Generating Instruction Set Simulators from Hardware Description with CIRCT MLIR

Nicholas Fry[1, 2] Fatma Jebali[2] Caaliph Andriamisaina[2]
Imperial College London[1], CEA-List[2]

Overview

Instruction set simulation (ISS) is critical for fast SW validation and compiler design. Typically, an ISS is written before the register-transfer level (RTL) HW implementation. As the HW evolves, extensive development and verification must be conducted to ensure **equivalence between RTL and ISS**. Writing bespoke simulators for existing hardware designs is time consuming and requires in-depth knowledge of the micro-architecture. This motivates a way of automatically deriving an ISS from its corresponding RTL implementation.

We propose a **WIP** method that leverages **CIRCT MLIR** to generate ISS functions (i.e **"state transition functions"**) in the form of **LLVM IR,** from a hardware design. The generated functions can then form the basis of a simulated assembly program. Using this tool, the **RTL and ISS may evolve in parallel**, and are **consistent by construction**.

The method works well on toy accelerators and sub modules of larger designs (e.g. RocketCore's ALU), but has yet to be tested on full architectures.

Advantages & Limitations

- Generated ISS is **consistent with hardware by construction**.
- Inherits **parallelism and flexibility from MLIR** code analysis.
- Can be used with **different hardware languages** (if the frontend is present in the CIRCT project - e.g. Chisel, SystemVerilog, Python).

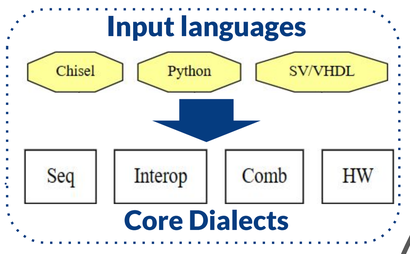
- Requires **knowledge of hardware registers** containing the **opcode** of the ISA instruction being generated.
- Requires an **estimate** of the upper bound hardware **latency**.
- Can't handle memories yet, however the tool is easily extensible.

Algorithm

1. Convert to MLIR

The hardware design is converted to MLIR using one of the CIRCT frontends

firtool, slang ...



2. Constant assignment & folding

Registers are set to capture the state of an instruction in cycle N of execution

Replace with constants

```
hw.module @SimpleCountBits(%clock: i1, %reset: i1, %io_optn: i2,
%io_dataIn: i2) -> (io_result: i2) {
  %false = hw.constant false
  %c-1_i2 = hw.constant -1 : i2
  %c1_i2 = hw.constant 1 : i2
  ...
```

Fold the design

3. State capture & propagation

The state of the design is captured and fed to the registers in the next cycle

```
%c-2_i2 = hw.constant -2 : i2
%c0_i2 = hw.constant 0 : i2
%0 = comb.mux %reset, %c0_i2, %c-2_i2 : i2
%counter = seq.compreg %0, %clock : i2
```

Save the def-use chain for each register

Reinsert the register state from cycle n-1 back into the original design

After LATENCY cycles

4. Lower to LLVM

The unrolled design is lowered to LLVM IR and optimised using standard passes **-O3**

Example

Hardware Description (Chisel)

```
class CountPositiveBits extends Module {
  val io = IO(new Bundle {
    val optn = Input(UInt(2.W))
    val dataIn = Input(UInt(32.W))
    val result = Output(UInt(32.W))
  })

  val word = Reg(UInt(32.W))
  val counter = RegInit(UInt(6.W), 0.U)

  word := Mux(io.optn === 1.U, io.dataIn, word)
  counter :=  Mux(io.optn === 2.U,
   32.U, Mux(counter > 0.U, counter-1.U, 0.U))
  io.result :=  RegNext(Mux(io.optn === 2.U, 0.U,
  (Mux(counter > 0.U, io.result+word(counter-1.U), io.result))))
}
```

Opcode, Registers, Latency

```
"Instructions": [
  {
    "name": "countOnes",
    "latency": 33,
    "signalEncoding": {
      "BitCountAndStore.optn": 2
    }
  }
],
"Neutral|NopState": {
  "BitCountAndStore.optn": 0
},
"HardwareRegisters": [
  "BitCountAndStore.optn",
  "BitCountAndStore.word"
]
```

CIRCT MLIR Dialects

```
hw.module @CountPositiveBits(%clock: i1, %reset: i1, %io_optn: i2, %io_dataIn: i32) -> (io_result: i32) {
  %c0_i31 = hw.constant 0 : i31
  %c-1_i6 = hw.constant -1 : i6
  %c1_i2 = hw.constant 1 : i2
  %c0_i26 = hw.constant 0 : i26
  %c-2_i2 = hw.constant -2 : i2
  %c-32_i6 = hw.constant -32 : i6
  %c0_i6 = hw.constant 0 : i6
  %c0_i32 = hw.constant 0 : i32
  %word = seq.firreg %1 clock %clock {firrtl.random_init_start = 0 : ui64} : i32
  %counter = seq.firreg %6 clock %clock reset sync %reset, %c0_i6 {firrtl.random_init_start = 32 : ui64} : i6
  %0 = comb.icmp bin eq %io_optn, %c1_i2 {sv.namehint = "_word_T"} : i2
  %1 = comb.mux bin %0, %io_dataIn, %word {sv.namehint = "_word_T_1"} : i32
  ...
```

State Transition Function (LLVM IR)

```
; ModuleID = 'countPositiveBits'

define i32 @countPositiveBits(ptr nocapture readonly %0) local_unnamed_addr #0 {
  %2 = load i32, ptr %0, align 4
  %3 = lshr i32 %2, 1
  %.lobit = and i32 %3, 1
  %4 = lshr i32 %2, 2
  %.lobit1 = and i32 %4, 1
  %5 = lshr i32 %2, 3
  %.lobit2 = and i32 %5, 1
  ; ... ETC
  %32 = lshr i32 %2, 30
  %.lobit29 = and i32 %32, 1
  %.lobit30 = lshr i32 %2, 31
  %33 = and i32 %2, 1
  %34 = add nuw nsw i32 %33, %.lobit30
  %35 = add nuw nsw i32 %34, %.lobit29
  ; ... ETC
  %62 = add nuw nsw i32 %61, %.lobit2
  %63 = add nuw nsw i32 %62, %.lobit1
  %64 = add nuw nsw i32 %63, %.lobit
  ret i32 %64
}
```