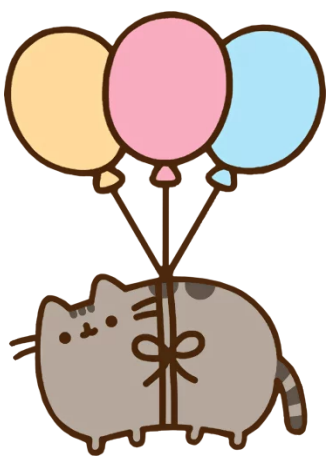
CIRCT 编译器的电路划分及 Arcilator 仿真并行化



Let's Rewind...

Arcilator 是一个基于 CIRCT 的电路仿真综合器，工作原理是将硬件表示直接转换为 LLVM IR。

$$\text{FIRRTL / Verilog / ...} \rightarrow \text{HW Dialect} \rightarrow \text{LLVM IR} \rightarrow \text{Executable}$$

Let’s Rewind...

Arcilator 是一个基于 CIRCT 的电路仿真综合器，工作原理是将硬件表示直接转换为 LLVM IR。

$$\text{FIRRTL / Verilog / ...} \rightarrow \text{HW Dialect} \rightarrow \text{LLVM IR} \rightarrow \text{Executable}$$

在本项目开始前只支持单线程。本项目的目标是为其引入多线程支持。

What’s done

核心是将仿真逻辑进行划分。

- 引入新的 IR 结构表示划分成的任务，以及任务之间的同步关系。
-

What's done

核心是将仿真逻辑进行划分。

- 引入新的 IR 结构表示划分成的任务，以及任务之间的同步关系。
- 在后端将任务分裂为不同的“入口”，允许运行时进行调度。

```
void model_eval(model_storage_t *storage);
```

What's done

核心是将仿真逻辑进行划分。

- 引入新的 IR 结构表示划分成的任务，以及任务之间的同步关系。
- 在后端将任务分裂为不同的“入口”，允许运行时进行调度。

```
void model_eval(model_storage_t *storage);
```

```
void model_eval_task_1(model_storage_t *storage);
void model_eval_task_2(model_storage_t *storage);
```

IR structure

引入了新的 IR 结构:

```
arc.task #bouba  { /* Do stuff... */ }
arc.task #kiki   { /* Do stuff... */ }
arc.task #foobar { /* Do stuff... */ }
```

arc.task 遵从 PC 内存序，名字相同的 task 将会进行合并，并在过程中静态检查违例。

IR structure

引入了新的 IR 结构:

```
arc.task #bouba  { /* Do stuff... */ }
arc.task #kiki   { /* Do stuff... */ }
arc.task #foobar { /* Do stuff... */ }
```

arc.task 遵从 PC 内存序，名字相同的 task 将会进行合并，并在过程中静态检查违例。

arc.task 只有来自 arc.state_read 和 arc.state_write 的内存副作用!

Semantic registers

→ unallocated `arc.State`

→ allocated `arc.State`

→ LLVM getelementptr

Semantic registers

→ **unallocated `arc.State`**

→ **allocated `arc.State`**

→ LLVM getelementptr

State updates

#7703 引入了新的状态下降方式：

```
reg state : UInt<1>, clk
state <= state_next
```

State updates

#7703 引入了新的状态下降方式:

```
reg state : UInt<1>, clk
state <= state_next
```

```
// Compute clk's value AFTER this eval
// Compute state_next's value BEFORE this eval
if(storage->old_clk != clk)
  storage->state = state_next;
storage->old_clk = clk;
```

State updates

假设新状态计算及状态更新由同一线程完成，将状态更新过程分裂为两步：

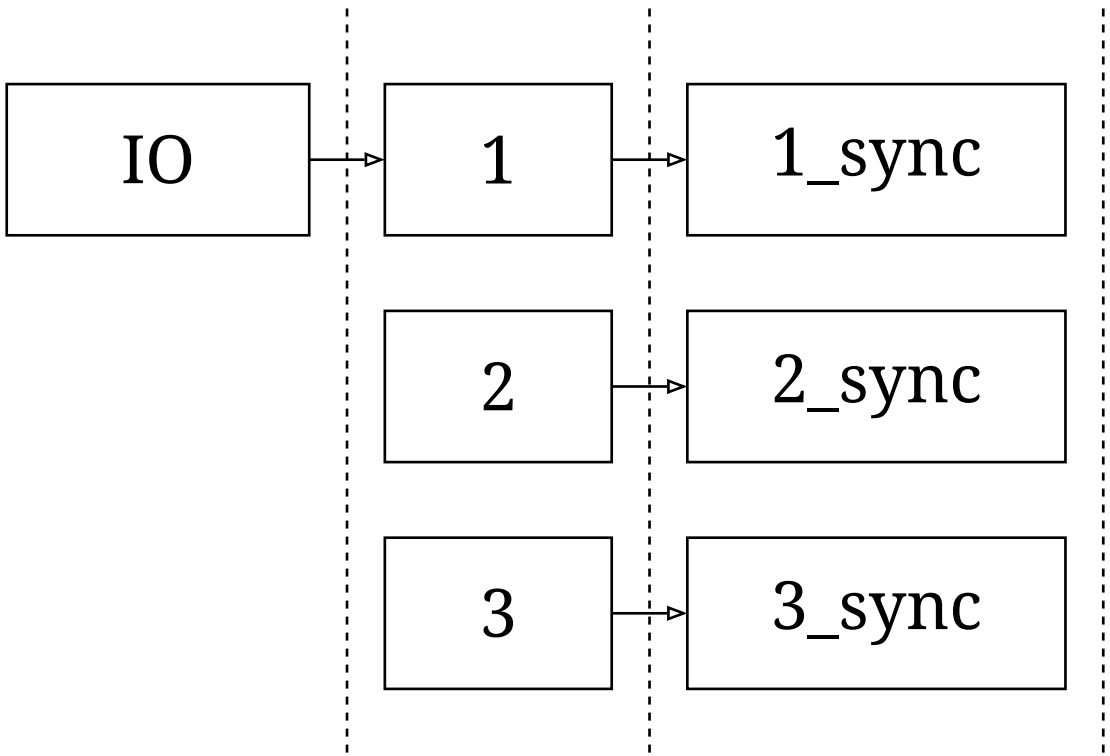
- 在原先的写入地点（`arc.state_write`），将新的状态值写入一个 Shadow register 中，仅本线程可见。
- 在整个 eval 结尾插入一个新的 Task，将 Shadow register 中的值搬运回本来写入的位置。

State updates

假设新状态计算及状态更新由同一线程完成，将状态更新过程分裂为两步：

- 在原先的写入地点（`arc.state_write`），将新的状态值写入一个 Shadow register 中，仅本线程可见。
- 在整个 eval 结尾插入一个新的 Task，将 Shadow register 中的值搬运回本来写入的位置。

后者称为 “Sync task”。


IO
1
1_sync
2
2_sync
3
3_sync

Partition planning

- 根据 SSA 爬出来状态之间的依赖关系，**按寄存器分配**
-

Partition planning

- 根据 SSA 爬出来状态之间的依赖关系，**按寄存器分配**
- METIS 一把梭

Results

Rocket 可以正确在多线程下执行，线程个数不定。

- Baseline: 354540 Hz
- 2-划分，并行执行: 294718 Hz
- 2-划分，串行执行: 180212 Hz

Results

Rocket 可以正确在多线程下执行，线程个数不定。

- Baseline: 354540 Hz
- 2-划分，并行执行: 294718 Hz
- 2-划分，串行执行: 180212 Hz

CIRCT #7650

Thank you!