Formal Verification of Hardware using MLIR

MSc. Thesis - Amelia Dobis (2nd October 2023 - 15th April 2024)

Advisors: Kevin Laeufer (UC Berkeley) & Prof. Zhendong Su (ETH Zürich)

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Overview

1)Creating a Formal Backend for the CIRCT compiler

2)Creating a lowering for SVA properties

3)Verifying the Compiler Passes

4)Conclusion

Part 1: Creating a Formal Back-end for CIRCT

Motivation: Formal Verification

```
class Counter extends Module {
    val count = RegInit(0.U(6.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
}
```

Motivation: Formal Verification

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
}
```

How do we verify this simple design ?

Motivation: Formal Verification

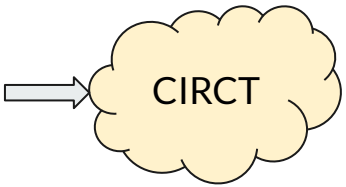
```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
    assert(count < 42.U)
}
```

How do we verify this simple design ?

→ Add a simple assertion + Formal Tools

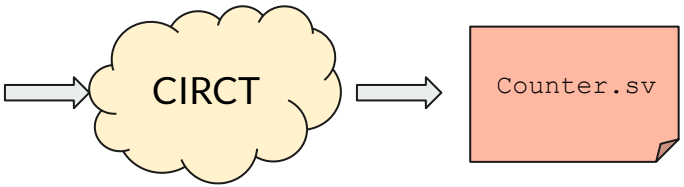
Motivation: Formal Verification

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      assert(count < 42.U)
}
```



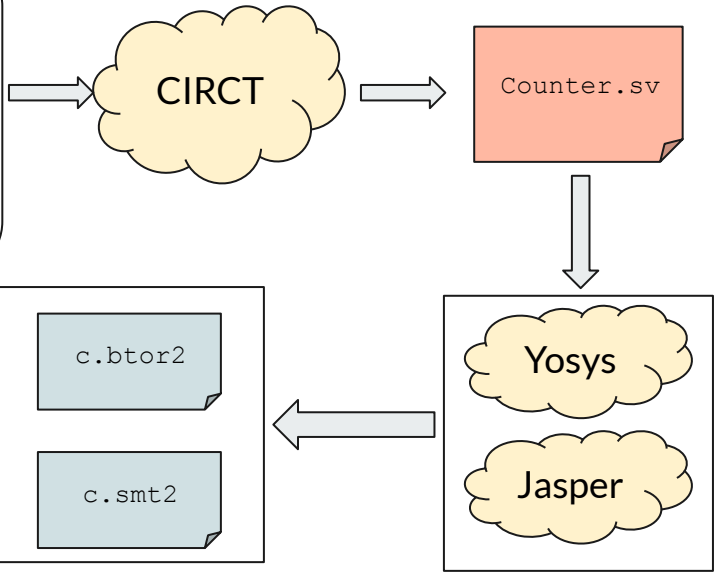
Motivation: Formal Verification

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      assert(count < 42.U)
}
```

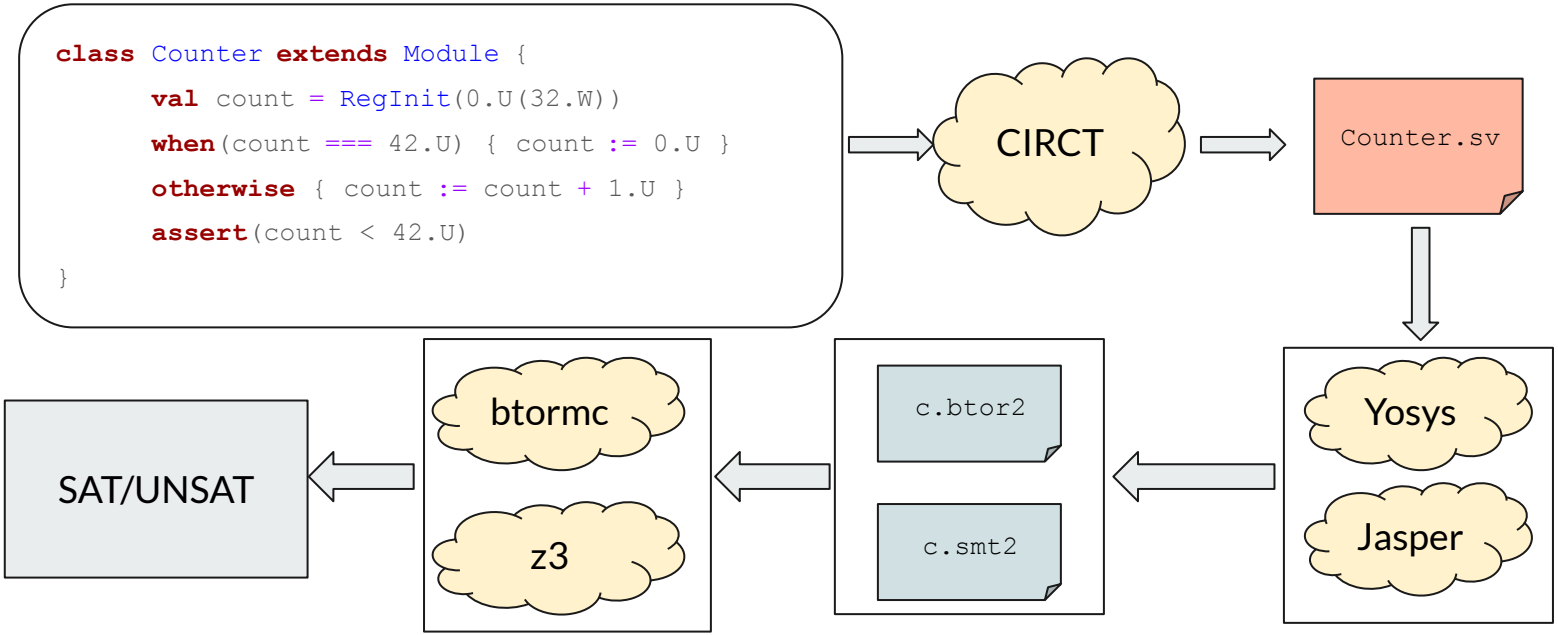


Motivation: Formal Verification

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      assert(count < 42.U)
}
```



Motivation: Formal Verification

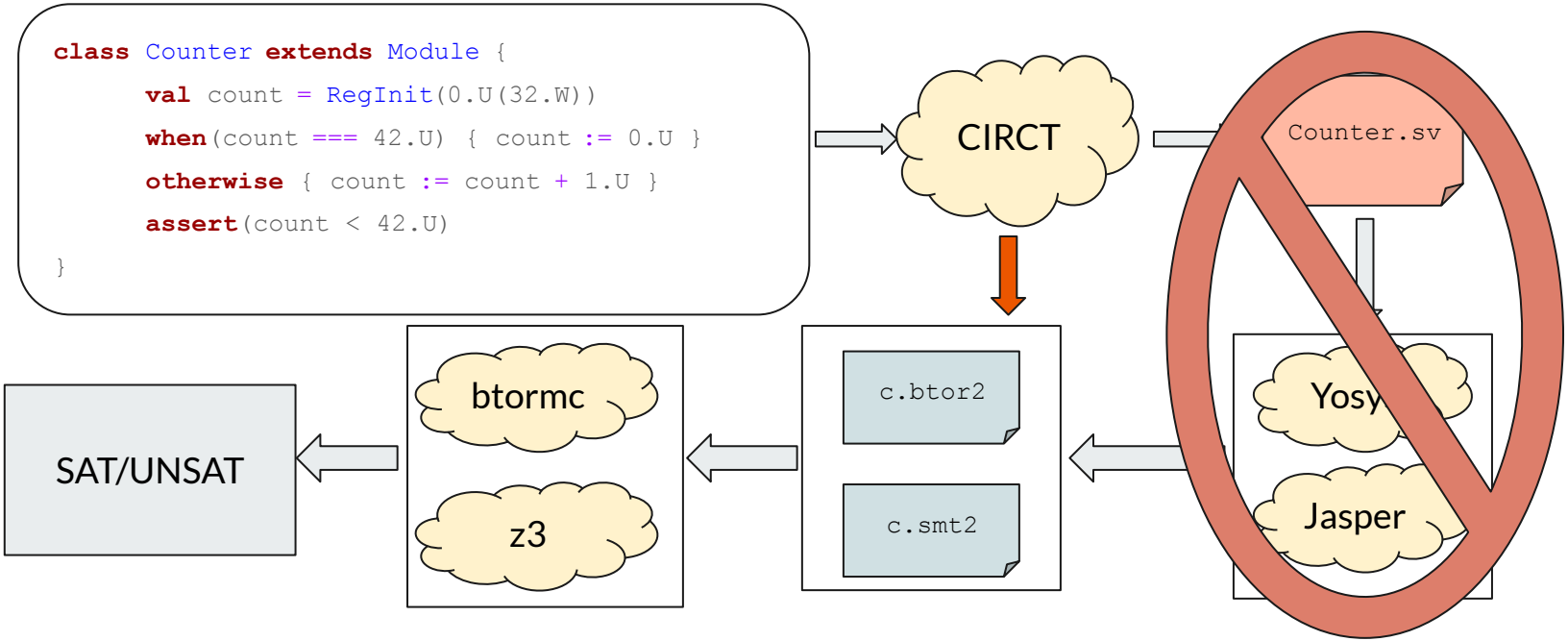


Goal: Unified Verification

- Most existing verification tooling relies on SystemVerilog.
- Formal Verification requires many intermediary steps.
- CIRCT aims to unify compilation for all hardware languages.

→ Can we unify verification for all hardware languages into a single open-source tool?

Motivation: Formal Verification



Background: Sequential vs Combinatorial Logic

Combinatorial Logic: Non-stateful logic that simply immediately transforms inputs into outputs.

```
val a = IO(Input(32.W))
val b = a + 1.U
assert (b > a)
```

Background: Sequential vs Combinatorial Logic

Combinatorial Logic: Non-stateful logic that simply immediately transforms inputs into outputs.

```
val a = IO(Input(32.W))
val b = a + 1.U
assert (b > a)
```

Sequential Logic: **Stateful logic** where the transformations happened **over several clock cycles**.

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      assert(count < 42.U)
}
```

Background: Hardware to SMT Logic

Combinatorial Logic: Non-stateful logic that simply immediately transforms inputs into outputs.

Verifying Combinatorial Circuits: Core of BMC, convert circuit and assertion into an SMT formula.

```
val a = IO(Input(32.W))
val b = a + 1.U
assert (b > a)
```

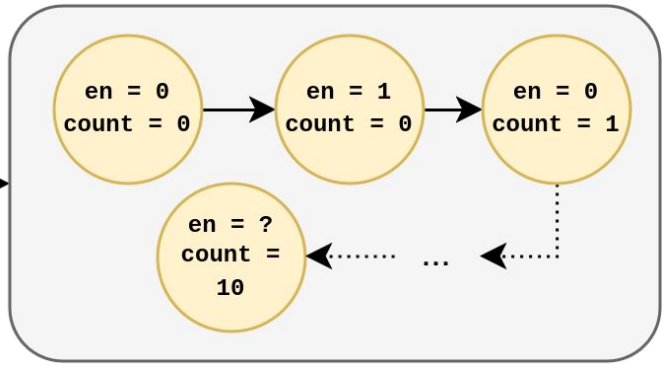
```
(and
      (eq b (add a 1)) // define b
      (not (gt b a)) // can assertion be violated?
)
```

Background: Hardware to SMT Logic

Sequential Logic: Stateful logic where the transformations happened over several clock cycles.

Verifying Sequential Circuits: Convert into state transition system -> bounded model checking

```
class MyCounter extends Module {
    val en = IO(Input(Bool()))
    val count = RegInit(0.U(32.W))
    when(en && count === 22.U) { count := 0.U }
    when(en && count =/= 22.U) { count := count + 1.U }
    assert(count =/= 10.U)
}
```



Background: BTOR2 and btormc

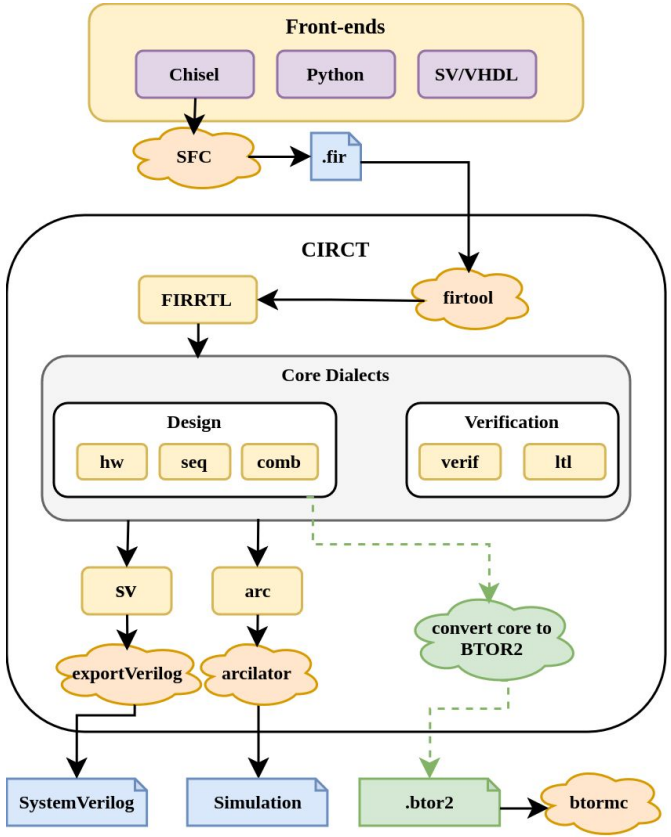
- BTOR2:
  - SMTLib-like format that allows for the explicit encoding of state-transition systems.
  - Supports **bitvector** and **array** theories.
  - No need to manually unroll states, e.g.

```
1 sort bitvector 32 ; declare a 32-bit type
2 state 1 count     ; declare a 32-bit state
3 one 1             ; declare a 32-bit constant of value 1
4 and 1 2 3
5 next 1 2 4        ; count := count & 1
```

- BTORMC:
  - Bounded Model Checker.
  - Supports btor2 format, uses the **boolector** SMT solver.
  - Optimized for solving in bitvector and array theories.

How do I do this in CIRCT?

- Front-ends get converted to the core dialects.
- Core dialects: Generalized representation of hardware.
- Core dialects are the convergence point of all front-ends.
- I can add a **conversion pass** to lower core dialects to btor2.



My BTOR2 Conversion

- Sequential logic: Registers are converted to:
  - **State:** The declaration of the register
  - **Next:** How the state's value transitions across cycles
  - **Init:** Give each state an initial value to avoid useless solver counter-examples

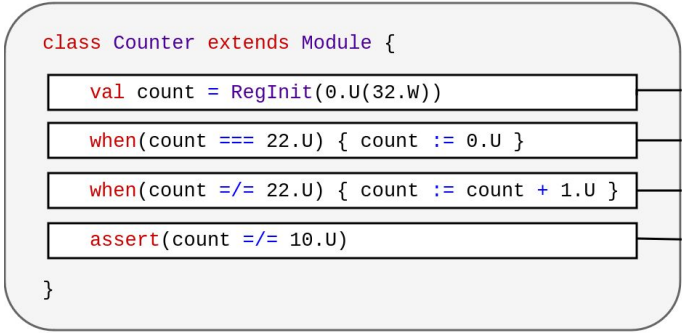
My BTOR2 Conversion

- Sequential logic: Registers are converted to:
  - **State:** The declaration of the register
  - **Next:** How the state's value transitions across cycles
  - **Init:** Give each state an initial value to avoid useless solver counter-examples
- Combinatorial Logic: Core dialects are in a form that is semantically similar to SMT logic, thus the conversion is straightforward.

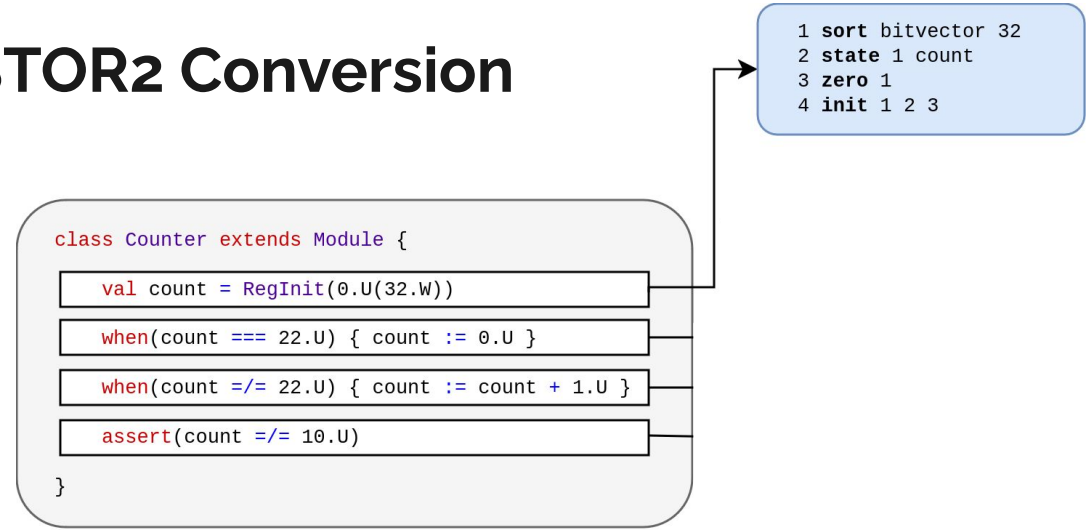
My BTOR2 Conversion

- Sequential logic: Registers are converted to:
  - **State:** The declaration of the register
  - **Next:** How the state's value transitions across cycles
  - **Init:** Give each state an initial value to avoid useless solver counter-examples
- Combinatorial Logic: Core dialects are in a form that is semantically similar to SMT logic, thus the conversion is straightforward.
- Assertions & Assumptions:
  - Assertions: negated then converted to a **bad** instruction.
  - Assumptions: turned into **constraint** instructions.

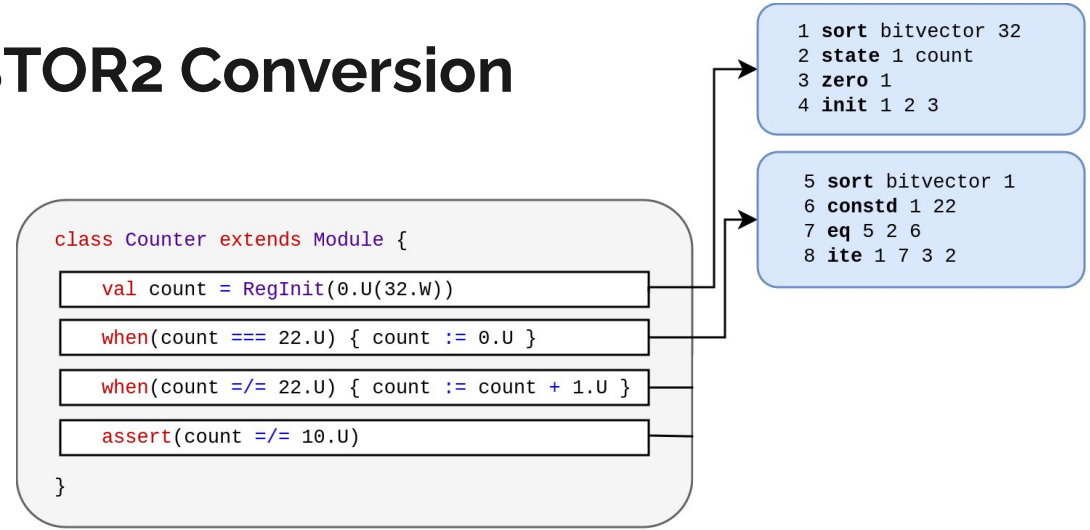
My BTOR2 Conversion



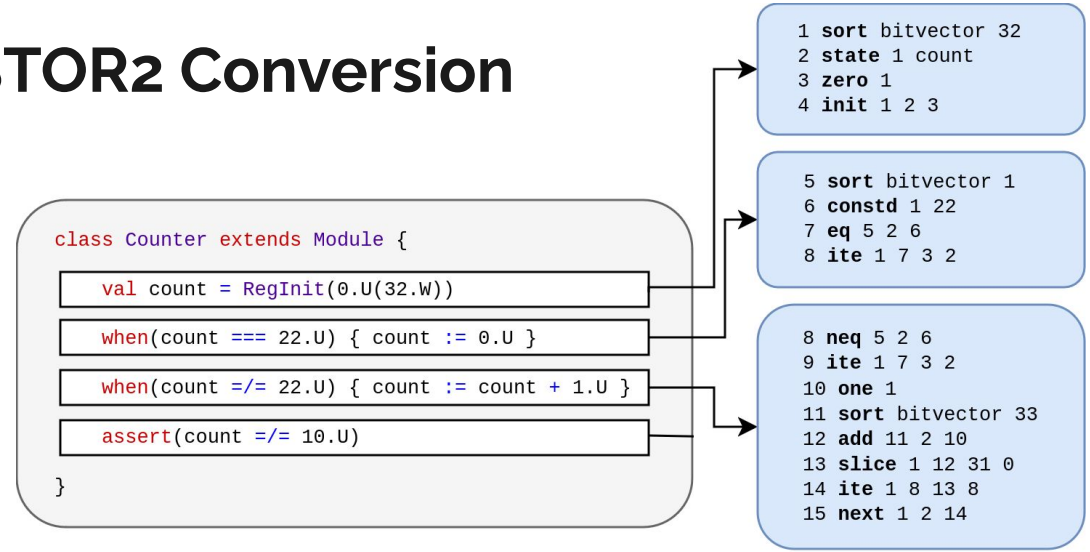
My BTOR2 Conversion



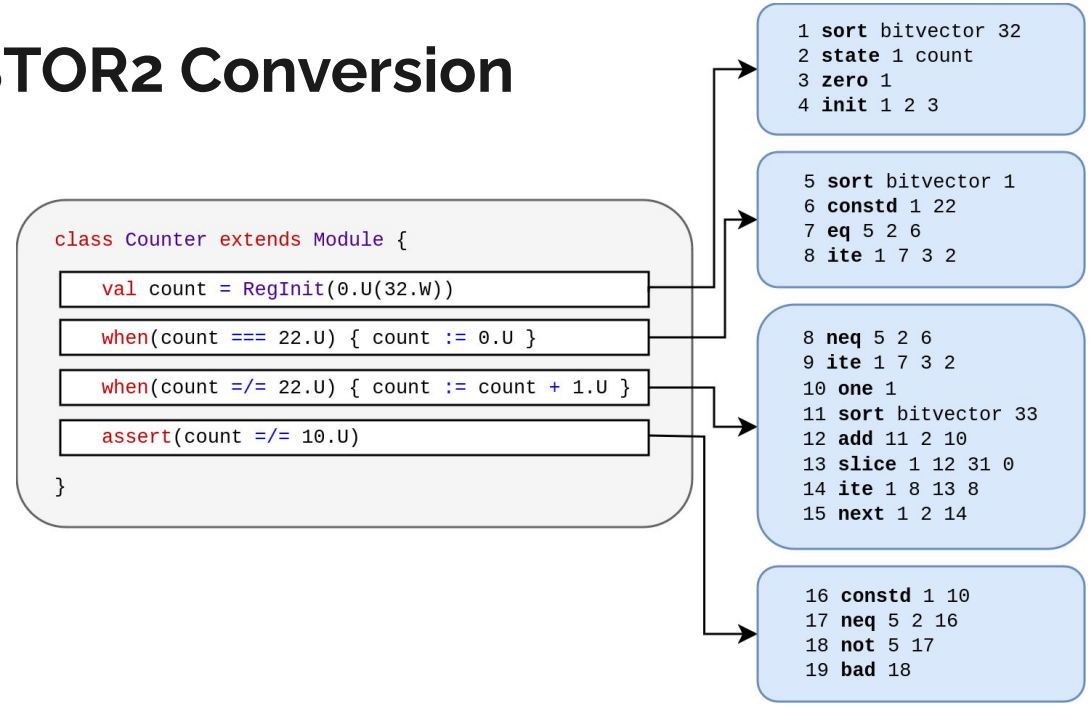
My BTOR2 Conversion



My BTOR2 Conversion



My BTOR2 Conversion

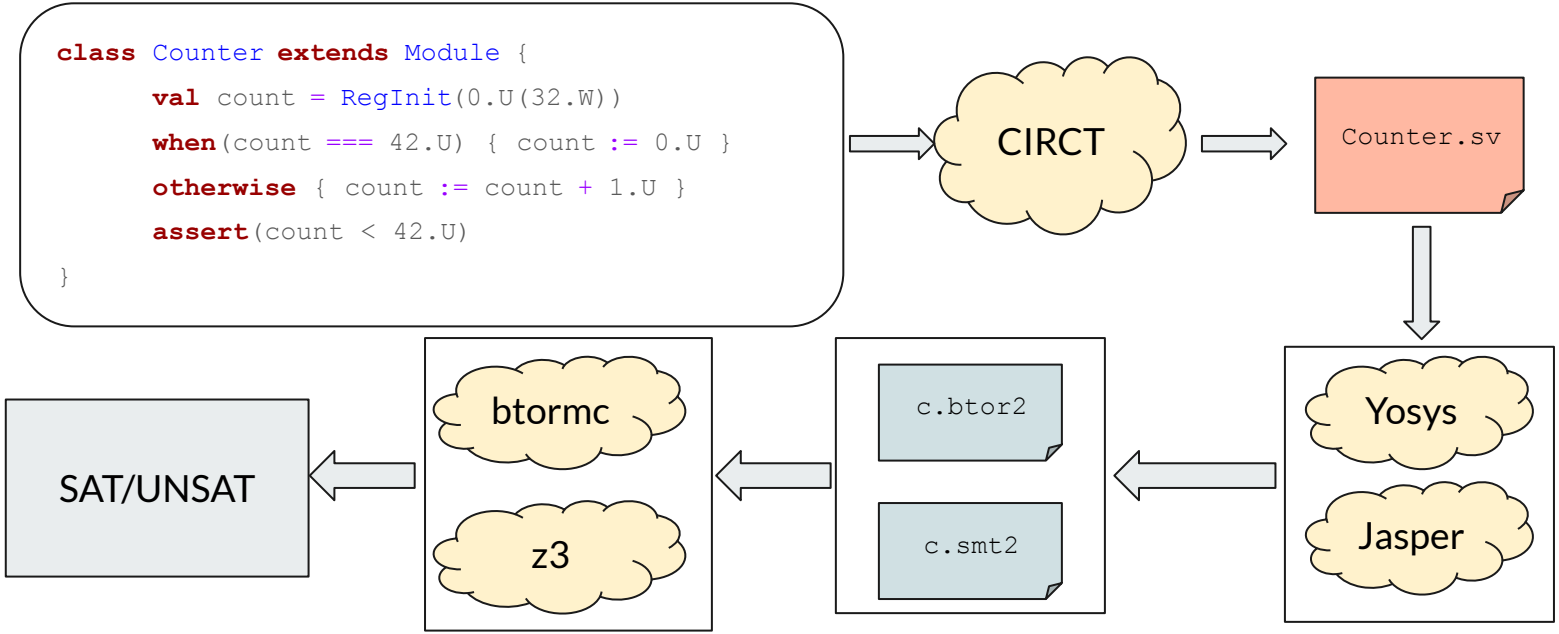


Result

- Fully up-streamed into the CIRCT compiler.
- Formal verification without the use of commercial SV-based tools.
- Fully open-source + single tool experience for user!
- Integrated into **firtool** via the **–btor2** flag.
- Use: **firtool –btor2 counter.fir >> counter.btor2**
- Also works with all other CIRCT front-ends (not just Chisel).

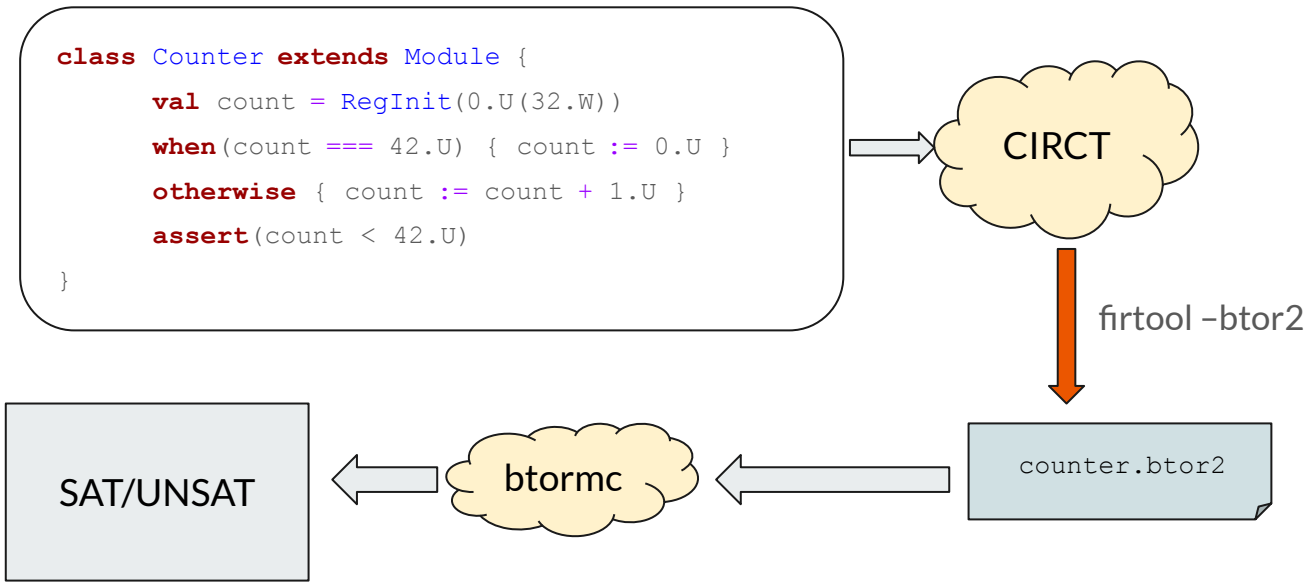
Result: Before

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      assert(count < 42.U)
}
```



Result: After

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      assert(count < 42.U)
}
```



Part 2 : Temporal Specifications in our front-ends

Motivation: Specifying Sequential Circuits

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
}
```

How to write a specification for this simple design ?

Motivation: Specifying Sequential Circuits

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
    assert(count < 42.U)
}
```

How to write a specification for this simple design ?

- Counter never exceeds 42 → `assert(count < 42.U)`

Motivation: Specifying Sequential Circuits

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
}
```

How do write specify this simple design (with assertions) ?

- Counter never exceeds 42 → `assert(count < 42.U)`
- Counter is monotonically increasing → `???`

Motivation: Specifying Sequential Circuits

Counter is monotonically increasing → No good way in Chisel

→ Usually go through SystemVerilog Assertion Properties

1) Compile Design down to SV
2) Modify design to add SVA property assertion:

```
assert property counter < 42 |-> counter > $past(counter)
```

Motivation: Specifying Sequential Circuits

Counter is monotonically increasing → No good way in Chisel

→ Usually go through SystemVerilog Assertion Properties

1) Compile Design down to SV
2) Modify design to add SVA property assertion:

```
assert property counter < 42 |-> counter > $past(counter)
```

Problem: Only supported by commercial SV tools

Challenge: Temporal Specifications

- Expressing temporal relations requires poorly supported SystemVerilog Assertion properties.
- These are mainly supported in commercial tools for SystemVerilog.

→ **How do we support temporal specifications for our open-source formal backend ?**

Background: What should we support?

- SVA Sequences: Defines a series of predicates that should hold.

```
sequence s;
     @(posedge clock) a ##[0:2] b ##1 c;
endsequence;
```

Background: What should we support?

- SVA Sequences: Defines a series of predicates that should hold.

```
sequence s;
    @(posedge clock) a ##[0:2] b ##1 c;
endsequence;
```

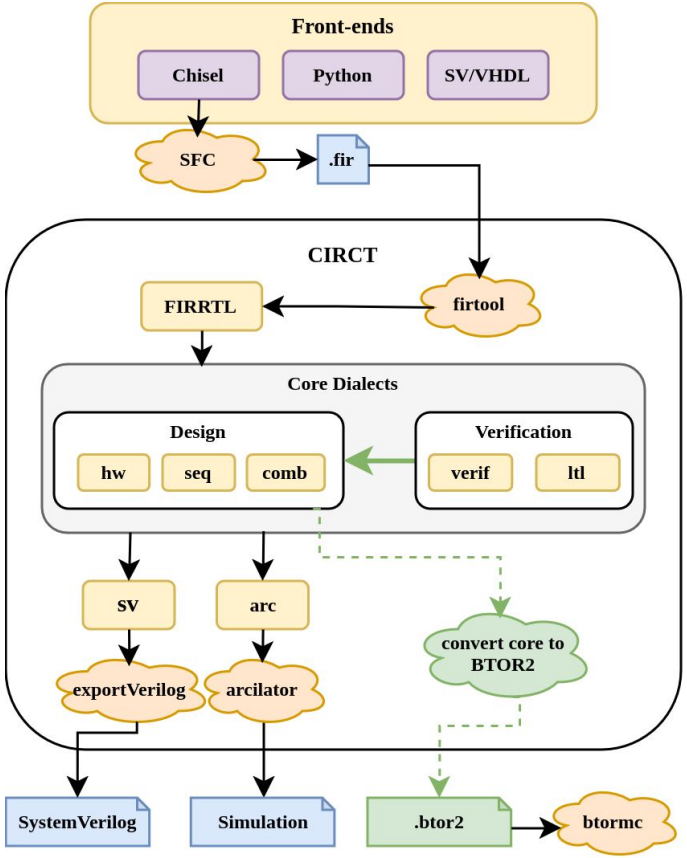
- SVA Property: Encodes a concurrently checked predicate that represents a relation between elements in a design. Can be disabled, can contain sequences.

```
assert property (@(posedge clock)
    disable iff (reset)
    a |=> b  // if a holds then b holds after one cycle
);
```

How can CIRCT support this?

- LTL dialect supports various temporal expressions:
  - `ltl.delay`: delays input by a given number of cycles.
  - `ltl.implication`: encodes an implication.
  - `ltl.concat` : encodes a sequence of events.
  - `ltl.disable`: disables a property on a condition.
  - `ltl.clock`: associates a clock to a property.

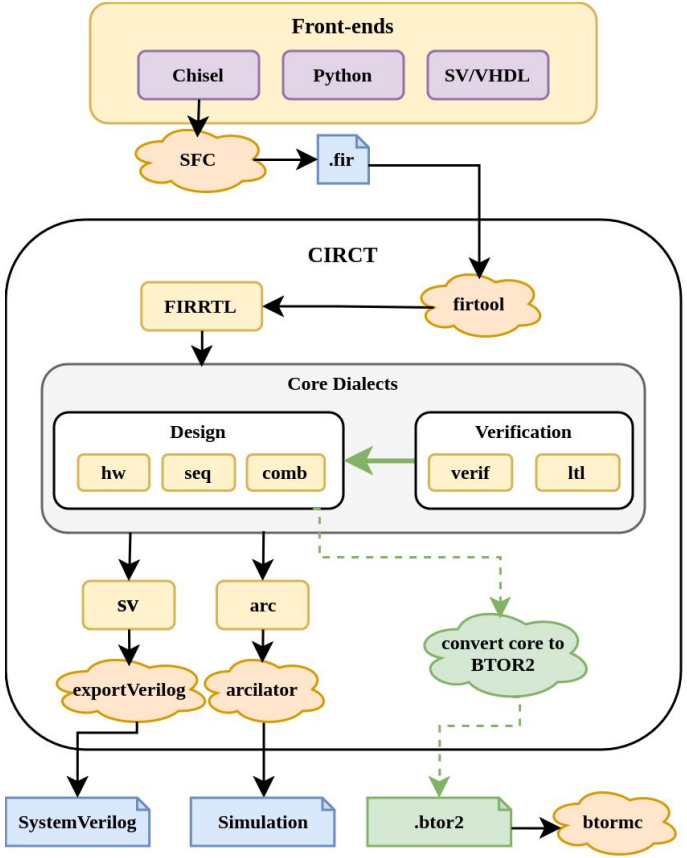
- Verif dialect supports assertions and assumptions.
  - `verif.has_been_reset` checks if the circuit has been reset yet or not. Useful for disabling properties before circuit is initialized.



How can CIRCT support this?

- Problem: LTL and Verif can only be lowered to SVA, not supported anywhere else.
- Solution: Lower LTL and Verif constructs to the core design dialects, so that they can be used in all targets (including btor2).

  → Can then be used with our formal back-end.



How do we lower SVA properties?

- **2 methods:**
  1) Build an **automaton** that monitors the property, make it deterministic and implement it as a FSM.
  2) Design custom direct lowerings for a select set of properties.
- **Problems:**
  - Method 1) requires an expensive automaton conversion into an internal representation
  - No good general method exists to encode properties as automaton in a modular way.
- → I opt for method 2)

How to Lower LTL and Verif to Core dialects?

1) Add support for property assertions.
2) Identify most common SVA properties and sequences.
3) Create a direct lowering for the identified properties.

First Step: Property Assertions

- **Properties assertions**: encode concurrently checked assertions.
  - Are always **clocked.**
  - Can be **disabled.**
    - By default are disabled as long as the circuit has not been reset.
  - Properties can encode temporal relations between signals in the design.

First Step: Support Property Assertions

```
AssertProperty(property, clock, disable=default)
```

```
%0 = seq.from_clock %clock

%true = hw.constant true
%9 = verif.has_been_reset %0, sync %reset
%10 = comb.xor bin %9, %true : i1
%13 = ltl.disable %property if %10 : i1

%14 = ltl.clock %13, posedge %0 : !ltl.property
verif.assert %14 : !ltl.property
```

First Step: Support Property Assertions

- **`verif.has_been_reset`:** Create a register that is set to its own value OR reset
  - `reg hbr := hbr || reset`
- **`ltl.disable %property if %disable`:**
  - `property := disable || reset || property`
  - Ignores the property if the circuit is in a reset cycle or the property is disabled
- **`ltl.clock + verif.assert`:** Directly map to supported sv dialect constructs

Second Step: Find which SVA properties to encode

- Previous work has analyzed which properties were most common in open-source designs:
  - **Result:**
    - Non-Overlapping Implication (NOI) (i.e. a implies b after n cycles)
    - Concatenation (a always holds n cycles after b)
- Engineers that rely on Chisel most commonly used properties:

Property	Description	Verilator Support
`a |-> b`	Simple implication	Yes
`a ##n b`	Constant delay concatenation	No
`a ##2 b ##[1:5] c`	Variable delay concatenation	No
`(disable iff (d)) ...`	Custom disabling of properties	No

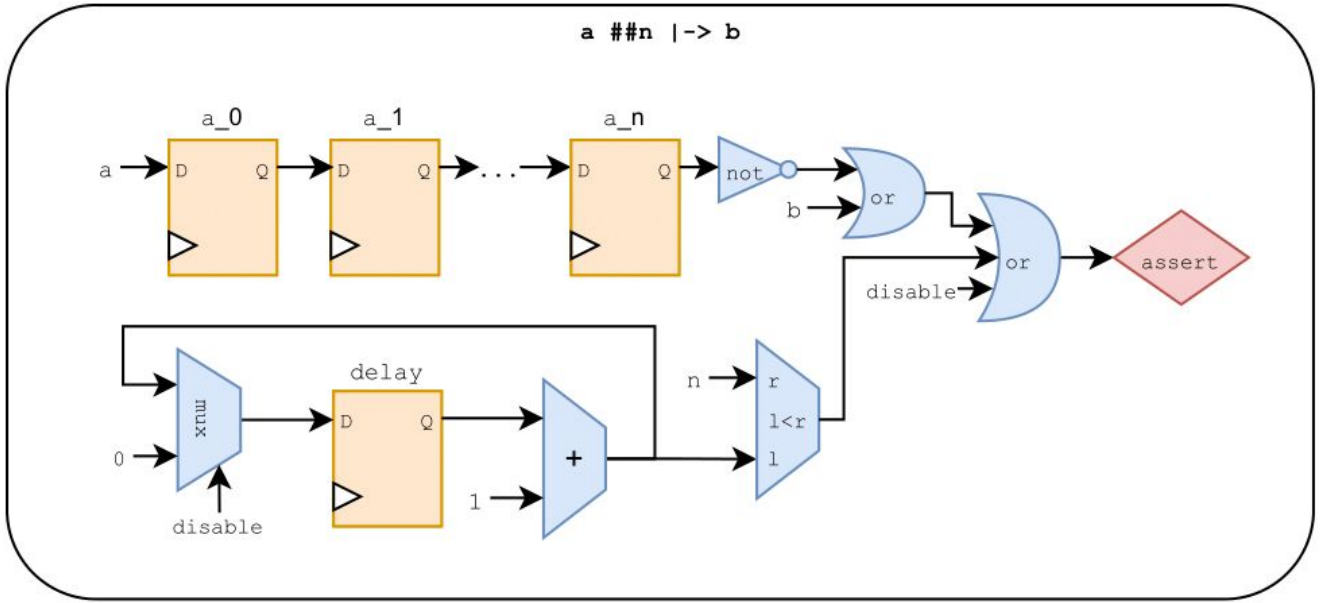
Third Step: Custom Lowerings

- **NOI** and **Concatenation** are chosen.
- NOI: *"a implies b after n cycles"*
  - Create a pipeline of registers to delay the antecedent by n cycles
  - Create a register to track the current cycle

```
a ##n true |-> b
```

```
reg delay, a_0, ..., a_n;
delay' = reset ? 0 : delay + 1
a_0' = a;
a_1' = a_0;
// ...
a_n' = a_(n-1)
assert (delay < n) || (a_n -> b) || reset
```

Lowering: a ##n true |-> b

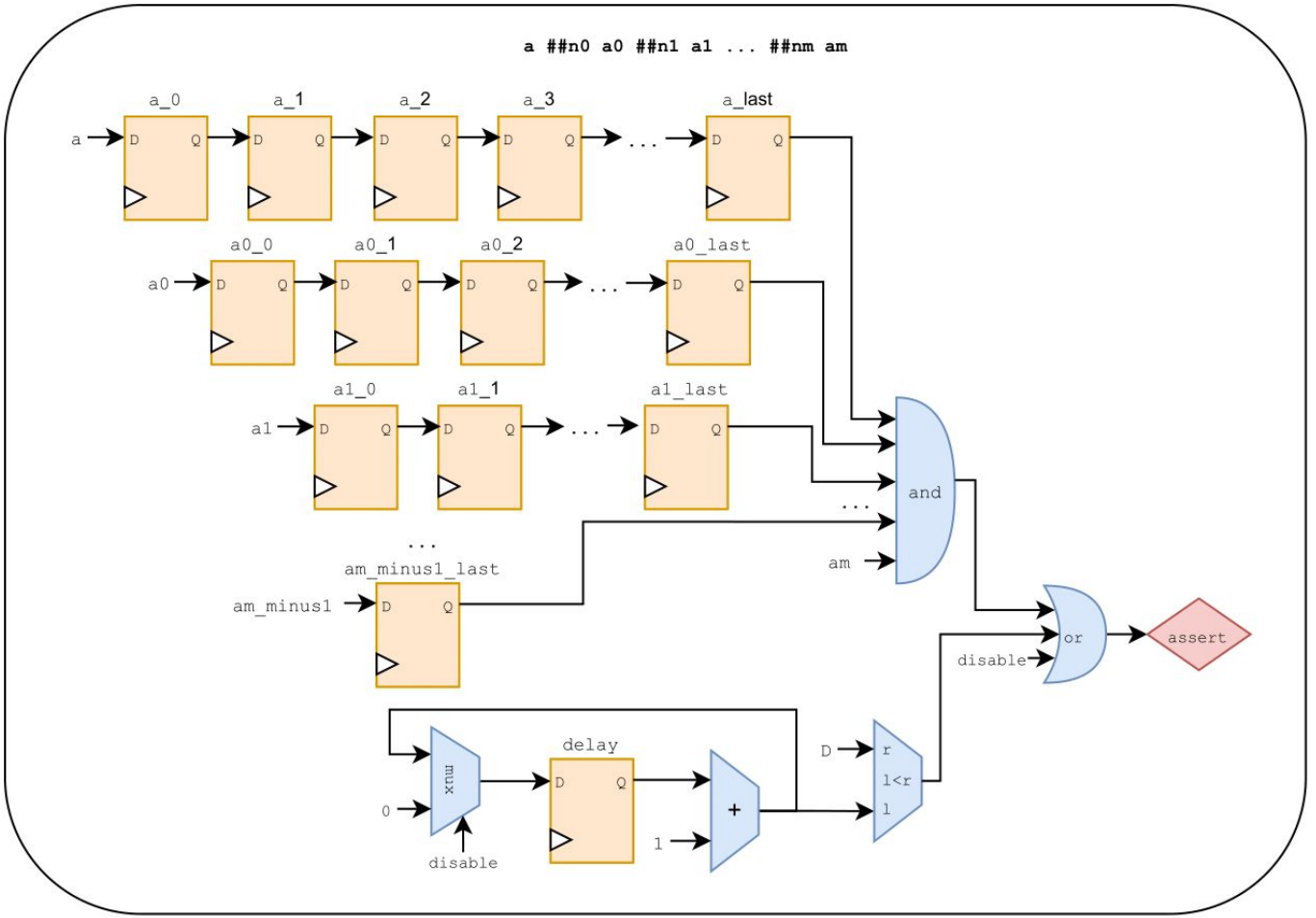


Third Step: Custom Lowerings

- Concatenation *"b holds n cycles after a"*
  - Create a pipeline of registers to delay the all registers in the sequence by a certain amount of cycles
  - Create a register to track the current cycle

```
a ##n0 a0 ##n1 a1 ... ##nm am
```

```
D = sum(i : 0..m)(ni) // total delay
reg delay;
reg a_0, ..., a_last; // D registers
reg a0_0, ..., a0_last; // (D - m) registers
reg a1_0, ..., a1_last; // (D - m - (m-1)) registers
//...
reg am-1_last;
delay' = reset ? 0 : delay + 1;
a_0' = a;
a_1' = a_0;
//...
a_last' = a_(D-1);
a0_0' = a_0;
a0_1' = a0_0;
//...
a0_last' = a0_(D - m);
//...
am-1_last' = am-1;
assert (delay < D) || a_last && and(i : 0..m)(ai_last) && am
       || reset
```


a ##n0 a0 ##n1 a1 ... ##nm am
a_0
a_1
a_2
a_3
a_last
a
D
Q
a0_0
a0_1
a0_2
a0_last
a0
a1_0
a1_1
a1_last
a1
am_minus1_last
am_minus1
and
am
or
disable
assert
delay
mux
0
disable
1
+
D
r
l<r
l

Result: SVA properties without tool support

- SVA properties and sequences can now be used regardless of the simulator or tool being used.
- LTL Lowering pass is integrated into firtool –btor2
- SVA properties and sequences can be used in the formal back-end.

Result: Specifying Sequential Circuits

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
}
```

How to write a specification for this simple design ?

Result: Specifying Sequential Circuits

```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
    AssertProperty(count < 42 |-> count > count.delay(1))
}
```

How to write a specification for this simple design ?

→ counter is monotonically increasing?

Result: Specifying Sequential Circuits

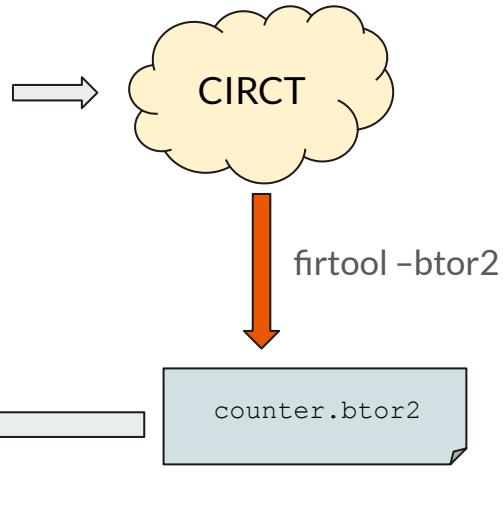
```
class Counter extends Module {
    val count = RegInit(0.U(32.W))
    when(count === 42.U) { count := 0.U }
    otherwise { count := count + 1.U }
    AssertProperty(count === 42.U |=> count === 0.U)
}
```

How to write a specification for this simple design ?

→ Is counter reset correctly?

Result: Integration into Formal flow

```
class Counter extends Module {
      val count = RegInit(0.U(32.W))
      when(count === 42.U) { count := 0.U }
      otherwise { count := count + 1.U }
      AssertProperty(count === 42.U |=> count === 0.U)
}
```



Part 3: Verifying the new passes

Verifying the new Passes

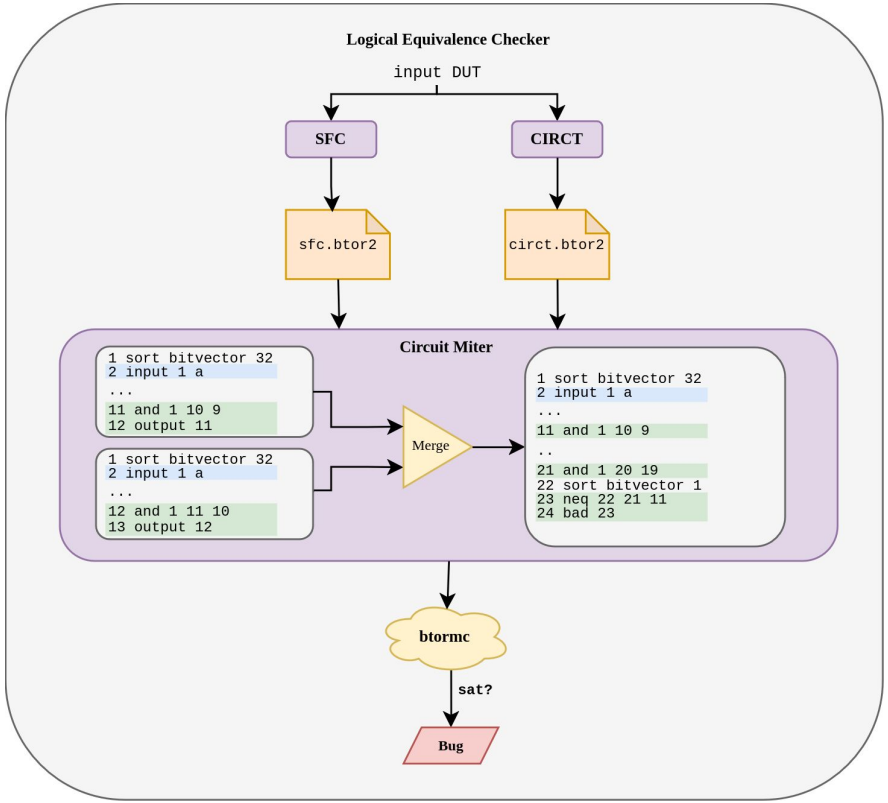
Goal: Design an **automated test suite** that verifies the correctness of the new passes.

1) BTOR2 emission: Compare with Scala FIRRTL Compiler using Equivalence Checking.
2) SVA Property Lowering: Exhaustively test all possible inputs within a given cycle-bound and compare SVA property to lowered version in a commercial simulator.

BTOR2 Equivalence Check

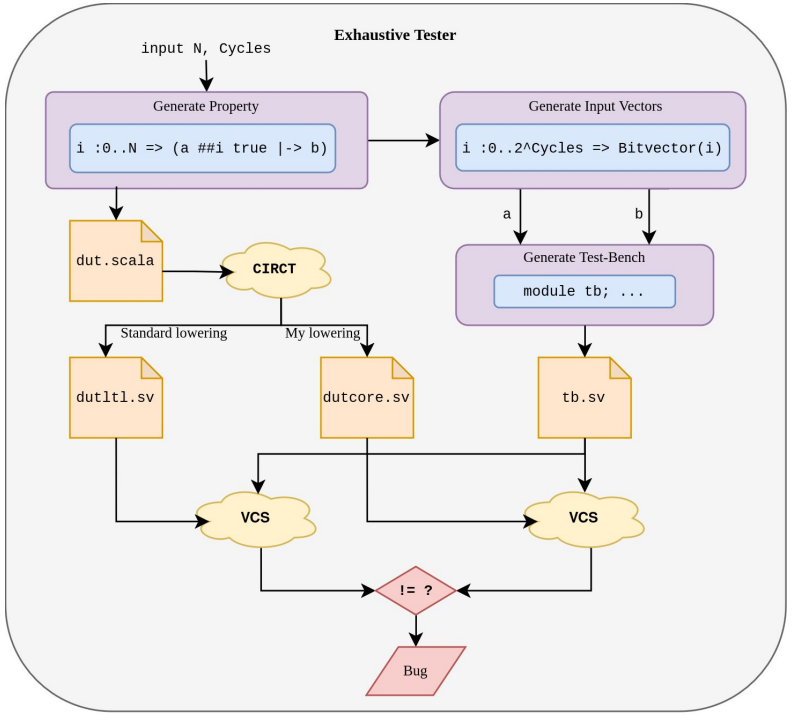
- Run same design through the Scala FIRRTL Compiler and CIRCT and create a miter circuit from the two outputs.
- Miter circuit: Merged circuit that checks the equivalence of the outputs of two designs.

- Result: Behavior differed due to **uninitialized registers** being handled differently.
- Tool is open-source: https://github.com/Dobios/btor2-opt



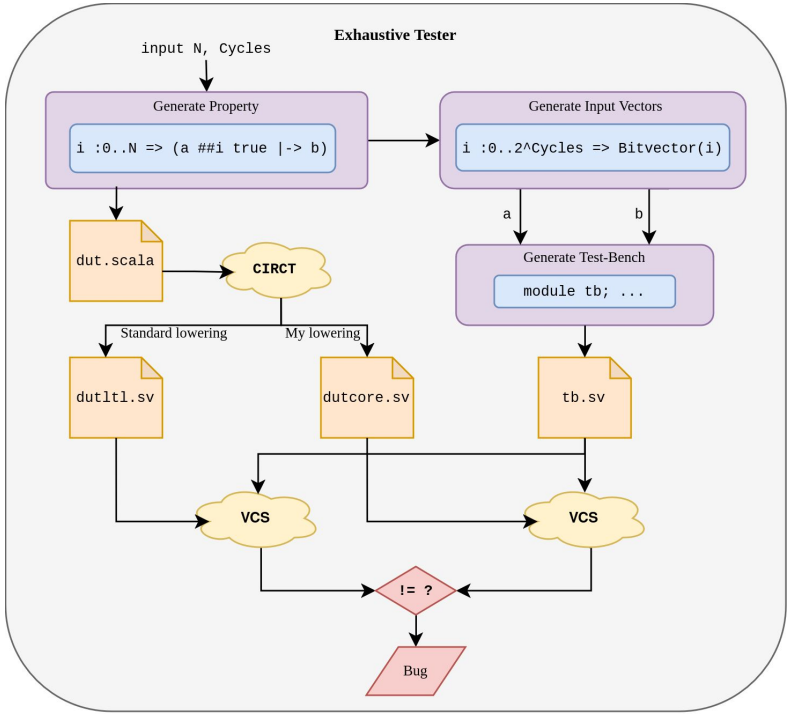
Cycle-Bound Exhaustive tester

- Generate all possible NOI properties within a given number of cycles.
    - Compile each property twice: through the **normal CIRCT pipeline** and through my **custom pipeline.**
- Generate all possible input vectors for a given number of cycles.
- Generate a testbench that stimulates both designs for each set of input vectors.
- Run both designs on the the testbench using Synopsys VCS and compare the outputs.



Cycle-Bound Exhaustive tester

- Result: Found a bug in how generated register were being reset in the lowered version using only 20 cycles of exhaustive testing.
- Tool is open-source: https://github.com/Dobios/SVExhaustiveTester

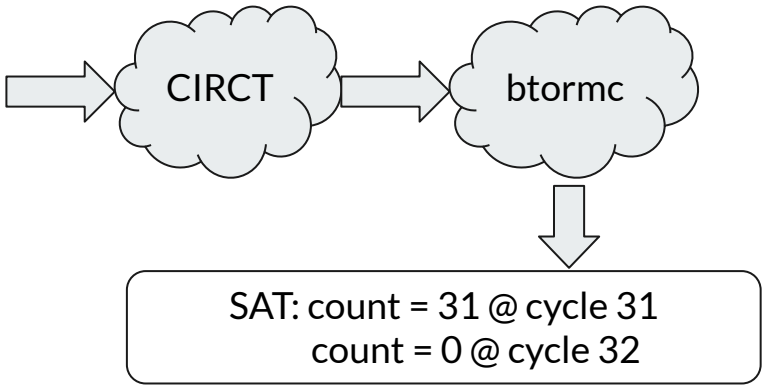


Conclusion

Conclusion: What is this for?

- Enables formal verification to be done in high level hardware languages like Chisel using SVA properties entirely in an open-source compiler.

```
class Counter extends Module {
      val count = RegInit(0.U(5.W))
      when(count === 32.U) { count := 0.U }
      otherwise { count := count + 1.U }
      AssertProperty(count < 32.U |=> count > 0.U)
}
```



Conclusion: Impact

- A lot of support from the CIRCT developer community
  - Triggered many other works (which I participate in) around ltl:
    - Automata dialect + FSM lowering with University of Cambridge
    - LTL dialect extensions
  - Many new works in the verifications space of CIRCT

→ The future looks bright for open-source verification support in high-level hardware languages!

Overview

1) Created a **unified formal backed** integrated into the CIRCT compiler
    a) Supports converting all of CIRCT's frontends into the **btor2** format

Overview

1) Created a **unified formal backed** integrated into the CIRCT compiler
   a) Supports converting all of CIRCT's frontends into the **btor2** format
2) Created **lowerings for property assertions** to a generally supported form
   a) Supports Non-overlapping Implication and Concatenation
   b) Is integrated into the formal backend

Overview

1) Created a **unified formal backed** integrated into the CIRCT compiler
   a) Supports converting all of CIRCT's frontends into the **btor2** format
2) Created **lowerings for property assertions** to a generally supported form
   a) Supports Non-overlapping Implication and Concatenation
   b) Is integrated into the formal backend
3) Designed an automated test-suited to verify to new additions to CIRCT.

Resources

- Written Thesis:
  - https://doi.org/10.3929/ethz-b-000668906
- BTOR2 Format:
  - https://link.springer.com/chapter/10.1007/978-3-319-96145-3_32
- CIRCT:
  - https://circt.llvm.org/docs/GettingStarted/
  - https://github.com/llvm/circt
- Verification works:
  - https://github.com/Dobios/btor2-opt
  - https://github.com/Dobios/SVExhaustiveTester

Any Questions ?