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.

**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`, `sclang` ...

2. **Constant assignment & folding**
   - Registers are set to capture the state of an instruction in cycle N of execution
   - The state of the design is captured and fed to the registers in the next cycle

3. **State capture & propagation**
   - The state transition function is captured and fed to the registers in the next cycle
   - The unrolled design is lowered to LLVM IR and optimised using standard passes

4. **Lower to LLVM**
   - The unrolled design is lowered to LLVM IR and optimised using standard passes

**Example**

**Hardware Description (Chisel)**

```chisel
module module_1(input clk, output [1:0] output);  // Assume some module logic
  wire [1:0] input;  // Input signal
  reg [1:0] output;  // Output signal
  always @(posedge clk) begin
    if (input == 1) begin
      output = 2;  // Change output based on input
    end
  end
endmodule
```

**Opcode, Registers, Latency**

- **Opcode**: `load`, `store`
- **Registers**: `reg_out`, `reg_in`
- **Latency**: 1 cycle

**State Transition Function (LLVM IR)**

```llvm
define i32 load(i32, i32, i32) local_unnamed_addr #0 (
  %2 = add %1, %3, %4
)
```

**CIRCT MLIR Dialects**

- Input languages
  - Chisel
  - SystemVerilog
  - Python

- Core Dialects
  - State Transition Function
  - LLVM IR