CIRCT

FSM (Finite-State Machine) Dialect

Hanchen Ye (PhD Student from UIUC)
Advisor: Andrew Lenharth

Aug. 19, 2021

Outline

- Motivation: Abstract FSMs in a higher level
- Background: MLIR and CIRCT (FIRRTL TNG)
- Operations of FSM Dialect
- Lower FSM to HW/SW
- Takeaways

Outline

- **Motivation: Abstract FSMs in a higher level**
- **Background: MLIR and CIRCT (FIRRTL TNG)**
- **Operations of FSM Dialect**
- **Lower FSM to HW/SW**
- **Takeaways**

Motivation: Abstract FSMs in a higher level

- **Aspects of many domains can be described as the form of FSM:**
  - **Hardware**: IP block control, Verification, Interface control, etc.
  - **Software**: Driver, Firmware, System control, etc.
  - **Protocol**: Networks, Distributed System, etc.

- **How to represent these FSMs in HW/SW design flows? Taking FIRRTL as example:**
  - FIRRTL is an amazing abstraction or IR of digital circuits
  - It's not straightforward to recognize, analyze, and transform FSMs in FIRRTL
  - An explicit FSM abstraction can help to simplify circuits analysis, generate verifications in a higher level, infer the control sequences of changing power state, etc.

- **Represent FSMs with an explicit and structural abstraction!**

Motivation: Abstract FSMs in a higher level

- **Some desired features of the FSM abstraction:**
  - Easy to analyze and transform
  - Target-agnostic:
    - Can be lowered to targets in different domains
  - Can be attached or integrated into existing abstractions
- **-> Design an FSM dialect in MLIR**

Background: MLIR and CIRCT (FIRRTL TNG)

MLIR: A Compiler Infrastructure for the End of Moore's Law

Chris Lattner * Google
Mehdi Amini Google
Uday Bondhugula IISc
Albert Cohen Google
Andy Davis Google

Jacques Pienaar Google
River Riddle Google
Tatiana Shpeisman Google
Nicolas Vasilache Google

Oleksandr Zinenko Google

Abstract

This work presents MLIR, a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

MLIR facilitates the design and implementation of code generators, translators and optimizers at different levels of abstraction and also across application domains, hardware targets and execution environments. The contribution of this work includes (1) discussion of MLIR as a research artifact, built for extension and evolution, and identifying the challenges and opportunities posed by this novel design point in design, semantics, optimization specification, system, and engineering. (2) evaluation of MLIR as a generalized infrastructure that reduces the cost of building compilers—describing diverse use-cases to show research and educational opportunities for future programming languages, compilers, execution environments, and computer architecture. The paper also presents the rationale for MLIR, its original design principles, structures and semantics.

https://mlir.llvm.org/

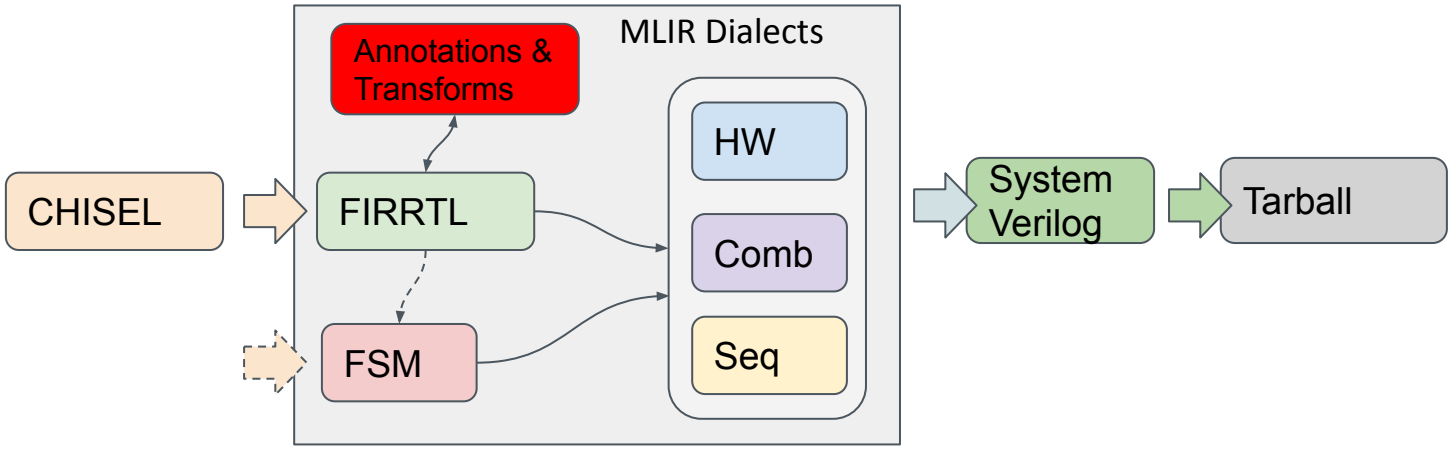
What is MLIR?

- *"Multi-level* IR"
- Part of the LLVM project
- Infrastructure for building new compilers
- Suitable for building new IRs
- Can handle *multiple* IRs at once
- Written in C++
- Extremely performant

Dialect: A C++ namespace in MLIR to define customized types, operations, attributes, etc.

Source: The Next Generation FIRRTL Compiler, Schuyler Eldridge, CCCC 2021.

Background: MLIR and CIRCT (FIRRTL TNG)



https://circt.llvm.org/

What is CIRCT (pronounced "circuit")?

- **An LLVM incubator project**
- **Compiler infrastructure for generating Verilog**
- **A collection of hardware dialects**
  - FIRRTL Dialect
  - SystemVerilog Dialect
  - RTL Dialects: Combinational, Sequential, Hardware
  - Timing Insensitive Dialects: Elastic Silicon Interconnect (ESI), Handshake
- **Eventually, a target for MLIR/LLVM**
- **A collection of tools for compiling circuits**

Source: The Next Generation FIRRTL Compiler, Schuyler Eldridge, CCCC 2021.

Outline

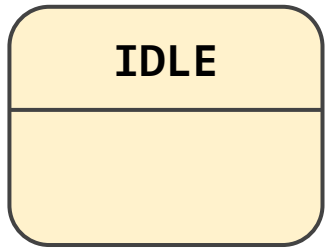
- Motivation: Abstract FSMs in a higher level
- Background: MLIR and CIRCT (FIRRTL TNG)
- **Operations of FSM Dialect**
- Lower FSM to HW/SW
- Takeaways

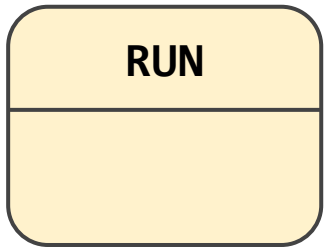
Operations of FSM Dialect

- **fsm.machine, fsm.state, and fsm.variable**

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR





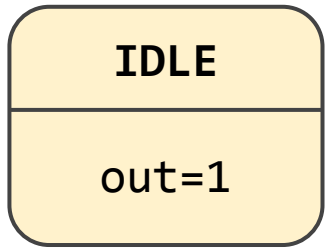
- fsm.machine includes a list of inputs and outputs and explicit state type
- fsm.state has a symbol name
- fsm.variable is “extended state” introduced to avoid state explosion

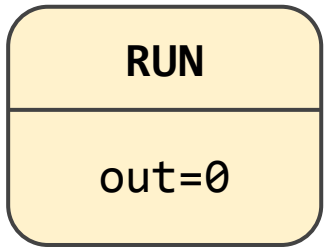
Operations of FSM Dialect

- **fsm.output and the output region of fsm.state**

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR





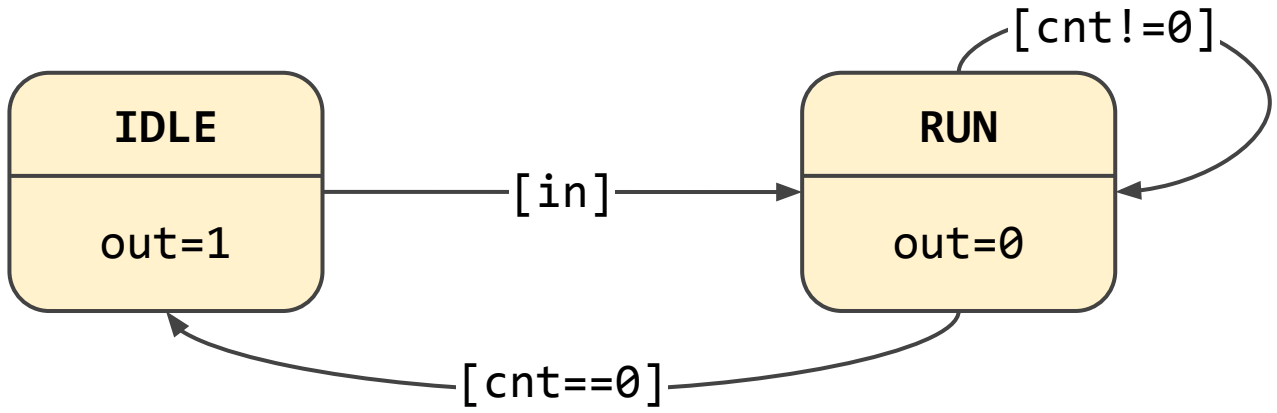
- The operand types of fsm.output should align with the output types of it parent fsm.machine
- Each fsm.state must have an output region with legal fsm.output as terminator

Operations of FSM Dialect

- **fsm.transition with guard region**

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR



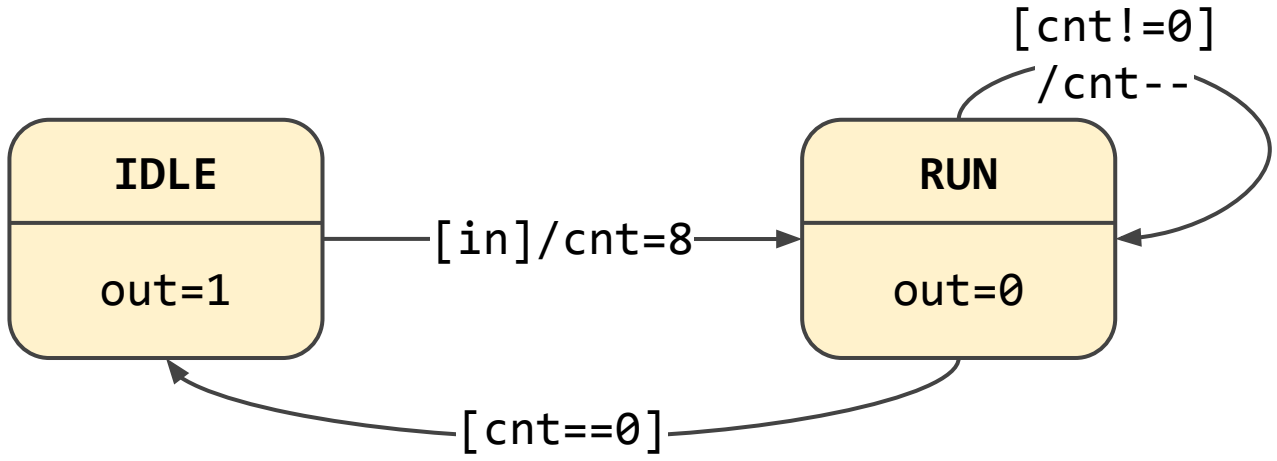
- fsm.transition has a symbol reference to the name of the next state
- The returned value of the guard region indicates whether a transition is taken
- If there are more than one transitions in one state, they are evaluated in a top-down order

Operations of FSM Dialect

- **fsm.update and the action region of fsm.transition**

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR



– fsm.update should only appear in action regions

Operations of FSM Dialect

- **Two ways of instantiation: fsm.hw_instance or fsm.instance + fsm.trigger**

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR

```
// Hardware-style instantiation.
hw.module @bar() {
 %in = hw.constant true
 %out = fsm.hw_instance "foo_inst" @foo(%in) : (i1) -> i1
}

// Software-style instantiation and triggering.
func @qux() {
 %foo_inst = fsm.instance "foo_inst" @foo

 %in0 = constant true
 %out0 = fsm.trigger %foo_inst(%in0) : (i1) -> i1

 %in1 = constant false
 %out1 = fsm.trigger %foo_inst(%in1) : (i1) -> i1
 return
}
```

FSM IR

Outline

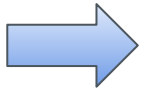
- Motivation: Abstract FSMs in a higher level
- Background: MLIR and CIRCT (FIRRTL TNG)
- Operations of FSM Dialect
- **Lower FSM to HW/SW**
- Takeaways

Lower FSM to HW/SW

- Lower to hardware and emit Verilog

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR



```
module foo(
 input  in0, clk, rst_n,
 output out0);
 reg [1:0]  state;     // <stdin>:3:14
 reg [15:0] cnt;       // <stdin>:4:12
 reg        w_out0;    // <stdin>:46:15

 always @(posedge clk or negedge rst_n) begin  // <stdin>:5:5
   if (!rst_n) begin   // <stdin>:5:5
     state <= 2'h0;    // <stdin>:42:16, :43:7
     cnt <= 16'h0;     // <stdin>:41:17, :44:7
   end
   else begin  // <stdin>:5:5
     state <= state;   // <stdin>:7:12, :8:7
     cnt <= cnt;       // <stdin>:6:12, :9:7
     casez (state)     // <stdin>:7:12, :10:7
       2'b00: begin
         if (in0) begin        // <stdin>:12:9
           state <= 2'h1;      // <stdin>:13:20, :15:11
           cnt <= 16'h100;     // <stdin>:14:23, :16:11
         end
       end
       2'b01: begin
         ... ...
       end
     endcase   // <stdin>:7:12, :10:7
   end
 end // always @(posedge or negedge)
 always_comb begin     // <stdin>:47:5
   w_out0 = 1'h0;      // <stdin>:49:16, :50:7
   casez (state)       // <stdin>:48:12, :51:7
     2'b00:
       w_out0 = 1'h1;  // <stdin>:53:17, :54:9
     2'b01:
       w_out0 = 1'h0;  // <stdin>:57:20, :58:9
   endcase     // <stdin>:48:12, :51:7
 end // always_comb
 assign out0 = w_out0; // <stdin>:61:10, :62:5
endmodule
```

SV File

Lower FSM to HW/SW

- **Lower to hardware and emit Verilog**
  - Lowered to HW+Comb+SV
  - Two-always block coding style and binary state encoding style, can be extended to support options of different coding style
  - fsm.variable converted to register
  - Simulate with Verilator, etc.

```
module foo(
 input  in0, clk, rst_n,
 output out0);
 reg [1:0]  state;     // <stdin>:3:14
 reg [15:0] cnt;       // <stdin>:4:12
 reg        w_out0;    // <stdin>:46:15

 always @(posedge clk or negedge rst_n) begin  // <stdin>:5:5
   if (!rst_n) begin   // <stdin>:5:5
     state <= 2'h0;    // <stdin>:42:16, :43:7
     cnt <= 16'h0;     // <stdin>:41:17, :44:7
   end
   else begin  // <stdin>:5:5
     state <= state;   // <stdin>:7:12, :8:7
     cnt <= cnt;       // <stdin>:6:12, :9:7
     casez (state)     // <stdin>:7:12, :10:7
       2'b00: begin
         if (in0) begin        // <stdin>:12:9
           state <= 2'h1;      // <stdin>:13:20, :15:11
           cnt <= 16'h100;     // <stdin>:14:23, :16:11
         end
       end
       2'b01: begin
         ... ...
       end
     endcase   // <stdin>:7:12, :10:7
   end
 end // always @(posedge or negedge)
 always_comb begin     // <stdin>:47:5
   w_out0 = 1'h0;      // <stdin>:49:16, :50:7
   casez (state)       // <stdin>:48:12, :51:7
     2'b00:
       w_out0 = 1'h1;  // <stdin>:53:17, :54:9
     2'b01:
       w_out0 = 1'h0;  // <stdin>:57:20, :58:9
   endcase     // <stdin>:48:12, :51:7
 end // always_comb
 assign out0 = w_out0; // <stdin>:61:10, :62:5
endmodule
```

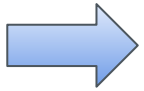
SV File

Lower FSM to HW/SW

- Lower to software

```
fsm.machine @foo(%in: i1) -> i1
attributes {stateType = i1} {
 %cnt = fsm.variable "cnt" : i16
 fsm.state "IDLE" output  {
   %true = constant true
   fsm.output %true : i1
 } transitions  {
   fsm.transition @RUN guard  {
     fsm.return %in : i1
   } action  {
     %c256_i16 = constant 8 : i16
     fsm.update %cnt, %c256_i16 : i16
   }
 }
 fsm.state "RUN" output  {
   %false = constant false
   fsm.output %false : i1
 } transitions  {
   fsm.transition @RUN ...
   fsm.transition @IDLE ...
 }
}
```

FSM IR



```
func @foo(%arg0: i1, %arg1: memref<i16>, %arg2: memref<i2>) -> i1 {
 %false = constant false
 %true = constant true
 %c1_i16 = constant 1 : i16
 %c0_i16 = constant 0 : i16
 %c256_i16 = constant 256 : i16
 %c1_i2 = constant 1 : i2
 %c0_i2 = constant 0 : i2
 %0 = memref.load %arg1[] : memref<i16>
 %1 = memref.load %arg2[] : memref<i2>
 %2 = cmpi eq, %1, %c0_i2 : i2
 scf.if %2 {
   scf.if %arg0 {
     memref.store %c1_i2, %arg2[] : memref<i2>
     memref.store %c256_i16, %arg1[] : memref<i16>
   }
 } else {
   ... ...
 }
 %3 = memref.load %arg2[] : memref<i2>
 %4 = cmpi eq, %3, %c0_i2 : i2
 %5 = select %4, %true, %false : i1
 return %5 : i1
}

func @qux() {
 %false = constant false
 %true = constant true
 %c0_i2 = constant 0 : i2
 %c0_i16 = constant 0 : i16
 %0 = memref.alloca() : memref<i16>
 memref.store %c0_i16, %0[] : memref<i16>
 %1 = memref.alloca() : memref<i2>
 memref.store %c0_i2, %1[] : memref<i2>
 %2 = call @foo(%true, %0, %1) : (i1, memref<i16>, memref<i2>) -> i1
 %3 = call @foo(%false, %0, %1) : (i1, memref<i16>, memref<i2>) -> i1
 return
}
```

Software IR

Lower FSM to HW/SW

- **Lower to software**
  - Lowered to Standard+MemRef+SCF (Structured Control Flow)
  - fsm.variable converted to alloca and lives in the caller function
  - fsm.trigger converted to function call
  - Can be further lowered to LLVM and simulate with mlir-runner, etc.

```
func @foo(%arg0: i1, %arg1: memref<i16>, %arg2: memref<i2>) -> i1 {
 %false = constant false
 %true = constant true
 %c1_i16 = constant 1 : i16
 %c0_i16 = constant 0 : i16
 %c256_i16 = constant 256 : i16
 %c1_i2 = constant 1 : i2
 %c0_i2 = constant 0 : i2
 %0 = memref.load %arg1[] : memref<i16>
 %1 = memref.load %arg2[] : memref<i2>
 %2 = cmpi eq, %1, %c0_i2 : i2
 scf.if %2 {
   scf.if %arg0 {
     memref.store %c1_i2, %arg2[] : memref<i2>
     memref.store %c256_i16, %arg1[] : memref<i16>
   }
 } else {
   ... ...
 }
 %3 = memref.load %arg2[] : memref<i2>
 %4 = cmpi eq, %3, %c0_i2 : i2
 %5 = select %4, %true, %false : i1
 return %5 : i1
}

func @qux() {
 %false = constant false
 %true = constant true
 %c0_i2 = constant 0 : i2
 %c0_i16 = constant 0 : i16
 %0 = memref.alloca() : memref<i16>
 memref.store %c0_i16, %0[] : memref<i16>
 %1 = memref.alloca() : memref<i2>
 memref.store %c0_i2, %1[] : memref<i2>
 %2 = call @foo(%true, %0, %1) : (i1, memref<i16>, memref<i2>) -> i1
 %3 = call @foo(%false, %0, %1) : (i1, memref<i16>, memref<i2>) -> i1
 return
}
```

Software IR

Outline

- Motivation: Abstract FSMs in a higher level
- Background: MLIR and CIRCT (FIRRTL TNG)
- Operations of FSM Dialect
- Lower FSM to HW/SW
- **Takeaways**

Takeaways

- **Many use cases desire a higher level abstraction of FSM**
- **An FSM dialect is designed with the following features:**
  - Provide explicit and structural representations of states, transitions, and internal variables of an FSM, allowing convenient analysis and transformation.
  - Provide a target-agnostic representation of FSM, allowing the state machine to be instantiated and attached to other dialects from different domains.
- **Two lowering passes are implemented to lower FSM to hardware and software**