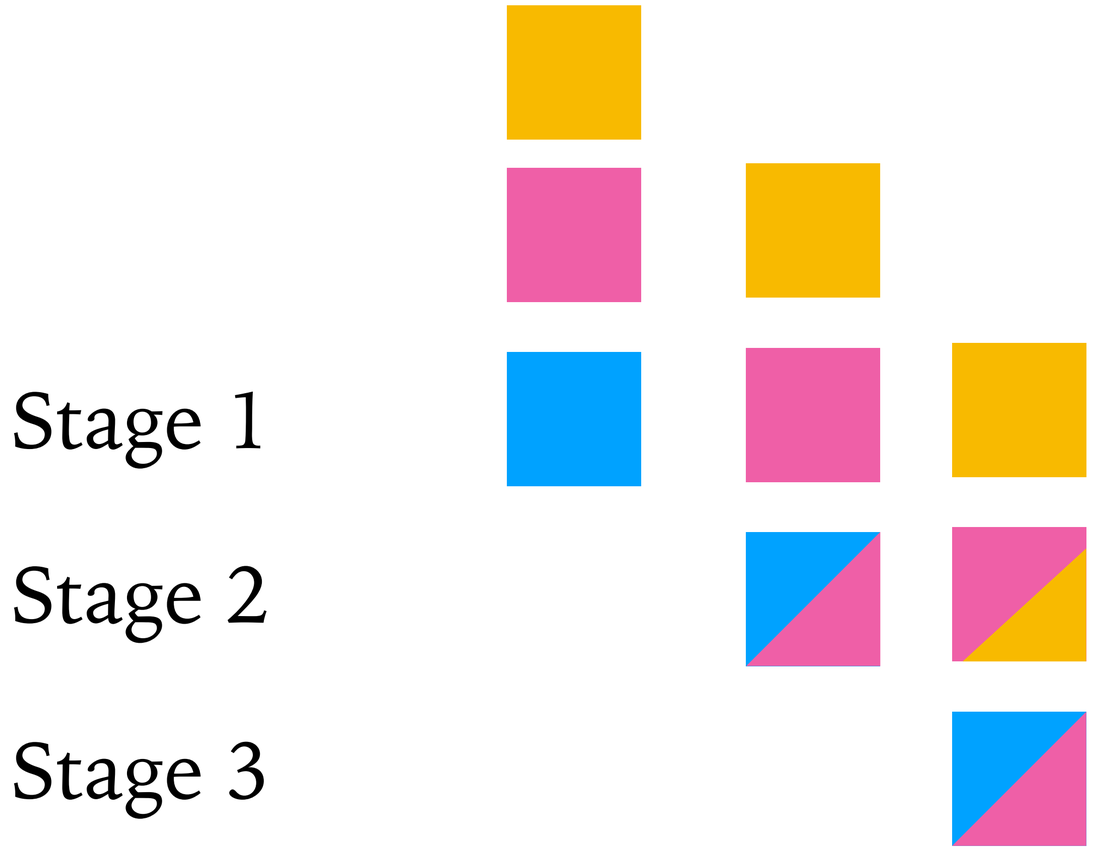
But what, if instructions influence each other?



But what, if instructions influence each other?



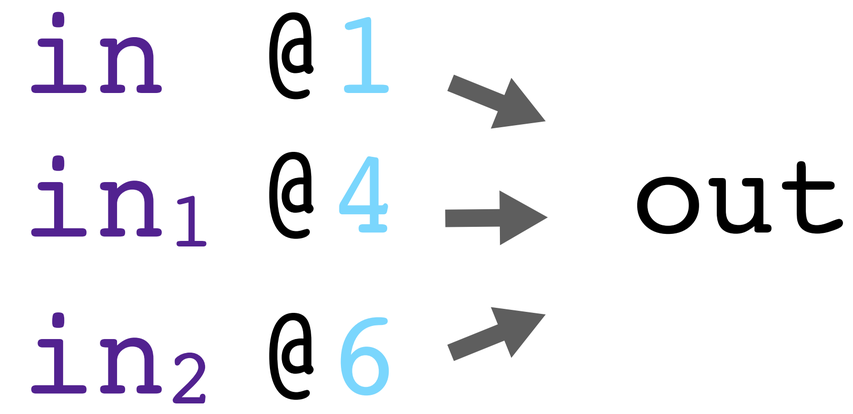
But what, if instructions influence each other?



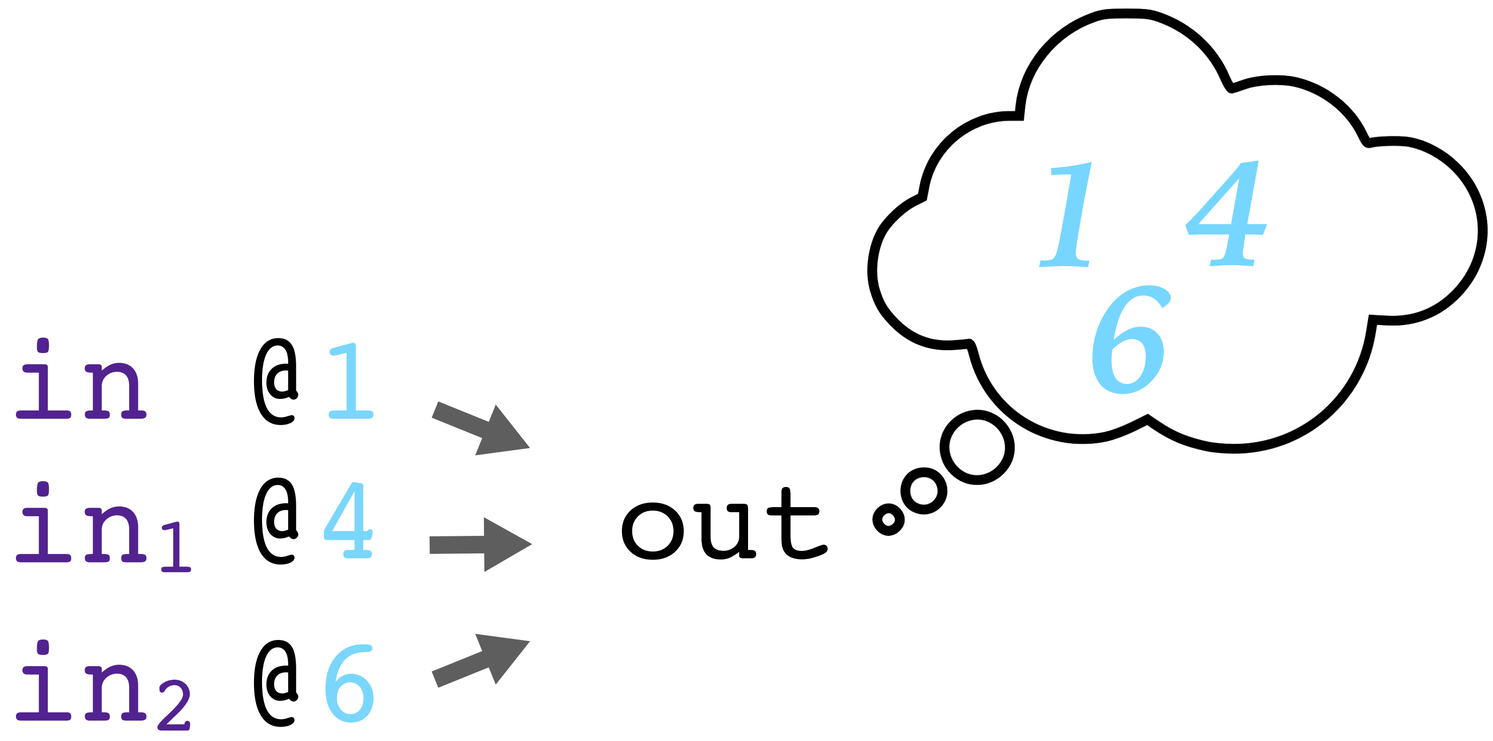
Instead, we

…track which inputs influenced a register

… at which cycle

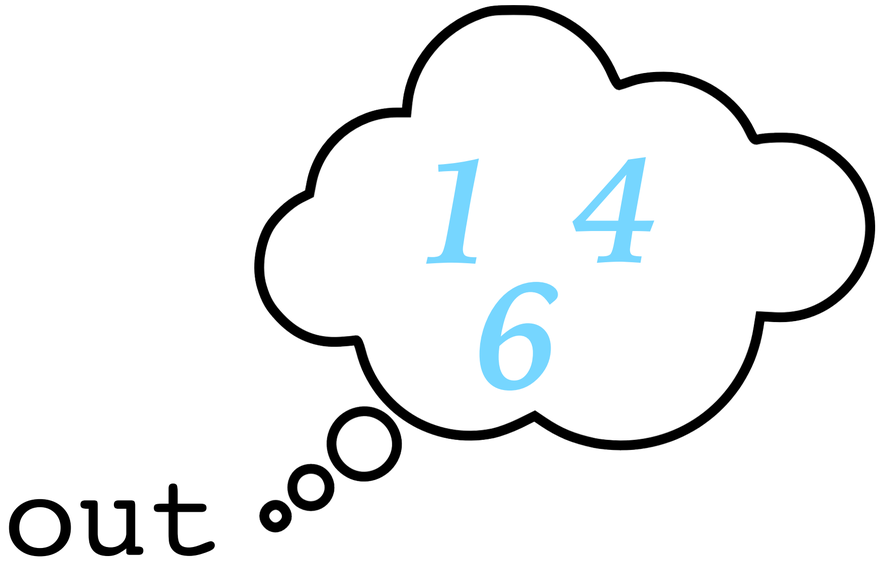


The set of all cycles is the influence set



… our notion of timing

Different influence sets = not constant time



For any two executions …

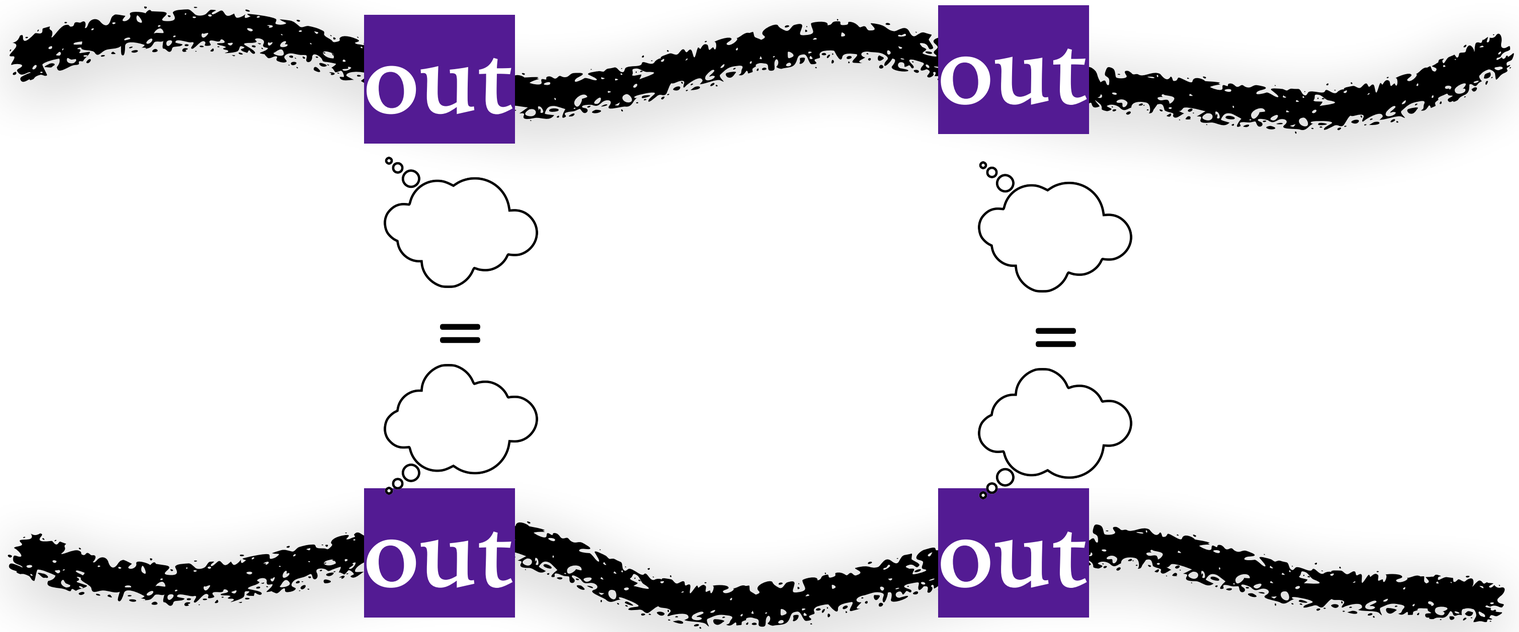
… regardless of secrets

Timing of outputs must be the same

Timing of outputs must be the same

all outputs, produce same influence sets

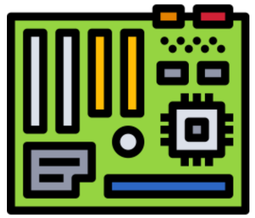
Timing of outputs must be the same



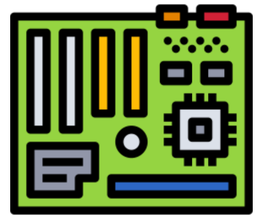
all outputs, produce same influence sets

Find two executions s.t. outputs have different influence sets

Two Executions



take *Fast Path*



take *Slow Path*

Fast Path: `x=0` and `y=1`

Fast Path: Values

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	0	1	⊥	⊥

Fast Path: Values

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	0	1	⊥	⊥
1	0	1	⊥	0

Fast Path: Values

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end
```

```
always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

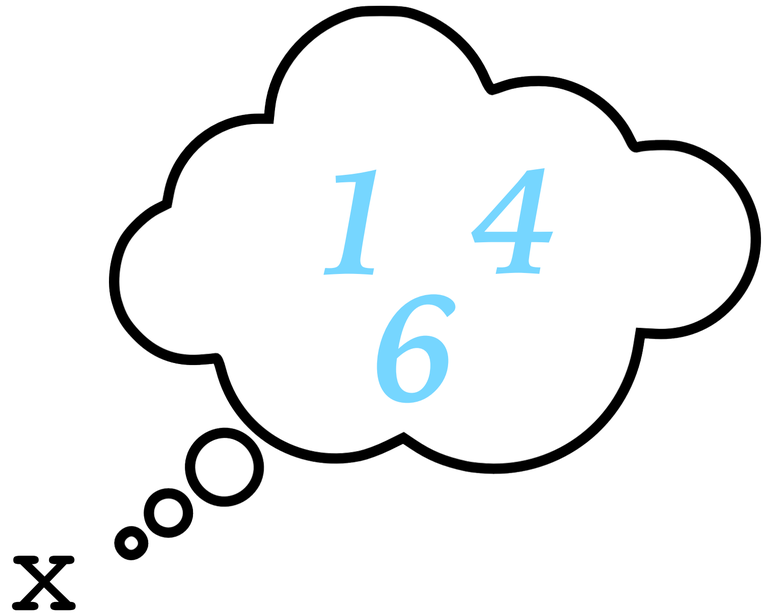
cycle	x	y	flp_res	out
0	0	1	⊥	⊥
1	0	1	⊥	0
…				
k-1	0	1	0	0

Fast Path: Values

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	0	1	⊥	⊥
1	0	1	⊥	0
…				
k-1	0	1	0	0
k	0	1	0	0

Influence Sets



Fast Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅

Fast Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}

Fast Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end
```

```
always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}

Fast Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}
k	{k}	{k}	{1}	{k-1}

Slow Path: $x=1$ and $y=1$

Slow Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅

Slow Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}

Slow Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end
```

```
always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}

Slow Path: Influence Sets

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	{0}	{0}	∅	∅
1	{1}	{1}	∅	{0}
…				
k-1	{k-1}	{k-1}	{0}	{k-2}
k	{k}	{k}	{1}	{0,k-1}

Fast Path vs. Slow Path

cycle	x	y	flp_res	out	
0	{0}	{0}	∅	∅	✓
1	{1}	{1}	∅	{0}	✓
…					
k-1	{k-1}	{k-1}	{0}	{k-2}	✓
k	{k}	{k}	{1}	{k-1}	✗

cycle	x	y	flp_res	out	
0	{0}	{0}	∅	∅	✓
1	{1}	{1}	∅	{0}	✓
…					
k-1	{k-1}	{k-1}	{0}	{k-2}	✓
k	{k}	{k}	{1}	{0,k-1}	✗

Different influence sets for `out`! Not constant time!

Outline

1. Definition
2. Verification: Iodine 
3. Evaluation

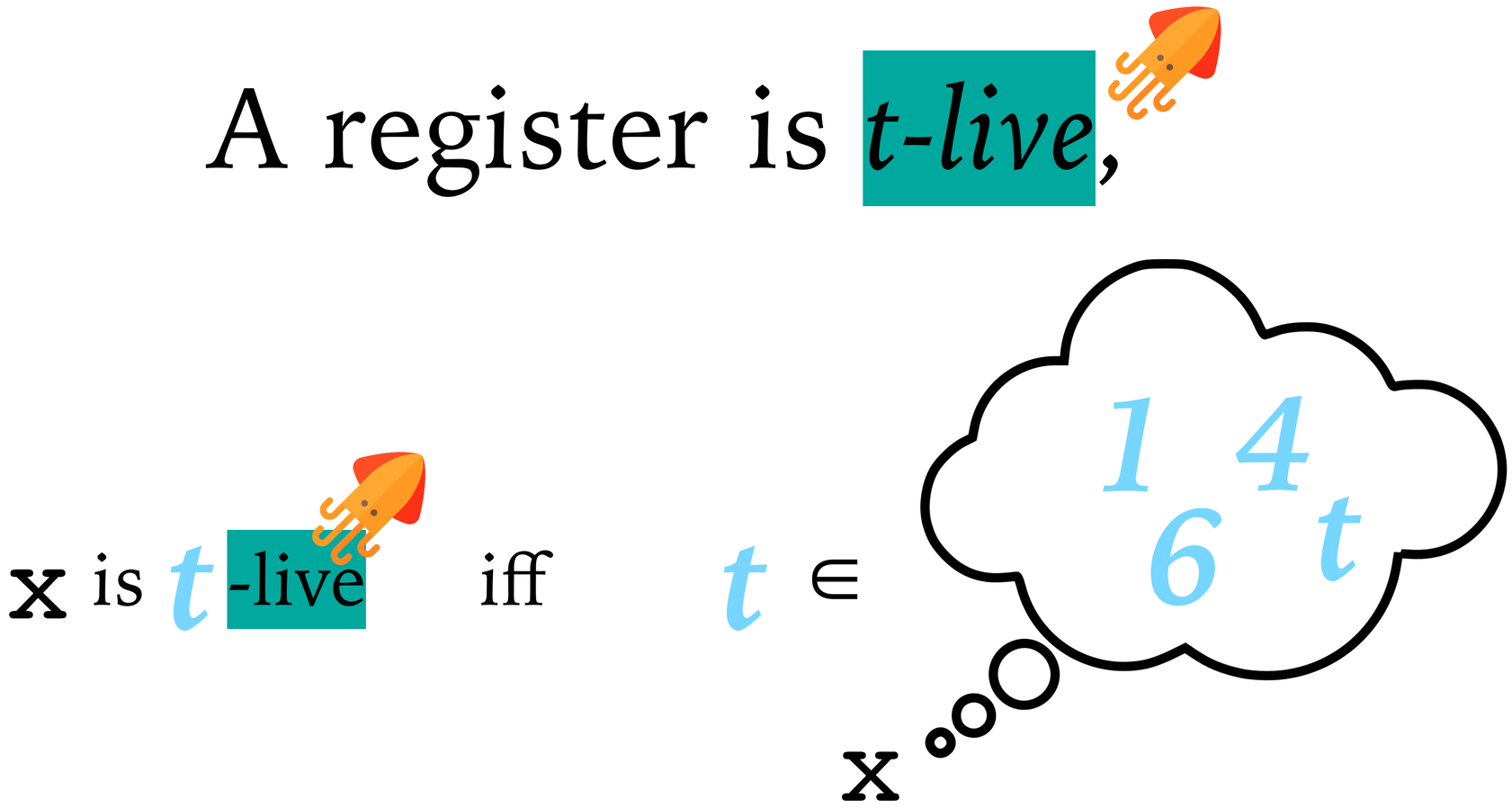
Equivalent:

Liveness Equivalence

A register is *t-live*, 

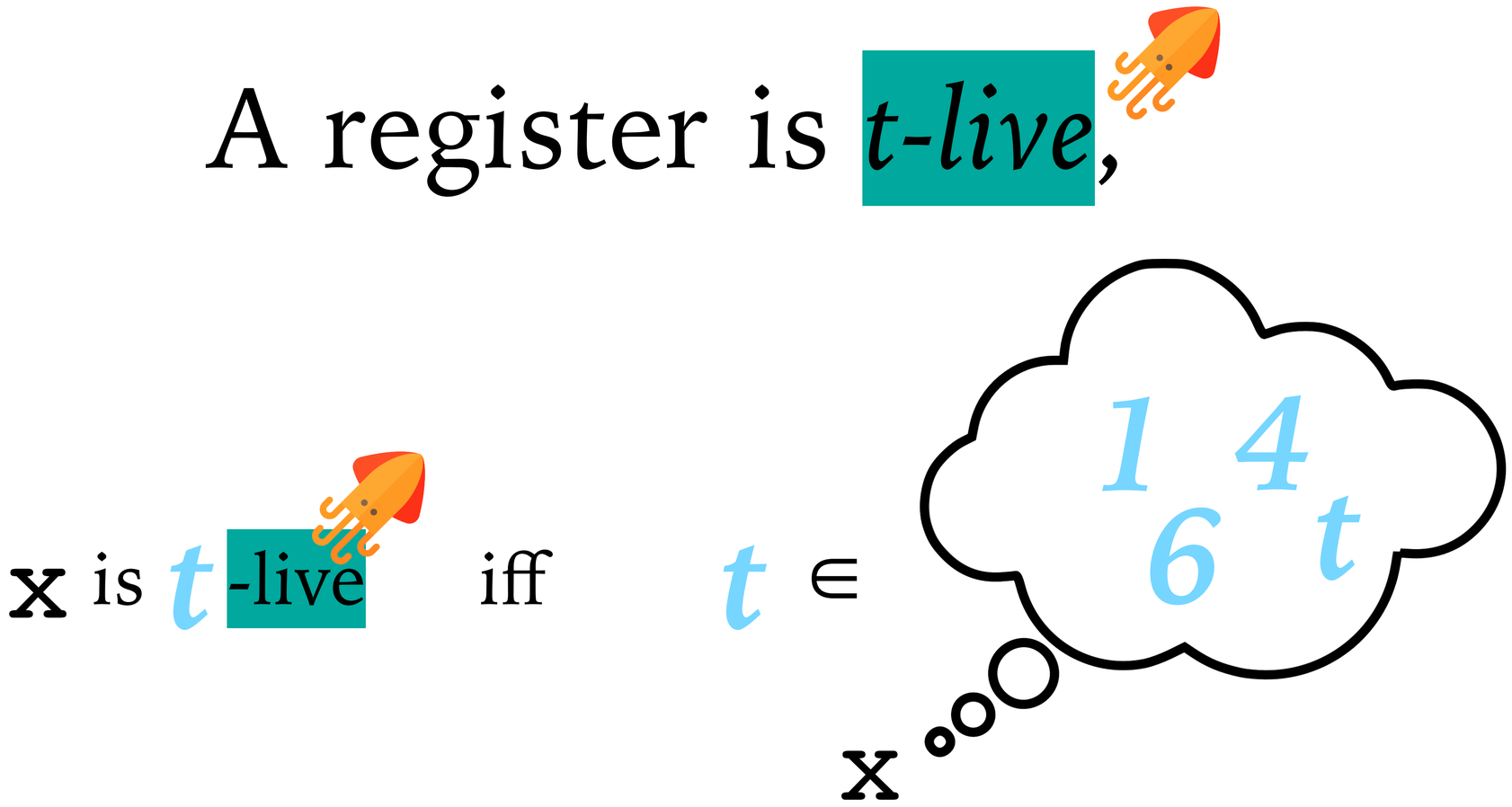
…if it's influenced by inputs from cycle t

A register is *t-live*,



…if it's influenced by inputs from cycle t

A register is *t-live*,

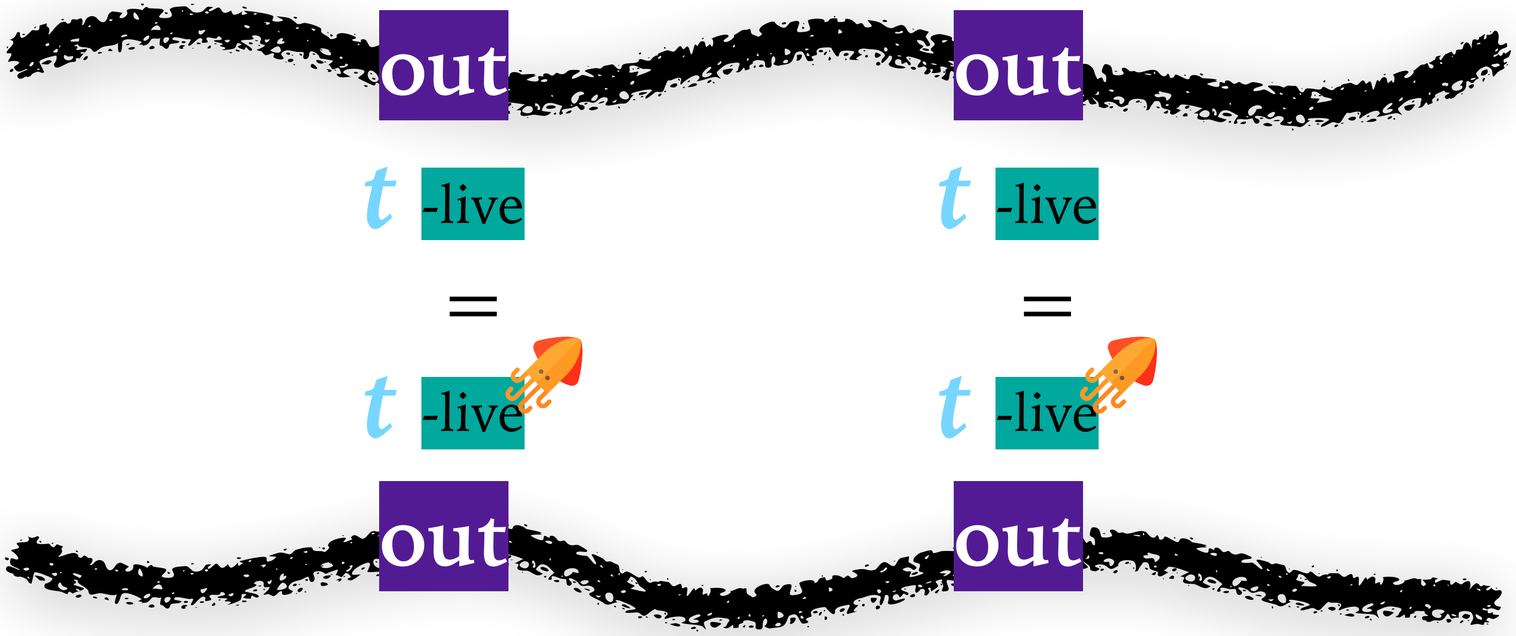


… i.e., if t is in its influence set

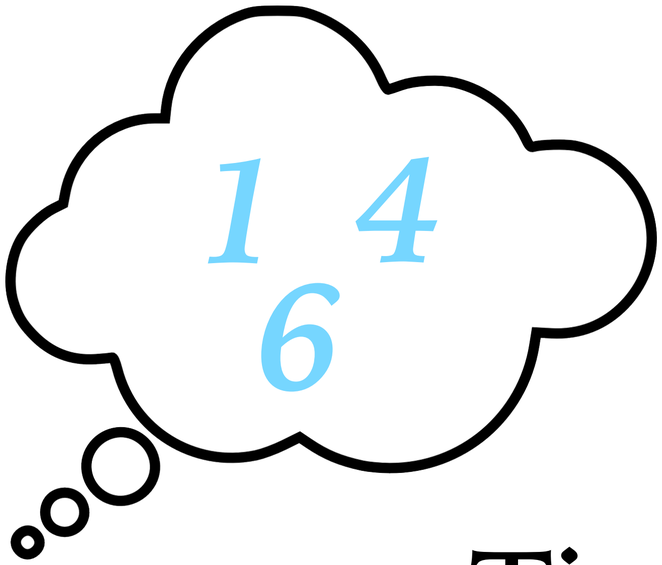
Two executions are *liveness-equivalent*, if …

Two executions are *liveness-equivalent*, if …

Two executions are *liveness-equivalent*, if …



all outputs are *t-live* equivalent, for all t



Constant Time

iff

Liveness Equivalence

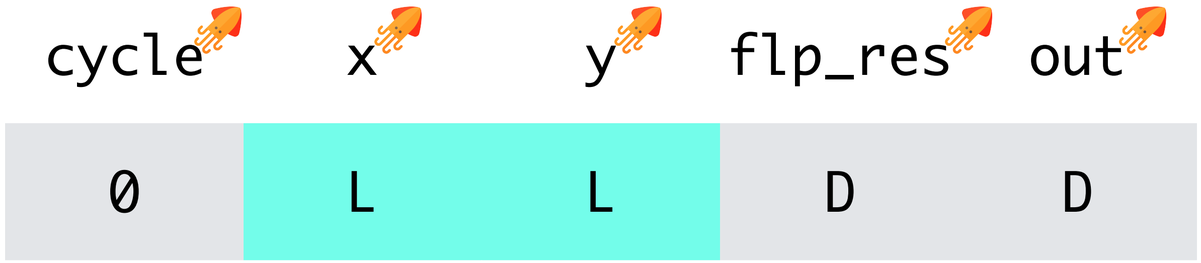
Show that the multiplier is

not constant time

(using liveness)

Fast Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```



Fast Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	L	L	D	D
1	D	D	D	L

Fast Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end
```

```
always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	L	L	D	D
1	D	D	D	L
…				
k-1	D	D	L	D

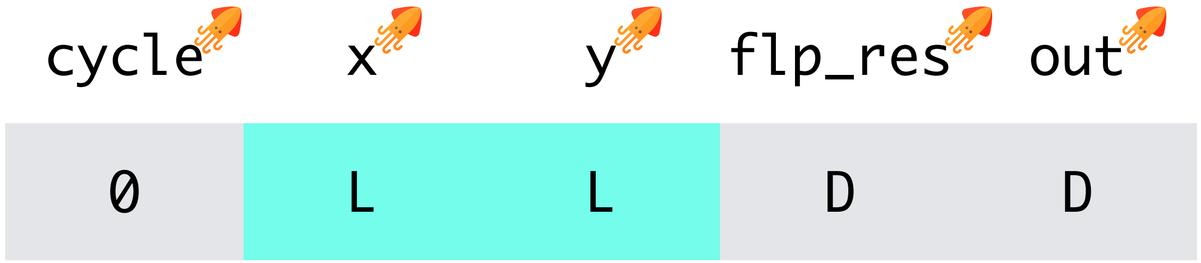
Fast Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	L	L	D	D
1	D	D	D	L
…				
k-1	D	D	L	D
k	D	D	D	D

Slow Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```



Slow Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	L	L	D	D
1	D	D	D	L

Slow Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end
```

```
always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	L	L	D	D
1	D	D	D	L
…				
k-1	D	D	L	D

Slow Path: Liveness

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
    if (iszero)
        out <= 0;
    else
        out <= flp_res;
end

always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

cycle	x	y	flp_res	out
0	L	L	D	D
1	D	D	D	L
…				
k-1	D	D	L	D
k	D	D	D	L

Fast Path vs. Slow Path

cycle	x	y	flp_res	out	
0	L	L	D	D	✓
1	D	D	D	L	✓
…					
k-1	D	D	L	D	✓
k	D	D	D	D	✗

cycle	x	y	flp_res	out	
0	L	L	D	D	✓
1	D	D	D	L	✓
…					
k-1	D	D	L	D	✓
k	D	D	D	L	✗

Let’s fix the FPU …

```
assign iszero = (x==0 || y==0);

always @(posedge clk) begin
     if (iszero)
       out <= 0;
      else
       out <= flp_res;
end
always @(posedge clk) begin
     flp_res <= ... //compute x*y
end
```

Let’s fix the FPU …

… by adding a constant-time mode

(like ARM DIT)

…if `ct` is set, always take the *slow path*

```
assign iszero = (x==0 || y==0);
```

```
always @(posedge clk) begin
    if (ct)
        out <= flp_res;
    else
        if (iszero)
          out <= 0;
        else
          out <= flp_res;
end
always @(posedge clk) begin
    flp_res <= ... //compute x*y
end
```

Verify constant time, if `ct` is set

For register x, we introduce liveness bit $x^{\bullet}$

```
always @(posedge clk) begin
  if (ct)
    assign iszero = (x==0 || y==0);
  if (iszero)
    out <= 0;
  else
    out <= flp_res;
end
...
```

For register x, we introduce liveness bit x•

```
assign iszero = (x==0 || y==0);
assign iszero• = (x• ∨ y•);
```

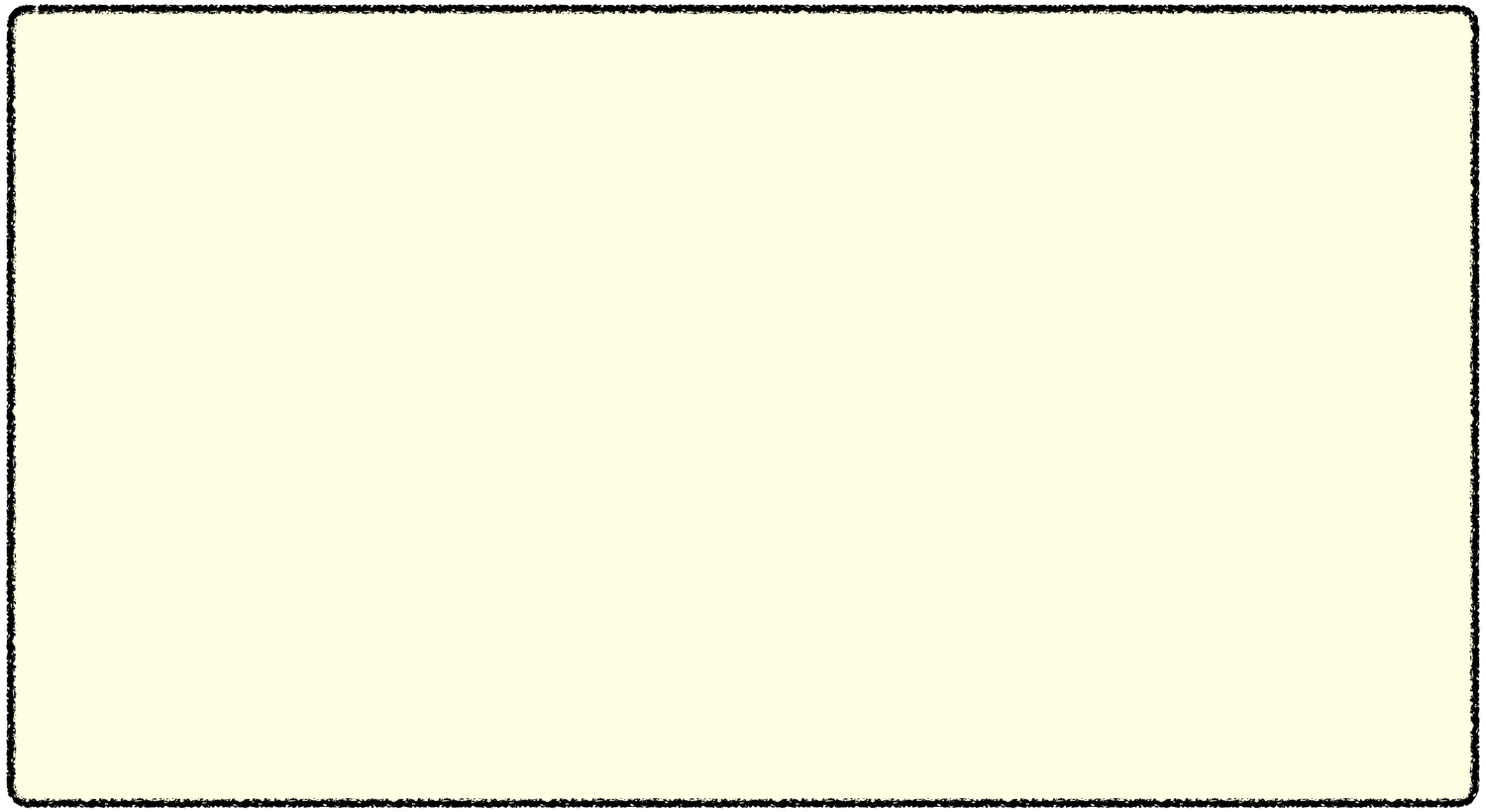
For register x, we introduce liveness bit x•

```
assign iszero = (x==0 || y==0);
assign iszero• = (x• ∨ y•);

always @(posedge clk) begin
    if (ct)
        out <= flp_res;
        out• = (flp_res• ∨ ct•);
    else
        if (iszero)
          out <= 0;
           out• = (flp_res• ∨ ct• ∨ iszero•);
        ...
```

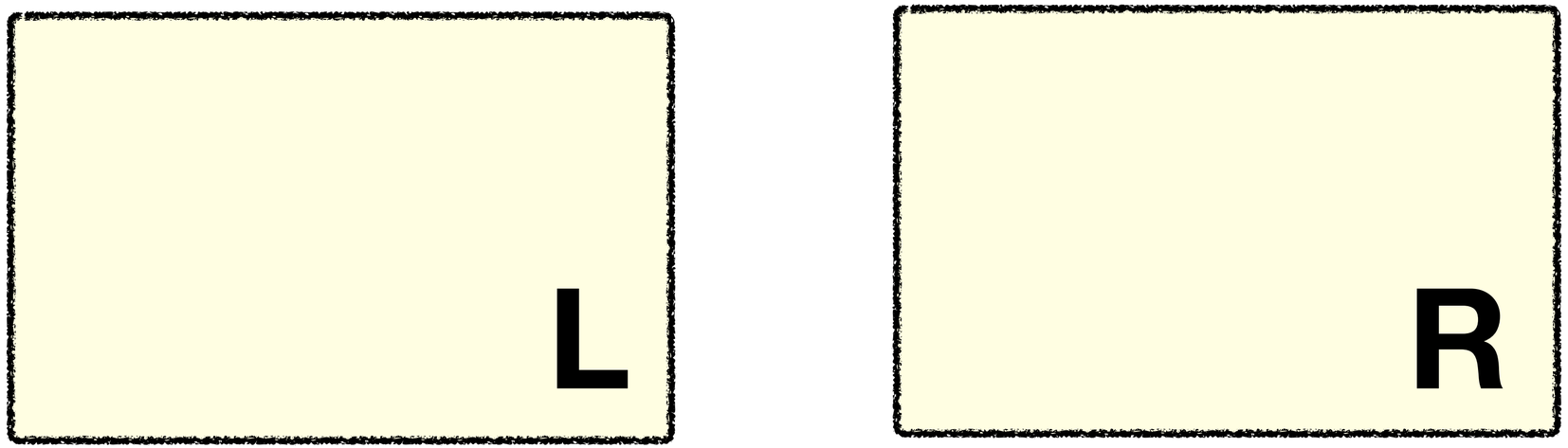
Make two copies

```
assign iszero = (x==0 || y==0);
assign iszero• = (x• ∨ y•);

always @(posedge clk) begin
    if (ct)
        out <= flp_res;
        out• = (flp_res• ∨ ct•);
    else
        if (iszero)
          out <= 0;
           out• = (flp_res• ∨ ct• ∨ iszero•);
        ...
```

Make two copies



Make two copies

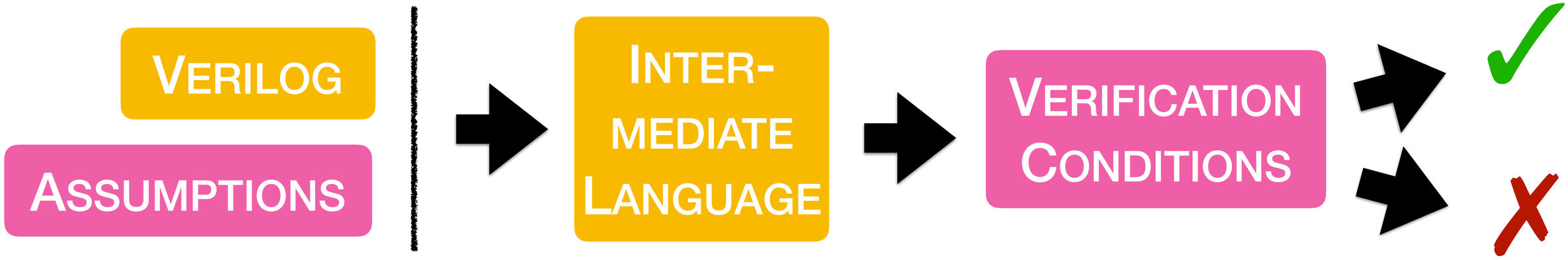


Verify $\mathrm{out}\bullet_L = \mathrm{out}\bullet_R$, for an arbitrary t

Overview

1. Definition
2. Verification: Iodine 
3. Evaluation

Iodine: Architecture



… finds proofs automatically!

Iodine: Architecture

- What can Iodine verify?
- What is the *annotation burden*?
- How *efficient* is Iodine?

What can Iodine verify?

✓ 472 MIPS

✓ Yarvi RISC-V

Simple CPUs

✓ Single precision FPU

✗ IEEE 754 FPU

✓ TIS-CT ALU

ALU/FPU

✓ Opencores Shacore SHA-256

✗ Fatestudio RSA 4096 RSA

Crypto Cores

Iodine: Architecture

- What can Iodine verify?
- What is the annotation burden?
- How efficient is Iodine?

Don’t have to prove constant time unconditionally

Public

Flushing

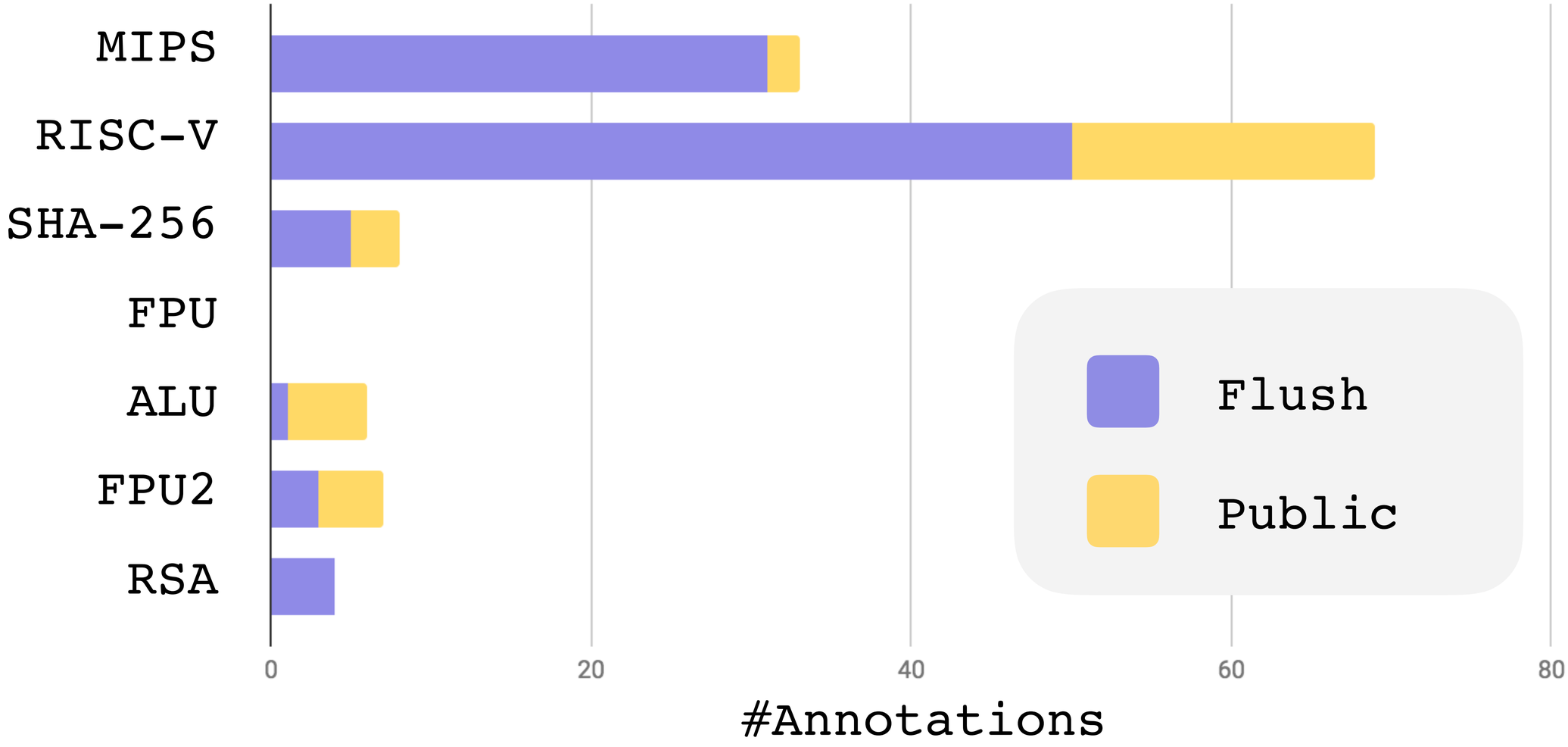
… but can rely on assumptions

Assume that registers are public, i.e., *free of secrets*

CPUs

- Instructions *are public*
- Memory access pattern *are public*
- Reset bits *are public*

Evaluation: Annotations

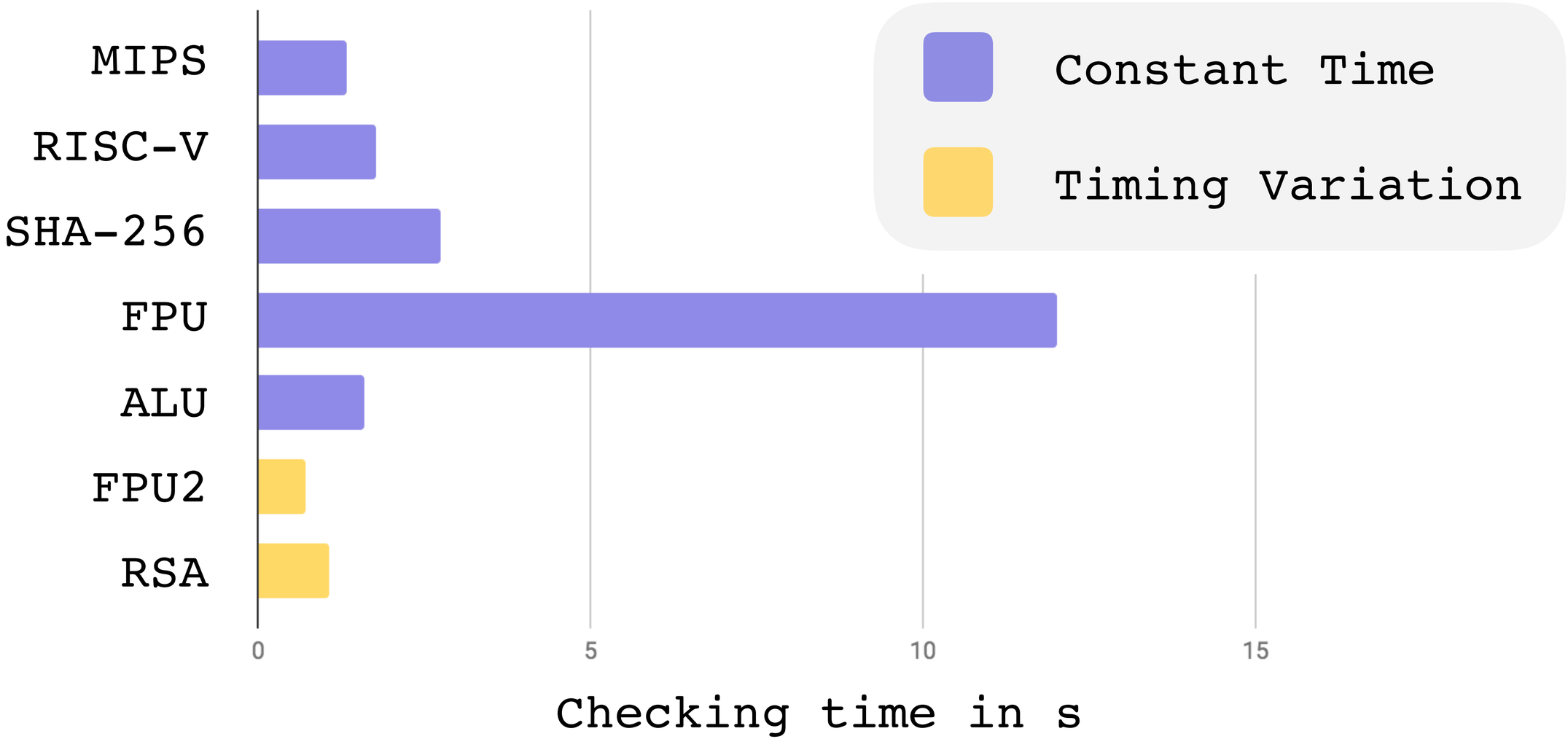


Recent Work:
Infer annotations, automatically

Iodine: Architecture

- Can we applicable is Iodine?
- What is the annotation burden?
- How efficient is Iodine?

How *efficient* is Iodine?



Summary

1. Definition
2. Verification: Iodine 
3. Evaluation

https://iodine.programming.systems

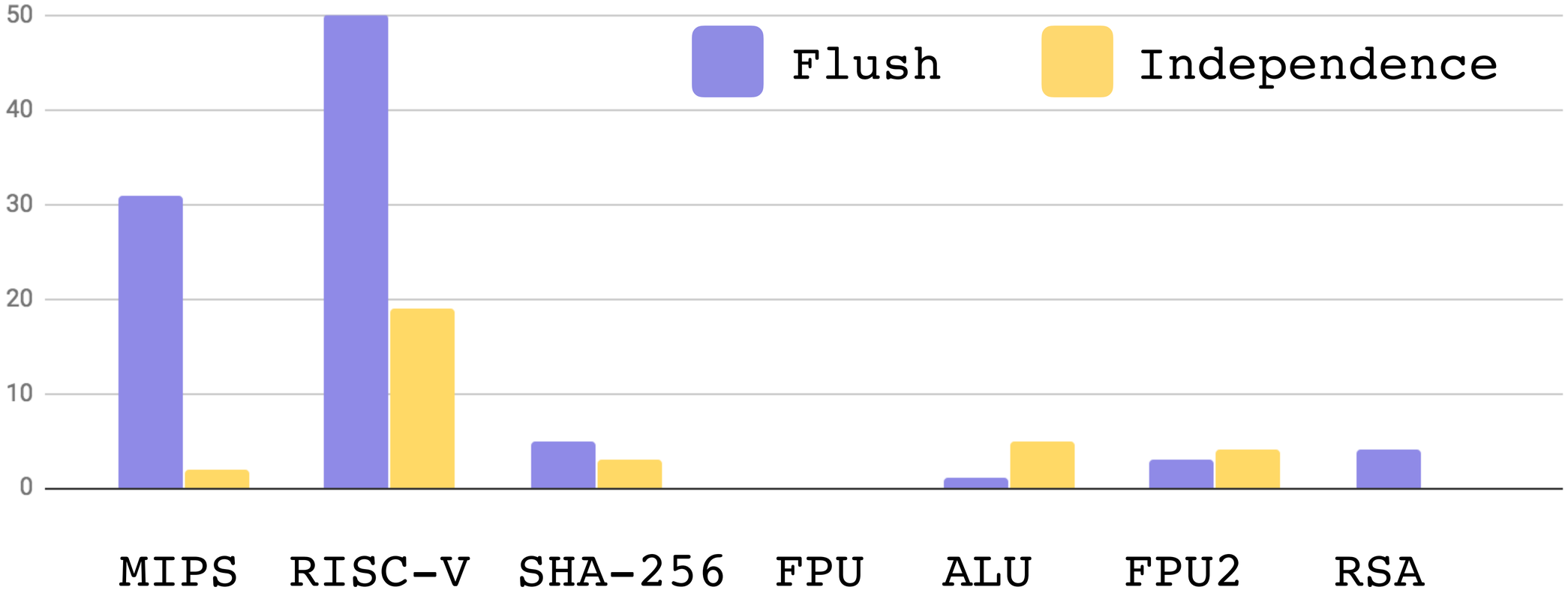
BACKUP

Evaluation: Table

Name	#LOC	#Flush	#Ind	CT	Check (s)
MIPS	434	31	2	✓	1.329
RISC-V	745	50	19	✓	1.787
SHA-256	651	5	3	✓	2.739
FPU	1182	0	0	✓	12.013
ALU	913	1	5	✓	1.595
FPU2	272	3	4	✗	0.705
RSA	870	4	0	✗	1.061
Total	5067	94	33	-	21.163

Leftovers

Evaluation: Annotations



Evaluation: Annotations