Novel Approaches for C vs. RTL Formal Verification of Vertex Attribute Address Generator Unit

Nianchen Wu, Christopher Starr, Xiushan Feng

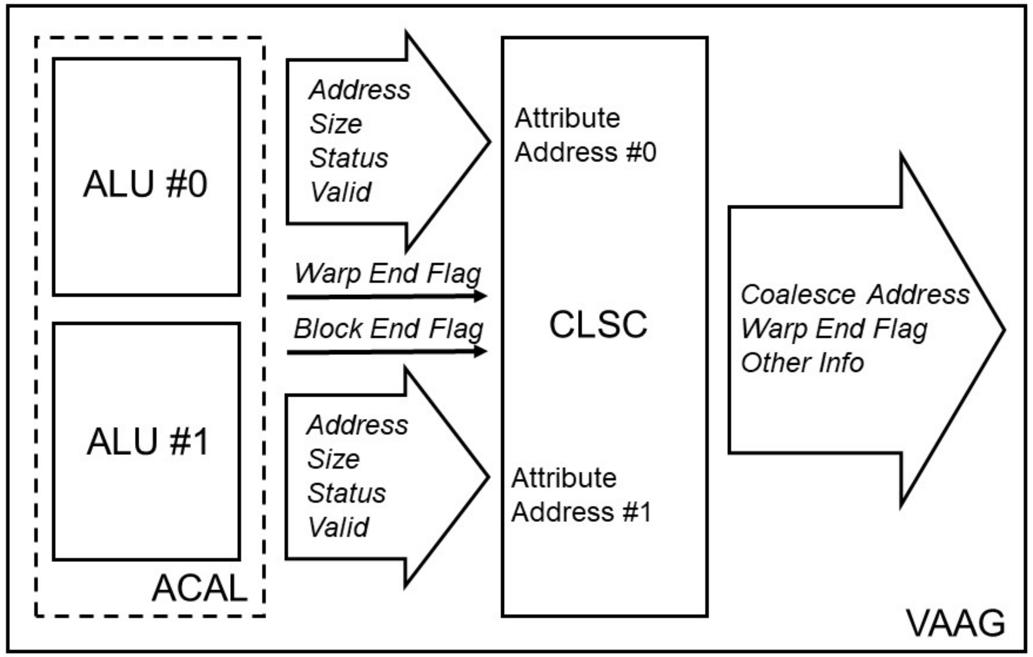
Samsung Austin R&D Center (SARC)

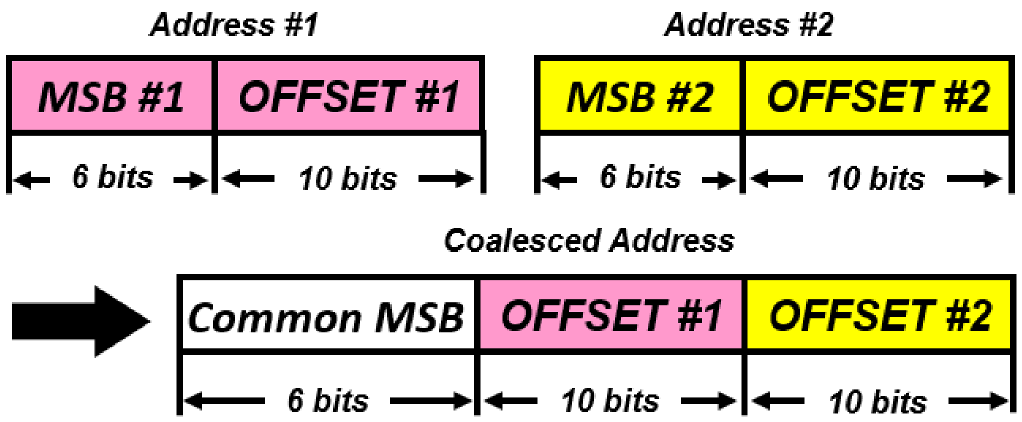
Outline

- **Background**
  - o Design introduction
  - o Selection of verification method and tool
  - o Verification challenges
- **Create a multi-cycle execution C model**
- **Verify the design through control flow graph (CFG)**
  - o Partition the state space into state transitions paths
  - o Verify the transition paths with symbolic trajectory evaluation (STE) method
- **Verification result, analysis and conclusion**

Introduction of VAAG

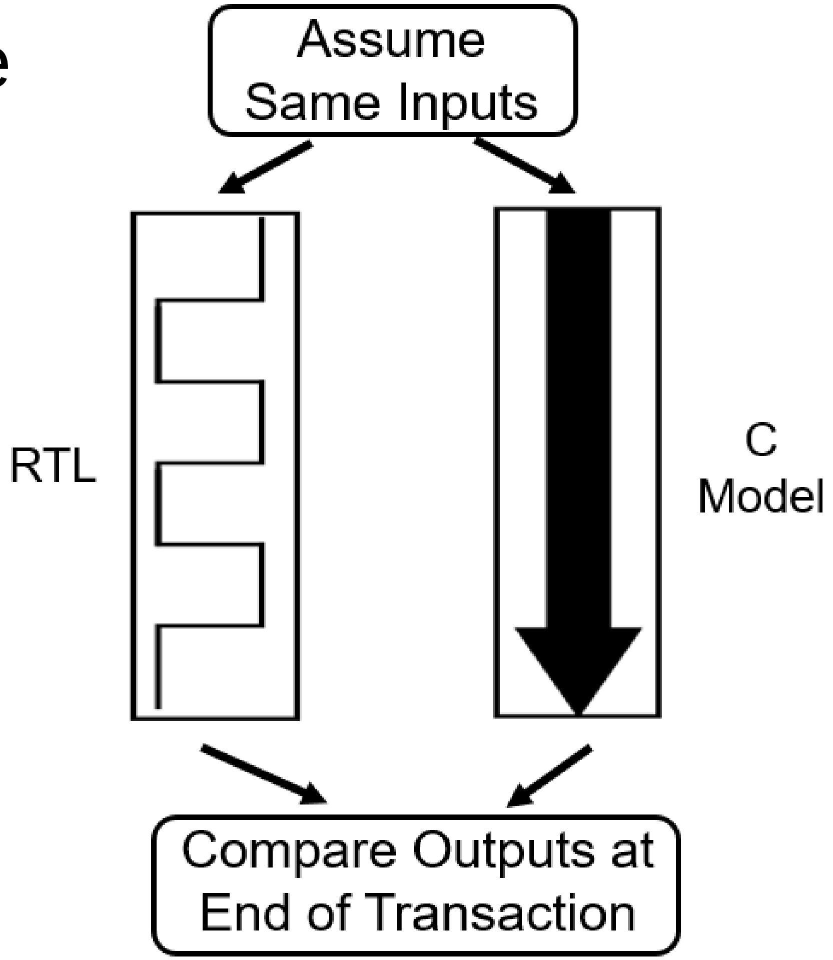
- Vertex attribute address generation
  - Generates vertex addresses for fetching vertex attributes
  - Two address calculation unit (ACAL)
  - One address coalescing unit (CLSC)
- Address coalescing
  - Merge multiple addresses that have the same most significant bits (MSB)





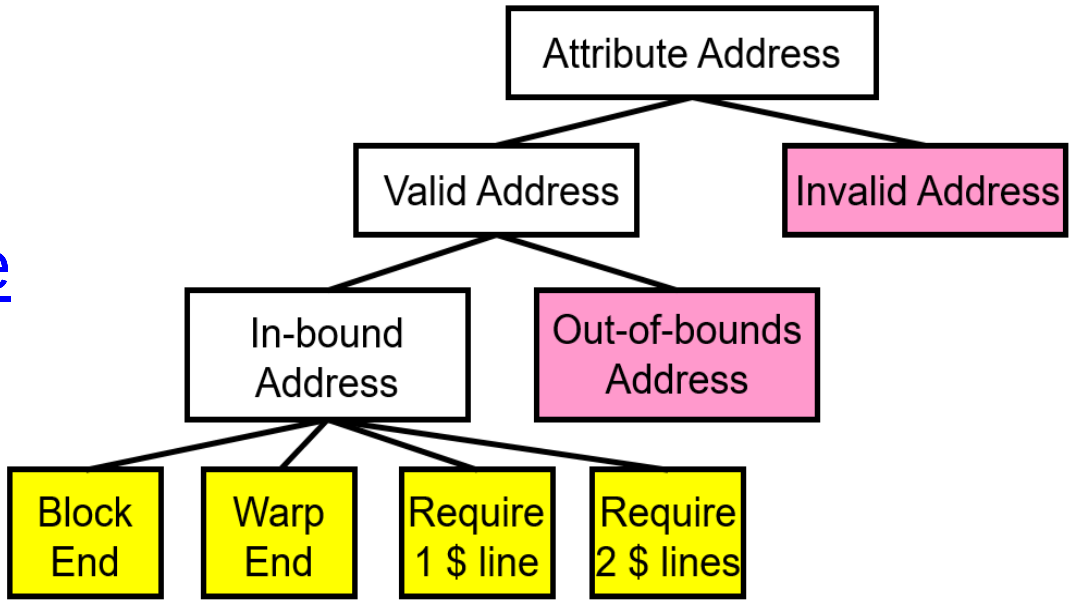
Verification Method and Tool Selection

- Complex arithmetic data paths cause huge state space inside VAAG
  - o Formal property verification (FPV): Hard to cover all state spaces, complicated constraints, hard to converge
  - o C vs. RTL formal verification (C2RTL): Exhaustively cover all possible cases, shorter test bench development time
- Tool: Synopsys Hector
  - o Verify RTL based on C model in ***cycle accuracy***



C2RTL Verification Challenges

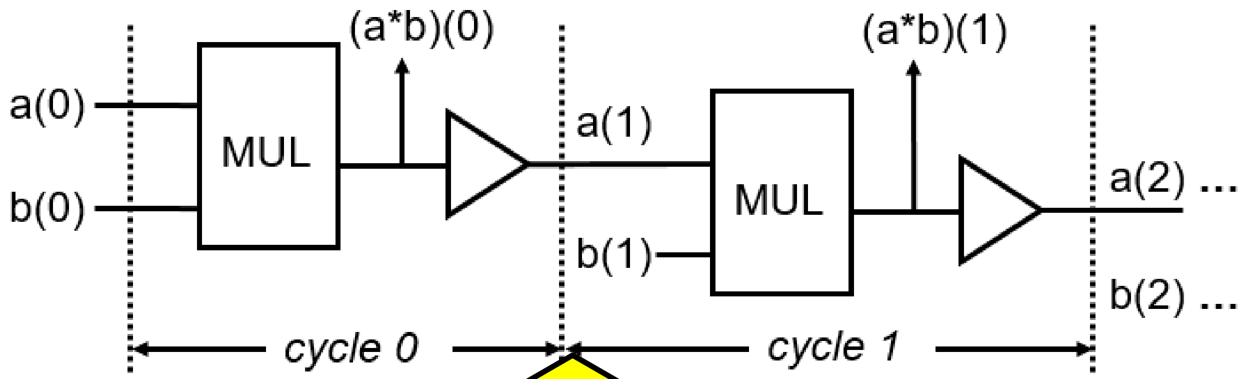
- We’ve fully verified ACAL, but faced big challenge on verifying CLSC
- Lots of features in attribute addresses
  - Coalescing latency varies with different features of the attribute address Example
- Huge mismatch between C model and RTL implementation
  - Delays, implementation algorithms...
  - Hard to prove output equivalence directly



Create a Multi-Cycle Execution C Model

- Split the original problem into pipeline stages
- “Unroll”: The modified C model ***only*** needs to generate result for the current cycle
- “Mapping”: The result (prime output) of cycle ***(n-1)*** will be mapped to the prime input of cycle ***n***

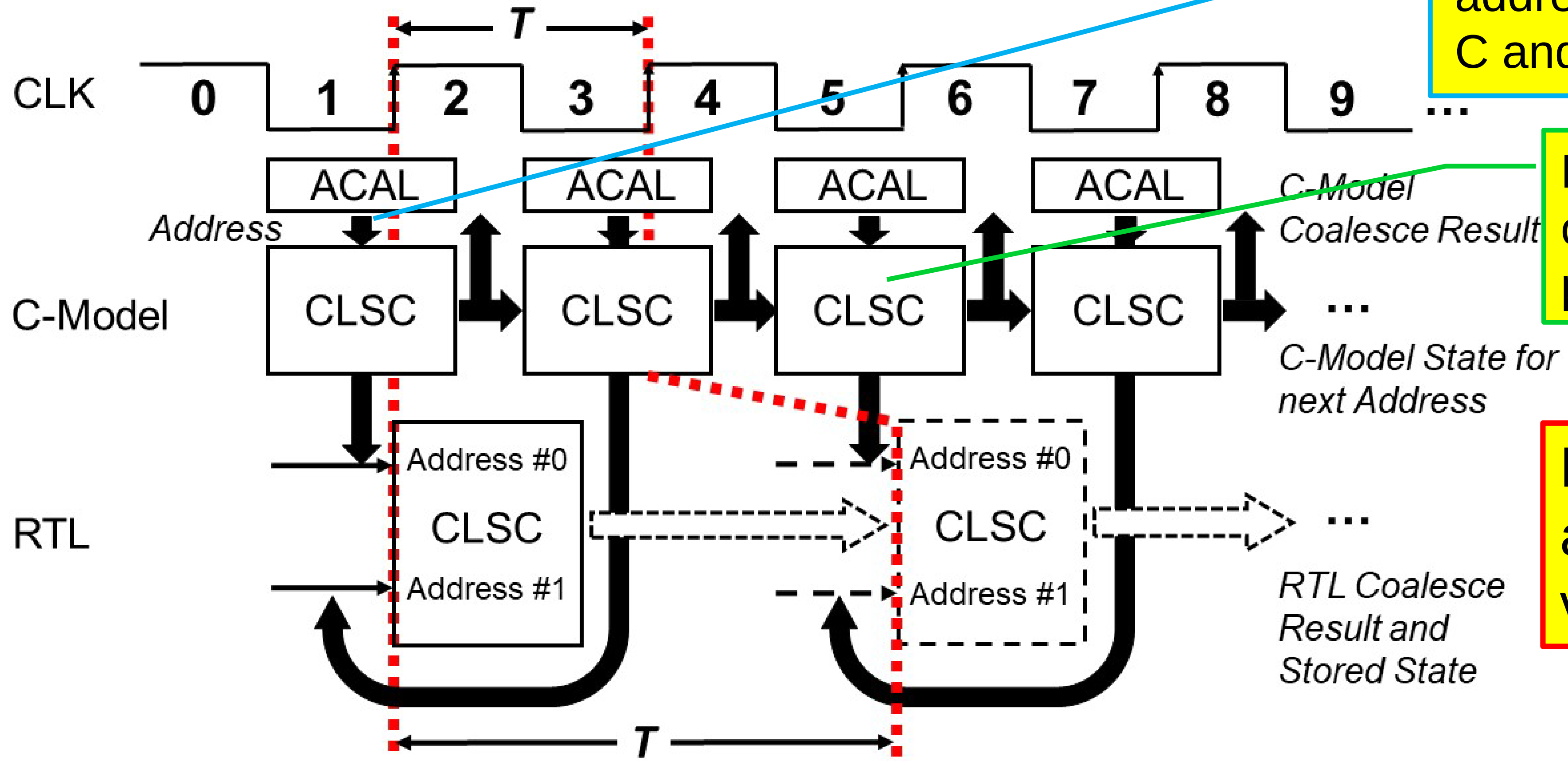
```
// Modified C Model
int a = 0, b = 3, c;
c = a * b;

// Time-Frame Expansion Mapping
assume a(cycle_0) = a_initial_value
assume a(cycle_1) = c(cycle_0)
assume a(cycle_2) = c(cycle_1)
assume a(cycle_3) = c(cycle_2)
```



This “unroll” and “mapping” process is like adding DFFs inside the C model

Application in Verifying CLSC

- Unroll the CLSC's C model and map to RTL



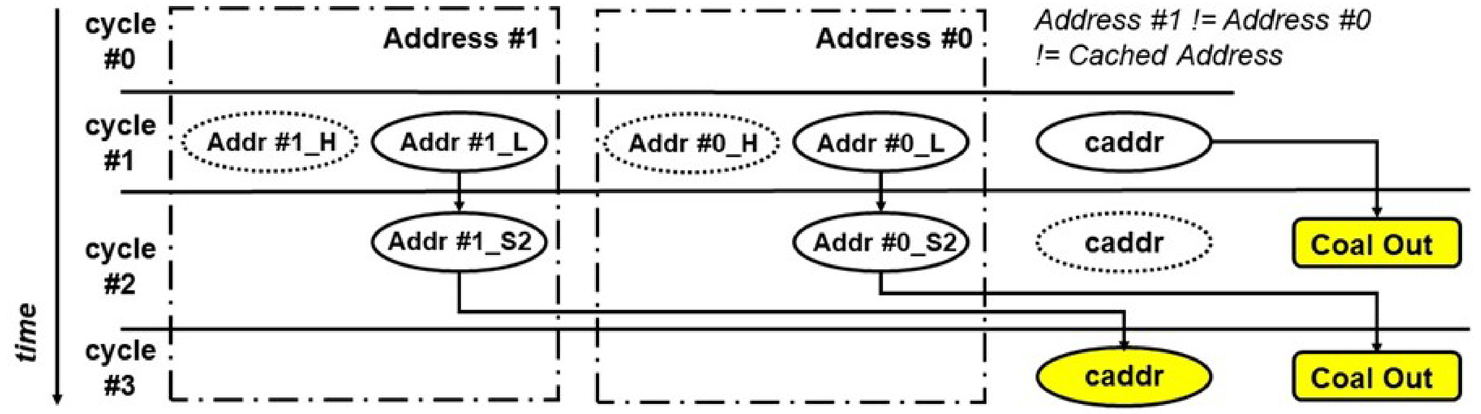
ACAL had been verified: Using ACAL's RTL to generate addresses and input to CLSC's C and RTL model

Modified CLSC's C model only processes **1** address per time

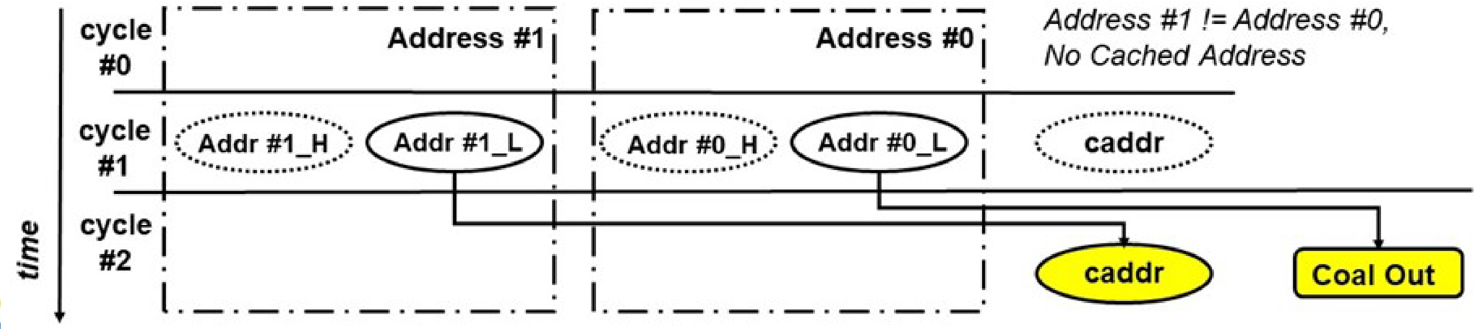
Mapping is achieved in formal verification tool

Verification through RTL Usage is Hard

- Complex RTL usage scenarios: Still might be incomplete!
- Verification requirement is different based on different RTL usage



(a)



(b)

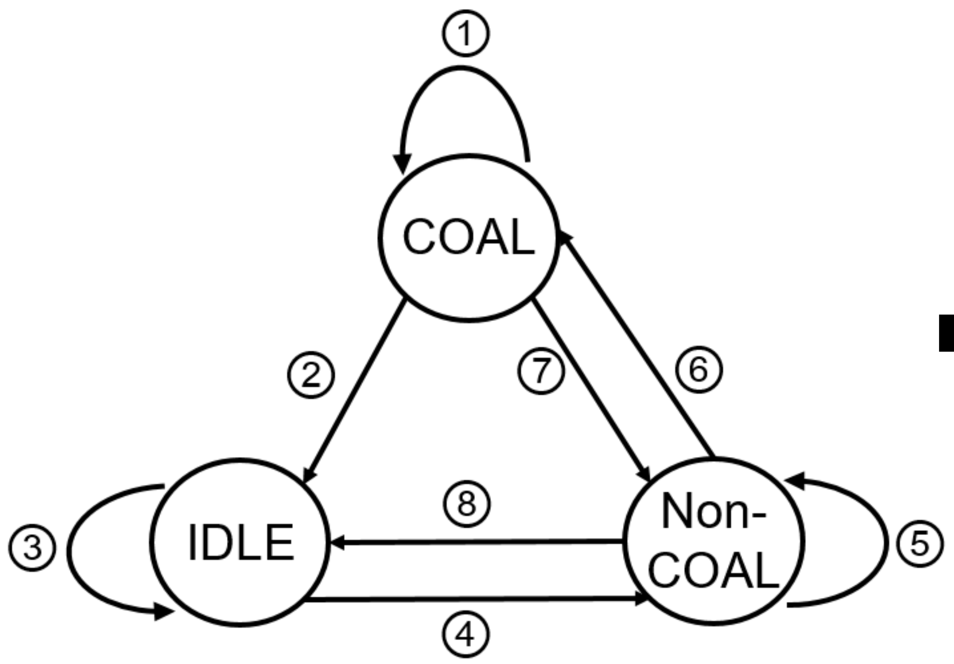
Legend:

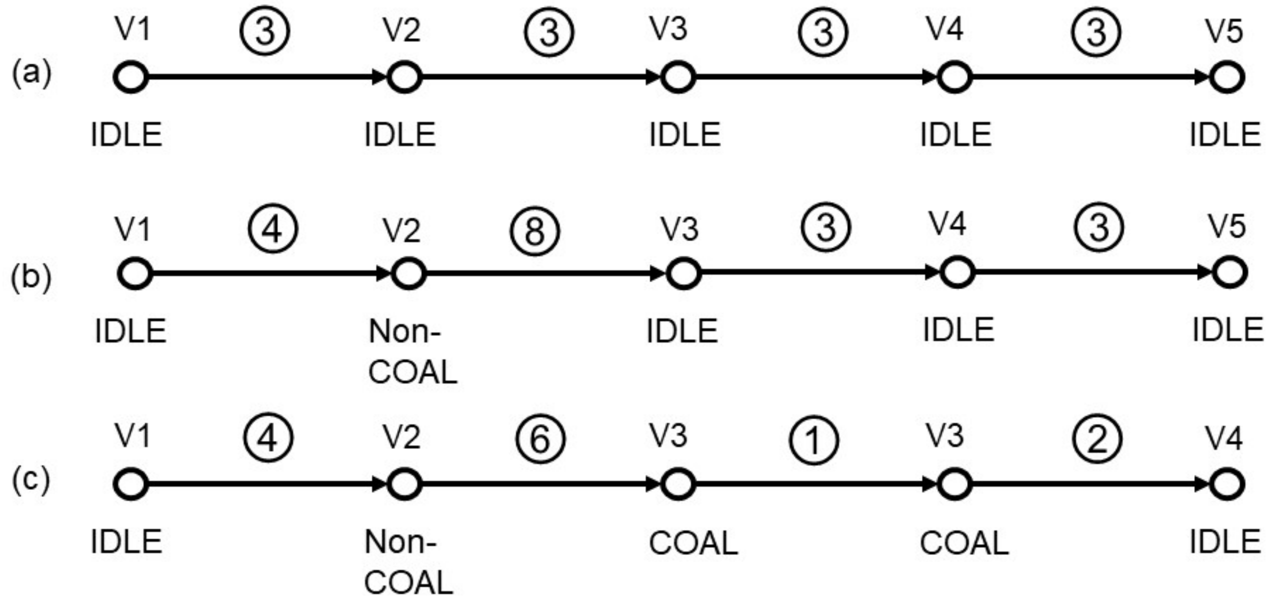
◌	: Invalid Address
○	: Valid Address
🟡 🟨	: Address must be Verified
caddr	: Cached Address
Coal Out	: Coalesce Result

BACK

Verification through Control Flow Graph

- The behavior of CLSC’s C model is relatively simple
- The number of valid transition paths is limited
- A transition path can be verified through the STE method



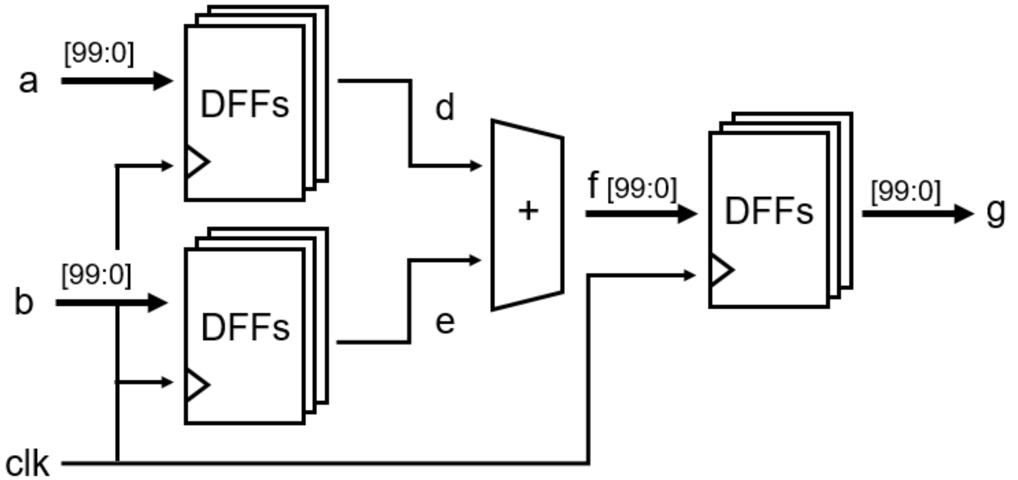


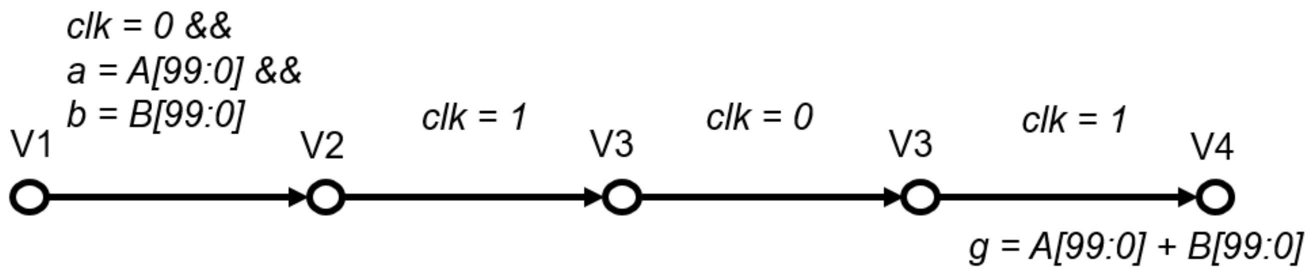
Show Transition Details

Show All Possible Paths

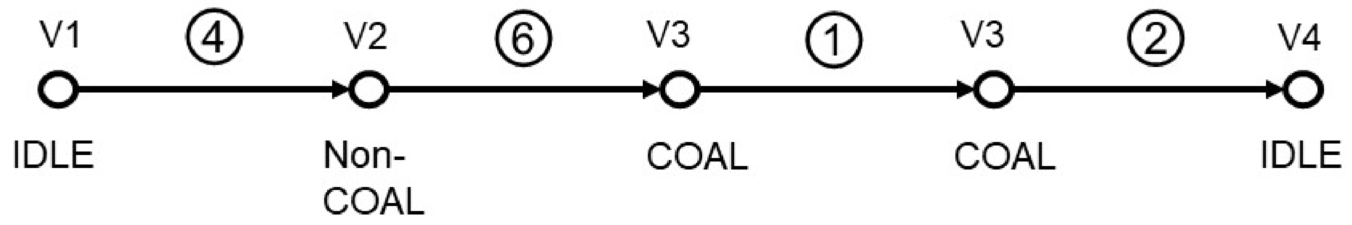
Introduction of STE Method

- Symbolic Trajectory Evaluation
  - A model checking technology that uses symbolic simulation
  - Example: The following 2-stage adder could be described as the following *STE assertion* and the *linear directive graph*:
    (clk == 0 && (a == A) && (b == B)) |-> ##2 (g == A + B)





Apply STE to Transition Paths Verification



- Check state transition: (S = State; T = Transition condition)

```
lemma v1_v2 = (S == IDLE) && T == C_4) |-> ##1 (S == Non-COAL)
lemma v2_v3 = (S == None-COAL) && T == C_6) |-> ##1 (S == COAL)
lemma v3_v3 = (S == COAL) && T == C_1) |-> ##1 (S == COAL)
lemma v3_v4 = (S == COAL) && T == C_2) |-> ##1 (S == IDLE)
```

- Check the coalescing result in this transition path: (p = Phase)

```
lemma result_p4 = (CLSC_Address_p3 == CLSC_Cached_Address_p4)
lemma result_p6 = (CLSC_Address_p3 == CLSC_Cached_Address_p6)
```

Verification Result: A RTL Bug Example

- Mismatch between the result generated by C model and RTL
  - o The most significant M bits are different in address #1 and address #0
  - o Address #0 could be coalesced with the address waiting in the cache
  - o FSM transitioned from state COAL to Non-COAL
- Root cause:
  - o Some internal registers were NOT reset properly (X-prop issue)
  - o A corner case difficult to be found by other verification methodologies

```
// Buggy Code
…
else if (taddr2clr_s3 | taddr2clr_s4) taddr2 <= ‘0;
…
```

→

```
// Correct Code
…
else taddr2 <= ‘0;
…
```

Verification Result: Performance Analysis

- Running time comparison for different verification strategy

	Proof Time for a Single Path / RTL Usage Scenario	Total Number of Paths / RTL Usage Scenarios	Total Proof Time Estimated
Verify CLSC through RTL Usage Scenario	45 second - 1.5 minutes	96	72 minutes - 150 minutes
Verify CLSC through Control Flow Graph	1.5 minutes - 5 minutes	12	20 minutes - 60 minutes

- Analysis of verifying CLSC by control flow graph

  (-) Verifying a single state transition path in CLSC usually needs more time

  (+) The number of paths is much less than the RTL usage scenarios

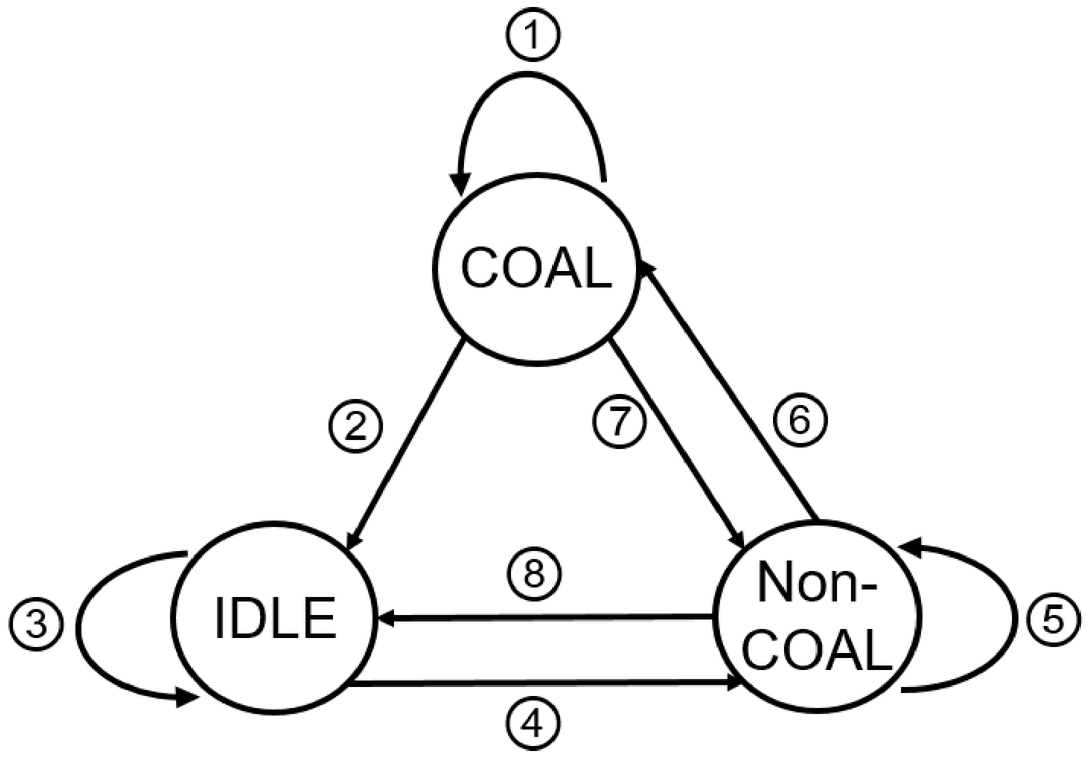
  (+) Could miss bug if the provided RTL scenarios is incomplete, but C model is always golden!

Conclusion

- A novel approach to solve complex sequential data path C2RTL verification problem
- Create a multi-cycle execution C model to simplify the original problem
- Split the state space and verify a design through control flow graph could be a reliable and effective verification strategy
- The verification method in this presentation could be applied for other sequential logic that has complex usage scenario but simple C model

Q & A

Appendix A: CLSC Finite State Machine

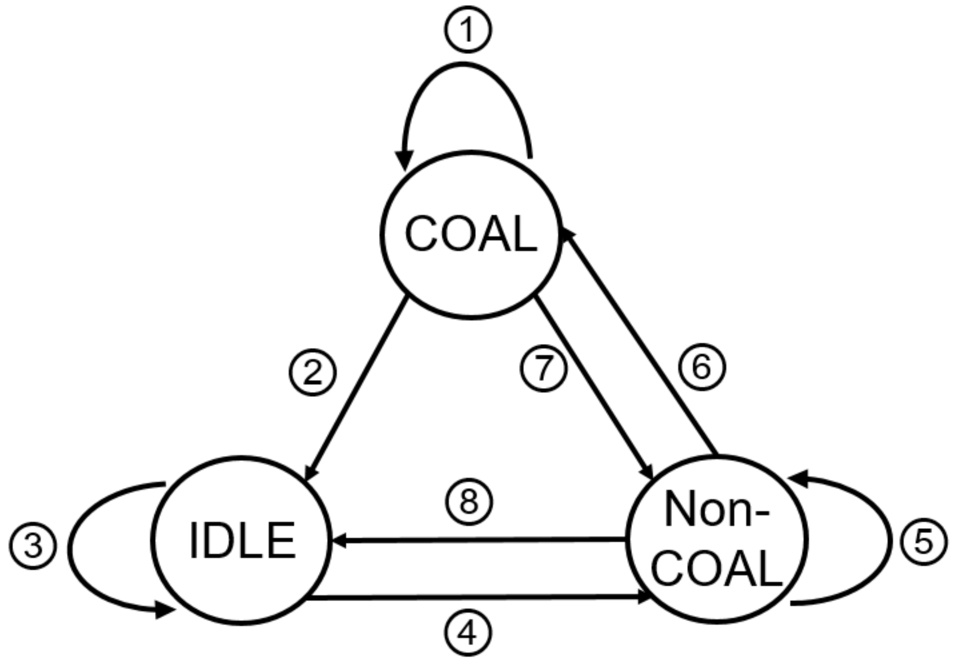


Back to Page 9

1. COAL => COAL:
   Address MSB == Cached Address MSB && Below the max coalesce number.
2. COAL => IDLE:
   1. Address MSB == Cached testbench_flatAddress MSB && Reach the max coalesce number.
      (Output 1 address)
   2. Address is out of boundary.
      (Output 2 addresses)
3. IDLE => IDLE:
   Address is out of boundary.
   (Output 1 address)
4. IDLE => Non-COAL
   Any valid, in bound address.
5. Non-COAL => Non-COAL:
   1. Address requires 2 cache line.
      (Output 1 address)
   2. Address MSB != Cached Address MSB. (Output 1 address)
6. Non-COAL => COAL:
   Address MSB == Cached Address MSB.
7. COAL => Non-COAL:
   1. Address requires 2 cache lines.
      (Output 1 address)
   2. Max coalesce number is 0.
   3. Address MSB != Cached Address MSB
8. Non-COAL => IDLE:
   1. Address is out of boundary.
      (Output 2 addresses)
   2. Address is Warp/Block end.
      (Output 1 address)

Appendix B: All Possible State Transition Path for CLSC FSM

1. IDLE → IDLE → IDLE → IDLE → IDLE
2. IDLE → NCOL → IDLE → IDLE → IDLE
3. IDLE → IDLE → NCOL → IDLE → IDLE
4. IDLE → IDLE → IDLE → NCOL → IDLE
5. IDLE → NCOL → NCOL → IDLE → IDLE
6. IDLE → NCOL → IDLE → NCOL → IDLE
7. IDLE → IDLE → NCOL → NCOL → IDLE
8. IDLE → NCOL → NCOL → NCOL → IDLE
9. IDLE → NCOL → COAL → IDLE → IDLE
10. IDLE → NCOL → NCOL → COAL → IDLE
11. IDLE → NCOL → COAL → NCOL → IDLE
12. IDLE → NCOL → COAL → COAL → IDLE



Back to Page 9