Hybrid Verification of a Hardware Modular Reduction Engine

Jun Sawada, Peter Sandon, Viresh Paruthi,
Jason Baumgartner,Michael Case, Hari Mony

IBM Austin Research Laboratory
IBM System and Technology Group

November 2, 2011

Outline

- Motivation
- Verification Tool
- Verification of Modular Reduction
- Results and Observation

A Brief Introduction to Cryptography

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \bmod N$

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \bmod N$
  - Modular inverse $A^{-1} \bmod N$

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \bmod N$
  - Modular inverse $A^{-1} \bmod N$
  - Modular exponentiation $A^{B} \bmod N$

A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \bmod N$
  - Modular inverse $A^{-1} \bmod N$
  - Modular exponentiation $A^B \bmod N$
  - Montgomery multiplier accelerates $A^B \bmod N$ computation.

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an *asymmetric math function accelerator*

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an *asymmetric math function accelerator*
  - Performs modular math for public key encryption.

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an *asymmetric math function accelerator*
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an *asymmetric math function accelerator*
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
  - Takes up to 4096-bit operands

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an *asymmetric math function accelerator*
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
  - Takes up to 4096-bit operands
  - Long delays: Thousands of clock cycles for a single operation

On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an *asymmetric math function accelerator*
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
  - Takes up to 4096-bit operands
  - Long delays: Thousands of clock cycles for a single operation
  - Implemented as a finite-state machine.

Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems

- Simulation is too slow to provide a decent coverage.

Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems

- Simulation is too slow to provide a decent coverage.
- Even post-silicon testing is slow because of slow reference model computation by software.

Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems

- Simulation is too slow to provide a decent coverage.
- Even post-silicon testing is slow because of slow reference model computation by software.
- Bit-level model-checking does not scale to thousands of cycles.

Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems

- Simulation is too slow to provide a decent coverage.
- Even post-silicon testing is slow because of slow reference model computation by software.
- Bit-level model-checking does not scale to thousands of cycles.
- Very time-consuming to analyze implementation details with a theorem prover.

Hybrid Verification Tool

- A hybrid verification tool is a combination of a model checker and a theorem prover.
    - e.g. Intel Forte based on symbolic trajectory evaluation.

Hybrid Verification Tool

- A hybrid verification tool is a combination of a model checker and a theorem prover.
  - e.g. Intel Forte based on symbolic trajectory evaluation.
- We believe the full potential of hybrid verification tools have not been utilized because:
  - Model checker is not tuned for this kind of proofs.
  - Theorem prover is hard-to-use and time-consuming for many engineers.

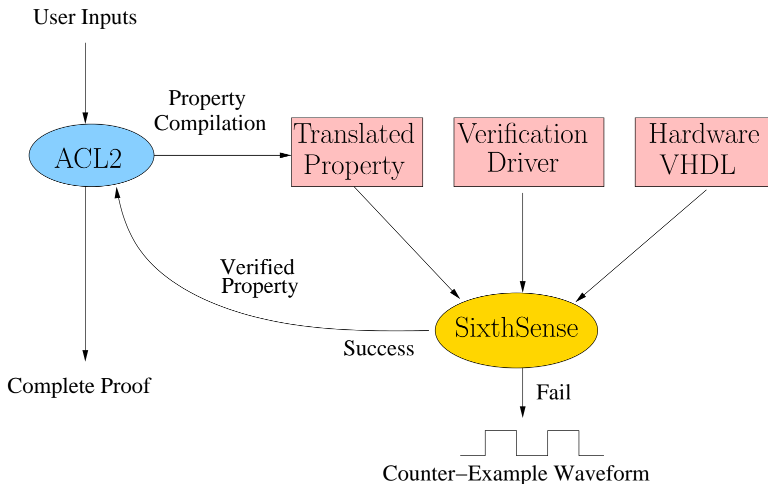
ACL2SIX

- Our tool *ACL2SIX* is a combination of
  - IBM SixthSense Formal Verification Tool (Model Checker)
  - ACL2 Theorem Prover

ACL2SIX

- Our tool *ACL2SIX* is a combination of
  - IBM SixthSense Formal Verification Tool (Model Checker)
  - ACL2 Theorem Prover
- ACL2SIX directly works on hardware given in HDL.
  - A quick translation of properties, not of hardware HDL.
  - The theorem prover does not deal with low-level details of hardware. The model checker abstracts them away.

ACL2SIX Platform Data Flow



ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
- Clock cycle is given by (variable + constant delay)

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
- Clock cycle is given by (variable + constant delay)
- Pre-defined and user-defined bit-vector functions can be used.

ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

```
(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six :hint '((:cycle-var n))))))
```

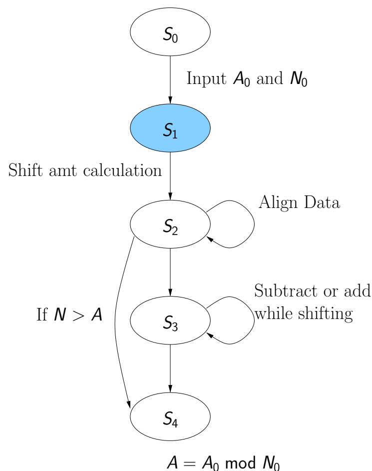
- Bit vectors are accessed by vhdl-sigvec with the syntax: (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
- Clock cycle is given by (variable + constant delay)
- Pre-defined and user-defined bit-vector functions can be used.
- Directive to call SixthSense from ACL2

Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00011100_2$$
$$N = 00000101_2$$

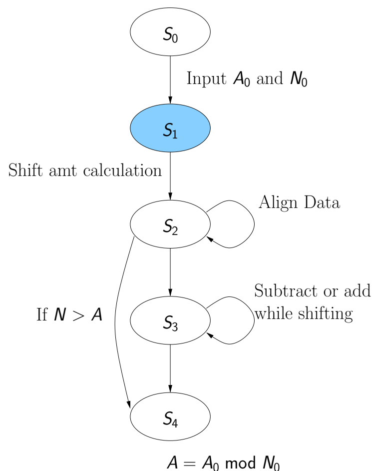


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00011100_2$$
$$N = 00000101_2$$

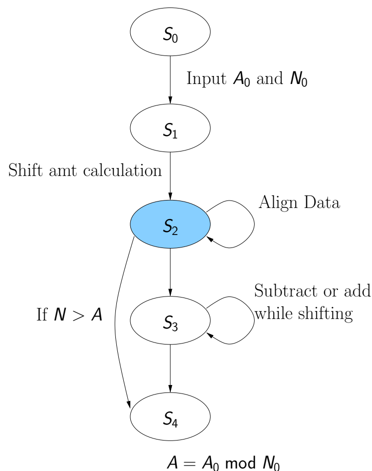


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00011100_2$$
$$N = 00000101_2$$

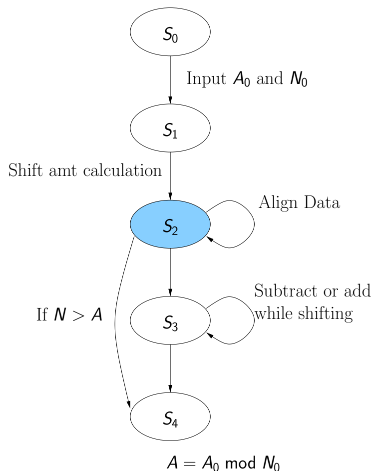


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00011100_2$$
$$N = 00001010_2$$

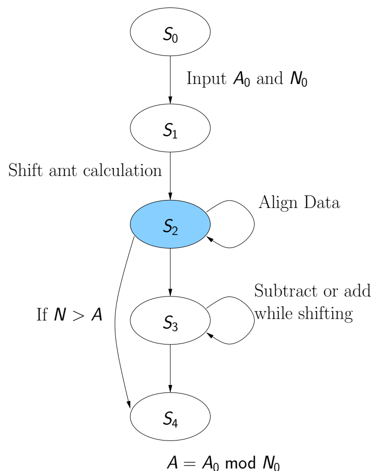


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00011100_2$$
$$N = 00010100_2$$

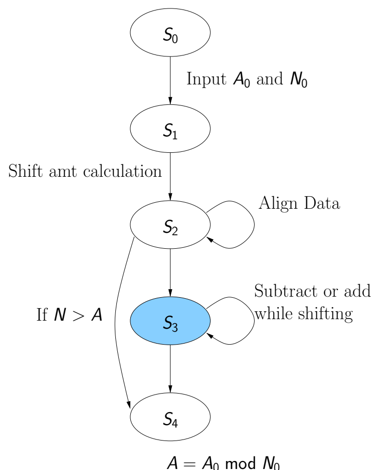


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00011100_2$$
$$N = 00010100_2$$

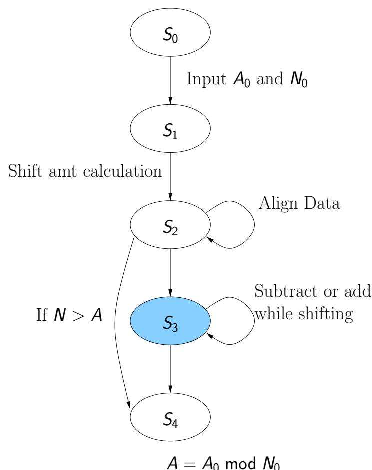


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00001000_2$$
$$N = 00010100_2$$

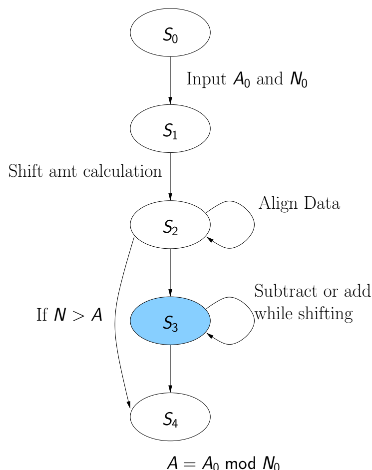


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00001000_2$$
$$N = 00001010_2$$

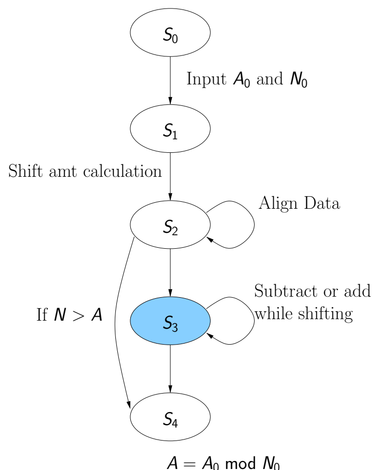


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 11111110_2$$
$$N = 00001010_2$$

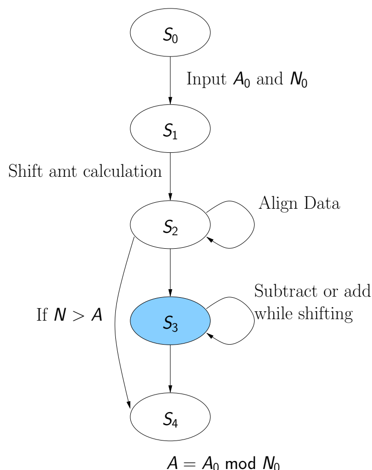


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 11111110_2$$
$$N = 00000101_2$$

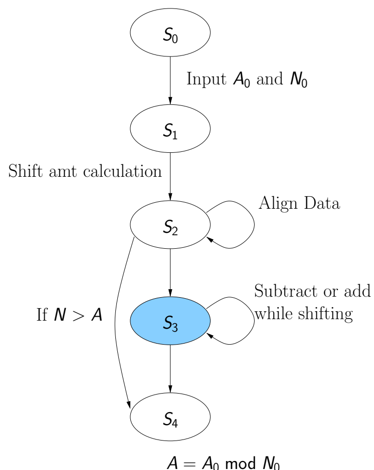


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00000011_2$$
$$N = 00000101_2$$

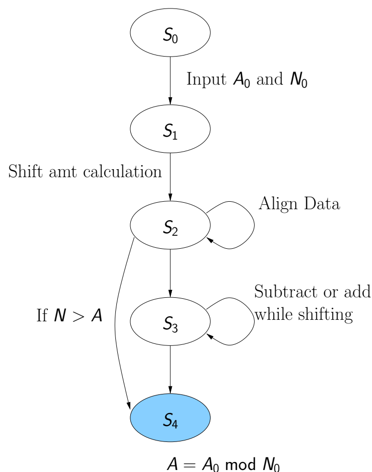


Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00000011_2$$
$$N = 00000101_2$$



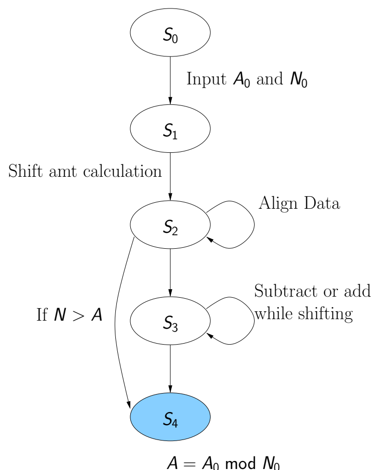
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00000011_2$$
$$N = 00000101_2$$

- Actual Operands are very long.



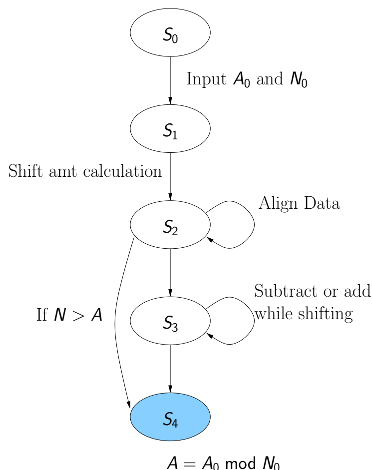
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00000011_2$$
$$N = 00000101_2$$

- Actual Operands are very long.
- Many arithmetic operations are repeated in each transition.



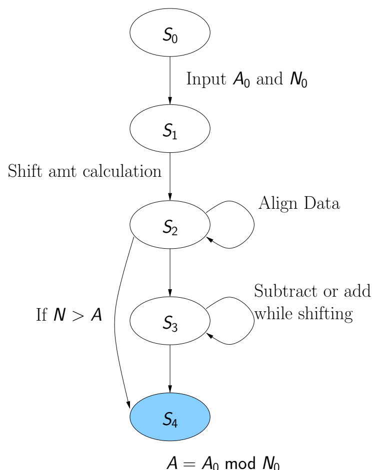
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute A_0 mod N_0.

- Example: compute 28 mod 5

$$A = 00000011_2$$
$$N = 00000101_2$$

- Actual Operands are very long.
- Many arithmetic operations are repeated in each transition.
- State transition takes fixed but long clock cycles.



Overall Approach to Verifying a State Transition Machine

- Use a divide-and-conquer approach.
  - Model checker is used to verify properties over each state transition.
  - Theorem prover is used to combine verified properties to form a complete proof, and also reason about high-level math.

Overall Approach to Verifying a State Transition Machine

- Use a divide-and-conquer approach.
  - Model checker is used to verify properties over each state transition.
  - Theorem prover is used to combine verified properties to form a complete proof, and also reason about high-level math.
- Make the model checker to work on bigger, more abstract sub-problems.
  - Hide the hardware details from the theorem prover.
  - Theorem prover requires smaller steps to create a proof.

How Should We Write Properties over State Transition?

- Typical state transition with pre-condition P_i and post-condition P_{i+1}:

$$P_i(n) \implies P_{i+1}(n + \Delta_i)$$

  - Δ_i is typically constant over 10 but less than 100.

How Should We Write Properties over State Transition?

- Typical state transition with pre-condition P_i and post-condition P_{i+1}:

$$P_i(n) \implies P_{i+1}(n + \Delta_i)$$

  - Δ_i is typically constant over 10 but less than 100.
- Actual conditions are written at high-level.
  - e.g. Multi-word subtraction is simply written as $A - N$ in P_i. The hardware may repeat multiple subtractions over discontinuous data.

How Should We Write Properties over State Transition?

- Typical state transition with pre-condition P_i and post-condition P_{i+1}:

$$P_i(n) \implies P_{i+1}(n + \Delta_i)$$

  - Δ_i is typically constant over 10 but less than 100.
- Actual conditions are written at high-level.
  - e.g. Multi-word subtraction is simply written as $A - N$ in P_i. The hardware may repeat multiple subtractions over discontinuous data.
- Frequently, we need to add global and state invariants to prove

$$(\mathrm{inv}(n) \wedge \mathrm{cond}_i(n) \wedge P_i(n)) \implies P_{i+1}(n + \Delta_i)$$

  - Invariant definitions are in VHDL and hidden from theorem prover.

Algorithm to verify $P_i(n) \implies P_{i+1}(n+\Delta_i)$

Algorithm

1. Convert $P_i(n) \implies P_{i+1}(n+\Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.

Algorithm to verify $P_i(n) \implies P_{i+1}(n+\Delta_i)$

Algorithm

1. Convert $P_i(n) \implies P_{i+1}(n+\Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.
2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q'_i(n)$. If $Q'_i(n) = T$, return.

Algorithm to verify $P_i(n) \implies P_{i+1}(n+\Delta_i)$

Algorithm

1. Convert $P_i(n) \implies P_{i+1}(n+\Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.
2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q_i'(n)$. If $Q_i'(n) = T$, return.
3. Prove $Q_i'(n)$ by k-induction. Base cases are proved by BMC. Inductive step is proved:
   $Q_i(n) \wedge Q_i(n+1) \wedge \cdots \wedge Q_i(n+k-1) \implies Q_i(n+k)$.

Algorithm to verify $P_i(n) \implies P_{i+1}(n + \Delta_i)$

Algorithm

1. Convert $P_i(n) \implies P_{i+1}(n + \Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.
2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q_i'(n)$. If $Q_i'(n) = T$, return.
3. Prove $Q_i'(n)$ by k-induction. Base cases are proved by BMC. Inductive step is proved:
   $Q_i(n) \wedge Q_i(n+1) \wedge \cdots \wedge Q_i(n+k-1) \implies Q_i(n+k)$.
4. Increase k and repeat Step 3.

Algorithm to verify $P_i(n) \implies P_{i+1}(n+\Delta_i)$

Algorithm

1. Convert $P_i(n) \implies P_{i+1}(n+\Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.
2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q'_i(n)$. If $Q'_i(n) = T$, return.
3. Prove $Q'_i(n)$ by k-induction. Base cases are proved by BMC. Inductive step is proved:
$Q_i(n) \wedge Q_i(n+1) \wedge \cdots \wedge Q_i(n+k-1) \implies Q_i(n+k)$.
4. Increase k and repeat Step 3.

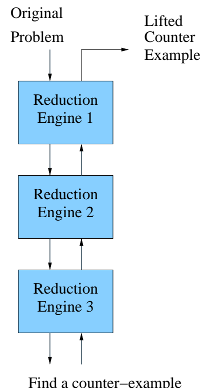
Step 1 is performed by the theorem prover. Step 2-4 by the model checker.

Generation of Counter-Examples for Induction Proof

- Often an induction proof fails and a counter-example helps debugging.

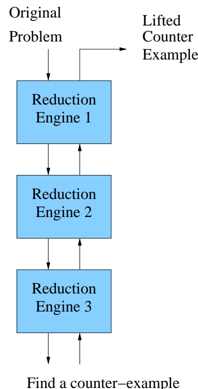
Generation of Counter-Examples for Induction Proof

- Often an induction proof fails and a counter-example helps debugging.
- Counter-example generation is difficult for transformation-based verification tool like SixthSense.
  - An inductive counter-example does not start with an initial state.
  - Some information is lost during transformation.



Generation of Counter-Examples for Induction Proof

- Often an induction proof fails and a counter-example helps debugging.
- Counter-example generation is difficult for transformation-based verification tool like SixthSense.
    - An inductive counter-example does not start with an initial state.
    - Some information is lost during transformation.
- Implemented a trace lifting to reflect true root cause of induction failure.



Verification Results of Modular Reduction

Data Width	56-bit	256-bit	384-bit	512-bit
Total Time	10442s	20646s	37607s	98199s
Theorem Prover Time	257s	289s	474s	1690s
Property Check Time	10188s	20261s	37139s	97012s
Avg. Time per Prop.	118s	151s	223s	489s
Max Time per Prop.	138s	368s	1232s	3456s

- We finished modular reduction proof up to 512-bit.

Verification Results of Modular Reduction

Data Width	56-bit	256-bit	384-bit	512-bit
Total Time	10442s	20646s	37607s	98199s
Theorem Prover Time	257s	289s	474s	1690s
Property Check Time	10188s	20261s	37139s	97012s
Avg. Time per Prop.	118s	151s	223s	489s
Max Time per Prop.	138s	368s	1232s	3456s

- We finished modular reduction proof up to 512-bit.
- 1024-bit operation has properties that time-out in 24 hours.

Verification Results of Modular Reduction

Data Width	56-bit	256-bit	384-bit	512-bit
Total Time	10442s	20646s	37607s	98199s
Theorem Prover Time	257s	289s	474s	1690s
Property Check Time	10188s	20261s	37139s	97012s
Avg. Time per Prop.	118s	151s	223s	489s
Max Time per Prop.	138s	368s	1232s	3456s

- We finished modular reduction proof up to 512-bit.
- 1024-bit operation has properties that time-out in 24 hours.
- Individual property time increases rapidly as both state transition delay and input data increase.

Verification Results of Modular Reduction

Data Width	56-bit	256-bit	384-bit	512-bit
Total Time	10442s	20646s	37607s	98199s
Theorem Prover Time	257s	289s	474s	1690s
Property Check Time	10188s	20261s	37139s	97012s
Avg. Time per Prop.	118s	151s	223s	489s
Max Time per Prop.	138s	368s	1232s	3456s

- We finished modular reduction proof up to 512-bit.
- 1024-bit operation has properties that time-out in 24 hours.
- Individual property time increases rapidly as both state transition delay and input data increase.
- Most time spent in the model checker.

Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier

Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
- Analysis of modular inverse uncovered an overflow problem.

Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
- Analysis of modular inverse uncovered an overflow problem.
- The key is to use a powerful model checker to verify a larger sub-problems. Reduced theorem proving effort.

Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
- Analysis of modular inverse uncovered an overflow problem.
- The key is to use a powerful model checker to verify a larger sub-problems. Reduced theorem proving effort.
- Still full 4096-bits operation is hard to verify. Need to improve model checker for this type of proof.

Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
- Analysis of modular inverse uncovered an overflow problem.
- The key is to use a powerful model checker to verify a larger sub-problems. Reduced theorem proving effort.
- Still full 4096-bits operation is hard to verify. Need to improve model checker for this type of proof.
- Theorem proving is still a bottleneck to apply in an industrial setting. Need more automation or more productivity.