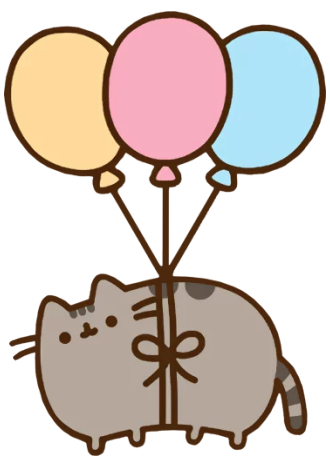
CIRCT 编译器的电路划分及 Arcilator 仿真并行化



Background

传统的 EDA 工具链：

$$\text{Verilog/VHDL} \rightarrow \text{Simulator} \rightarrow \text{Trace / Assertions / DPI Calls...}$$
$$\rightarrow \text{Synthesis} \rightarrow \text{Netlist / Floorplan...}$$

问题：Verilog/VHDL 不是一个很“好”的电路描述语言：

- 本质上是一个过程式语言
- 可综合和不可综合的语言子集没有明显划分

```
@always begin
  #1;
  a = 1;
  #1;
  a = 2;
end
```

Background

一个更理想的以最终产生电路为目标的 HDL 语言：Chisel / FIRRTL：以寄存器和连线为

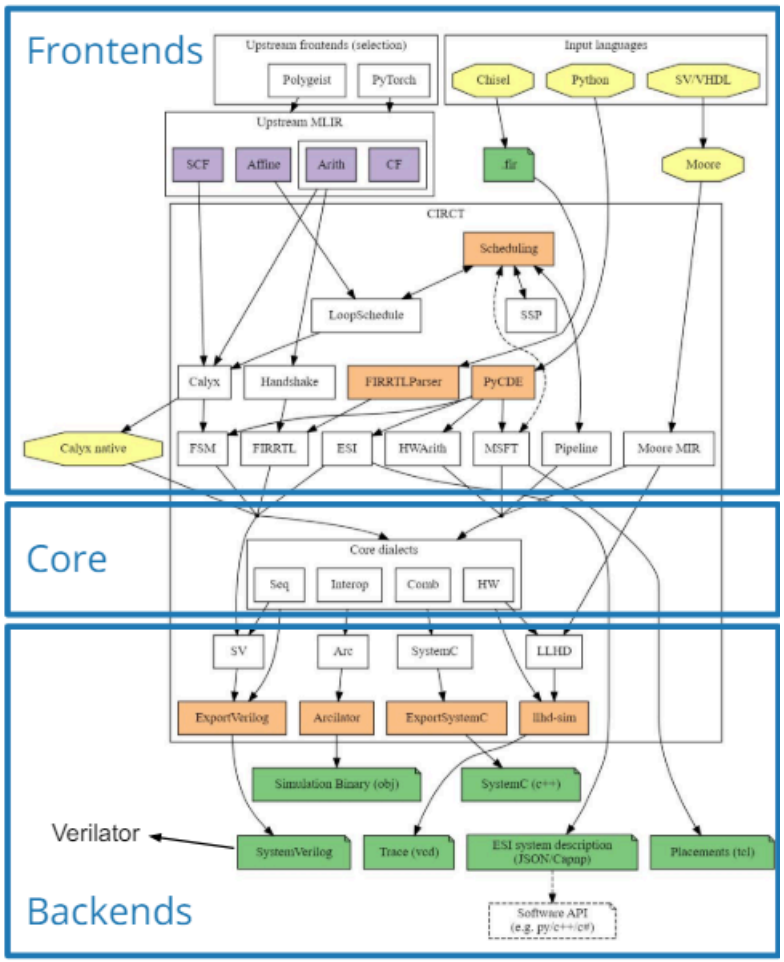
$$\text{State}_{n+1} \leftarrow f(\text{State}_n)$$

目前 FIRRTL 的仿真综合后端是基于 MLIR 的 CIRCT 编译器。

Background

CIRCT 编译器表示提供了一种新的电路仿真方法:

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


Frontends
Upstream frontends (selection)
Polygeist
PyTorch
Input languages
Chisel
Python
SV/VHDL
Upstream MLIR
SCF
Affine
Arith
CF
fir
Moore
CIRCT
Scheduling
LoopSchedule
SSP
Calyx
Handshake
FIRRTLParser
PyCDE
Calyx native
FSM
FIRRTL
ESI
HWArith
MSFT
Pipeline
Moore MIR
Core
Core dialects
Seq
Interop
Comb
HW
SV
Arc
SystemC
LLHD
ExportVerilog
Arcilator
ExportSystemC
llhd-sim
Simulation Binary (obj)
SystemC (c++)
Verilator
SystemVerilog
Trace (vcd)
ESI system description (JSON/Capnp)
Placements (tcl)
Backends
Software API (e.g. py/c++/c#)

```
hw.module @SumOfFibonacci(
    in %clock : !seq.clock,
    in %rst : i1,
    in %en : i1,
    out out : i32)
{
  %c0_i32 = hw.constant 0 : i32
  %c12_i32 = hw.constant 12 : i32
  %0 = comb.mux %rst, %c0_i32, %4 : i32
  %reg = seq.compreg %0, %clock : i32
  %fib.out, %fib.count = hw.instance "fib" @Fibonacci(
    clock: %clock: !seq.clock, rst: %rst: i1, en: %2: i1)
    -> (out: i32, count: i32)
  %1 = comb.icmp ult %fib.count, %c12_i32 : i32
  %2 = comb.and %en, %1 : i1
  %3 = comb.add %reg, %fib.out : i32
  %4 = comb.mux %2, %3, %reg : i32
  hw.output %reg : i32
}
```

Instantiation

Combinational logic

```
hw.module private @Fibonacci(
    in %clock : !seq.clock,
    in %rst : i1,
    in %en : i1,
    out out : i32,
    out count : i32)
{
  %c1_i32 = hw.constant 1 : i32
  %c0_i32 = hw.constant 0 : i32
  %0 = comb.mux %rst, %c0_i32, %5 : i32
  %reg_0 = seq.compreg %0, %clock : i32
  %1 = comb.mux %rst, %c1_i32, %3 : i32
  %reg_1 = seq.compreg %1, %clock : i32
  %2 = comb.mux %rst, %c0_i32, %7 : i32
  %counter = seq.compreg %2, %clock : i32
  %3 = comb.mux %en, %reg_0, %reg_1 : i32
  %4 = comb.add %reg_0, %reg_1 : i32
  %5 = comb.mux %en, %4, %reg_0 : i32
  %6 = comb.add %counter, %c1_i32 : i32
  %7 = comb.mux %en, %6, %counter : i32
  hw.output %reg_1, %counter : i32, i32
}
```

Registers

Motivation

目前 Arc 只支持单线程的仿真：

- 缺乏多个线程之间状态的同步
- 缺乏线程的创建和调度
- **缺乏电路划分**

当硬件规模增大，需要添加多线程支持。计算划分同时可以用于 FPGA 多片互联的自动综合。

Motivation

Verilator 仿真器的多线程支持：

- Verilog 的仿真是 Task-based 的：每个 always 块，带有特定延迟的语句或者一个 task 块是一个 Task。
- Task queue 动态调度
- DPI 全局加锁，Trace 有部分多线程支持

Motivation

倾向于静态划分

- Arcilator 所处理的电路表示不存在 Task
- 目前不存在依赖分析，也就不存在事件触发的仿真
- 静态划分可以大幅增加局部性

Plan

第一步：电路划分

- 添加一级 LLVM Pass，在 Arc 之前对 CIRCT 核心方言构成的电路树进行划分
- 对每个子电路分别经过 Arc 的 Lowering 过程
- 直接将划分出的部分粘起来：创建跨分块通信所需的 Buffer

Plan

第二步：简单多线程支持

- 添加线程的创建和销毁，Buffer 移动到全局
- 添加同步

这部分工作可以使用运行时实现

Plan

本项目目标：

可以至少将 Rocket 规模的 FIRRTL，切分为至少两个线程并行执行，保证仿真的正确性

暂时没有性能指标。

Future work

- 考虑如何和 DPI、Trace 协同工作
- 考虑添加动态调度的可能性

Thank you!