Unifying Static and Dynamic Intermediate Languages for Accelerator Generators

Caleb Kim*, Pai Li***,** Anshuman Mohan, Andrew Butt,

Adrian Sampson, Rachit Nigam

“Unifying Static and Dynamic Intermediate Languages for Accelerator Generators”

“Unifying Static and Dynamic Intermediate Languages for **Accelerator Generators**”

- **Compiler** for hardware accelerator design

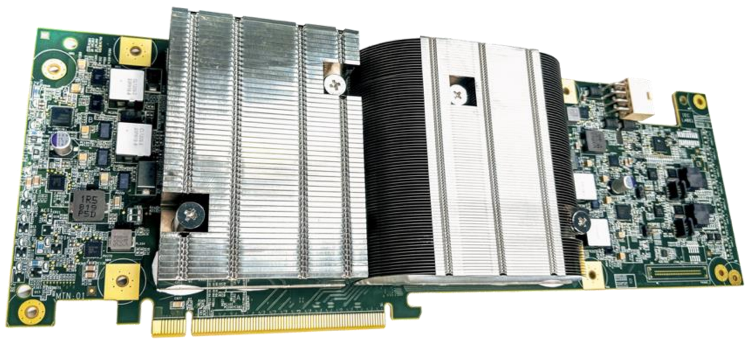
“Unifying **Static and Dynamic Intermediate Languages** for Accelerator Generators”

- ILs have two paradigms (**static** and **dynamic**) with expressiveness/efficiency trade-offs

“**Unifying** Static and Dynamic Intermediate Languages for Accelerator Generators”

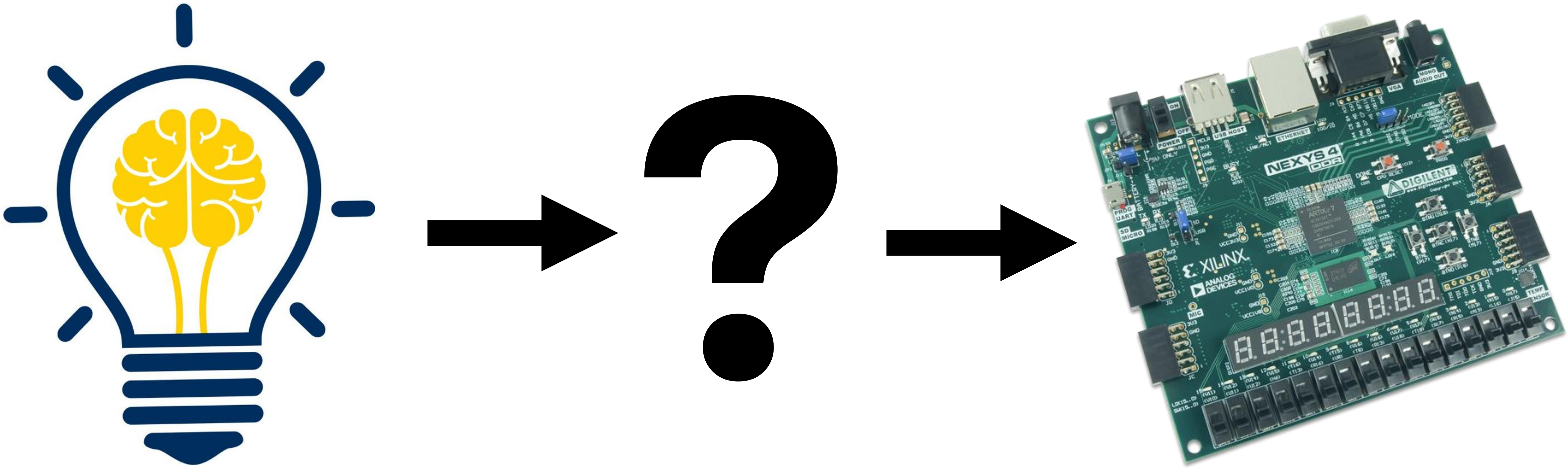
- We apply **semantic refinement** to fluidly **unify** these two styles, allowing us to build compilers that can **do things that neither style can do alone**.

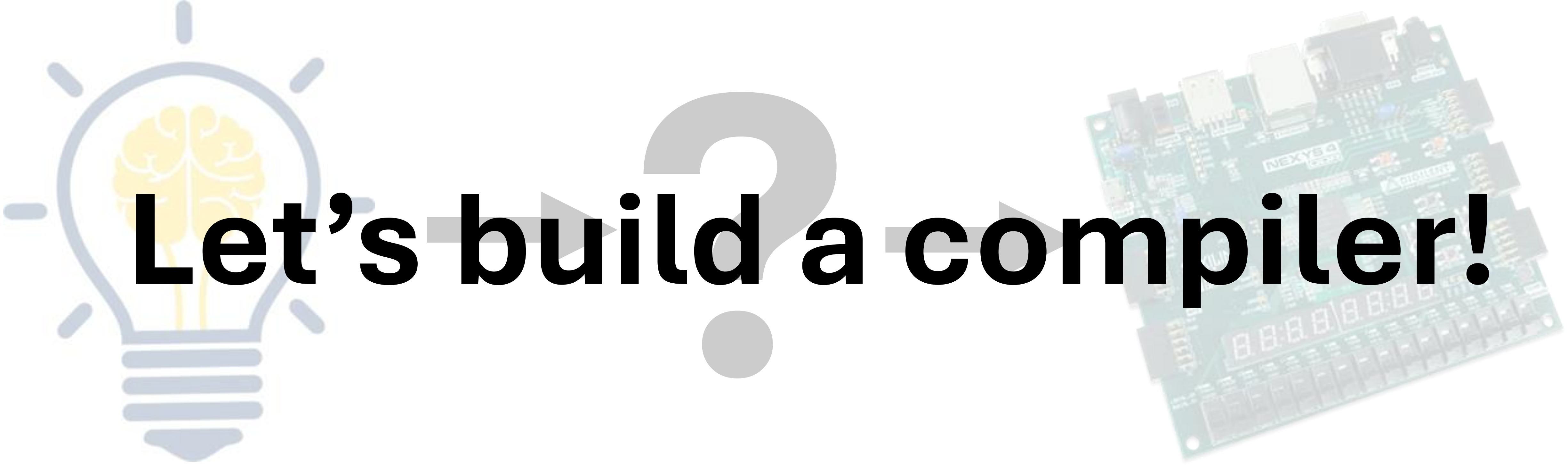
Google
Tensor Processing Unit



Google
Video Coding Unit

Microsoft
Catapult


?
NEXYS 4 DDR
DIGILENT
XILINX
ANALOG DEVICES

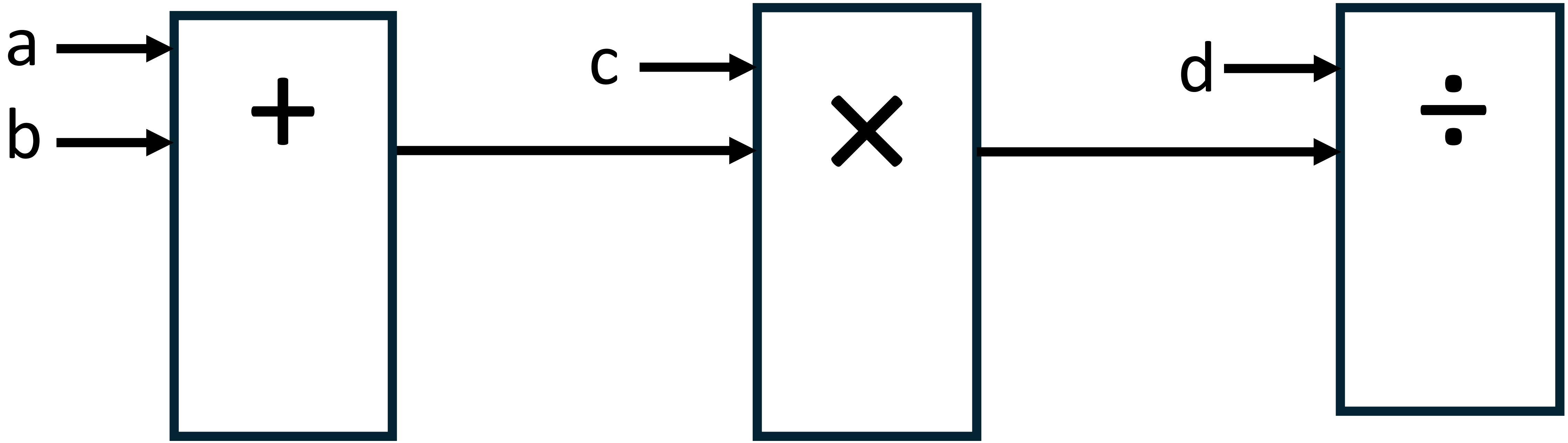


Let’s build a compiler!

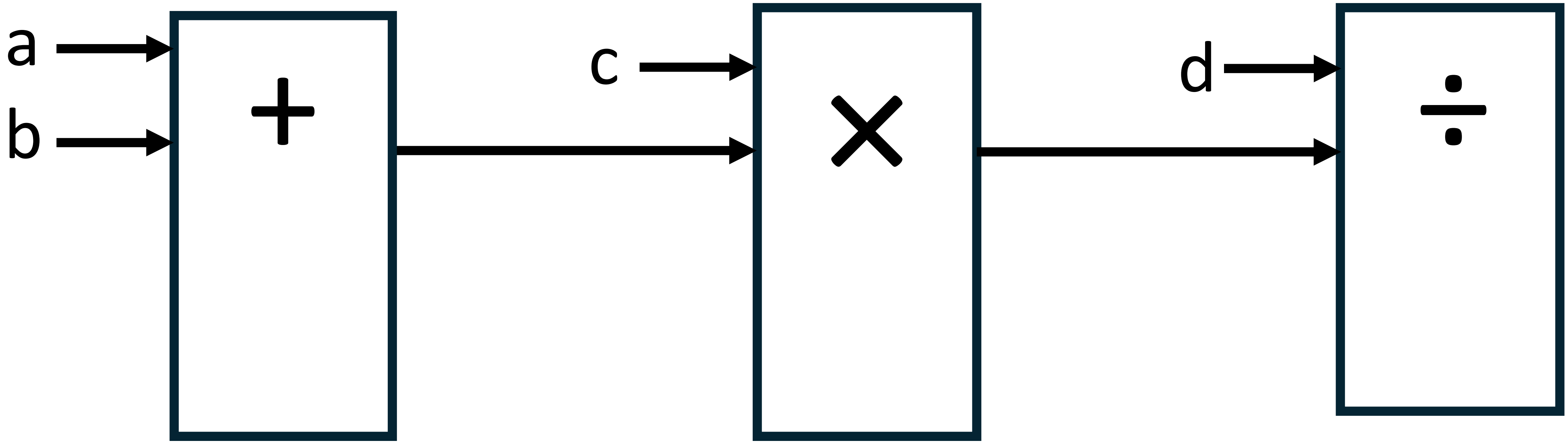
Toy Language: (simple) math expressions on integers

$$((a+b)\times c)\div d$$

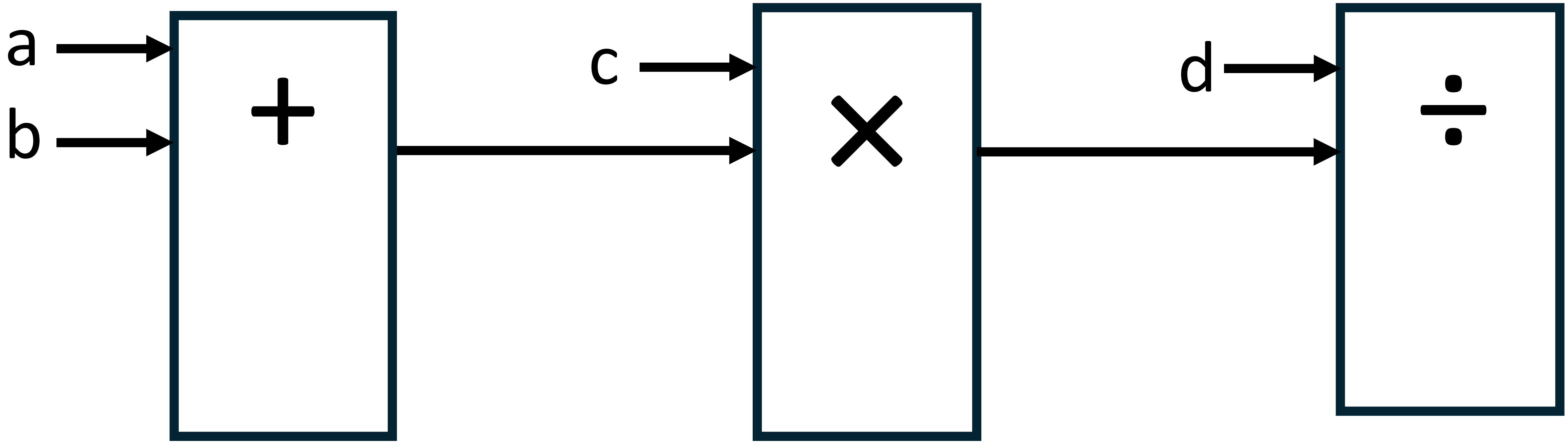
$((a+b)\times c)\div d$



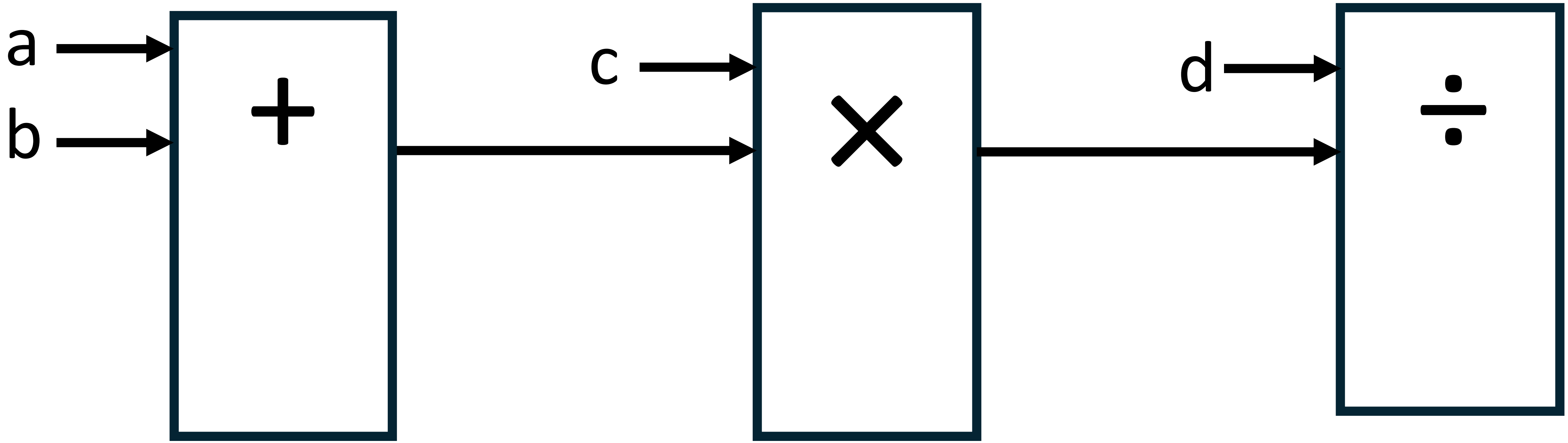
Structure (✓)



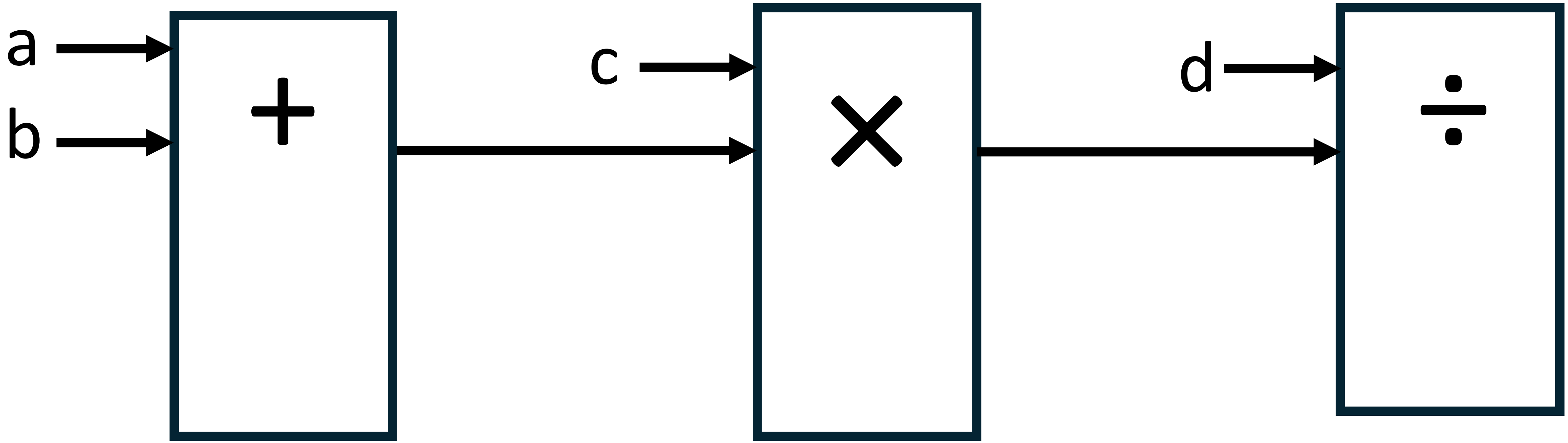
Structure (✓) Timing (X)



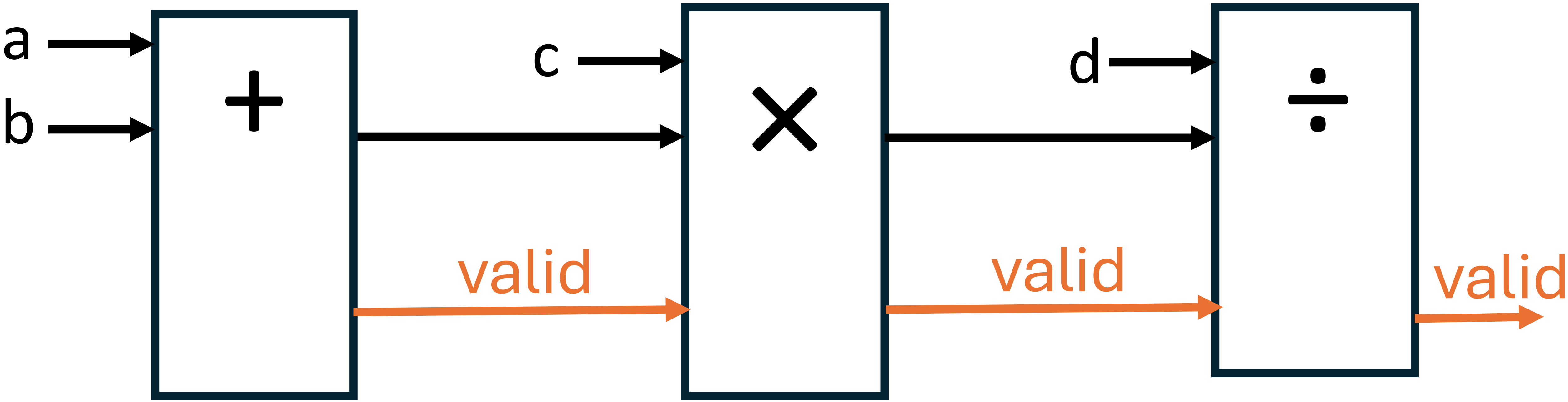
Timing



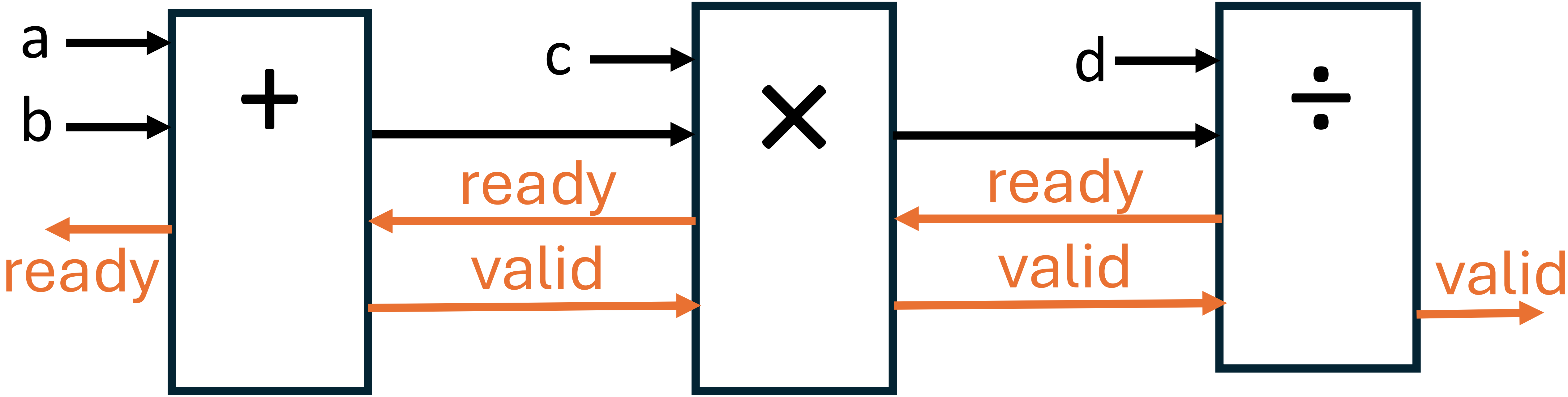
Dynamic (Latency-Insensitive) Interfaces



Dynamic (Latency-Insensitive) Interfaces

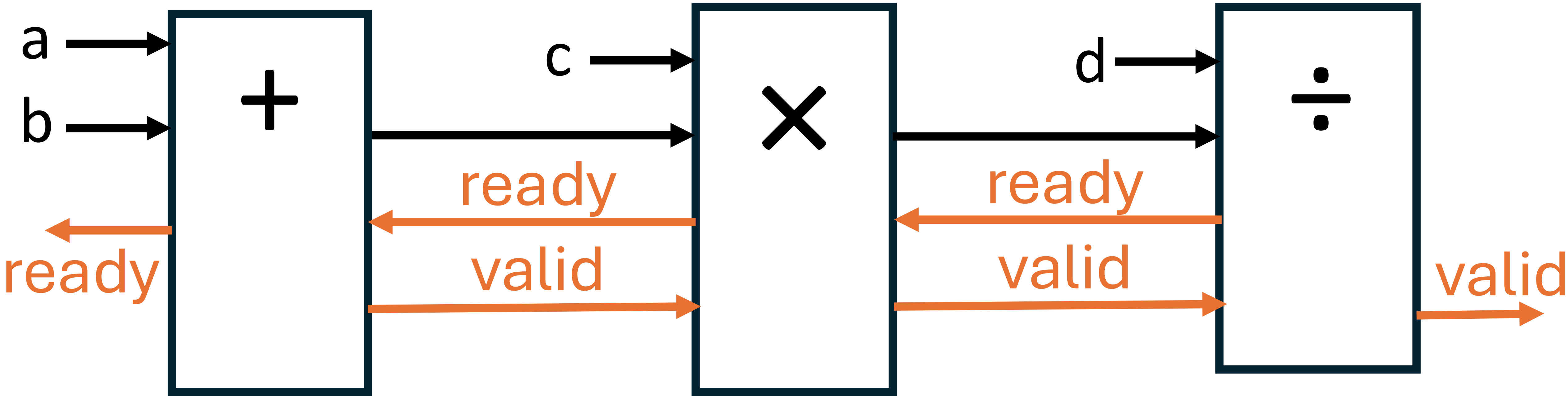


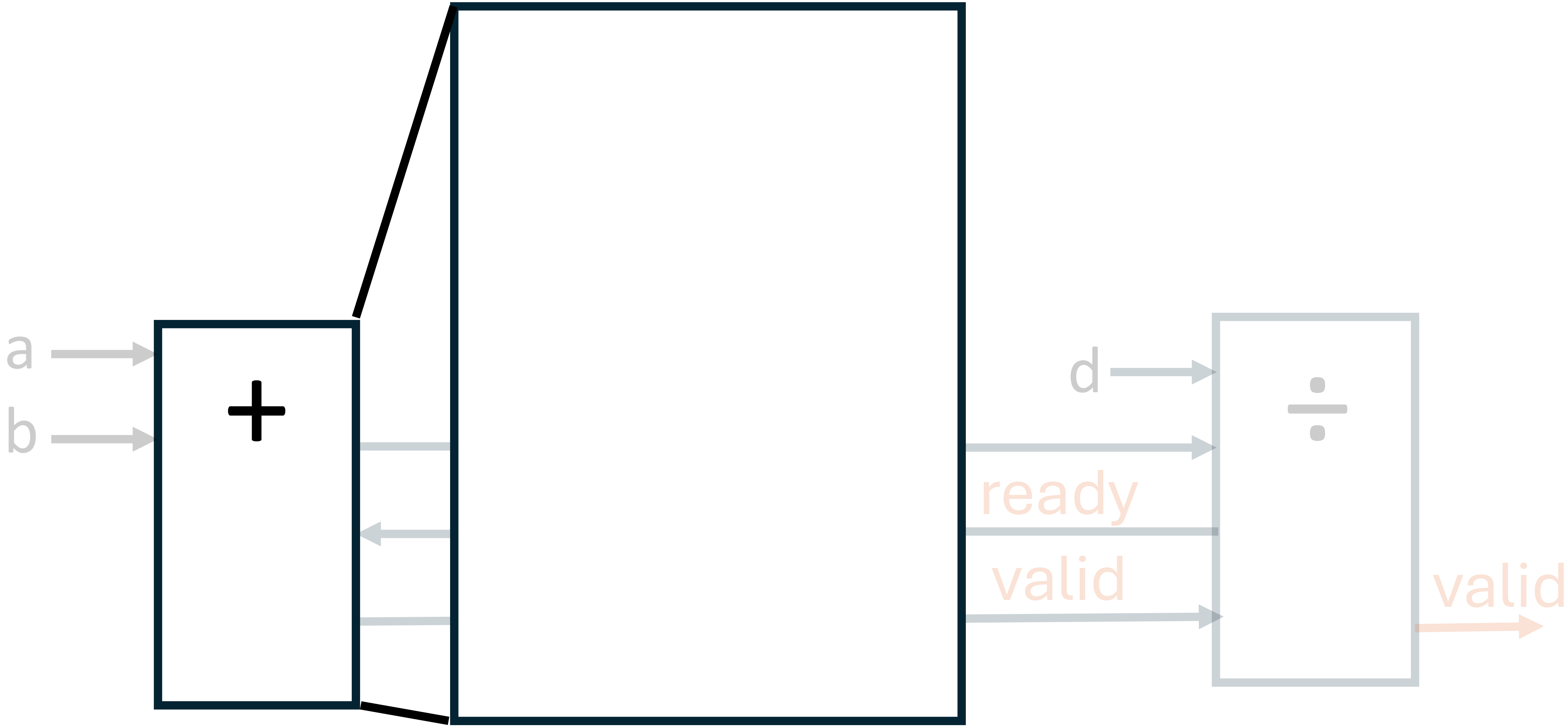
Dynamic (Latency-Insensitive) Interfaces

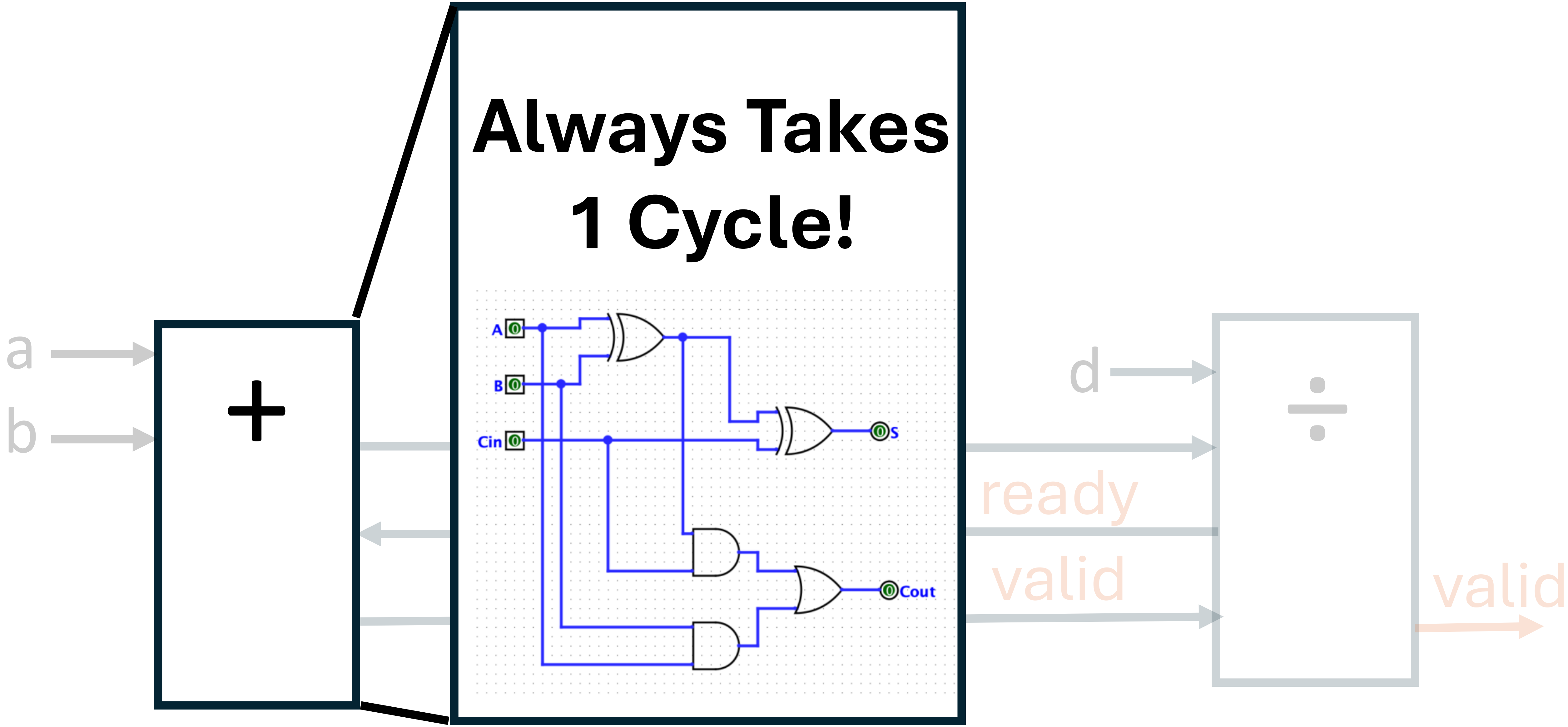


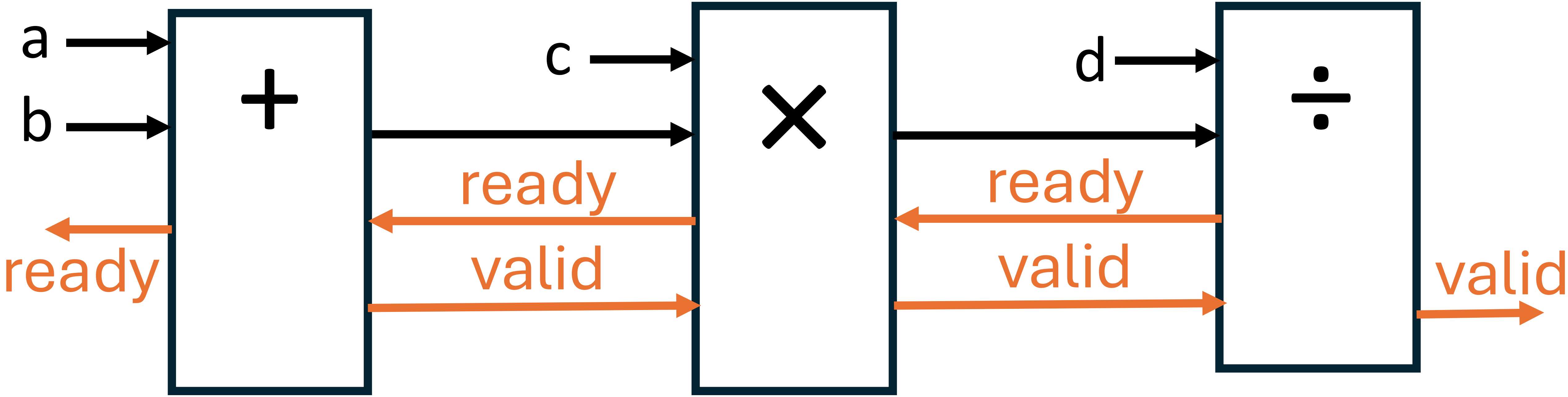
Dynamic (Latency-Insensitive) Interfaces

Simple, Modular

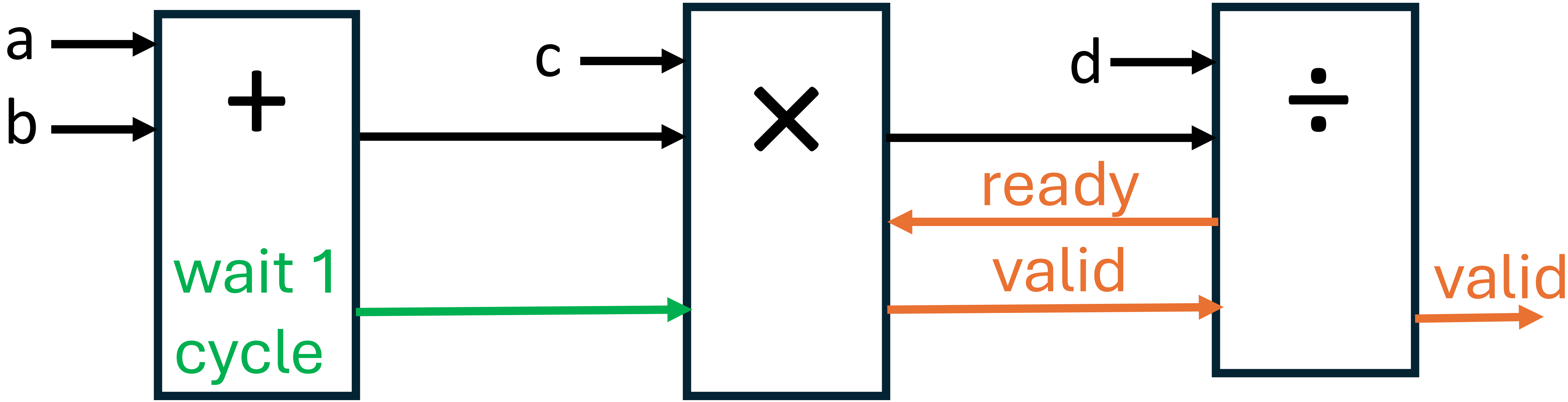



a
b
+
d
÷
ready
valid
valid


a
b
+
Always Takes
1 Cycle!
A
B
Cin
S
Cout
d
÷
ready
valid
valid

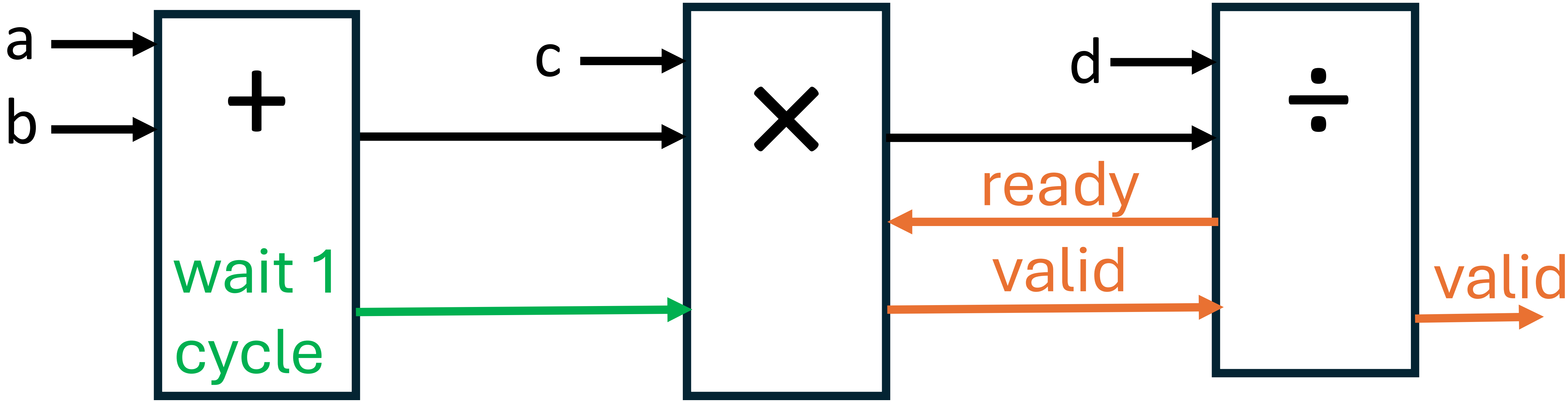

a
b
+
c
×
d
÷
ready
valid
ready
valid
ready
valid

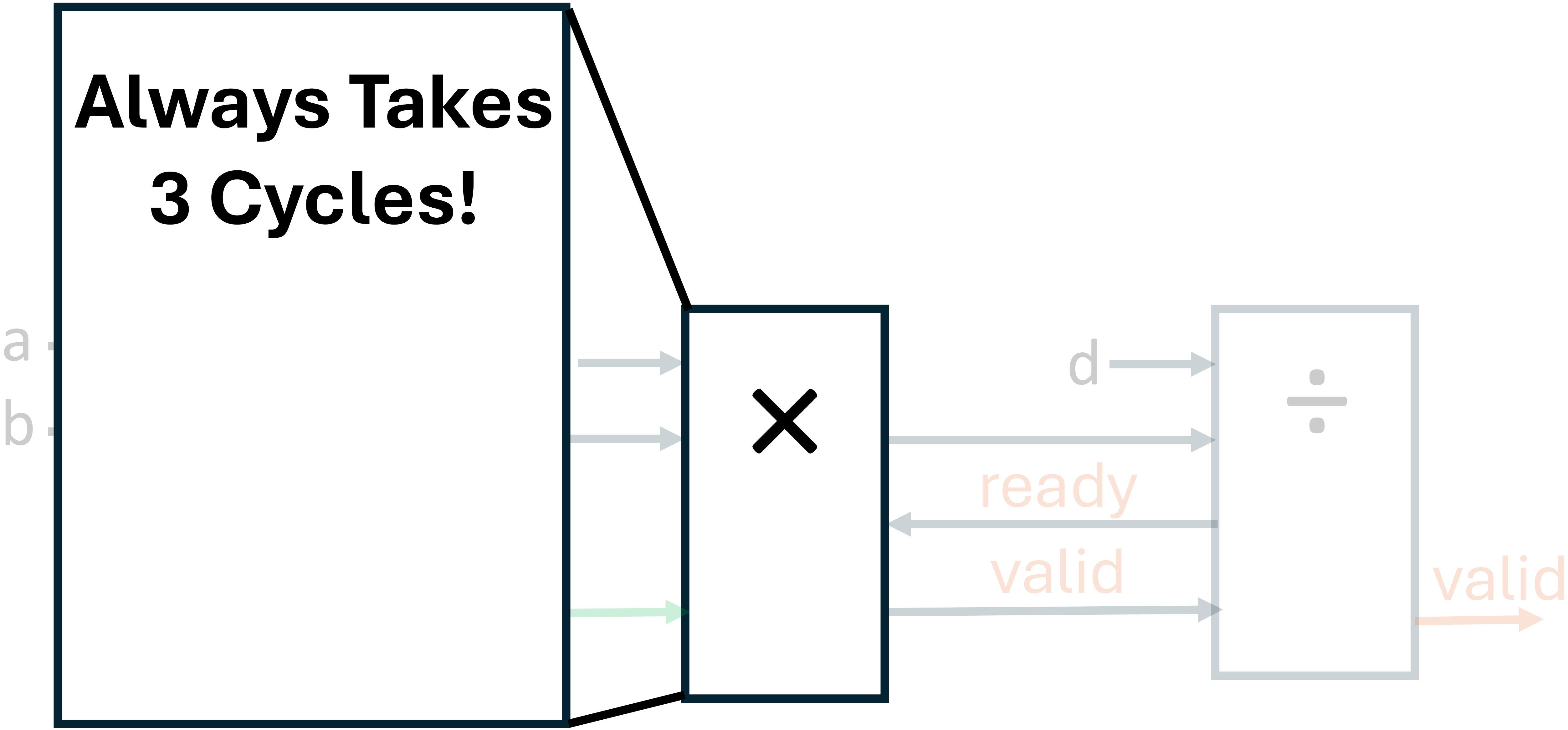
Static (Latency-Sensitive) Interfaces



Static (Latency-Sensitive) Interfaces

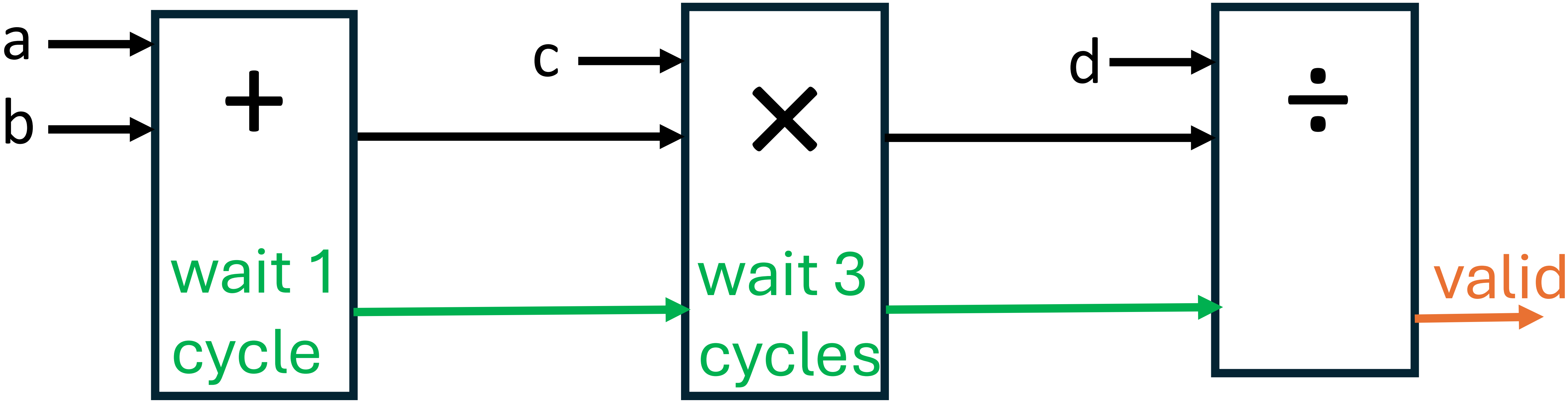
Eliminate ready/valid wiring, ↑ efficiency

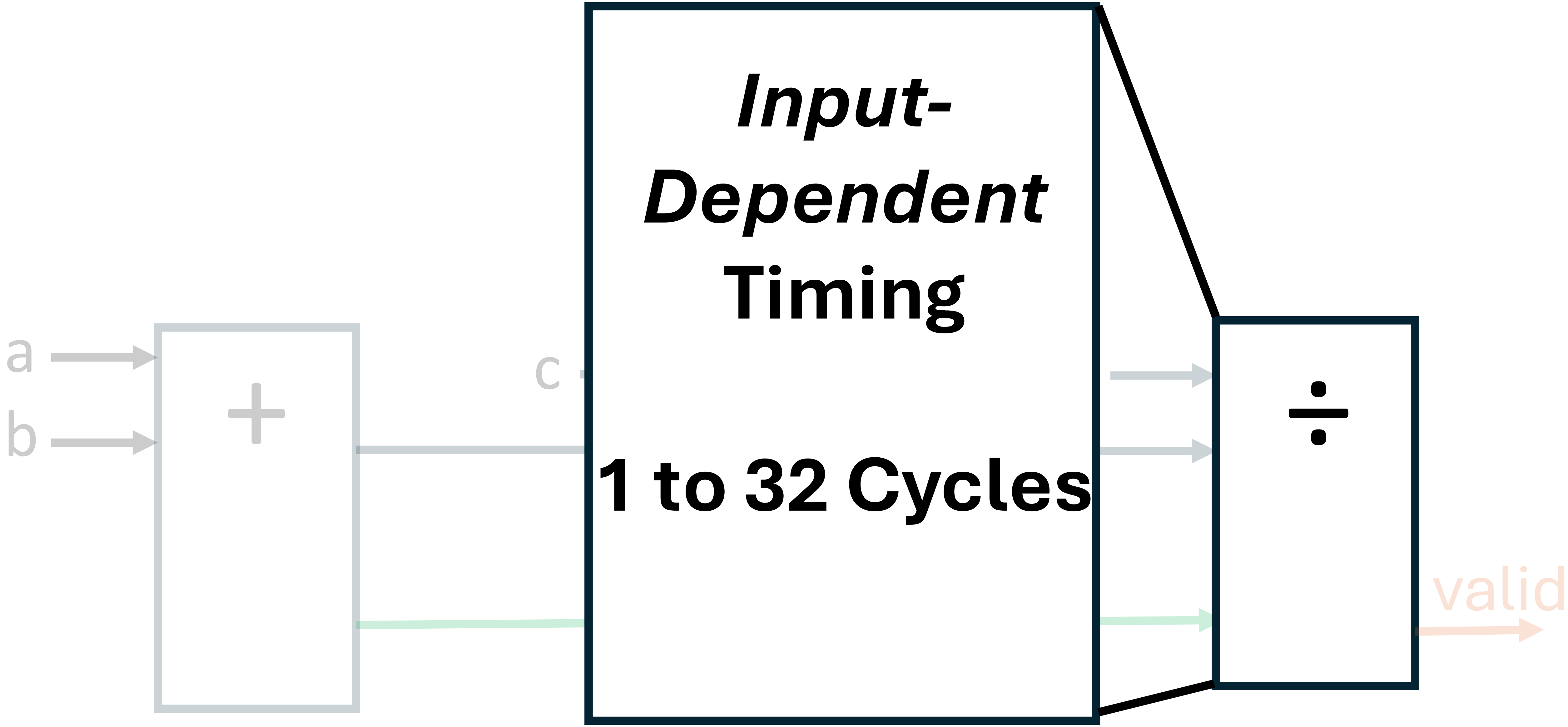


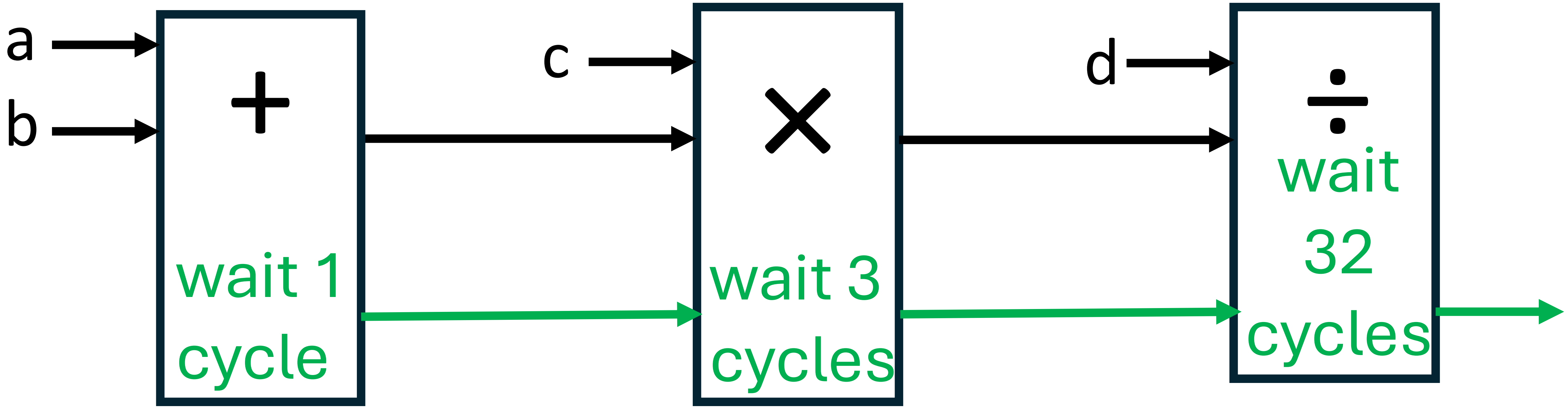


Always Takes
3 Cycles!
a
b
×
d
÷
ready
valid
valid

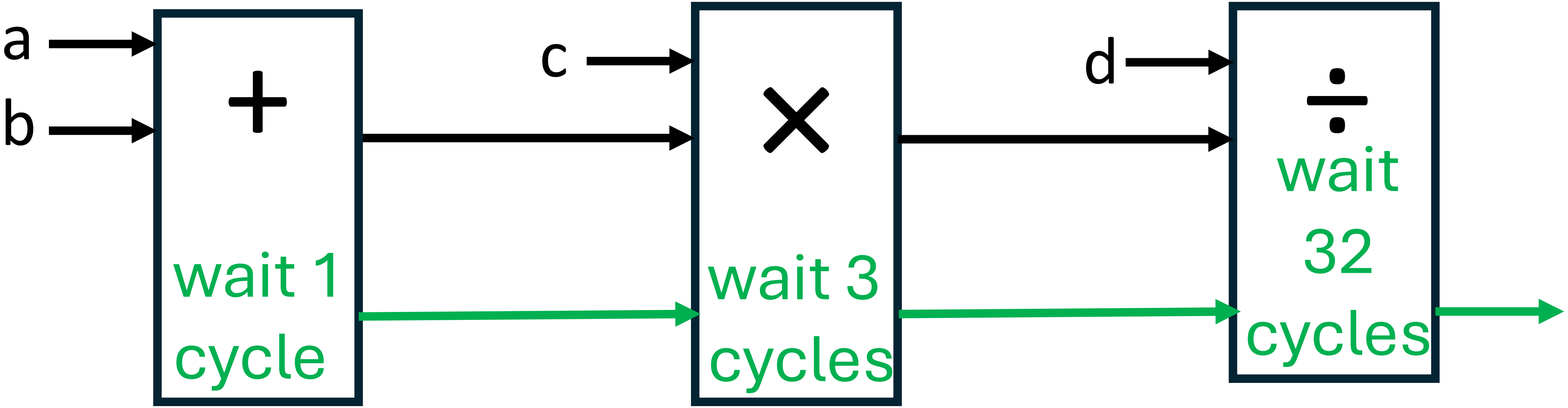
Static (Latency-Sensitive) Interfaces

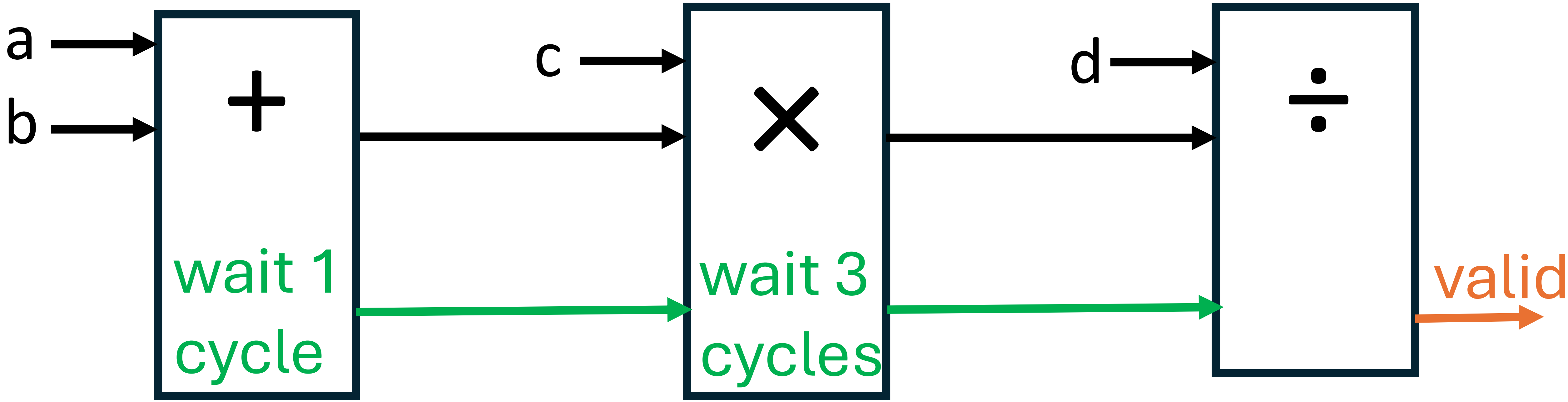



Input-Dependent Timing
1 to 32 Cycles
a
b
+
c
÷
valid


a
b
+
wait 1 cycle
c
×
wait 3 cycles
d
÷
wait 32 cycles

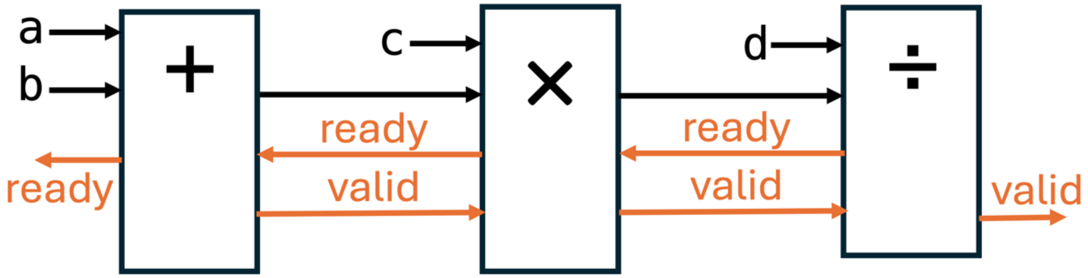
Too Conservative, Wastes Cycles




a
b
+
wait 1
cycle
c
×
wait 3
cycles
d
÷
valid

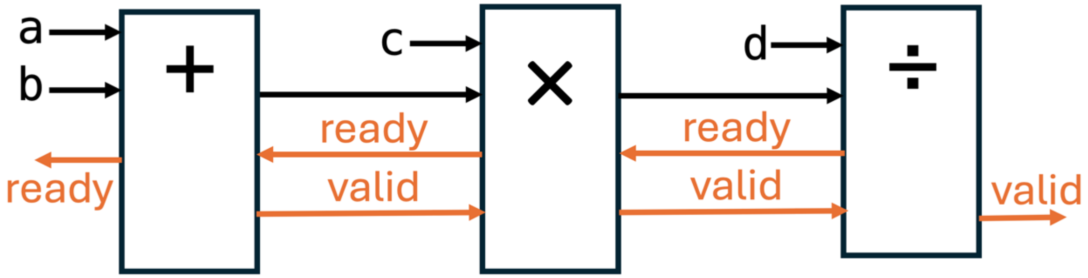
Recap: So Many Choices!

Recap: So Many Choices!



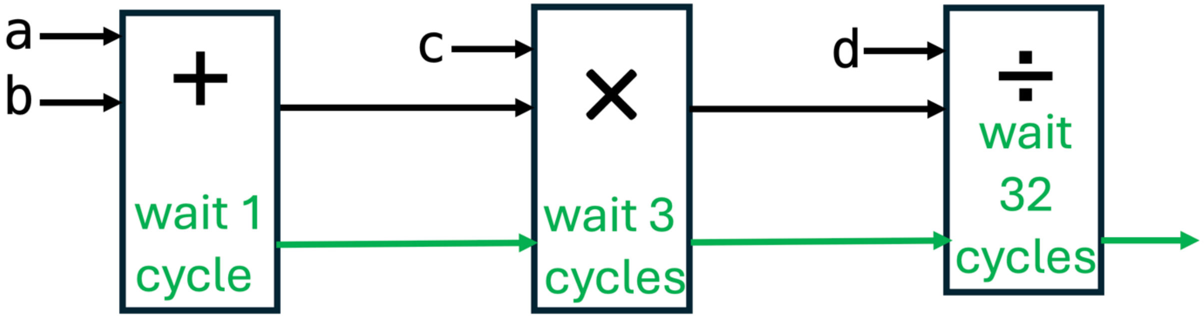
Dynamic Only

Recap: So Many Choices!

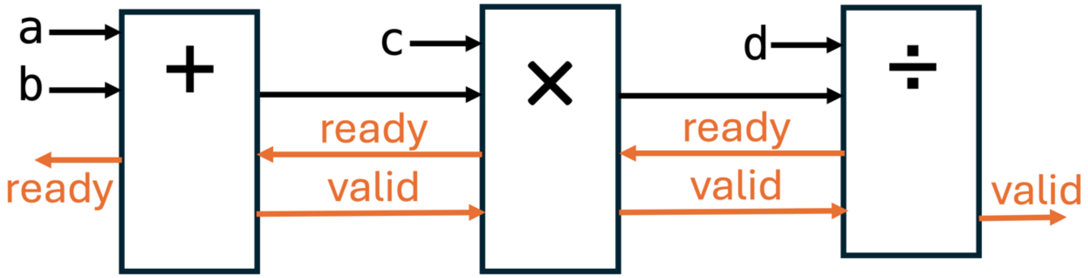


Dynamic Only

Static Only

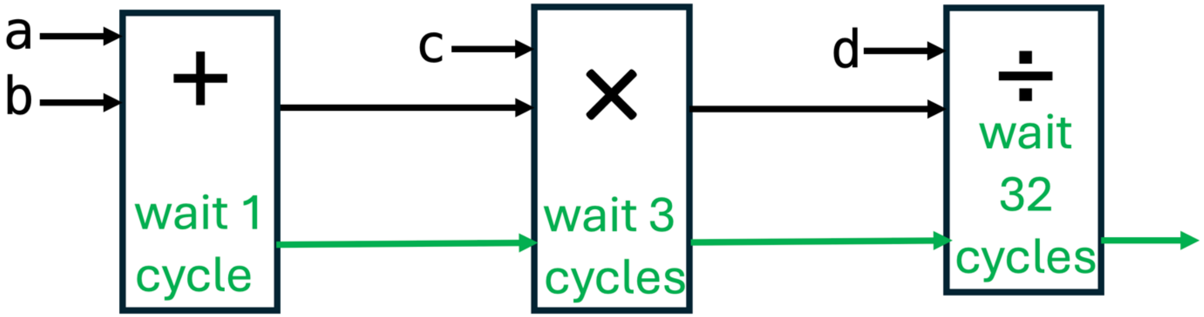


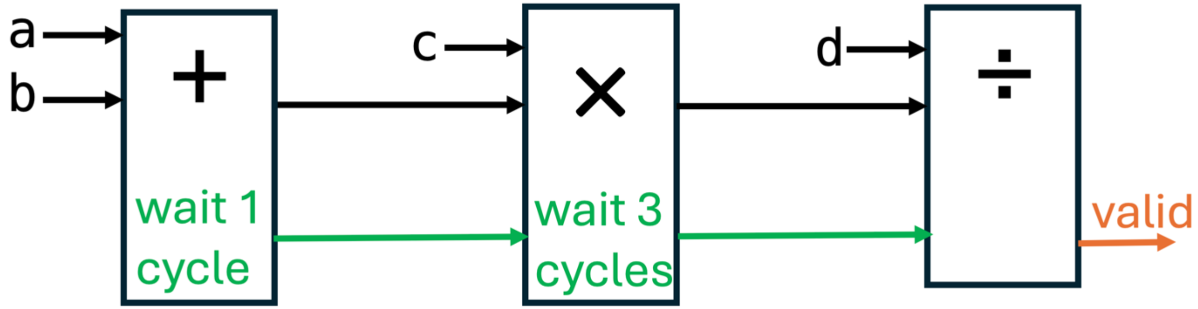
Recap: So Many Choices!



Dynamic Only

Static Only





Mixed Static Dynamic

Key Idea #1: Two paradigms (**static** and **dynamic**) for hardware compiler IRs with severe **trade-offs**

Recap: So Many Choices!

Recap: So Many Choices!

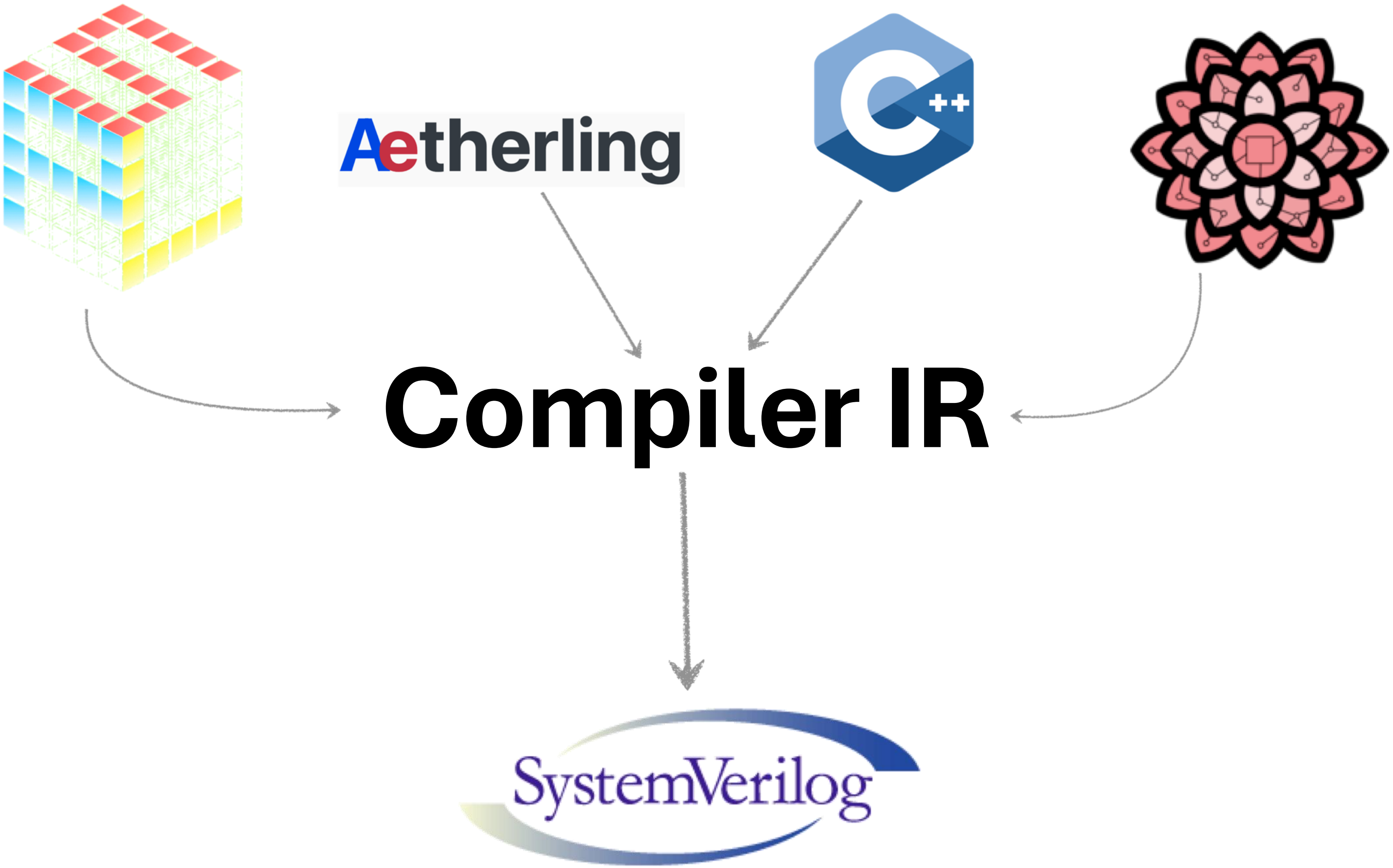
… just for $((a+b)\times c)\div d$!

What about “real” languages?

What about "real" languages?

```
let a: float[10];
let b: float[10 bank 2];
for (let i = 0 .. 10) unroll 2 {
  a[i] := b[i] * 2.0;
}
```

What about “real” languages?



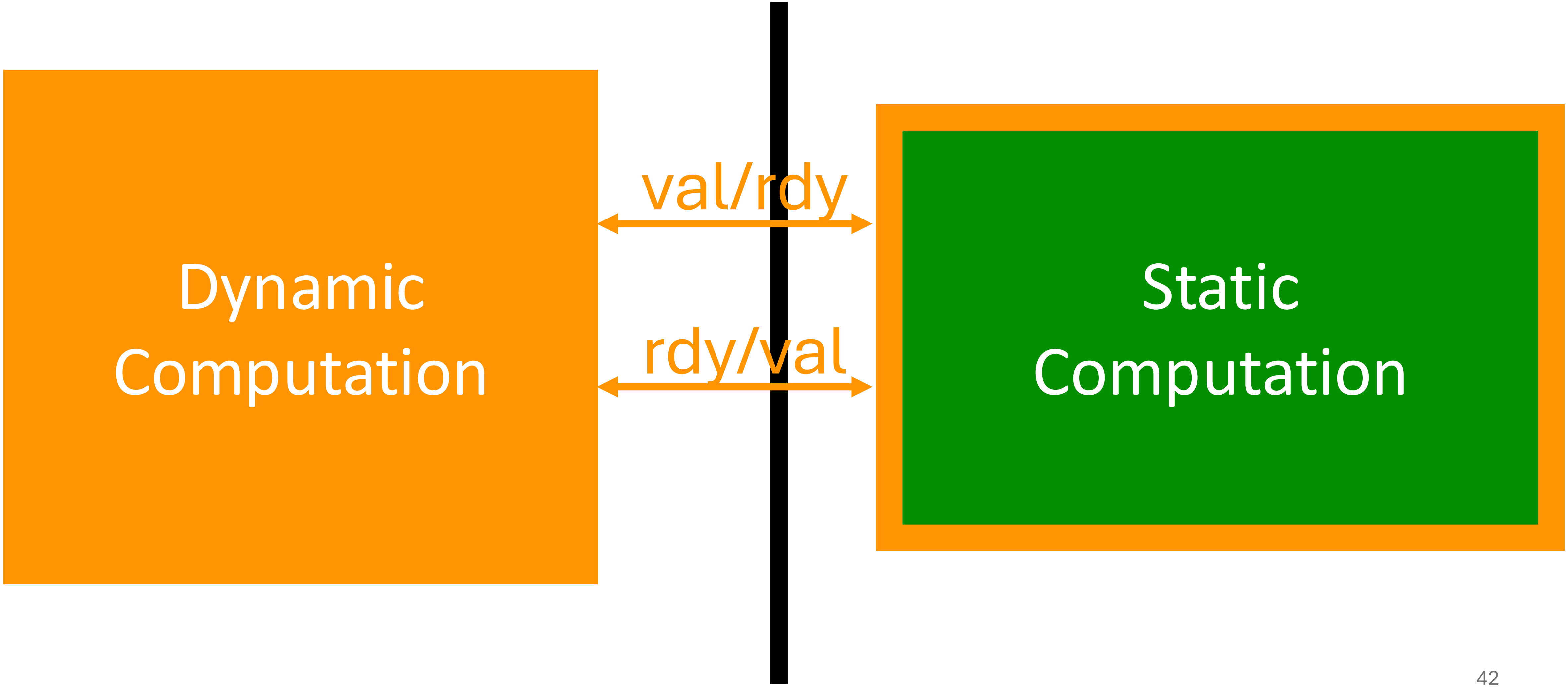
Static and dynamic compiler IRs live in *separate worlds*

Static and dynamic compiler IRs live in *separate worlds*

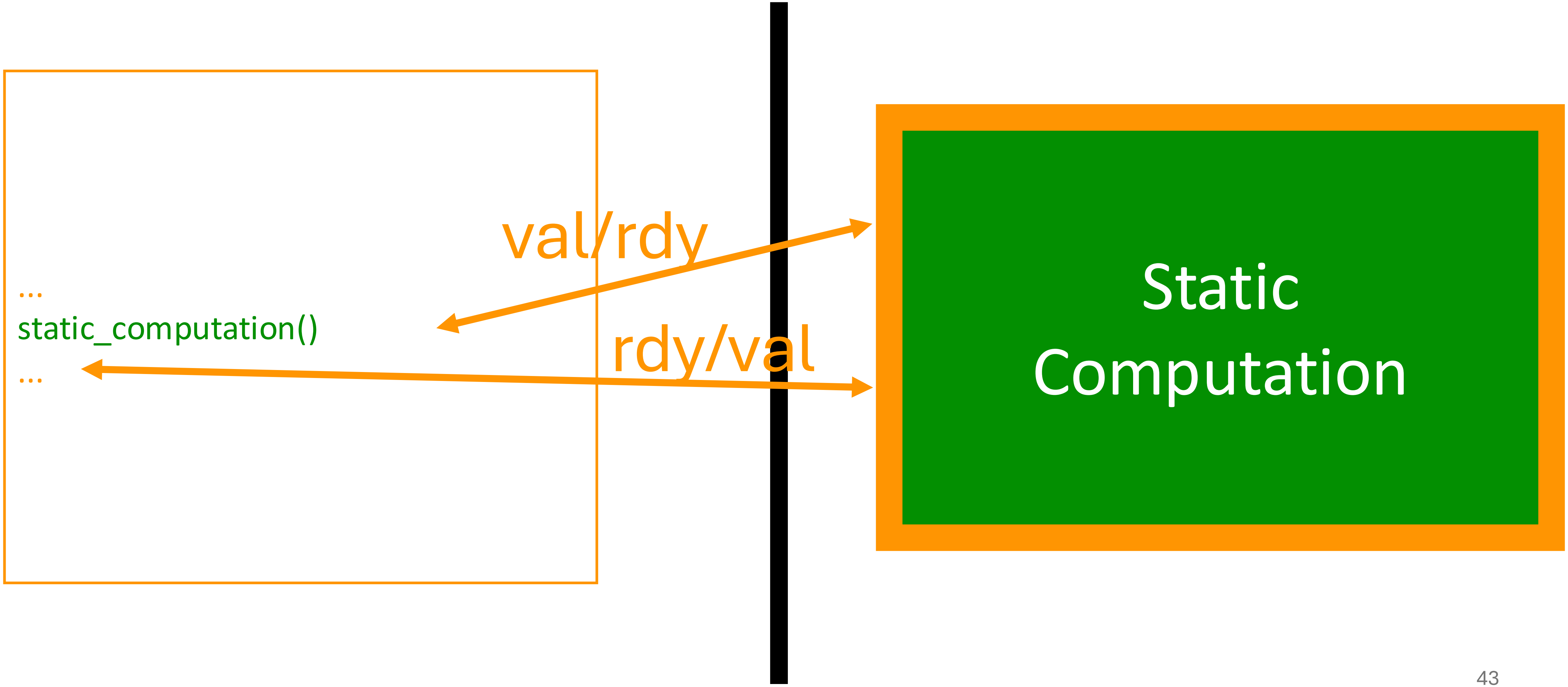
Dynamic Computation

Static Computation

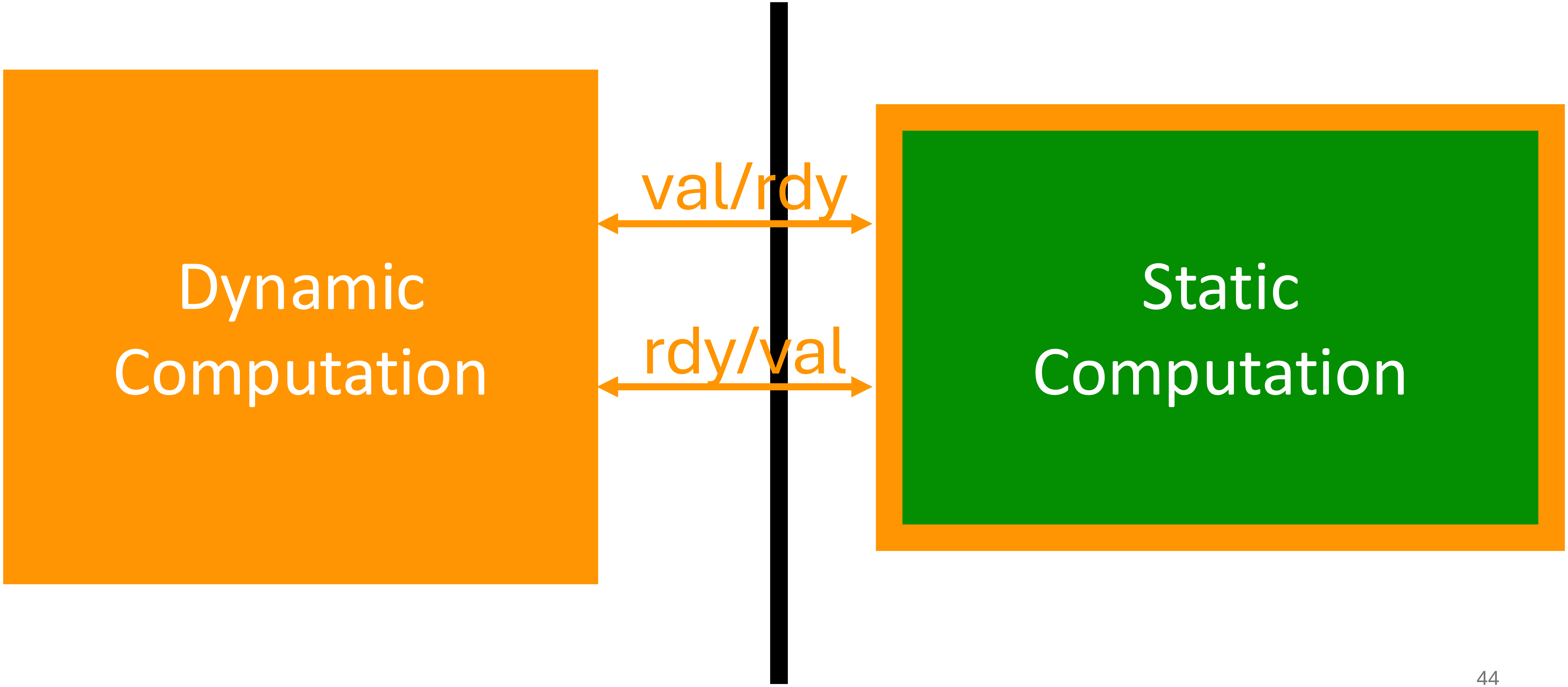
Static and dynamic compiler IRs live in *separate worlds*



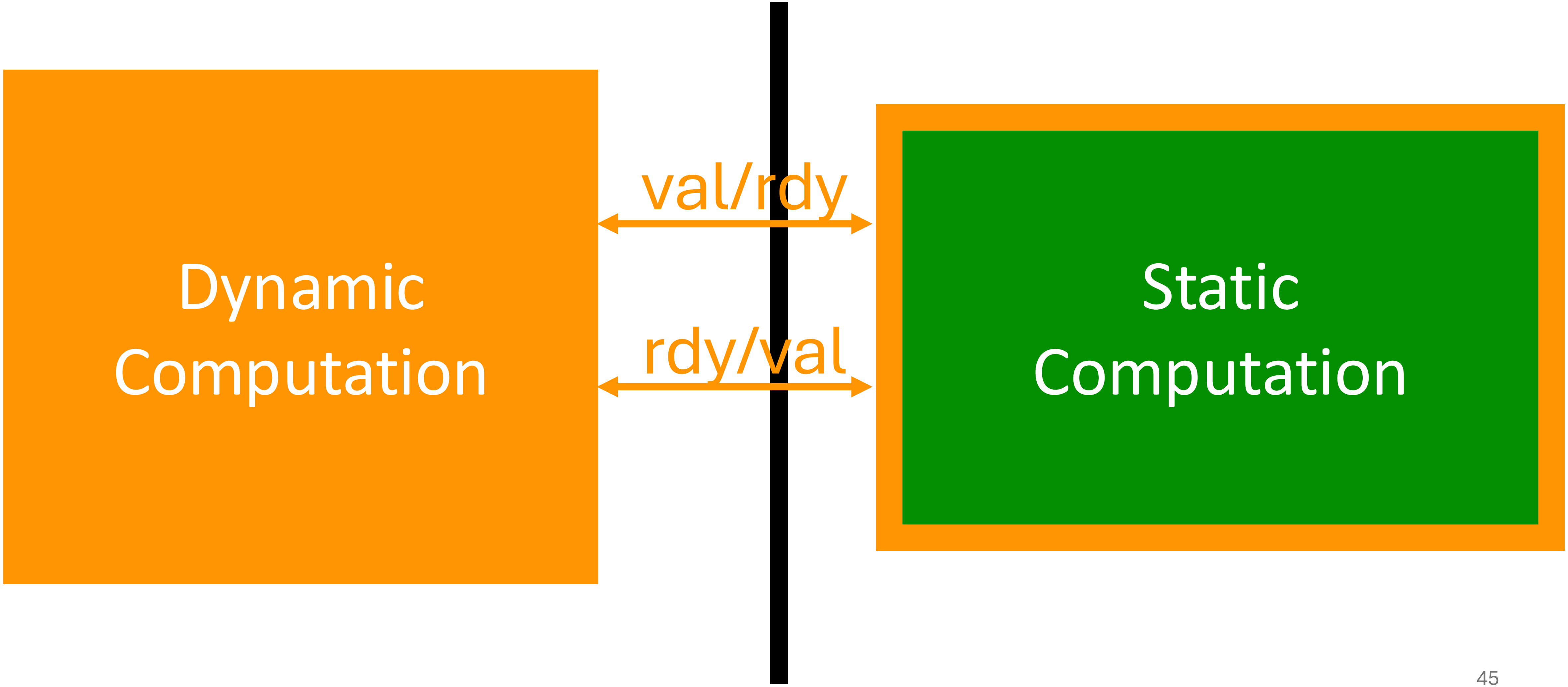
Static and dynamic compiler IRs live in *separate worlds*



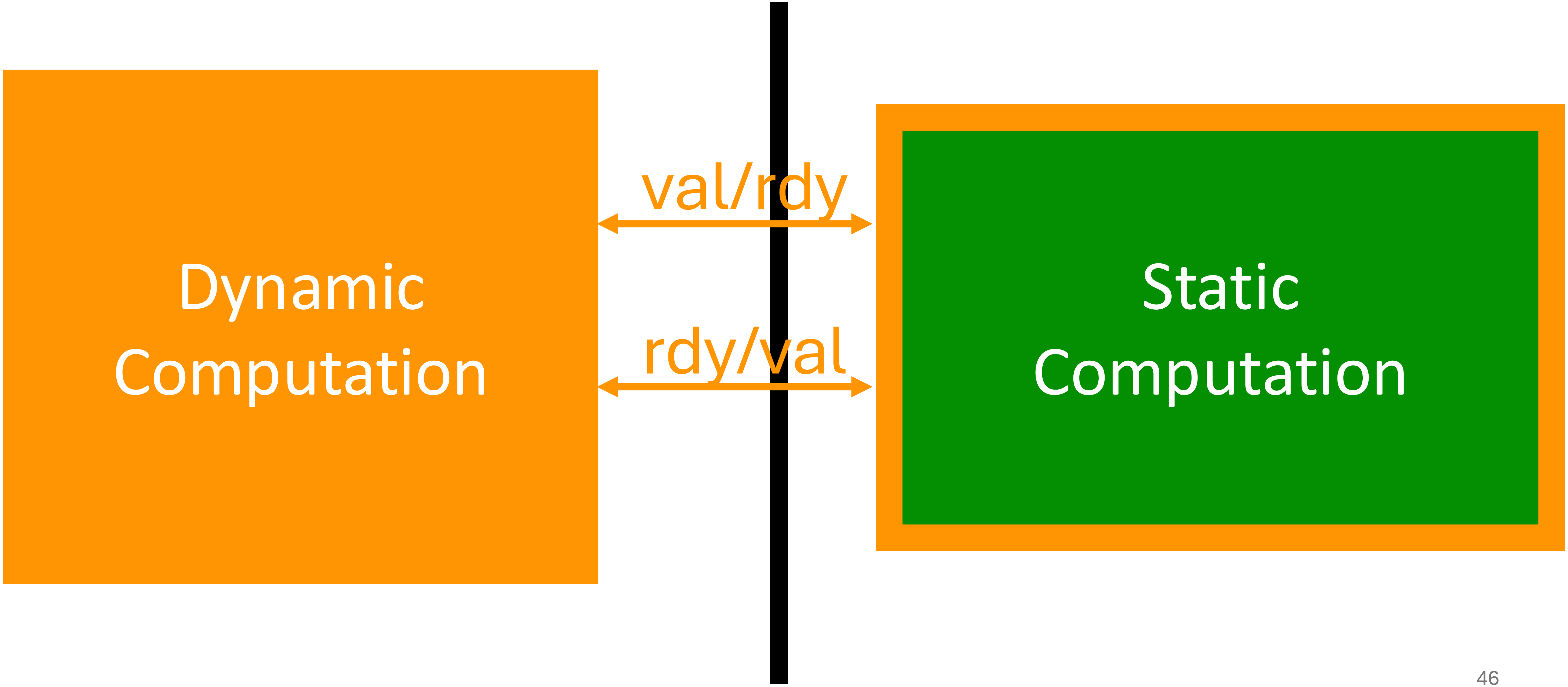
Compiler developers must maintain 2 IRs



Optimizations must analyze and transform 2 IRs



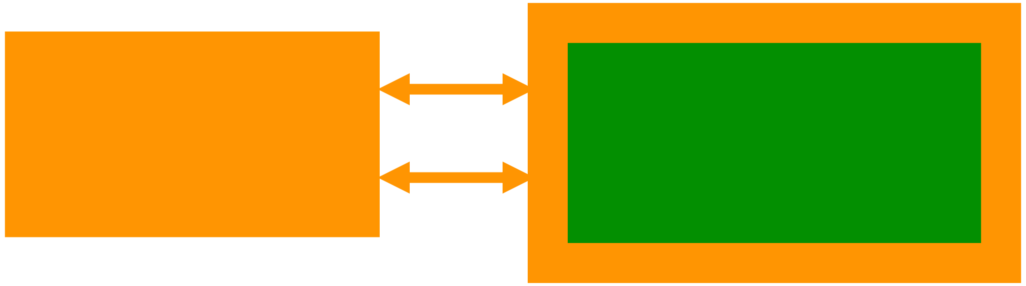
Global optimizations are limited by the static-dynamic boundary



Summary: Existing Compilers

Stratified

Static-Dynamic



Summary: Existing Compilers

Stratified

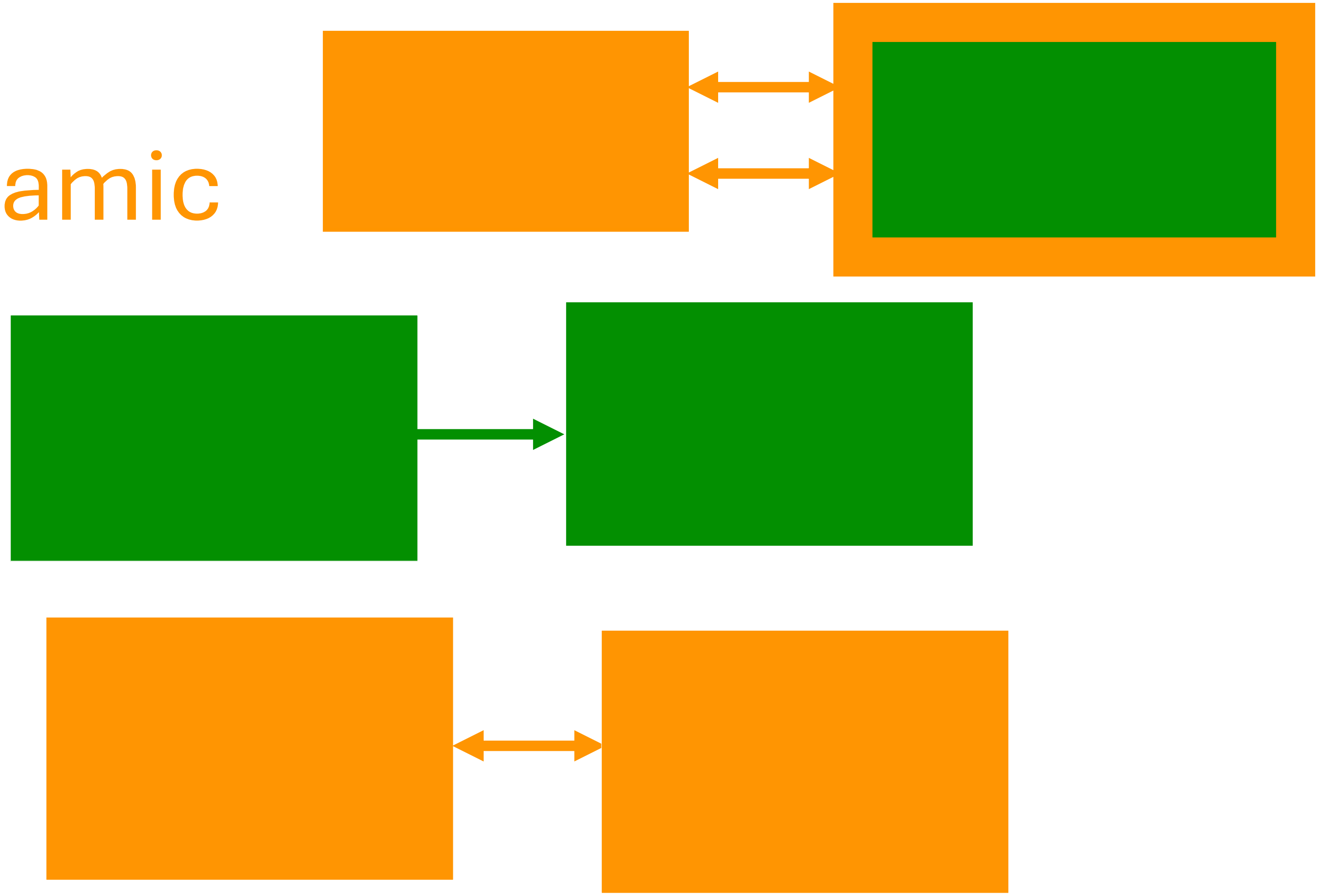
Static-Dynamic

Static

Only

Dynamic

Only



Key Idea #2: Existing compilers IRs **stratify** static and dynamic sections into **separate languages/representations**

Our work: *unifying* static and dynamic compiler IRs

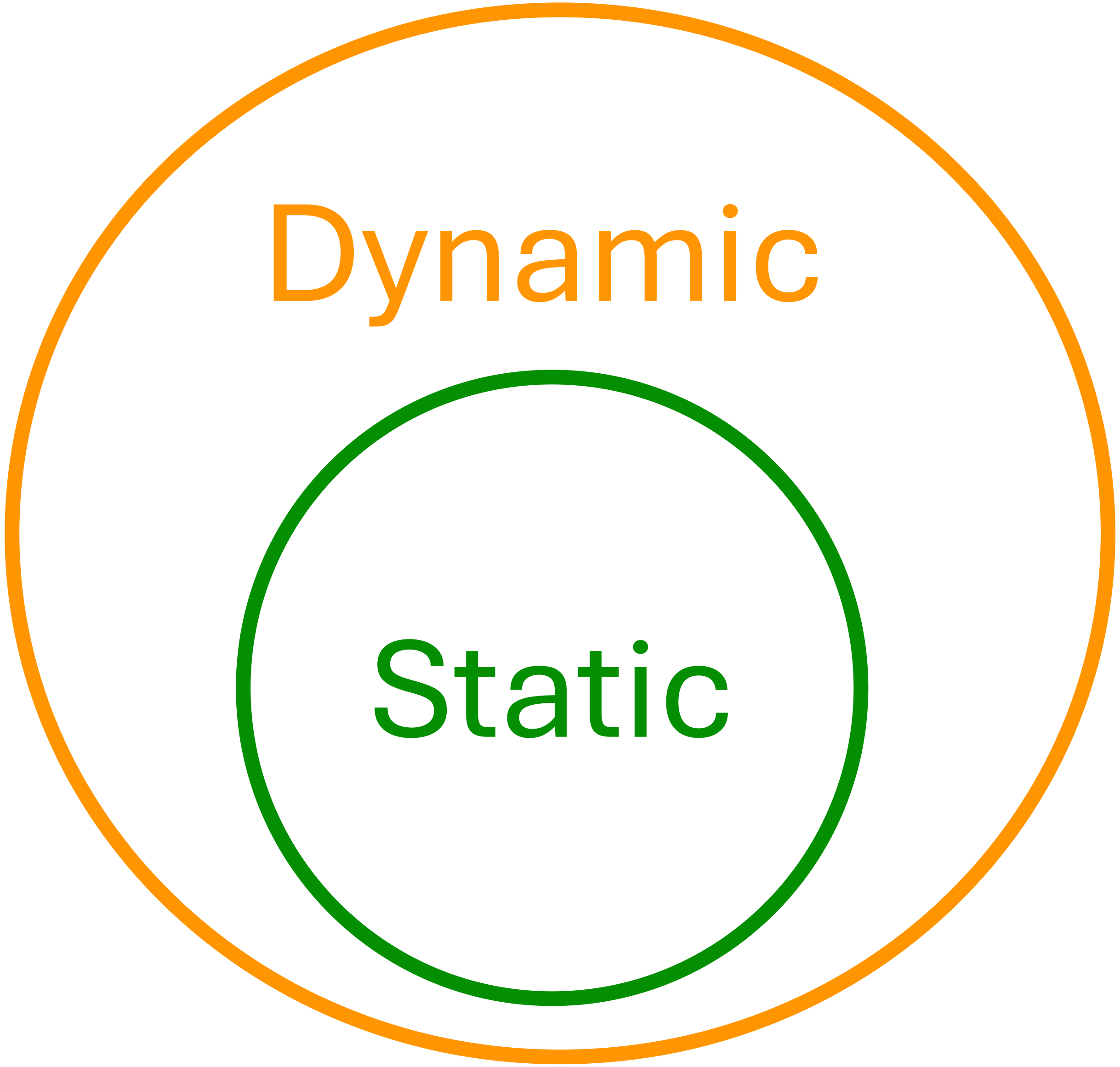
Our work: *unifying* static and dynamic compiler IRs through *semantic refinement*

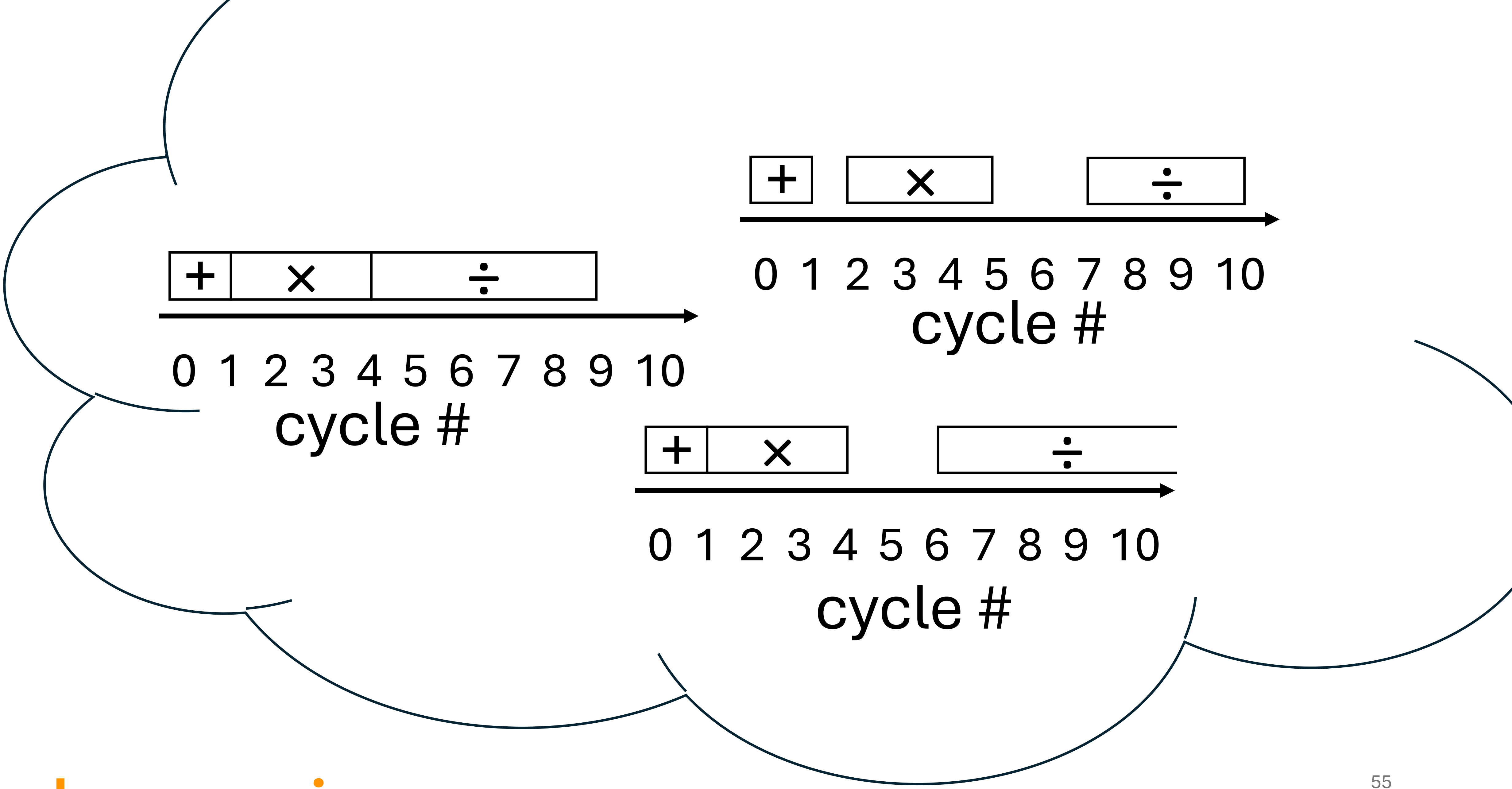
Semantic Refinement

Semantic Refinement

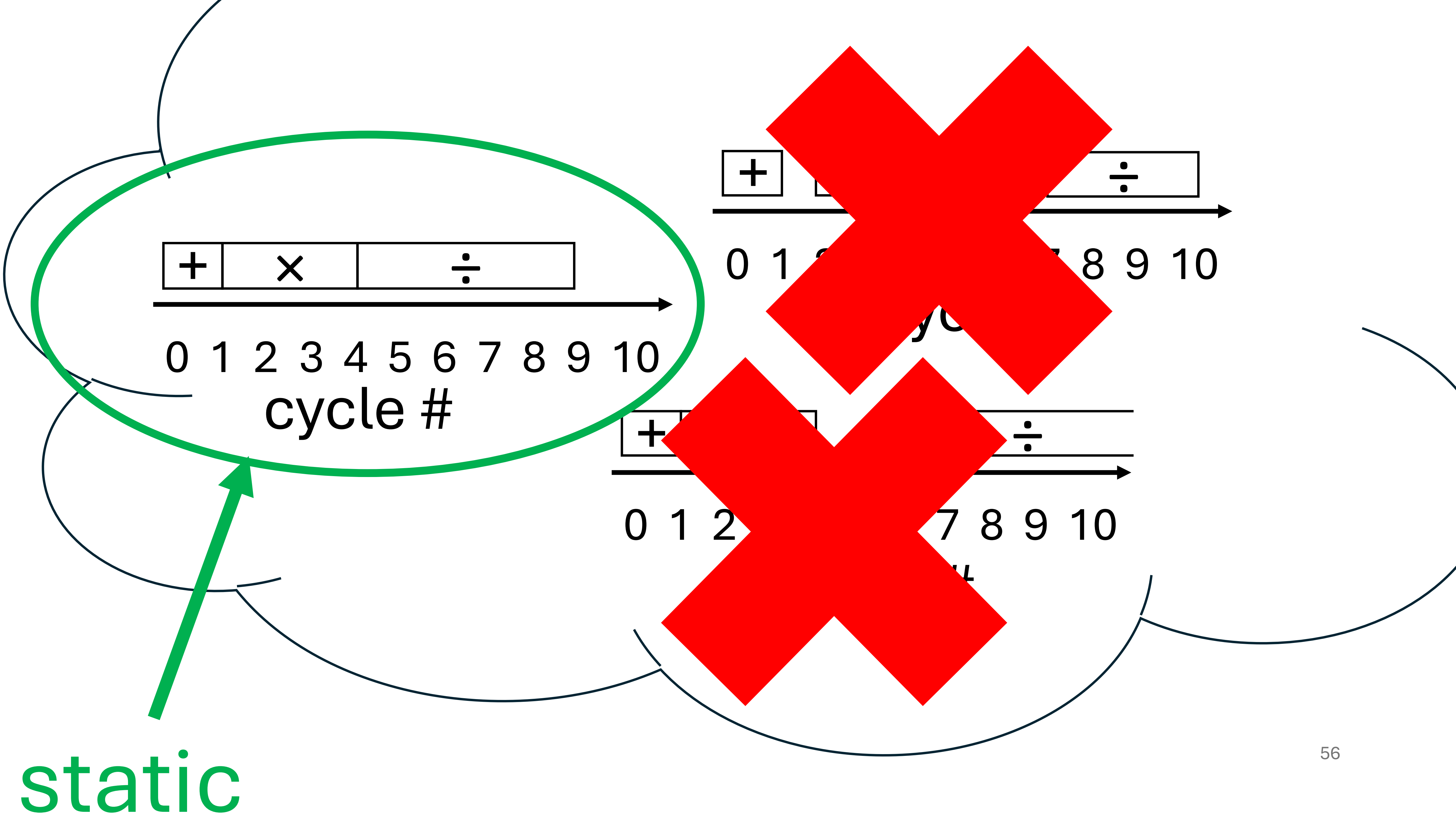
"P refines Q iff P allows a subset of the behaviors that Q allows."

static *refines* dynamic

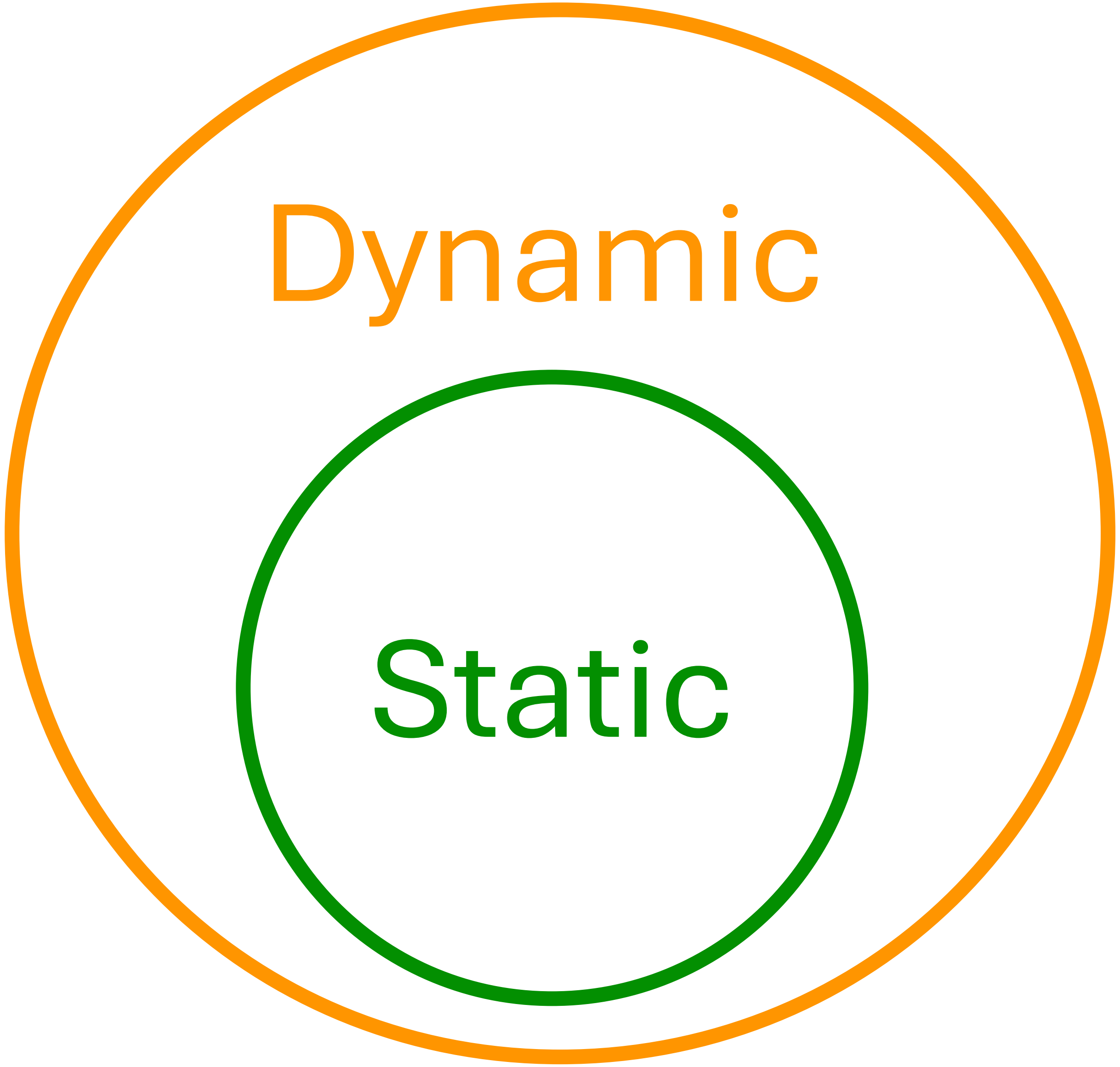



+
×
÷
0 1 2 3 4 5 6 7 8 9 10
cycle #
+
×
÷
0 1 2 3 4 5 6 7 8 9 10
cycle #
+
×
÷
0 1 2 3 4 5 6 7 8 9 10
cycle #

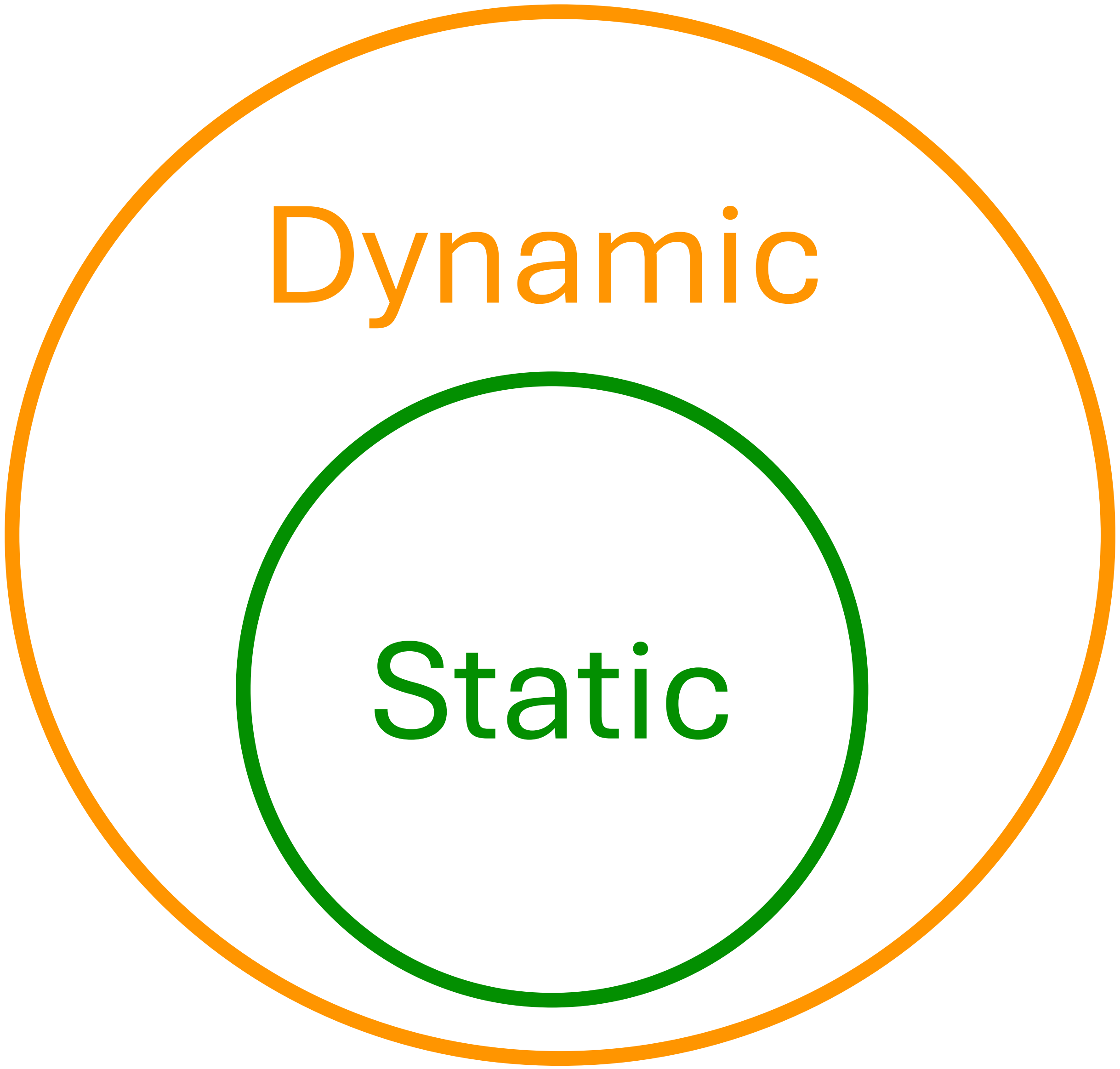
dynamic


+
×
÷
0 1 2 3 4 5 6 7 8 9 10
cycle #
static

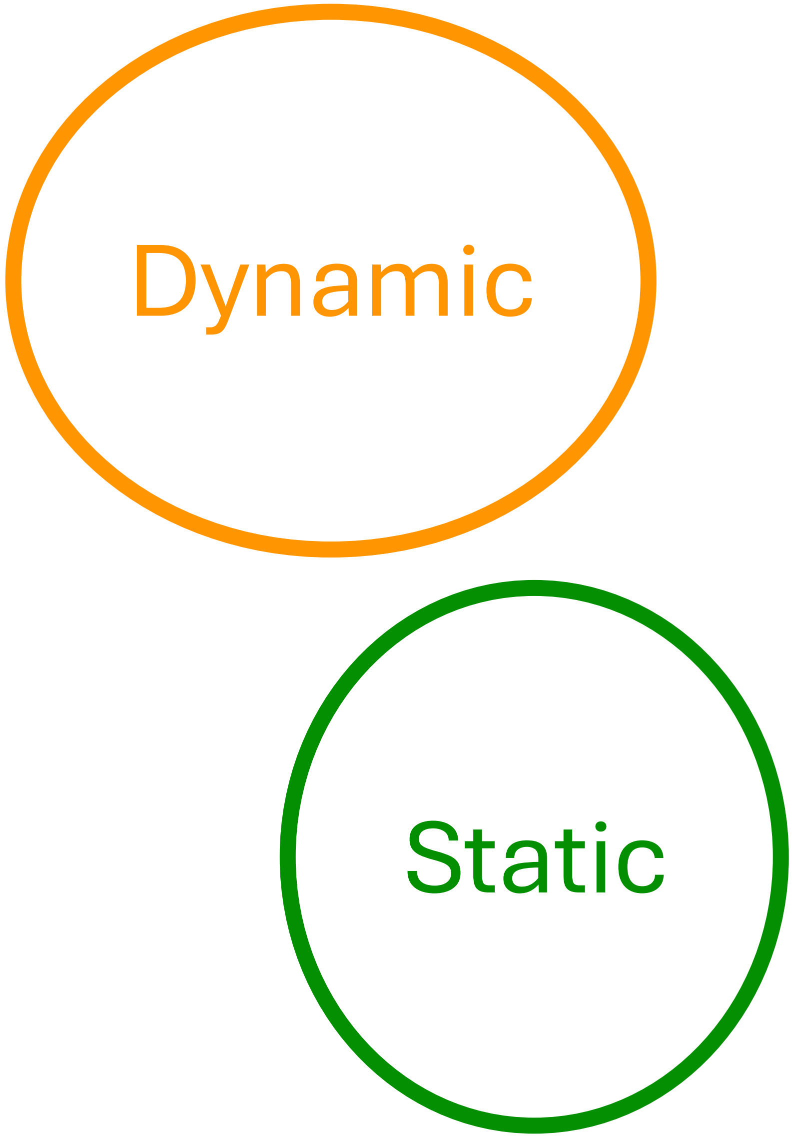
static *refines* dynamic

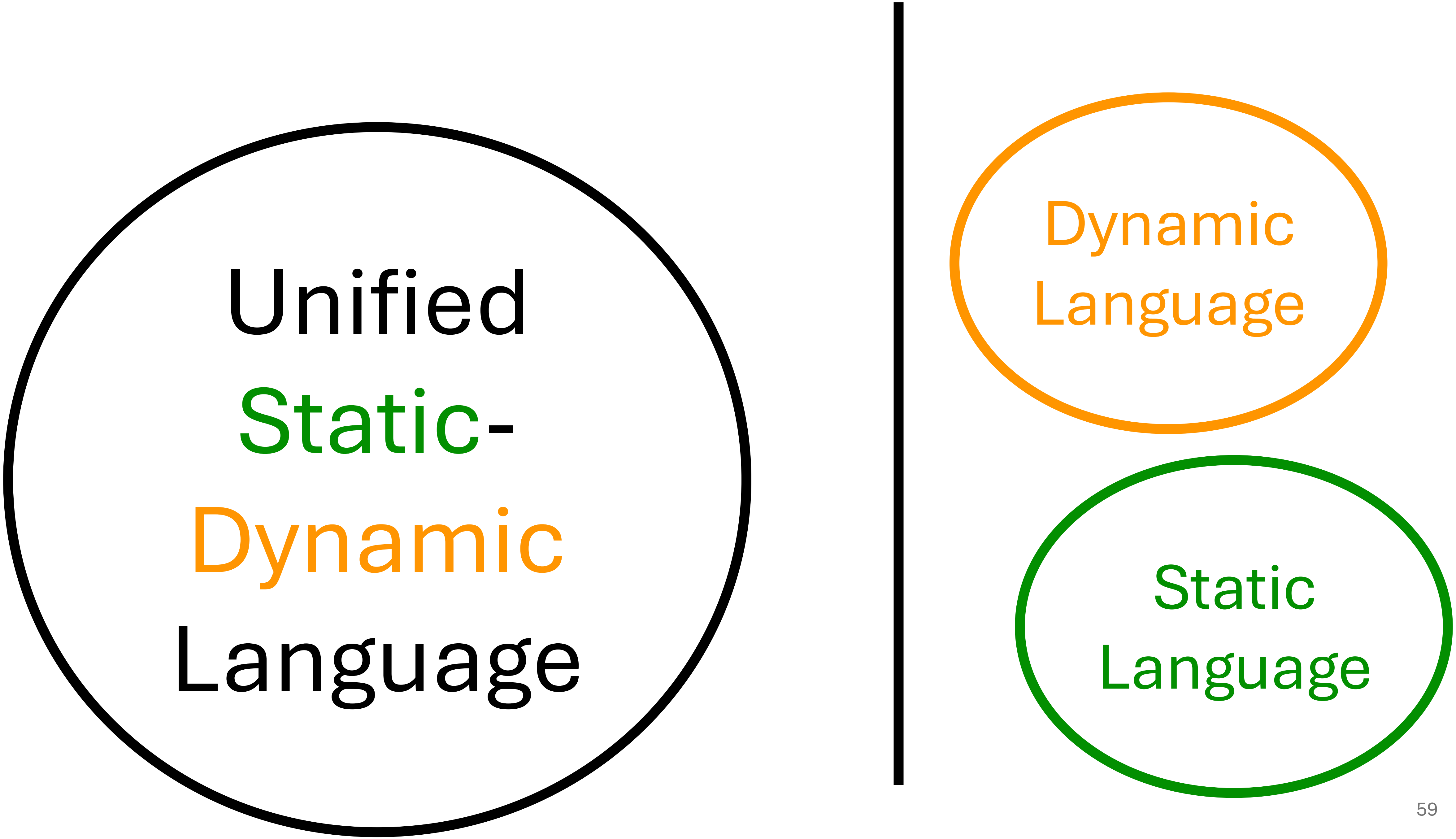


static *refines* dynamic



separate behaviors




Unified
Static-
Dynamic
Language
Dynamic
Language
Static
Language

Our work builds on a purely **dynamic** language

Dynamic

```
wires {
 group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
 }
 group do_mult {
  mult.left = add.out;
  mult.right = c;
  do_mult[done] = mult.done;
 }
 group do_div {
  div.go = 1;
  div.left = mult.out;
  div.right = d;
  do_div[done] = div.done;
 }
}
control {
  seq {
   seq { do_add; do_mult; }
   do_div;
  }
 }
}
```

Dynamic + Static

```
wires {
 group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
 }
 group do_mult {
  mult.left = add.out;
  mult.right = c;
  do_mult[done] = mult.done;
 }
 group do_div {
  div.go = 1;
  div.left = mult.out;
  div.right = d;
  do_div[done] = div.done;
 }
}
control {
  seq {
   seq { do_add; do_mult; }
   do_div;
  }
 }
}
```

```
wires {
 static<1> group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
 }
 static<3> group do_mult {
  mult.left = add.out;
  mult.right = c;
  do_mult[done] = mult.done;
 }
 group do_div {
  div.go = 1;
  div.left = mult.out;
  div.right = d;
  do_div[done] = div.done;
 }
}
control {
  seq {
   static seq { do_add; do_mult; }
   do_div;
  }
 }
}
```

Dynamic
+
Static
=
Unification

```
wires {
  group do_add {
    add.left = a;
    add.right = b;
    do_add[done] = add.done;
  }
  group do_mult {
    mult.left = add.out;
    mult.right = c;
    do_mult[done] = mult.done;
  }
  group do_div {
    div.go = 1;
    div.left = mult.out;
    div.right = d;
    do_div[done] = div.done;
  }
}
control {
  seq {
    seq { do_add; do_mult; }
    do_div;
  }
}
```

```
wires {
  static<1> group do_add {
    add.left = a;
    add.right = b;
    do_add[done] = add.done;
  }
  static<3> group do_mult {
    mult.left = add.out;
    mult.right = c;
    do_mult[done] = mult.done;
  }
  group do_div {
    div.go = 1;
    div.left = mult.out;
    div.right = d;
    do_div[done] = div.done;
  }
}
control {
  seq {
    static seq { do_add; do_mult; }
    do_div;
  }
}
```

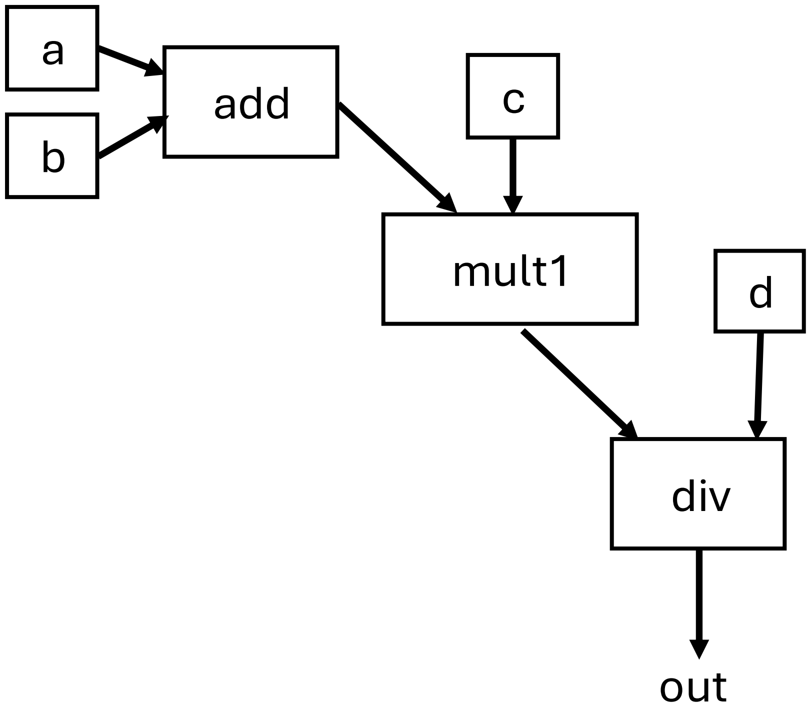
Key Idea #3: Our work uses the idea of **semantic refinement** to **unify** static and dynamic sub-languages

Concrete benefits of unification?

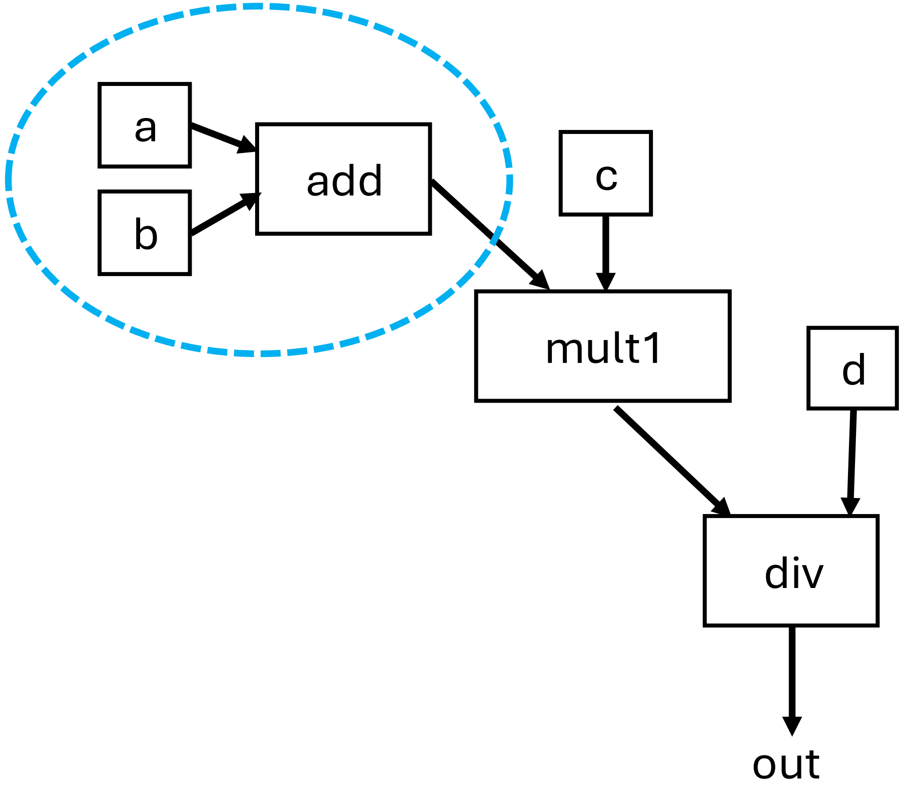
Background: Calyx

$((a+b)\times c)/d$

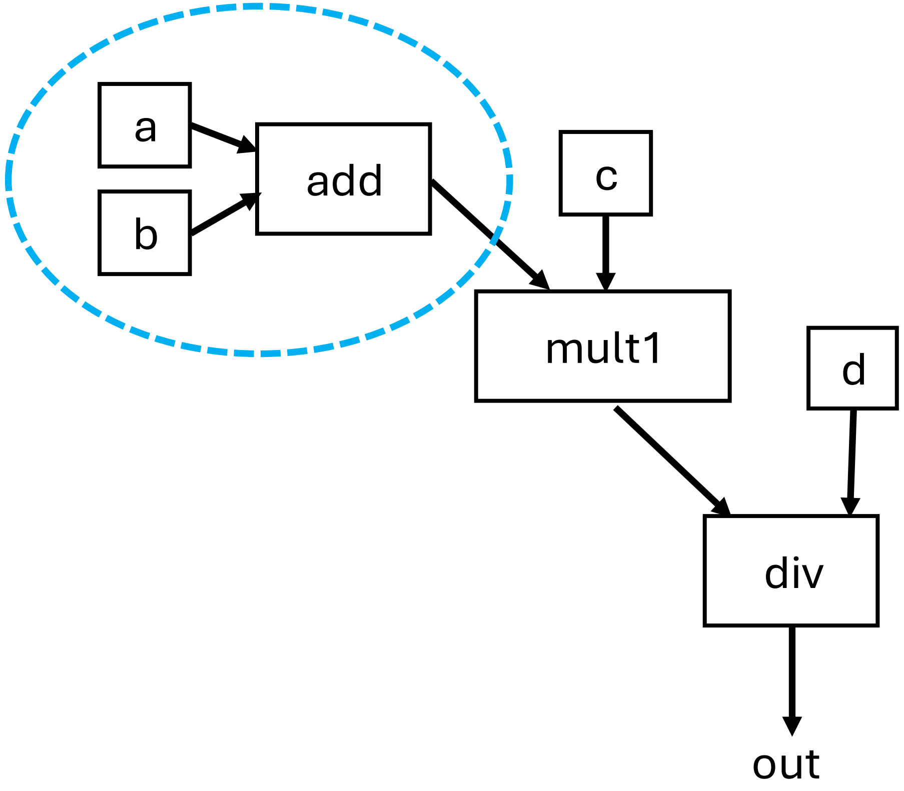
$$((a+b)\times c)/d$$



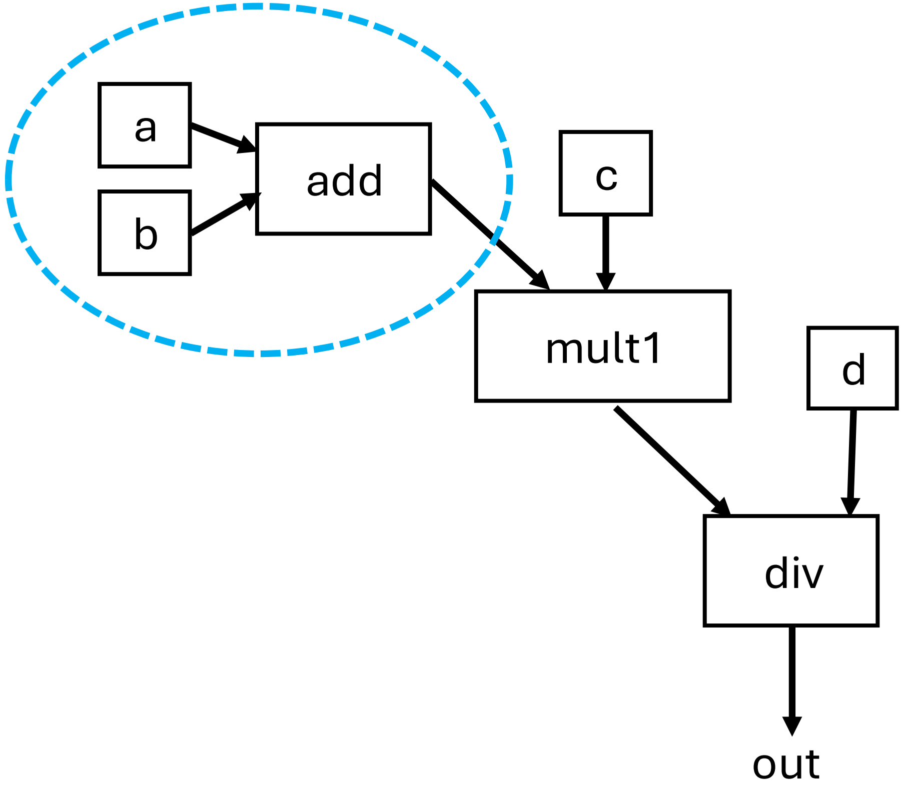
$$((a+b)\times c)/d$$



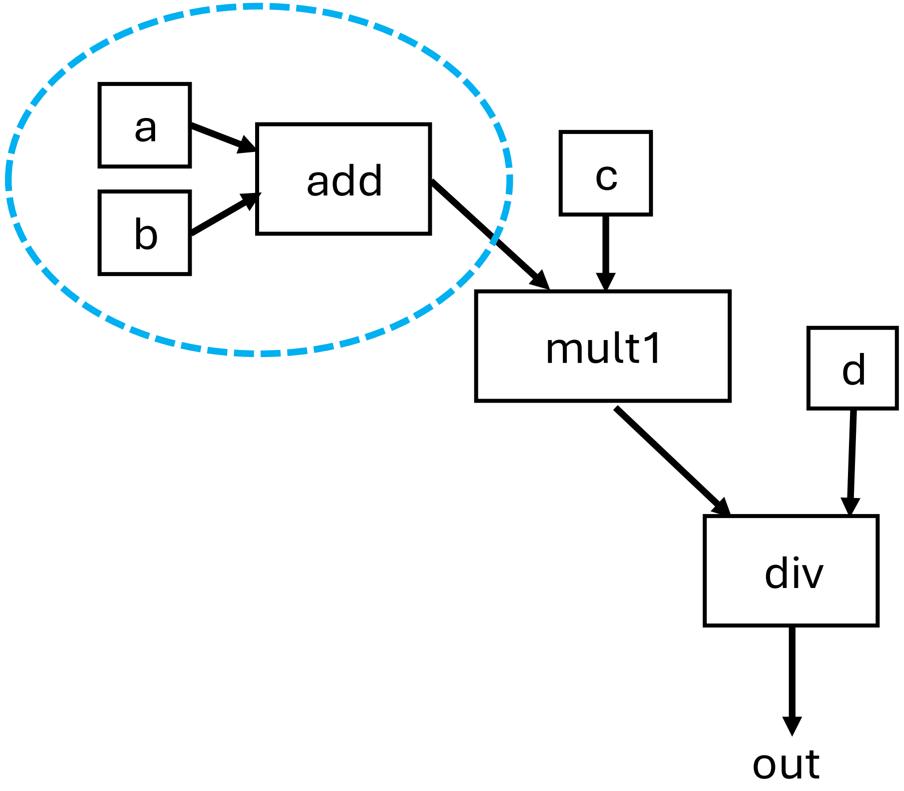
```
group do_add {


}
```



```
group do_add {
  add.left = a;
  add.right = b;

}
```

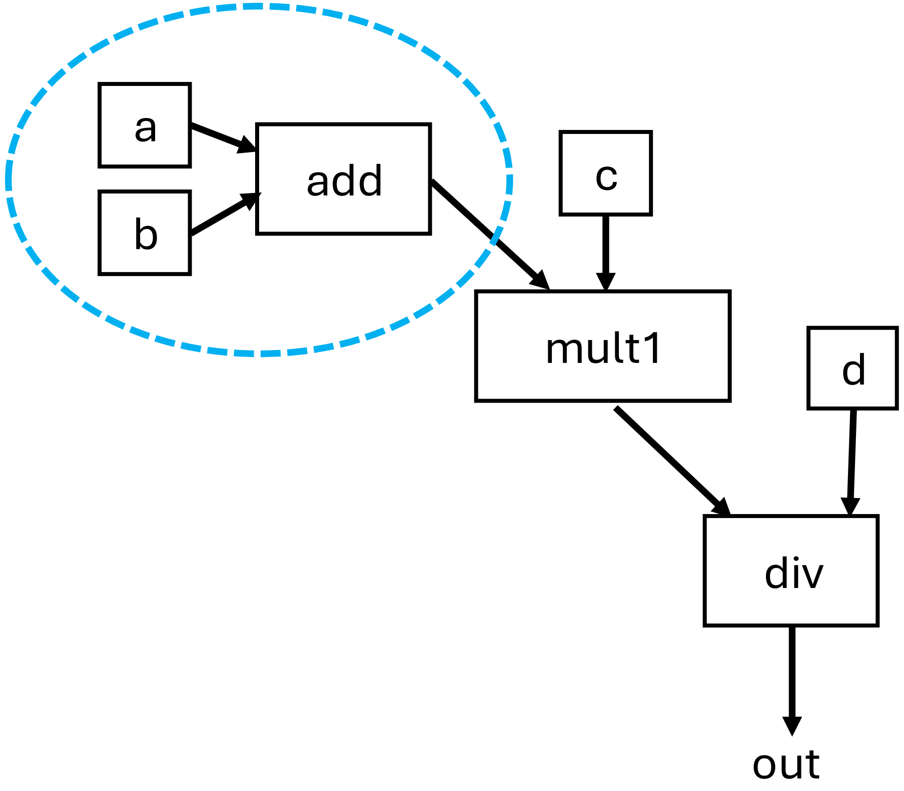


```
group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
}
```

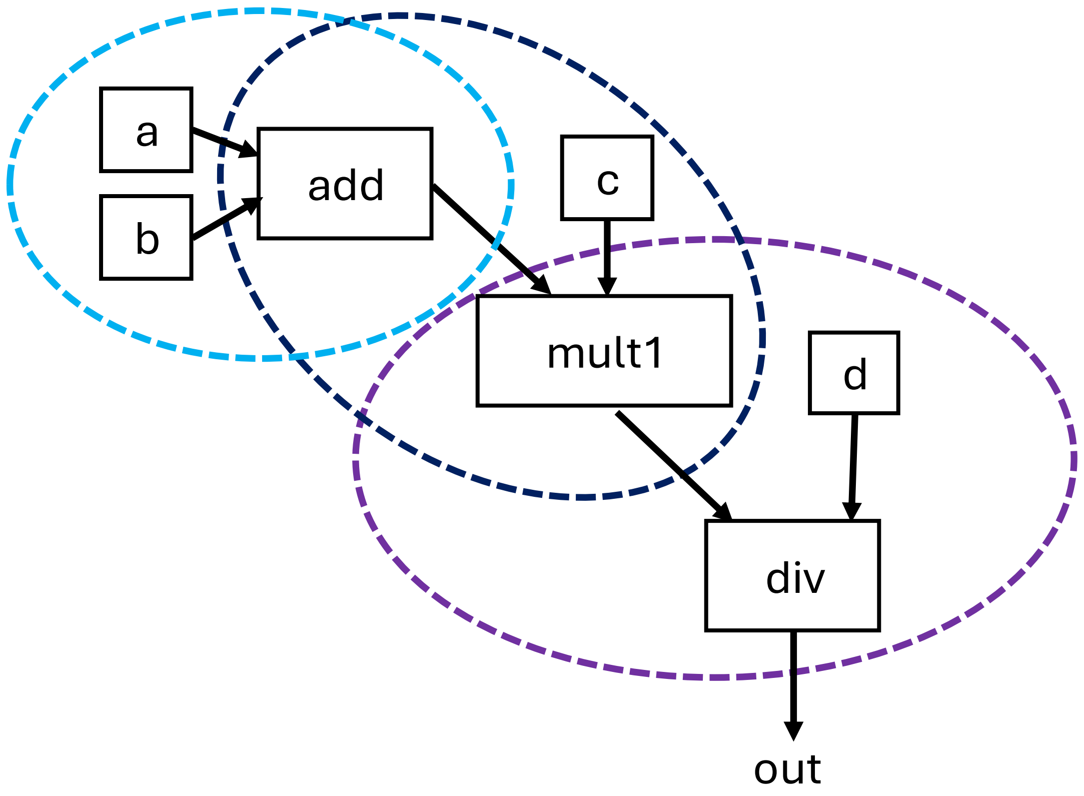


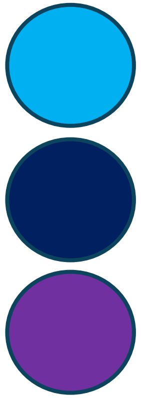
```
group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
}
```

Dynamic Interface

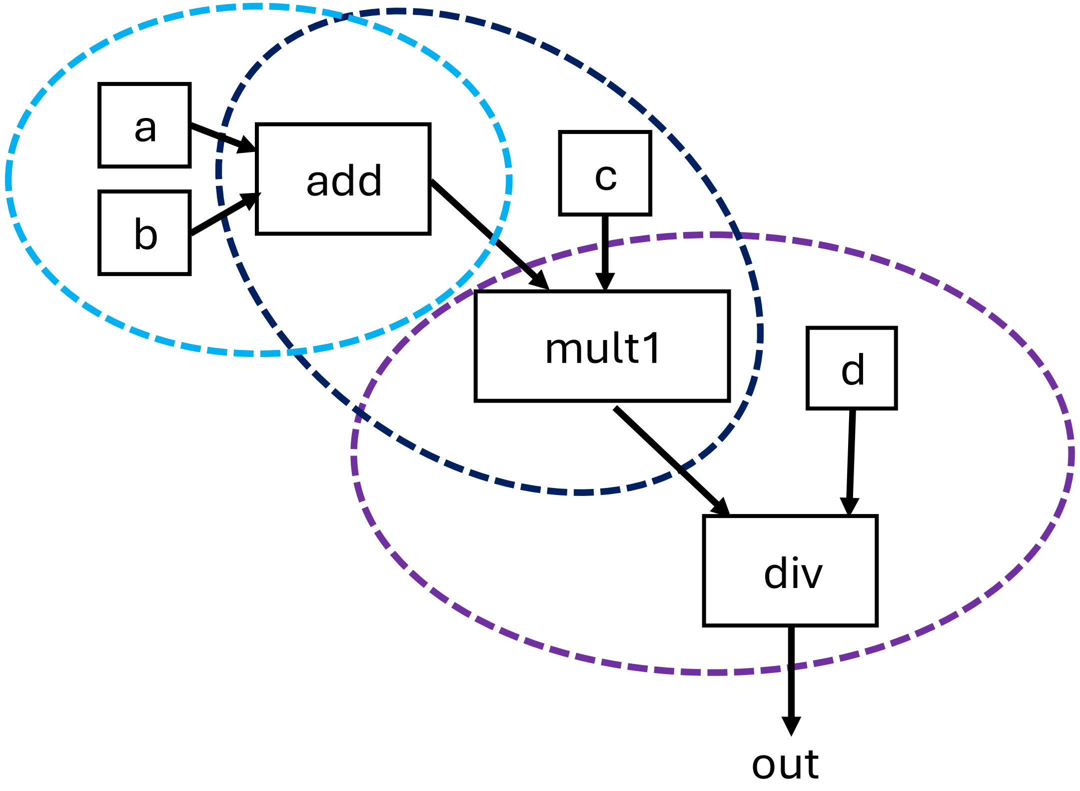


```
group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
}

group do_mult {…}

group do_div {…}
```

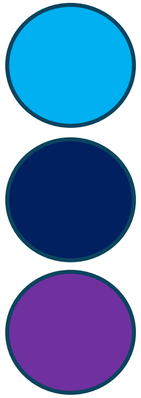


```
control {
  seq {
```


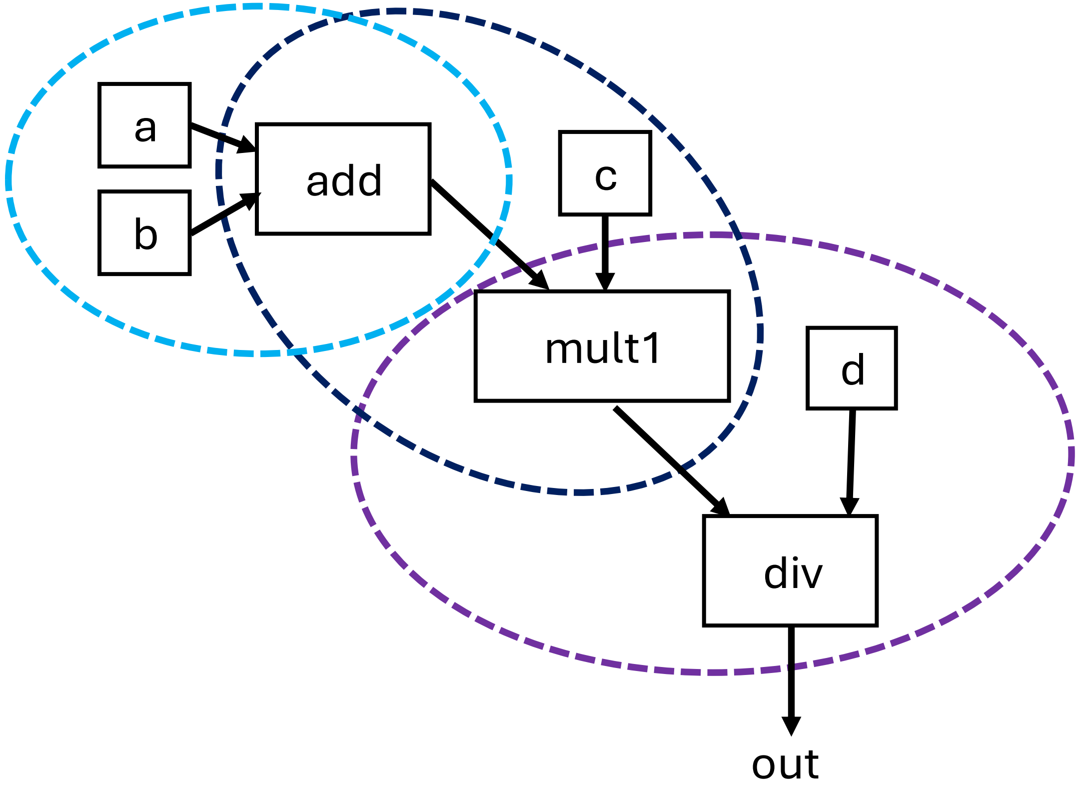
```
  }
}
```



control {
  seq {



  }
}



Also par, while, if

Piezo is a set of lightweight extensions to **Calyx** which introduce static interfaces

```
group do_add {
   add.left = a;
   add.right = b;
   do_add[done] = add.done;
}
```

```
group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
}
```

```
static<1> group do_add {
  add.left = a;
  add.right = b;
}
```

```
group do_add {
  add.left = a;
  add.right = b;
  do_add[done] = add.done;
}
```

```
static<1> group do_add {
  add.left = a;
  add.right = b;
}
```

do_add takes 1 cycle

par {…}

seq {…}

while cond {…}

if {…}

par {…} → static par {…}

seq {…} → static seq {…}

while cond {…} → static repeat n {…}

if {…} → static if {…}

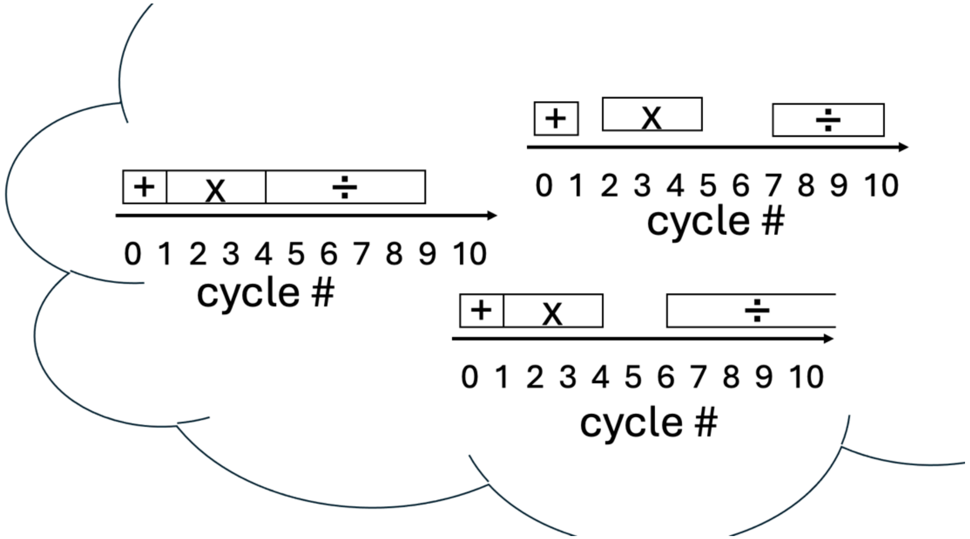
adding static keyword ***refines*** dynamic control

par {…} → static par {…}

seq {…} → static seq {…}

while cond {…} → static repeat n {…}

if {…} → static if {…}

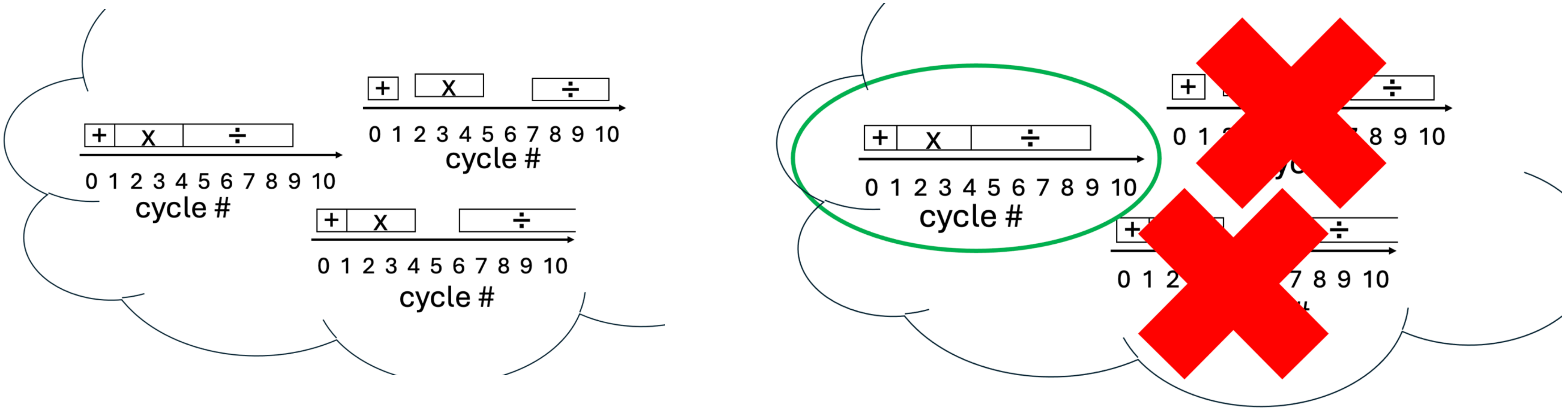


par {...} → static par {...}

seq {...} → static seq {...}

while cond {...} → static repeat n {...}

if {...} → static if {...}



par {...} → static par {...}

seq {...} → static seq {...}

while cond {...} → static repeat n {...}

if {...} → static if {...}

no <n> annotation

par {…} → static par {…}

seq {…} → static seq {…}

while cond {…} → static repeat n {…}

if {…} → static if {…}

no <n> annotation

→ latency inferred by compiler

par {...} → static par {...}

seq {...} → static seq {...}

while cond {...} → static repeat n {...}

if {...} → static if {...}

no <n> annotation

→ latency inferred by compiler

→ all “child” control is static

par {…} → static par {…}

seq {…} → static seq {…}

while cond {…} → static repeat n {…}

if {…} → static if {…}

Dynamic “ocean” with static “islands”

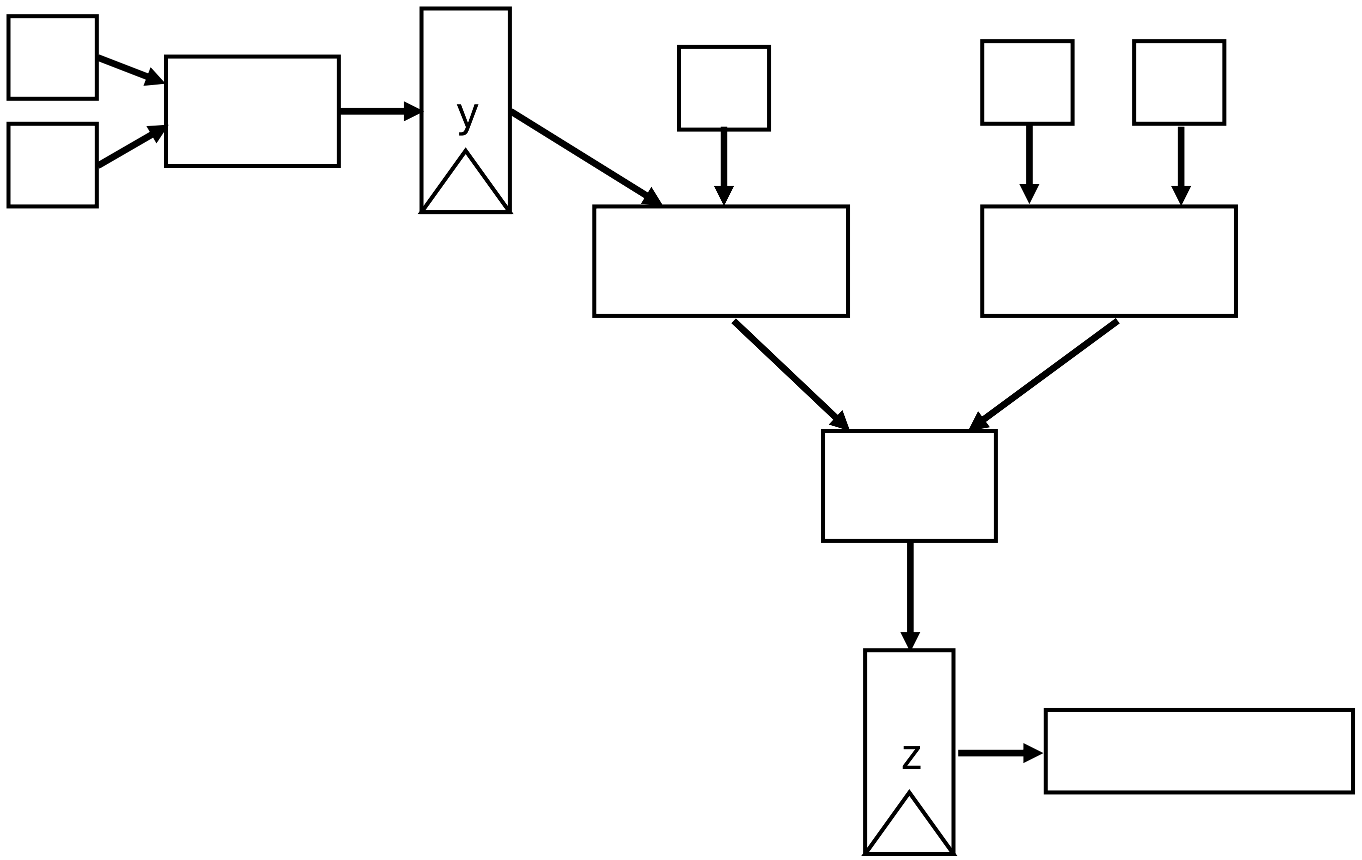
Dynamic “ocean” with static “islands”

```
control {
  seq {
      A;





    D;
  }
}
```

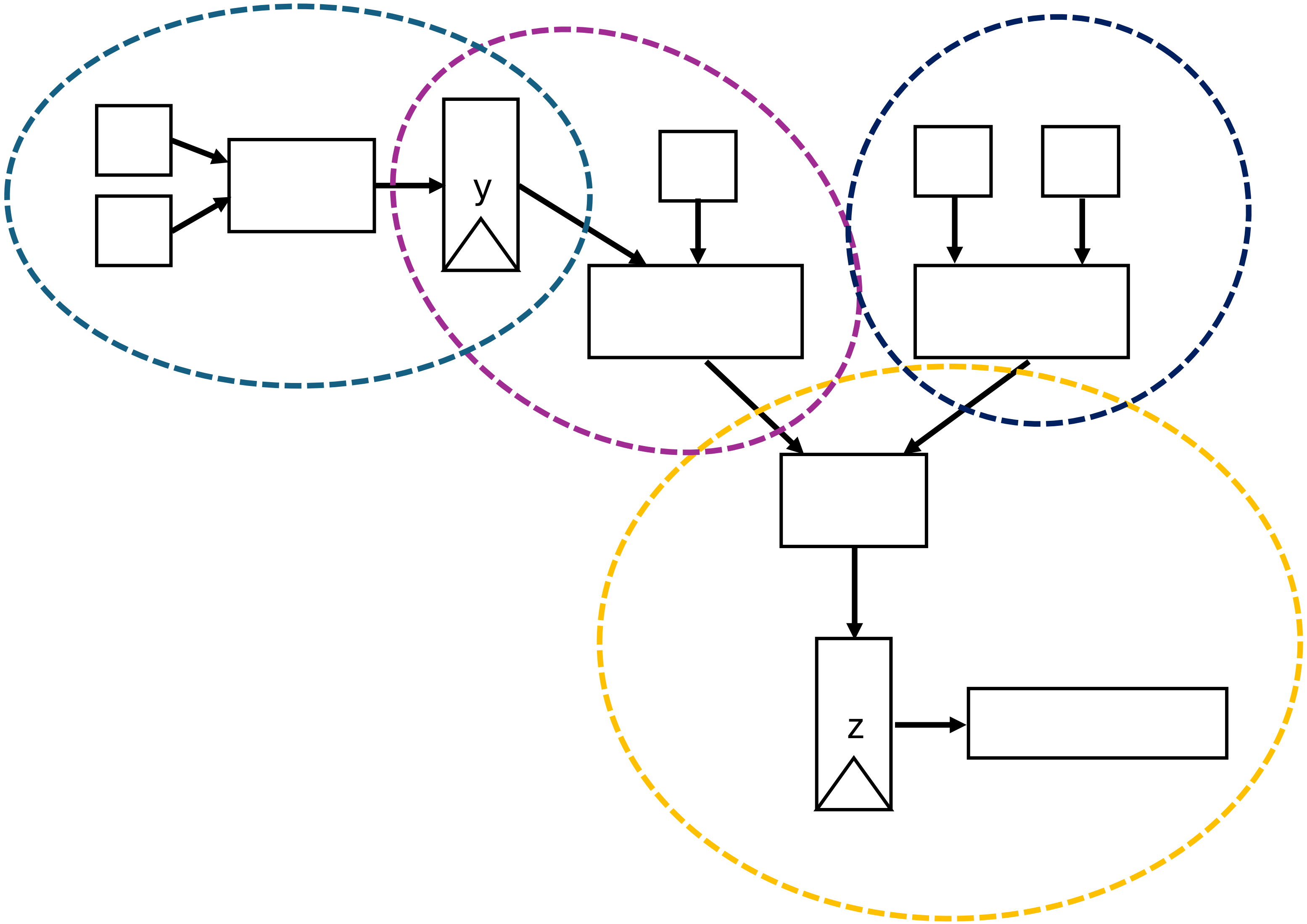
Dynamic “ocean” with static “islands”

```
control {
  seq {
      A;
    static par {
      B;
      C;
    }
    D;
  }
}
```

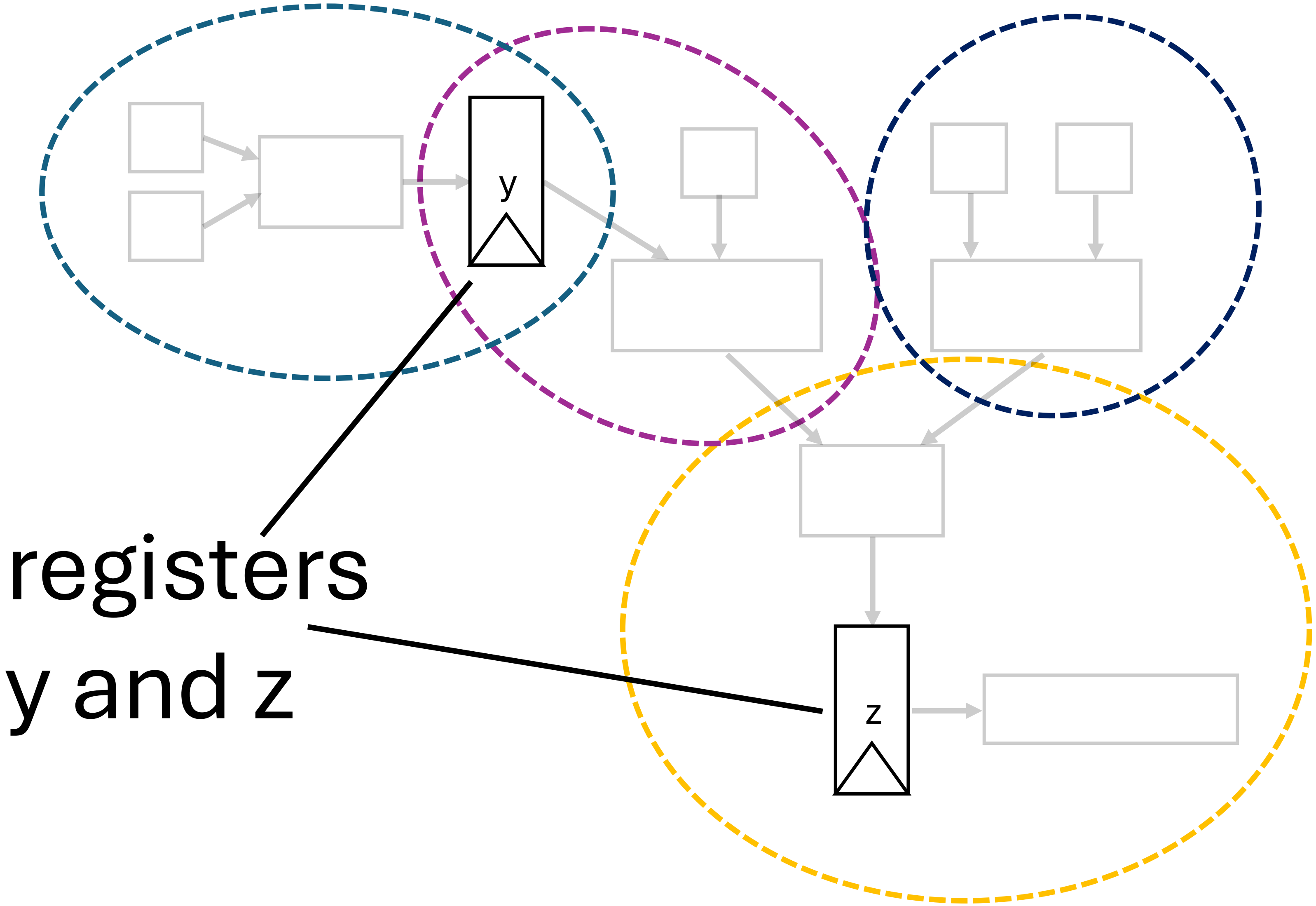
```
control {
  seq {
      A;
    static par {
      B;
      C;
    }
    D;
  }
}
```

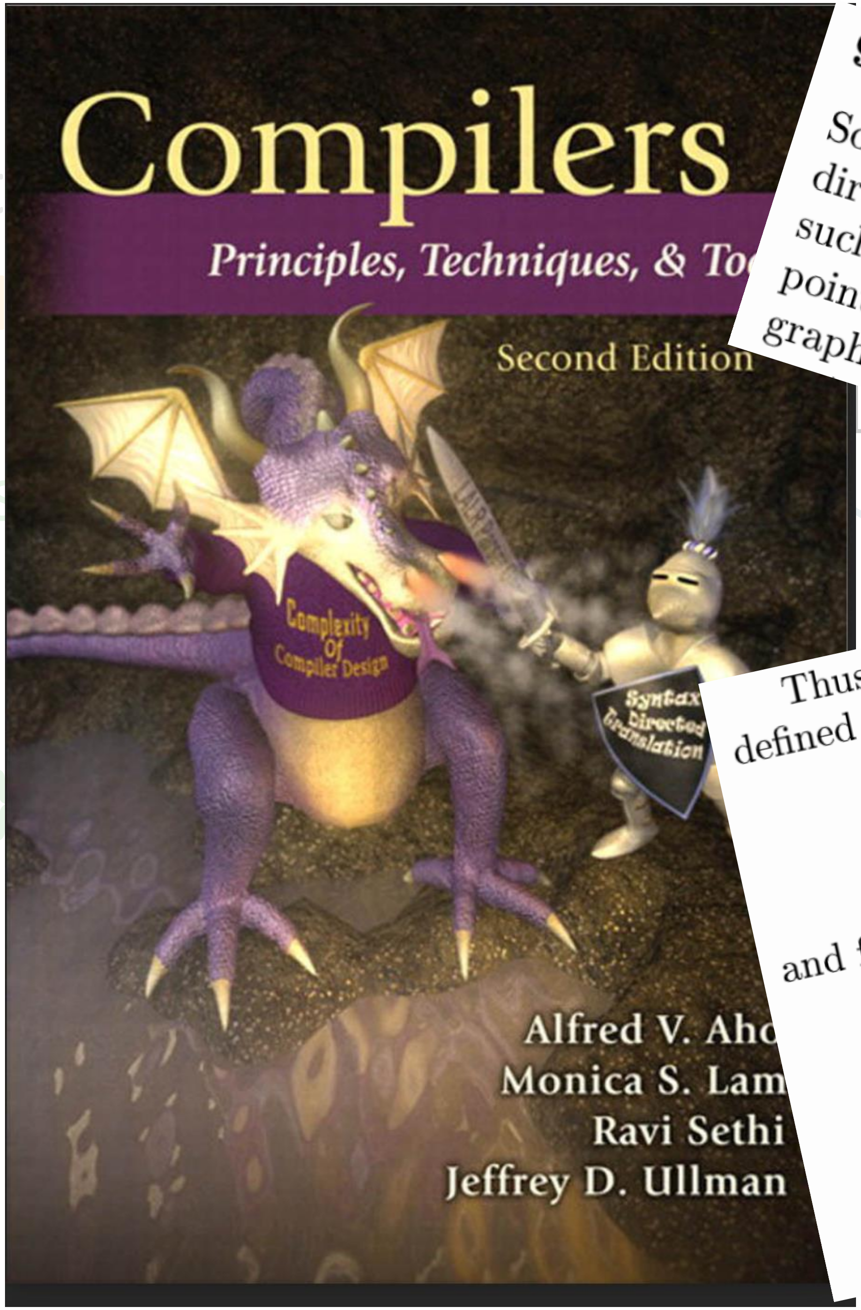


```
control {
  seq {

    static par {

    }

  }
}
```



```
control {
  seq {

      static par {


      }

  }
}
```





9.2.5 Live-Variable Analysis

Some code-improving transformations depend on information computed in the direction opposite to the flow of control in a program; we shall examine one such example now. In *live-variable analysis* we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p. If so, we say x is *live* at p; otherwise, x is *dead* at p.

Thus, the equations relating *def* and ... defined as follows:

$$\text{IN}[\text{EXIT}] = \emptyset$$

and for all basic blocks B other than EXIT,

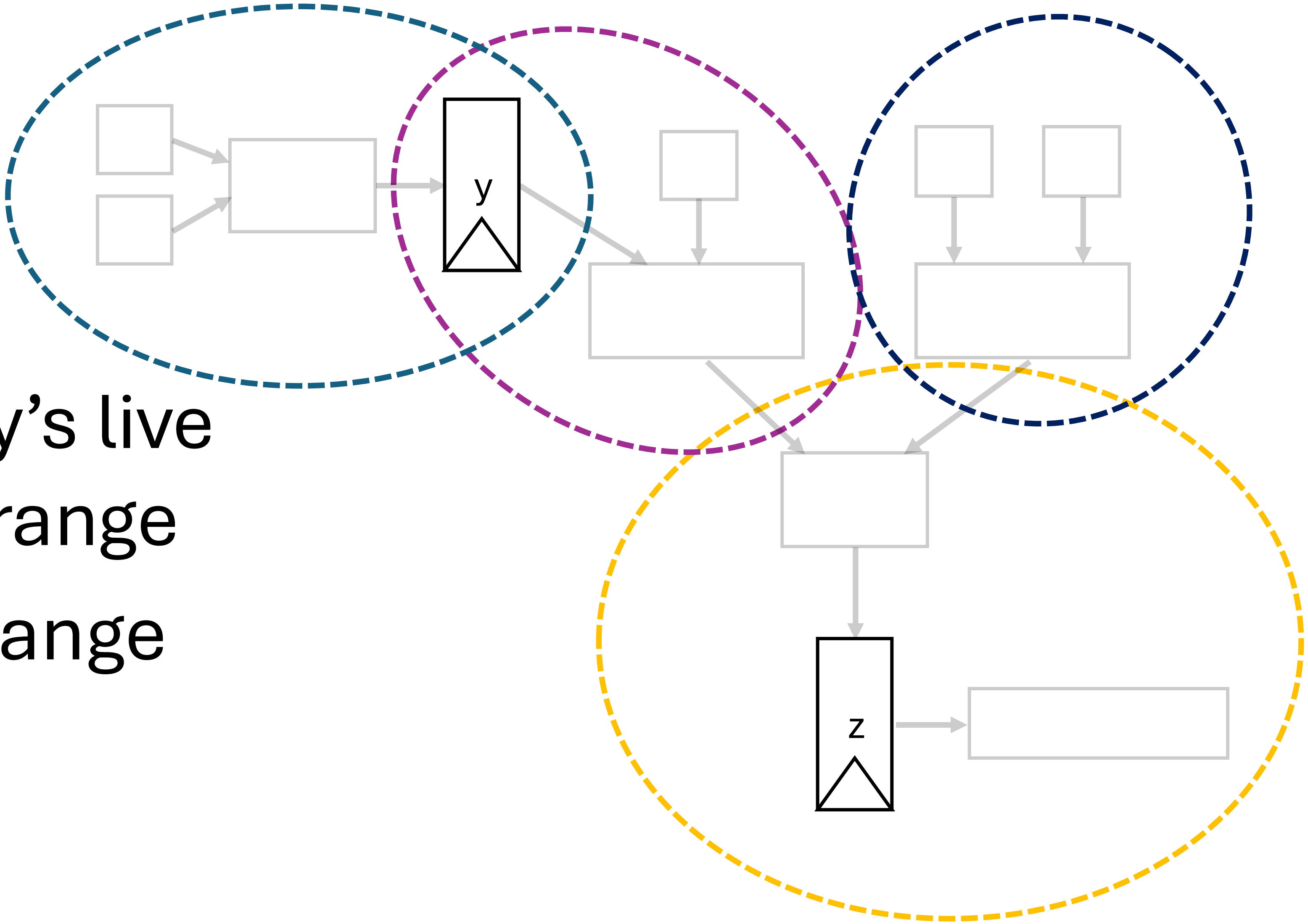
$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$

$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

```
control {
  seq {

     static par {


     }

  }
}
```

y's live range

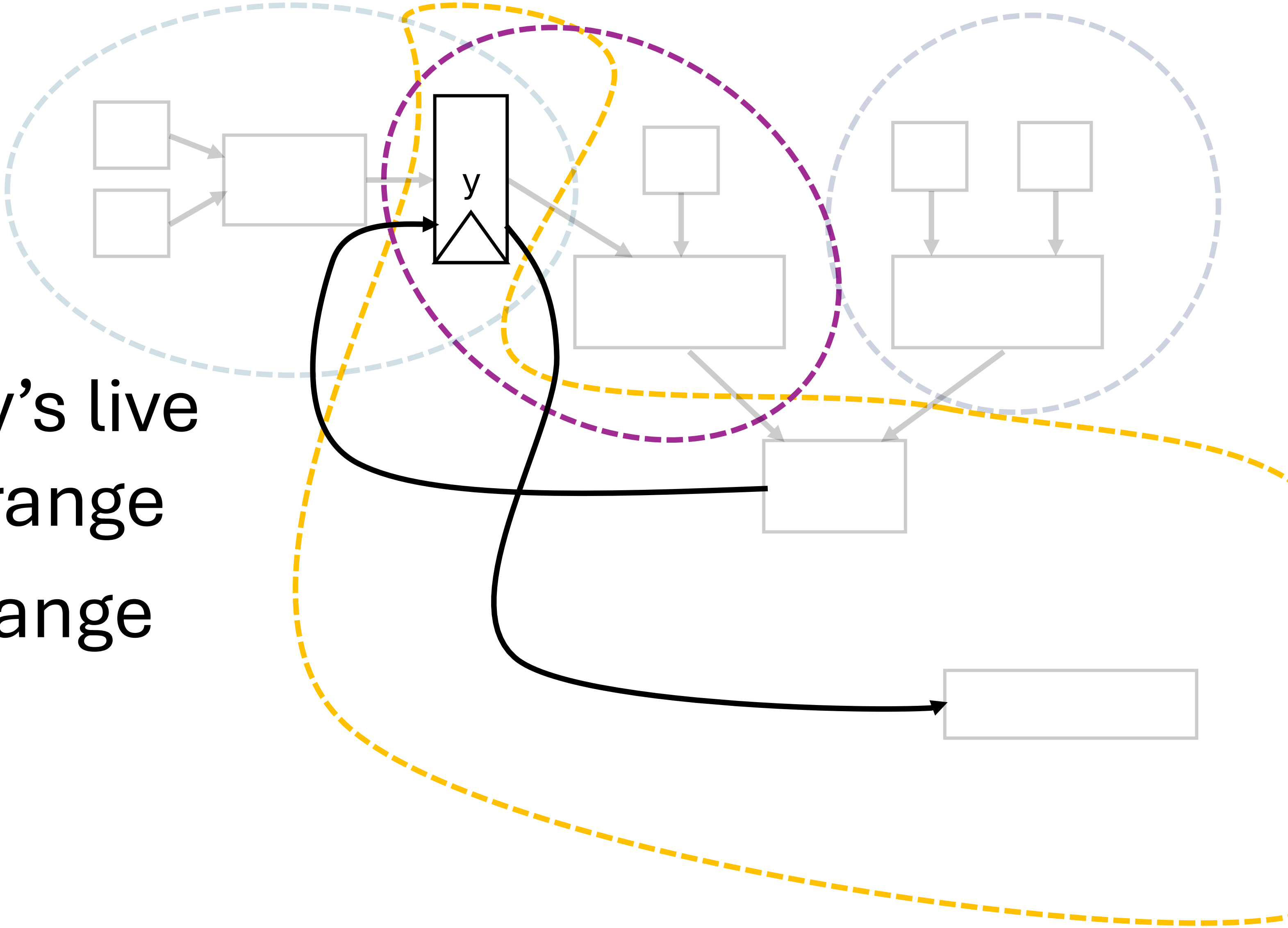
z's live range



```
control {
  seq {
    static par {
    }
  }
}
```

y's live range

z's live range

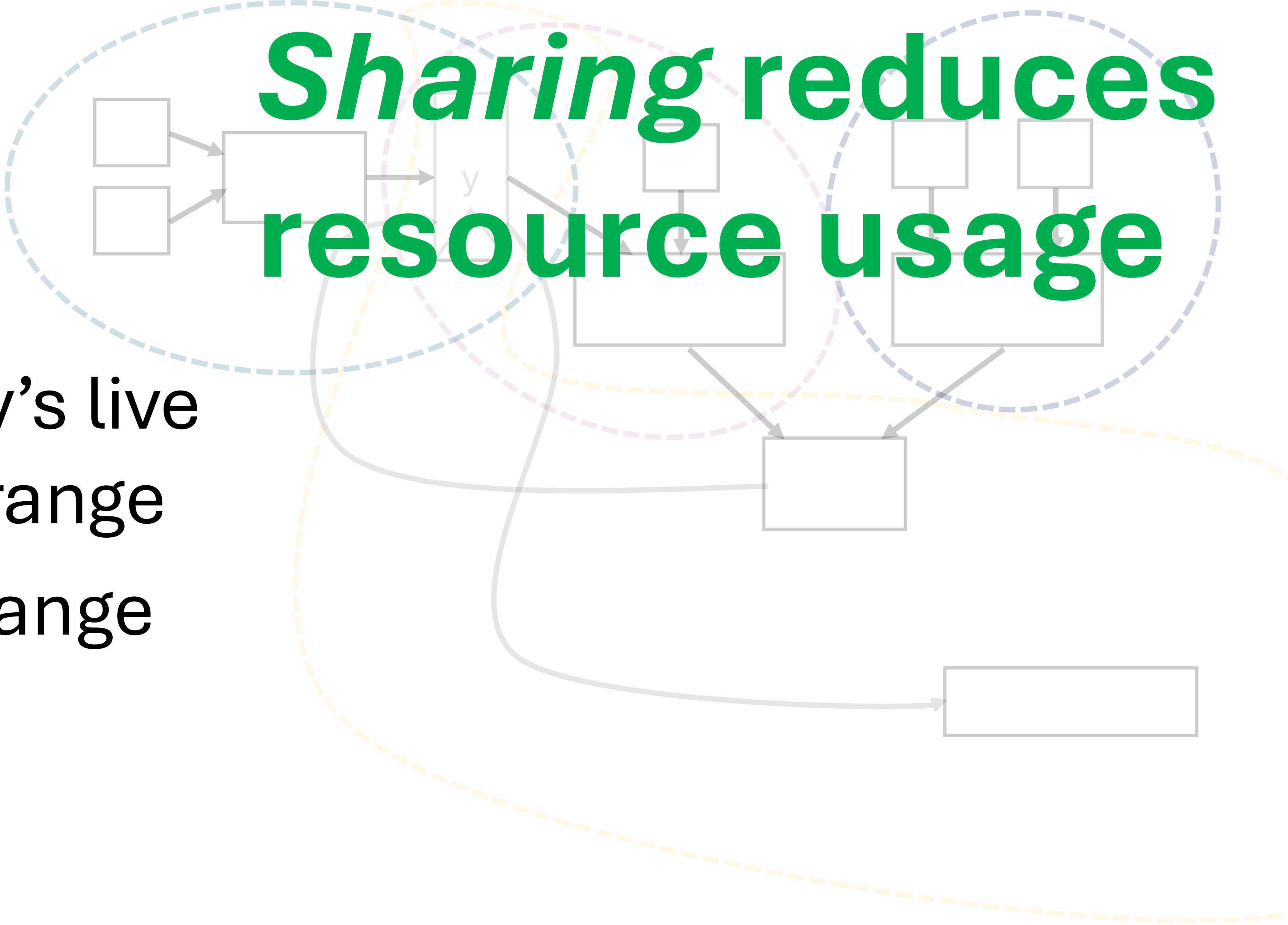


Sharing reduces resource usage

```
control {
  seq {

    static par {

    }

  }
}
```

y's live range

z's live range

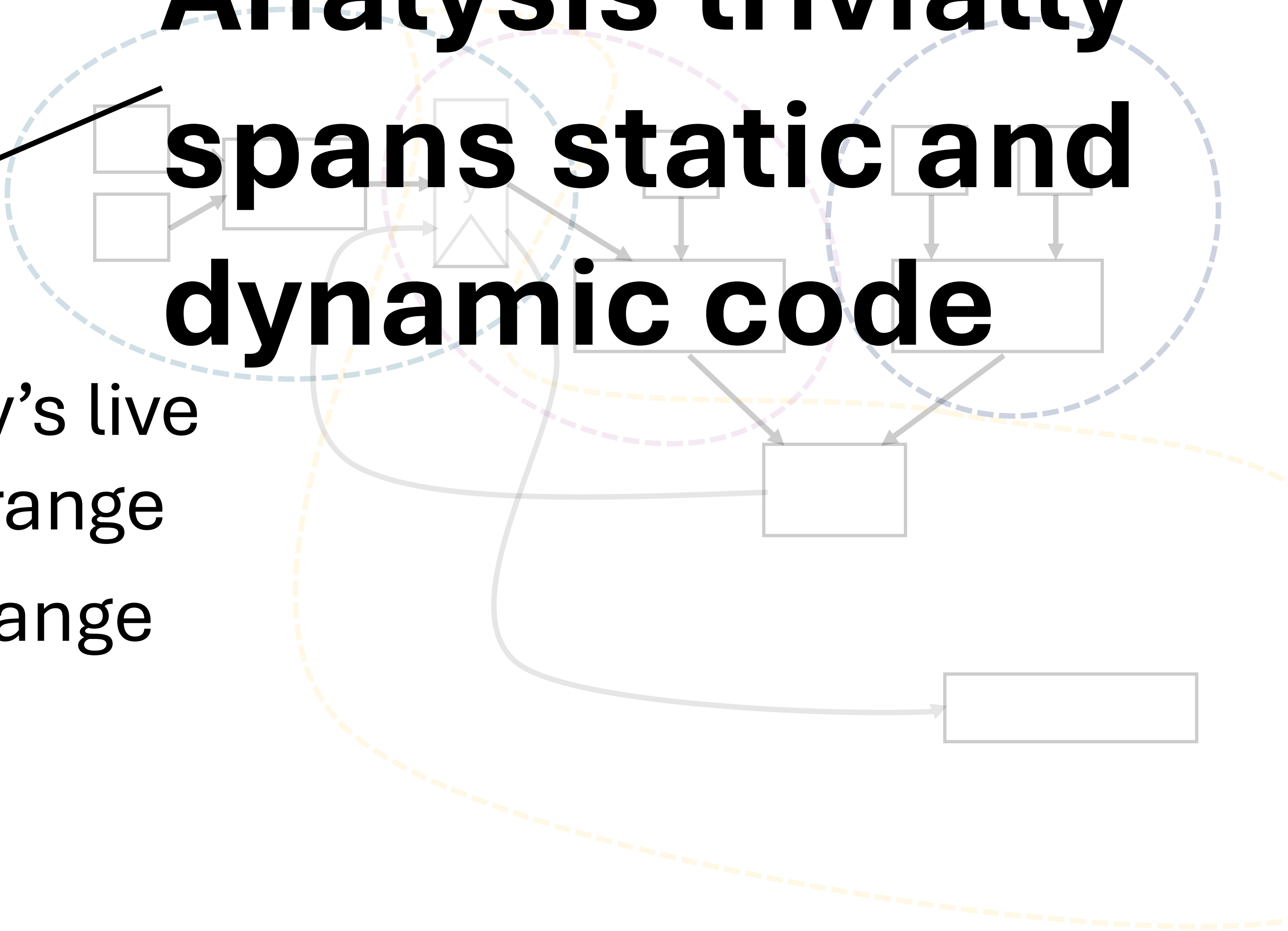


Analysis trivially spans static and dynamic code

```
control {
  seq {
      static par {
      }
  }
}
```

y’s live range

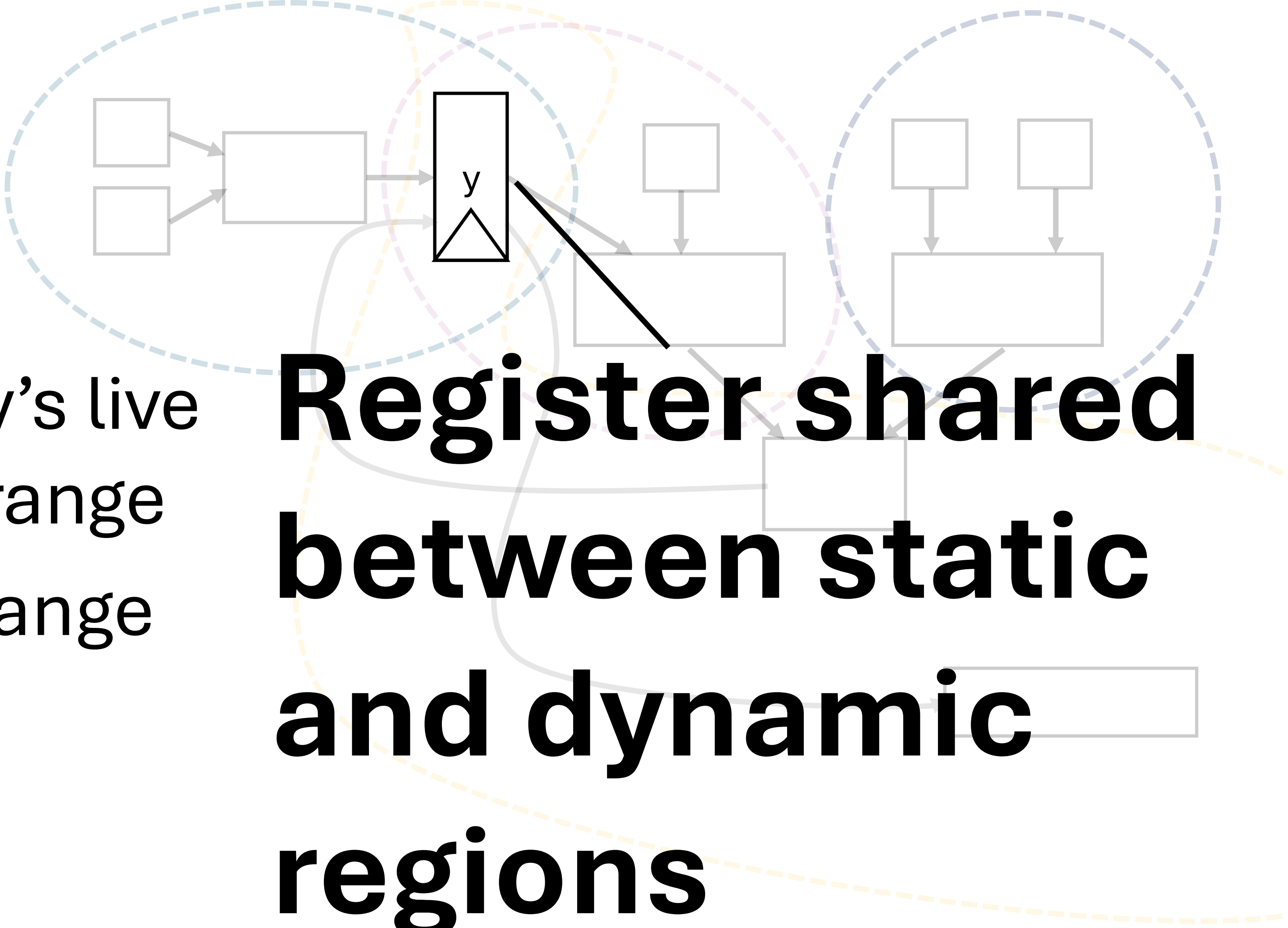
z’s live range



```
control {
  seq {
      static par {
      }
  }
}
```

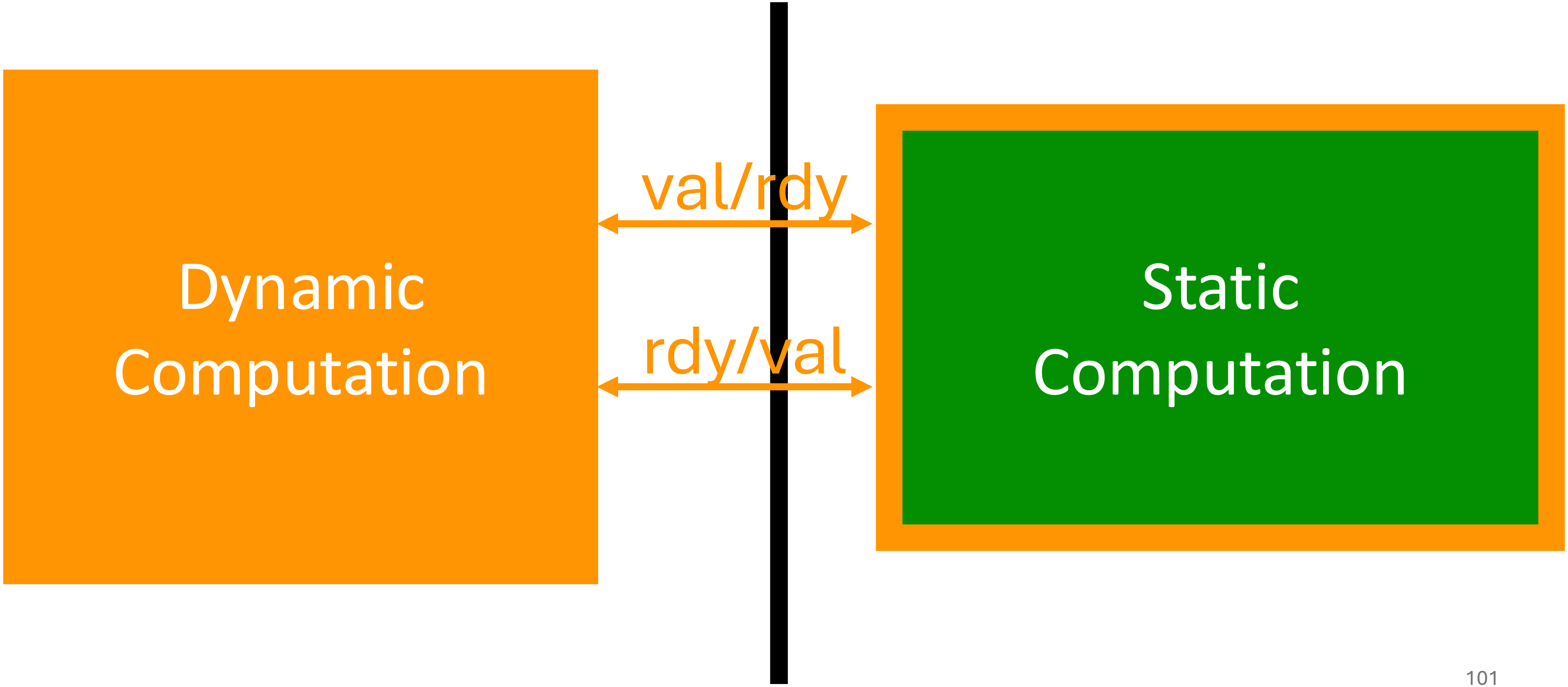
y’s live range

z’s live range



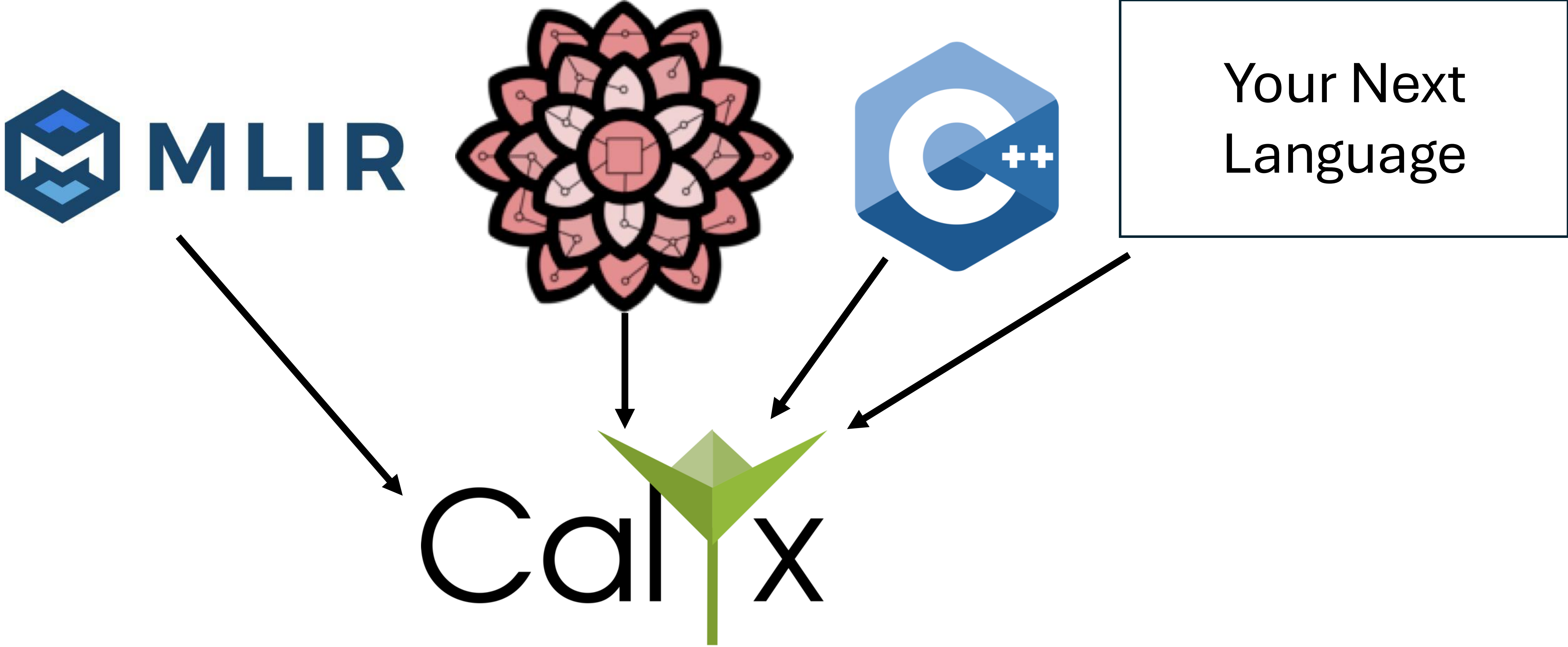
Register shared between static and dynamic regions

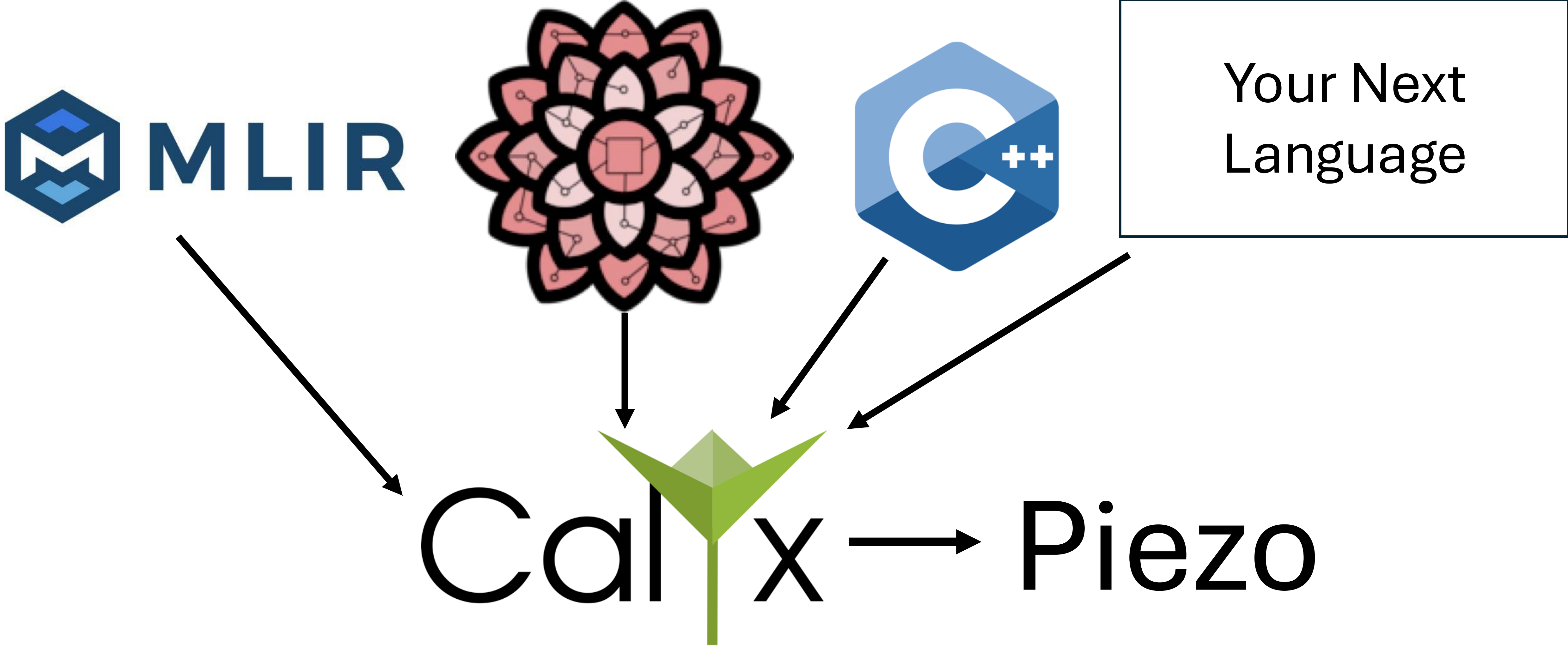
Reminder: Unclear how to accomplish this in a stratified IR

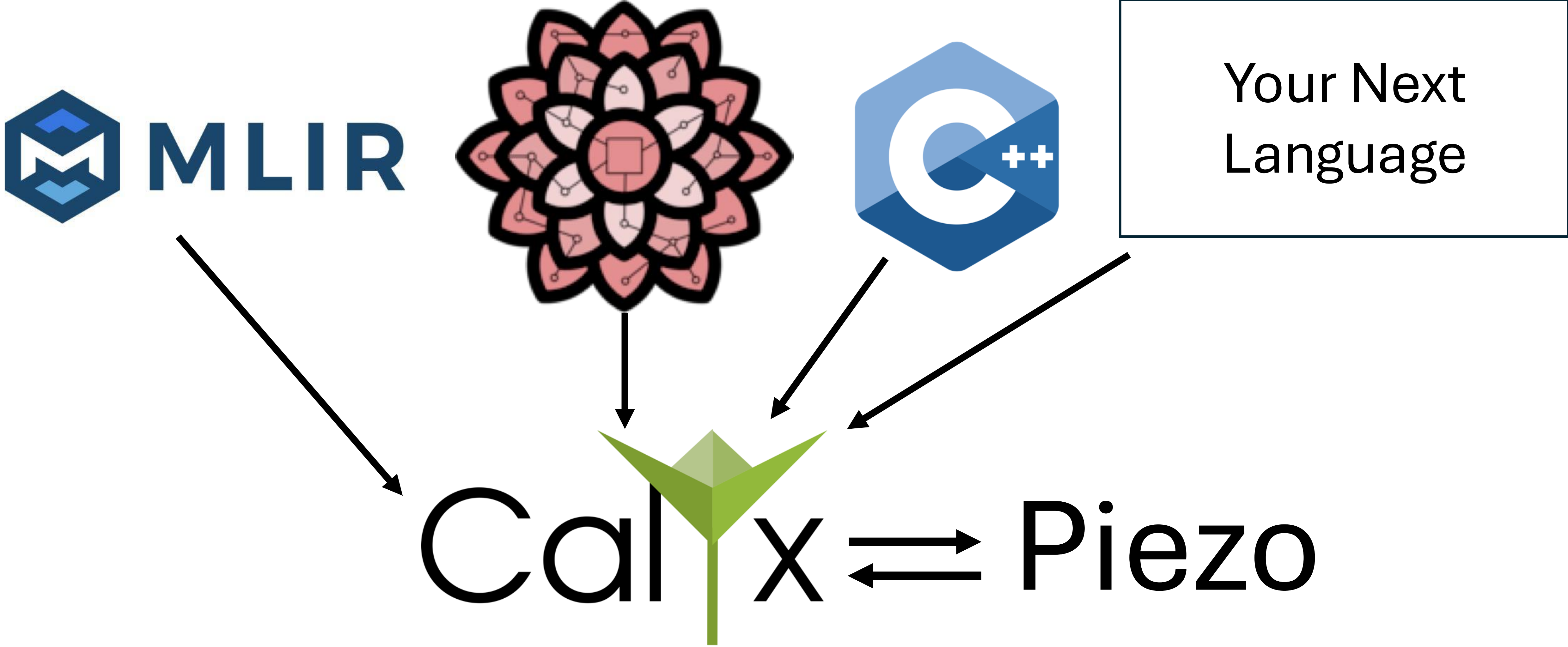


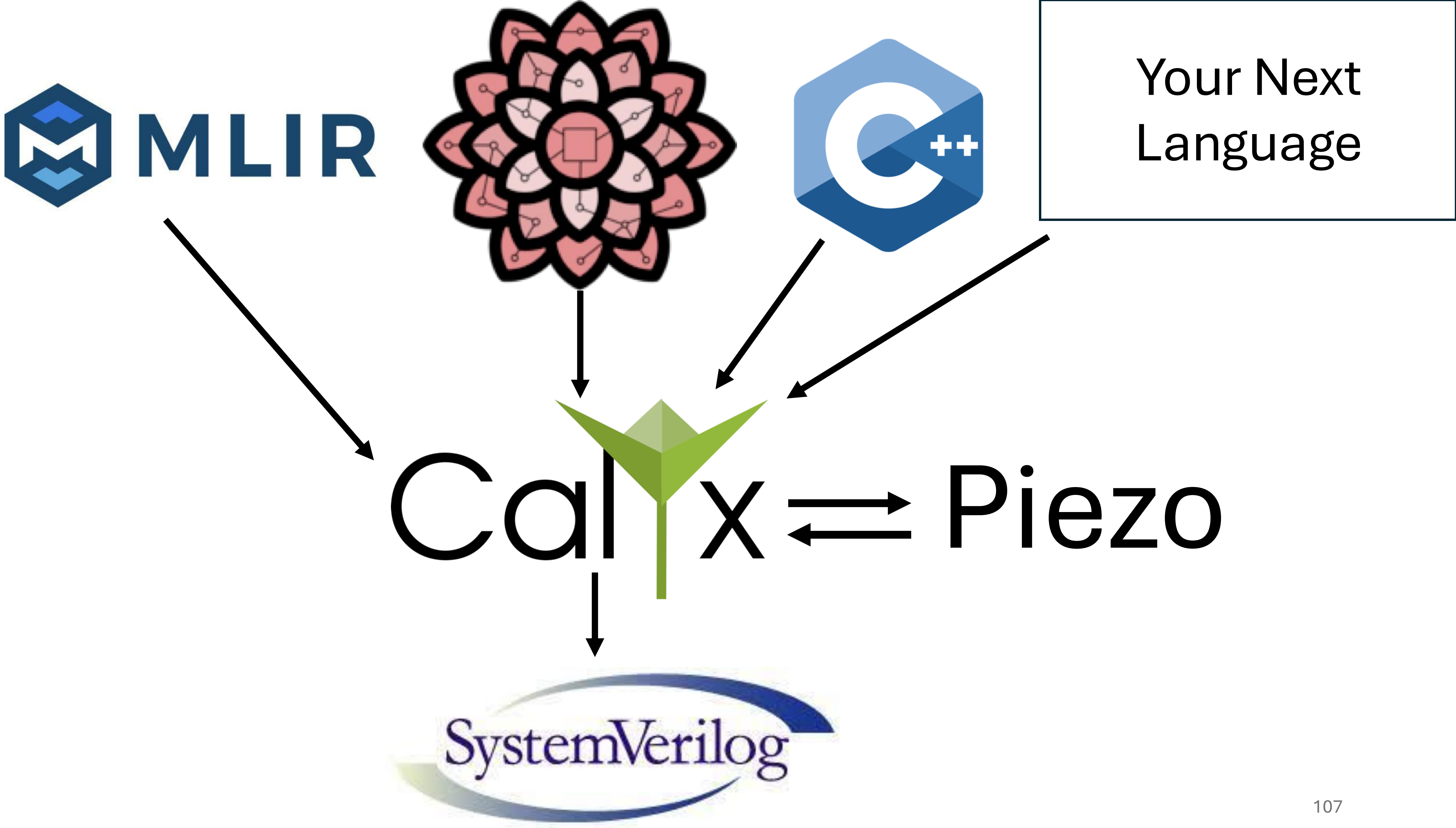
Key Idea #4: Unification lets you write compiler optimizations that were **difficult** or **impossible** before

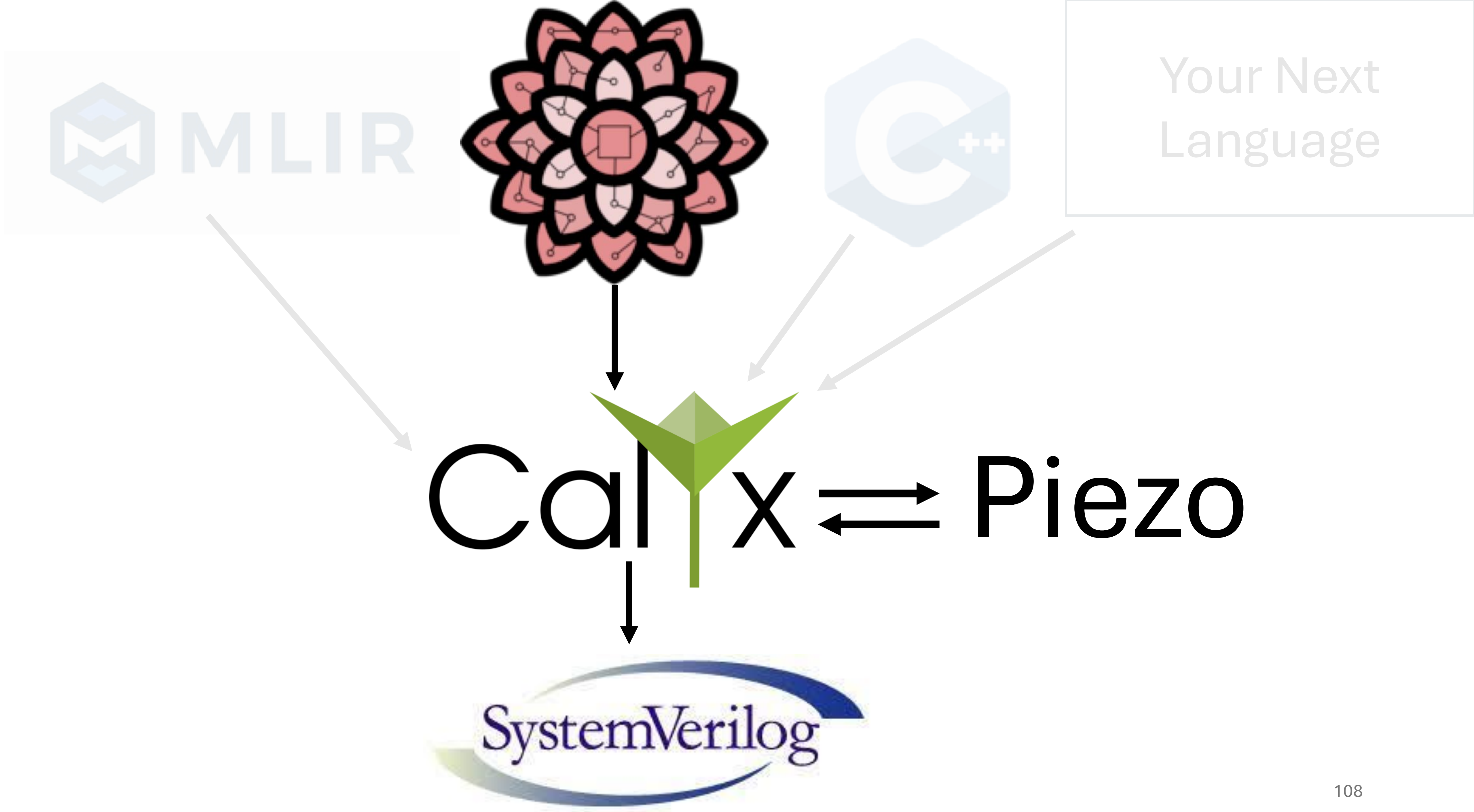
Evaluation


MLIR
Your Next Language
Calyx

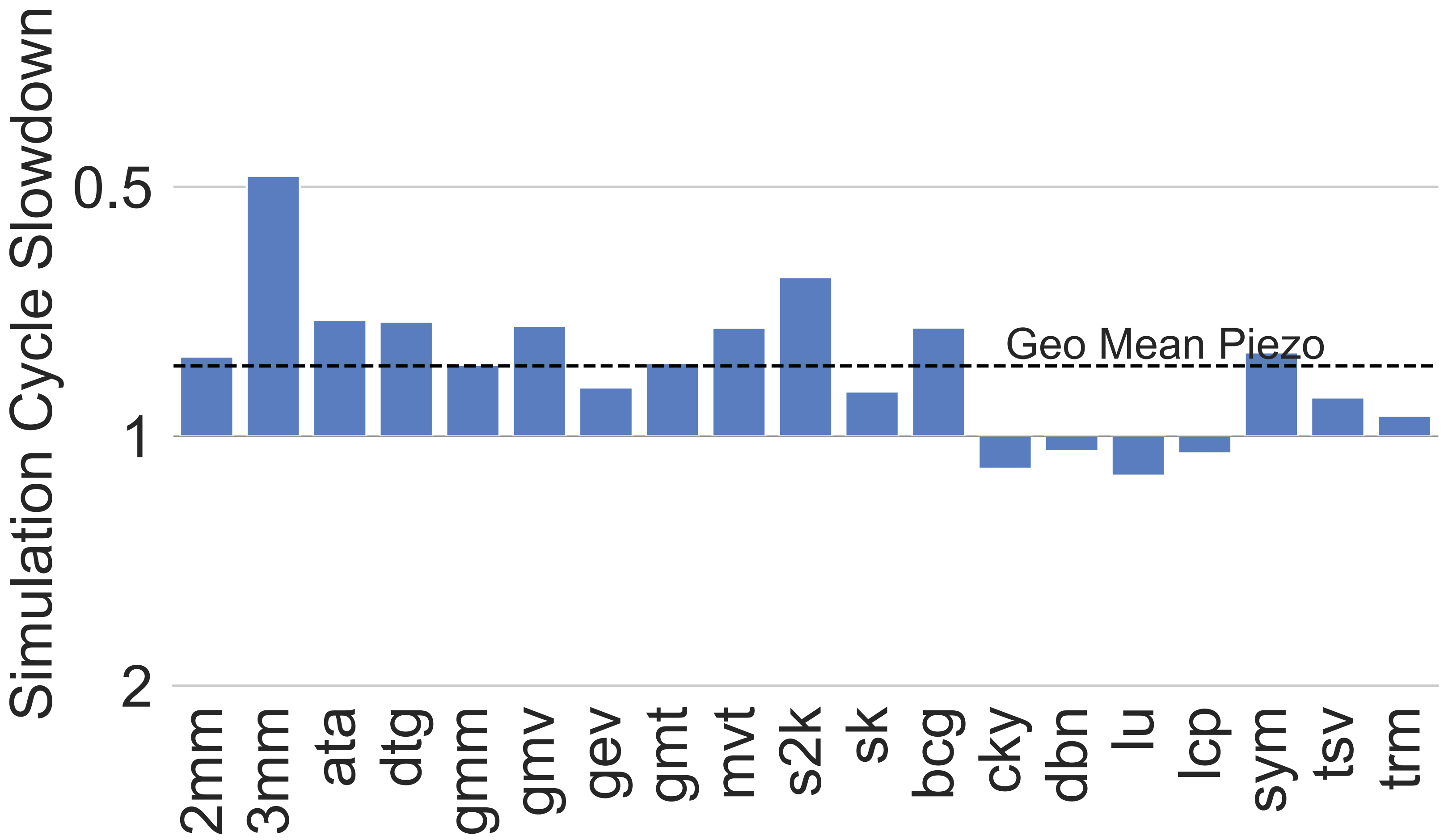

MLIR
Your Next Language
Calyx
Piezo


MLIR
Your Next Language
Calyx
Piezo


MLIR
Your Next Language
Calyx
Piezo
SystemVerilog

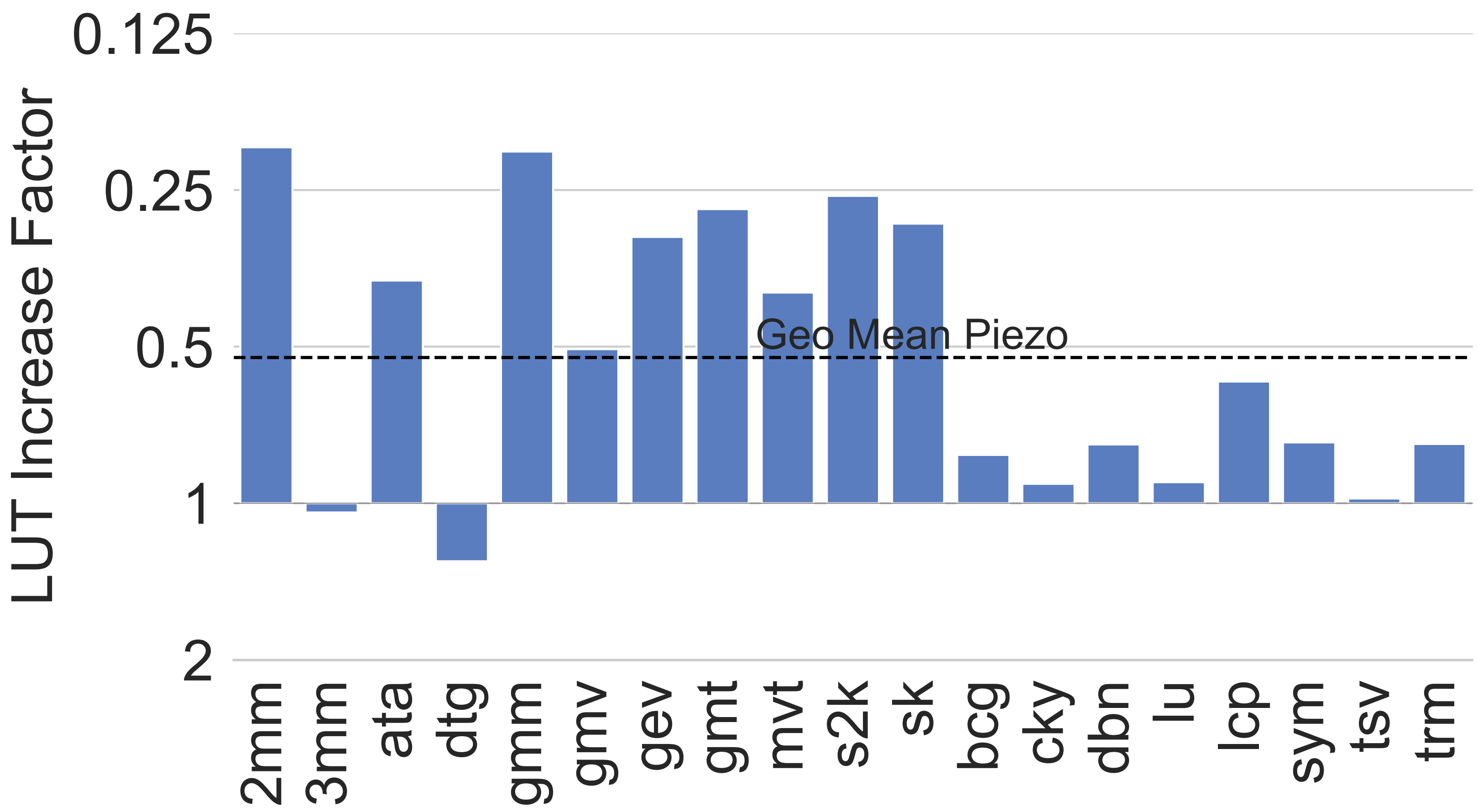

MLIR
Your Next
Language
Calyx
Piezo
SystemVerilog

Performance (Cycle Count)



Piezo Normalized to Calyx (Higher is better)

Area (LUT)



Piezo Normalized to Calyx (Higher is better)

Recap

Key Idea #1: Two paradigms (**static** and **dynamic**) for hardware compiler IRs with severe **trade-offs**

Key Idea #2: Existing compilers IRs **stratify** static and dynamic sections into **separate languages/representations**

Key Idea #3: Our work uses the idea of **semantic refinement** to **unify** static and dynamic sub-languages

Key Idea #4: Unification lets you write compiler optimizations that were **difficult** or **impossible** before

Another Optimization

```
seq {A; B; C; D;}
```

```
seq {A; B; C; D;}
```

```
group A {
   reg.in = 10;
   reg.write_en = 1;
   A[done] = reg.done;
}
```

seq {A; B; C; D;}

```
group A {
    reg.in = 10;
    reg.write_en = 1;
    A[done] = reg.done;
}
```

Always takes 1 cycle

$$seq\ \{A_1;\ B;\ C;\ D;\}$$

```
static<1> group A {
    reg.in = 10;
    reg.write_en = 1;
}
```

seq {A_1; B; C; D;}

```
static<1> group A{
    reg.in = 10;
    reg.write_en = 1;
}
```

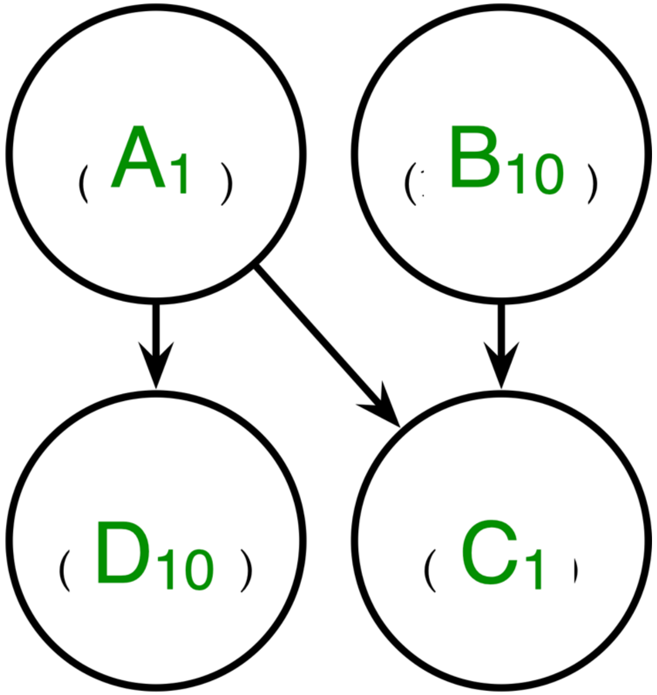
Promotion

seq {A_1; B; C; D;}

seq $\{A_1; B_{10}; C_1; D_{10};\}$

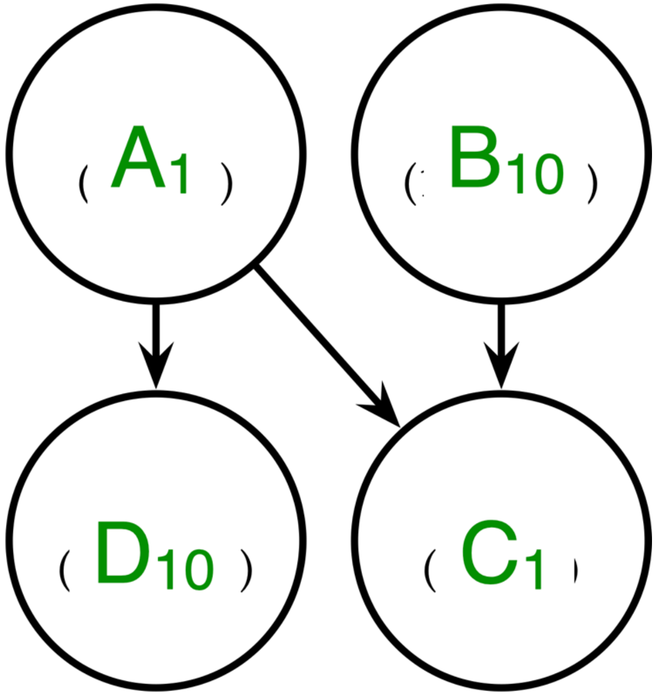
$\mathrm{seq}_{22}\ \{A_1;\ B_{10};\ C_1;\ D_{10};\}$

seq_{22} {A_1; B_{10}; C_1; D_{10};}



Dependency Graph

$$
\begin{aligned}
&\mathrm{par}_{11}\ \{\\
&\quad \mathrm{seq}_{11}\ \{A_1; D_{10};\}\\
&\quad \mathrm{seq}_{11}\ \{B_{10}; C_1;\}\\
&\}
\end{aligned}
$$



Dependency Graph

$$
\begin{aligned}
&\mathrm{par}_{11}\ \{\\
&\quad \mathrm{seq}_{11}\ \{A_1;D_{10};\}\\
&\quad \mathrm{seq}_{11}\ \{B_{10};C_1;\}\\
&\}
\end{aligned}
$$

Compaction

```
seq {A; B; C; D;}
```

seq {A; B; C; D;}

$\downarrow$

```
par₁₁ {
  seq₁₁ {A₁;D₁₀;}
  seq₁₁ {B₁₀;C₁;}
}
```

$seq\ \{A;\ B;\ C;\ D;\}$

$\big\downarrow$ *Refinement*

$par_{11}\ \{$

$seq_{11}\ \{A_1;D_{10};\}$

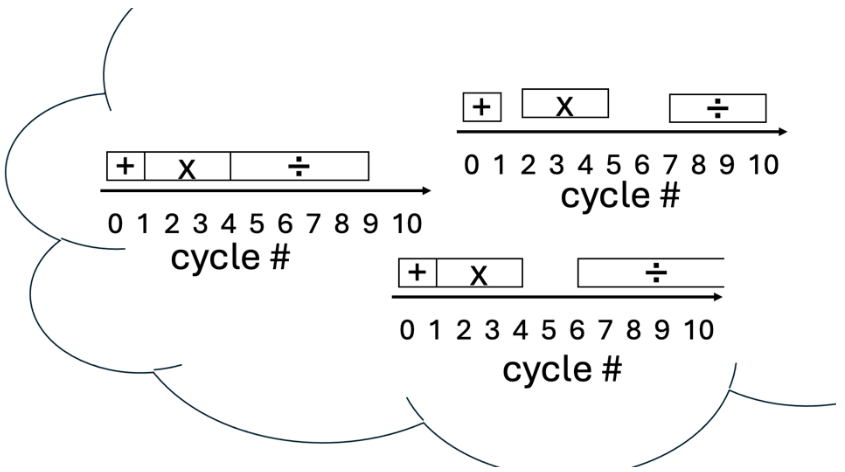
$seq_{11}\ \{B_{10};C_1;\}$

$\}$

seq {A; B; C; D;}

Refinement

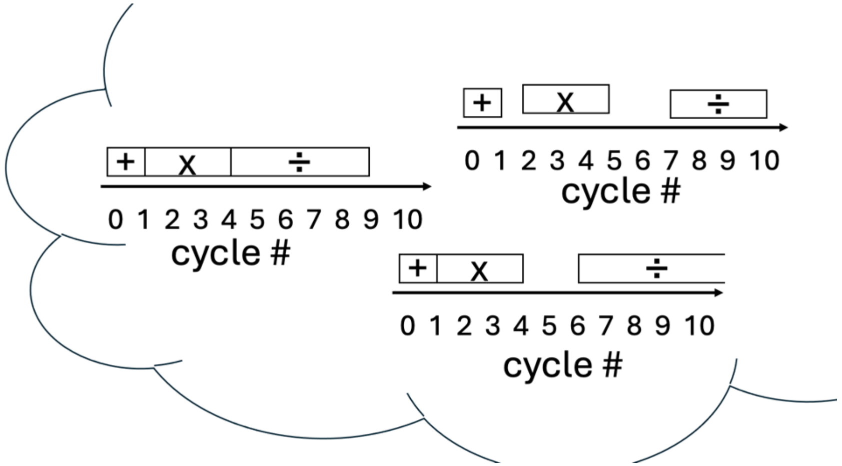
par_{11} {
  seq_{11} {A_1;D_{10};}
  seq_{11} {B_{10};C_1;}
}

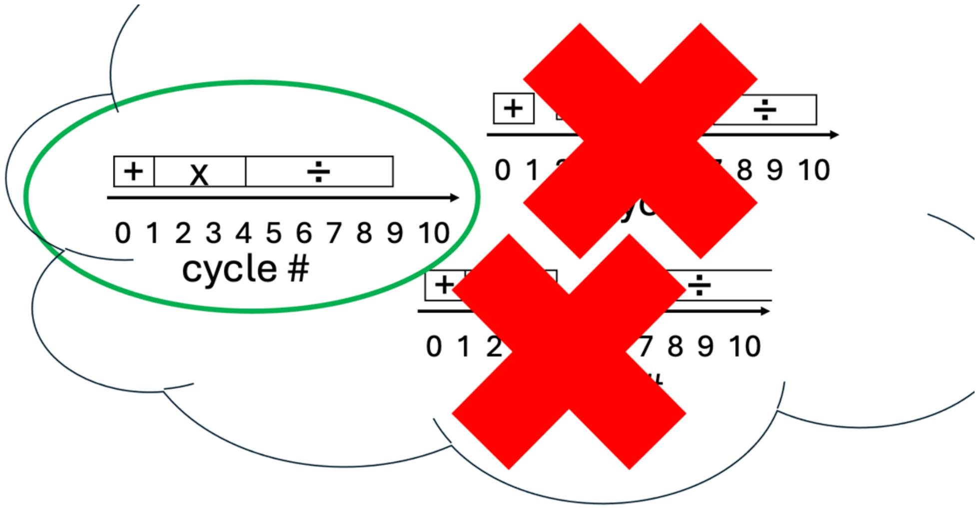


$\texttt{seq}\ \{A;\ B;\ C;\ D;\}$

$\big\downarrow$ *Refinement*

$\texttt{par}_{11}\ \{$
$\quad \texttt{seq}_{11}\ \{A_1; D_{10};\}$
$\quad \texttt{seq}_{11}\ \{B_{10}; C_1;\}$
$\}$





How does this unified approach help us?