Setup

- A: Download the docker image
  - Please follow the handout or our website
  - https://ericlyun.me/tutorial-asplos2025

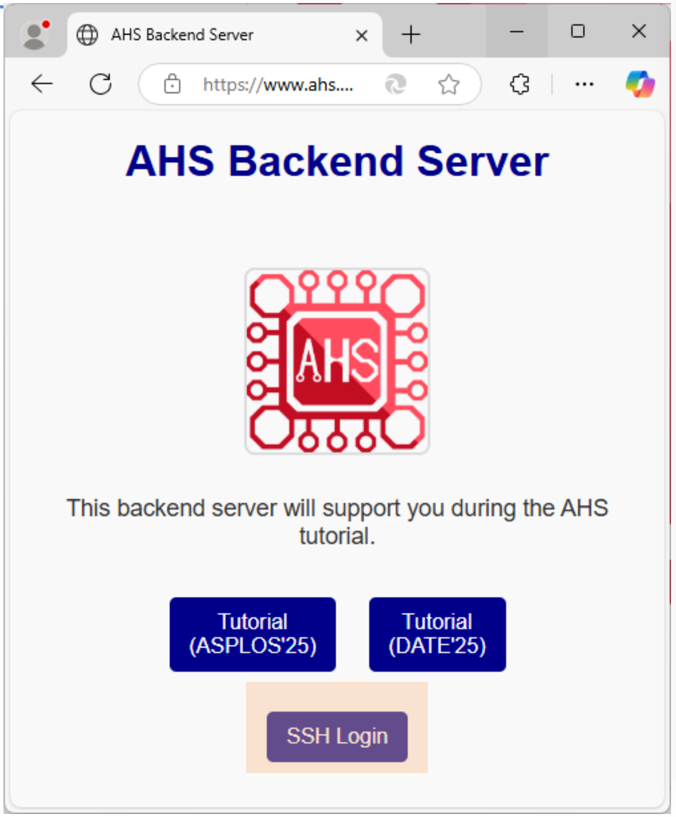
- B: Login our server
  - First, you need a SSH key
  - check if your SSH directory exists
    - win: C:\Users\<username>/.ssh
    - linux: $HOME/.ssh
    - mac: /Users/<username>/.ssh
  - if not exists, you need to create one
    - reference link

```
# Windows, open PowerShell
⌨  ssh-keygen -t ed25519 -C "<email>"
# Linux, open terminal
⌨  ssh-keygen -t ed25519 -C "<email>"
# Mac, open terminal
⌨  ssh-keygen -t ed25519 -C "<email>"
```

SSH public key can be found in <SSH>/id_ed25519.pub

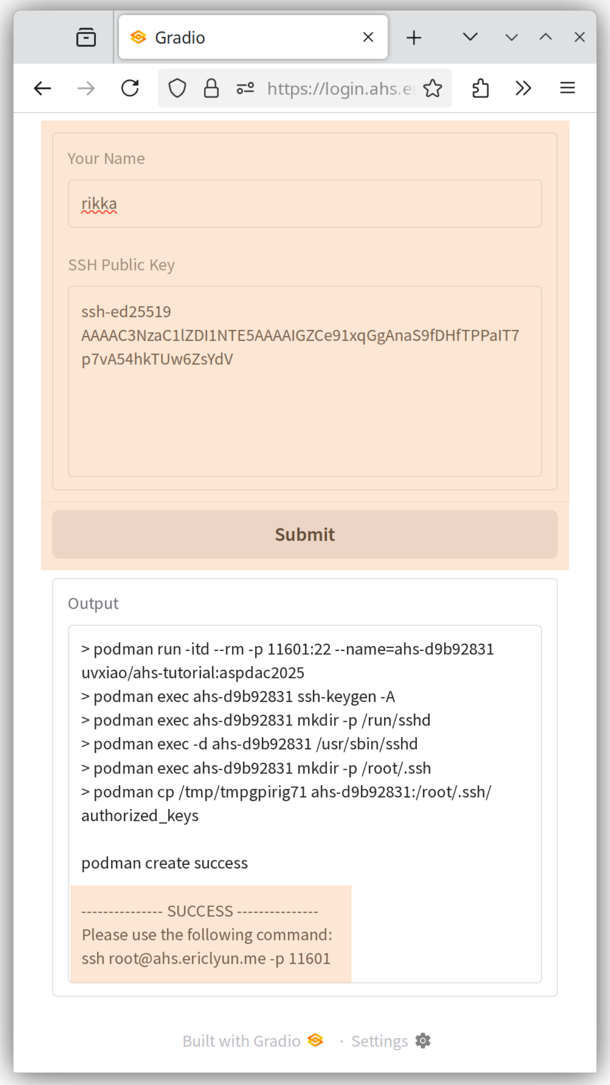
Setup

- A: Download the docker image
  - Please follow the handout or our website
  - https://ericlyun.me/tutorial-asplos2025
- B: Login our server
  - www.ahs.ericlyun.me
  - Click SSH Login
  - Enter your name and ssh key
  - Follow the output to connect



AHS Backend Server

This backend server will support you during the AHS tutorial.



Your Name

rikka

SSH Public Key

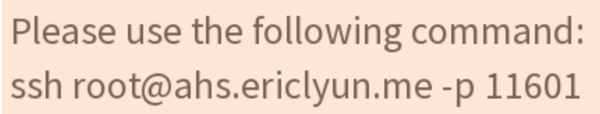
```
ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAIGZCe91xqGgAnaS9fDHfTPPaIT7
p7vA54hkTUw6ZsYdV
```

Output

```
> podman run -itd --rm -p 11601:22 --name=ahs-d9b92831
uvxiao/ahs-tutorial:aspdac2025
> podman exec ahs-d9b92831 ssh-keygen -A
> podman exec ahs-d9b92831 mkdir -p /run/sshd
> podman exec -d ahs-d9b92831 /usr/sbin/sshd
> podman exec ahs-d9b92831 mkdir -p /root/.ssh
> podman cp /tmp/tmpgpirig71 ahs-d9b92831:/root/.ssh/
authorized_keys

podman create success

-------------- SUCCESS --------------
Please use the following command:
ssh root@ahs.ericlyun.me -p 11601
```

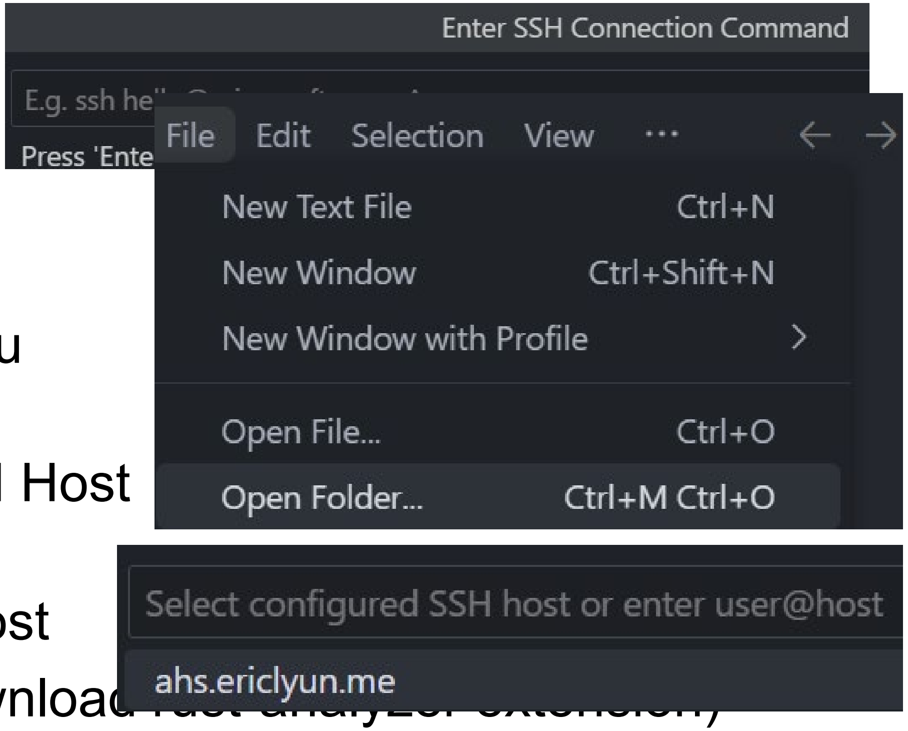
Setup

- A: Download the docker image
  - Please follow the handout or our website
  - https://ericlyun.me/tutorial-asplos2025



Please use the following command:
ssh root@ahs.ericlyun.me -p 11601

- B: Login our server
  - You can use PowerShell/terminal
  - Or, you can use VSCode
    - Ctrl-Shift-P / ⇧⌘P, type Remote: Show Remote Menu
      - select SSH to install the extension
    - Ctrl-Shift-P / ⇧⌘P, type Remote-SSH: Add New SSH Host
      - paste “ssh root@ahs.ericlyun.me –p xxxxx”
    - Ctrl-Shift-P / ⇧⌘P, type Remote-SSH: Connect to Host
    - Ctrl-K + Ctrl-O, open /root/repos folder (optional: download

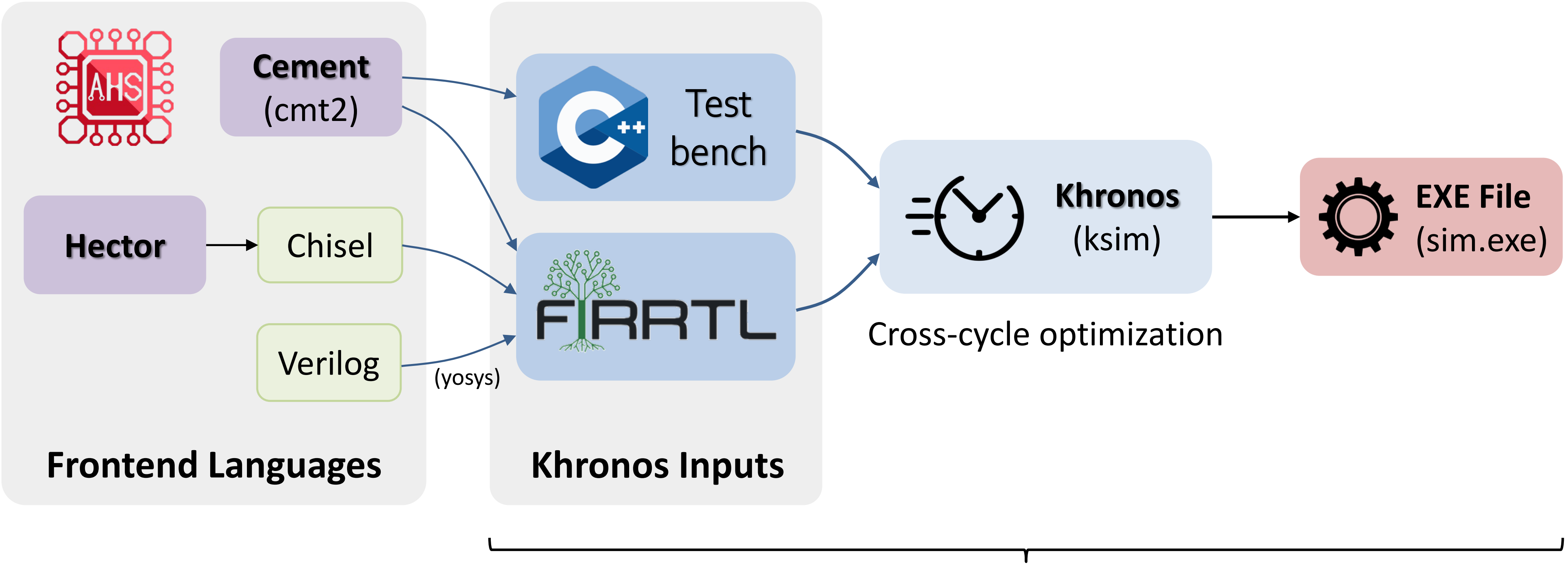


RTL Simulator with Cross-Cycle Optimization

Presenter: Youwei Xiao

Author: Kexing Zhou

Khronos: RTL Simulator with Cross-Cycle Optimization



Covered in **This Section**

Outline

- Compilation workflow of Khronos
  - Input Preparation
  - Header and Code Generation
  - Testbench Preparation
  - Compile and Linking
- Intermediate Representation of Khronos
  - Flattening, Fusing, Lowering, Queuing
- Real Application Evaluation

Outline

- Compilation workflow of Khronos
  - Input Preparation
  - Header and Code Generation
  - Testbench Preparation
  - Compile and Linking
- Intermediate Representation of Khronos
  - Flattening, Fusing, Lowering, Queuing
- Real Application Evaluation

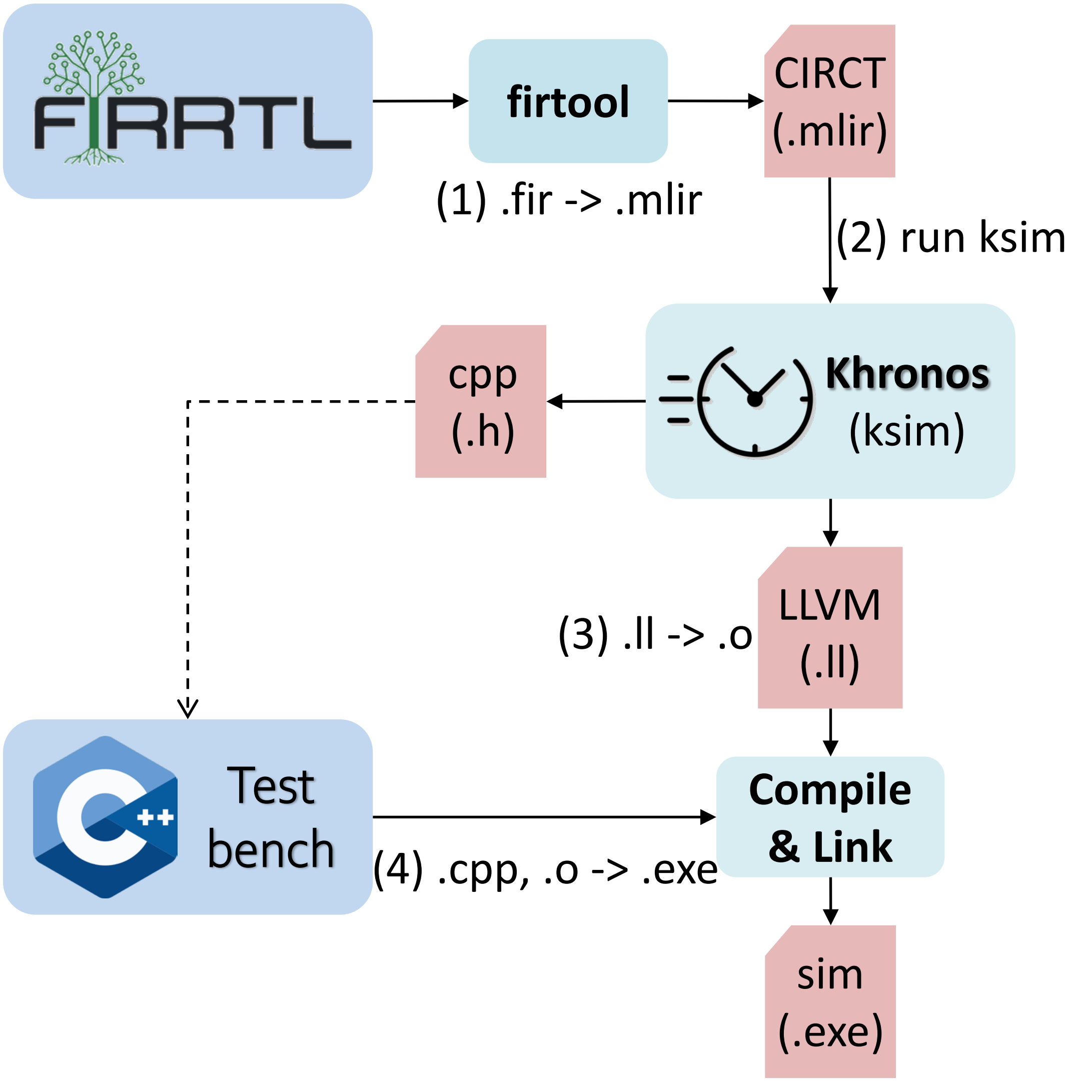
Compilation Flow of Khronos

- Khronos is based on MLIR & CIRCT

1. `firtool`: Translate input to CIRCT IR
2. `ksim`:
   - Translate hw logic to llvmir
   - Generate header file for testbench
3. `llc`: compile llvmir to linkable object
4. `clang++`: compile and link testbench

Command Cheet Sheet

```
(1) firtool-ksim --ir-hw $1.fir -o $1.mlir
(2) ksim $1.mlir -v -o $1.ll --out-header=$1.h
(3) llc-ksim -O2 -filetype=obj $1.ll -o $1.o
(4) clang++ -O2 $1.o $1.cpp -o $1
```



FIRRTL

firtool

CIRCT (.mlir)

(1) .fir -> .mlir

(2) run ksim

Khronos (ksim)

cpp (.h)

LLVM (.ll)

(3) .ll -> .o

Test bench

Compile & Link

(4) .cpp, .o -> .exe

sim (.exe)

Khronos Flow: Prepare Input

simplefir.fir

```
circuit SimpleFIR:
  module SimpleFIR:
    input clock: Clock
    input reset: UInt<1>
    input in:    SInt<16>
    output sum:  SInt<16>

    reg r1:      SInt<16>, clock
    reg r2:      SInt<16>, clock
    reg r3:      SInt<16>, clock
    r1 <= in
    r2 <= r1
    r3 <= r2

    sum <= add(add(in, r1), add(r2, r3))
```

Download fir file from our website (or copy and paste it)

```
wget https://ericlyun.me/tutorial-asplos2025/rsrc/khronos/simplefir.fir \
          -O simplefir.fir
```

Top modules names the same as circuit

Basic type, `UInt` and `SInt`

Use `<=` to connect signals

Translate it to CIRCT IR

```
firtool-ksim simplefir.fir -ir-hw \
  -o simplefir.mlir
```

module => `hw.module`

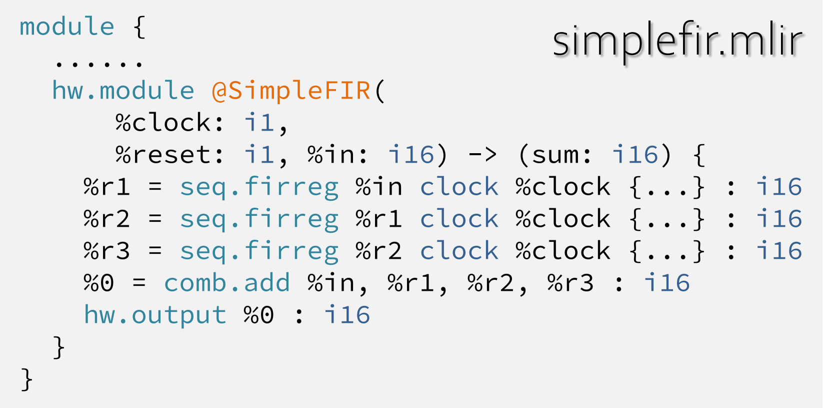
registers => `seq.firreg`

comb logics => `comb.xxx`

simplefir.mlir

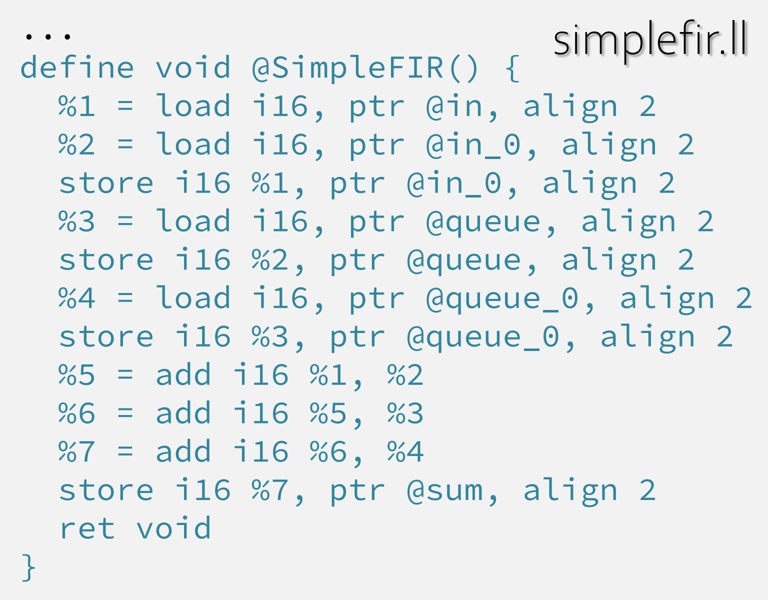
```
module {
  ......
  hw.module @SimpleFIR(
      %clock: i1,
      %reset: i1, %in: i16) -> (sum: i16) {
    %r1 = seq.firreg %in clock %clock {...} : i16
    %r2 = seq.firreg %r1 clock %clock {...} : i16
    %r3 = seq.firreg %r2 clock %clock {...} : i16
    %0 = comb.add %in, %r1, %r2, %r3 : i16
    hw.output %0 : i16
  }
}
```

Khronos Flow: Generate Header and IR



simplefir.mlir

```
module {
  ......
  hw.module @SimpleFIR(
      %clock: i1,
      %reset: i1, %in: i16) -> (sum: i16) {
    %r1 = seq.firreg %in clock %clock {...} : i16
    %r2 = seq.firreg %r1 clock %clock {...} : i16
    %r3 = seq.firreg %r2 clock %clock {...} : i16
    %0 = comb.add %in, %r1, %r2, %r3 : i16
    hw.output %0 : i16
  }
}
```



simplefir.ll

```
...
define void @SimpleFIR() {
  %1 = load i16, ptr @in, align 2
  %2 = load i16, ptr @in_0, align 2
  store i16 %1, ptr @in_0, align 2
  %3 = load i16, ptr @queue, align 2
  store i16 %2, ptr @queue, align 2
  %4 = load i16, ptr @queue_0, align 2
  store i16 %3, ptr @queue_0, align 2
  %5 = add i16 %1, %2
  %6 = add i16 %5, %3
  %7 = add i16 %6, %4
  store i16 %7, ptr @sum, align 2
  ret void
}
```

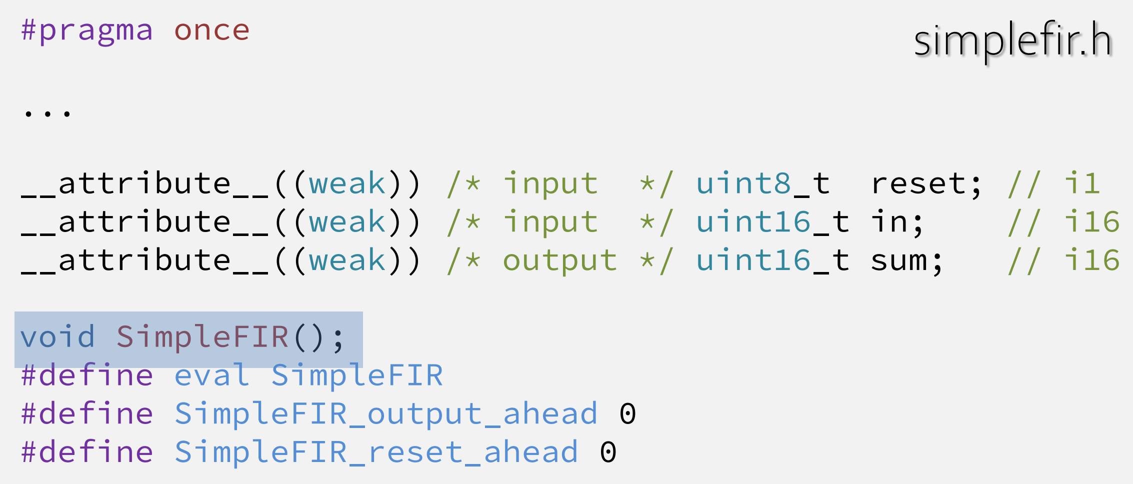
Compile the file using Khronos

```
ksim simplefir.mlir \
    --out-header=simplefir.h \
    -o simplefir.ll
```

.ll file contains the logic of simulator

.h file contains IO signals

- `clock` is removed in IO signals
- call the eval function to advance 1 cycle



simplefir.h

```
#pragma once

...

__attribute__((weak)) /* input  */ uint8_t  reset; // i1
__attribute__((weak)) /* input  */ uint16_t in;    // i16
__attribute__((weak)) /* output */ uint16_t sum;   // i16

void SimpleFIR();
#define eval SimpleFIR
#define SimpleFIR_output_ahead 0
#define SimpleFIR_reset_ahead 0
```

Khronos Flow: Prepare Testbench

Khronos can generate a skeleton testbench

```
ksim simplefir.mlir \
    --out-driver=simplefir.cpp \
    -o simplefir.ll
```

simplefir.cpp

```
int main(int argc, char ** argv) {
  int cnt = atoi(argv[1]);
  reset = 1;
  for(auto i = SimpleFIR_reset_ahead; i >= 0; i--) {
    SimpleFIR();
    reset = 0;
  }
  auto start = system_clock::now();
  for(auto i = 0; i < cnt; i++) {
    SimpleFIR();
  }
  auto stop = system_clock::now();
  // ...
  return 0;
}
```

Add custom peek/poke statements in the testbench

Set in port `in` to the cycle number

Print the IO signal of the circuit

simplefir.cpp

```
int main(int argc, char ** argv) {
  int cnt = atoi(argv[1]);
  reset = 1;
  for(auto i = SimpleFIR_reset_ahead; i >= 0; i--) {
    SimpleFIR();
    reset = 0;
  }
  auto start = system_clock::now();
  for(auto i = 0; i < cnt; i++) {
    in = i;
    SimpleFIR();
    printf("cycle=%d in=%d sum=%d\n", i, in, sum);
  }
  auto stop = system_clock::now();
  // ...
  return 0;
}
```

Khronos Flow: Compile, Link and Run

Use llc (llvmir compiler) to compile the generated code

```
llc-ksim --filetype=obj \
  simplefir.ll -o simplefir.o
```

Use clang++ to compile testbench and link exe file

```
clang++ simplefir.cpp simplefir.o -o simplefir
```

Run the simulator

```
./simplefir 10
```

```
❯ ./simplefir 10
cycle=0 in=0 sum=0
cycle=1 in=1 sum=1
cycle=2 in=2 sum=3
cycle=3 in=3 sum=6
cycle=4 in=4 sum=10
cycle=5 in=5 sum=14
cycle=6 in=6 sum=18
cycle=7 in=7 sum=22
cycle=8 in=8 sum=26
cycle=9 in=9 sum=30
112
```

↑

(This is elapsed time)

Outline

- Compilation workflow of Khronos
  - Input Preparation
  - Header and Code Generation
  - Testbench Preparation
  - Compile and Linking

- Intermediate Representations in Khronos
  - Flattening, Fusing, Lowering, Queuing

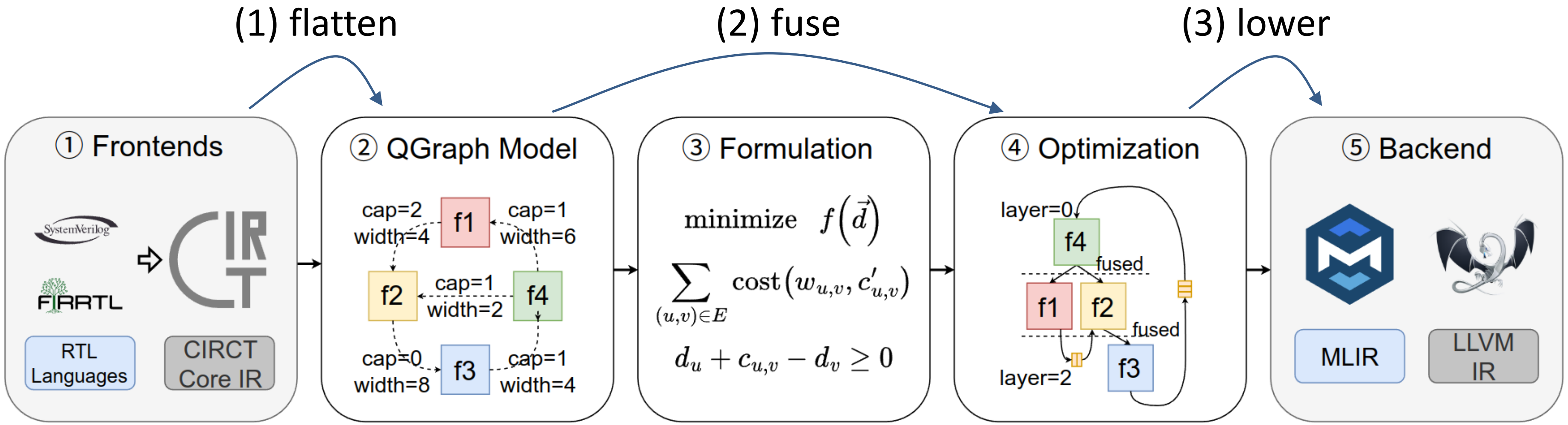
- Real Application Evaluation

Intermediate Representation of Khronos

- Khronos has many internal IR
  - `flattened`: flatten modules to build graph
  - `fused`: fuse registers
  - `low`: register => queue

```
ksim --help
```

```
--out=<value>    - Output file type
  =core          -   Default circt core IR
  =flattened     -   Ksim high level ir
  =fused         -   Ksim fused ir
  =low           -   KSim low level ir
  =llvm          -   LLVM dialect
  =llvmir        -   LLVM IR
```



Phase 1: Flatten

tree.fir

```
circuit Tree:
  module Tree:
    input clock: Clock
    input reset: UInt<1>
    input in: SInt<16>[4]
    output sum: SInt<16>

    reg r0: SInt<16>, clock
    reg r1: SInt<16>, clock
    reg r2: SInt<16>, clock
    r0 <= add(in[0], in[1])
    r1 <= add(in[2], in[3])
    r2 <= add(r0, r1)
    sum <= r2
```

Download fir file from our website (or copy and paste it)

```
wget https://ericlyun.me/tutorial-asplos2025/rsrc/khronos/tree.fir \
    -O tree.fir
```

Translate it to mlir

```
firtool-ksim tree.fir -ir-hw -o tree.mlir
```

Show ksim IR:

```
ksim tree.mlir -out=flattened -o tree.flattened.mlir
```

tree.flattened.mlir

```
module {
  hw.module @Tree(%clock: i1, %reset: i1,
      %in_0: i16, %in_1: i16,
      %in_2: i16, %in_3: i16) -> (sum: i16) {
    %r0 = seq.firreg %0 clock %clock {...} : i16
    %r1 = seq.firreg %1 clock %clock {...} : i16
    %r2 = seq.firreg %2 clock %clock {...} : i16
    %0 = comb.add %in_0, %in_1 : i16
    %1 = comb.add %in_2, %in_3 : i16
    %2 = comb.add %r0, %r1 : i16
    hw.output %r2 : i16
  }
}
```

Only one module,
flattened IR is identical to mlir

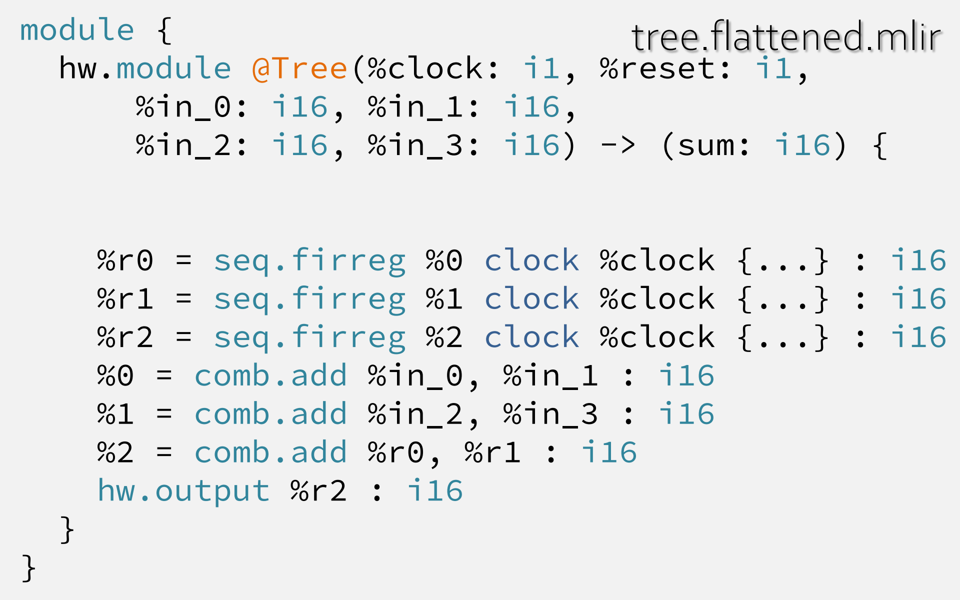
Phase 2: Fusing

Run state fusion algorithm in Khronos

```
ksim tree.flattened.mlir \
    -in=flattened -out=fused \
    -o tree.fused.mlir
```

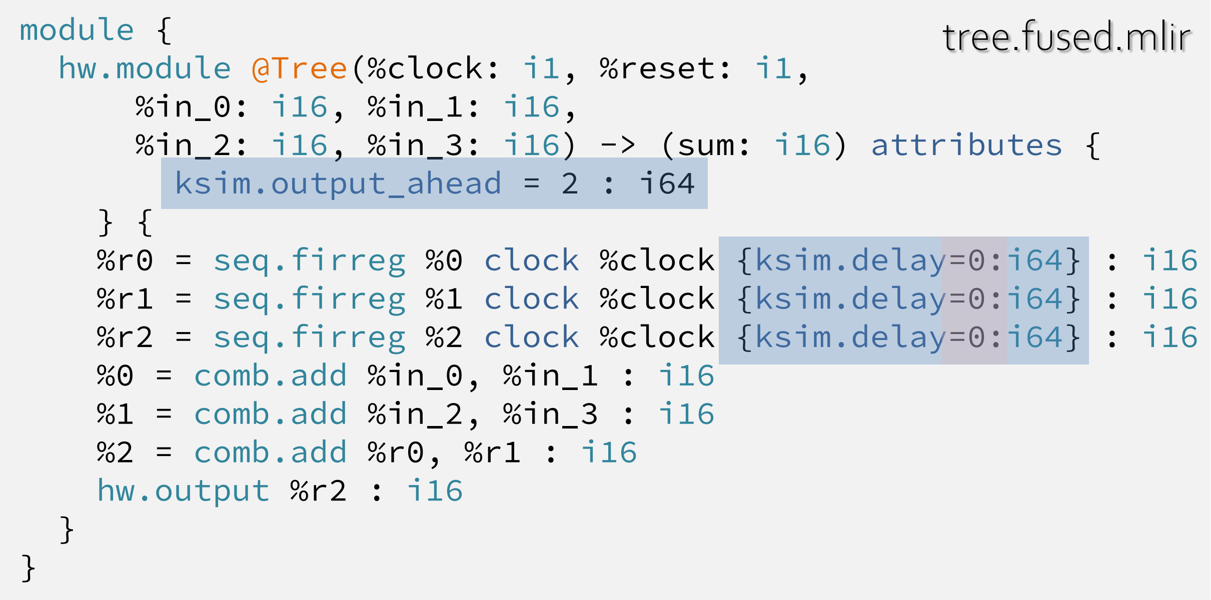
After state fusion

- “ksim.delay” attribute is attached to each register
- if delay = 0, it is fused, and will be remove later
- “output_ahead” captures how many cycles output advanced



tree.flattened.mlir

```
module {
  hw.module @Tree(%clock: i1, %reset: i1,
      %in_0: i16, %in_1: i16,
      %in_2: i16, %in_3: i16) -> (sum: i16) {


    %r0 = seq.firreg %0 clock %clock {...} : i16
    %r1 = seq.firreg %1 clock %clock {...} : i16
    %r2 = seq.firreg %2 clock %clock {...} : i16
    %0 = comb.add %in_0, %in_1 : i16
    %1 = comb.add %in_2, %in_3 : i16
    %2 = comb.add %r0, %r1 : i16
    hw.output %r2 : i16
  }
}
```



tree.fused.mlir

```
module {
  hw.module @Tree(%clock: i1, %reset: i1,
      %in_0: i16, %in_1: i16,
      %in_2: i16, %in_3: i16) -> (sum: i16) attributes {
        ksim.output_ahead = 2 : i64
    } {
    %r0 = seq.firreg %0 clock %clock {ksim.delay=0:i64} : i16
    %r1 = seq.firreg %1 clock %clock {ksim.delay=0:i64} : i16
    %r2 = seq.firreg %2 clock %clock {ksim.delay=0:i64} : i16
    %0 = comb.add %in_0, %in_1 : i16
    %1 = comb.add %in_2, %in_3 : i16
    %2 = comb.add %r0, %r1 : i16
    hw.output %r2 : i16
  }
}
```

Phase 3: Lowering

Lower the fused IR to low level queue IR

```
ksim tree.fused.mlir \
    -in=fused -out=low \
    -o tree.low.mlir
```

IO, registers are converted to queue

Register R/W ops are converted to `get_queue`, `push_queue` ops

tree.low.mlir

```
module {
  ksim.low.def_queue @clock depth 1 : i1 delay [0]
  ksim.low.def_queue @reset depth 1 : i1 delay [0]
  ksim.low.def_queue @in_0 depth 1 : i16 delay [0]
  ksim.low.def_queue @in_1 depth 1 : i16 delay [0]
  ksim.low.def_queue @in_2 depth 1 : i16 delay [0]
  ksim.low.def_queue @in_3 depth 1 : i16 delay [0]
  ksim.low.def_queue @sum depth 1 : i16 delay [0]
  func.func @Tree() attributes {ksim.output_ahead = 2 : i64} {
    %0 = ksim.low.get_queue @in_0[0] : i16
    %1 = ksim.low.get_queue @in_1[0] : i16
    %2 = ksim.low.get_queue @in_2[0] : i16
    %3 = ksim.low.get_queue @in_3[0] : i16
    %4 = comb.add %0, %1, %2, %3 : i16
    ksim.low.push_queue @sum %4 : i16
    return
  }
}
```

Outline

- Compilation workflow of Khronos
  - Input Preparation
  - Header and Code Generation
  - Testbench Preparation
  - Compile and Linking
- Intermediate Representation of Khronos
  - Flattening, Fusing, Lowering, Queuing
- Real Application Evaluation

Real Application Evaluation

- We run khronos on real world design, Gemmini
  - A full run is time consuming, we only run khronos on its core PE array

Download the design and testbench file from our website

```
wget https://ericlyun.me/tutorial-asplos2025/rsrc/khronos/Mesh6x6.fir \
    -O Mesh6x6.fir
wget https://ericlyun.me/tutorial-asplos2025/rsrc/khronos/Mesh6x6.tb.cpp \
    -O Mesh6x6.tb.cpp
```

Compile it using khronos

```
firtool-ksim Mesh6x6.fir --ir-hw -o Mesh6x6.mlir
ksim Mesh6x6.mlir \
    --out-header=Mesh6x6.h \
    --out-driver=Mesh6x6.cpp \
    -o Mesh6x6.ll
llc-ksim -O2 \
    --relocation-model=dynamic-no-pic \
    --filetype=obj \
    Mesh6x6.ll -o Mesh6x6.ksim.o
clang++ -O2 Mesh6x6.cpp Mesh6x6.ksim.o -o Mesh6x6
```

Compile it using verilator

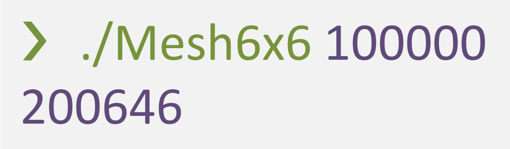
```
firtool-ksim Mesh6x6.fir -o Mesh6x6.v
verilator --cc --exe --build \
    Mesh6x6.v Mesh6x6.tb.cpp
```

The compilation is a little slow

Real Application Evaluation

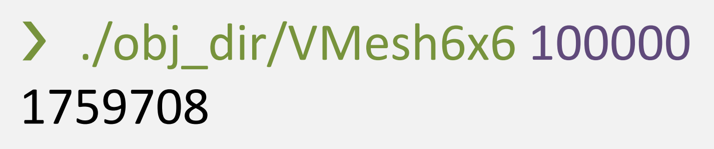
- The testbench will print elapsed time, run each tb and compare the result
- Result varies on your platform

Khronos



```
❯  ./Mesh6x6 100000
200646
```

Verilator



```
❯  ./obj_dir/VMesh6x6 100000
1759708
```

8.77x faster

Peking University

HDL and Framework for Chip Design

Presenter: Youwei Xiao

Author: Youwei Xiao, Zizhang Luo

Cement: Evolved HDL and Framework for Chip Design

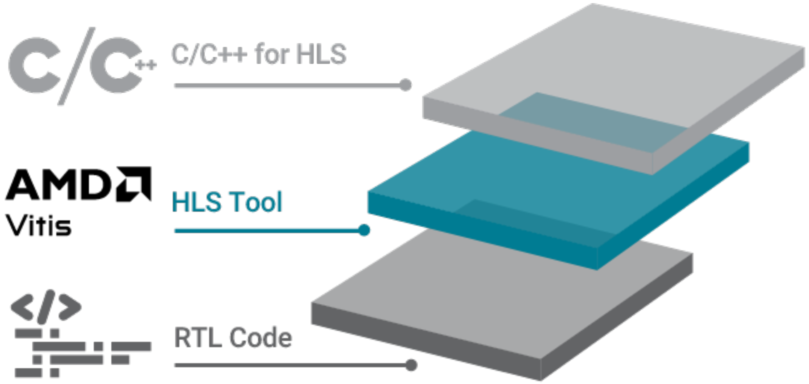
SystemVerilog

Traditional HDL
tedious hardware code

?

CHISEL

Embedded HDL (eHDL)
design hardware in advanced programming language (Scala, Python)



HLS: design hardware in C/C++

Rust-embedded HDL

software-like control description

Cement
(cmt2!)

keep evolving…

Cement (cmt1)

published at FPGA’24

Youwei Xiao, Zizhang Luo, Kexing Zhou, and Yun Liang. 2024. Cement: Streamlining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis. FPGA '24. https://doi.org/10.1145/3626202.3637561

➕ *rule-based design*

bluespec™

High-level HDL: describe hardware as rules

Today, we introduce cmt2!

as the successor of …

Cement (cmt1)

published at FPGA’24

cmt2!

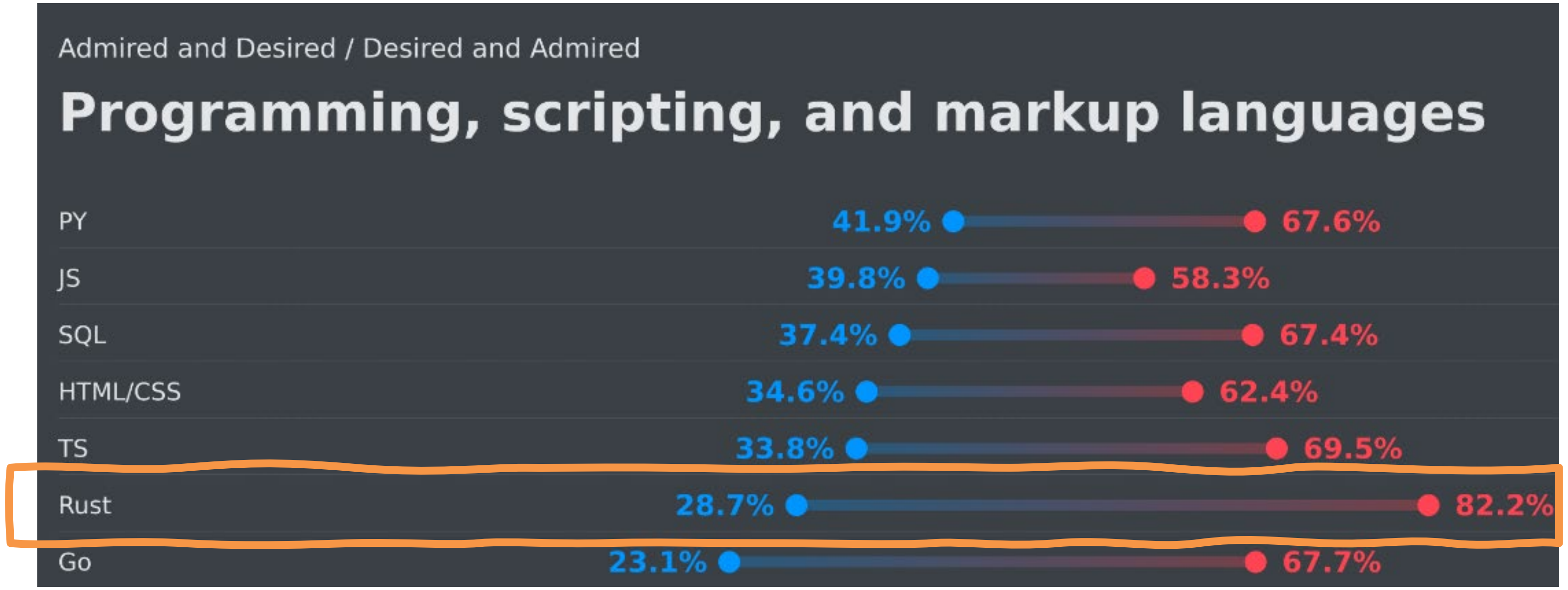
Rust-embedded HDL

rule-based design

software-like control description

We choose Rust, because…

Rust has been the top admired programming language for **7 consecutive years**



Admired and Desired / Desired and Admired

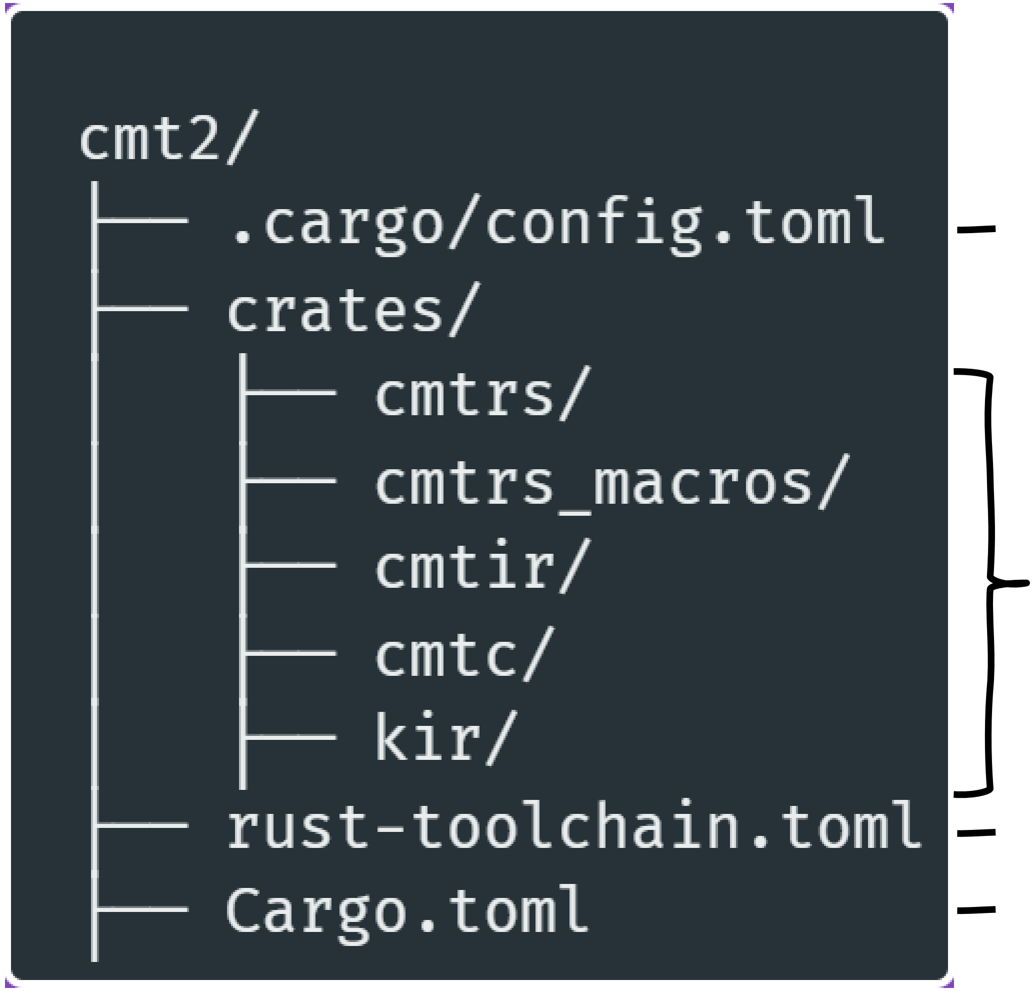
Programming, scripting, and markup languages

Rust is admired by 82.2%!

In our own experience, it’s powerful, efficient, user-friendly

In Rust, cmt2 project looks like…

```
⌨ cd $REPOS/cmt2 && ls -a
```



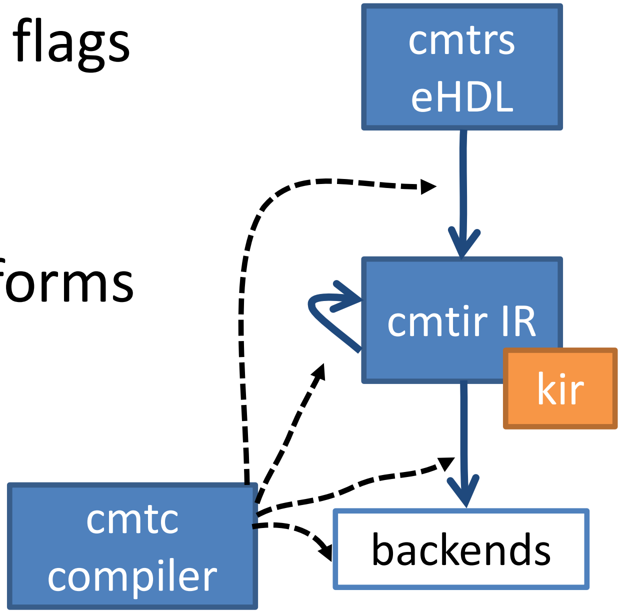
```
cmt2/
├── .cargo/config.toml
├── crates/
│   ├── cmtrs/
│   ├── cmtrs_macros/
│   ├── cmtir/
│   ├── cmtc/
│   └── kir/
├── rust-toolchain.toml
└── Cargo.toml
```

include necessary build flags

Here, 5 library *crates* forms the **cmt2** *package*

specify rust version

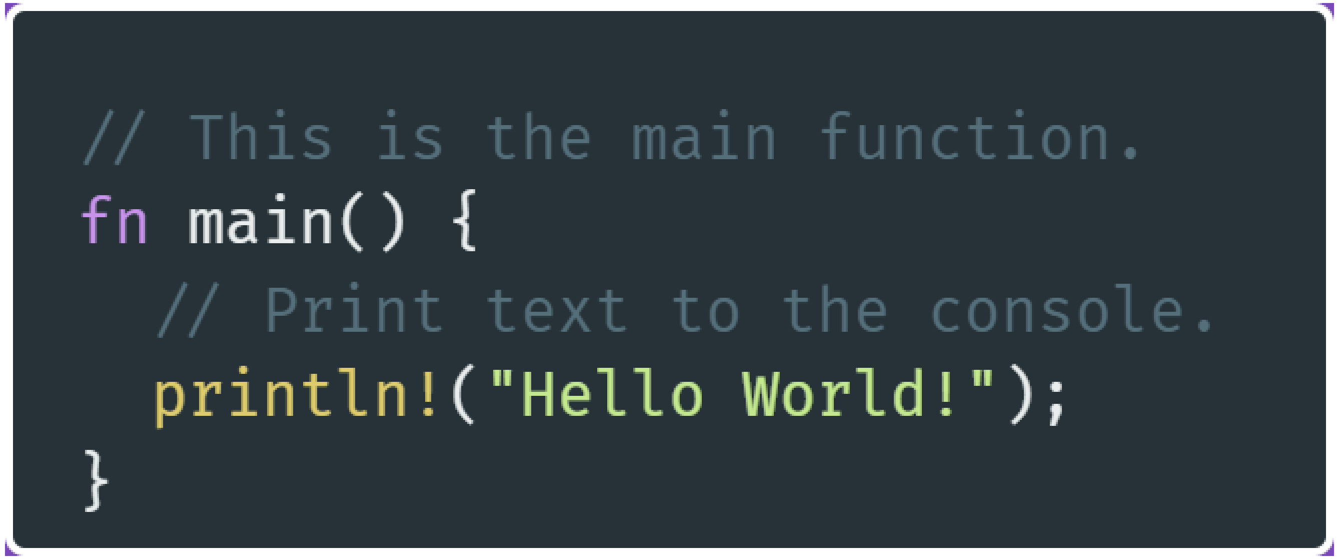
specify package information and dependencies



Hello Rust World!

⌨
```
touch
$REPOS/cmt2/crates/cmtc/examples/hello.rs
```

create `hello.rs` file ↑



```
// This is the main function.
fn main() {
  // Print text to the console.
  println!("Hello World!");
}
```

click [run] above `fn main`, or ...

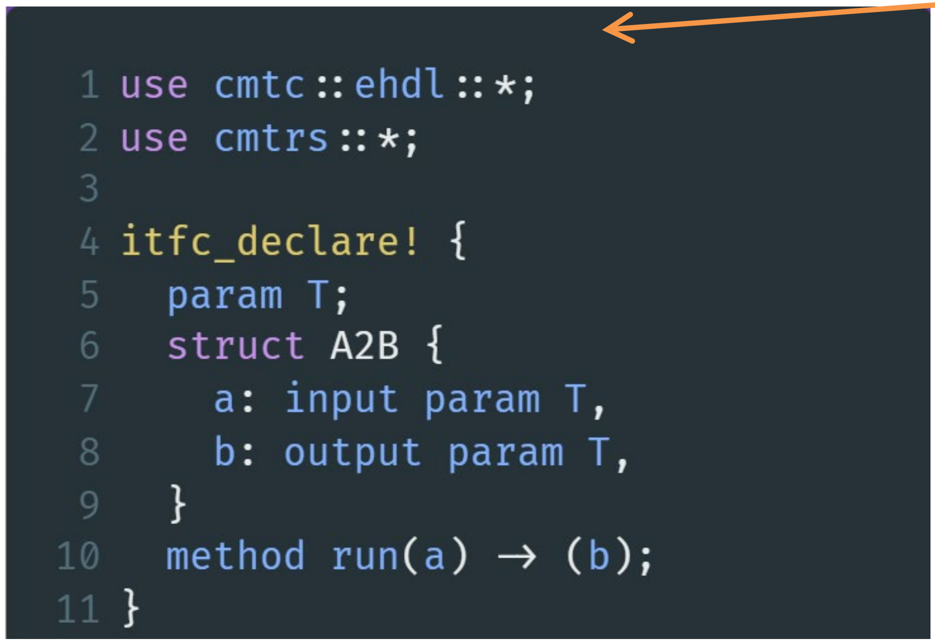
under directory `$REPOS/cmts`

⌨
```
cargo run --example hello
```

Cargo is Rust's build system and package manager. It's easy-to-use!
More user-friendly than sbt for Scala/Chisel!

Hello cmt2!

at the top of `hello.rs`



```rust
use cmtc::ehdl::*;
use cmtrs::*;

itfc_declare! {
  param T;
  struct A2B {
    a: input param T,
    b: output param T,
  }
  method run(a) -> (b);
}
```

use clauses import cmt2's eHDL and compiler features

itfc_declare! is a *macro*

macro in Rust transforms code at the token-level

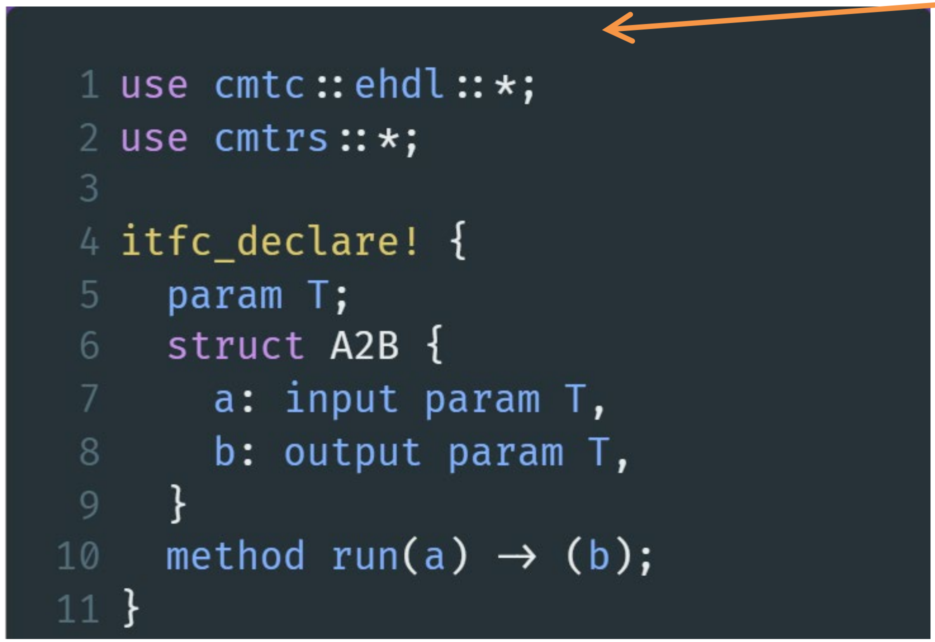
If curious, feel free to run (if have internet connection):

```
⌨ cargo install cargo-expand
⌨ cd $REPOS/cmt2/crates/cmtc
⌨ cargo expand --example hello
```

to see the transformed code!

Hello cmt2!

at the top of `hello.rs`



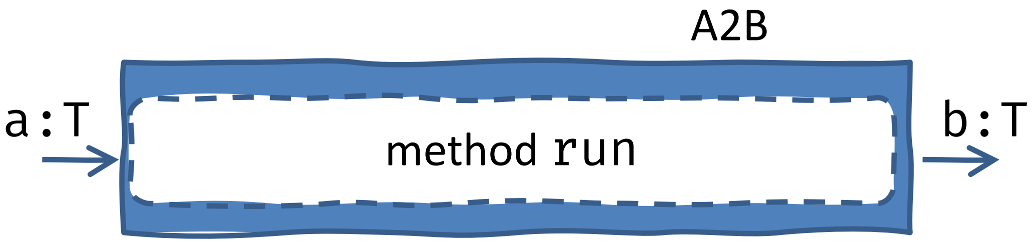
```
use cmtc::ehdl::*;
use cmtrs::*;

itfc_declare! {
  param T;
  struct A2B {
    a: input param T,
    b: output param T,
  }
  method run(a) → (b);
}
```

macro is an advanced feature in Rust!

Please leave cmt2 developers to suffer 😂
and… just use them! 😄

The left code creates …

- A module interface named A2B;
- It has one input port `a`, one output port `b`, both of which have type `T`.
- It exposes a method `run` for external world to invoke!



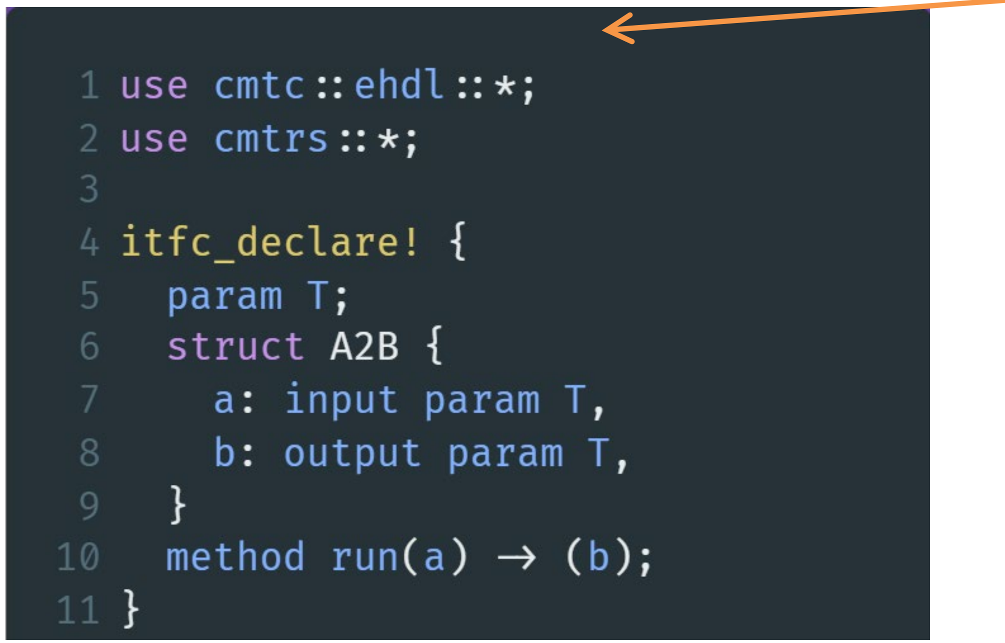
A2B

a:T

method run

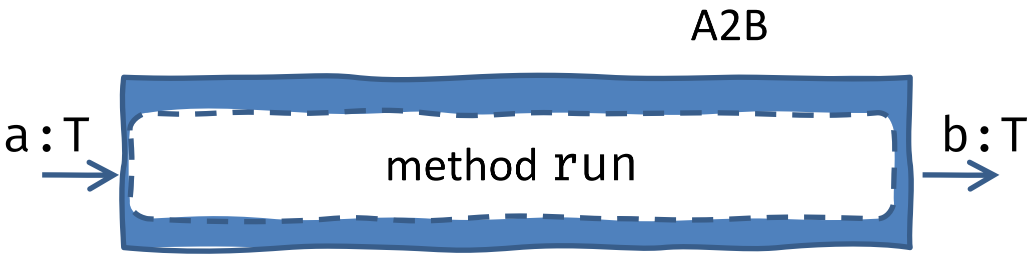
b:T

cmt2 is rule-based HDL



```
use cmtc::ehdl::*;
use cmtrs::*;

itfc_declare! {
  param T;
  struct A2B {
    a: input param T,
    b: output param T,
  }
  method run(a) → (b);
}
```

at the top of `hello.rs`



cmt2! *rule-based design*

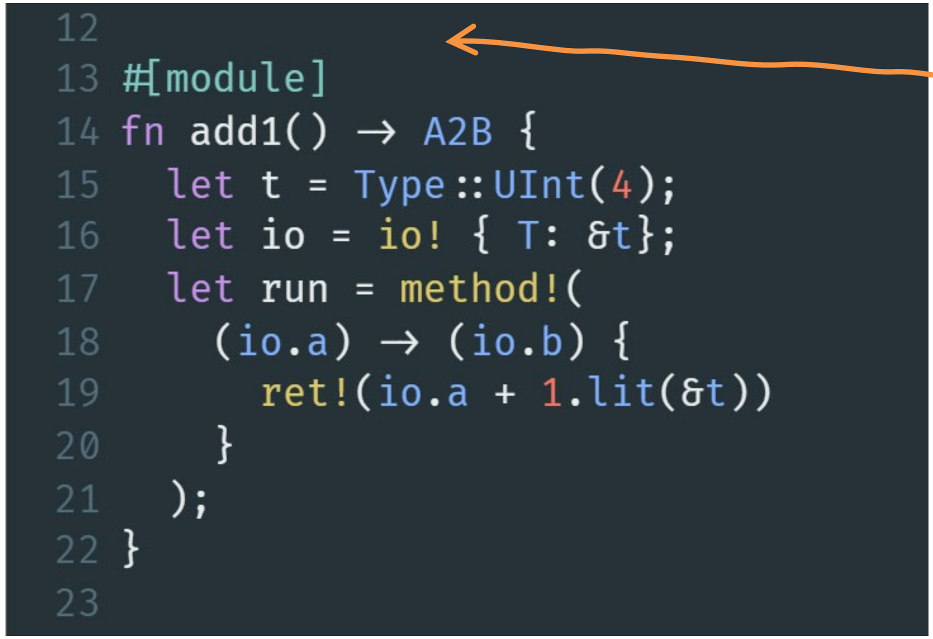
Hardware logic is organized as **rules** – the execution unit!

Don't be afriad! 😄
Just consider one rule as a group of operations that execute together

In cmt2, we have two types of rules: *method* rules and *always* rules:

- *method* rule need to be invoked (Bluespec method)
- *always* rule runs actively (Bluespec rule)

cmt2 is rule-based HDL

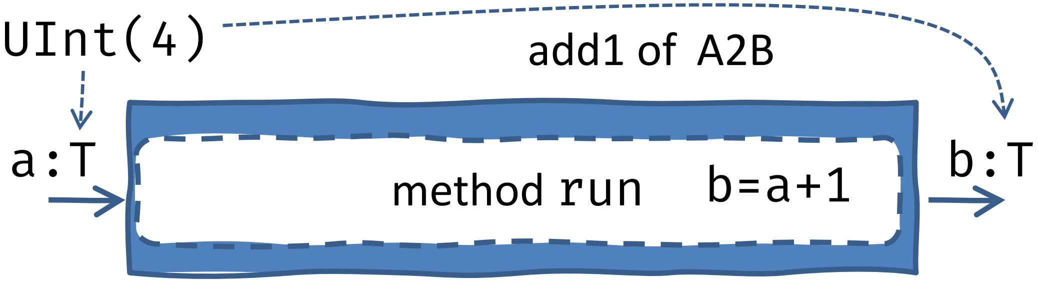


```
#[module]
fn add1() -> A2B {
  let t = Type::UInt(4);
  let io = io! { T: &t};
  let run = method!(
    (io.a) -> (io.b) {
      ret!(io.a + 1.lit(&t))
    }
  );
}
```

following `itfc_declare!`

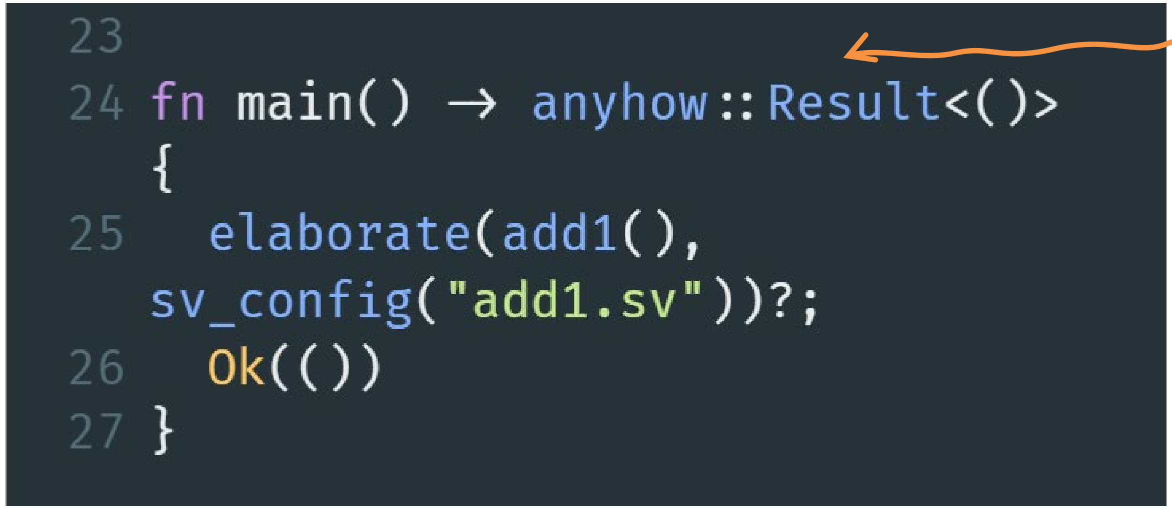
fn starts a function!

The `#[module]` *macro* transfoms the function `add1` to create a module of the interface `A2B`



Now, types are given, rules are filled -- Module is completely described!

cmt2 generates SystemVerilog



```rust

fn main() -> anyhow::Result<()>
{
  elaborate(add1(),
sv_config("add1.sv"))?;
  Ok(())
}
```

following `fn add1`

The new `main` function to call `elaborate` for the `add1` module

click [run] above `fn main`, or ...

```
⌨ cargo run --example hello
⌨ cat add1.sv
```

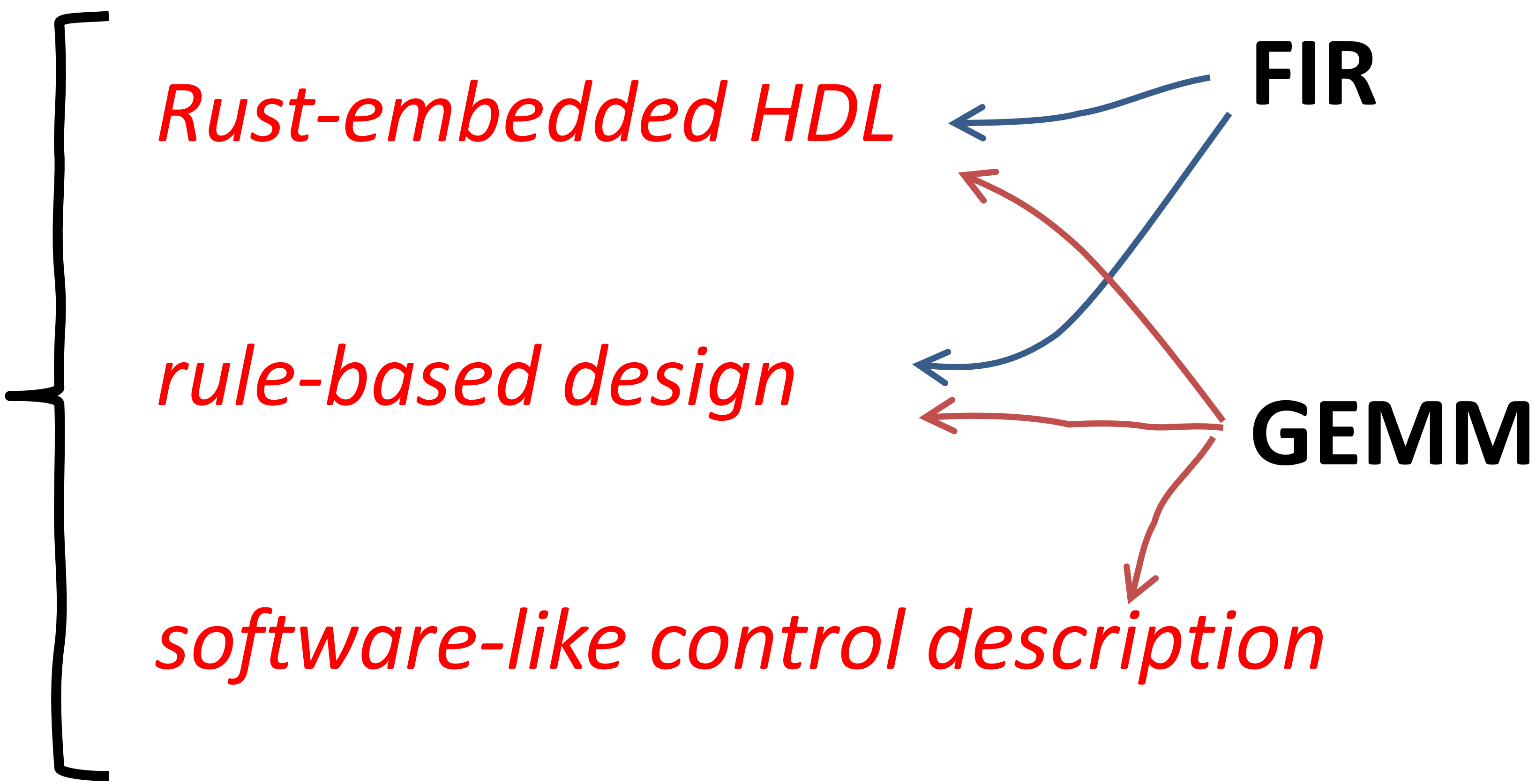
Show features by examples

as the successor of …

Cement (cmt1)

published at FPGA’24

cmt2!



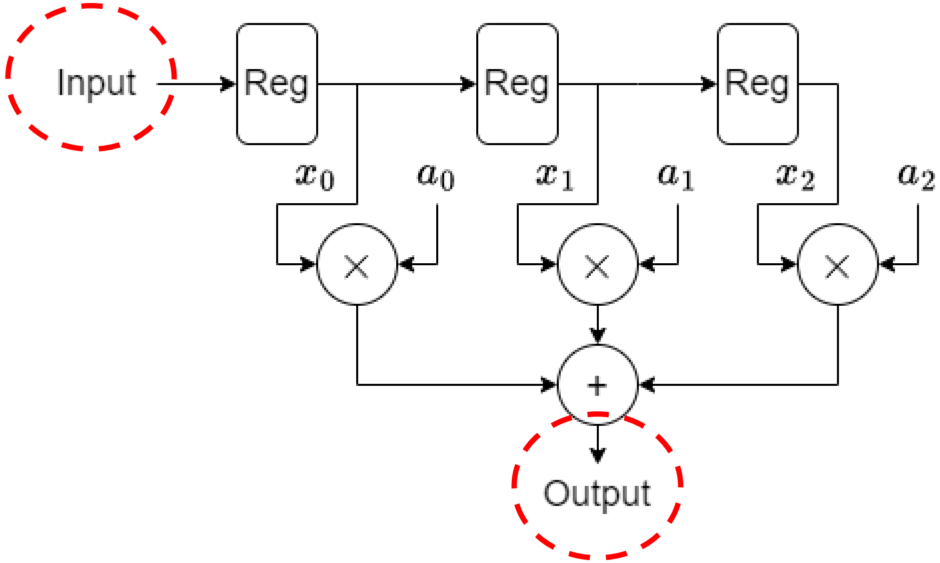
Rust-embedded HDL

rule-based design

software-like control description

Finite Impulse Response

Rust-embedded HDL

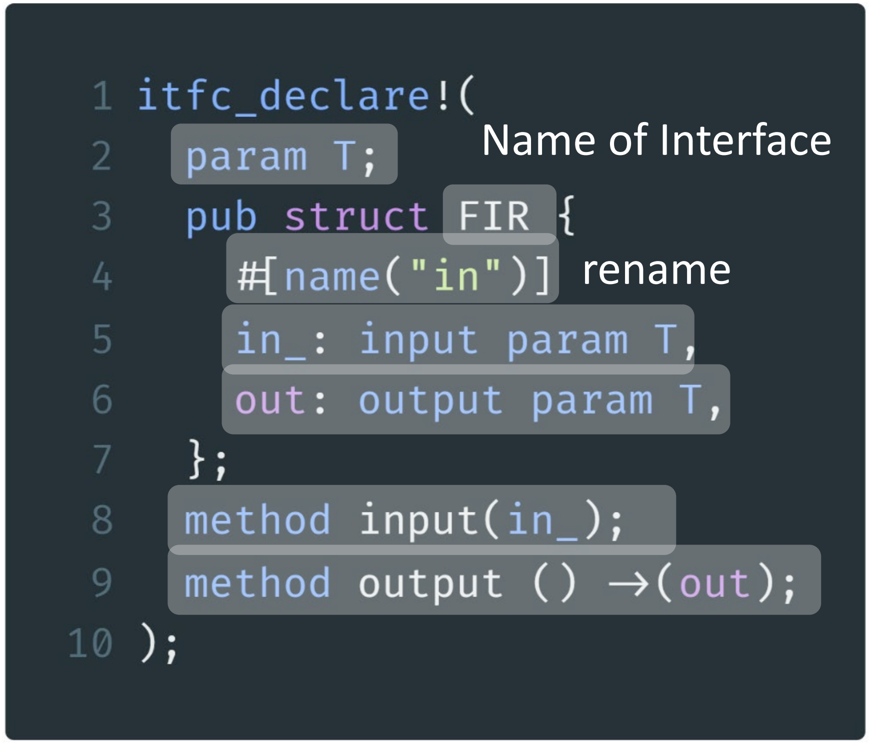


Type Parameter

An input port named “in_” of type T

A method to input data

A method to output data



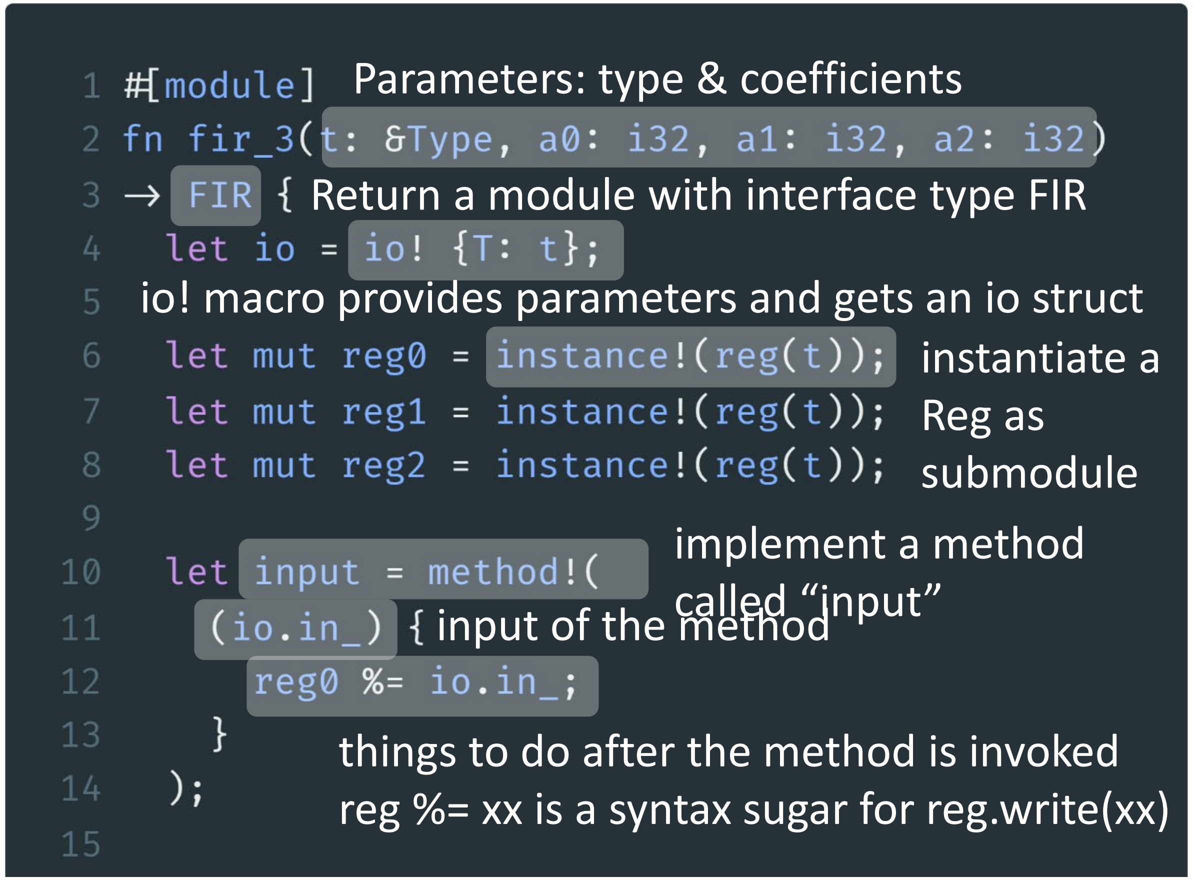
```
itfc_declare!(
  param T;
  pub struct FIR {
    #[name("in")]
    in_: input param T,
    out: output param T,
  };
  method input(in_);
  method output () ->(out);
);
```

Name of Interface

rename

./crates/cmtc/examples/fir.rs   line 7-16

FIR Example1: fixed length



```
#[module]
fn fir_3(t: &Type, a0: i32, a1: i32, a2: i32)
→ FIR {
  let io = io! {T: t};

  let mut reg0 = instance!(reg(t));
  let mut reg1 = instance!(reg(t));
  let mut reg2 = instance!(reg(t));

  let input = method!(
    (io.in_) {
      reg0 %= io.in_;
    }
  );

```

Parameters: type & coefficients

Return a module with interface type FIR

io! macro provides parameters and gets an io struct

instantiate a Reg as submodule

implement a method called "input"

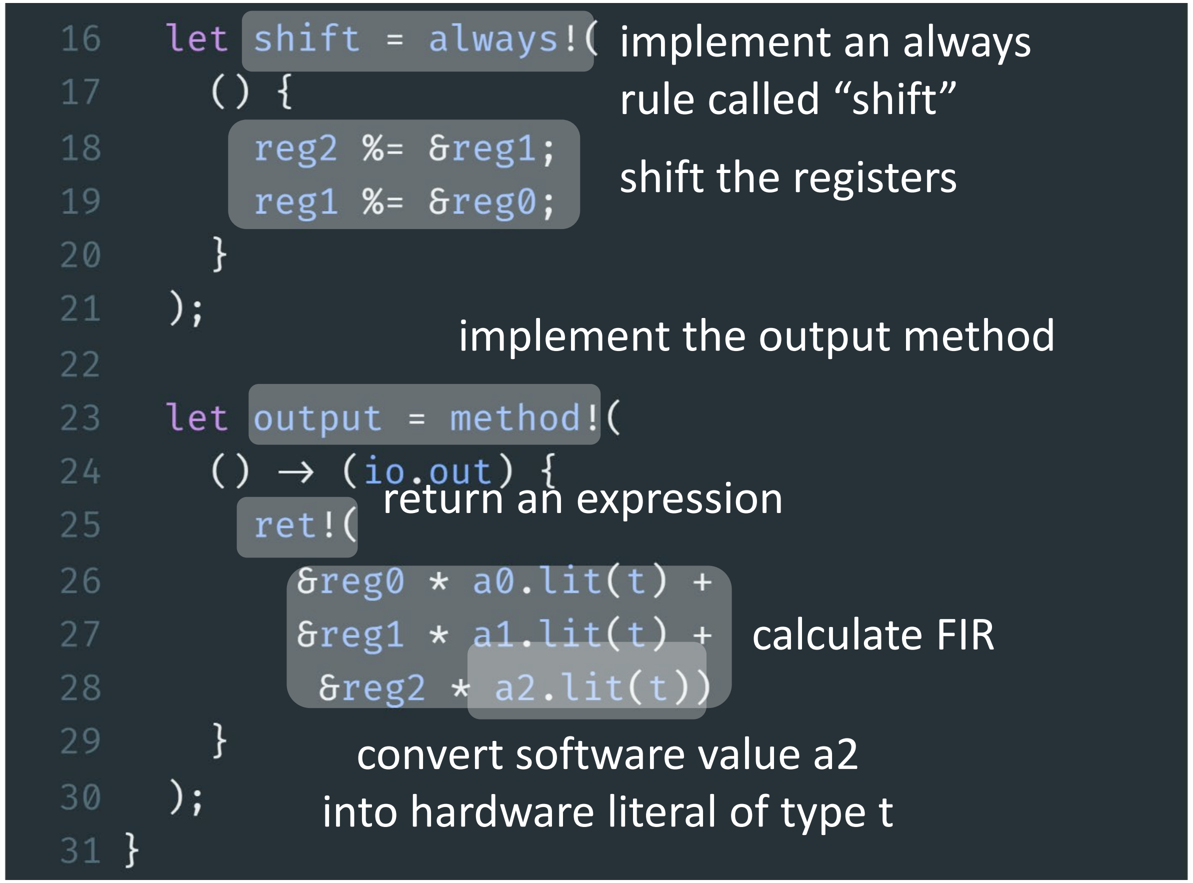
input of the method

things to do after the method is invoked
reg %= xx is a syntax sugar for reg.write(xx)

./crates/cmtc/examples/fir.rs line 18-46

rule-based design

FIR Example1: fixed length



```
  let shift = always!(
    () {
      reg2 %= &reg1;
      reg1 %= &reg0;
    }
  );

  let output = method!(
    () → (io.out) {
      ret!(
        &reg0 * a0.lit(t) +
        &reg1 * a1.lit(t) +
         &reg2 * a2.lit(t))
    }
  );
}
```

implement an always rule called “shift”

shift the registers

implement the output method

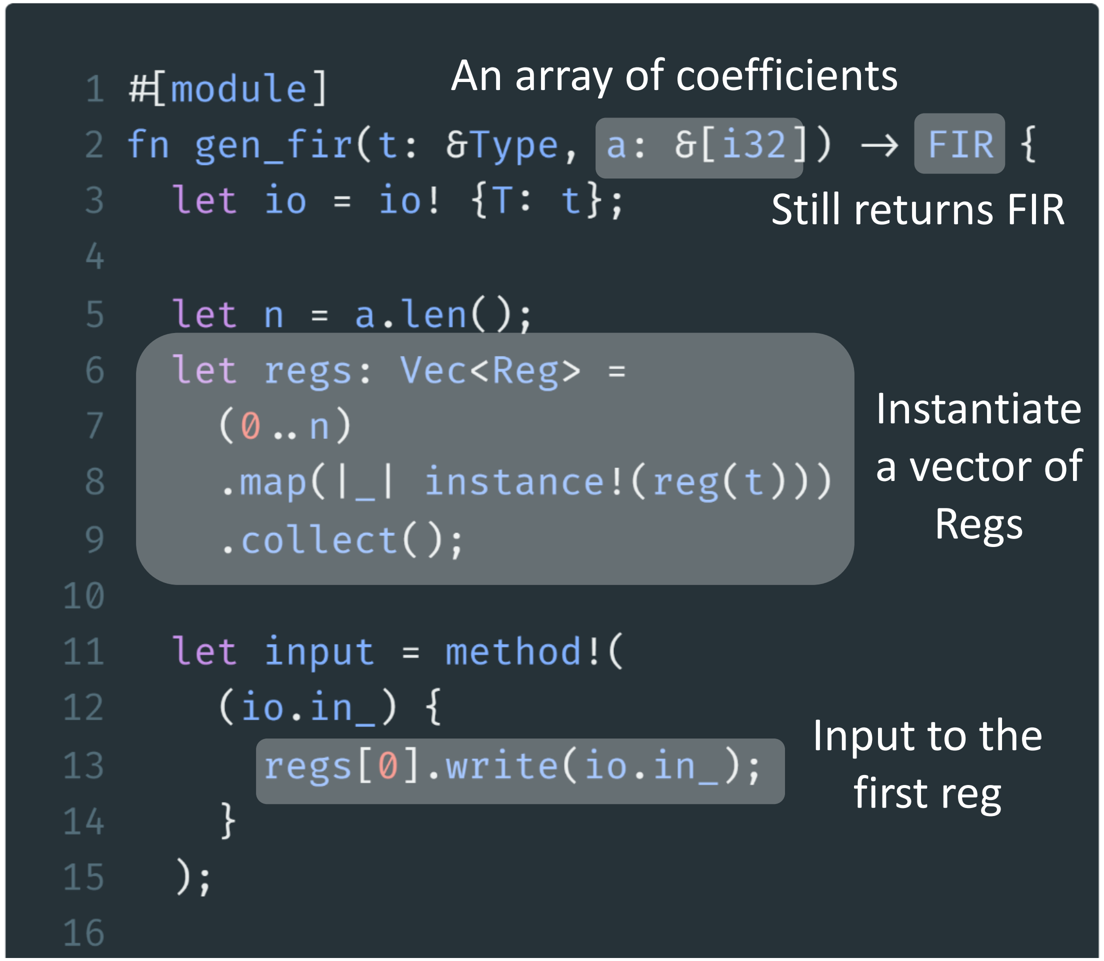
return an expression

calculate FIR

convert software value a2 into hardware literal of type t

./crates/cmtc/examples/fir.rs   line 18-46

FIR Example2: arbitrary length



```
#[module]
fn gen_fir(t: &Type, a: &[i32]) -> FIR {
  let io = io! {T: t};

  let n = a.len();
  let regs: Vec<Reg> =
    (0..n)
    .map(|_| instance!(reg(t)))
    .collect();

  let input = method!(
    (io.in_) {
      regs[0].write(io.in_);
    }
  );

```

An array of coefficients

Still returns FIR

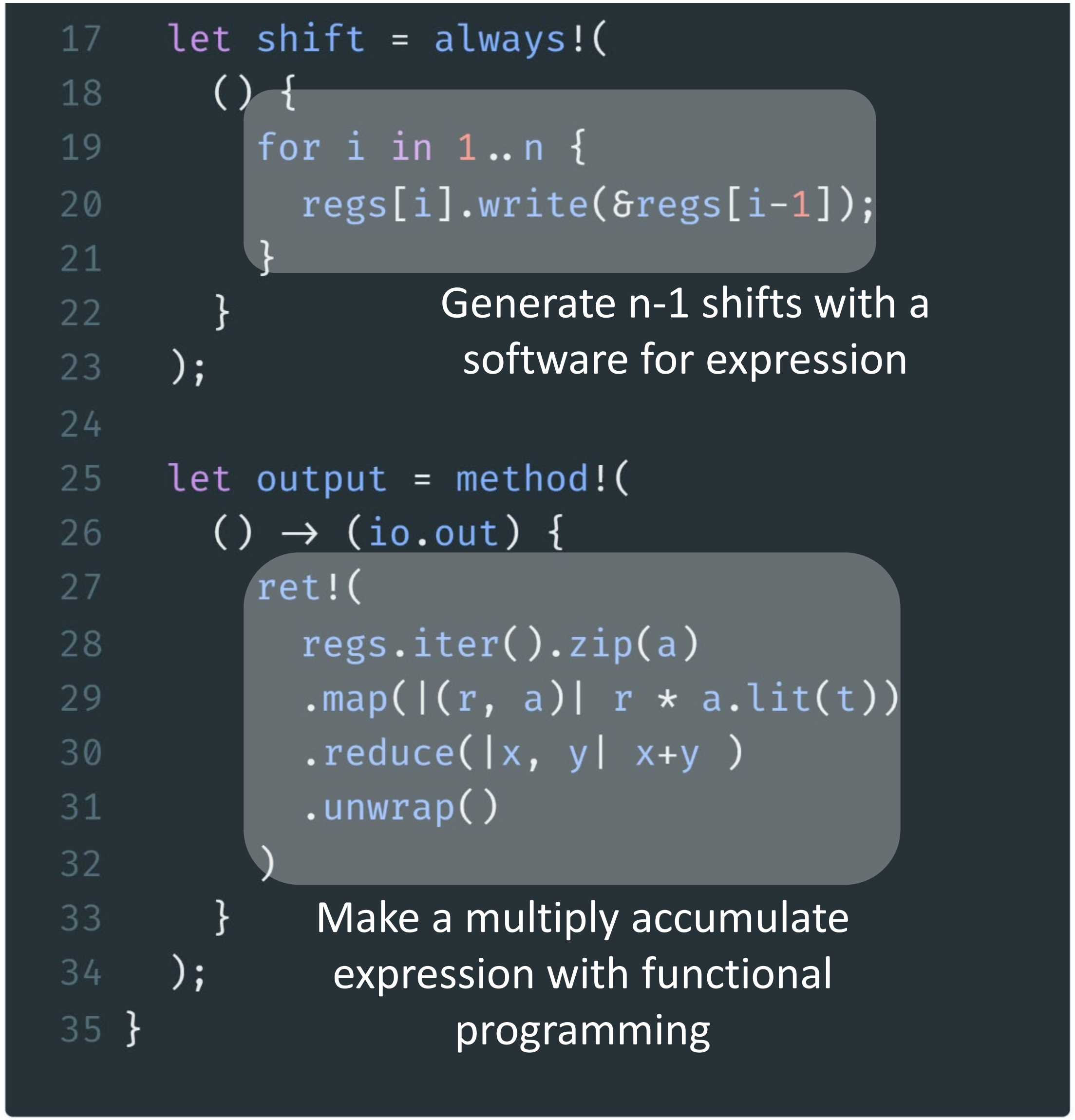
Instantiate a vector of Regs

Input to the first reg

Rust-embedded HDL

./crates/cmtc/examples/fir.rs
line 48-78

FIR Example2: arbitrary length



```
  let shift = always!(
    () {
      for i in 1..n {
        regs[i].write(&regs[i-1]);
      }
    }
  );

  let output = method!(
    () → (io.out) {
      ret!(
        regs.iter().zip(a)
        .map(|(r, a)| r * a.lit(t))
        .reduce(|x, y| x+y )
        .unwrap()
      )
    }
  );
}
```

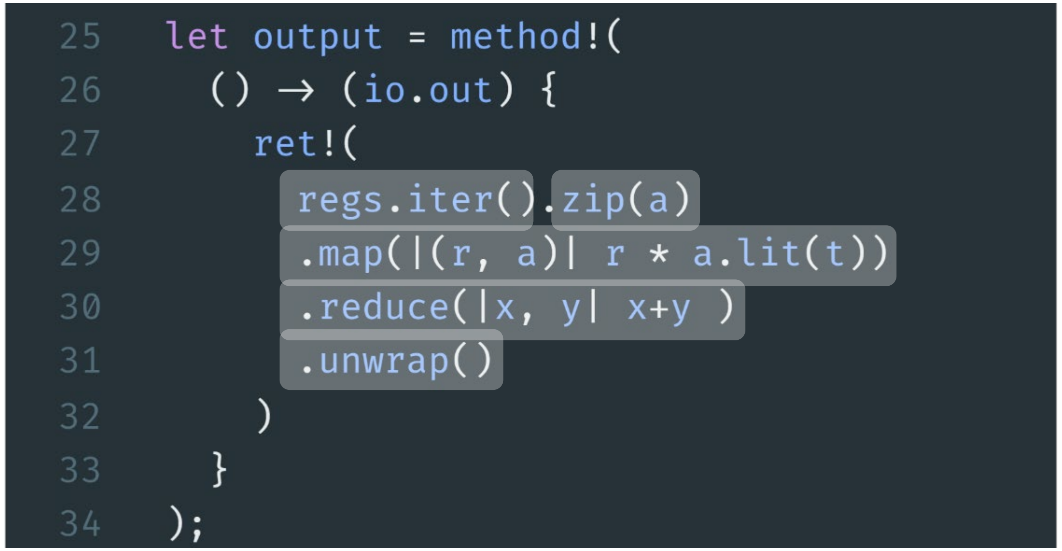
Generate n-1 shifts with a software for expression

Make a multiply accumulate expression with functional programming

./crates/cmtc/examples/fir.rs
line 48-78

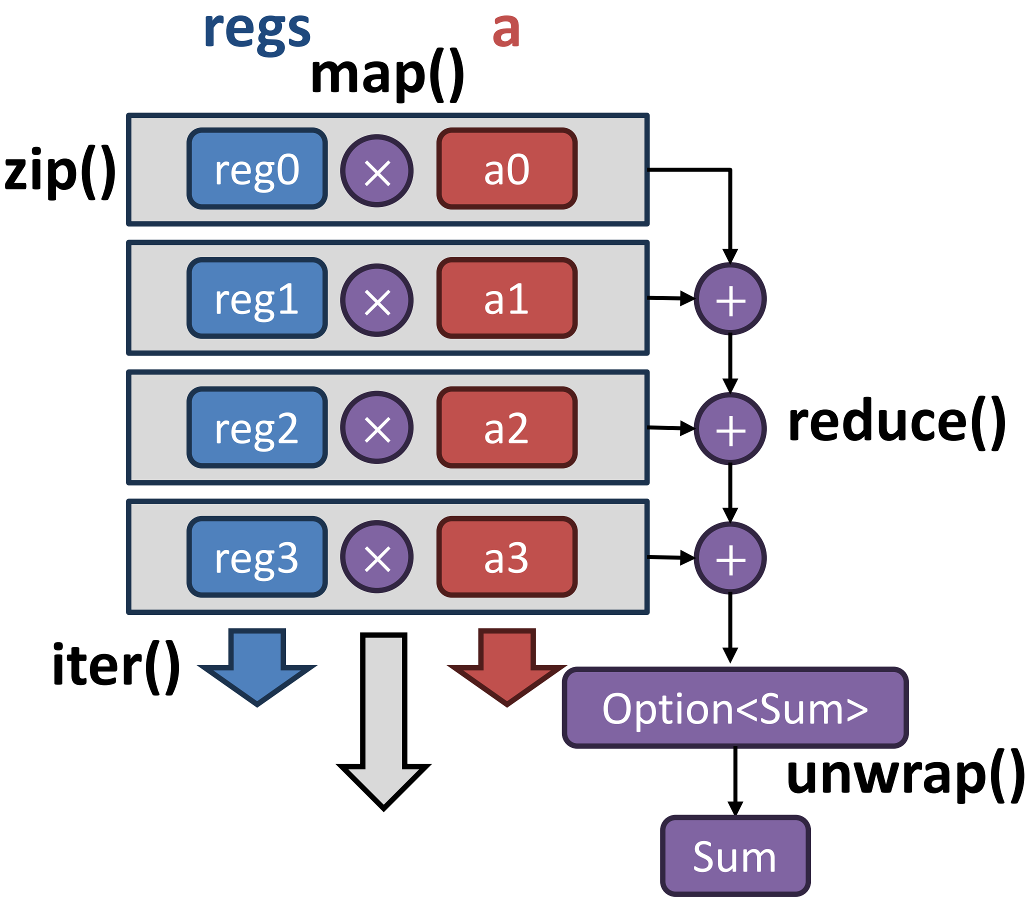
Explanation on the last expression

For audiences that are not familiar with functional programming

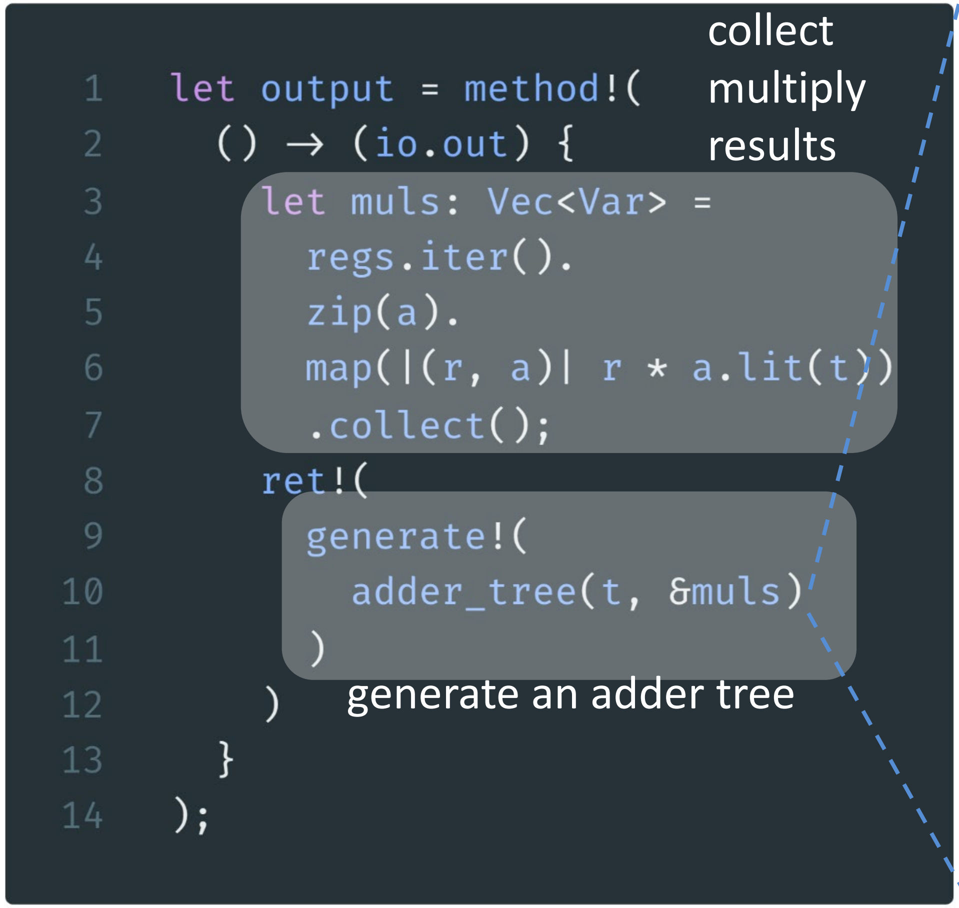


```
let output = method!(
  () → (io.out) {
    ret!(
      regs.iter().zip(a)
      .map(|(r, a)| r * a.lit(t))
      .reduce(|x, y| x+y )
      .unwrap()
    )
  }
);
```

Rust-embedded HDL



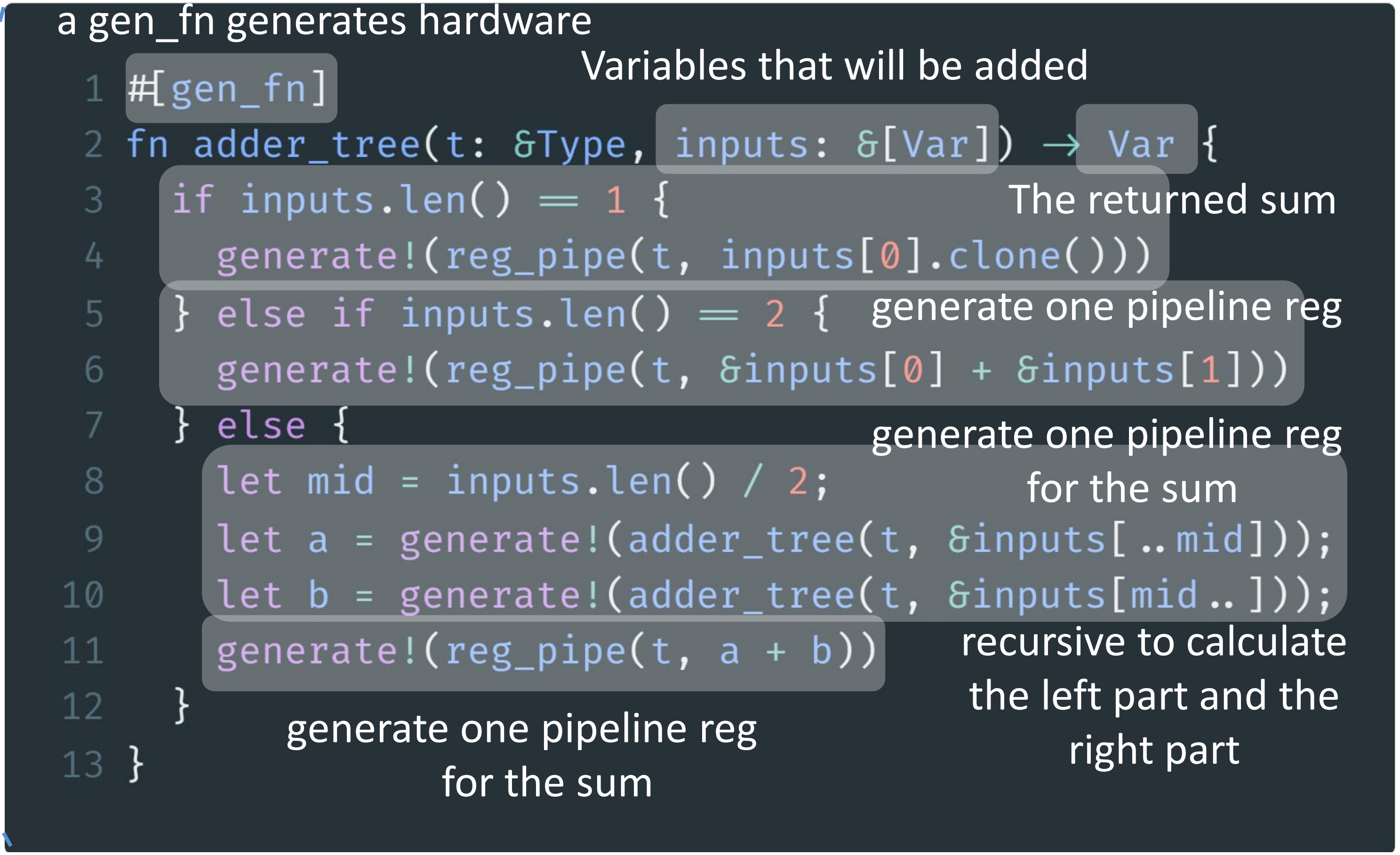
FIR Example3: adder tree



```
let output = method!(
  () -> (io.out) {
    let muls: Vec<Var> =
      regs.iter().
      zip(a).
      map(|(r, a)| r * a.lit(t))
      .collect();
    ret!(
      generate!(
        adder_tree(t, &muls)
      )
    )
  }
);
```

collect multiply results

generate an adder tree



a gen_fn generates hardware

Variables that will be added

```
#[gen_fn]
fn adder_tree(t: &Type, inputs: &[Var]) -> Var {
  if inputs.len() == 1 {
    generate!(reg_pipe(t, inputs[0].clone()))
  } else if inputs.len() == 2 {
    generate!(reg_pipe(t, &inputs[0] + &inputs[1]))
  } else {
    let mid = inputs.len() / 2;
    let a = generate!(adder_tree(t, &inputs[..mid]));
    let b = generate!(adder_tree(t, &inputs[mid..]));
    generate!(reg_pipe(t, a + b))
  }
}
```

The returned sum

generate one pipeline reg

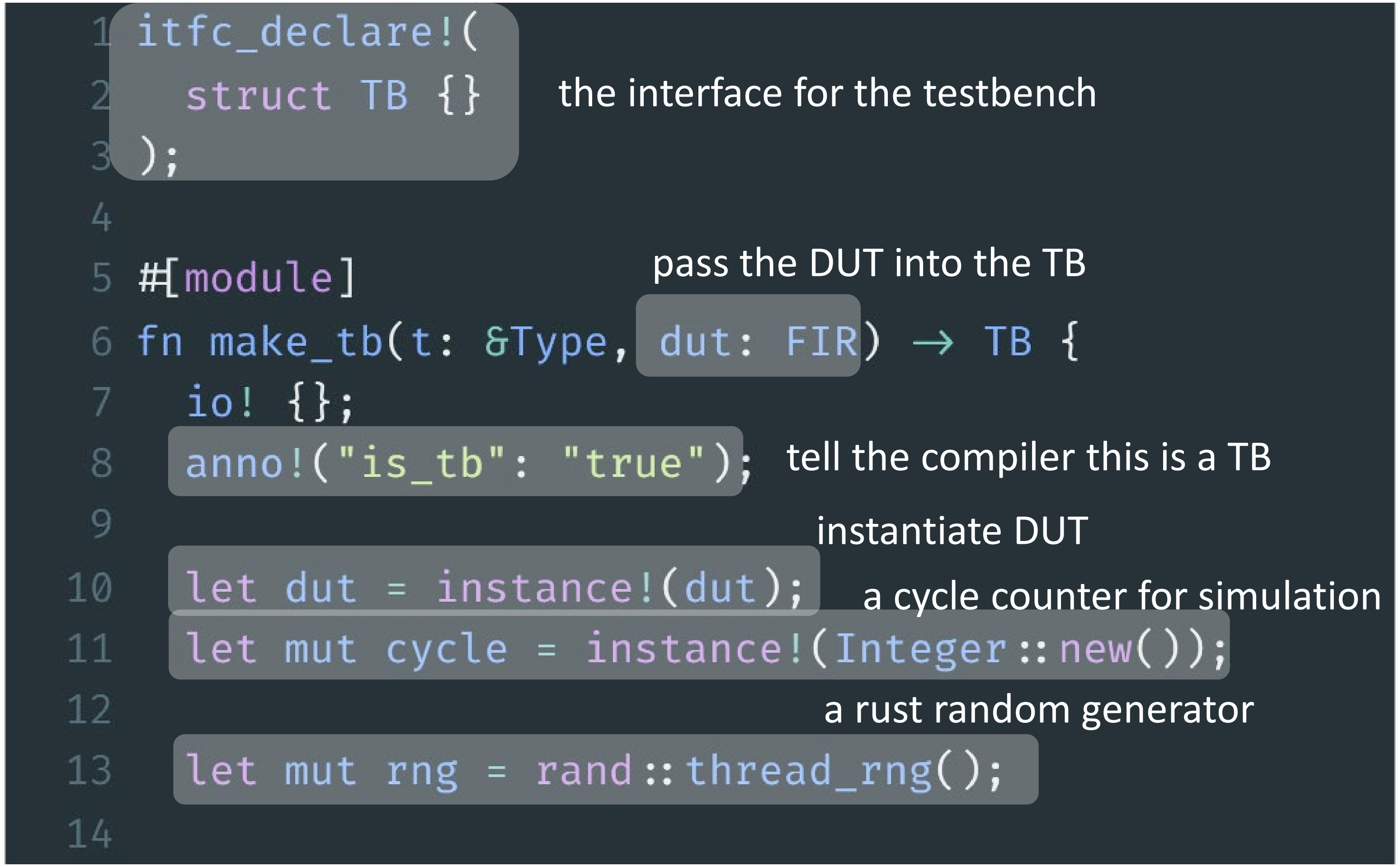
generate one pipeline reg for the sum

recursive to calculate the left part and the right part

generate one pipeline reg for the sum

./crates/cmtc/examples/fir.rs
line 83-123

FIR Example: Testbench preparation



```rust
itfc_declare!(
  struct TB {}
);

#[module]
fn make_tb(t: &Type, dut: FIR) -> TB {
  io! {};
  anno!("is_tb": "true");

  let dut = instance!(dut);
  let mut cycle = instance!(Integer::new());

  let mut rng = rand::thread_rng();

```

the interface for the testbench

pass the DUT into the TB

tell the compiler this is a TB

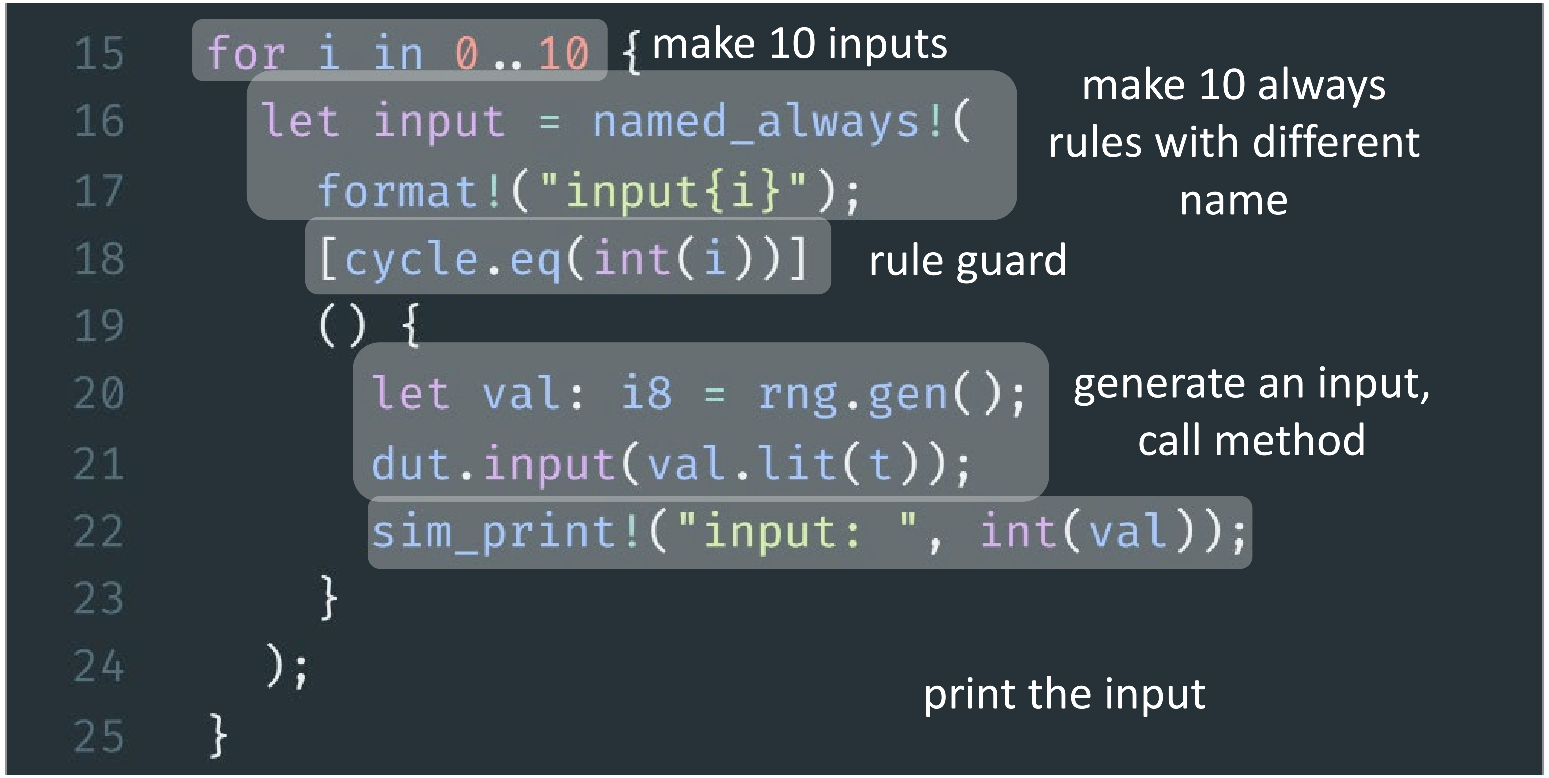
instantiate DUT

a cycle counter for simulation

a rust random generator

./crates/cmtc/examples/fir.rs
line 132-174

FIR Example: Testbench inputs



```
for i in 0..10 {
  let input = named_always!(
    format!("input{i}");
    [cycle.eq(int(i))]
    () {
      let val: i8 = rng.gen();
      dut.input(val.lit(t));
      sim_print!("input: ", int(val));
    }
  );
}
```

make 10 inputs

make 10 always rules with different name

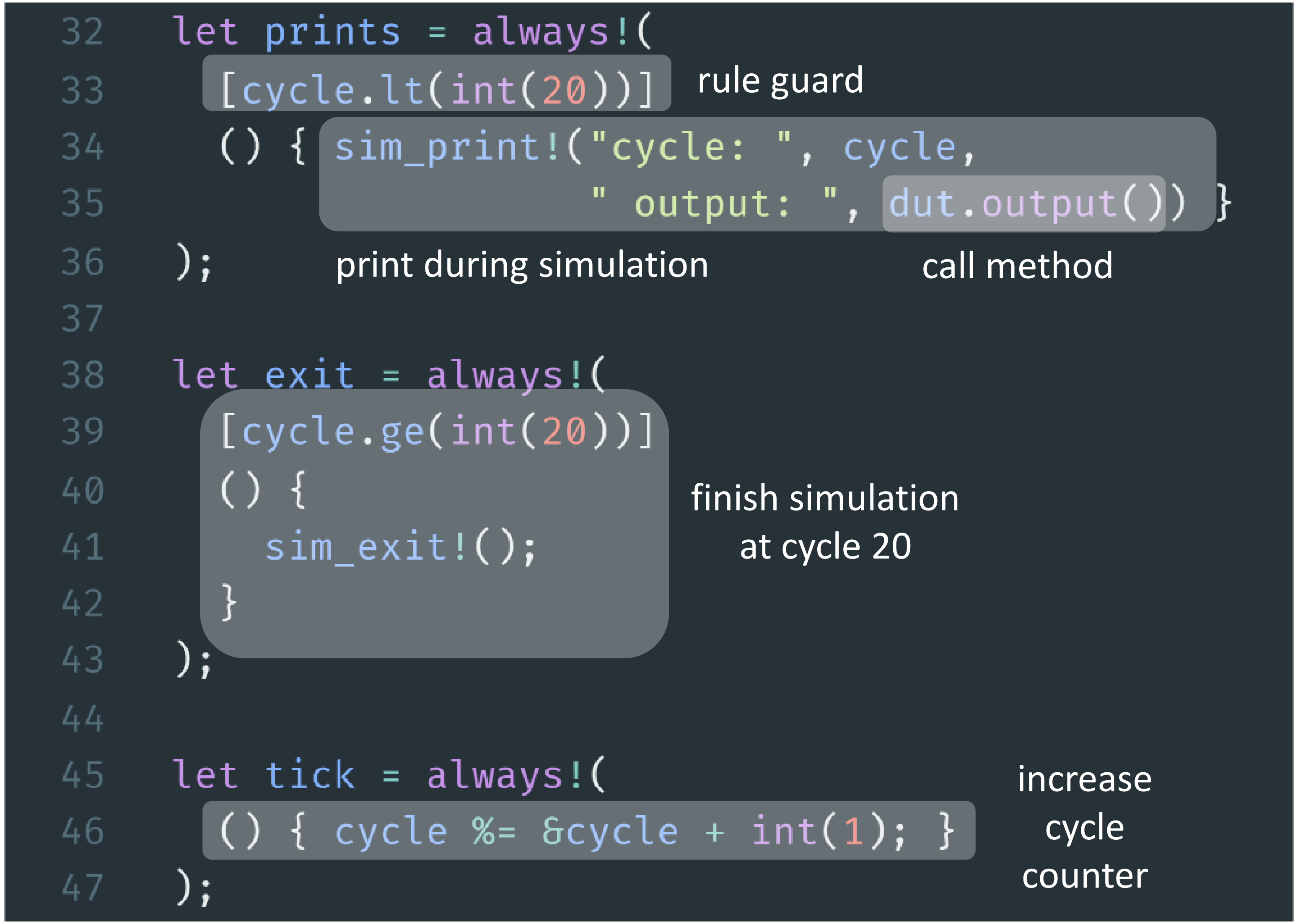
rule guard

generate an input, call method

print the input

./crates/cmtc/examples/fir.rs
line 132-174

FIR Example: Testbench inputs



```
let prints = always!(
  [cycle.lt(int(20))]
  () { sim_print!("cycle: ", cycle,
                  " output: ", dut.output()) }
);

let exit = always!(
  [cycle.ge(int(20))]
  () {
    sim_exit!();
  }
);

let tick = always!(
  () { cycle %= &cycle + int(1); }
);
```

rule guard

print during simulation

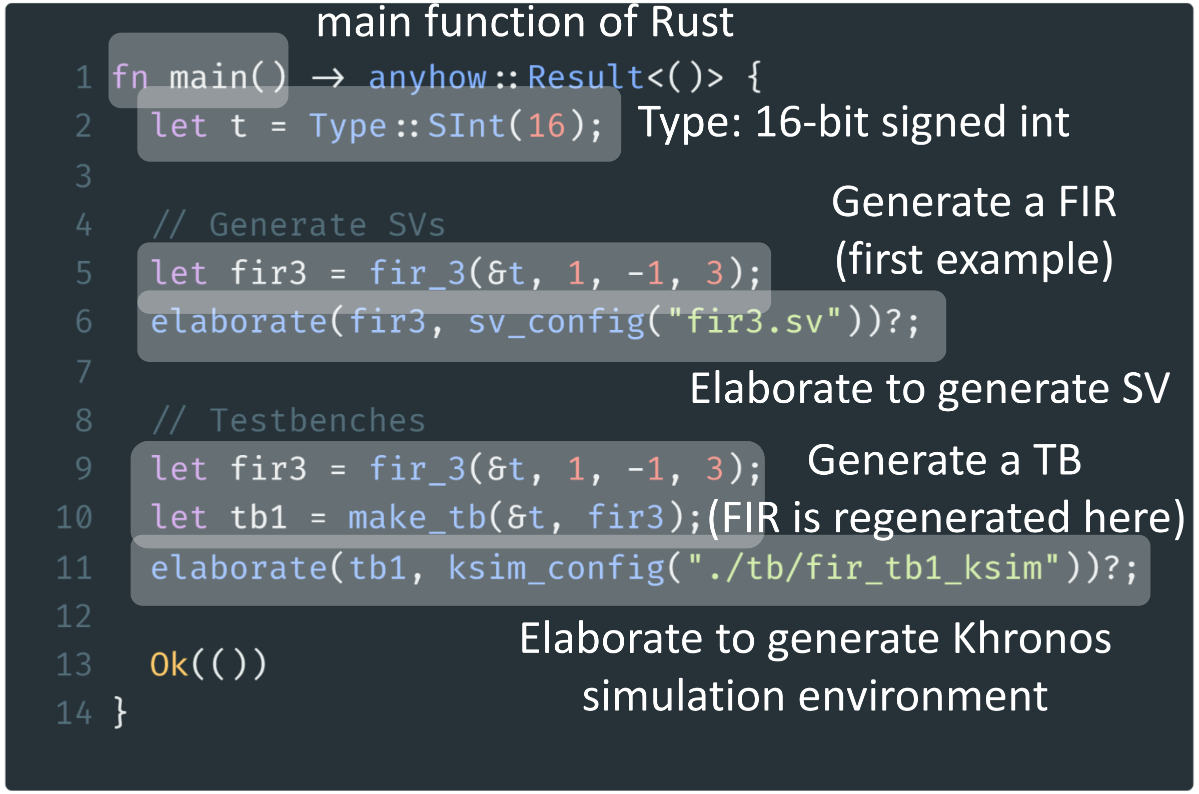
call method

finish simulation at cycle 20

increase cycle counter

./crates/cmtc/examples/fir.rs line 132-174

FIR Example: Elaboration & Simulation



```rust
fn main() -> anyhow::Result<()> {
  let t = Type::SInt(16);

  // Generate SVs
  let fir3 = fir_3(&t, 1, -1, 3);
  elaborate(fir3, sv_config("fir3.sv"))?;

  // Testbenches
  let fir3 = fir_3(&t, 1, -1, 3);
  let tb1 = make_tb(&t, fir3);
  elaborate(tb1, ksim_config("./tb/fir_tb1_ksim"))?;

  Ok(())
}
```

main function of Rust

Type: 16-bit signed int

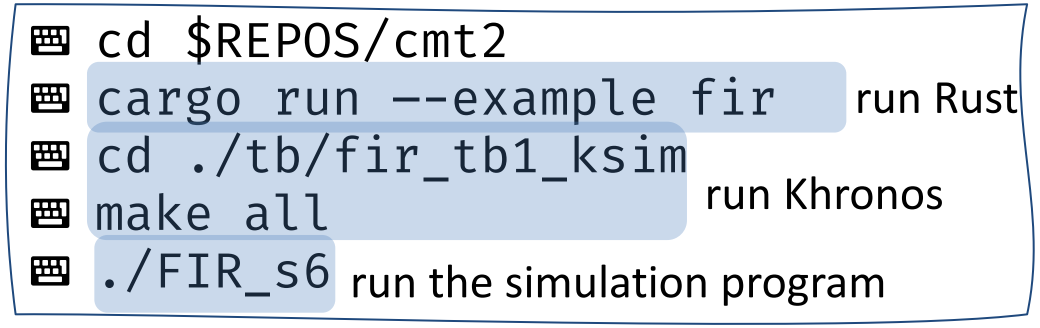
Generate a FIR (first example)

Elaborate to generate SV

Generate a TB (FIR is regenerated here)

Elaborate to generate Khronos simulation environment

./crates/cmtc/examples/fir.rs line 176-204

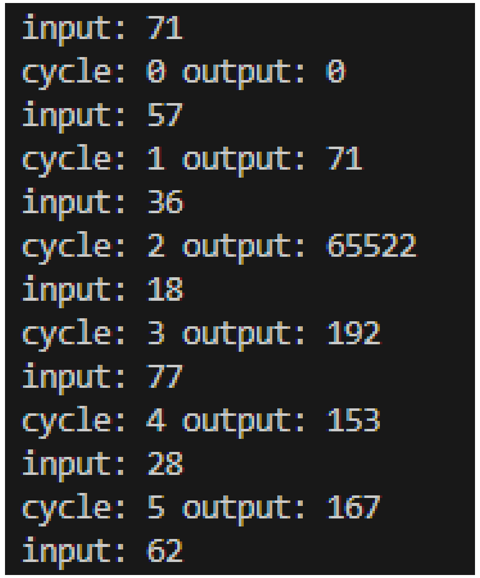


```
cd $REPOS/cmt2
cargo run --example fir
cd ./tb/fir_tb1_ksim
make all
./FIR_s6
```

run Rust

run Khronos

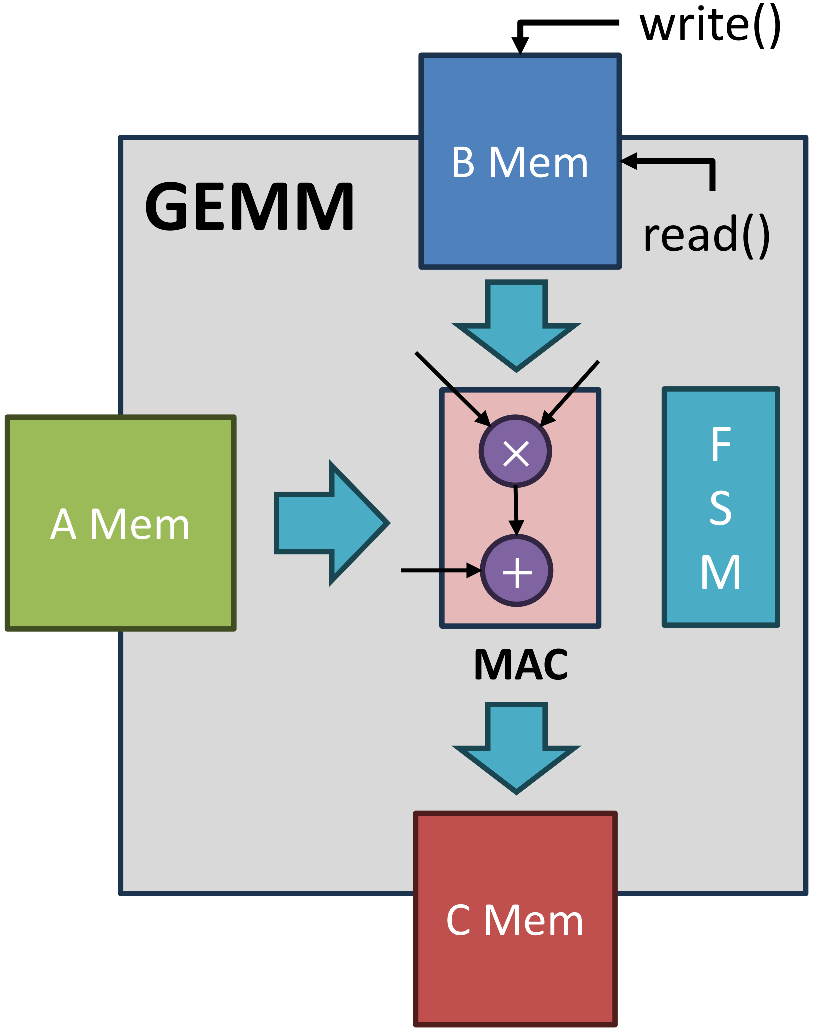
run the simulation program



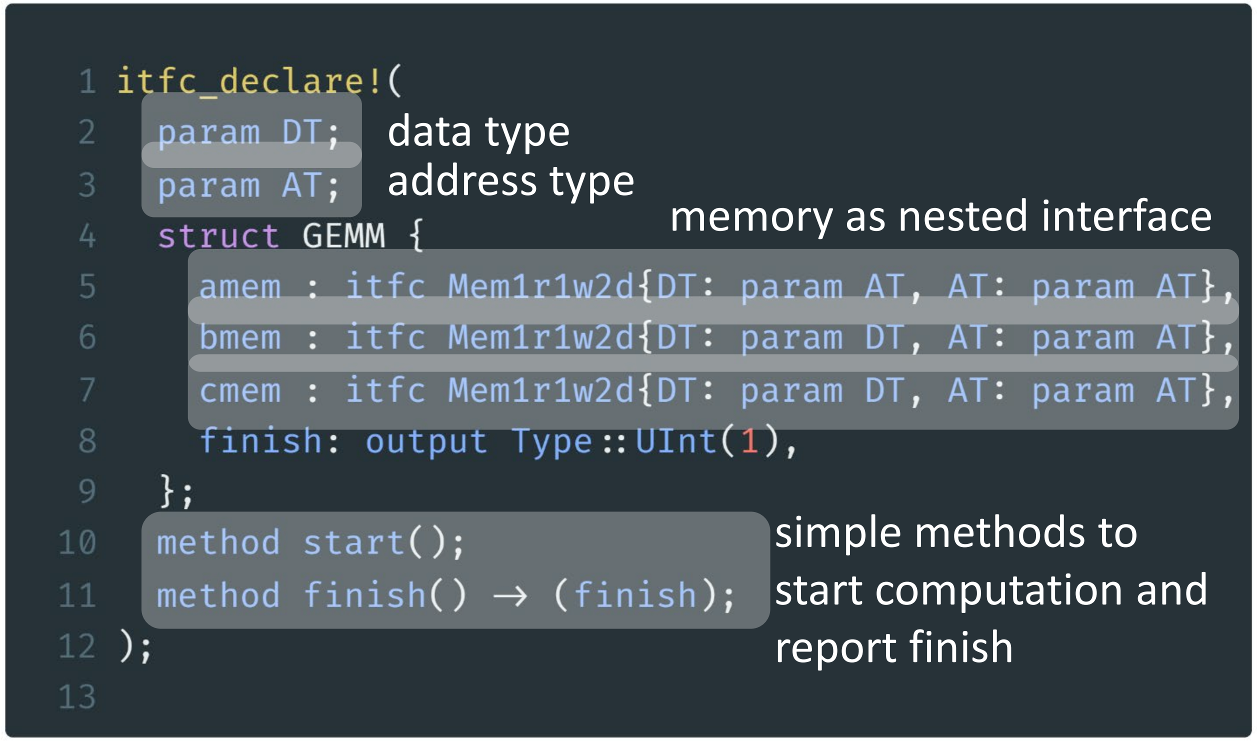
```
input: 71
cycle: 0 output: 0
input: 57
cycle: 1 output: 71
input: 36
cycle: 2 output: 65522
input: 18
cycle: 3 output: 192
input: 77
cycle: 4 output: 153
input: 28
cycle: 5 output: 167
input: 62
```

Possible Outputs

General Matrix Multiplication



software-like control description



```
itfc_declare!(
  param DT;
  param AT;
  struct GEMM {
    amem : itfc Mem1r1w2d{DT: param AT, AT: param AT},
    bmem : itfc Mem1r1w2d{DT: param DT, AT: param AT},
    cmem : itfc Mem1r1w2d{DT: param DT, AT: param AT},
    finish: output Type::UInt(1),
  };
  method start();
  method finish() → (finish);
);
```

data type

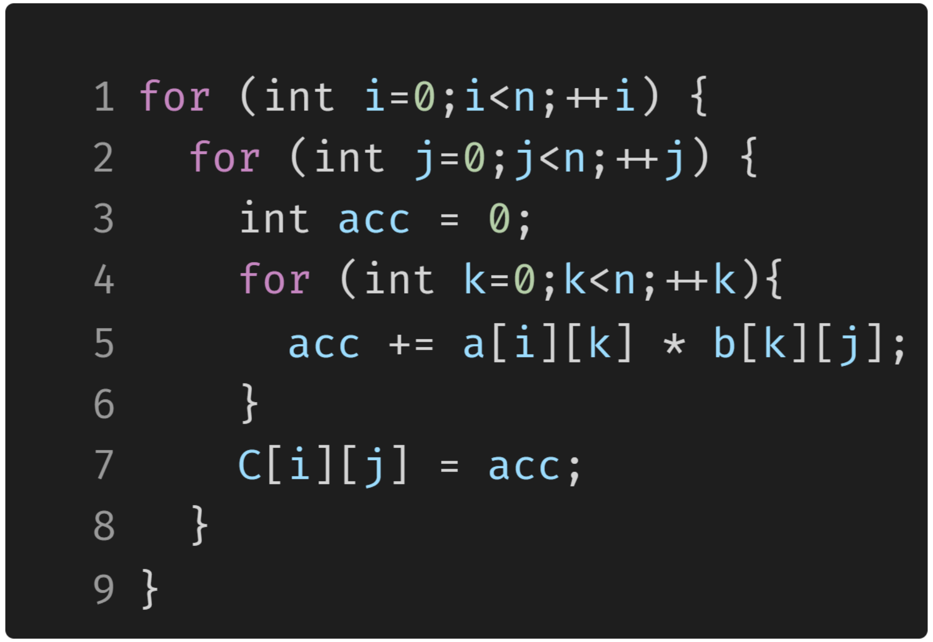
address type

memory as nested interface

simple methods to start computation and report finish

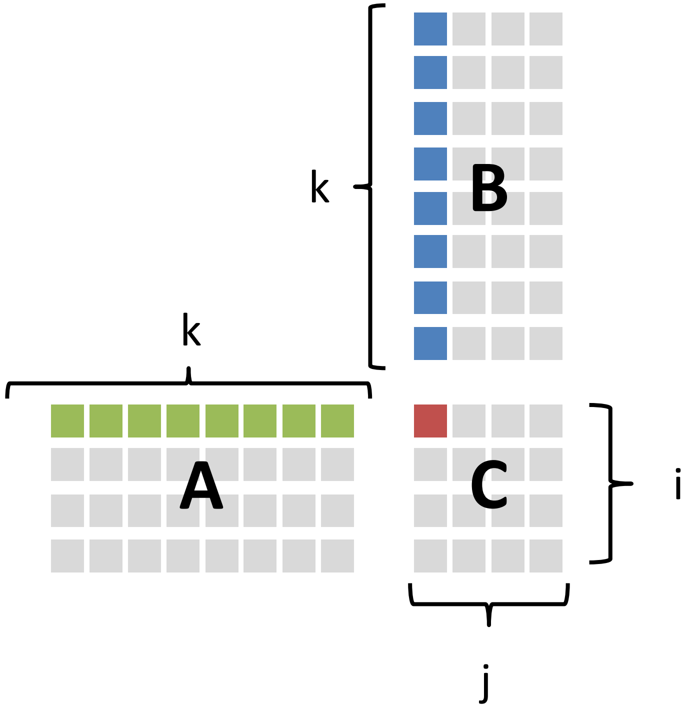
./crates/cmtc/examples/gemm.rs
line 25-36

GeMM Example: Software

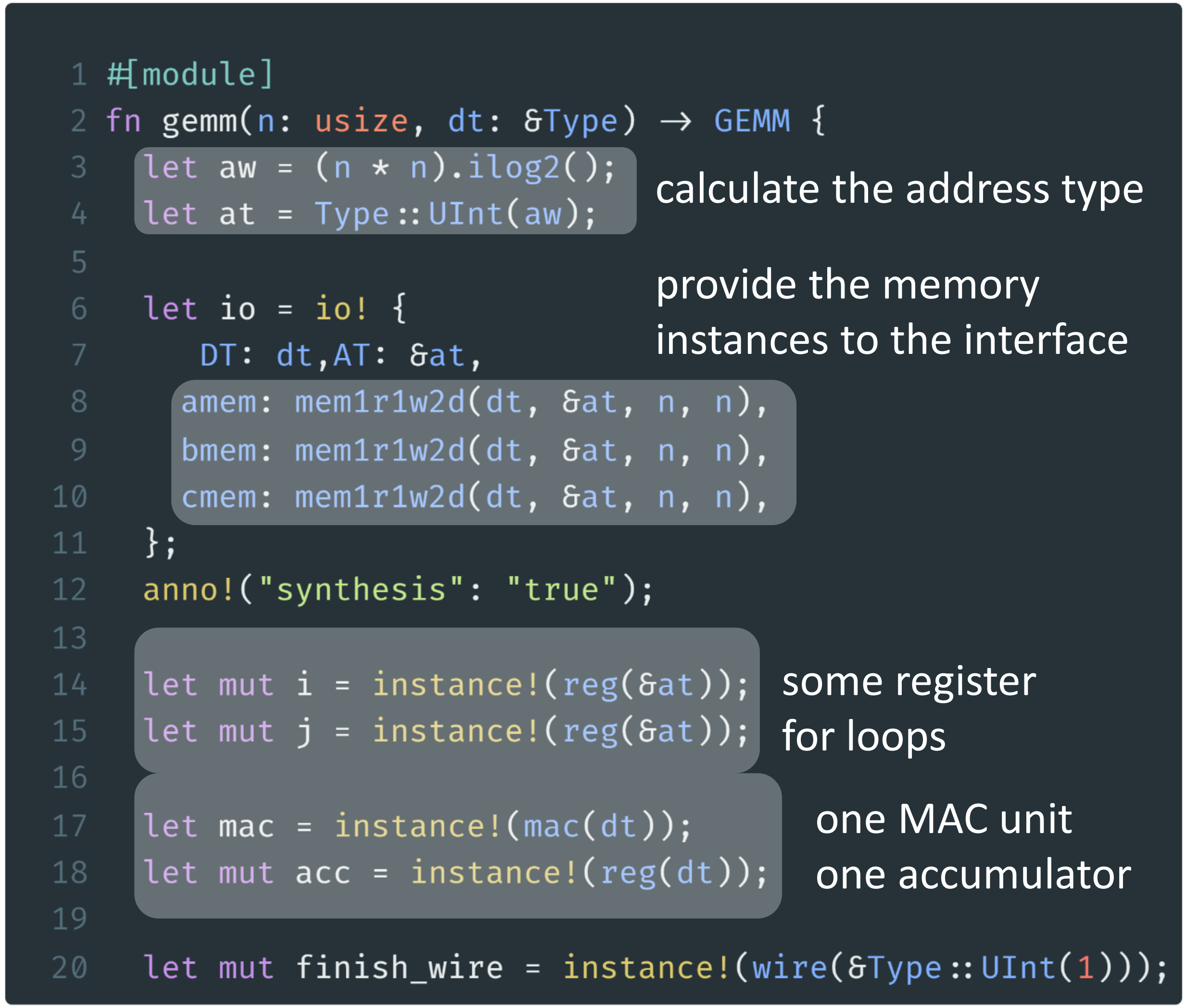


```c
for (int i=0;i<n;++i) {
  for (int j=0;j<n;++j) {
    int acc = 0;
    for (int k=0;k<n;++k){
      acc += a[i][k] * b[k][j];
    }
    C[i][j] = acc;
  }
}
```

GeMM in C using three for-loops



GeMM Example: Preperation



```
#[module]
fn gemm(n: usize, dt: &Type) -> GEMM {
  let aw = (n * n).ilog2();
  let at = Type::UInt(aw);

  let io = io! {
     DT: dt,AT: &at,
    amem: mem1r1w2d(dt, &at, n, n),
    bmem: mem1r1w2d(dt, &at, n, n),
    cmem: mem1r1w2d(dt, &at, n, n),
  };
  anno!("synthesis": "true");

  let mut i = instance!(reg(&at));
  let mut j = instance!(reg(&at));

  let mac = instance!(mac(dt));
  let mut acc = instance!(reg(dt));

  let mut finish_wire = instance!(wire(&Type::UInt(1)));
```

calculate the address type

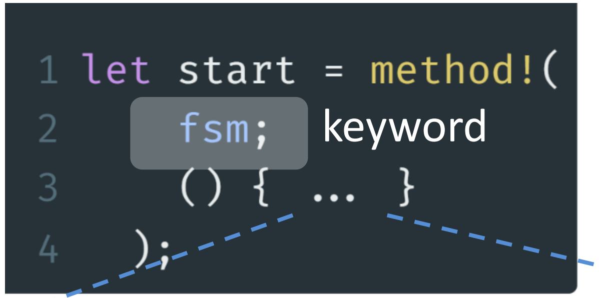
provide the memory instances to the interface

some register for loops

one MAC unit
one accumulator

./crates/cmtc/examples/gemm.rs
line 38-97

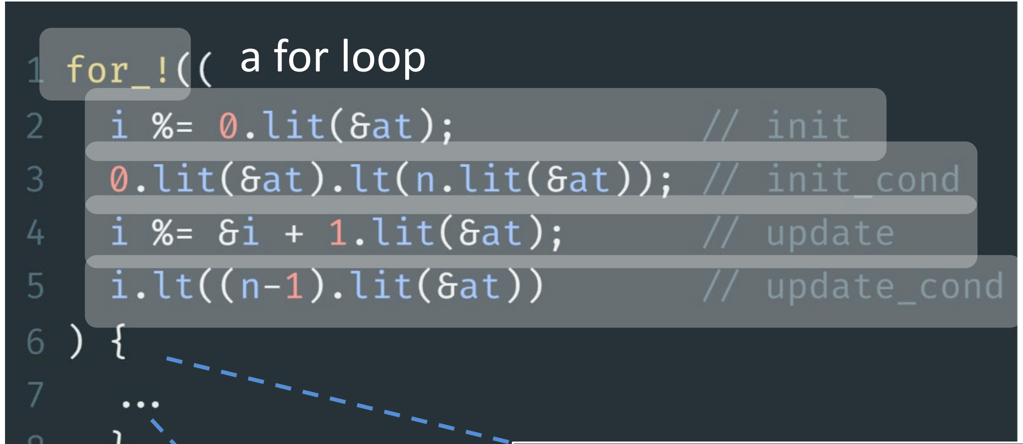
GeMM Example: FSM



```
let start = method!(
    fsm;
    () { … }
);
```

keyword

software-like control description



```
for_!((
  i %= 0.lit(&at);                 // init
  0.lit(&at).lt(n.lit(&at));       // init_cond
  i %= &i + 1.lit(&at);            // update
  i.lt((n-1).lit(&at))             // update_cond
) {
  …
}
```

a for loop

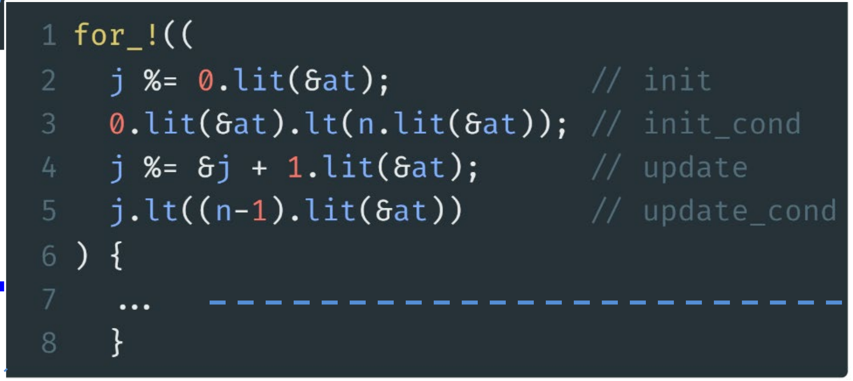
i= 0 to n-1

init: i=0

i<n *i=0* => 0<n

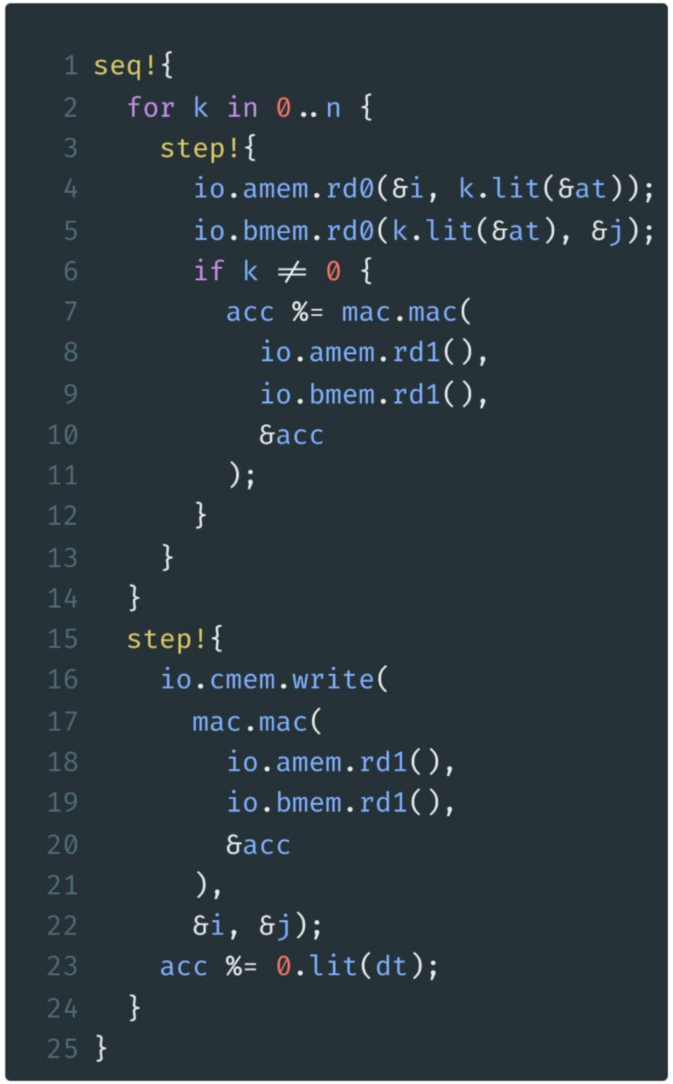
update: i=i+1

i<n *i=i+1* => i<n-1



```
for_!((
  j %= 0.lit(&at);                 // init
  0.lit(&at).lt(n.lit(&at));       // init_cond
  j %= &j + 1.lit(&at);            // update
  j.lt((n-1).lit(&at))             // update_cond
) {
  …
}
```

j= 0 to n-1

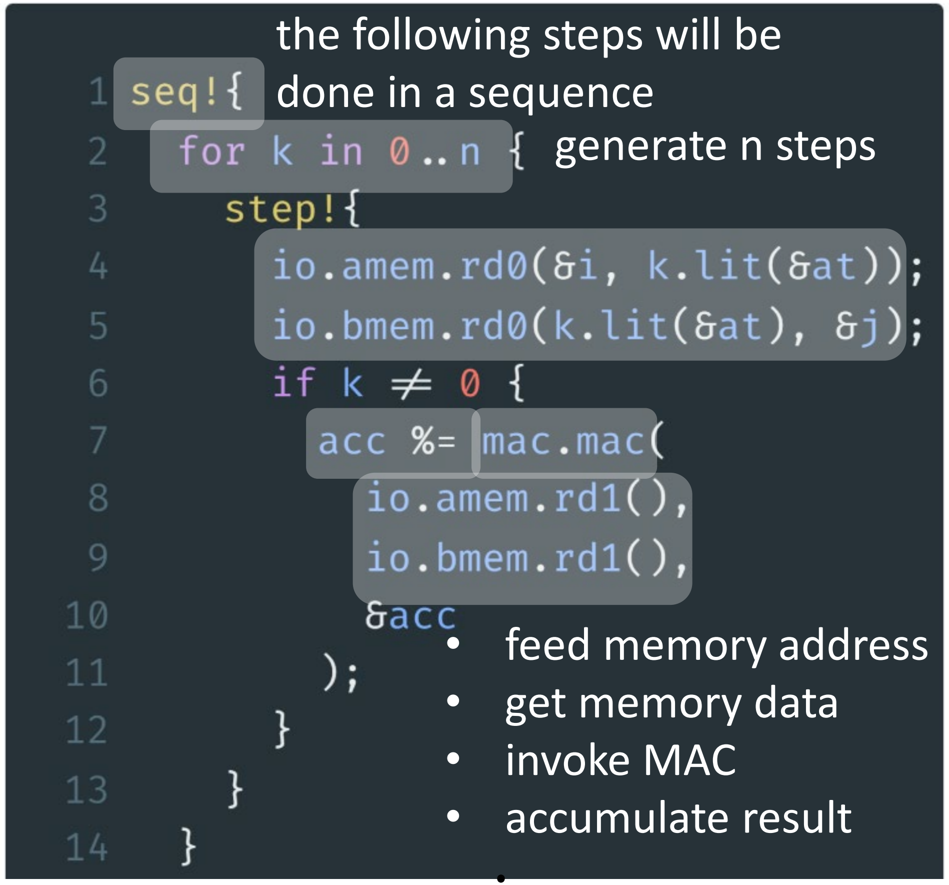


```
seq!{
  for k in 0..n {
    step!{
      io.amem.rd0(&i, k.lit(&at));
      io.bmem.rd0(k.lit(&at), &j);
      if k ≠ 0 {
        acc %= mac.mac(
          io.amem.rd1(),
          io.bmem.rd1(),
          &acc
        );
      }
    }
  }
  step!{
    io.cmem.write(
      mac.mac(
        io.amem.rd1(),
        io.bmem.rd1(),
        &acc
      ),
      &i, &j);
    acc %= 0.lit(dt);
  }
}
```

./crates/cmtc/examples/gemm.rs
line 38-97

GeMM Example: FSM innermost body

- for i in 0..n
  - for j in 0..n

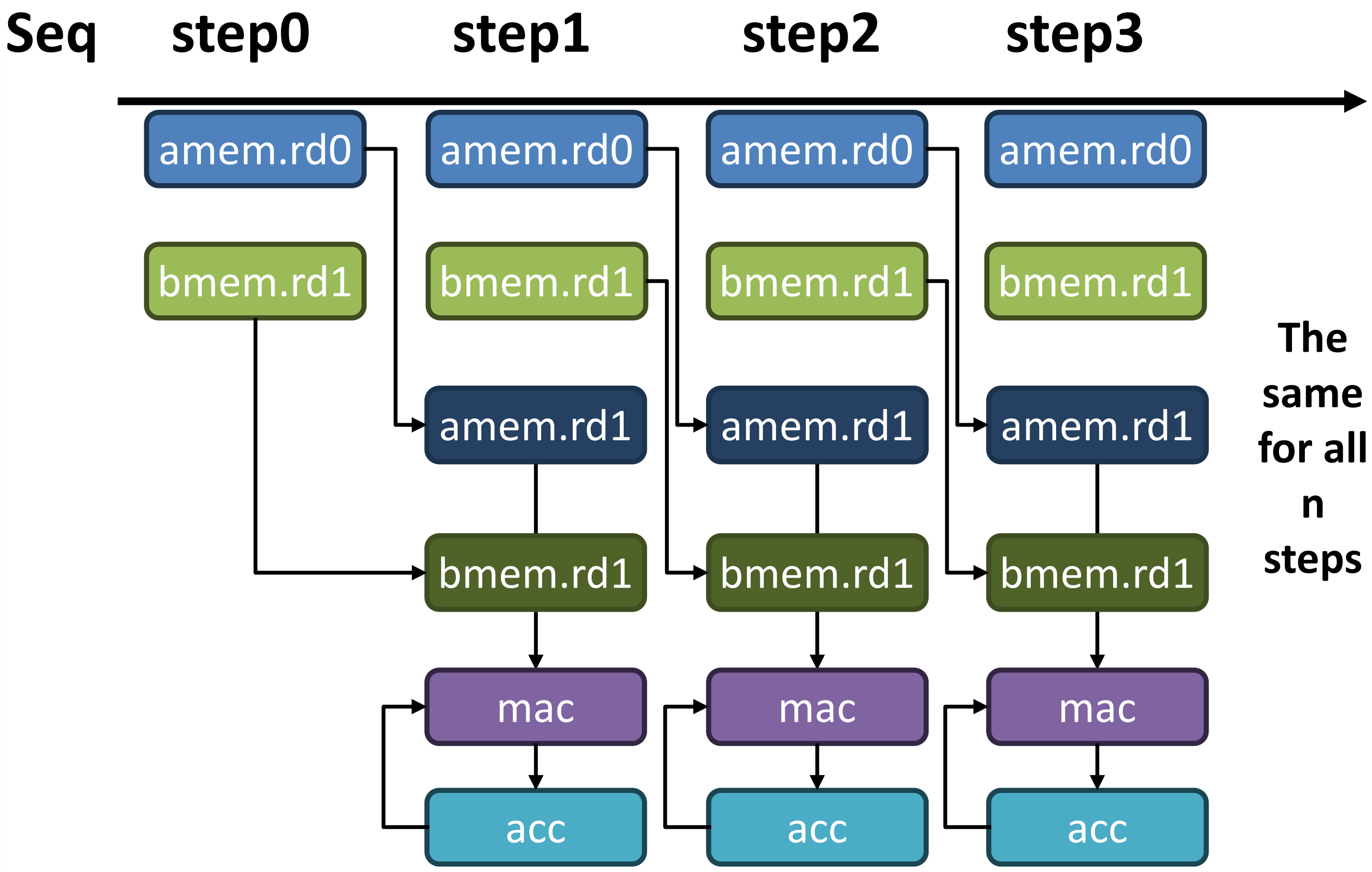


the following steps will be done in a sequence

```
seq!{
  for k in 0..n {
    step!{
      io.amem.rd0(&i, k.lit(&at));
      io.bmem.rd0(k.lit(&at), &j);
      if k ≠ 0 {
        acc %= mac.mac(
          io.amem.rd1(),
          io.bmem.rd1(),
          &acc
        );
      }
    }
  }
```

generate n steps

- feed memory address
- get memory data
- invoke MAC
- accumulate result

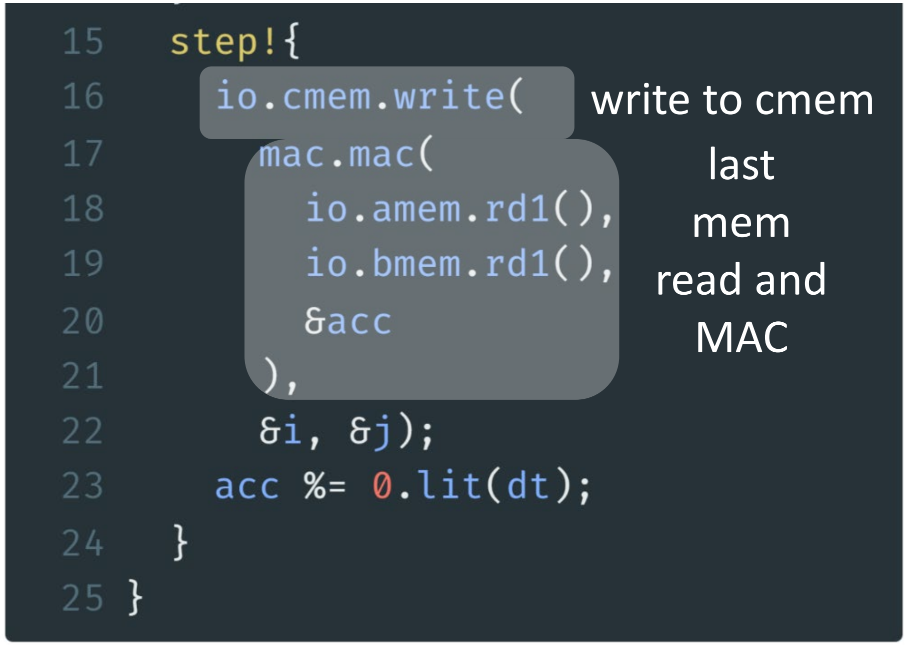


./crates/cmtc/examples/gemm.rs
line 38-97

GeMM Example: FSM last step

- for i in 0..n
  - for j in 0..n

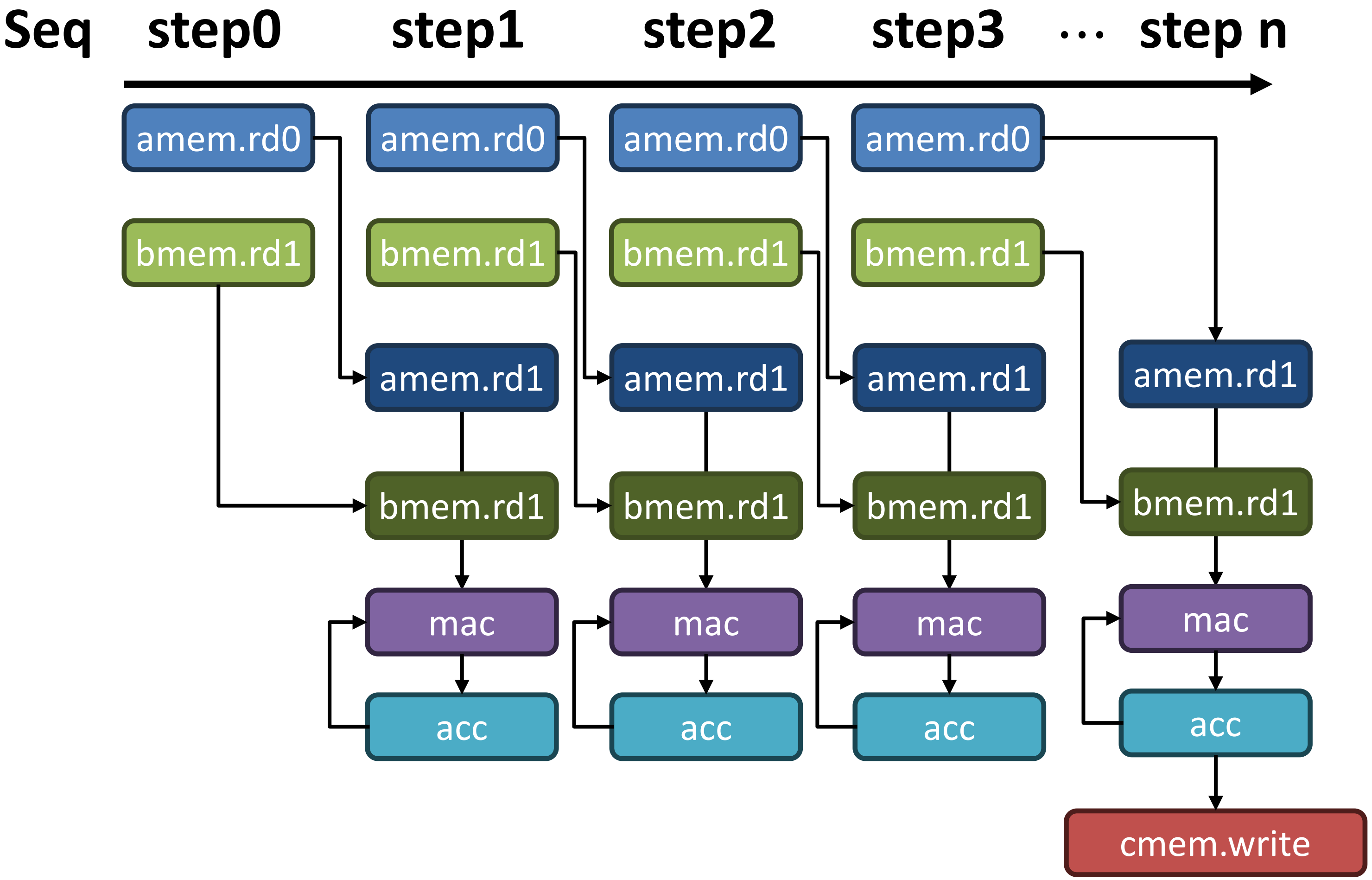
⋮



```
    step!{
      io.cmem.write(
        mac.mac(
          io.amem.rd1(),
          io.bmem.rd1(),
          &acc
        ),
        &i, &j);
      acc %= 0.lit(dt);
    }
}
```

write to cmem

last mem read and MAC

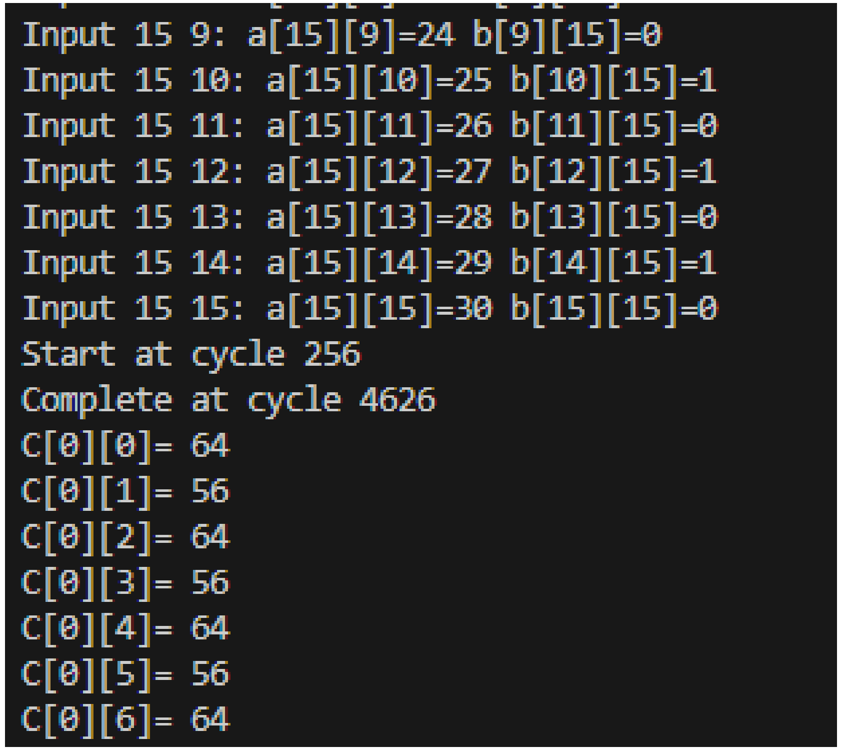


./crates/cmtc/examples/gemm.rs
line 38-97

GeMM Example: Simulation

```
cd $REPOS/cmt2
cargo run --example gemm
cd ./tb/gemm_ksim
make all
./GEMM_s16_i8
```

Commands to run simulation



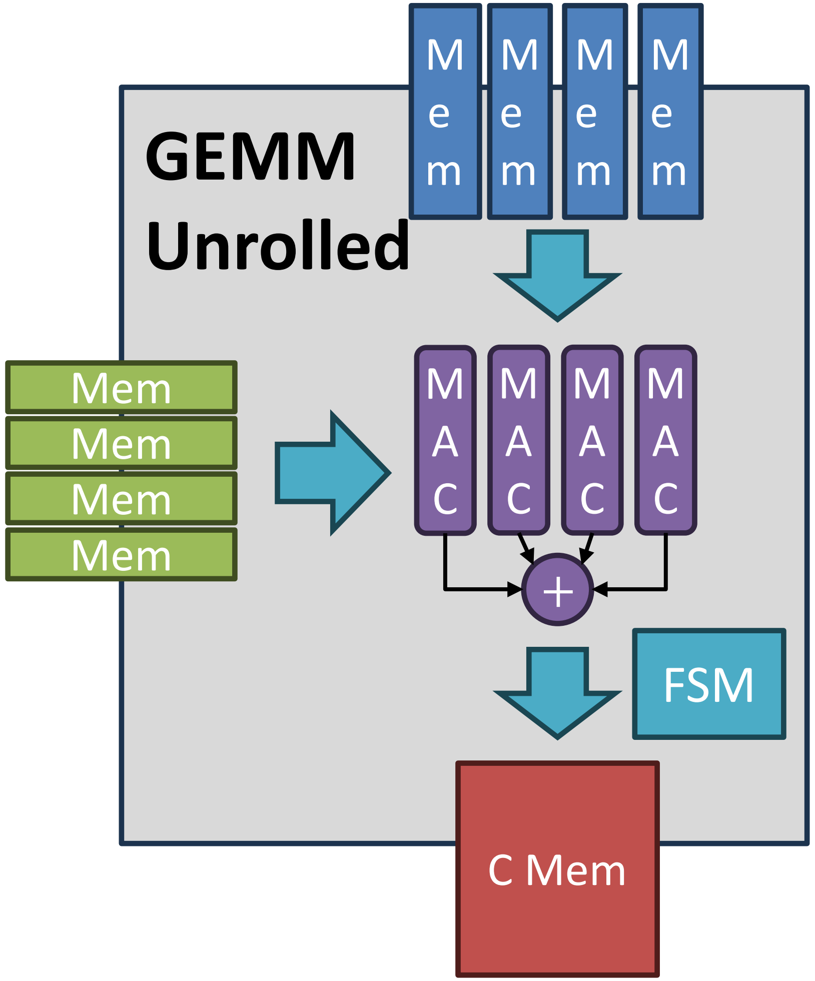
```
Input 15 9: a[15][9]=24 b[9][15]=0
Input 15 10: a[15][10]=25 b[10][15]=1
Input 15 11: a[15][11]=26 b[11][15]=0
Input 15 12: a[15][12]=27 b[12][15]=1
Input 15 13: a[15][13]=28 b[13][15]=0
Input 15 14: a[15][14]=29 b[14][15]=1
Input 15 15: a[15][15]=30 b[15][15]=0
Start at cycle 256
Complete at cycle 4626
C[0][0]= 64
C[0][1]= 56
C[0][2]= 64
C[0][3]= 56
C[0][4]= 64
C[0][5]= 56
C[0][6]= 64
```

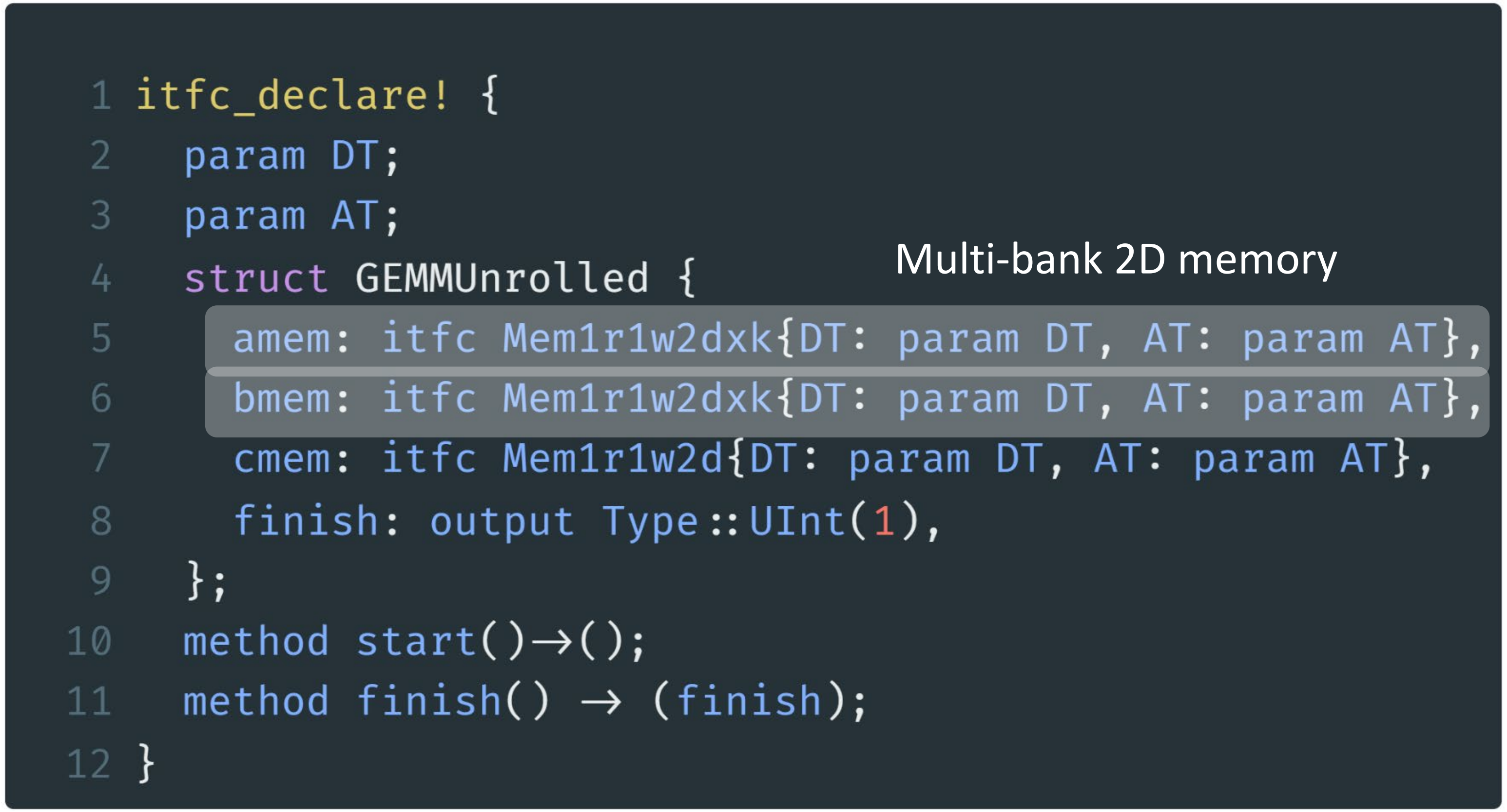
4370

Expected results

Cycles= $((16 + 1)$ //k
$\times 16 + 1)$ //j + 1 cycle to init
$\times 16 + 1$ //i + 1 cycle to init
= 4369 + 1 cycle to write finish

GeMM unrolled Example: Interface Declare



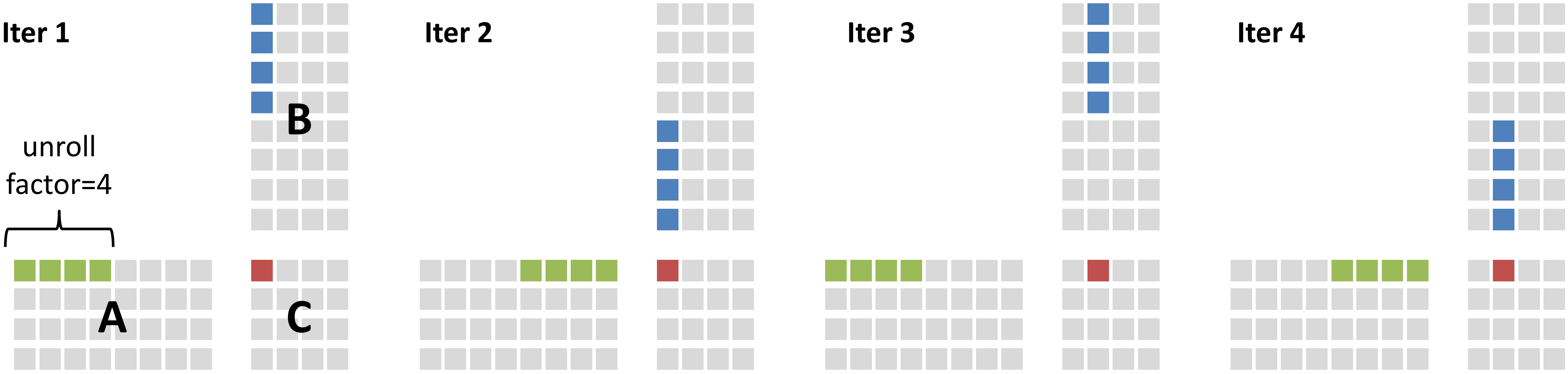


Multi-bank 2D memory

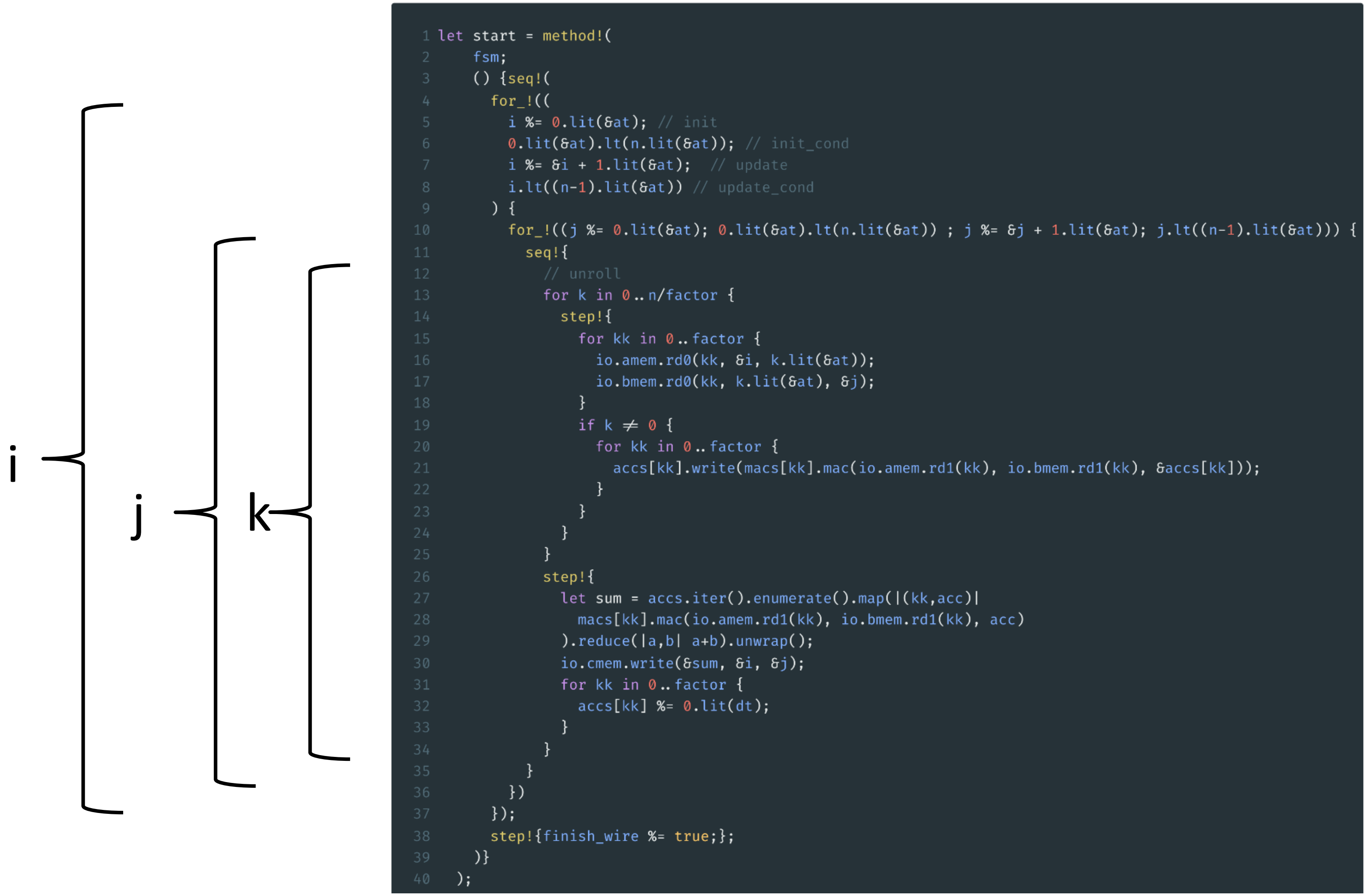
```rust
itfc_declare! {
  param DT;
  param AT;
  struct GEMMUnrolled {
    amem: itfc Mem1r1w2dxk{DT: param DT, AT: param AT},
    bmem: itfc Mem1r1w2dxk{DT: param DT, AT: param AT},
    cmem: itfc Mem1r1w2d{DT: param DT, AT: param AT},
    finish: output Type::UInt(1),
  };
  method start()->();
  method finish() -> (finish);
}
```

./crates/cmtc/examples/gemm_unrolled.rs
line 25-36

GeMM unrolled Example: unroll factor



GeMM unrolled Example

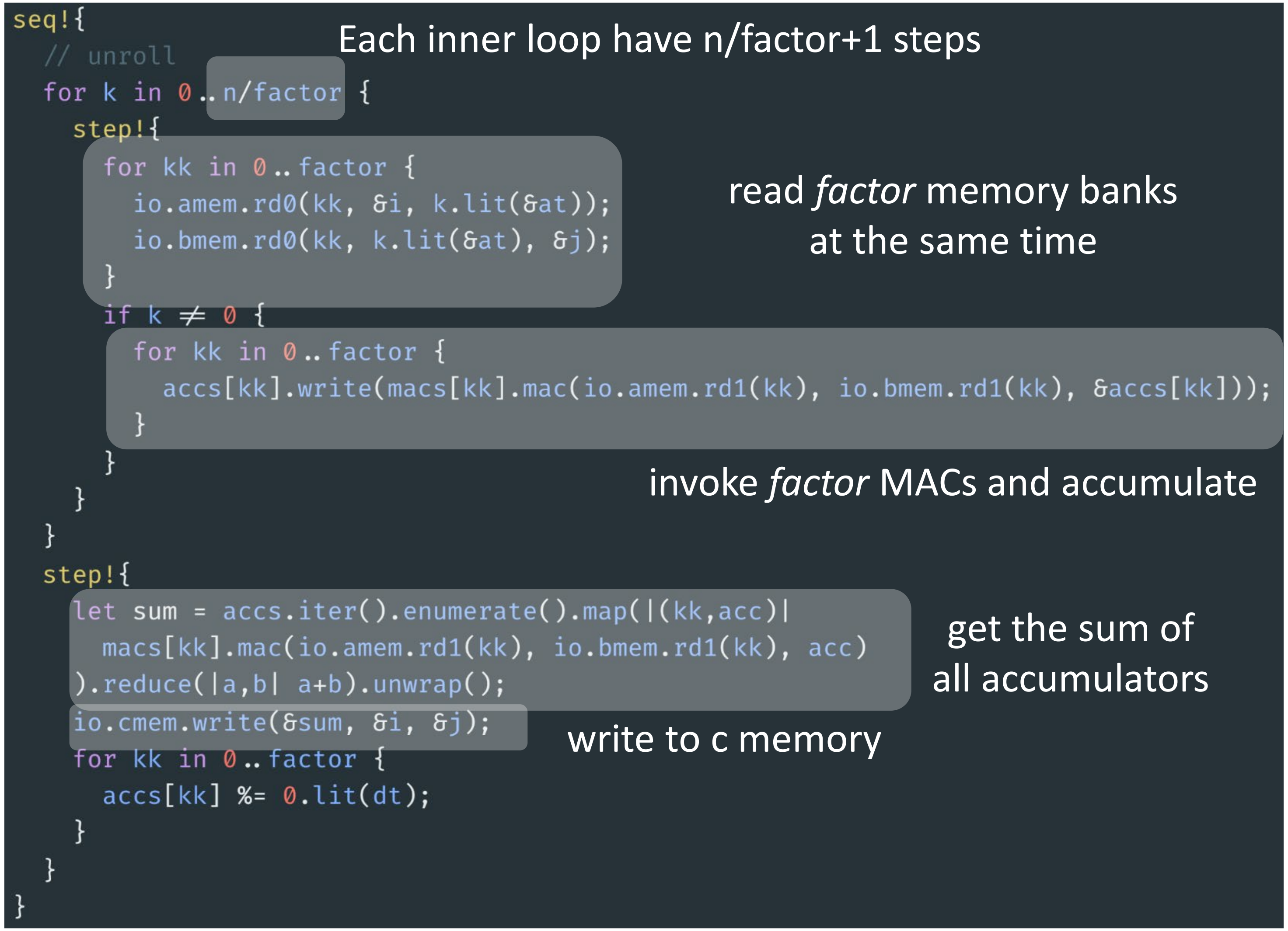


```
let start = method!(
    fsm;
    () {seq!(
      for_!((
        i %= 0.lit(&at); // init
        0.lit(&at).lt(n.lit(&at)); // init_cond
        i %= &i + 1.lit(&at);  // update
        i.lt((n-1).lit(&at)) // update_cond
      ) {
        for_!((j %= 0.lit(&at); 0.lit(&at).lt(n.lit(&at)) ; j %= &j + 1.lit(&at); j.lt((n-1).lit(&at))) {
          seq!{
            // unroll
            for k in 0..n/factor {
              step!{
                for kk in 0..factor {
                  io.amem.rd0(kk, &i, k.lit(&at));
                  io.bmem.rd0(kk, k.lit(&at), &j);
                }
                if k ≠ 0 {
                  for kk in 0..factor {
                    accs[kk].write(macs[kk].mac(io.amem.rd1(kk), io.bmem.rd1(kk), &accs[kk]));
                  }
                }
              }
            }
            step!{
              let sum = accs.iter().enumerate().map(|(kk,acc)|
                macs[kk].mac(io.amem.rd1(kk), io.bmem.rd1(kk), acc)
              ).reduce(|a,b| a+b).unwrap();
              io.cmem.write(&sum, &i, &j);
              for kk in 0..factor {
                accs[kk] %= 0.lit(dt);
              }
            }
          }
        })
      });
      step!{finish_wire %= true;};
    )}
  );
```

./crates/cmtc/examples
/gemm_unrolled.rs
line 38-114

GeMM unrolled Example: innermost

- for i in 0..n
  - for j in 0..n



```
seq!{
  // unroll
  for k in 0..n/factor {
    step!{
      for kk in 0..factor {
        io.amem.rd0(kk, &i, k.lit(&at));
        io.bmem.rd0(kk, k.lit(&at), &j);
      }
      if k ≠ 0 {
        for kk in 0..factor {
          accs[kk].write(macs[kk].mac(io.amem.rd1(kk), io.bmem.rd1(kk), &accs[kk]));
        }
      }
    }
  }
  step!{
    let sum = accs.iter().enumerate().map(|(kk,acc)|
      macs[kk].mac(io.amem.rd1(kk), io.bmem.rd1(kk), acc)
    ).reduce(|a,b| a+b).unwrap();
    io.cmem.write(&sum, &i, &j);
    for kk in 0..factor {
      accs[kk] %= 0.lit(dt);
    }
  }
}
```

Each inner loop have n/factor+1 steps

read *factor* memory banks at the same time

invoke *factor* MACs and accumulate

get the sum of all accumulators

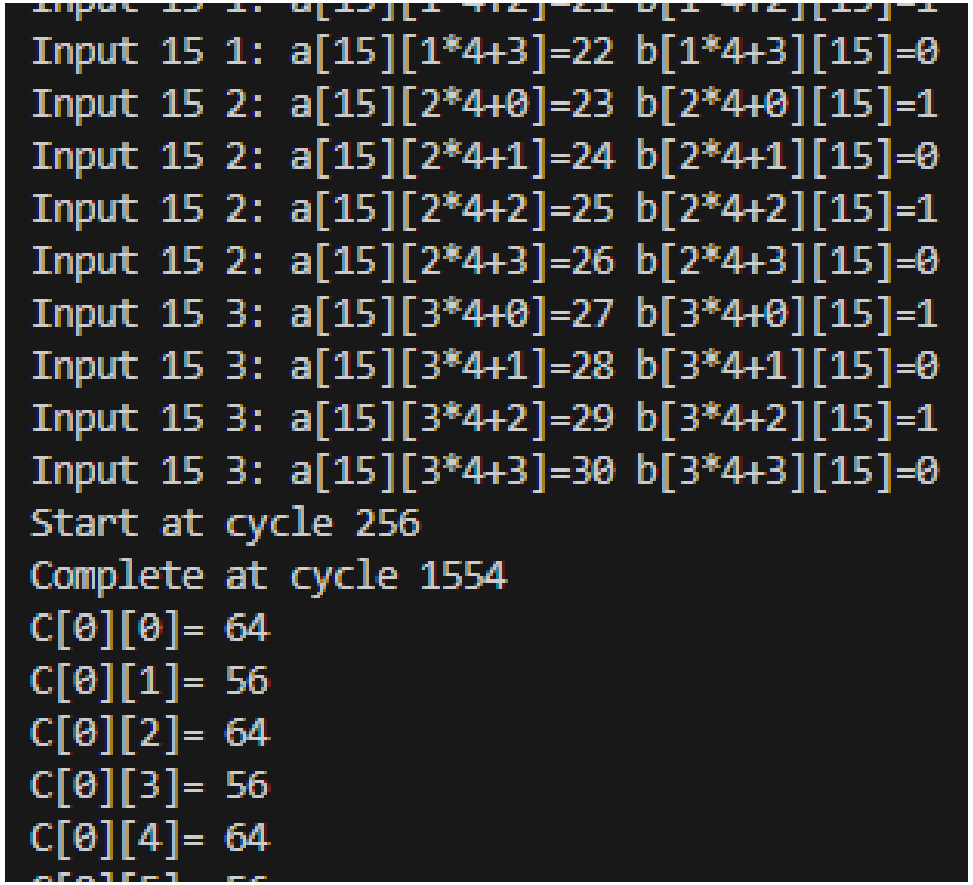
write to c memory

./crates/cmtc/examples
/gemm_unrolled.rs
line 38-114

GeMM unrolled Example: Simulation

```
cd $REPOS/cmt2
cargo run --example gemm_unrolled
cd ./tb/gemm_unrolled_ksim
make all
./GEMMUnrolled_s16_i8
```

Commands to run simulation



```
Input 15 1: a[15][1*4+3]=22 b[1*4+3][15]=0
Input 15 2: a[15][2*4+0]=23 b[2*4+0][15]=1
Input 15 2: a[15][2*4+1]=24 b[2*4+1][15]=0
Input 15 2: a[15][2*4+2]=25 b[2*4+2][15]=1
Input 15 2: a[15][2*4+3]=26 b[2*4+3][15]=0
Input 15 3: a[15][3*4+0]=27 b[3*4+0][15]=1
Input 15 3: a[15][3*4+1]=28 b[3*4+1][15]=0
Input 15 3: a[15][3*4+2]=29 b[3*4+2][15]=1
Input 15 3: a[15][3*4+3]=30 b[3*4+3][15]=0
Start at cycle 256
Complete at cycle 1554
C[0][0]= 64
C[0][1]= 56
C[0][2]= 64
C[0][3]= 56
C[0][4]= 64
```

1298

4370/1298≈3.37

Factor=4 results

Cycles= $((4+1)$ //k
$\times 16+1)$ //j
$\times 16+1$ //i
= 1297

use cmt2 as a library

When you're building a rust project…

```
⌨ cd $REPOS && cargo new hello
```

Add rust version and build flag

```
⌨ cp -r ./cmt2/.cargo ./cmt2/rust-toolchain.toml ./hello
```

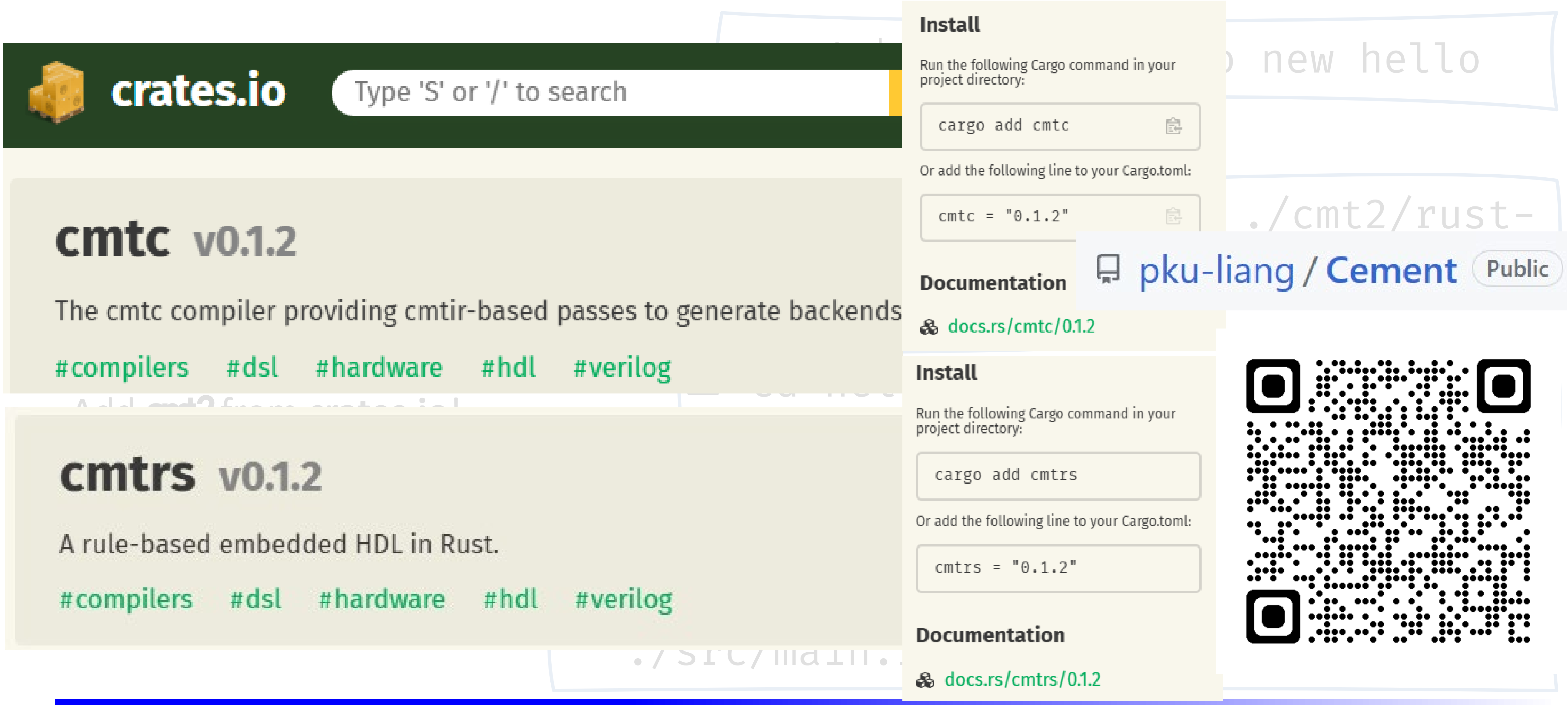
Add **cmt2** from **crates.io**!
This need internet!

```
⌨ cd hello
⌨ cargo add cmtrs cmtc anyhow
```

Now, feel free to use **cmt2** for hardware design

```
⌨ cp ../cmt2/crates/cmtc/examples/hello.rs ./src/main.rs && cargo run
```

use cmt2 as a library



cmtc v0.1.2

The cmtc compiler providing cmtir-based passes to generate backends

#compilers #dsl #hardware #hdl #verilog

cmtrs v0.1.2

A rule-based embedded HDL in Rust.

#compilers #dsl #hardware #hdl #verilog

Install

Run the following Cargo command in your project directory:

```
cargo add cmtc
```

Or add the following line to your Cargo.toml:

```
cmtc = "0.1.2"
```

Documentation

docs.rs/cmtc/0.1.2

pku-liang / Cement Public

Install

Run the following Cargo command in your project directory:

```
cargo add cmtrs
```

Or add the following line to your Cargo.toml:

```
cmtrs = "0.1.2"
```

Documentation

docs.rs/cmtrs/0.1.2

Peking University

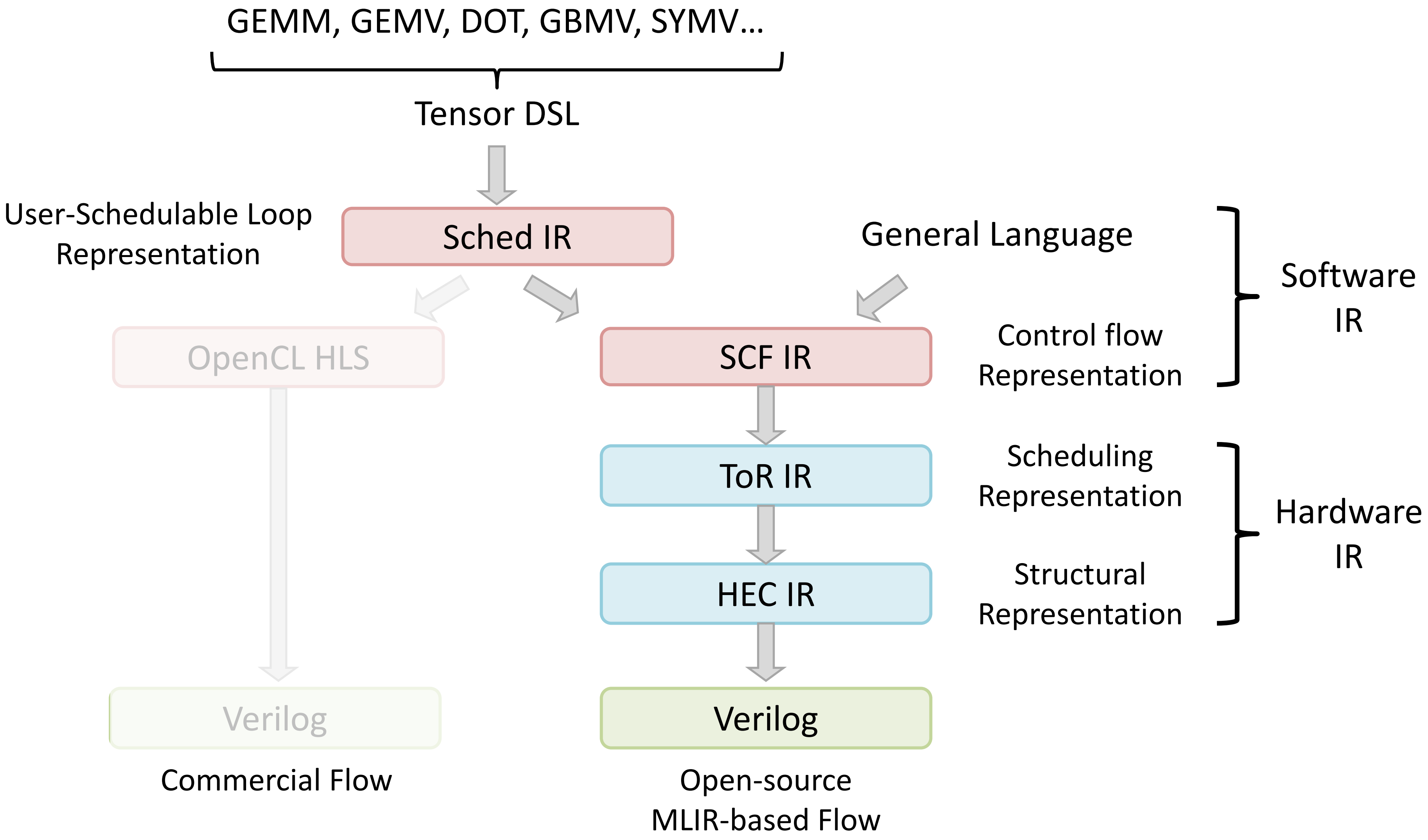
Multi-level IR and debugger for HLS

Speaker: Xiaochen Hao

What You Will Learn

- **Systolic array generation**
  - Learn how to build a systolic array through a DSL
- **BLAS optimizations**
  - Learn how to express optimizations for BLAS routines
- **High-level synthesis and debugger**
  - Learn hardware generation and debugging mechanism

Synthesis Flow



GEMM, GEMV, DOT, GBMV, SYMV…

Tensor DSL

User-Schedulable Loop Representation

Sched IR

General Language

Software IR

OpenCL HLS

SCF IR

Control flow Representation

ToR IR

Scheduling Representation

HEC IR

Structural Representation

Hardware IR

Verilog

Verilog

Commercial Flow

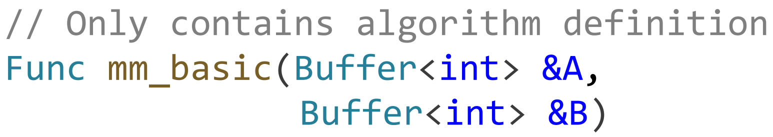
Open-source MLIR-based Flow

Quick Try

- **Run:**

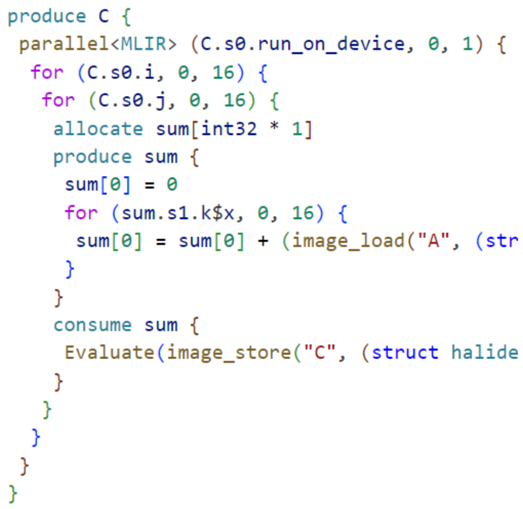
```
cd popa/examples
bash ./hls-tutorial.sh generate basic
```

- **Then you will get:**



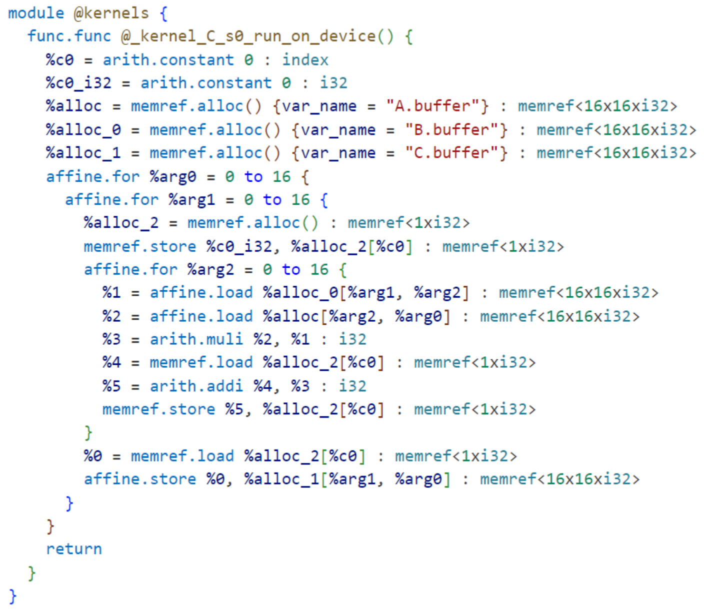
```
// Only contains algorithm definition
Func mm_basic(Buffer<int> &A,
              Buffer<int> &B)
```

Specification
(matrix-multiply.cpp)



```
produce C {
 parallel<MLIR> (C.s0.run_on_device, 0, 1) {
  for (C.s0.i, 0, 16) {
   for (C.s0.j, 0, 16) {
    allocate sum[int32 * 1]
    produce sum {
     sum[0] = 0
     for (sum.s1.k$x, 0, 16) {
      sum[0] = sum[0] + (image_load("A", (str
     }
    }
    consume sum {
     Evaluate(image_store("C", (struct halide
    }
   }
  }
 }
}
```

Sched IR file
(SchedIR_basic)



```
module @kernels {
  func.func @_kernel_C_s0_run_on_device() {
    %c0 = arith.constant 0 : index
    %c0_i32 = arith.constant 0 : i32
    %alloc = memref.alloc() {var_name = "A.buffer"} : memref<16x16xi32>
    %alloc_0 = memref.alloc() {var_name = "B.buffer"} : memref<16x16xi32>
    %alloc_1 = memref.alloc() {var_name = "C.buffer"} : memref<16x16xi32>
    affine.for %arg0 = 0 to 16 {
      affine.for %arg1 = 0 to 16 {
        %alloc_2 = memref.alloc() : memref<1xi32>
        memref.store %c0_i32, %alloc_2[%c0] : memref<1xi32>
        affine.for %arg2 = 0 to 16 {
          %1 = affine.load %alloc_0[%arg1, %arg2] : memref<16x16xi32>
          %2 = affine.load %alloc[%arg2, %arg0] : memref<16x16xi32>
          %3 = arith.muli %2, %1 : i32
          %4 = memref.load %alloc_2[%c0] : memref<1xi32>
          %5 = arith.addi %4, %3 : i32
          memref.store %5, %alloc_2[%c0] : memref<1xi32>
        }
        %0 = memref.load %alloc_2[%c0] : memref<1xi32>
        affine.store %0, %alloc_1[%arg1, %arg0] : memref<16x16xi32>
      }
    }
    return
  }
}
```

SCF IR file
(SCF_basic.mlir)

Domain-specific Language (DSL)

- **Let us look at:**

```
Buffer<float> A(N, M), B(S, N);
Var i("i"), j("j");
RDom k(0, N, "k");
Func C(Place::Device);
C(j, i) = sum(A(k, i) * B(j, k));
```

Declare input buffers

Declare loop variables (k is an reduction loop)

Define computations in a functional expression

```
C.set_bounds(i, 0, M,
             j, 0, S);
```

Specify loop schedules
(How to optimize the computations)

```
Target target = get_host_target();
target.set_feature(Target::MLIR);
C.compile_to_device(“SCF_basic.mlir”, {}, target);
```

Let us target MLIR

Specification
(Func mm_basic,
matrix_multiply.cpp)

IR Lowering

```
   for (C.s0.i, 0, 16) {
    for (C.s0.j, 0, 16) {
     allocate sum[float32 * 1]
     produce sum {
      sum[0] = 0.000000f
      for (sum.s1.k$x, 0, 16) {
        sum[0] = sum[0] + (image_load("A",…)
*image_load("B",…))
      }
     }
     consume sum {
      image_store("C", …, sum[0])
     }
    }
   }
```

Loop structure: tiling, unrolling, vectorization…

Statement: storage flattening, loop invariant hoisting…

```
affine.for %arg0 = 0 to 16 {
  affine.for %arg1 = 0 to 16 {
    %alloc_2 = memref.alloc()
    memref.store %c0_i32, %alloc_2[%c0]
    affine.for %arg2 = 0 to 16 {
      %1 = affine.load %alloc_0[%arg1, %arg2]
      %2 = affine.load %alloc[%arg2, %arg0]
      %3 = arith.muli %2, %1
      %4 = memref.load %alloc_2[%c0]
      %5 = arith.addi %4, %3
      memref.store %5, %alloc_2[%c0]
    }
    %0 = memref.load %alloc_2[%c0]
    affine.store %0, %alloc_1[%arg1, %arg0]
  }
}
```

Sched IR File
(SchedIR_basic)

Lowering →

Lowered SCF IR File
(SCF_basic.mlir)

Systolic Array

- **How to build a systolic array?**

Dataflow

Software World

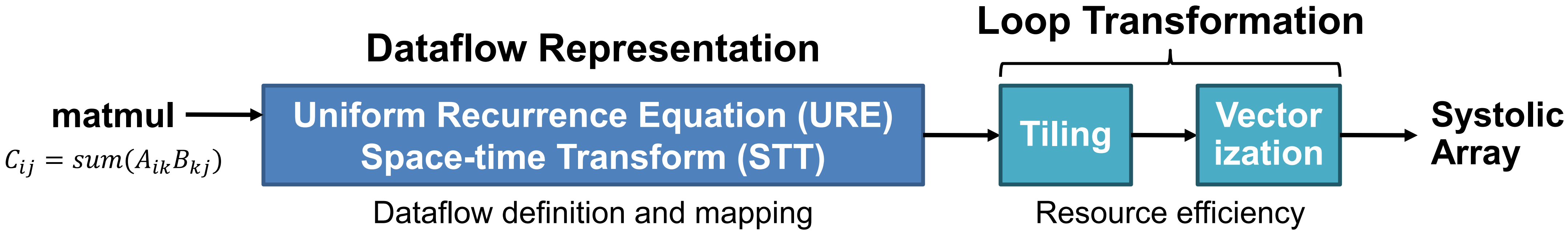
Temporal instruction pipeline

Load-store data access…

Hardware World

Spatial processing elements

Direct data propagation…



Dataflow Representation

Loop Transformation

matmul → **Uniform Recurrence Equation (URE)** **Space-time Transform (STT)** → **Tiling** → **Vectorization** → **Systolic Array**

$C_{ij} = sum(A_{ik}B_{kj})$

Dataflow definition and mapping

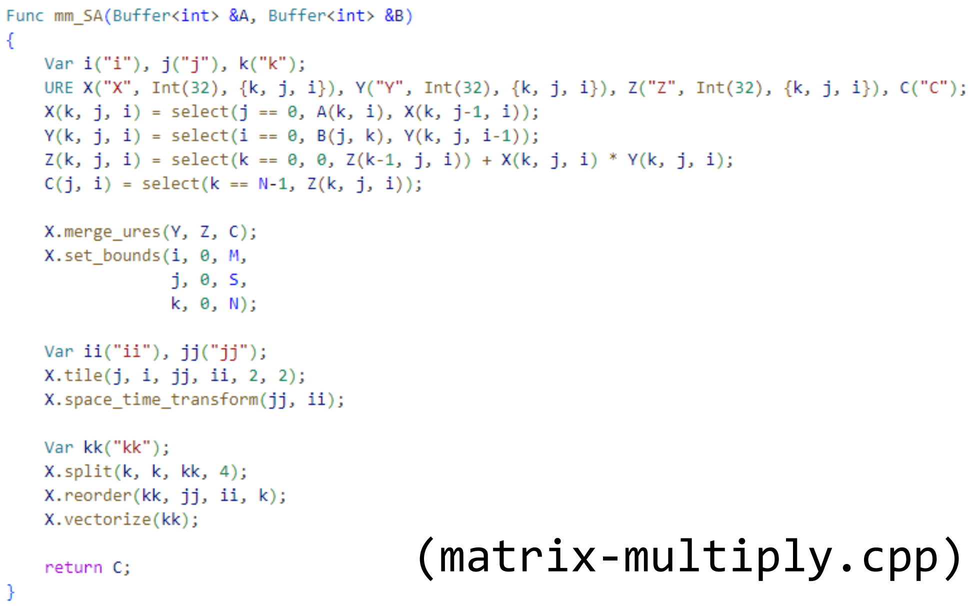
Resource efficiency

Quick Try

- **Run:**

```
cd popa/examples
bash ./hls-tutorial.sh generate SA
```

- **Learn the specification**



```
Func mm_SA(Buffer<int> &A, Buffer<int> &B)
{
    Var i("i"), j("j"), k("k");
    URE X("X", Int(32), {k, j, i}), Y("Y", Int(32), {k, j, i}), Z("Z", Int(32), {k, j, i}), C("C");
    X(k, j, i) = select(j == 0, A(k, i), X(k, j-1, i));
    Y(k, j, i) = select(i == 0, B(j, k), Y(k, j, i-1));
    Z(k, j, i) = select(k == 0, 0, Z(k-1, j, i)) + X(k, j, i) * Y(k, j, i);
    C(j, i) = select(k == N-1, Z(k, j, i));

    X.merge_ures(Y, Z, C);
    X.set_bounds(i, 0, M,
                 j, 0, S,
                 k, 0, N);

    Var ii("ii"), jj("jj");
    X.tile(j, i, jj, ii, 2, 2);
    X.space_time_transform(jj, ii);

    Var kk("kk");
    X.split(k, k, kk, 4);
    X.reorder(kk, jj, ii, k);
    X.vectorize(kk);

    return C;
}
```

Uniform Recurrence Equation (URE)

Space-time Transform (STT)

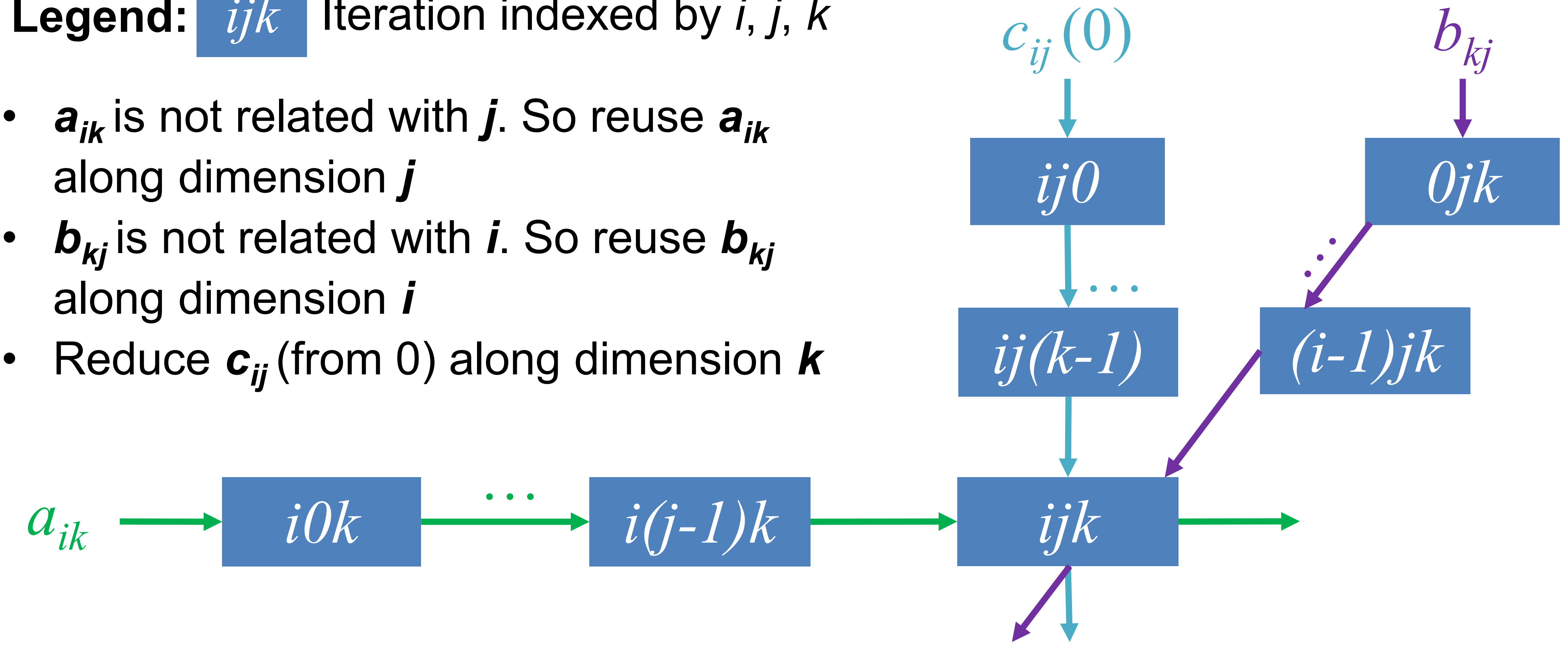
Loop Transformation

(matrix-multiply.cpp)

A dataflow of matrix multiply

Legend: ijk Iteration indexed by *i*, *j*, *k*

- $\boldsymbol{a_{ik}}$ is not related with ***j***. So reuse $\boldsymbol{a_{ik}}$ along dimension ***j***
- $\boldsymbol{b_{kj}}$ is not related with ***i***. So reuse $\boldsymbol{b_{kj}}$ along dimension ***i***
- Reduce $\boldsymbol{c_{ij}}$ (from 0) along dimension ***k***



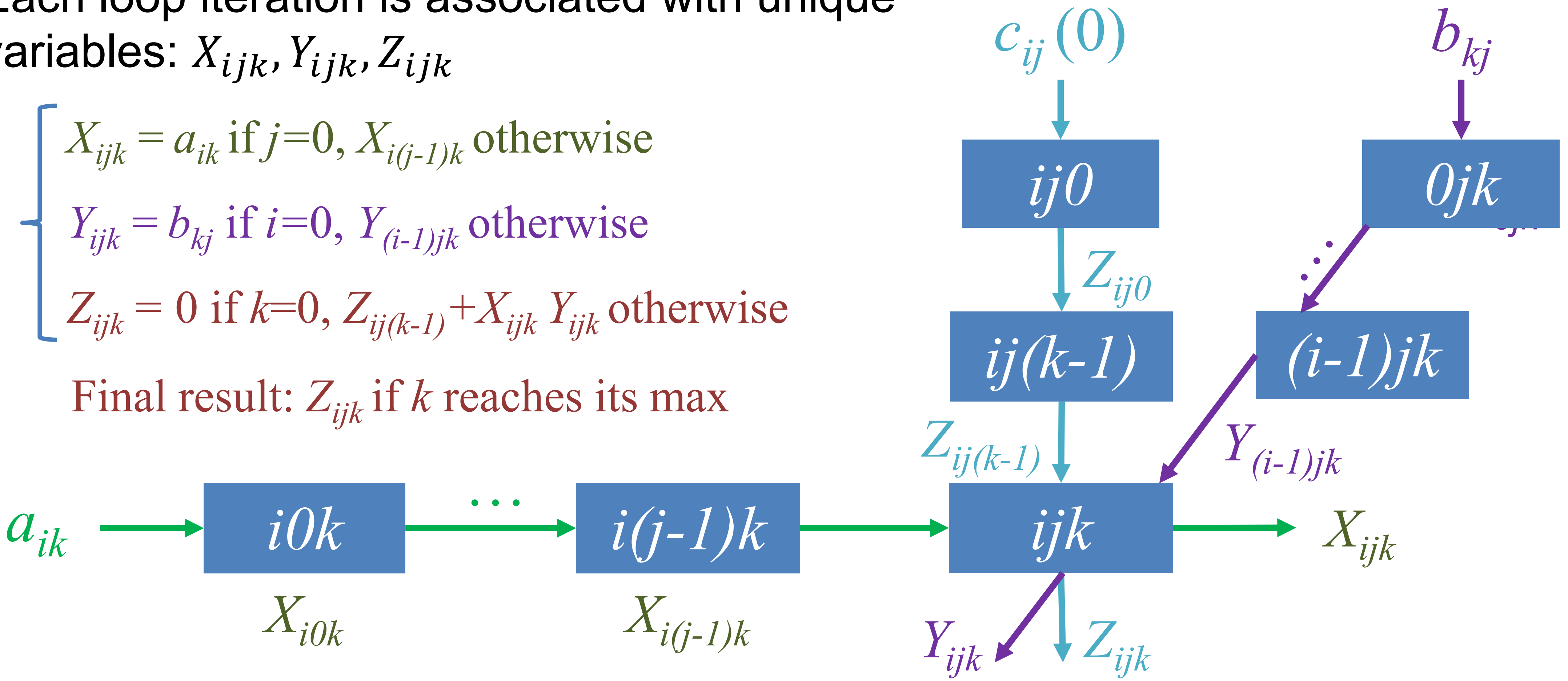
How to express the dataflow

Each loop iteration is associated with unique variables: $X_{ijk}, Y_{ijk}, Z_{ijk}$

UREs

- $X_{ijk} = a_{ik}$ if j=0, $X_{i(j\text{-}1)k}$ otherwise
- $Y_{ijk} = b_{kj}$ if i=0, $Y_{(i\text{-}1)jk}$ otherwise
- Z_{ijk} = 0 if k=0, $Z_{ij(k\text{-}1)} + X_{ijk}\, Y_{ijk}$ otherwise

Final result: Z_{ijk} if k reaches its max



How to express the dataflow

Each loop iteration is associated with unique variables: $X_{ijk}, Y_{ijk}, Z_{ijk}$

UREs
$$\begin{cases} X_{ijk} = a_{ik} \text{ if } j=0,\ X_{i(j\text{-}1)k} \text{ otherwise} \\ Y_{ijk} = b_{kj} \text{ if } i=0,\ Y_{(i\text{-}1)jk} \text{ otherwise} \\ Z_{ijk} = 0 \text{ if } k=0,\ Z_{ij(k\text{-}1)} + X_{ijk}\, Y_{ijk} \text{ otherwise} \end{cases}$$

Final result: Z_{ijk} if k reaches its max

Uniform Recurrence Equations (UREs):

- Recursive relationship
- Uniform dependency
- Dynamic single assignment form

Specification

```
X(k, j, i) = select(j == 0, A(k, i), X(k, j-1, i));
Y(k, j, i) = select(i == 0, B(j, k), Y(k, j, i-1));
Z(k, j, i) = select(k == 0, 0, Z(k-1, j, i)) + X(k, j, i) * Y(k, j, i);
C(j, i) = select(k == K-1, Z(k, j, i));
```

select(condition, x, y)
= (condition ? x : y)

How to express the dataflow

- **Forming loop nest**
  - Each URE is defined within a separate iteration space
  - Group them together under the same loop nest

$$\text{UREs}\begin{cases} X_{ijk} = a_{ik} \text{ if } j{=}0,\ X_{i(j\text{-}1)k} \text{ otherwise} \\ Y_{ijk} = b_{kj} \text{ if } i{=}0,\ Y_{(i\text{-}1)jk} \text{ otherwise} \\ Z_{ijk} = 0 \text{ if } k{=}0,\ Z_{ij(k\text{-}1)} + X_{ijk}\ Y_{ijk} \text{ otherwise} \end{cases}$$

Specification

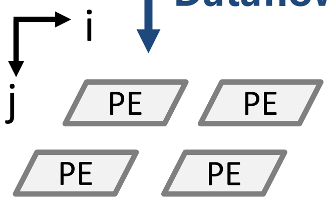
```
…
X.merge_ures(Y, Z, C)
```

Put the UREs `Y, Z, C` in order, into the loop nest of `X`

Initial IR

```
for (i = 0; i < I; i++)
  for (j = 0; j < J; j++)
    for (k = 0; k < K; k++)
      X(k, j, i) = …
      Y(k, j, i) = …
      Z(k, j, i) = …
      C(   j, i) = …
```

Dataflow mapping



Dataflow Mapping

- **Space-time Transform**
  - Mapping loop instances onto spatial PEs
  - Schedule instance execution in a temporal order

Map each point to a new space of $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{t})$

$$STT\left(\begin{bmatrix} i \\ j \\ k \end{bmatrix}\right) = \begin{bmatrix} x \\ y \\ t \end{bmatrix} \begin{array}{l} \left.\vphantom{\begin{array}{c} x \\ y \end{array}}\right\} \text{Space} \\ \left.\vphantom{t}\right\} \text{Time} \end{array}$$

Projection matrix: $$P = \begin{array}{c} \begin{array}{ccc} i & j & k \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array} \begin{array}{l} x = i \\ y = j \end{array}$$

Map point (i, j, k) to (x, y)

Scheduling vector: $$\vec{s}^T = \begin{array}{c} \begin{array}{ccc} i & j & k \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \end{array} \quad t = k + j + i$$

Schedule point (i, j, k) at time t

Dataflow Mapping

- **Space-time Transform**
  - Mapping loop instances onto spatial PEs
  - Schedule instance execution in a temporal order

PE(0,0) t=0:

Map a_{00}->X_{000}, b_{00}->Y_{000}, 0->Z_{000}

PE(0,0) t=1:

Map a_{01}->X_{001}, b_{10}->Y_{001}, Z_{000}->Z_{001}

Map X_{000}->X_{010} (reuse of a_{00})

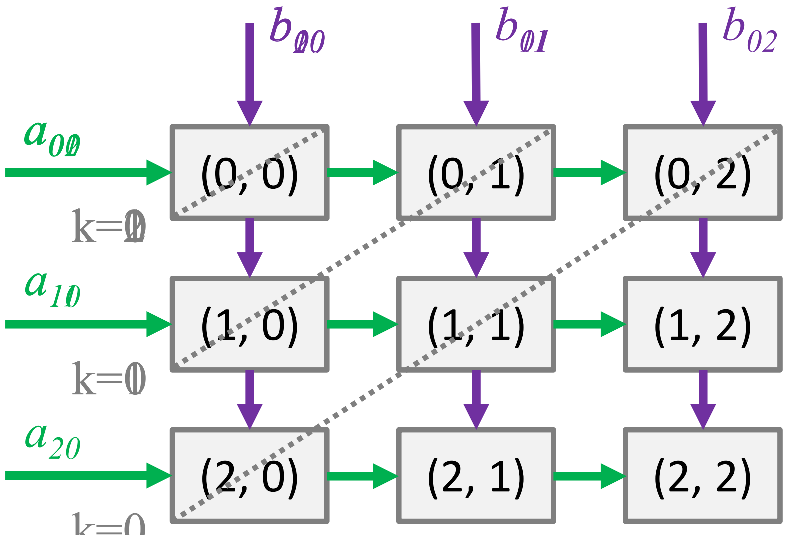
Map Y_{000}->Y_{100} (reuse of b_{00})

PE(0,0) t=2:

Map a_{02}->X_{002}, b_{20}->Y_{002}, Z_{001}->Z_{002}

Map X_{001}->X_{011} (reuse of a_{01})

Map Y_{001}->Y_{101} (reuse of b_{10})



PE Array

Dataflow Mapping

- **Specify space-time transform**
  - Unscheduled STT:

```
…
X.space_time_transform(j, i)  Specify only space loops
```

Specification

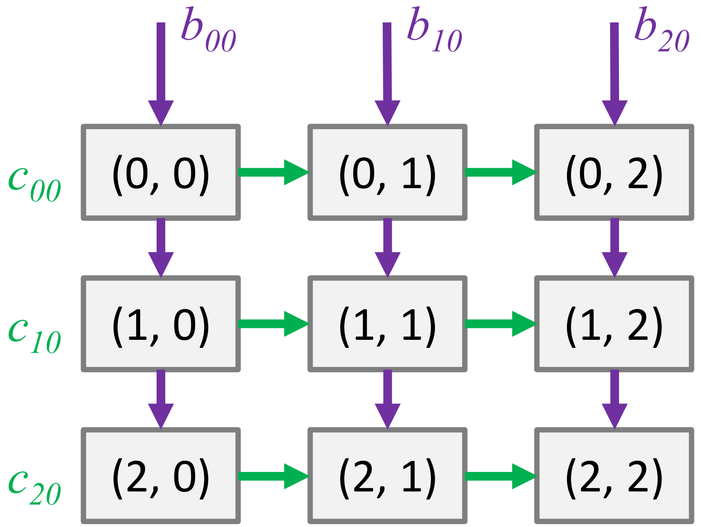
  - Scheduled STT:

```
X.space_time_transform({k, j, i}    Source loops
                       {t, y, x},   Destination loops
                       {1, 1, 1,
                        0, 1, 0,    Projection matrix
                        0, 0, 1})   Scheduling vector
```

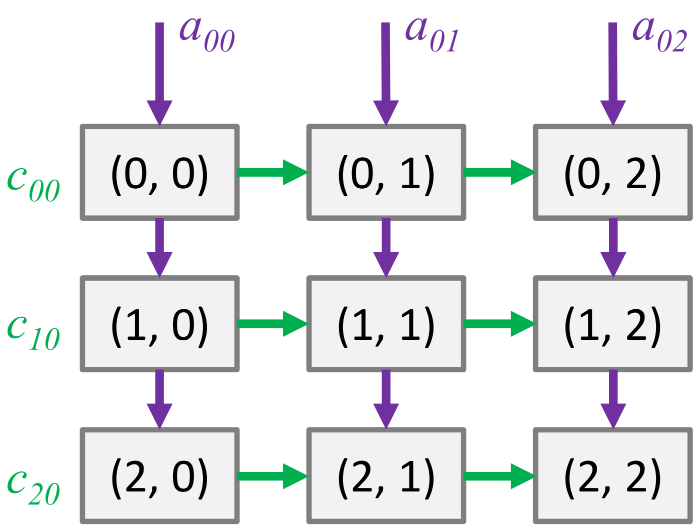
Specification (Equivalent version)

Dataflow Mapping

- **Exploring dataflow designs using space-time transform**
  - Keeping input A stationary
    - `space_time_transform(k, i)`
  - Keeping input B stationary
    - `space_time_transform(k, j)`



PE Array



PE Array

Compilation

- **IR transformation and code generation**

```
…
realize X.shreg[0, 16], [0, 16] {
 unrolled for (X.s0.i, 0, 16) {
  unrolled for (X.s0.j, 0, 16) {
   pipelined for (X.s0.k, 0, 16) {
    write_shift_reg("X.shreg", X.s0.j, X.s0.i,
      select(X.s0.j == 0, image_load("A",…),
        read_shift_reg("X.shreg", X.s0.j-1, X.s0.i)))
    write_shift_reg("Y.shreg",…)
    write_shift_reg("Z.shreg",…)
   }
   image_store("C", …)
  }
 }
}
```

Loop annotation

Mapped Dataflow

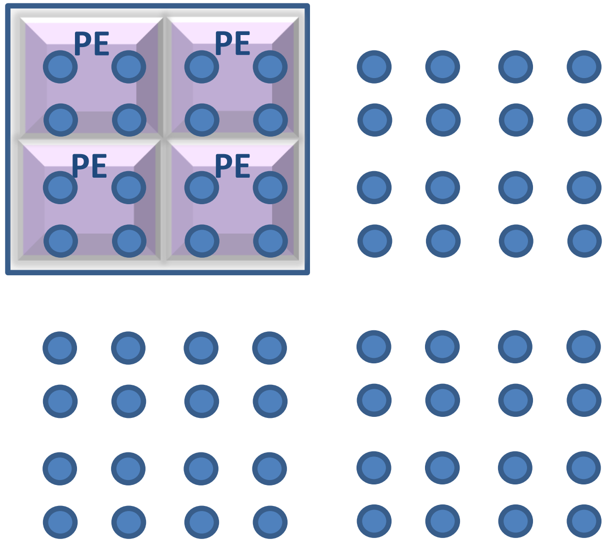
Sched IR

```
…
%alloc_4 = memref.alloc() : memref<16x16xi32>
affine.for %arg0 = 0 to 16 {
 affine.for %arg1 = 0 to 16 {
  affine.for %arg2 = 0 to 16 {
    %1 = affine.load %alloc_4[%arg1 - 1, %arg0]
    %2 = affine.load %alloc[%arg2, %arg0]
    %5 = arith.select %4, %2, %1
    affine.store %5, %alloc_4[%arg1, %arg0]
    …
  } {pipeline = 1}
  %0 = affine.load %alloc_2[%arg1, %arg0]
  affine.store %0, %alloc_1[%arg1, %arg0]
 } {unroll = 0}
} {unroll = 0}
```

Lowered SCF IR

Tiling

- **Scaling to large matrices**
  - Partition the output matrix into tiles
  - Compute tile-by-tile using a systolic array
- **Restructure the loops**



```
X.tile(j, i, jj, ii, 2, 2)
X.space_time_transform(jj, ii)
```

Specification

```
for (X.s0.i, 0, 8) {
 for (X.s0.j, 0, 8) {
  pipelined (X.s0.k, 0, 16) {
   unrolled (X.s0.ii, 0, 2) {
    unrolled (X.s0.jj, 0, 2) {
     write_shift_reg("X.shreg", …, read_shift_reg("X.shreg", X.s0.jj + -1, X.s0.ii))
     write_shift_reg("Y.shreg", …, read_shift_reg("Y.shreg", X.s0.jj, X.s0.ii + -1))
     …
}}}}}
```

Time loop

Space loop

Sched IR

Tiling

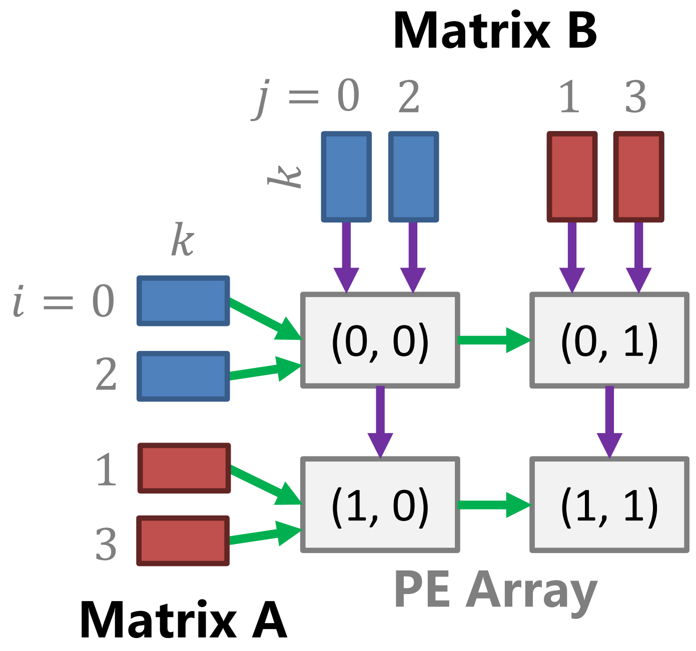
- **Array Partitioning**
  - Splitting an array into small partitions
  - Partitions can be accessed in parallel
- **Annotation based on loop unrolling**

```
%alloc = memref.alloc() { Matrix A
  partition_cyclic_array = [1],
  partition_dim_array = [1],
  partition_factor_array = [2]}
%alloc_0 = memref.alloc() { Matrix B
  partition_cyclic_array = [1],
  partition_dim_array = [0],
  partition_factor_array = [2]}
```

Partitioning dimension i cyclically

Partitioning dimension j cyclically

SCF IR



Vectorization

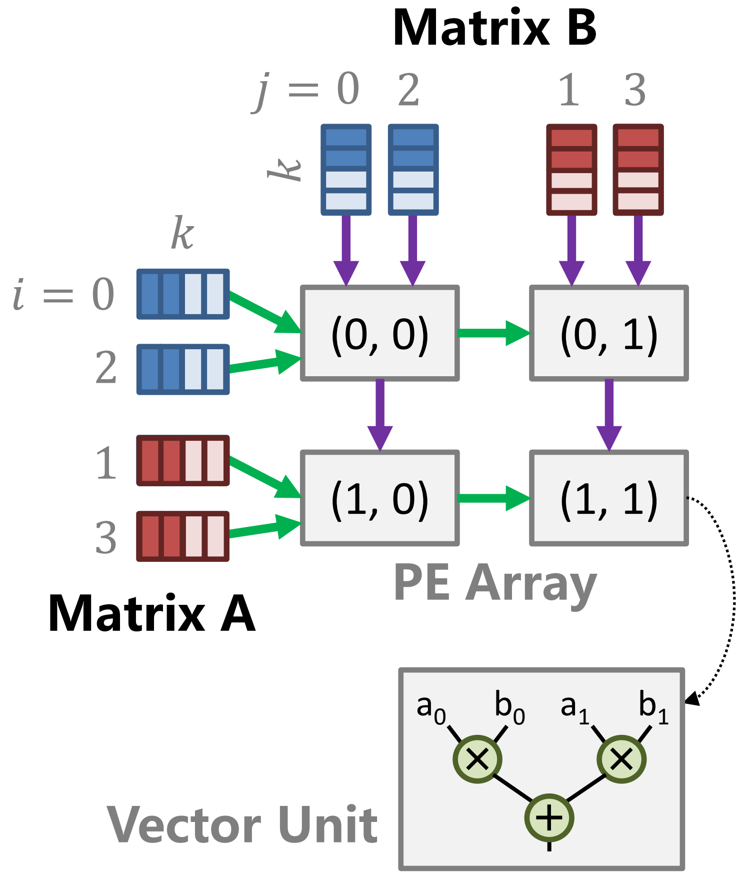
- **Enabling more parallelism**
  – Duplicating compute units in each PE
  – Improve resource efficiency
- **Restructure the loops**

```
X.split(k, k, kk, 4)
X.vectorization(kk)
```

Specification

```
for (X.s0.i, 0, 8)
 for (X.s0.j, 0, 8)
  pipelined (X.s0.k, 0, 4)
   unrolled (X.s0.ii, 0, 2)
    unrolled (X.s0.jj, 0, 2)
     unrolled (X.s0.kk, 0, 4)
      …
```

Sched IR



Quick Try

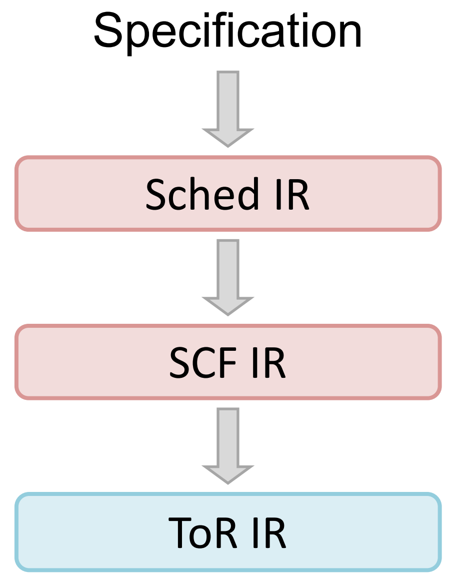
- **Run:**

```
bash ./hls-tutorial.sh run basic
bash ./hls-tutorial.sh run SA
```

- **Then you will get:**

	basic	SA
Cycle	15411	1691

The systolic array achieves a 9X speedup compared to the basic version



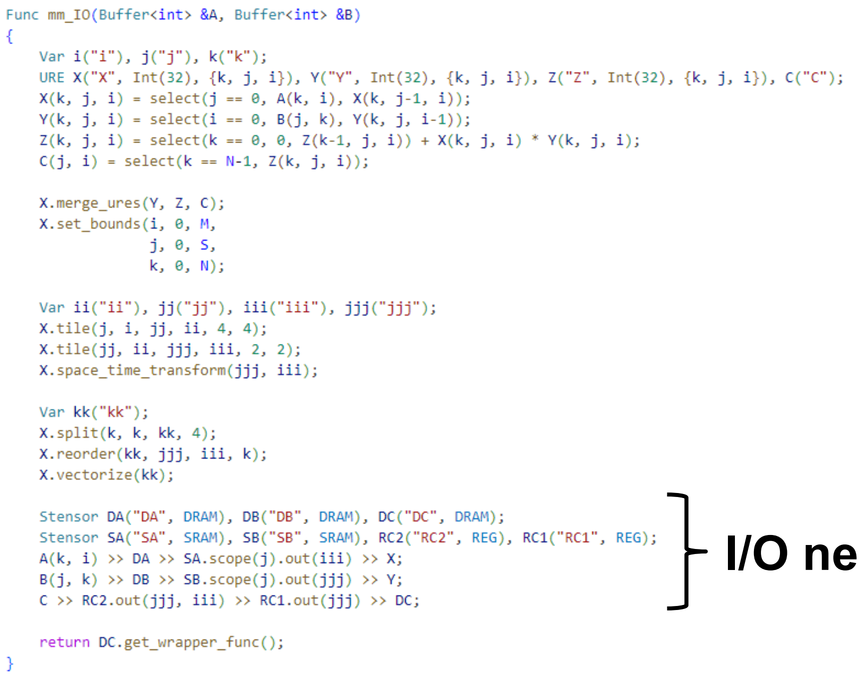
Interpreted execution at the ToR IR level

Quick Try

- **Run:**

```
cd popa/examples
bash ./hls-tutorial.sh generate IO
```

- **Learn the specification:**



```
Func mm_IO(Buffer<int> &A, Buffer<int> &B)
{
    Var i("i"), j("j"), k("k");
    URE X("X", Int(32), {k, j, i}), Y("Y", Int(32), {k, j, i}), Z("Z", Int(32), {k, j, i}), C("C");
    X(k, j, i) = select(j == 0, A(k, i), X(k, j-1, i));
    Y(k, j, i) = select(i == 0, B(j, k), Y(k, j, i-1));
    Z(k, j, i) = select(k == 0, 0, Z(k-1, j, i)) + X(k, j, i) * Y(k, j, i);
    C(j, i) = select(k == N-1, Z(k, j, i));

    X.merge_ures(Y, Z, C);
    X.set_bounds(i, 0, M,
                 j, 0, S,
                 k, 0, N);

    Var ii("ii"), jj("jj"), iii("iii"), jjj("jjj");
    X.tile(j, i, jj, ii, 4, 4);
    X.tile(jj, ii, jjj, iii, 2, 2);
    X.space_time_transform(jjj, iii);

    Var kk("kk");
    X.split(k, k, kk, 4);
    X.reorder(kk, jjj, iii, k);
    X.vectorize(kk);

    Stensor DA("DA", DRAM), DB("DB", DRAM), DC("DC", DRAM);
    Stensor SA("SA", SRAM), SB("SB", SRAM), RC2("RC2", REG), RC1("RC1", REG);
    A(k, i) >> DA >> SA.scope(j).out(iii) >> X;
    B(j, k) >> DB >> SB.scope(j).out(jjj) >> Y;
    C >> RC2.out(jjj, iii) >> RC1.out(jjj) >> DC;

    return DC.get_wrapper_func();
}
```

(matrix-multiply.cpp)

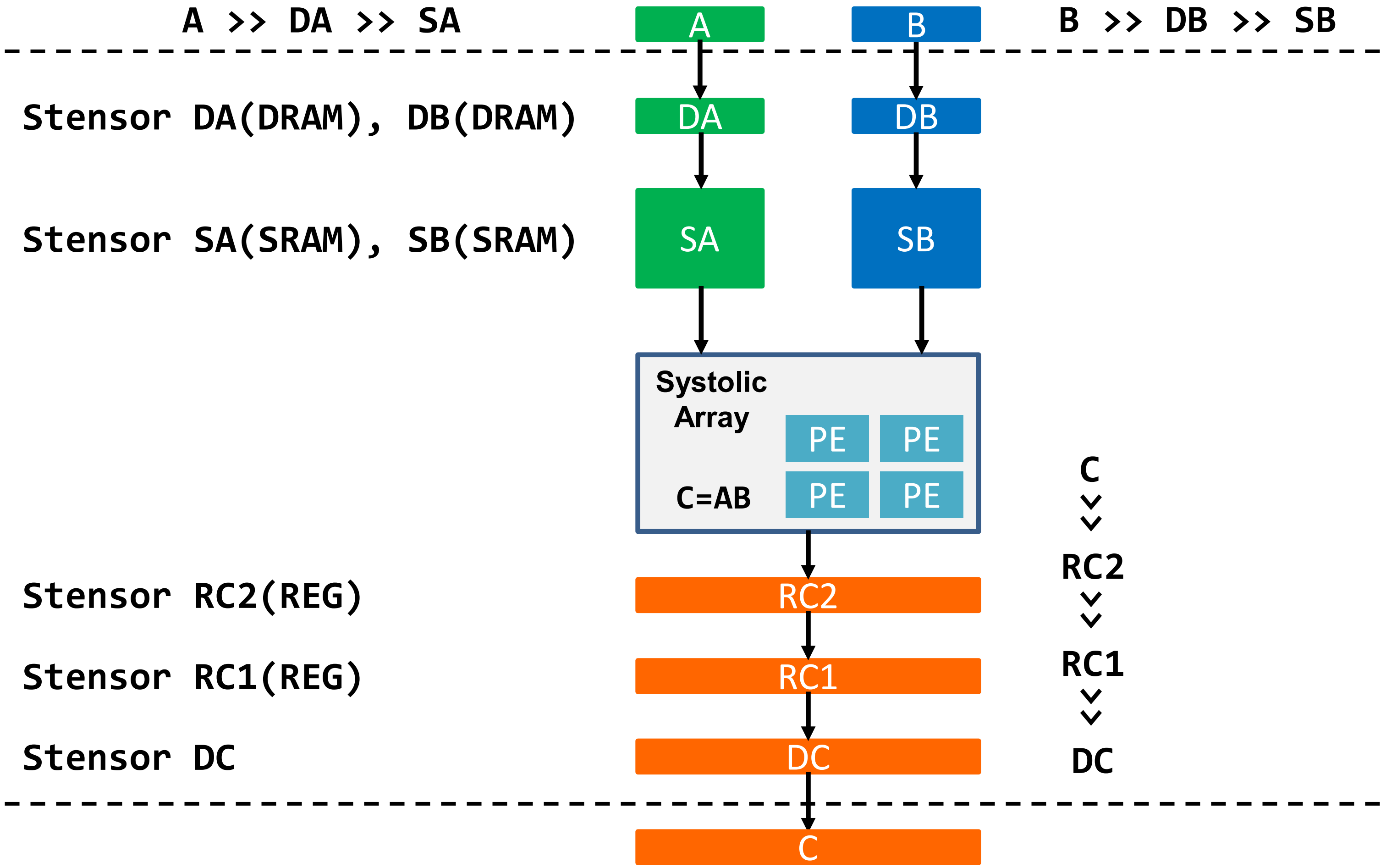
} **I/O network (Streaming tensors, stensor)**

Streaming Tensors

- **Abstract, self-managing, customizable storage of a tensor**
  - Memory Level
  - Data movements

Specification (mm_IO)

```
Stensor DA(DRAM), DB(DRAM)
Stensor SA(SRAM), SB(SRAM)
Stensor RC2(REG), RC1(REG)
Stensor DC(DRAM)
A(k, i) >> DA >> SA >> X
B(j, k) >> DB >> SB >> Y
C >> RC2 >> RC1 >> DC
```



A >> DA >> SA

B >> DB >> SB

Stensor DA(DRAM), DB(DRAM)

Stensor SA(SRAM), SB(SRAM)

Stensor RC2(REG)

Stensor RC1(REG)

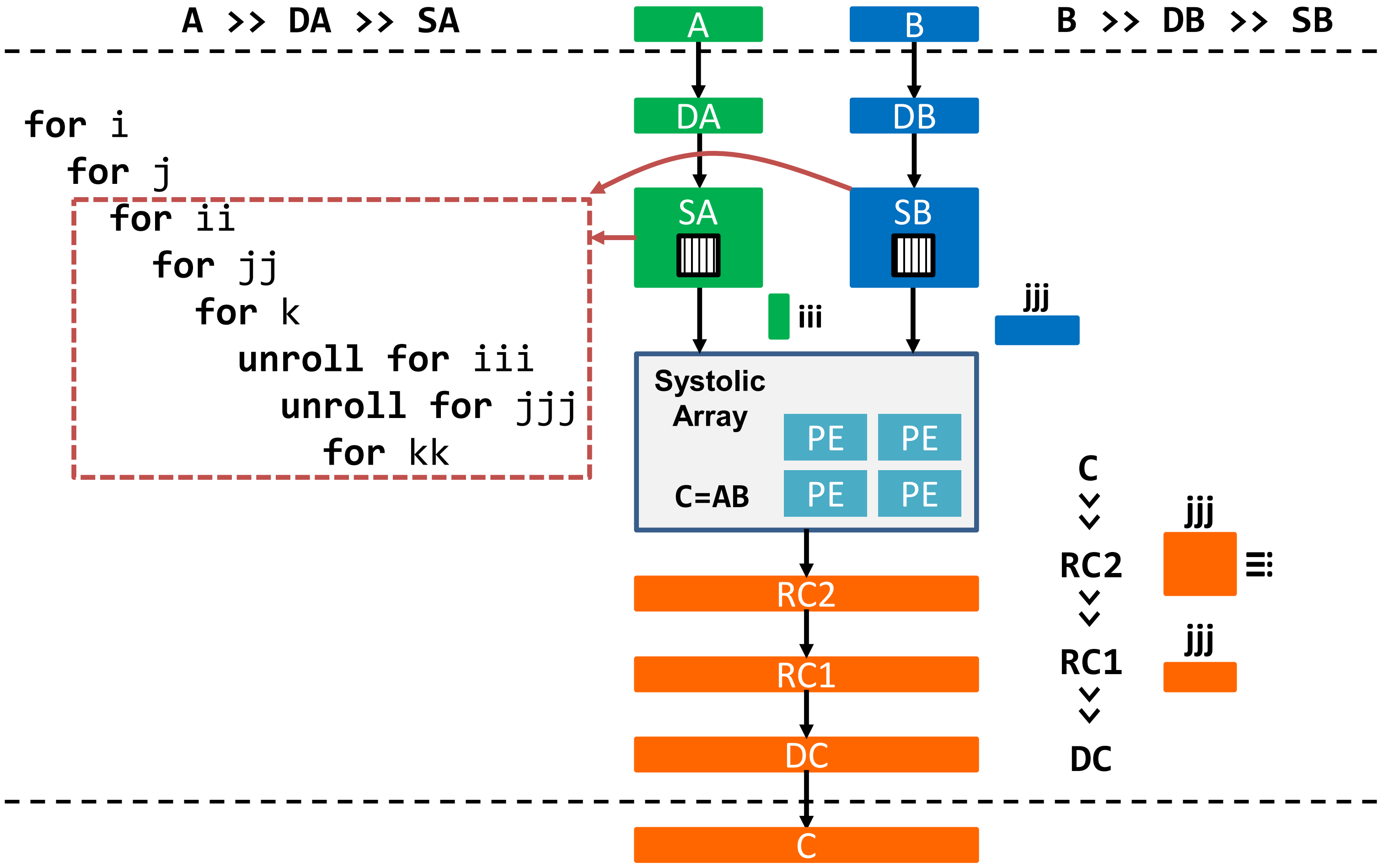
Stensor DC

Streaming Tensors

- **Abstract, self-managing, customizable storage of a tensor**
  - Memory Level
  - Data movements
  - Buffer insertion
  - Transfer unit

Specification (mm_IO)

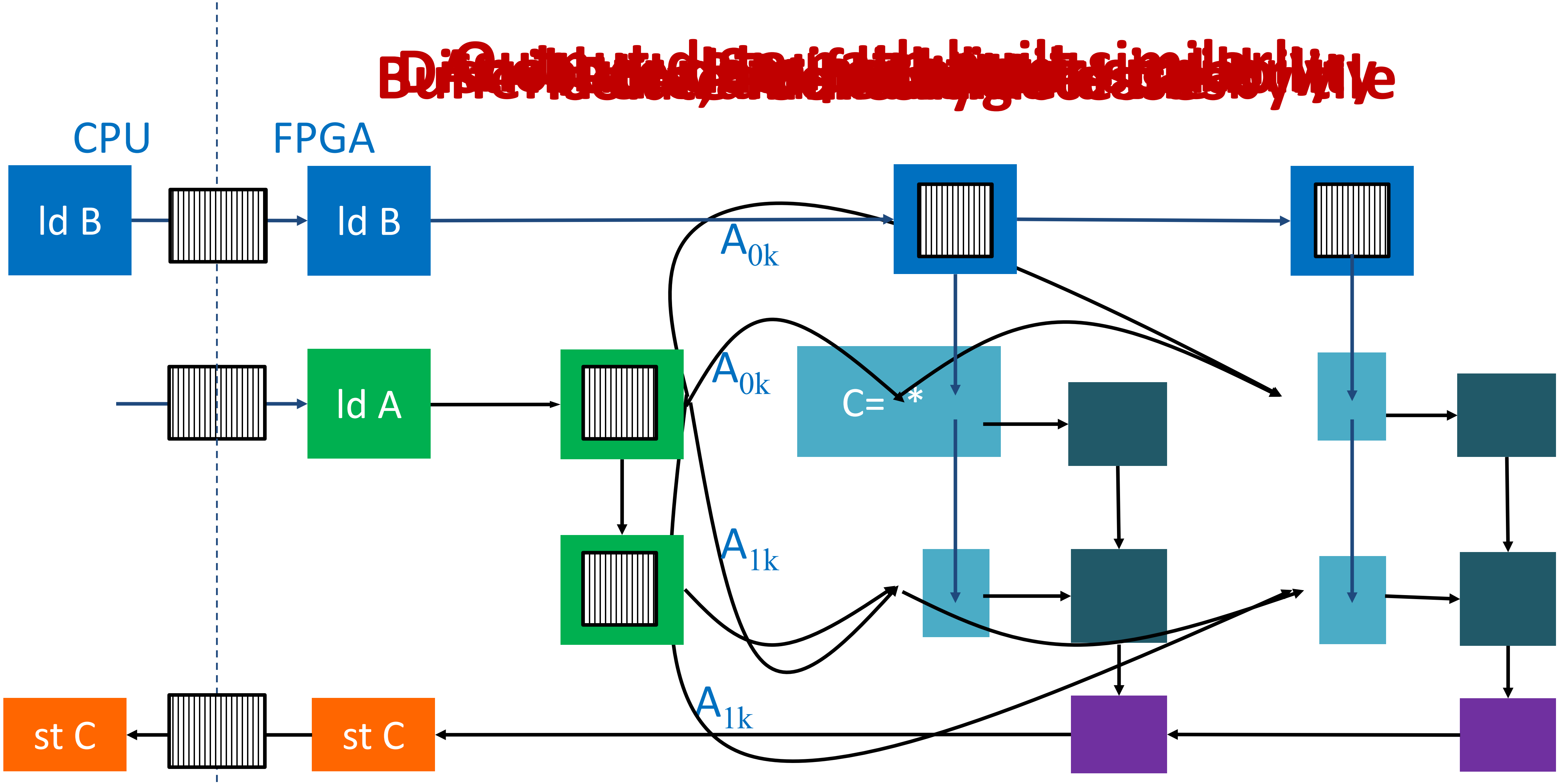
```
A(k, i) >> DA
        >> SA.scope(j).out(iii)
        >> X
B(j, k) >> DB
        >> SB.scope(j).out(jjj)
        >> Y
C >> RC2.out(jjj, iii)
  >> RC1.out(jjj)
  >> DC
```



```
for i
  for j
    for ii
      for jj
        for k
          unroll for iii
            unroll for jjj
              for kk
```

A systolic array with efficient I/O network

[illegible]



Memory Isolation

- **Isolating memory accesses into separate kernels**

```
for (SA.s0.i.i, 0, 4)
 …
  for (SA.s0.k.kk, 0, 4)
   DA(…) = select(…, A(k, i))
```

Initial IR for DA

```
for (SA.s0.i.i, 0, 4)
 …
  for (SA.s0.k.kk, 0, 4)
   SA(…) = select(…,  DA(…)  )
```

Initial IR for SA

```
for (X.s0.i.i, 0, 4)
 …
  for (X.s0.k.kk, 0, 4)
   X(…) = select(…,  SA(…)  , X(…))
```

Initial IR for UREs

Specification (mm_IO)

```
X(k, j, i) = select(j == 0, A(k, i), X(k, j-1, i));
…
A(k, i) >> DA >> SA >> X;
```

Buffer Insertion

Buffer Scope

```
for i
 for j
  for ii
   ~~for jj~~
    for k
     unroll for iii
      ~~unroll for jjj~~
       for kk
        DA(…) = A(kk+4*k,
                  iii+2*ii+4*i)
```

Producer (DA)

```
for i
 for j
  allocate Buf(…)
  for ii, k
    Buf(…) = DA(…)
  for ii, jj, kk
    unroll for iii
      SA(…) = DA(…)
```

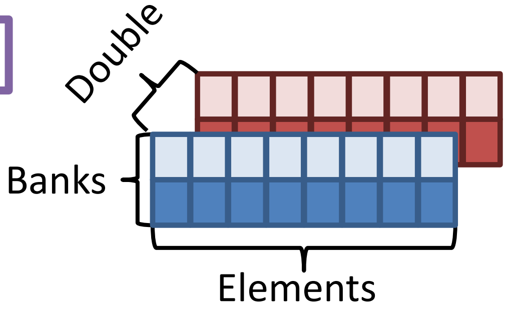
Buffer Allocation

Write only once

Read repeatedly

Consumer (SA)

```
Allocate
  Buf(2,
      #Elements,
      #Banks)
```

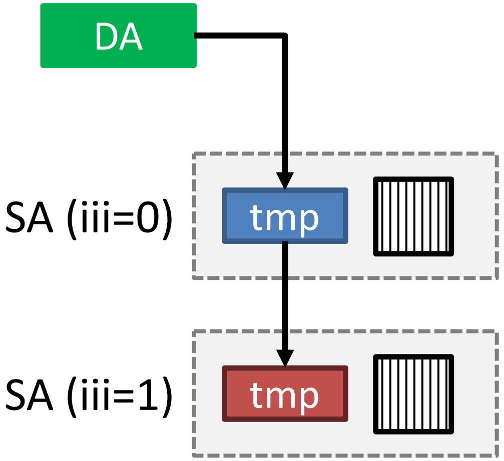


Specification (mm_IO)

```
A(k, i) >> DA >> SA.scope(j).out(iii) >> X
```

Data Scattering

- **Input data path**



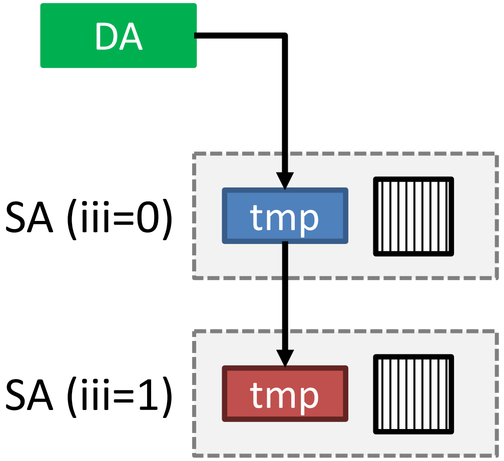
```
unroll for iii
 tmp(iii) =
  select(iii==0, DA(…),
                 tmp(iii-1))
 if current bank is active:
  SA(iii) = tmp(iii)
```

Specification (mm_IO)

```
A(k, i) >> DA >> SA.scope(j).out(iii)
```

Data Gathering

- **Input data path**

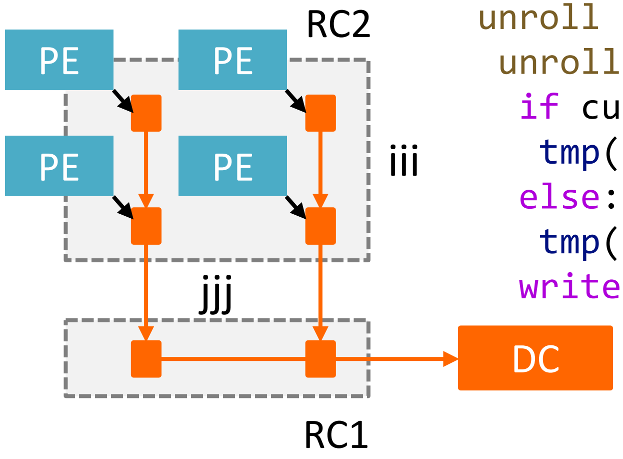


```
unroll for iii
 tmp(iii) =
  select(iii==0, DA(…),
                 tmp(iii-1))
 if current bank is active:
  SA(iii) = tmp(iii)
```

Specification (mm_IO)

```
A(k, i) >> DA >> SA.scope(j).out(iii)
```

- **Output data path**

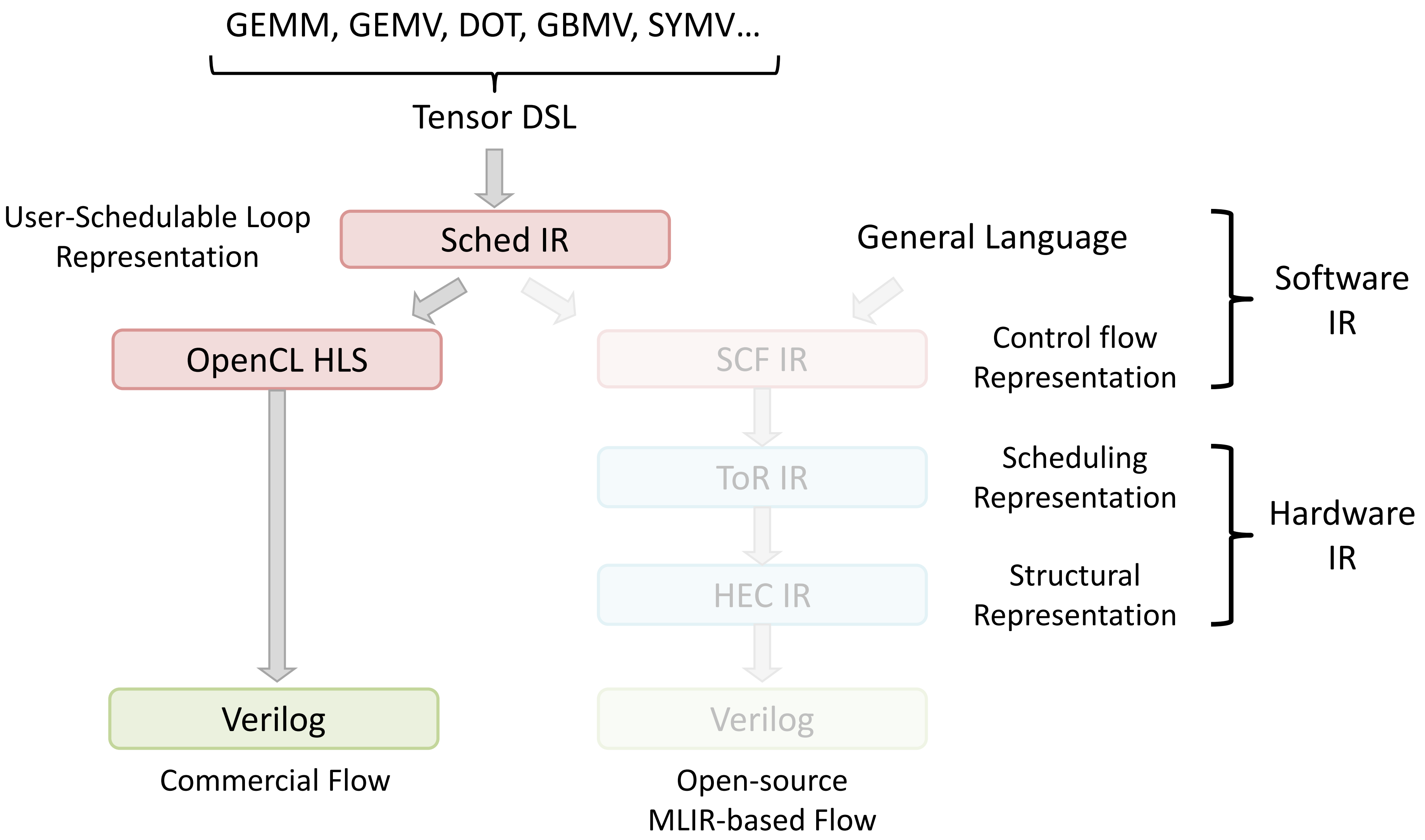


```
unroll for iii
 unroll for jjj
  if current row is active:
   tmp(iii, jjj) = C(…)
  else:
   tmp(iii, jjj) = tmp(iii-1,jjj)
  write tmp(III-1,jjj) into DC
```

Specification (mm_IO)

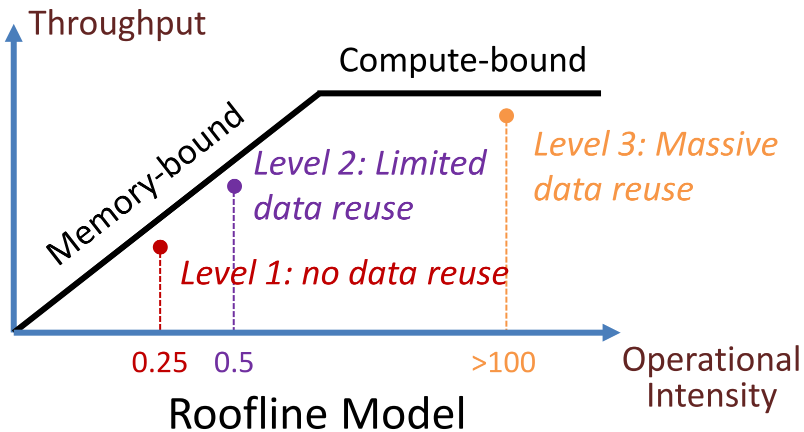
```
C >> RC2.out(jjj, iii) >> RC1.out(jjj) >> DC
```

Synthesis Flow


GEMM, GEMV, DOT, GBMV, SYMV…
Tensor DSL
User-Schedulable Loop Representation
Sched IR
General Language
Software IR
OpenCL HLS
SCF IR
Control flow Representation
Scheduling Representation
ToR IR
Hardware IR
HEC IR
Structural Representation
Verilog
Verilog
Commercial Flow
Open-source MLIR-based Flow

BLAS Library

- **BLAS：Basic Linear Algebra Subprograms**
  - **Level 1:** scalar, vector and vector-vector operations
  - **Level 2:** matrix-vector operations
  - **Level 3:** matrix-matrix operations
  - Either compute- or memory-bound



Roofline Model

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \quad * \quad y = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix} \quad = \quad \begin{bmatrix} 4 \end{bmatrix}$$

Level 1: DOT

$$\overset{A}{\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}} * \overset{x}{\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}} = \overset{y}{\begin{bmatrix} 4 \\ 6 \\ 4 \\ 6 \end{bmatrix}}$$

Level 2: GEMV

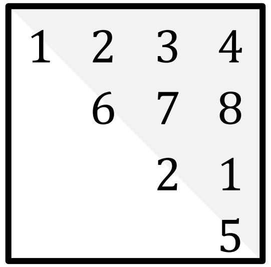
$$\overset{A}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}} * \overset{B}{\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}} = \overset{C}{\begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 1 & 3 \\ 2 & 4 \end{bmatrix}}$$

Level 3: GEMM

Solution

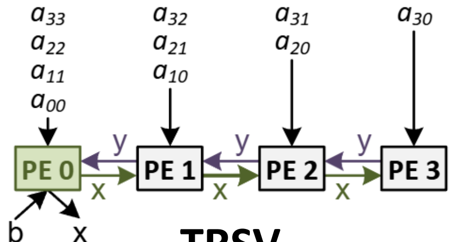
- **Using the DSL to develop BLAS routines**
  - Navigate to `popa/examples/BLAS`

- **Leveraging the triangularity, symmetry, and band structure of matrices**
  - **TRSV:** Triangular matrix-vector solve
  - **SYMV:** Symmetric matrix-vector multiplication
  - **GBMV:** Banded matrix-vector multiplication



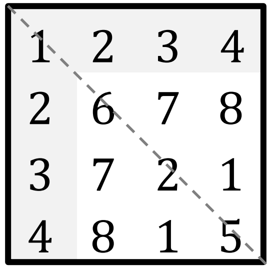
$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ & 6 & 7 & 8 \\ & & 2 & 1 \\ & & & 5 \end{array}$$

Triangular



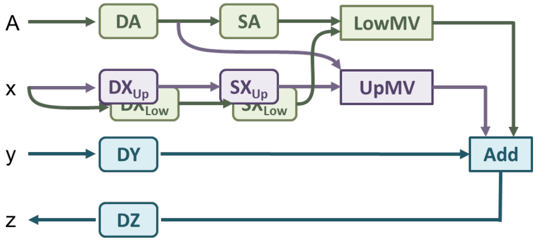
a_{33} a_{22} a_{11} a_{00}
a_{32} a_{21} a_{10}
a_{31} a_{20}
a_{30}
PE 0
PE 1
PE 2
PE 3
y
x
b
x

TRSV

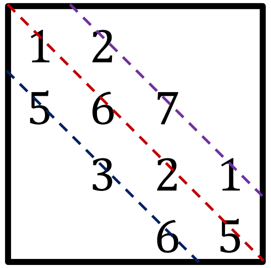


$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 6 & 7 & 8 \\ 3 & 7 & 2 & 1 \\ 4 & 8 & 1 & 5 \end{array}$$

Symmetric

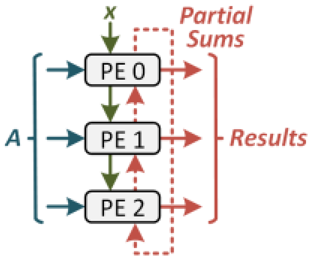


SYMV



$$\begin{array}{cccc} 1 & 2 & & \\ 5 & 6 & 7 & \\ & 3 & 2 & 1 \\ & & 6 & 5 \end{array}$$

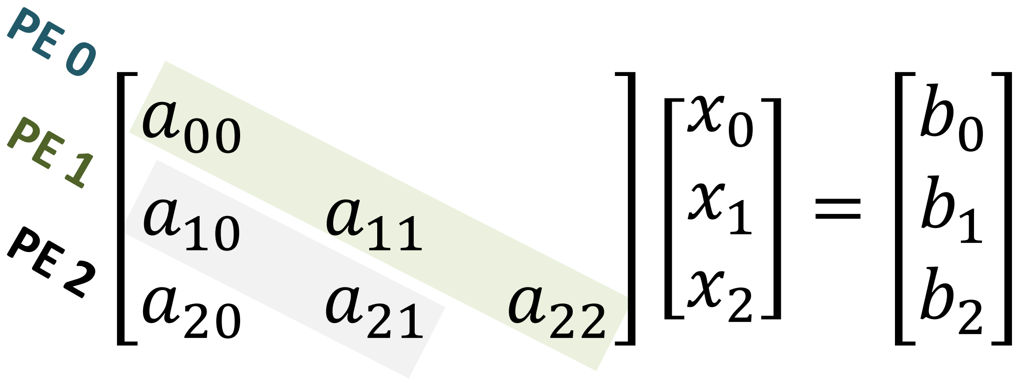
Banded



GBMV

Example 1: TRSV

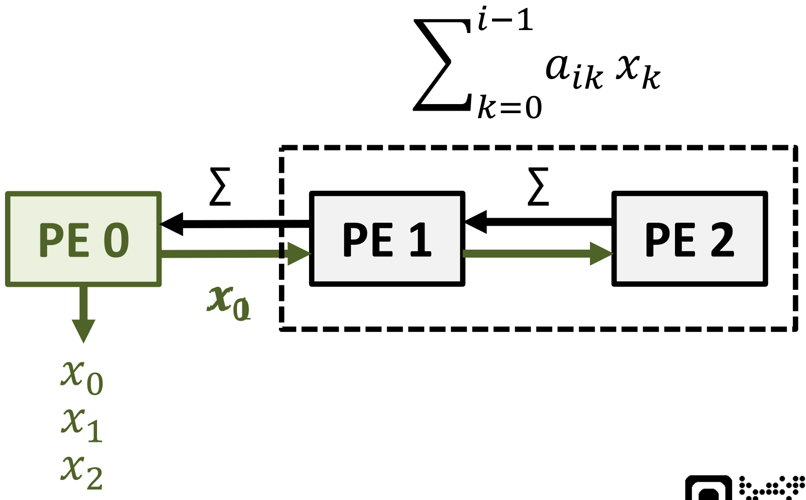
- **TRSV: Solve Ax=b where A is a triangular matrix**
  - A forward substitution process



$$\begin{bmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

$$x_0 = b_0 / a_{00}$$

$$x_i = (b_i - \sum_{k=0}^{i-1} a_{ik} \, x_k) / a_{ii}$$

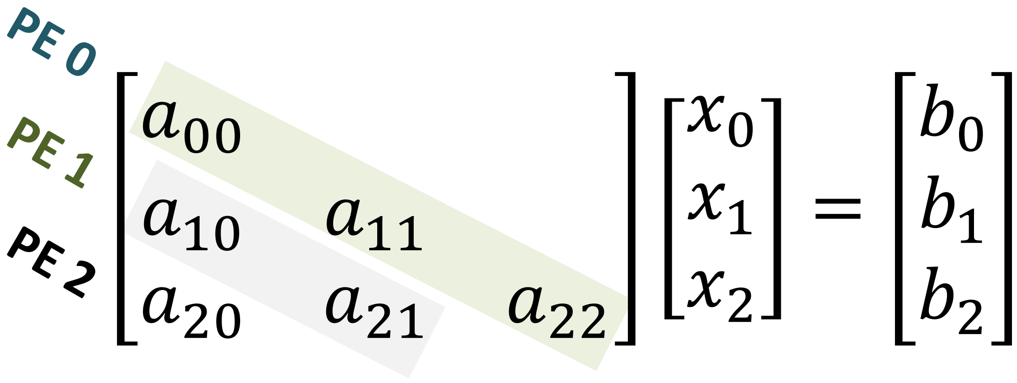


Our recent work - A dataflow accelerator for SpTRSV that exploits structural sparsity patterns in solving PDEs (ISCA'25)

github.com/pku-liang/telos

Example 1: TRSV

- **TRSV: Solve Ax=b where A is a triangular matrix**
  - A forward substitution process



$$\begin{bmatrix} a_{00} & & \\ a_{10} & a_{11} & \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

$$x_0 = b_0 / a_{00}$$

$$x_i = (b_i - \sum_{k=0}^{i-1} a_{ik}\, x_k) / a_{ii}$$

Express the dataflow using URE and STT

```
uA(i, k) = A(i, k)
fX(i, k) = select(i==k,
             (b(k) - select(k==0, 0, fY(i, k-1)))/uA(i, k),
             fX(i-1, k))
fY(i, k) = select(k==0, 0, fY(i, k-1))
             + uA(i, k)*fX(i, k)
Out(k) = select(i==k, fX(i, k))

uA.space_time_transform({i, k},
                        {s, t},
                        {1, -1,
                         0, 1})
```

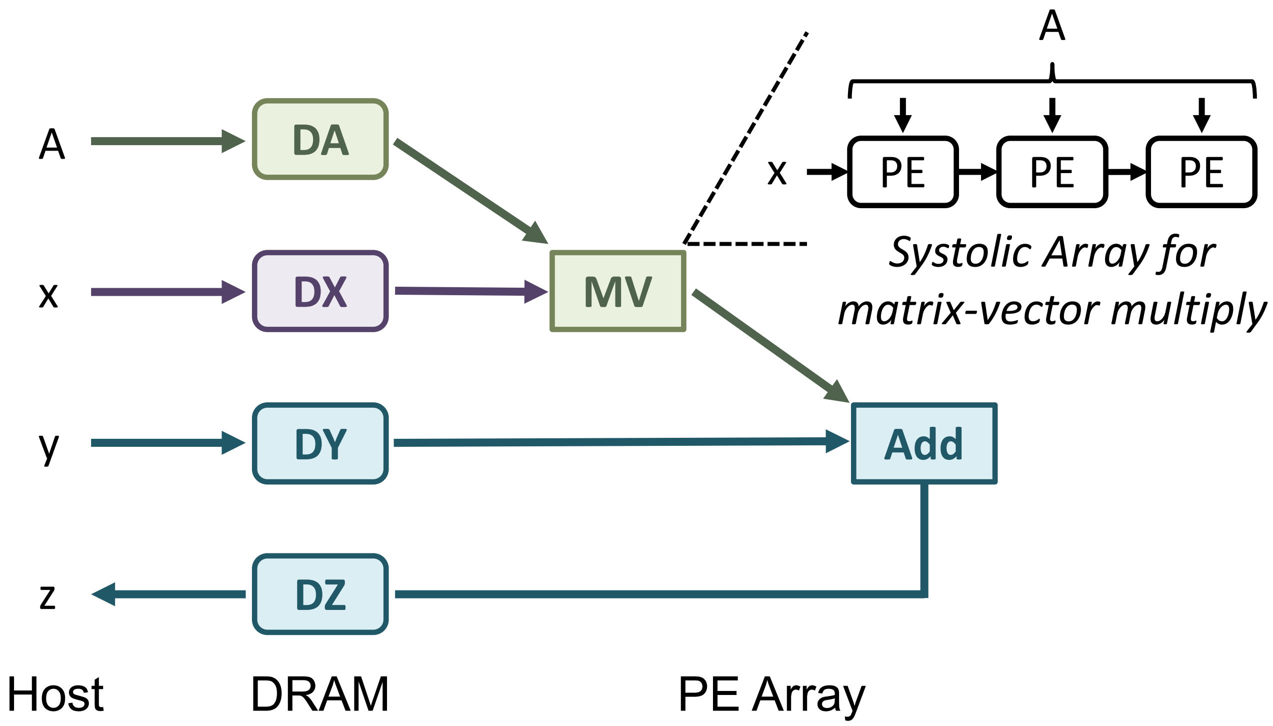
Compute or forward x

Forward partial sum $y = \sum a_{ik} x_k$

Space loop: $s = i - k$
Time loop: $t = k$

Example 2: SYMV

- **A vanilla design**
  - Cannot take advantage of the symmetry to save bandwidth

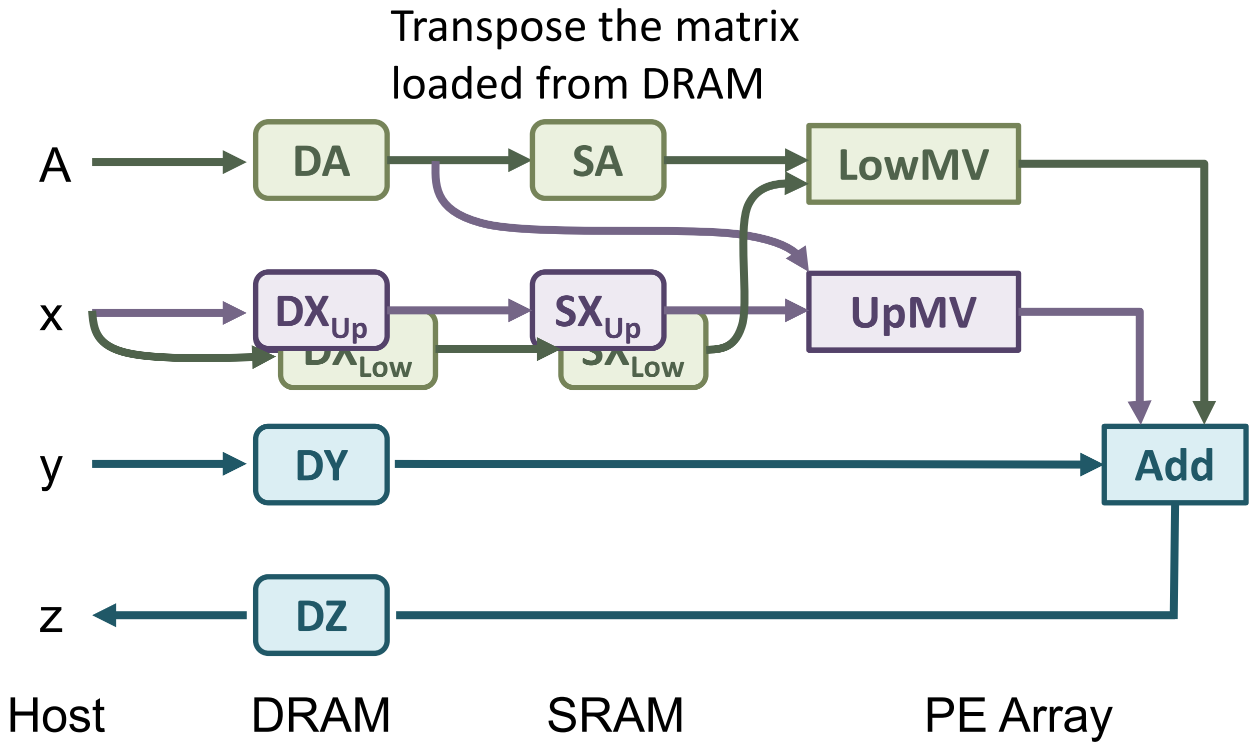


Systolic Array for matrix-vector multiply

- **SYMV:** $z_i = \alpha \sum_{k=0}^{N-1} A_{i,k} x_k + \beta y_i$, A is a symmetric matrix

Example 2: SYMV

- Two systolic arrays to compute lower and upper triangles
- Load the upper triangle from DRAM and generate the lower triangle on the fly in SRAM
- Load the corresponding x values from the reuse buffers



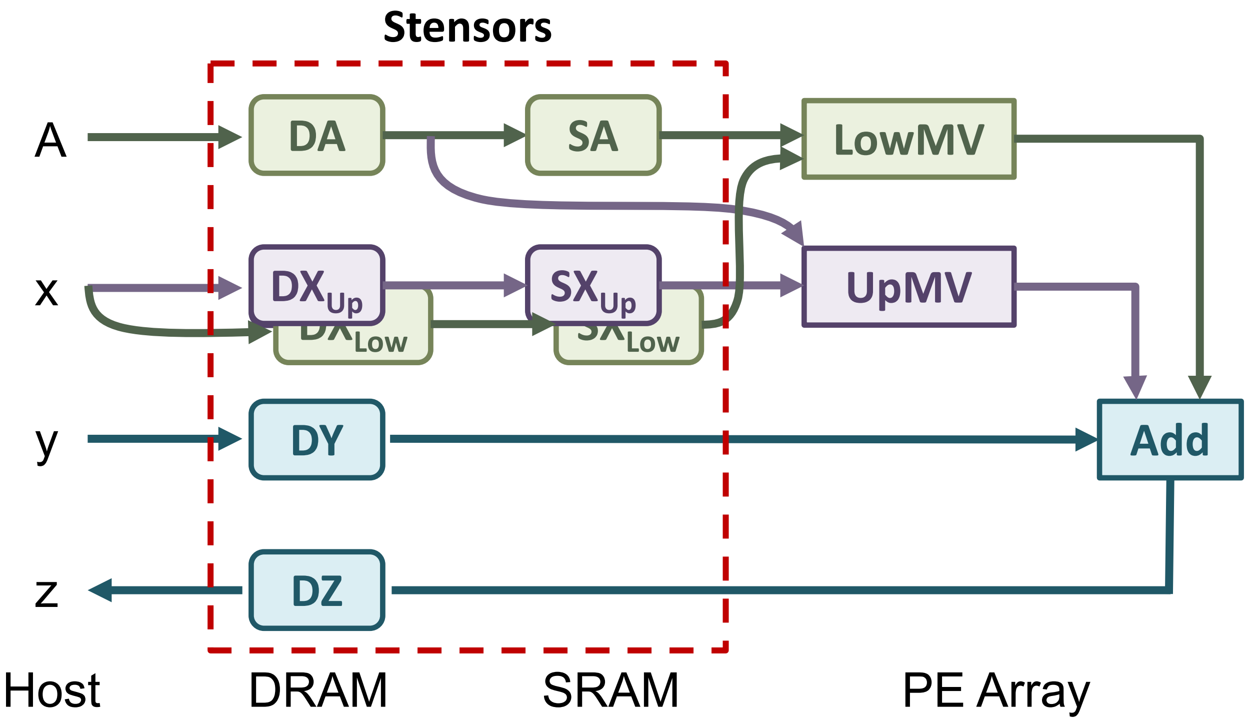
Transpose the matrix loaded from DRAM

- **SYMV:** $z_i = \alpha \sum_{k=0}^{N-1} A_{i,k} x_k + \beta y_i$, A is a symmetric matrix

Example 2: SYMV

Memory schedule with stensors:

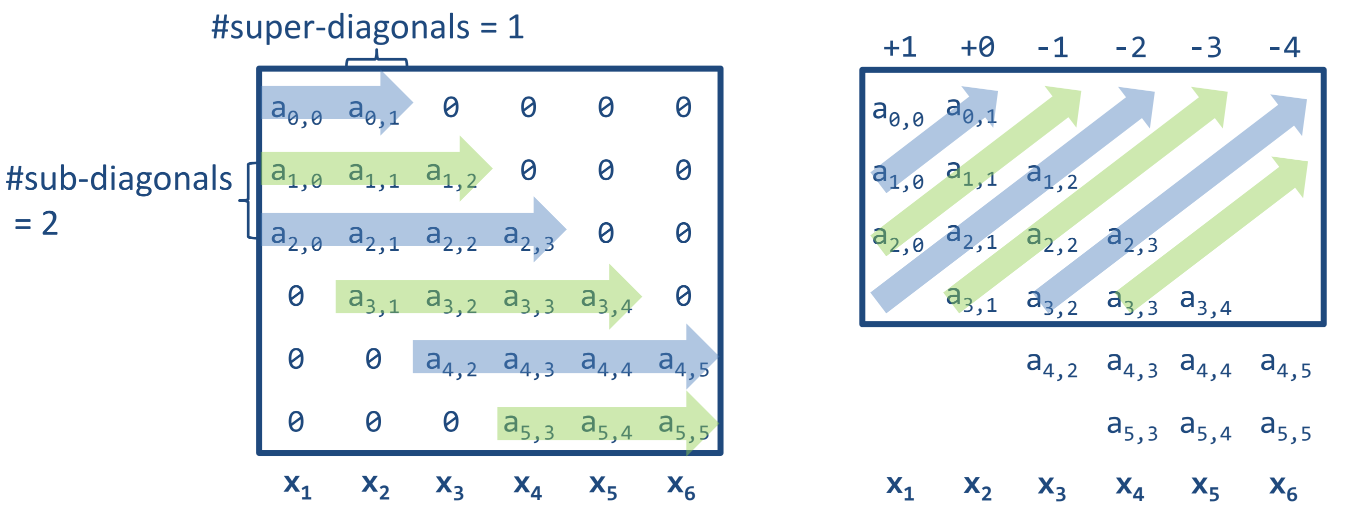
```
A >> DA >> {UpMV, SA};
SA.transpose() >> LowMV;
x >> {DX_Up, DX_Low};
DX_Up >> SX_Up >> UpMV;
DX_Low >> SX_Low >> LowMV;
y >> DY >> Add;
Add >> DZ >> z;
```



- **SYMV:** $z_i = \alpha \sum_{k=0}^{N-1} A_{i,k} x_k + \beta y_i$, A is a symmetric matrix

Example 3: GBMV

- **Customized dataflow for special computation patterns**
  - **GBMV:** $z = \alpha Ax + \beta y$, where A is a banded matrix
  - Utilizing matrix sparsity through banded storage format



General storage of matrix A

#super-diagonals = 1

#sub-diagonals = 2

$a_{0,0}$	$a_{0,1}$	0	0	0	0
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	0	0	0
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	0	0
0	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	0
0	0	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$
0	0	0	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$
x_1	x_2	x_3	x_4	x_5	x_6

Band storage of matrix A

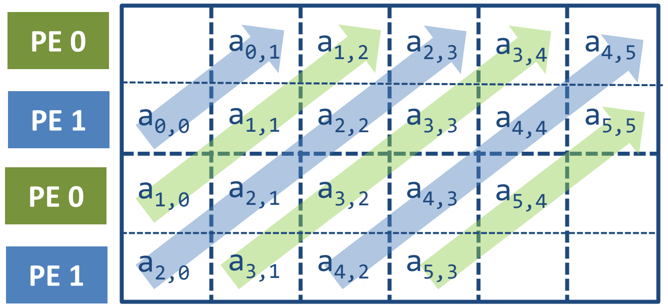
+1	+0	-1	-2	-3	-4
$a_{0,0}$	$a_{0,1}$				
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$			
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		
	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	
		$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$
			$a_{5,3}$	$a_{5,4}$	$a_{5,5}$
x_1	x_2	x_3	x_4	x_5	x_6

- **General storage → Band storage**
  - Move column j down by $\#superdiagnoals - j$ rows
- The values are reduced along a **row** in the general storage, but along a **diagonal** in the band storage

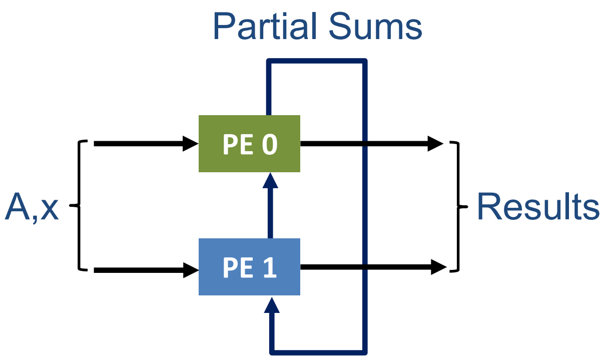
Example 3: GBMV

- **Customized dataflow for special computation patterns**
  - **GBMV:** $z = \alpha A x + \beta y$, where A is a banded matrix
  - Utilizing matrix sparsity through banded storage format

Band storage of matrix A



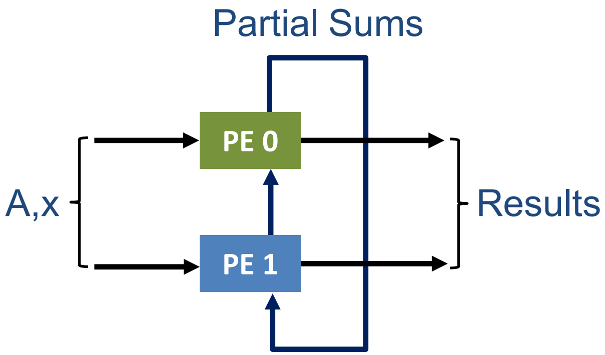
PE 0		$a_{0,1}$	$a_{1,2}$	$a_{2,3}$	$a_{3,4}$	$a_{4,5}$
PE 1	$a_{0,0}$	$a_{1,1}$	$a_{2,2}$	$a_{3,3}$	$a_{4,4}$	$a_{5,5}$
PE 0	$a_{1,0}$	$a_{2,1}$	$a_{3,2}$	$a_{4,3}$	$a_{5,4}$	
PE 1	$a_{2,0}$	$a_{3,1}$	$a_{4,2}$	$a_{5,3}$		



- **Tile the band storage**
  - Each tile has multiple rows, 1 column
  - Iterate the tiles top down, and from left to right
  - One PE is responsible for one row
- **Data dependencies are mapped to cyclically connected PEs.**

Example 3: GBMV

- **Customized dataflow for special computation patterns**
  - **GBMV:** $z = \alpha Ax + \beta y$, where A is a banded matrix
  - Utilizing matrix sparsity through banded storage format



- Expressing dataflow using URE and STT

```
fX(k, i) = select(i == 0, x(k), fX(k, i-1))
MV(k, i) = select(k == 0 || i == I-1, 0,
                  MV(k-1, i+1)
                 ) + A(k, i)*fX(k, i)
Tout(k) = select(i == 0, MV(k, i))
Rout(i) = select(k == K-1, MV(k, i))
fX.merge_ures(MV, Tout, Rout)
fX.space_time_transform(i)
```

Reuse x_k

Reduce along anti-diagonals

Collect results along boundaries

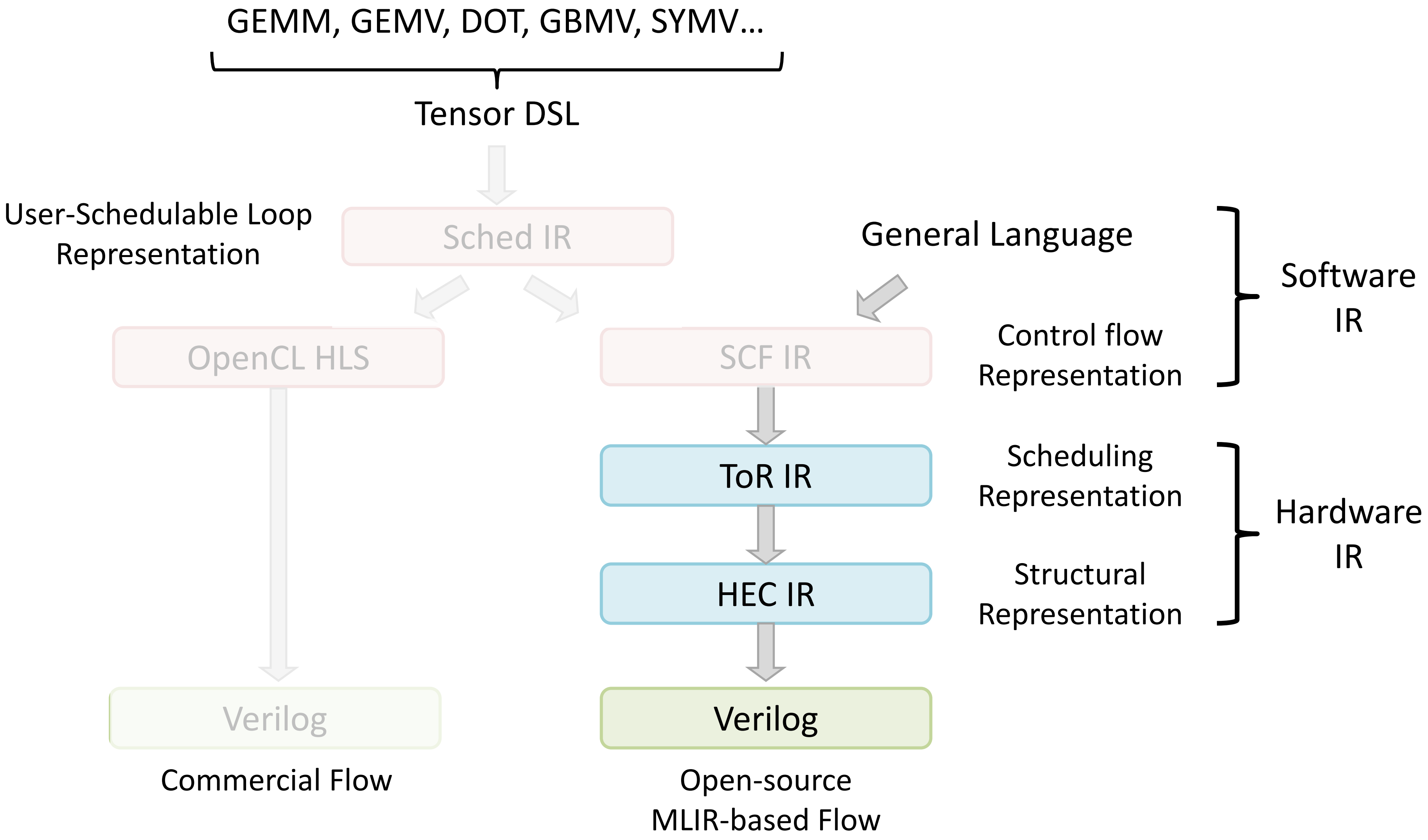
Map rows to PEs

Results

Level	Kernel	Name	Compute	Frequency	LUTs	DSPs	Throughputs	Speedup
Level 3	GEMM	matrix-matrix multiply	$C = \alpha AB + \beta C$	244 Mhz	49%	86%	620 GFlops	-
	SYMM	symmetric matrix-matrix multiply	$C = \alpha AB + \beta C, A = A^T$	244 Mhz	49%	86%	620 GFlops	-
	SYRK	symmetric rank-k update to a matrix	$C = \alpha AA^T + \beta C$	259 Mhz	43%	68%	513 GFlops	1.93X
	SYR2K	symmetric rank-2k update to a matrix	$C = \alpha AB^T + \alpha BA^T + \beta C$	253 Mhz	48%	68%	476 GFlops	1.81X
	TRSM	Solves a triangular matrix equation	Solve $Ax = B$	239 Mhz	51%	72%	402 GFlops	-
	TRMM	triangular matrix-matrix multiply	$B = \alpha AB$	238 Mhz	44%	68%	471 GFlops	1.93X
Level 2	GEMV	matrix-vector multiply	$y = \alpha Ax + \beta y$	282 Mhz	20%	2%	16 GFlops	-
	GBMV	banded matrix-vector multiply	$y = \alpha Ax + \beta y$	277 Mhz	21%	2%	16 GFlops	7.35X
	SYMV	symmetric matrix-vector multiply	$y = \alpha Ax + \beta y$	267 Mhz	39%	4%	15 GFlops	1.79X
	TRMV	triangular matrix-vector multiply	$x = Ax$	254 Mhz	23%	2%	15 GFlops	1.75X
	TRSV	Solves a triangular matrix equation	Solve $Ax = B$	303 Mhz	18%	2%	16 GFlops	-

- Synthesized using Intel OpenCL SDK, tested on Intel A10 FPGA
- 1.8X-7.4X speedup by leveraging special matrix properties

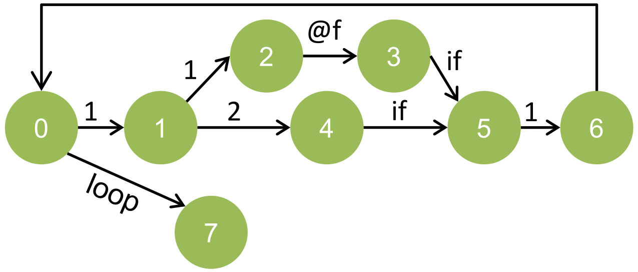
Synthesis Flow


GEMM, GEMV, DOT, GBMV, SYMV…
Tensor DSL
User-Schedulable Loop Representation
Sched IR
General Language
Software IR
OpenCL HLS
SCF IR
Control flow Representation
ToR IR
Scheduling Representation
Hardware IR
HEC IR
Structural Representation
Verilog
Verilog
Commercial Flow
Open-source MLIR-based Flow

ToR IR

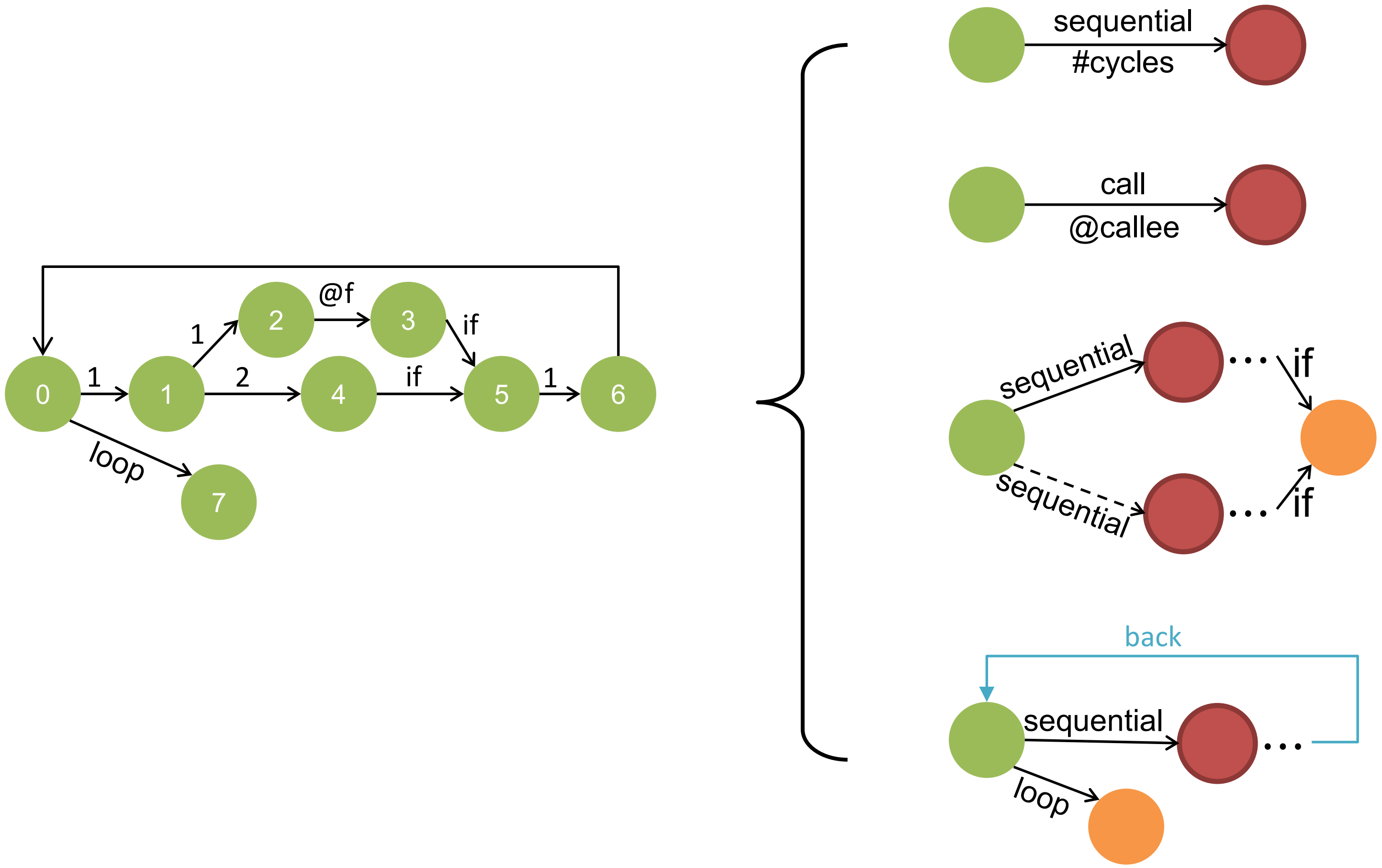
- **Provide a high-level control flow for hardware information**
  - Two parts, time graph and functional operations

```
tor.topo (0 to 7) {
  tor.from 0 to 1 "seq:1"
  tor.from 1 to 2 "seq:1"
  tor.from 2 to 3 "call"
  tor.from 1 to 4 "seq:2"
  tor.from 3, 4 to 5 "if"
  tor.from 5 to 6 "seq:1"
  tor.from 0 to 7 "for"
}
```



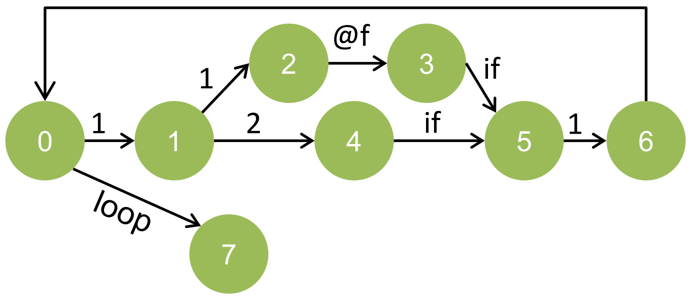
Time Graph

- **Four types of nodes including normal, call, if and loop**



Functional operations

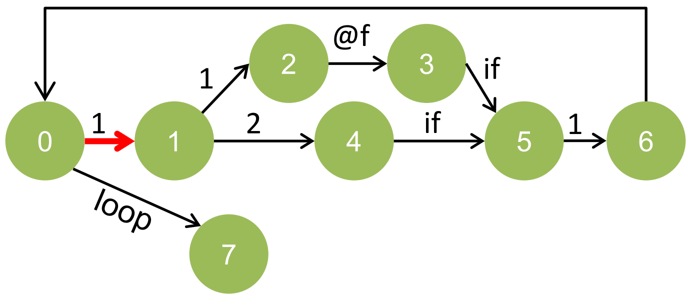
- **Algorithmic specification with high-level information**
  - Bind each operation to the element of the time graph



```
tor.for %i = %c0 to %c10 step %c1 {
  %m = tor.load %mask[%i] on (0 to 1)
  %a = tor.if %m then {
    %x = tor.addi %i %c1 on (1 to 2)
    %y = tor.subi %i %c1 on (1 to 2)
    %fx = tor.call @f(%x, %y) on (2 to 3)
    tor.yield %fx
  } else {
    %ii = tor.muli %i %i on (1 to 4)
    tor.yield %ii
  } on (1 to 5)
  tor.store %a to %A[%i] on (5 to 6)
} on (0 to 7)
```

Functional operations

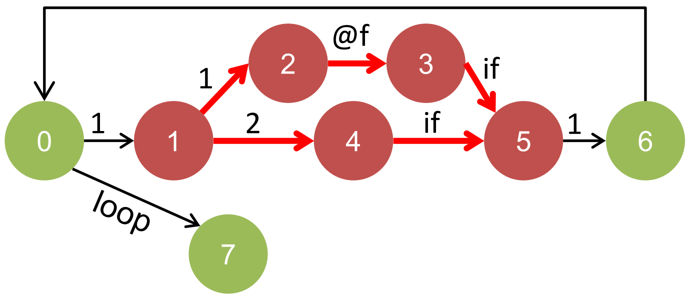
- **Algorithmic specification with high-level information**
  - Bind each operation to the element of the time graph



```
tor.for %i = %c0 to %c10 step %c1 {
  %m = tor.load %mask[%i] on (0 to 1)
  %a = tor.if %m then {
    %x = tor.addi %i %c1 on (1 to 2)
    %y = tor.subi %i %c1 on (1 to 2)
    %fx = tor.call @f(%x, %y) on (2 to 3)
    tor.yield %fx
  } else {
    %ii = tor.muli %i %i on (1 to 4)
    tor.yield %ii
  } on (1 to 5)
  tor.store %a to %A[%i] on (5 to 6)
} on (0 to 7)
```

Functional operations

- **Algorithmic specification with high-level information**
  - Bind each operation to the element of the time graph



```
tor.for %i = %c0 to %c10 step %c1 {
  %m = tor.load %mask[%i] on (0 to 1)
  %a = tor.if %m then {
    %x = tor.addi %i %c1 on (1 to 2)
    %y = tor.subi %i %c1 on (1 to 2)
    %fx = tor.call @f(%x, %y) on (2 to 3)
    tor.yield %fx
  } else {
    %ii = tor.muli %i %i on (1 to 4)
    tor.yield %ii
  } on (1 to 5)
  tor.store %a to %A[%i] on (5 to 6)
} on (0 to 7)
```

Supplementary information

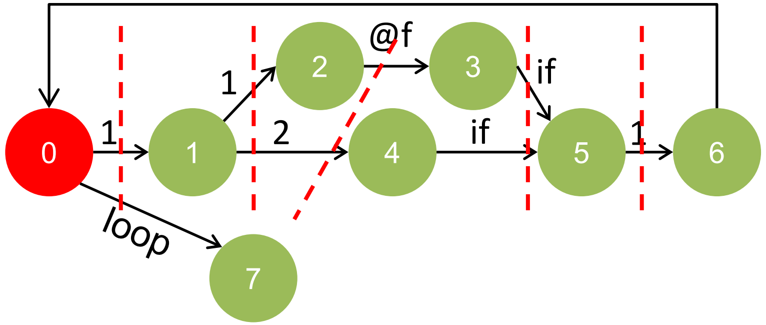
- Timing information like latency is attached to time graph

```
tor.topo (0 to 7) {
  tor.from 0 to 1 "seq:1"
  tor.from 1 to 2 "seq:1"
  tor.from 2 to 3 "call"
  tor.from 1 to 4 "seq:2"
  tor.from 3, 4 to 5 "if"
  tor.from 5 to 6 "seq:1"
  tor.from 0 to 7 "for"
}
```

Supplementary information

- **Timing information like latency is attached to time graph**
- Three scheduling manners are supported in ToR IR
  - Static, pipeline and dynamic

```
tor.topo (0 to 7) {
  tor.from 0 to 1 "seq:1"
  tor.from 1 to 2 "seq:1"
  tor.from 2 to 3 "call"
  tor.from 1 to 4 "seq:2"
  tor.from 3, 4 to 5 "if"
  tor.from 5 to 6 "seq:1"
  tor.from 0 to 7 "pipeline-for:1"
}
```

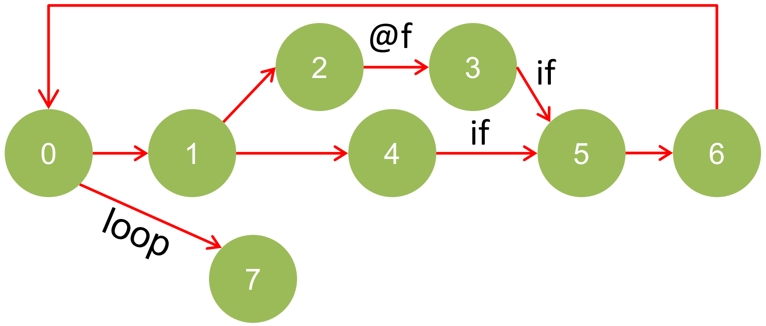


```
tor.for {...} on (0 to 7) {pipeline, II = 1}
```

Supplementary information

- **Timing information like latency is attached to time graph**
- **Three scheduling manners are supported in ToR IR**
  - Static, pipeline and dynamic

```
tor.topo (0 to 7) {
  tor.from 0 to 1 "dynamic"
  tor.from 1 to 2 "dynamic"
  tor.from 2 to 3 "dynamic-call"
  tor.from 1 to 4 "dynamic"
  tor.from 3, 4 to 5 "dynamic-if"
  tor.from 5 to 6 "dynamic"
  tor.from 0 to 7 "dynamic-for"
}
```



Quick Try

- **Run:**

```
cd hector
bash examples/hls_script.sh hector examples
```

- **Then you will get several directories:**

```
examples/tor-raw/
examples/tor-split/
examples/hec/
examples/chisel/
```

SCF to TOR Compilation

- cat examples/tor-raw/gemm_ncubed_kernel.mlir

```
tor.timegraph (0 to 15){
  tor.succ 1 : [0 : i32] [{type = “static”}]
  tor.succ 2 : [1 : i32] [{type = "static"}]
  tor.succ 3 : [2 : i32] [{type = "static"}]
  tor.succ 4 : [3 : i32] [{type = "static:1"}]
  tor.succ 5 : [4 : i32] [{type = "static"}]
  tor.succ 6 : [5 : i32] [{type = "static"}]
  tor.succ 7 : [6 : i32] [{type = "static:1"}]
  tor.succ 8 : [7 : i32] [{type = "static:1"}]
  tor.succ 9 : [8 : i32] [{type = "static:9"}]
  tor.succ 10 : [9 : i32] [{type = "static:13"}]
  tor.succ 11 : [5 : i32] [{II = 14 : i32, pipeline = 1 : i32, type = "static-for"}]
  tor.succ 12 : [11 : i32] [{type = "static"}]
  tor.succ 13 : [12 : i32] [{type = "static:1"}]
  tor.succ 14 : [2 : i32] [{type = "static-for"}]
  tor.succ 15 : [1 : i32] [{type = "static-for"}]
}
```

SCF to TOR Compilation

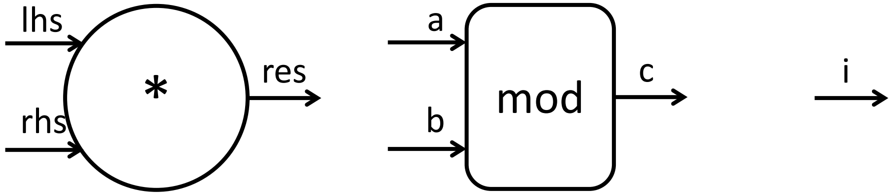
```
tor.for %arg0 = (%c0_i32 : i32) to (%c62_i32 : i32) step (%c1_i32 : i32)
    on (1 to 14) {
  tor.for %arg1 = (%c0_i32 : i32) to (%c62_i32 : i32) step (%c1_i32 : i32)
      on (2 to 13) {
    %3 = arith.shli %arg0, %c6_i32 {endtime = 4 : i32, starttime = 3 : i32} : i32
    %4 = tor.for %arg2 = (%c0_i32 : i32) to (%c62_i32 : i32) step (%c1_i32 : i32)
        on (5 to 10) iter_args(%arg3 = %cst) -> (f64) {
      %6 = arith.shli %arg2, %c6_i32 {endtime = 7 : i32, starttime = 6 : i32} : i32
      %7 = tor.addi %3 %arg2 on (6 to 7) : (i32, i32) -> i32
      %8 = tor.load %0[%7] on (7 to 8) : !tor.memref<4096xf64, [], "r">[i32]
      %9 = tor.addi %6 %arg1 on (6 to 7) : (i32, i32) -> i32
      %10 = tor.load %1[%9] on (7 to 8) : !tor.memref<4096xf64, [], "r">[i32]
      %11 = tor.mulf %8 %10 on (8 to 9) : f64
      %12 = tor.addf %arg3 %11 on (9 to 10) : f64
      tor.yield %12 : f64
    } {II = 14 : i32, pipeline = 1 : i32}
    %5 = tor.addi %3 %arg1 on (12 to 13) : (i32, i32) -> i32
    tor.store %4 to %2[%5] on (12 to 13) : (f64, !tor.memref<4096xf64, [], "w">[i32])
  }
}
```

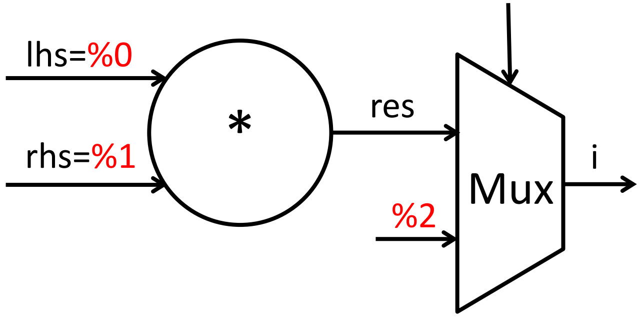
HEC IR

- **Adopt an allocate-assign mechanism**

```
%m.lhs, rhs, res = primitive "mul_integer"
%sub.a, b, c = instance "sub" @mod
%i = wire "i"
```

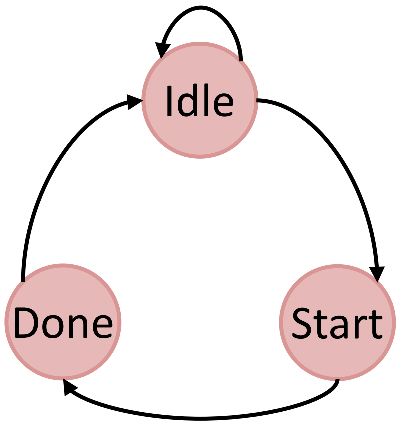
```
hec.assign %m.lhs = %0
hec.assign %m.rhs = %1
//Case #1:
    hec.assign %i = %m.res
//Case #2:
    hec.assign %i = %2
```





HEC IR

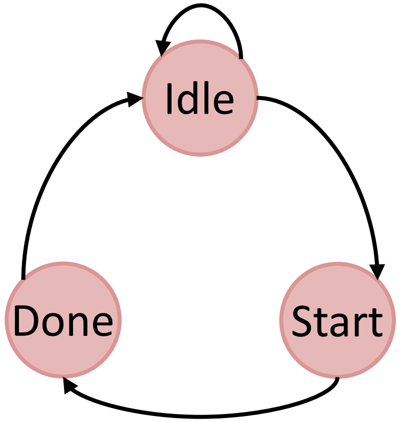
- **Three component styles corresponding to ToR**
  - State Transition Graph, Pipeline stages, Handshake



```
component @STG {
  // allocations
  stateset {
    state @Idle {
      // assigns
      transition {
        goto @Start if %cond
      }
    }
    state @Start {
      transition {
        goto @Done
      }
    }
    state @Done {
      transition {
        goto @Idle
      }
    }
  }
} {style = "stg"}
```

HEC IR

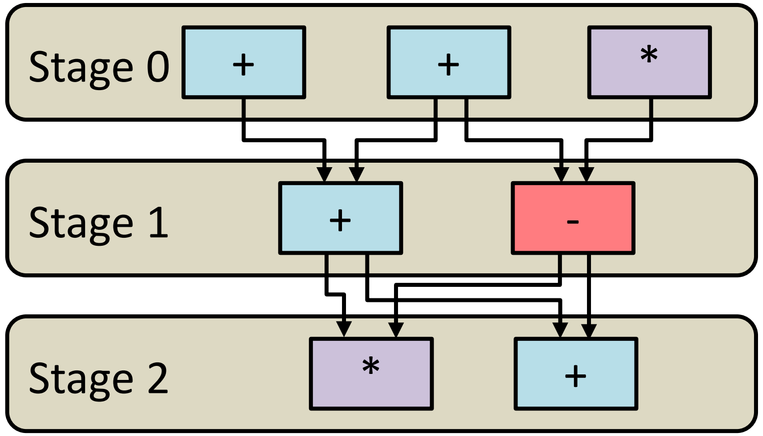
- **Three component styles corresponding to ToR**
  - State Transition Graph, Pipeline stages, Handshake



```
component @STG {
  // allocations
  stateset {
    state @Idle {
      // assigns
      transition {
        goto @Start if %cond
      }
    }
    state @Start {
      transition {
        goto @Done
      }
    }
    state @Done {
      transition {
        goto @Idle
      }
    }
  }
} {style = "stg"}
```

HEC IR

- **Three component styles corresponding to ToR**
  - State Transition Graph, Pipeline stages, Handshake

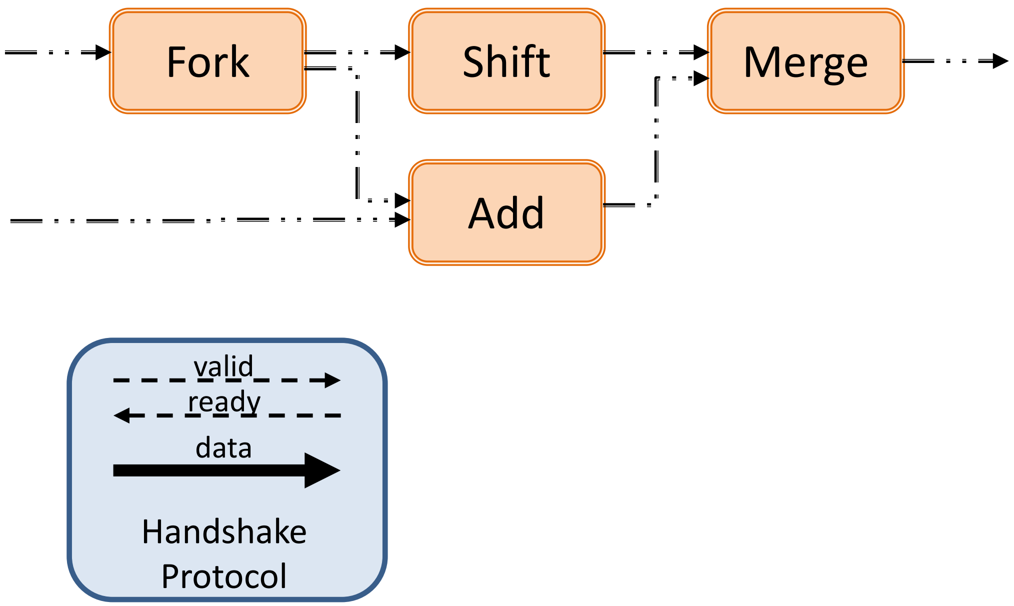


```
component @Pipe {
  // allocations
  stageset {
    stage @stage0 {
      // assigns
    }
    stage @stage1 {
      // assigns
      hec.assign %add2.lhs = %add0.res
      hec.assign %add2.rhs = %add1.res
    }
    stage @stage2 {
      // assigns
    }
    stage @stage3 {
      // assigns
      deliver %mul1.res, %add3.res
    }
  }
} {"pipeline",II=1}
```

HEC IR

- **Three component styles corresponding to ToR**
  - State Transition Graph, Pipeline stages, Handshake

Latency-insensitive Design



```
component @Handshake (%in1,%in2,%out){
  // allocations
  %shift.in,res = primitive . "shift"
  %add.lhs,rhs,res = primitive . "addi"
  // elastic units
  %merge.i1,i2,o = primitive . "merge"
  %fork.i,o1,o2 = primitive . "fork"

  graph {
    assign %fork.i = %in1
    assign %shift.in = %fork.o1
    assign %add.lhs = %in2
    assign %add.rhs = %f.o2
    assign %merge.i1 = %shift.res
    assign %merge.i2 = %add.res
    assign %out  = %merge.o
  }
}{"handshake"}
```

TOR to HEC Compilation

- cat examples/hec/gemm_ncubed_kernel.mlir

```
hec.stage @s1{
  %1 = hec.shift_left %r_5_i_1.reg %c6_i32 {dump = "comb_0"} : i32
  hec.assign %r_8_2.reg = %1 : i32 -> i32
  %2 = hec.addi %r_3_1.reg %r_5_i_1.reg {dump = "comb_1"} : (i32, i32) -> i32
  hec.assign %r_9_2.reg = %2 : i32 -> i32
  %3 = hec.addi %1 %r_4_1.reg {dump = "comb_2"} : (i32, i32) -> i32
  hec.assign %r_10_2.reg = %3 : i32 -> i32
  hec.assign %r_5_i_2.reg = %r_5_i_1.reg : i32 -> i32
  hec.assign %r_6_2.reg = %r_6_1.reg : f64 -> f64
}
hec.stage @s2{
  hec.assign %mem_global_0.addr = %r_9_2.reg : i32 -> i12
  hec.enable %mem_global_0.r_en : i1
  hec.assign %mem_global_1.addr = %r_10_2.reg : i32 -> i12
  hec.enable %mem_global_1.r_en : i1
  hec.assign %r_5_i_3.reg = %r_5_i_2.reg : i32 -> i32
  hec.assign %r_6_3.reg = %r_6_2.reg : f64 -> f64
  hec.assign %r_9_3.reg = %r_9_2.reg : i32 -> i32
  hec.assign %r_10_3.reg = %r_10_2.reg : i32 -> i32
}
```

TOR to HEC Compilation

- cat examples/hec/gemm_ncubed_kernel.mlir

```
hec.component @outline_0(%arg0: i32,%arg1: i32,%arg2: i32,%arg3: i32,%arg4: i32,%arg5:
i1) -> (%arg6: f64,%arg7: i1)
{interface="naked", style="pipeline"}{
  %r_13_13.reg = hec.primitive "r_13_13" is "register" : f64
  %mulf_outline_0_0.operand0, operand1, result = hec.primitive
    "mulf_outline_0_0" is "mul_float" : f64, f64, f64
}

hec.component @main(%arg0: i1) -> (%arg1: i1)
    {interface="naked", style="STG"}{
  %r_main_0.reg = hec.primitive “r_main_0” is “register” : i1
  %r_main_1.reg = hec.primitive "r_main_1" is "register" : i1
  %r_main_2.reg = hec.primitive "r_main_2" is "register" : i32
  %r_main_3.reg = hec.primitive "r_main_3" is "register" : i32
  %outline_0_0.in0, in1, in2, in3, in4, go, out0, done = hec.instance
    "outline_0_0" @outline_0 : i32, i32, i32, i32, i32, i1, f64, i1
}
```

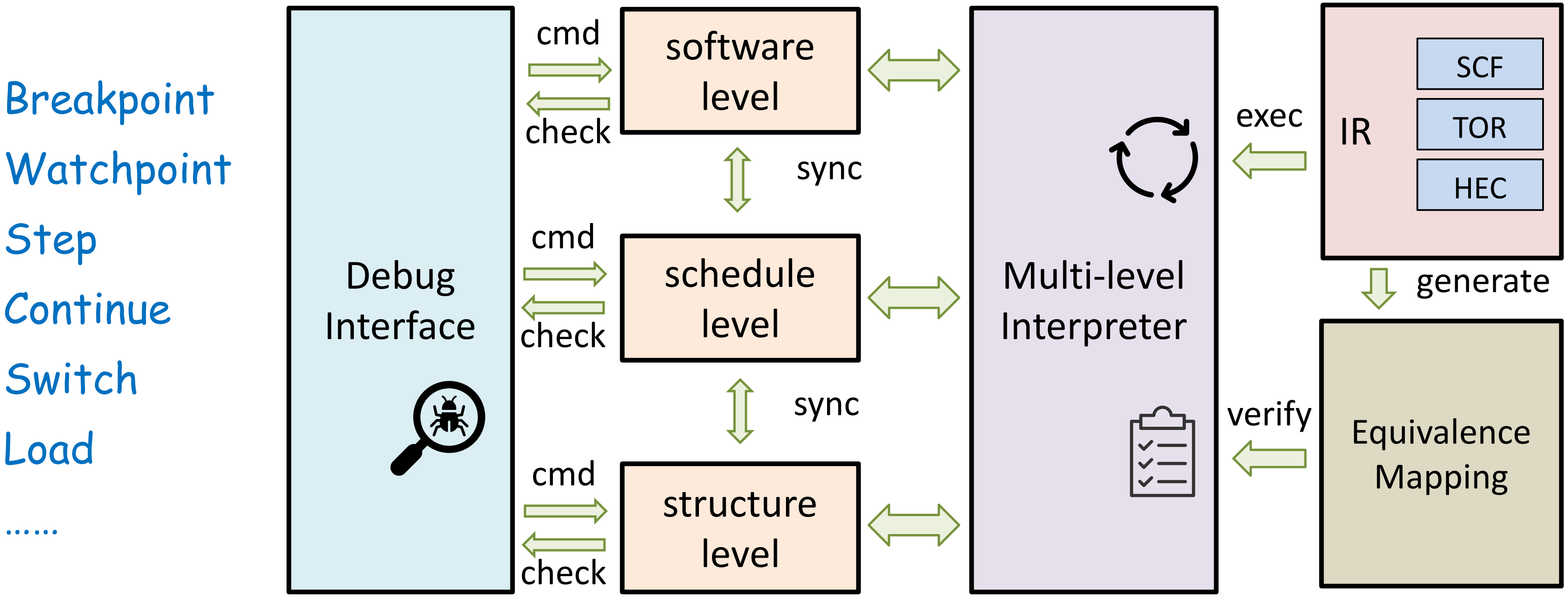
RTL Generation

- cat examples/chisel/gemm_ncubed_kernel.scala

```
class main extends MultiIOModule {
  val done = IO(Output(Bool()))
  done := 0.U
  val outline_0_0 = Module(new outline_0)
  val var8 = IO(Output(UInt(64.W)))
  object State extends ChiselEnum { val s0, s1, s1_entry, s2, s2_entry, s3,
s4, s5, s5_wait, s6, s7, s8, s9, s10 = Value }
  val state = RegInit(State.s0)
  switch (state) {
    is (State.s1) {
      val var90 = 0.U <= 62.U
      var76 := var90
      val var91 = !var90
      state := State.s2;
      when (var91.asBool()) { state := State.s10; }
    }
    is (State.s10) { done := 1.U }
  }
}
```

Hestia

- An efficient cross-level debugger for HLS designs that enables breakpoints and stepping at multiple granularity



Debugging at SCF

- ➢ `cd hestia/tutorial/examples/case1`
- ➢ `hestia command.tcl`
- ➢ `continue`
- ➢ `mem op_3`
- ➢ `exit`
- ➢ `cat data/D_out.txt`

```
load scf.json
load_memory_file op_0 data/A.txt
load_memory_file op_1 data/B.txt
load_memory_file op_2 data/C.txt
call main
```

```
RETURN:
op_3 Memory {
  store: [
    I32(30773412), I32(43773702), I32(39399768),
I32(42375816),
    I32(39878346), I32(46819968), I32(38864916),
I32(42650712),
    I32(41010798), I32(41823036), I32(32292810),
I32(33290802),
    I32(41601426), I32(40812594), I32(42230898),
I32(43038654),
    I32(32263200), I32(44749656), I32(40505076), …
  ]
}
```

Differs from output result.

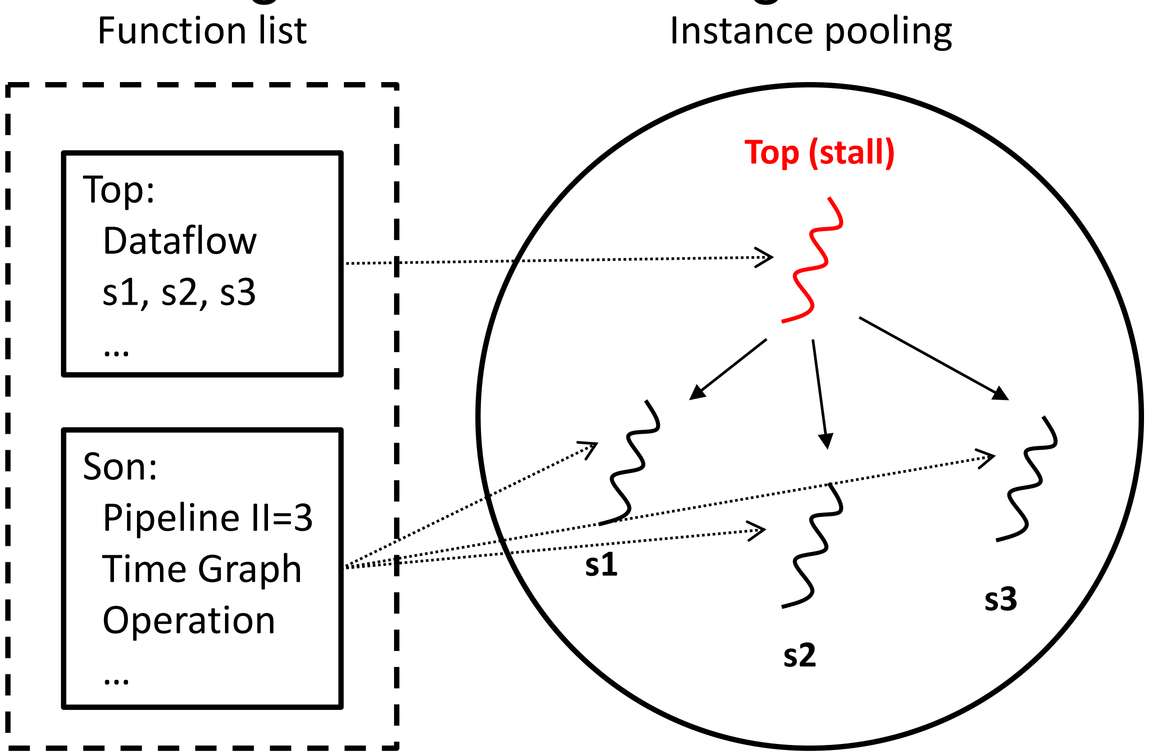
Debugging interface

- hestia command.tcl
- breakpoint op_116
- continue
- var op_115
- unset_breakpoint op_116
- watch op_135_b
- watch op_131_b
- watch op_115
- step 10000
- show_op
- step 2

```
SHOW VALUE:
        op_115 I32(0)
SHOW VALUE:
             op_115 I32(6)
        op_135_b I32(1)
        op_131_b I32(3)
        op_115 I32(6)
        op_135_b I32(1)
        op_131_b I32(3)
        op_115 I32(6) …
Computation { operands: ["op_135_b", "op_113"], op_type:
"shift_left", name: "op_114", ret_type: "i32" }
SHOW VALUE:
             op_115 I32(6)
        op_135_b I32(1)
        op_131_b I32(3)
        op_115 I32(7)
        op_135_b I32(1)
        op_131_b I32(3)
```

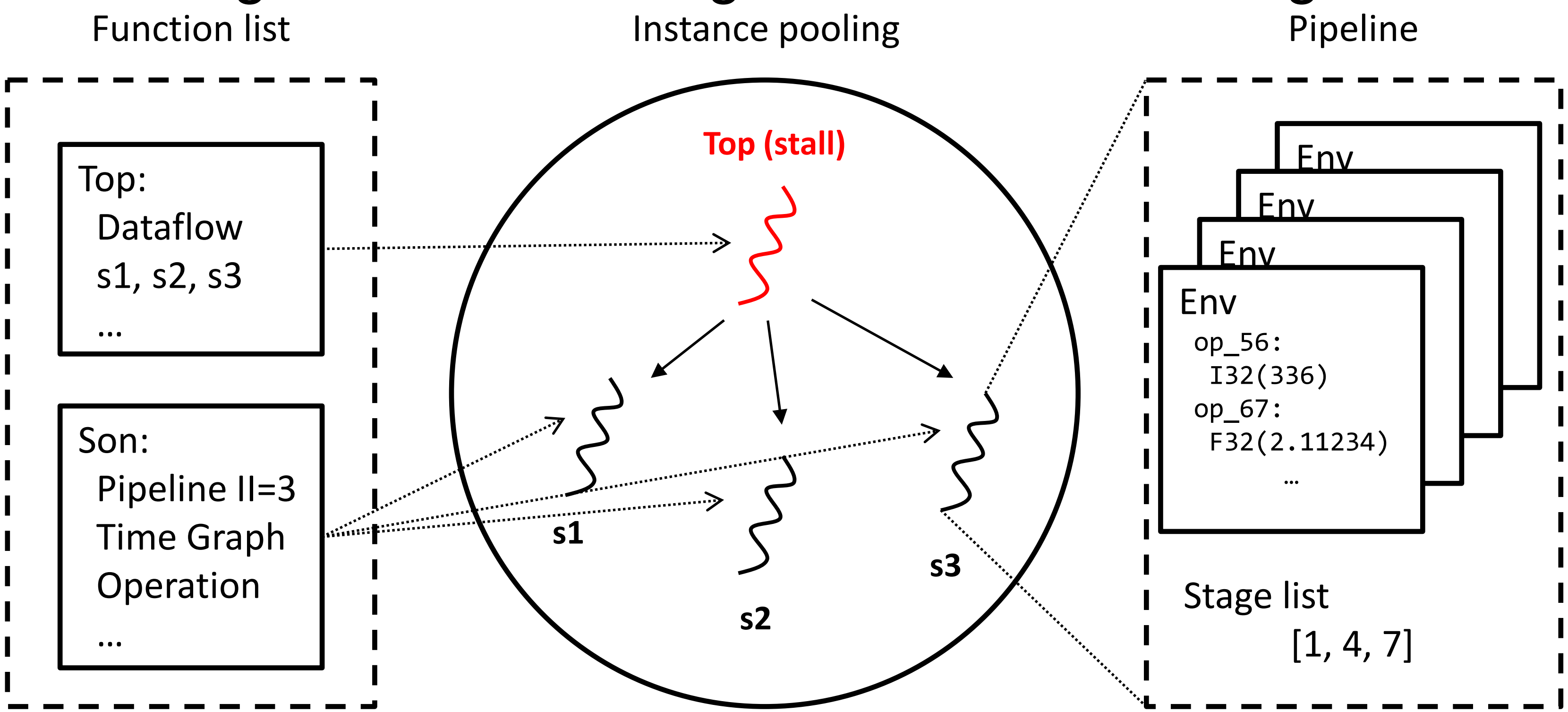
Multi-level Interpreter

- **Schedule simulation is organized as an instance pool**
  - Valuable insights into the timing behavior of the design



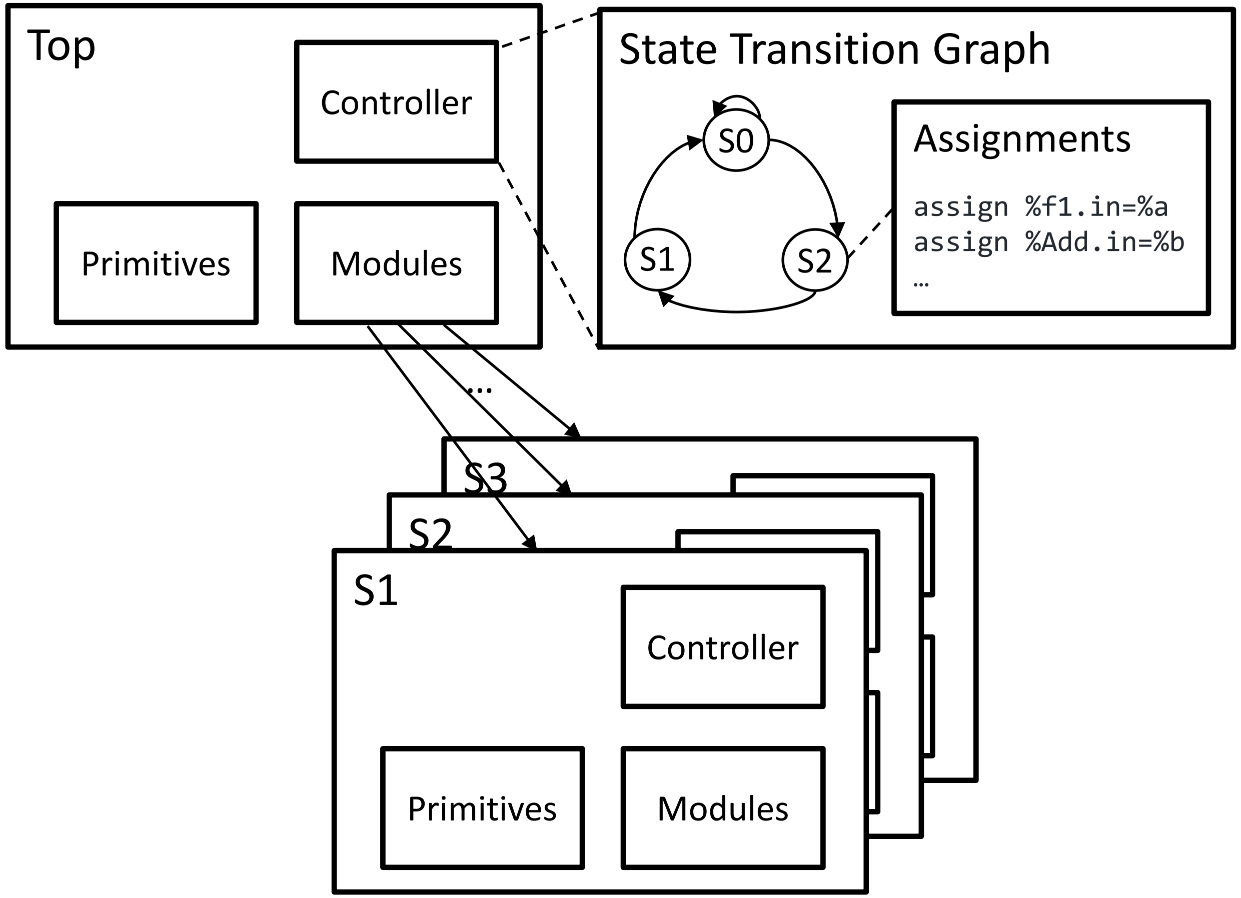
Multi-level Interpreter

- **Schedule simulation is organized as an instance pool**
  - Valuable insights into the timing behavior of the design



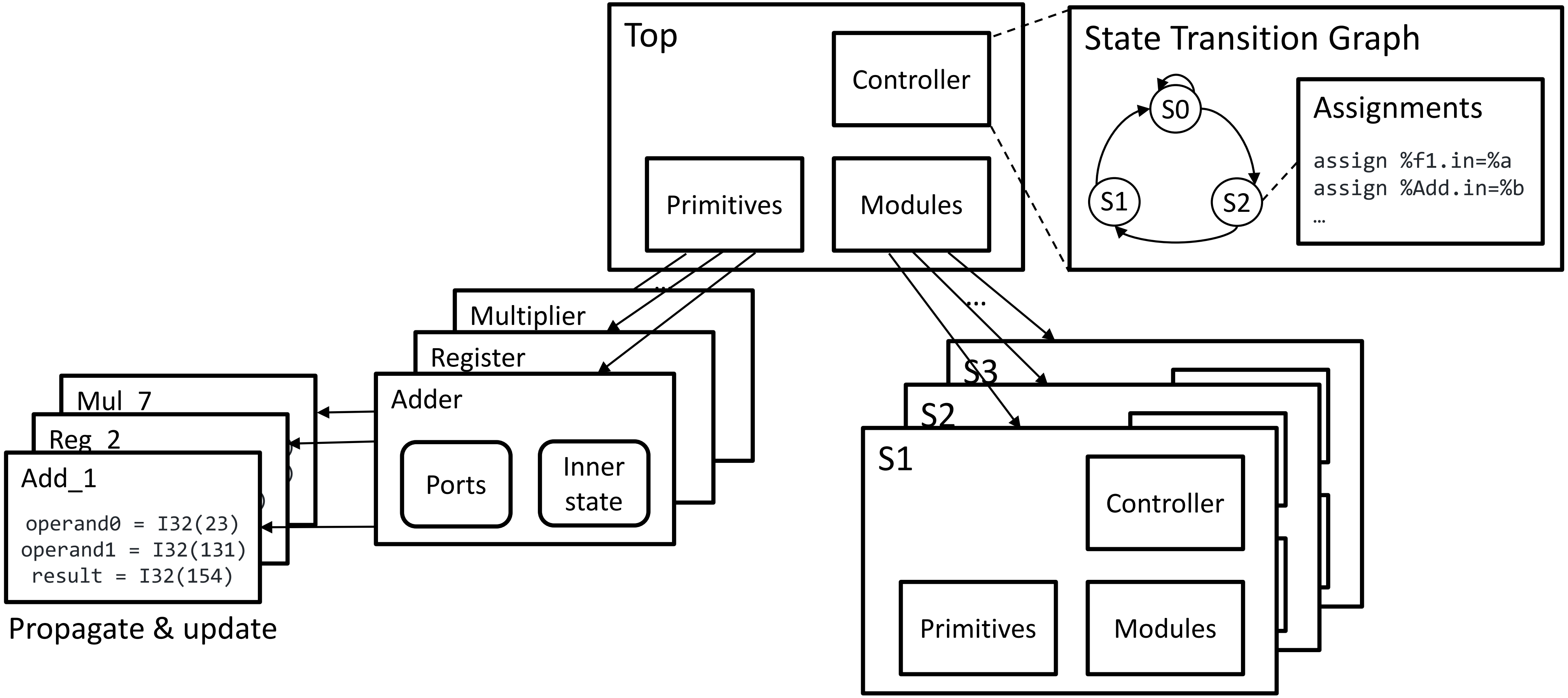
Multi-level Interpreter

- **Structural interpreter constructs a hierarchical structure**
  - Controller, primitives, sub-modules



Multi-level Interpreter

- **Structural interpreter constructs a hierarchical structure**
  - Controller, primitives, sub-modules



Simulation Speedup

```
cd ../../benchmarks/collection1
```

```
time hestia command/aeloss_push/tor.tcl
```

```
time hestia command/aeloss_push/hec.tcl
```

	TOR IR	HEC IR
Cycle	1502609	1502706
Time	2.019s	37.711s

Debugging at TOR & HEC levels

```
➢ cd ../../examples/case2
➢ hestia tor.tcl
➢ cat data/C_out.txt
```

```
load tor.json
load_memory_file op_0 data/C.txt
load_memory_file op_1 data/A.txt
load_memory_file op_2 data/B.txt
call main
continue
mem op_0
exit
```

Same as the output result.

```
Cycle count: 26283
op_0 Memory{
  store: [
    I32(738698), I32(34), I32(90), I32(26), I32(24), I32(57),
    I32(14), I32(68), I32(5), I32(58), I32(12), I32(86), I32(0),
    I32(46), I32(26), I32(94), I32(1099936), I32(1070152),
    I32(78), I32(29), I32(46), I32(90), I32(47), I32(70),
I32(51),
    I32(80), I32(31), I32(93), I32(57), I32(27), I32(12), …
  ]
}
```

Debugging at TOR & HEC levels

- ➢ `hestia hec_wrong.tcl`

```
load hec_wrong.json
load_memory_file mem_global_0 data/C.txt
load_memory_file mem_global_1 data/A.txt
load_memory_file mem_global_2 data/B.txt
call main
continue
mem mem_global_0
exit
```

```
thread 'main' panicked at src/lib/basetype.rs:168:35:
index out of bounds: the len is 256 but the index is 256
```

Debugging at TOR & HEC levels

- `hestia cosim.tcl`

```
thread 'main' panicked at src/lib/equal.rs:77:17:
assertion `left == right` failed:
  Value Mismatch:
  operation "op_44" and primitive "muli_main_0”
    at state @s14
  left: I32(624)
 right: I32(432)
```

```
load tor.json
load_memory_file op_0 data/C.txt
load_memory_file op_1 data/A.txt
load_memory_file op_2 data/B.txt
call main
load hec_wrong.json
#load hec.json
load_memory_file mem_global_0 data/C.txt
load_memory_file mem_global_1 data/A.txt
load_memory_file mem_global_2 data/B.txt
call main
load_equal equal.json
cosim
exit
```

Debugging at TOR & HEC levels

- `hestia cosim.tcl`
- `diff hec_wrong.json hec.json`

```
< "dst": "r_main_2.reg",
---
> "dst": "r_main_5.reg",
```

Cosimulation
Pass

```
load tor.json
load_memory_file op_0 data/C.txt
load_memory_file op_1 data/A.txt
load_memory_file op_2 data/B.txt
call main
#load hec_wrong.json
load hec.json
load_memory_file mem_global_0 data/C.txt
load_memory_file mem_global_1 data/A.txt
load_memory_file mem_global_2 data/B.txt
call main
load_equal equal.json
cosim
exit
```

Summary

- **Multi-level IR infrastructure**
  - Shows remarkable flexibility to domain-specific applications
- **Systolic array generation**
  - Achieves optimization in specific tasks through sophisticated dataflow and loop transformations, enhancing performance.
- **High-level synthesis and debugger**
  - Essential for reliable hardware design and verification

Thank you!