A DE Verification Framework for Asynchronous Circuit Verification

Cuong Chau

ckcuong@cs.utexas.edu

Department of Computer Science

The University of Texas at Austin

January 27, 2017

Outline

Outline

Introduction

Synchronous circuits (Clocked circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (Self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communication between components is performed via **local communication protocols**.

Introduction

Synchronous circuits (Clocked circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (Self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communication between components is performed via **local communication protocols**.

Why asynchronous?

Introduction

Synchronous circuits (Clocked circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (Self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communication between components is performed via **local communication protocols**.

Why asynchronous?

- Low power consumption,
- High operating speed,
- Better composability and modularity,
- ...

Introduction

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.

- Developing new approaches for modeling and verifying asynchronous systems in ACL2. Dealing with:

Introduction

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
  - The DE verification system [W. Hunt, 2000].
- Developing new approaches for modeling and verifying asynchronous systems in ACL2. Dealing with:

Introduction

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
  - The DE verification system [W. Hunt, 2000].
- Developing new approaches for modeling and verifying asynchronous systems in ACL2. Dealing with:
  - no global clock signal,

Introduction

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
  - The DE verification system [W. Hunt, 2000].
- Developing new approaches for modeling and verifying asynchronous systems in ACL2. Dealing with:
  - no global clock signal,
  - local communication protocols,

Introduction

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
  - The DE verification system [W. Hunt, 2000].
- Developing new approaches for modeling and verifying asynchronous systems in ACL2. Dealing with:
  - no global clock signal,
  - local communication protocols,
  - non-deterministic behavior due to *variable delays* in wires and gates.

Outline

The DE Language

DE is a formal occurrence-oriented hardware description language for describing Mealy machines. It allows **hierarchical module definition**, and multiple copies of a module are identified by reference (their appearance in an occurrence).

The DE Language

DE is a formal occurrence-oriented hardware description language for describing Mealy machines. It allows **hierarchical module definition**, and multiple copies of a module are identified by reference (their appearance in an occurrence).

A DE description is an ACL2 constant containing an **ordered list of modules**, which we call a **netlist**.

The DE Language

DE is a formal occurrence-oriented hardware description language for describing Mealy machines. It allows **hierarchical module definition**, and multiple copies of a module are identified by reference (their appearance in an occurrence).

A DE description is an ACL2 constant containing an **ordered list of modules**, which we call a **netlist**.

Each module consists of five elements: a netlist-unique module name, inputs, outputs, internal states, and occurrences.

The DE Language

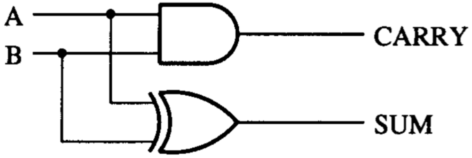
DE is a formal occurrence-oriented hardware description language for describing Mealy machines. It allows **hierarchical module definition**, and multiple copies of a module are identified by reference (their appearance in an occurrence).

A DE description is an ACL2 constant containing an **ordered list of modules**, which we call a **netlist**.

Each module consists of five elements: a netlist-unique module name, inputs, outputs, internal states, and occurrences.

Each occurrence consists of four elements: a module-unique occurrence name, outputs, **a reference to a primitive or defined module**, and inputs.

Half-Adder



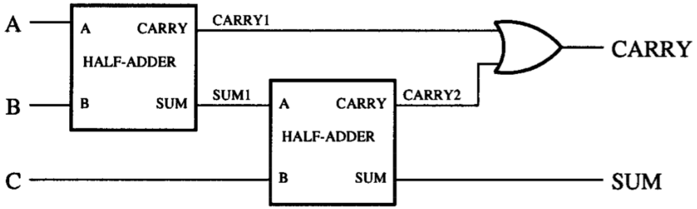
```
(defconst *half-adder*
  '((half-adder    ;; module name
     (a b)         ;; module inputs
     (sum carry)   ;; module outputs
     ()            ;; internal states
      ;; occurrences
     ((g0       ;; occurrence name
       (sum)    ;; occurrence outputs
       b-xor    ;; a primitive reference
       (a b))   ;; occurrence inputs
      (g1 (carry) b-and (a b))))))
```

The DE Primitive Database

The evaluation of a DE netlist eventually results in the interpretation of **primitives**, which are specified in the DE primitive database.

- Logic gates: AND, OR, NOT, XOR,...
- State-holding primitives: latches, flip-flops,...

Full-Adder

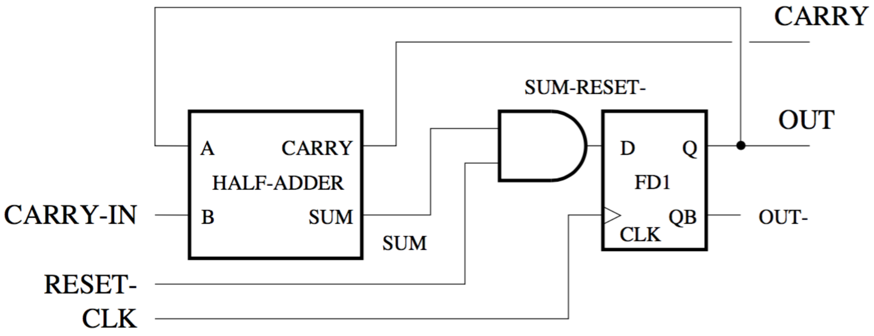

A
B
C
A
CARRY
HALF-ADDER
B
SUM
CARRY1
SUM1
A
CARRY
HALF-ADDER
B
SUM
CARRY2
CARRY
SUM

Full-Adder

```
(defconst *full-adder*
  (cons
   '(full-adder
     (a b c)
     (sum carry)
     ()
     ((t0 (sum1 carry1) half-adder           (a b))
      (t1 (sum  carry2) half-adder        (sum1 c))
      (t2 (carry)       b-or       (carry1 carry2))))

   *half-adder*))
```

One-Bit Counter



One-Bit Counter

```
(defconst *one-bit-counter*
  (cons
   '(one-bit-counter
     (clk carry-in reset-)
     (out carry)
     (g0)
     ((g0 (out out~)   fd1         (clk sum-reset-))
      (g1 (sum carry)  half-adder   (carry-in out))
      (g2 (sum-reset-) b-and          (sum reset-))))

   *half-adder*))
```

The DE Simulator

Semantics of the DE language: a simulator computing module's outputs and next state, given current inputs and current state.

The DE Simulator

Semantics of the DE language: a simulator computing module's outputs and next state, given current inputs and current state.

The DE simulator is composed of two sets of mutually recursive functions.

- The `se` function computes the **outputs** of a module being evaluated given its inputs and its current state.

The DE Simulator

Semantics of the DE language: a simulator computing module's outputs and next state, given current inputs and current state.

The DE simulator is composed of two sets of mutually recursive functions.

- The `se` function computes the **outputs** of a module being evaluated given its inputs and its current state. The `se-occ` function, which is mutually recursive with `se`, iteratively computes the **outputs** of each occurrence declared in the module.

The DE Simulator

Semantics of the DE language: a simulator computing module's outputs and next state, given current inputs and current state.

The DE simulator is composed of two sets of mutually recursive functions.

- The `se` function computes the **outputs** of a module being evaluated given its inputs and its current state. The `se-occ` function, which is mutually recursive with `se`, iteratively computes the **outputs** of each occurrence declared in the module.
- The `de` function computes the **next state** of a module being evaluated given its inputs and its current state.

The DE Simulator

Semantics of the DE language: a simulator computing module's outputs and next state, given current inputs and current state.

The DE simulator is composed of two sets of mutually recursive functions.

- The `se` function computes the **outputs** of a module being evaluated given its inputs and its current state. The `se-occ` function, which is mutually recursive with `se`, iteratively computes the **outputs** of each occurrence declared in the module.
- The `de` function computes the **next state** of a module being evaluated given its inputs and its current state. The `de-occ` function, which is mutually recursive with `de`, iteratively computes the (possibly empty) **next state** of each occurrence declared in the module.

The DE Verification System

The DE verification system supports **hierarchical verification**:

The DE Verification System

The DE verification system supports **hierarchical verification**:
Prove two lemmas for each module: a **value lemma** specifying the module's outputs and a **state lemma** specifying the module's next state.

The DE Verification System

The DE verification system supports **hierarchical verification**:
Prove two lemmas for each module: a **value lemma** specifying the module's outputs and a **state lemma** specifying the module's next state.

- If a module doesn't have an internal state (purely combinational), only the value lemma needs to be proven.

The DE Verification System

The DE verification system supports **hierarchical verification**:
Prove two lemmas for each module: a **value lemma** specifying the module's outputs and a **state lemma** specifying the module's next state.

- If a module doesn't have an internal state (purely combinational), only the value lemma needs to be proven.
- These lemmas are used to prove the correctness of yet larger modules containing these submodules, without the need to dig into any details about the submodules.

The DE Verification System

The DE verification system supports **hierarchical verification**:
Prove two lemmas for each module: a **value lemma** specifying the module's outputs and a **state lemma** specifying the module's next state.

- If a module doesn't have an internal state (purely combinational), only the value lemma needs to be proven.
- These lemmas are used to prove the correctness of yet larger modules containing these submodules, without the need to dig into any details about the submodules.

This approach has been demonstrated its **scalability** to large systems, as shown on contemporary x86 designs at Centaur Technology [A. Slobodova et al., 2011].

Outline

Modeling

- No global clock signal
- Local communication protocols
- Non-deterministic behavior

Modeling

- No global clock signal
  $\Rightarrow$ Every state-holding device is governed by its own clock signal.
- Local communication protocols

- Non-deterministic behavior

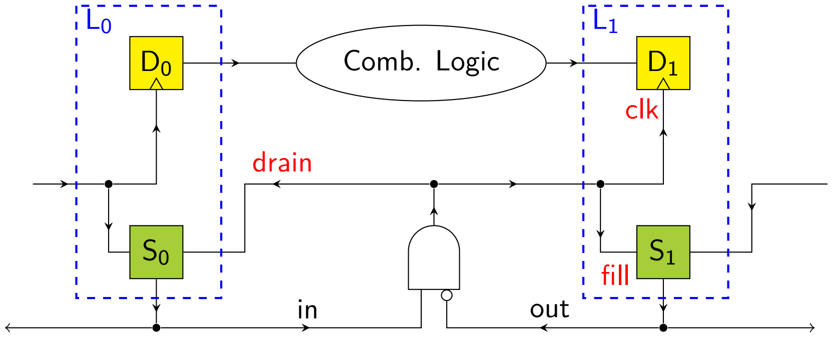
Modeling

- No global clock signal
  $\Rightarrow$ Every state-holding device is governed by its own clock signal.
- Local communication protocols
  $\Rightarrow$ Modeling the link-joint interface introduced by Roncken et al. [M. Roncken et al., 2015].
- Non-deterministic behavior

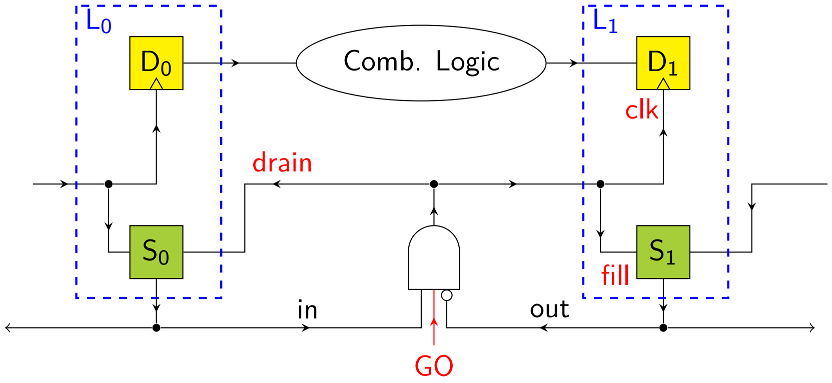
Modeling

- No global clock signal
  $\Rightarrow$ Every state-holding device is governed by its own clock signal.
- Local communication protocols
  $\Rightarrow$ Modeling the link-joint interface introduced by Roncken et al. [M. Roncken et al., 2015].
- Non-deterministic behavior
  $\Rightarrow$ Employing an oracle.

The Link-Joint Interface



The Link-Joint Interface



Verification

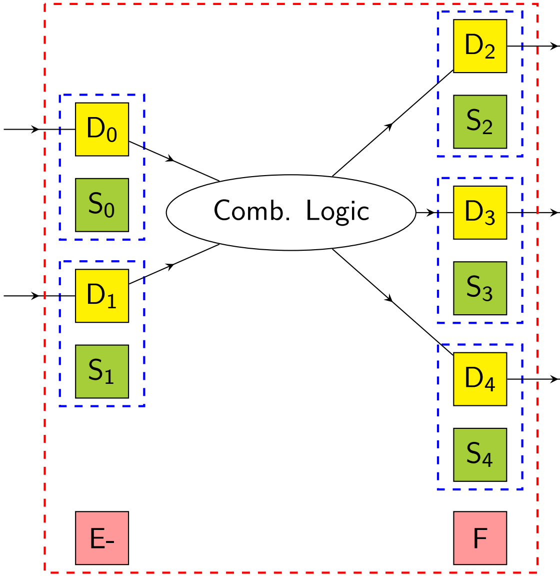
Hierarchical verification is still applicable and critical to asynchronous circuit verification.

Verification

Hierarchical verification is still applicable and critical to asynchronous circuit verification.

Asynchronous modules are treated as **communication links** that communicate with each other via local communication protocols.

Async Modules vs. Primitive Links

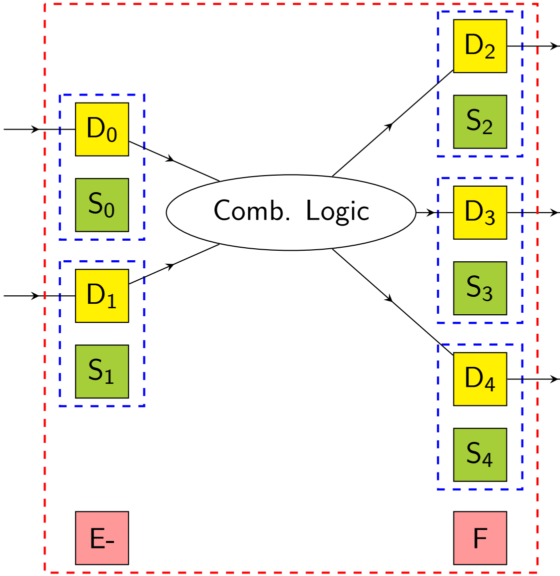


Async Modules vs. Primitive Links

Communication status:

Async modules: full, empty, both full and empty, not ready (neither full nor empty).

Primitive links: either full or empty.



Async Modules vs. Primitive Links

Communication status:

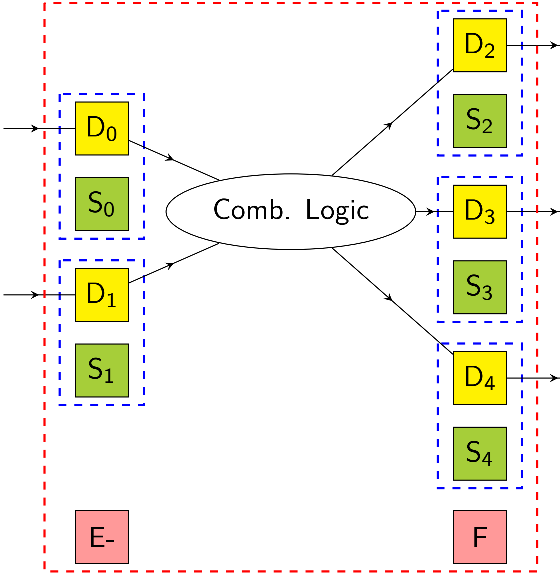
Async modules: full, empty, both full and empty, not ready (neither full nor empty).

Primitive links: either full or empty.

Communication signals:

Async modules use separate incoming and outgoing communication signals.

Primitive links only need one signal for both incoming and outgoing communications.



Dealing with Non-determinism

Non-deterministic behavior in asynchronous circuits makes the verification task much more challenging than in synchronous circuits.

Dealing with Non-determinism

Non-deterministic behavior in asynchronous circuits makes the verification task much more challenging than in synchronous circuits.

- Computing **invariance properties** in asynchronous systems becomes much more complicated and less systematic.

Dealing with Non-determinism

Non-deterministic behavior in asynchronous circuits makes the verification task much more challenging than in synchronous circuits.

- Computing **invariance properties** in asynchronous systems becomes much more complicated and less systematic.
- Non-determinism results in asynchronous circuits having a large state space.

Dealing with Non-determinism

Non-deterministic behavior in asynchronous circuits makes the verification task much more challenging than in synchronous circuits.

- Computing **invariance properties** in asynchronous systems becomes much more complicated and less systematic.
- Non-determinism results in asynchronous circuits having a large state space.

Simplifying the verification task by reducing non-determinism, and consequently reducing the state space.

Dealing with Non-determinism

Non-deterministic behavior in asynchronous circuits makes the verification task much more challenging than in synchronous circuits.

- Computing **invariance properties** in asynchronous systems becomes much more complicated and less systematic.
- Non-determinism results in asynchronous circuits having a large state space.

Simplifying the verification task by reducing non-determinism, and consequently reducing the state space.

- Imposing extra **sequential dependencies** among operations in asynchronous circuits.

Dealing with Non-determinism

Non-deterministic behavior in asynchronous circuits makes the verification task much more challenging than in synchronous circuits.

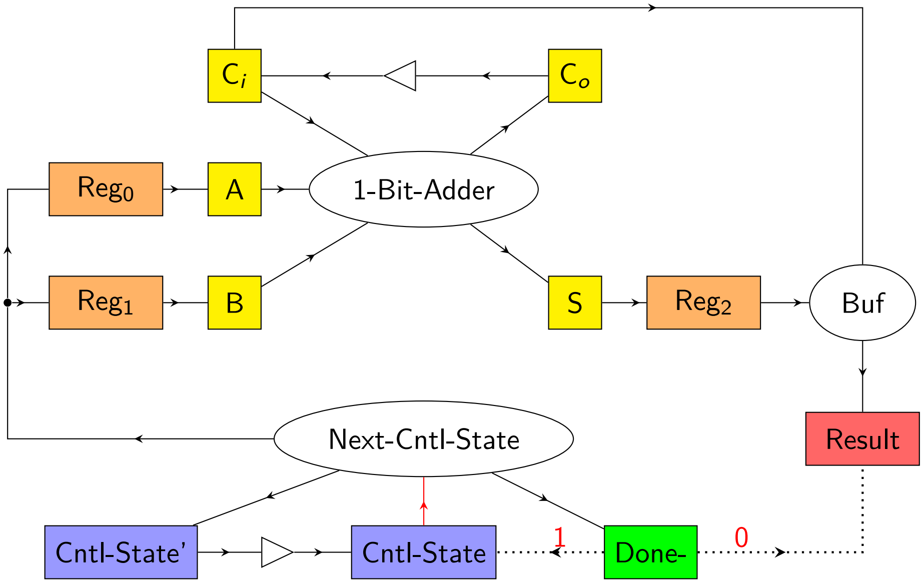
- Computing **invariance properties** in asynchronous systems becomes much more complicated and less systematic.
- Non-determinism results in asynchronous circuits having a large state space.

Simplifying the verification task by reducing non-determinism, and consequently reducing the state space.

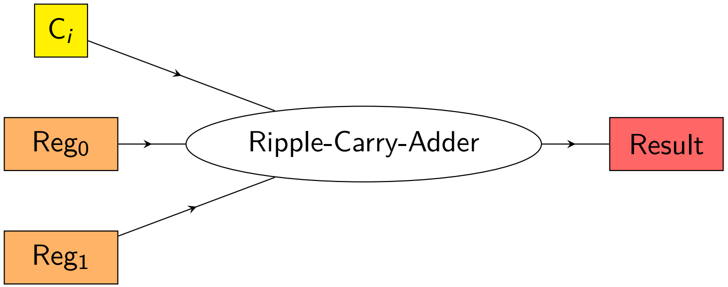
- Imposing extra **sequential dependencies** among operations in asynchronous circuits. In particular, a module is ready to communicate with other modules only if **it finishes all of its internal operations and becomes quiescent**.

Outline

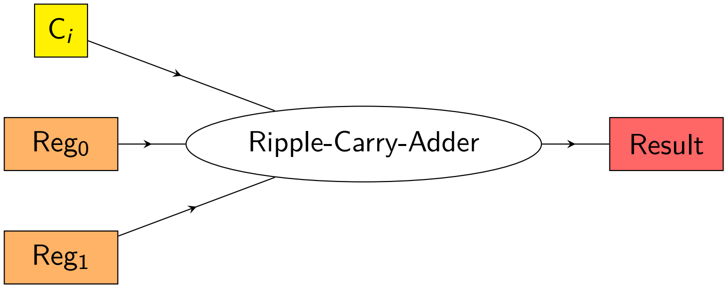
Serial Adder



Ripple-Carry Adder

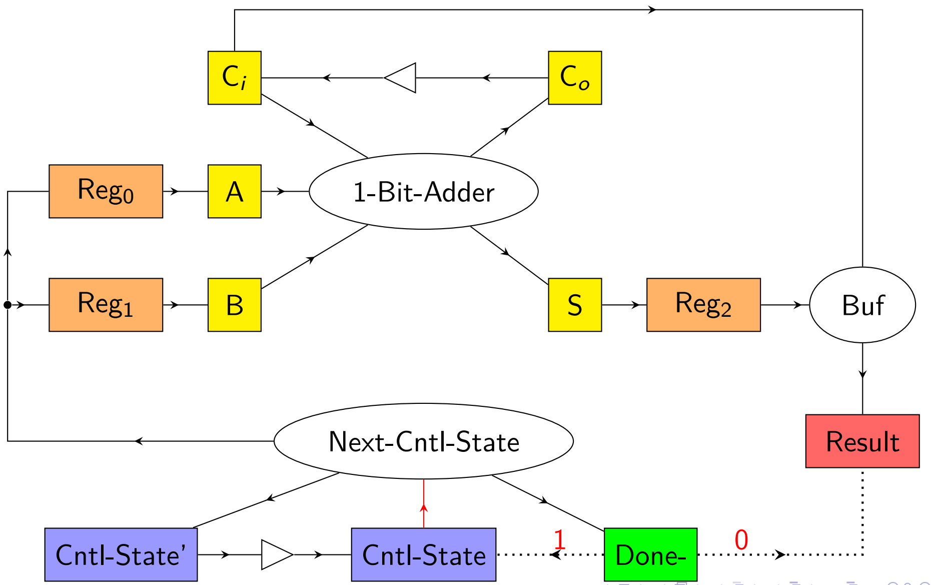


Ripple-Carry Adder

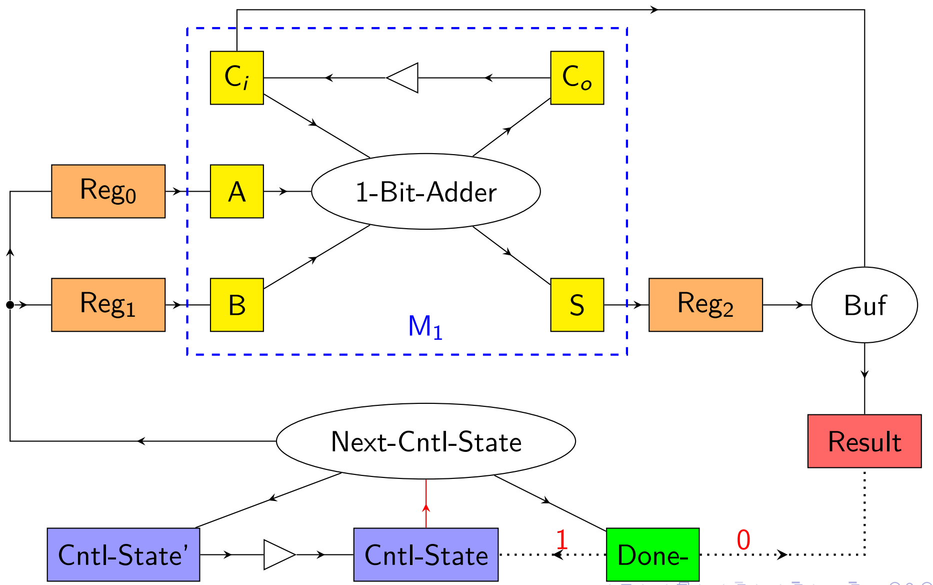


Goal: Given an appropriate initial condition, prove that the asynchronous serial adder produces the same result with the ripple-carry adder.

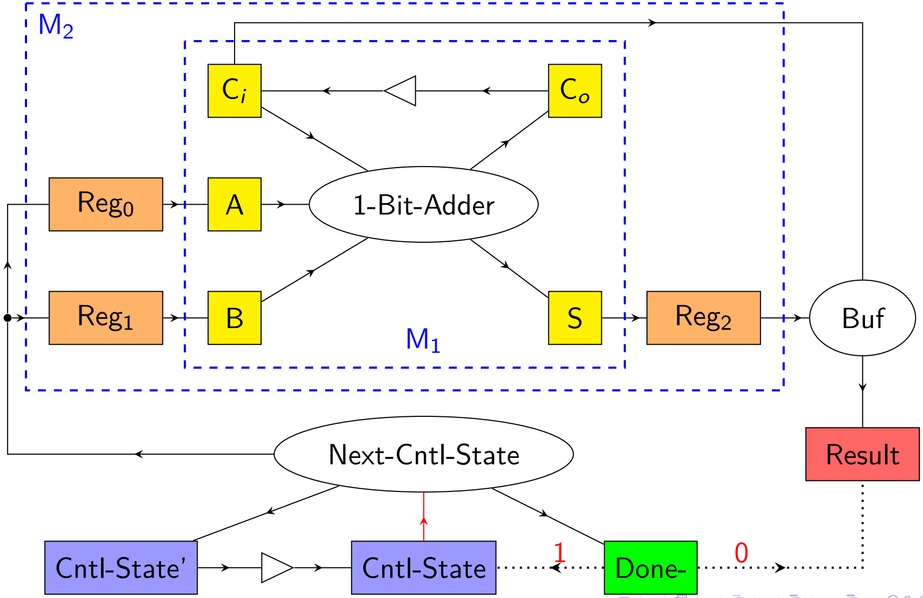
Serial Adder



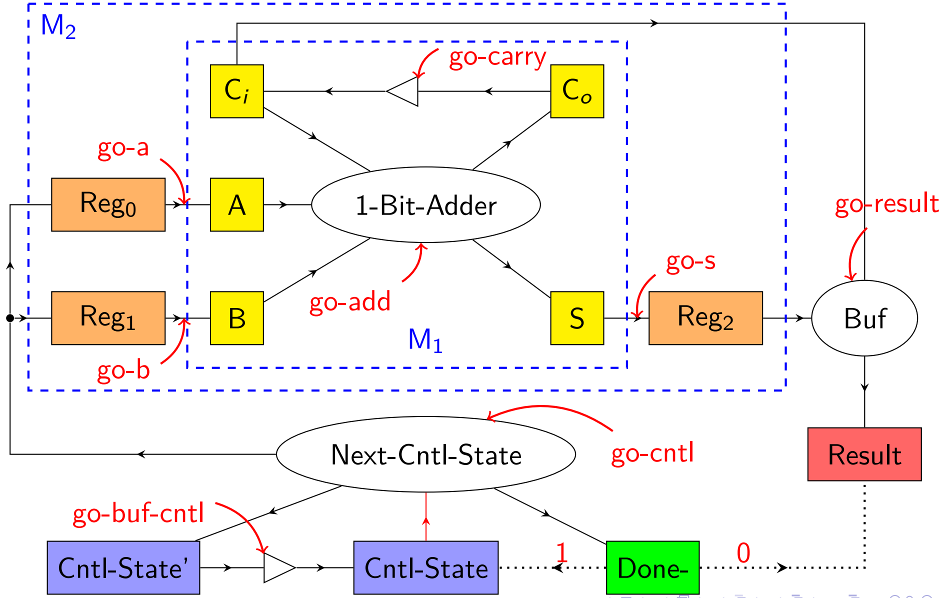
Serial Adder



Serial Adder



Serial Adder



Dependency

$$\begin{tikzcd}
\text{go-a} \arrow[dr] & & & & \\
& \text{go-add} \arrow[r] & \text{go-carry} \arrow[r] & \text{go-s} \arrow[dr] & \\
\text{go-b} \arrow[ur] & & & & \text{go-cntl} \arrow[uullll, red, bend right] \arrow[llll, red, bend left] \arrow[dl, red, bend left] \\
& & & \text{go-buf-cntl} \arrow[ur] &
\end{tikzcd}$$

Dependency

$$\begin{tikzcd}
\text{go-a} \arrow[dr] & & & & \\
& \text{go-add} \arrow[r] & \text{go-carry} \arrow[r] & \text{go-s} \arrow[dr] & \\
\text{go-b} \arrow[ur] & & & & \text{go-cntl} \arrow[uullll, bend right, red] \arrow[llll, bend left, red] \arrow[dl, bend left, red] \\
& & & \text{go-buf-cntl} \arrow[ur] &
\end{tikzcd}$$

$$\begin{tikzcd}
\text{go-a} \arrow[dr] & & & & \\
& \text{go-add} \arrow[r] & \text{go-carry} \arrow[r] & \text{go-s} \arrow[r] & \text{go-result} \\
\text{go-b} \arrow[ur] & & & &
\end{tikzcd}$$

Interleaving Example

Inputs = `((...  go-a0 go-b0) (...  go-a1 go-b1) ...)`

Interleaving Example

Inputs = `((... go-a0 go-b0) (... go-a1 go-b1) ...)`

- `(go-a go-b)` ⇒ `((... T F) (... go-a1 T) ...)`

Interleaving Example

Inputs = `((...  go-a0 go-b0) (...  go-a1 go-b1) ...)`

- `(go-a go-b)` ⇒ `((...  T F) (...  go-a1 T) ...)`
- `(go-b go-a)` ⇒ `((...  F T) (...  T go-b1) ...)`

Interleaving Example

Inputs = `((...  go-a0 go-b0) (...  go-a1 go-b1) ...)`

- `(go-a go-b)` ⇒ `((...  T F) (...  go-a1 T) ...)`
- `(go-b go-a)` ⇒ `((...  F T) (...  T go-b1) ...)`
- `((go-a go-b))` ⇒ `((...  T T) ...)`

Outline

Conclusions

We have presented our framework for modeling and verifying asynchronous circuits using the DE system.

Hierarchical verification is critical to circuit verification.

Reasoning with highly non-deterministic behavior is burdensome to asynchronous circuit verification.

References

W. Hunt (2000)
The DE Language
Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers Norwell, MA, USA, 151 – 166.

W. Hunt & E. Reeber (2006)
Applications of the DE2 Language
DCC 2006, Vienna, Austria.

B. Brock & W. Hunt (1997)
The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor
Formal Methods in System Design, 11, 71 – 104.

M. Roncken, S. Gilla, H. Park, N. Jamadagni, C. Cowan, I. Sutherland (2015)
Naturalized Communication and Testing
ASYNC 2015, 77 – 84.

A. Slobodova, J. Davis, S. Swords, and W. Hunt (2011)
A Flexible Formal Verification Framework for Industrial Scale Validation
MEMOCODE 2011, 89 – 97.

PhD Plan

Phase 1: Pick a topic

Phase 2: ?

Phase 3: Defend

Questions?