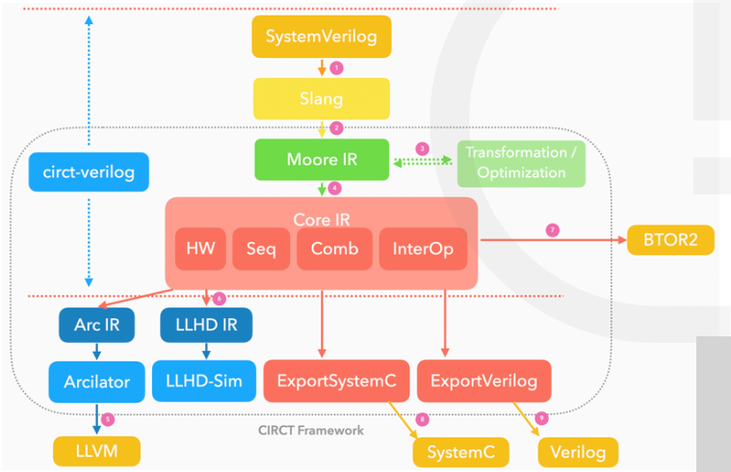
Propelling SystemVerilog into the Future : A Leap Towards CIRCT Ecosystem Integration

Hailong Sun, Fabian Schuiki, Martin Erhart,
Buyun Xu, Zuoye Shi and Anqi Yu

Overview

CIRCT stands as an formidable MLIR initiative, centered on the domain of hardware design. It supports diverse input formats such as FIRRTL, SystemVerilog, and other MLIR variants, catering to the demand for a standardized intermediate representation within the ever-advancing sphere of hardware design. The newly introduced Moore dialect endeavors to accurately encapsulate the complete SystemVerilog types and semantics produced by the Slang front-end compiler. Moreover, it serves as a medium for transformation passes, facilitating the resolution of linguistic peculiarities, enabling high-level design analysis, and ultimately translating the design into the core dialects.



The numerical indicators on the graph represent the transition from the higher-level language compilation phase to the subsequent lower-level instructions. Certain details have been excluded from the present framework for brevity.

Compilation Flow

I. Generate AST from SystemVerilog Code by Slang

The slang compiler provides an incredible ability to do parsing and lexing, which generates stable syntax trees for subsequent lowerings.

```
module LocalVar;
  int x, y, z;
  always_comb begin
    int a;
    a = x + 1;
    z = a + a;
    a += 1;
    y = a;
  end
endmodule
```

```
.....
"name": "LocalVar",
"kind": "Instance",
"addr": 140444961080432,
"body": {
  "name": "LocalVar",
  "kind": "InstanceBody",
  "addr": 140444961078560,
  "members": [
    {
      "name": "x",
      "kind": "Variable",
      "addr": 140444961078736,
      "type": "int",
      "lifetime": "Static"
.....
```

II. ImportVerilog pass in CIRCT

The primary objective of this pass is to maximize the processing of diverse tokens within the AST by adopting a greedy approach. This method makes it feasible to generate a Moore Intermediate Representation (IR).

```
.....
always @(posedge clk) begin
  q <= ~in;
end
.....
```

```
.....
moore.procedure always {
  %1 = moore.read %clk_0 : i1
  moore.wait_event posedge %1 : i1
  %2 = moore.read %in_1 : i1
  %3 = moore.not %2 : i1
  moore.nonblocking_assign %q, %3 : i1
}
.....
```

```
moore.module @LocalVar() {
  %x = moore.variable : <i32>
  %y = moore.variable : <i32>
  %z = moore.variable : <i32>
  moore.procedure always_comb {
    %a = moore.variable : <i32>
    %0 = moore.read %x : i32
    %1 = moore.constant 1 : i32
    %2 = moore.add %0, %1 : i32
    moore.blocking_assign %a, %2 : i32
    %3 = moore.read %a : i32
    moore.blocking_assign %z, %3 : i32
    %4 = moore.read %a : i32
    %5 = moore.constant 1 : i32
    %6 = moore.add %4, %5 : i32
    moore.blocking_assign %a, %6 : i32
    %7 = moore.read %a : i32
    moore.blocking_assign %y, %7 : i32
  }
  moore.output
}
```

III. Transformation and Optimization of the Moore Layer

The transformation and optimization passes implemented within the Moore layer play a critical role in disassembling high-level language features into their corresponding lower-level components.

Take an example for optimizing local variable and inline contents in the always_comb procedure:

```
int x, y, z;
always_comb begin
  automatic int a;
  a = x + 1;
  y = a;
  a = a + 1;
  z = a;
end
```

```
int x, y, z;
y = x + 1;
z = (x + 1) + 1;
```

As depicted below, we have implemented interfaces that differentiate operations based on whether they entail side effects, in order to invoke the mem2reg pass for the elimination of superfluous local variables. Subsequent to this optimization, inlining can be effectively executed.

```
moore.module @LocalVar() {
  %x = moore.variable : <i32>
  %y = moore.variable : <i32>
  %z = moore.variable : <i32>
  moore.procedure always_comb {
    %a = moore.variable : <i32>
    %0 = moore.read %x : i32
    %1 = moore.constant 1 : i32
    %2 = moore.add %0, %1 : i32
    moore.blocking_assign %a, %2 : i32
    %3 = moore.read %a : i32
    moore.blocking_assign %z, %3 : i32
    %4 = moore.read %a : i32
    %5 = moore.constant 1 : i32
    %6 = moore.add %4, %5 : i32
    moore.blocking_assign %a, %6 : i32
    %7 = moore.read %a : i32
    moore.blocking_assign %y, %7 : i32}
  moore.output
```

```
moore.module @LocalVar() {
  %x = moore.variable : <i32>
  %y = moore.variable : <i32>
  %z = moore.variable : <i32>
  %0 = moore.read %x : i32
  %1 = moore.constant 1 : i32
  %2 = moore.add %0, %1 : i32
  moore.blocking_assign %z, %2 : i32
  %3 = moore.constant 1 : i32
  %4 = moore.add %2, %3 : i32
  moore.blocking_assign %y, %4 : i32
  moore.output
```

IV & V. Convert to Core IR and utilize Arcilator to generate LLVM code.

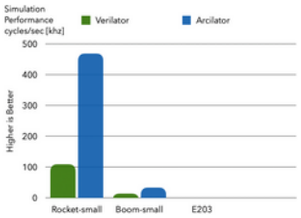
The core IR within the CIRCT framework has been subdivided into discrete components. The HW dialect encapsulates the structural aspects of hardware design, while the Seq and Comb dialects specifically denote the sequential and combinational elements, respectively, as indicated by their respective nomenclature.

```
hw.module @LocalVar() {
  %c0_i32 = hw.constant 0 : i32
  %x = hw.wire %c0_i32 : i32
  %z = hw.wire %1 : i32
  %y = hw.wire %2 : i32
  %c1_i32 = hw.constant 1 : i32
  %0 = comb.add %x, %c1_i32 : i32
  %1 = comb.add %0, %0 : i32
  %c1_i32_0 = hw.constant 1 : i32
  %2 = comb.add %0, %c1_i32_0 : i32
  hw.output
}
```

```
; ModuleID = 'LLVMDialectModule'
source_filename = "LLVMDialectModule"

define void @LocalVar_eval(ptr %0) {
  ret void
}
llvm.module.flags = !{!0}
!0 = !{i32 2, !"Debug Info Version", i32 3}
```

Leverage the new cycle-accurate hardware simulator, Arcilator, for the generation of LLVM Intermediate Representation (IR). Our team has successfully transitioned the hardware code into a software model. The subsequent figure delineates the performance metrics of various projects emulated utilizing Arcilator, encompassing data derived from Verilator tests.



Eco-Applications

VI & VII. Do Event-driven simulation or verification

Generate LLHD IR and perform event-driven simulation with llhd-sim or produce btor2 code for design verification using btor2sim.

```
1 sort bitvec 32
2 constd 1 0
3 constd 1 1
4 add 1 2 3
5 add 1 4 4
6 constd 1 1
7 add 1 4 6
```

```
llhd.entity @LocalVar () -> () {
  %c0_i32 = hw.constant 0 : i32
  %c1_i32 = hw.constant 1 : i32
  %x = hw.wire %c0_i32 : i32
  %0 = comb.add %x, %c1_i32 : i32
  %1 = comb.add %0, %0 : i32
  %c1_i32_0 = hw.constant 1 : i32
  %2 = comb.add %0, %c1_i32_0 : i32
  %z = hw.wire %1 : i32
  %y = hw.wire %2 : i32}
```

VIII & IX. Generate SystemC code or .....

The exportSystemC pass generates systemC code for implementation, and the compilation route even supports converting Verilog to Verilog.

```
emitc.include <"systemc.h">
systemc.module @LocalVar () {
  systemc.ctor {
    systemc.method %innerLogic
  }
  %innerLogic = systemc.func {
    %c0_i32 = hw.constant 0 : i32
    %c1_i32 = hw.constant 1 : i32
    %0 = comb.add %c0_i32, %c1_i32 : i32
    %1 = comb.add %0, %0 : i32
    %c1_i32_0 = hw.constant 1 : i32
    %2 = comb.add %0, %c1_i32_0 : i32}
}
```

```
module LocalVar();
  wire [31:0] x = 32'h0;
  wire [31:0] _GEN = x + 32'h1;
  wire [31:0] z = _GEN + _GEN;
  wire [31:0] y = _GEN + 32'h1;
endmodule
```

Future Plan

The aforementioned features are currently available in the circt upstream branch or are pending integration. Ongoing advancements and optimizations within the compiler are scheduled for subsequent updates, including:

- Builtin function and task support
- Further enhancements to the Moore IR
- Improved support for the circt-verilog utility
- Enhanced testing for the SystemVerilog compilation process
- Upgrade slang version that CIRCT depends on

Our team is adapting other SystemVerilog projects to evaluate and monitor the progress of our developments. We extend an invitation to all parties with an interest in these endeavors to participate actively!