A Compiler Infrastructure

For Accelerator Generators

Rachit Nigam*, Samuel Thomas*, Zhijing Li, Adrian Sampson

https://github.com/cucapra/futil

Computer **A**rchitecture and
Programming **A**bstractions Group

Accelerator Generator

Compiler from High-Level Specifications
to Hardware Designs

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

Mapping resources

Timed specification

Structural encoding for control-flow

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

Software IRs

SUIF Compiler System

GCC

Hardware IRs

FIRRTL

CoreIR

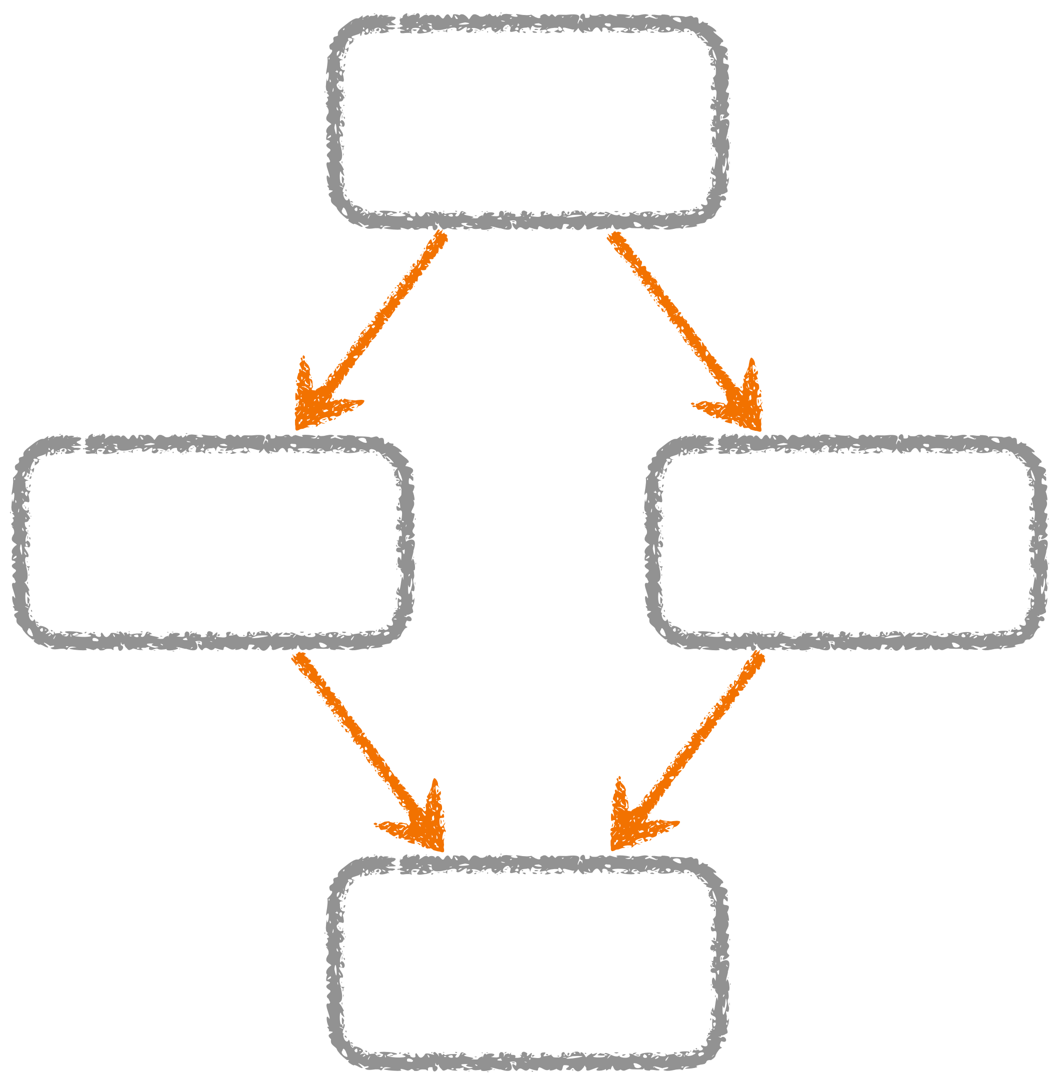
LLHD

SSA

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

```
%0 = mod count, 10
br %0, tru, fal

tru:
  %1 = add a, b; jmp join;
fal:
  %2 = add c, d; jmp join;
join:
  x = phi [%2, fal] [%1, tru]
```

SSA



Control flow facts

```
%0 = mod count, 10
br %0, tru, fal

tru:
  %1 = add a, b; jmp join;
fal:
  %2 = add c, d; jmp join;
join:
  x = phi [%2, fal] [%1, tru]
```

SSA

Untimed Specification

Logical instances

```
%0 = mod count, 10
br %0, tru, fal

tru:
  %1 = add a, b; jmp join;
fal:
  %2 = add c, d; jmp join;
join:
  x = phi [%2, fal] [%1, tru]
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

Structural Constructs

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

Multi-cycle operation

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

Lack of
control flow

RTL

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

Disjoint execution

Share!

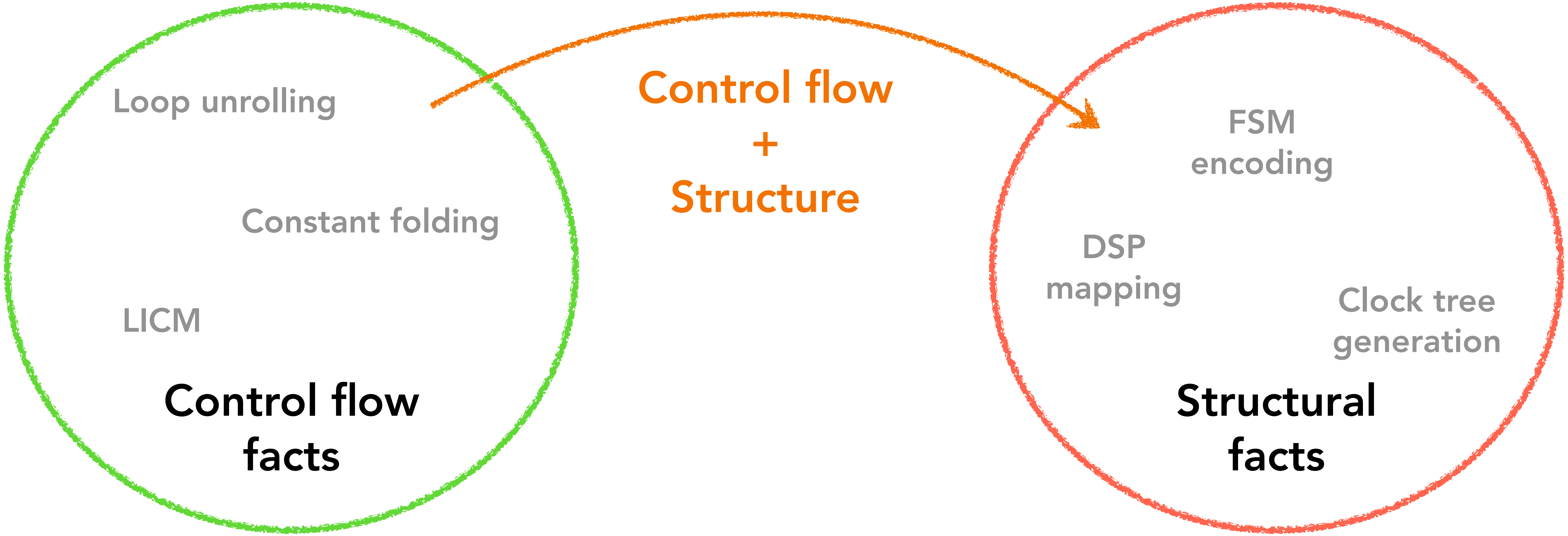
```
register x (x_in, x_we)
adder add0 (a_out, b_out)
adder add1 (c_out, d_out)
modulus mod0 (count_out, 10)

assign x_in =
  mod0_out ? add0_out
           : add1_out

assign x_we = mod0_done
```

```
if (count % 10):
  x := a + b
else:
  x := c + d
```

Control flow facts

```
register x (x_in, x_we)
adder add₀ (a_out, b_out)
adder add₁ (c_out, d_out)
modulus mod₀ (count_out, 10)

assign x_in =
  mod₀_out ? add₀_out
           : add₁_out

assign x_we = mod₀_done
```

Structural facts

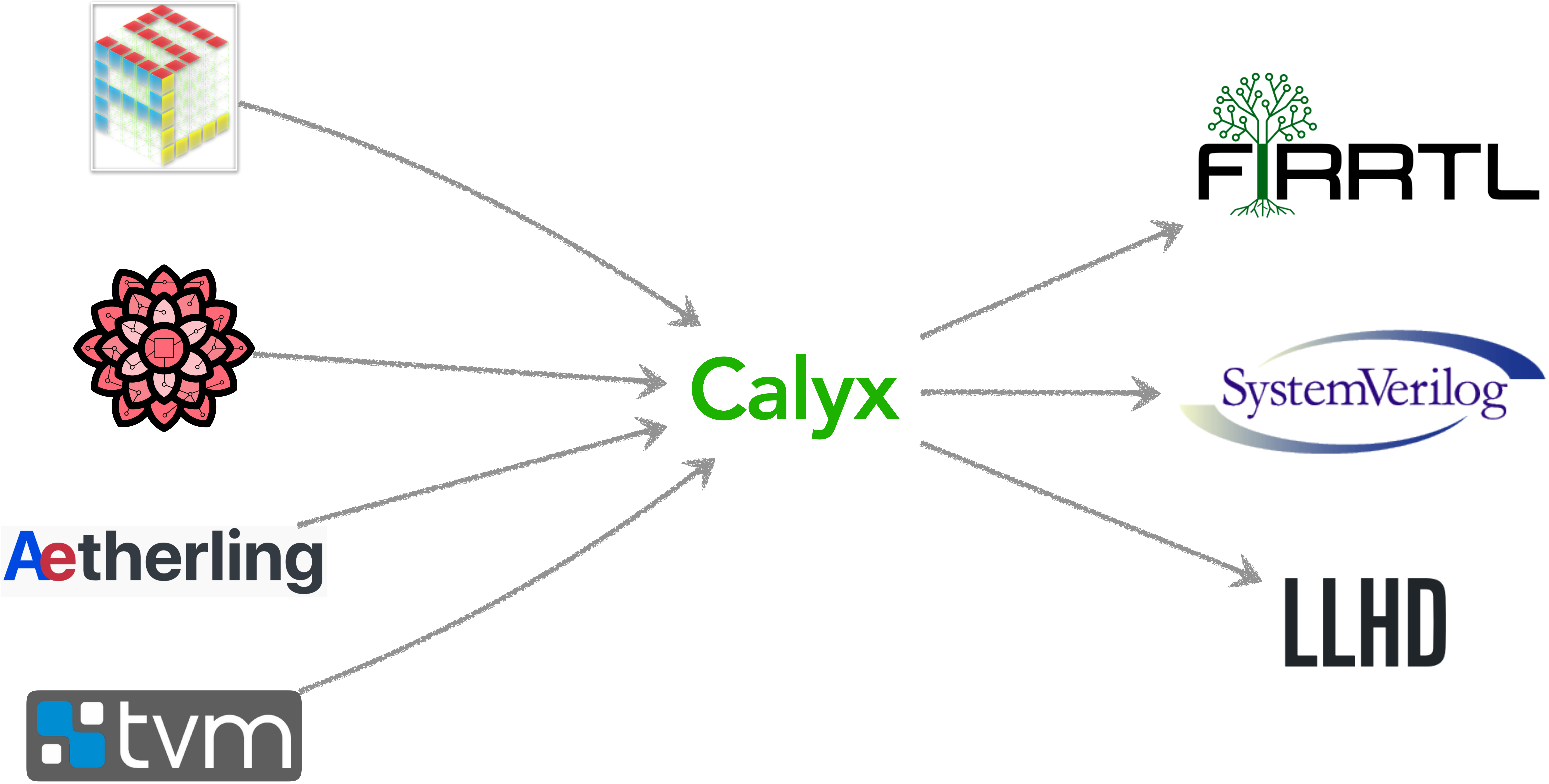

Loop unrolling
Constant folding
LICM
Control flow facts
Control flow + Structure
FSM encoding
DSP mapping
Clock tree generation
Structural facts

Calyx

High-level control flow

Low-level structure

Image by Eucalyp


Aetherling
tvm
Calyx
FIRRTL
SystemVerilog
LLHD

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires { … }

  control { … }
}
```

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires { … }

  control { … }
}
```

Input and output **ports** with size in bits

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires { … }

  control { … }
}
```

Structural **sub-components** used within this component

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires { … }

  control { … }
}
```

Components instantiated with **parameters**

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires {
    add.left = c.out;
    add.right = 32'd1;
  }

  control { … }
}
```

Connections between sub-components

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires {
    add.left = c.out;
    add.right = 32'd1;
  }

  control { … }
}
```

Non-blocking assignments to ports on components

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires {
    add.left = cmp.out ? c.out;
    add.right = 32'd1;
  }

  control { … }
}
```

Guarded assignment to ports on components

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires {
    add.left = cmp.out ? c.out;
    add.left = !cmp.out ? 32'd0;
    add.right = 32'd1;
  }

  control { … }
}
```

All guarded assignments should have **unique drivers**

Calyx **does not verify** this!

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells {
    c = std_reg(32);
    add = std_add(32);
    cmp = std_le(32);
 }

  wires {
    add.left = cmp.out ? c.out;
    add.right = 32'd1;
  }

  control { … }
}
```

Guarded assignments can describe arbitrary RTL program*

*With a single clock signal

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      add.left = c.out;
      add.right = 32'd1;
      c.in = add.out;
    }
  }

  control { … }
}
```

Groups provide a named encapsulation over guarded assignments

Groups can also encode **arbitrary** RTL program

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      add.left = c.out;
      add.right = 32'd1;
      c.in = add.out;
    }
  }

  control { … }
}
```

Assignments within a group must have a **unique driver**

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      add.left = c.out;
      add.right = 32'd1;
      c.in = add.out;
    }
    group reset_count {
      c.in = 32'd0;
    }
  }

  control { … }
}
```

However, multiple groups can **drive the same port**

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      add.left = c.out;
      add.right = 32'd1;
      c.in = add.out;
    }
    group reset_count {
      c.in = 32'd0;
    }
  }

  control { … }
}
```

Execution schedule for the component

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      add.left = c.out;
      add.right = 32'd1;
      c.in = add.out;
    }
    group reset_count {
      c.in = 32'd0;
    }
  }

  control { seq { reset_count; incr_count } }
}
```

Sequencing operator: Run sub-programs in sequence

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      add.left = c.out;
      add.right = 32'd1;
      c.in = add.out;
    }
    group reset_count {
      c.in = 32'd0;
    }
  }

  control { seq { reset_count; incr_count } }
}
```

When does control flow **return**?

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      …
      incr_count[done] = c.done ? 1'd1;
    }
    group reset_count {
      …
      reset_count[done] = c.done ? 1'd1;
    }
  }

  control { seq { reset_count; incr_count } }
}
```

Groups must define a **done condition** to return control flow

Calyx

```
component counter(end: 32) -> (out: 32) {
  cells { … }

  wires {
    group incr_count {
      …
      incr_count[done] = c.done ? 1'd1;
    }
    group reset_count {
      …
      reset_count[done] = c.done ? 1'd1;
    }
  }

  control { seq { reset_count; incr_count } }
}
```

```
cells {
  count = std_reg(32);
  add = std_add(32);
  cmp = std_lt(32);
}
```

Instantiate sub-components

```
group reset {
   count.in = 32'd0;
   count.write_en = 1'd1;
   reset[done] = count.done;
 }
 group incr {
   add.left = count.out;
   add.right = 32'd1;
   count.in = add.out;
   count.write_en = 1'd1;
   incr[done] = count.done;
 }
 group cond {
   cmp.left = count.out;
   cmp.right = 32'd10;
   cond[done] = 1'd1;
 }
```

Groups

```
control {
  seq {
    reset;
    while cmp.out with cond {
      incr;
    }
  }
}
```

Execution Schedule

Calyx

Intermediate languages enable **analysis** and **transformation**

Calyx

clk-insertion
collapse-control
compile-control
compile-empty
compile-invoke
component-interface-inserter
dead-cell-removal
externalize
go-insertion
hole-inliner

infer-static-timing
merge-assign
minimize-regs
papercut
remove-external-memories
resource-sharing
simplify-guards
static-timing
top-down-cc
well-formed

Resource Sharing

```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

```
if (a > c):
  x := a + b
else:
  x := c + d
```

Resource Sharing

```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

```
if (a > c):
  x := a + b
else:
  x := c + d
```

Resource Sharing

Analyze · Calculate · Transform

```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Disjoint executions

Resource Sharing

Analyze · Calculate · Transform

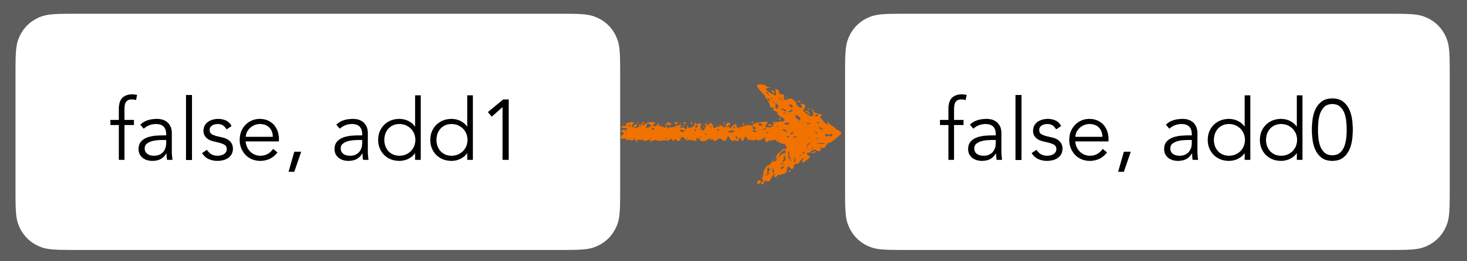
```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Non-stateful **shareable** components

Resource Sharing

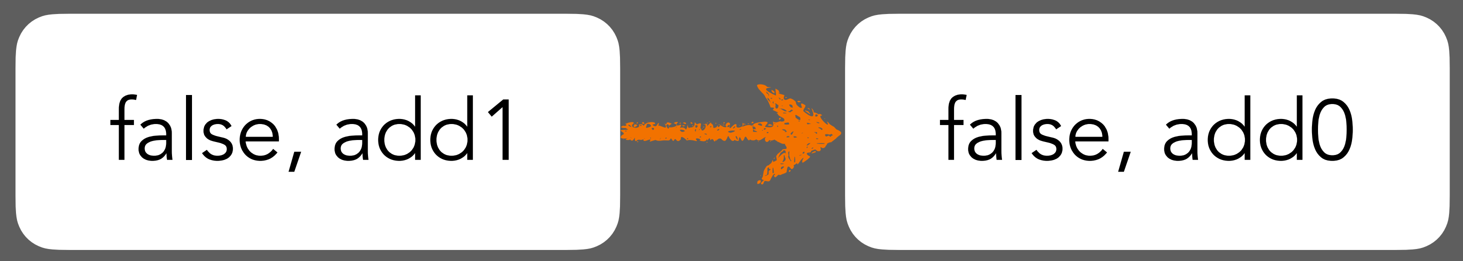
Analyze · Calculate · Transform

```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```



Resource Sharing


```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add0.left = c.out;
  add0.right = d.out;
  x.in = add0.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```



Calyx

Lowering control to structure

Calyx

```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```


Compile Control

Does not assume anything about groups

Calyx

```
group cond { … }
group true {
  …
  true[done] = x.done;
}
group false {
  …
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Compile Control

```
cells {
}

group comp_if {
}
```

Compilation is represented as a **new group**

Calyx

```
group cond { … }
group true {
  …
  true[done] = x.done;
}
group false {
  …
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Compile Control

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  cond[go] = !cs.out ? 1'd1;
  c.in = cond[done] ? cmp.out;
  cs.in = cond[done] ? 1'd1;

}
```

Run cond and **Save** the value of the condition

Calyx

```
group cond { … }
group true {
  …
  true[done] = x.done;
}
group false {
  …
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Compile Control

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  cond[go] = !cs.out ? 1'd1;
  c.in = cond[done] ? cmp.out;
  cs.in = cond[done] ? 1'd1;

}
```

Run cond and **Save** the value of the condition

Calyx

```
group cond { … }
group true {
  …
  true[done] = x.done;
}
group false {
  …
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Compile Control

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  c.in = cond[done] ? cmp.out;
  cs.in = cond[done] ? 1'd1;

  true[go] = c.out & cs.out ? 1'd1;
  false[go] = !c.out & cs.out ? 1'd1;

}
```

Run branch based on the condition

Calyx

```
group cond { … }
group true {
  …
  true[done] = x.done;
}
group false {
  …
  false[done] = x.done;
}

control {
  if cmp.out with cond { true } else { false }
}
```

Compile Control

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  c.in = cond[done] ? cmp.out;
  cs.in = cond[done] ? 1'd1;

  true[go] = c.out & cs.out ? 1'd1;
  false[go] = !c.out & cs.out ? 1'd1;

  comp_if[done] =
    true[done] || false[done];
}
```

Done when one of the branches is done

Calyx

```
group cond { … }
group true {
  …
  true[done] = x.done;
}
group false {
  …
  false[done] = x.done;
}

control {
  comp_if;
}
```

Replace control program with group

Compile Control

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  c.in = cond[done] ? cmp.out;
  cs.in = cond[done] ? 1'd1;

  true[go] = c.out & cs.out ? 1'd1;
  false[go] = !c.out & cs.out ? 1'd1;

  comp_if[done] =
    true[done] || false[done];
}
```

Calyx

```
group cond { … }
group true {
  add0.left = a.out;
  add0.right = b.out;
  x.in = add0.out;
  true[done] = x.done;
}
group false {
  add1.left = c.out;
  add1.right = d.out;
  x.in = add1.out;
  false[done] = x.done;
}

control {
  comp_if;
}
```

Compile Groups

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  c.in = cond[done] ? cmp.out;
  cs.in = cond[done] ? 1'd1;

  true[go] = c.out & cs.out ? 1'd1;
  false[go] = !c.out & cs.out ? 1'd1;

  comp_if[done] =
    true[done] || false[done];
}
```

Groups are an abstraction, not real hardware

Calyx

```
group cond { … }
group true {
  add0.left = true[go] ? a.out;
  add0.right = true[go] ? b.out;
  x.in = true[go] ? add0.out;
  true[done] = x.done;
}
group false {
  add1.left = false[go] ? c.out;
  add1.right = false[go] ? d.out;
  x.in = false[go] ? add1.out;
  false[done] = x.done;
}

control {
  comp_if;
}
```

Compile Groups

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  c.in = cond[done] & … ? cmp.out;
  cs.in = cond[done] & … ? 1'd1;

  true[go] = … ? 1'd1;
  false[go] = … ? 1'd1;

  comp_if[done] =
    true[done] || false[done];
}
```

Guard all assignments in groups

Calyx

```
group cond { … }
group true {
  add0.left = true[go] ? a.out;
  add0.right = true[go] ? b.out;
  x.in = true[go] ? add0.out;
  true[done] = x.done;
}
group false {
  add1.left = false[go] ? c.out;
  add1.right = false[go] ? d.out;
  x.in = false[go] ? add1.out;
  false[done] = x.done;
}

control {
  comp_if;
}
```

Compile Groups

```
cells {
  c, cs = std_reg(1);
}

group comp_if {
  c.in = cond[done] & … ? cmp.out;
  cs.in = cond[done] & … ? 1'd1;

  true[go] = … ? 1'd1;
  false[go] = … ? 1'd1;

  comp_if[done] =
    true[done] || false[done];
}
```

Inline go and done guards

Calyx

```
wires {

  add0.left = c.out & cs.out ? a.out;
  add0.right = c.out & cs.out ? b.out;
  x.in = c.out & cs.out ? add0.out;

  add1.left = !c.out & cs.out ? c.out;
  add1.right = !c.out & cs.out ? d.out;
  x.in = !c.out & cs.out ? add1.out;

  done = x.done || x.done;
}
control {}
```

Purely structural Calyx

Compile Groups

```
cells {
  c, cs = std_reg(1);
}
```

Calyx

Mixed latency sensitive and insensitive compilation

Calyx

```
group one { … }
group two { … }
group three { … }

control {
  seq { one; two; three }
}
```

Static Timing

Calyx

```
group one<“static”=1> { … }
group two<“static”=2> { … }
group three { … }

control {
  seq { one; two; three }
}
```

Group Attributes

Static Timing

Calyx

```
group one<“static”=1> { … }
group two<“static”=2> { … }
group three { … }

control {
  seq { one; two; three }
}
```


Static Timing

Static timing: Number of cycles between **go** and **done**.

Calyx

```
group one<“static”=1> { … }
group two<“static”=2> { … }
group three { … }

control {
  seq { one; two; three }
}
```

Static Timing

Compile static groups with cycle-accurate FSM

Calyx

```
group one<“static”=1> { … }
group two<“static”=2> { … }
group static_seq<“static”=3> { … }
group three { … }

control {
  seq { static_seq; three }
}
```

Static Timing

Compile static groups with cycle-accurate FSM

Calyx

```
group one<“static”=1> { … }
group two<“static”=2> { … }
group three { … }

control {
  seq { static_seq; three }
}
```

Static Timing

Remaining control is lowered by **compile-control**

Calyx

High-level
control flow

Low-level
structure

https://github.com/cucapra/futil

Image by Eucalyp