Formal Specification and Verification of the FM9001 Microprocessor Using the DE System

Cuong Chau

ckcuong@cs.utexas.edu

Department of Computer Science

The University of Texas at Austin

May 23, 2017

Outline

Outline

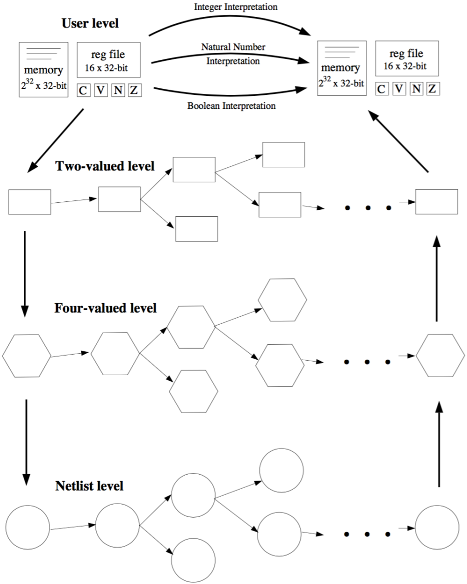
Introduction

FM9001 is a general-purpose 32-bit microprocessor whose gate-level netlist was originally specified and verified in the Nqthm logic using the DUAL-EVAL system [Brock & Hunt:1997].

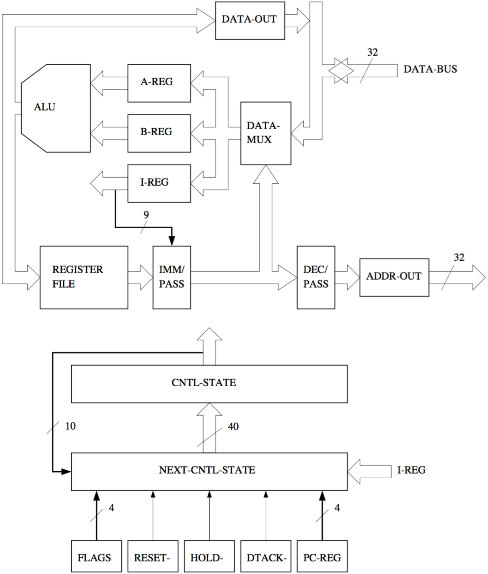
We re-specify and re-verify the FM9001 netlist in the ACL2 logic using the DE system.

Motivation: This work provides a library of verified hardware circuit generators that can be applied when reasoning about the synthesis of hardware circuits using DE.

FM9001 Specification Levels



Block Diagram of the FM9001



FM9001 Verification

The proof of correctness of the FM9001 gate-level design consists of three major lemmas:

1. The FM9001 can be forced to a known state, i.e., reset, from **any initial state** by a suitable sequence of inputs.
2. Given a set of initial conditions, the gate-level model correctly implements the high-level instruction interpreter.
3. The state at the end of the reset sequence satisfies the initial conditions for the previous lemma.

Strategy:

Prove that the desired reset state can be reached from an initial state of all `X` (unknown) values.

By monotonicity of the DE semantics, we then prove that the desired reset state can be reached from **any initial state**.

FM9001 Verification

The proof of correctness of the FM9001 gate-level design consists of three major lemmas:

1. The FM9001 can be forced to a known state, i.e., reset, from **any initial state** by a suitable sequence of inputs.
2. Given a set of initial conditions, the gate-level model correctly implements the high-level instruction interpreter.
3. The state at the end of the reset sequence satisfies the initial conditions for the previous lemma.

Strategy:

Prove that the desired reset state can be reached from an initial state of all X (unknown) values.

By monotonicity of the DE semantics, we then prove that the desired reset state can be reached from **any initial state**.

FM9001 Verification

The proof of correctness of the FM9001 gate-level design consists of three major lemmas:

1. The FM9001 can be forced to a known state, i.e., reset, from **any initial state** by a suitable sequence of inputs.
2. Given a set of initial conditions, the gate-level model correctly implements the high-level instruction interpreter.
3. The state at the end of the reset sequence satisfies the initial conditions for the previous lemma.

Strategy:

Prove that the desired reset state can be reached from an initial state of all `X` (unknown) values.

By monotonicity of the DE semantics, we then prove that the desired reset state can be reached from **any initial state**.

Challenge

The original work modeled the memory model using Nqthm's shell principle.

- There is no such principle in ACL2.

Challenge

The original work modeled the memory model using Nqthm's shell principle.

- There is no such principle in ACL2.

Need a different approach to formalizing the memory model for FM9001.

Approach

The original work used Nqthm's shell principle to introduce three new data structures for a memory cell:

1. ROM tags **read-only** locations of the memory.
2. RAM tags **read-write** locations of the memory.
3. STUB represents **"unimplemented"** portions.

Approach

The original work used Nqthm's shell principle to introduce three new data structures for a memory cell:

1. ROM tags **read-only** locations of the memory.
2. RAM tags **read-write** locations of the memory.
3. STUB represents **"unimplemented"** portions.

Our approach: Represent a memory cell as a proper list of two elements:

1. The first element is a flag specifying the **memory type** of the cell (i.e., ROM, or RAM, or STUB).
2. The second element is the **value** of the cell.

Approach

The original work used Nqthm's shell principle to introduce three new data structures for a memory cell:

1. ROM tags **read-only** locations of the memory.
2. RAM tags **read-write** locations of the memory.
3. STUB represents **"unimplemented"** portions.

Our approach: Represent a memory cell as a proper list of two elements:

1. The first element is a flag specifying the **memory type** of the cell (i.e., ROM, or RAM, or STUB).
2. The second element is the **value** of the cell.

This change does not affect the proof strategy for FM9001 created in the previous work, except for establishing the monotonicity property for DE, which is part of the FM9001 verification procedure.

Outline

The DE Language

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines [Hunt:2000].

The DE Language

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines [Hunt:2000].

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 developed in ACL2 for describing Mealy machines [Hunt:2000].

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

The operational semantics for the DE language is implemented as an **output evaluator**, `se`, and a **state evaluator**, `de`.

The DE Language

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines [Hunt:2000].

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

The operational semantics for the DE language is implemented as an **output evaluator**, `se`, and a **state evaluator**, `de`.

- The `se` function evaluates a module and returns its **outputs** as a function of its inputs and its internal state.

The DE Language

DE is a formal occurrence-oriented hardware description language developed in ACL2 for describing Mealy machines [Hunt:2000].

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

The operational semantics for the DE language is implemented as an **output evaluator**, `se`, and a **state evaluator**, `de`.

- The `se` function evaluates a module and returns its **outputs** as a function of its inputs and its internal state.
- The `de` function evaluates a module and returns its **next state**; this state will be structurally identical to the module's current state, but with updated values.

Outline

FM9001 Verification

The proof of correctness of the FM9001 gate-level design consists of three major lemmas:

1. The FM9001 can be forced to a known state, i.e., reset, from **any initial state** by a suitable sequence of inputs.
2. Given a set of initial conditions, the gate-level model correctly implements the high-level instruction interpreter.
3. The state at the end of the reset sequence satisfies the initial conditions for the previous lemma.

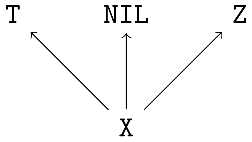
Strategy:

Prove that the desired reset state can be reached from an initial state of all `X` (unknown) values.

By monotonicity of the DE semantics, we then prove that the desired reset state can be reached from **any initial state**.

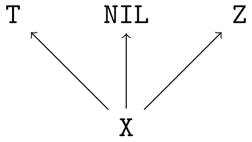
Monotonicity

We define a **partial ordering** with a binary relation $\leq$ over the four-valued constants: $a \leq b$ if $a = b$ or $a =$ X.



Monotonicity

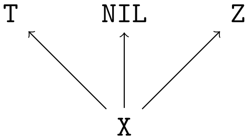
We define a **partial ordering** with a binary relation $\leq$ over the four-valued constants: $a \leq b$ if $a = b$ or $a = \mathtt{X}$.



A function $f(x)$ is monotonic if $a \leq b \Rightarrow f(a) \leq f(b)$.

Monotonicity

We define a **partial ordering** with a binary relation $\leq$ over the four-valued constants: $a \leq b$ if $a = b$ or $a = \mathtt{X}$.

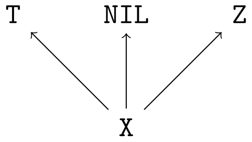


A function $f(x)$ is monotonic if $a \leq b \Rightarrow f(a) \leq f(b)$.

A function $f(x_1, x_2, ..., x_n)$ is monotonic if
$a_1 \leq b_1$ & $a_2 \leq b_2$ & ... & $a_n \leq b_n \Rightarrow f(a_1, a_2, ..., a_n) \leq f(b_1, b_2, ..., b_n)$.

Monotonicity

We define a **partial ordering** with a binary relation $\leq$ over the four-valued constants: $a \leq b$ if $a = b$ or $a = \texttt{X}$.



A function $f(x)$ is monotonic if $a \leq b \Rightarrow f(a) \leq f(b)$.

A function $f(x_1, x_2, ..., x_n)$ is monotonic if
$a_1 \leq b_1 \;\&\; a_2 \leq b_2 \;\&\; ... \;\&\; a_n \leq b_n \Rightarrow f(a_1, a_2, ..., a_n) \leq f(b_1, b_2, ..., b_n)$.

Primitive four-valued logic functions (e.g., `F-AND`, `F-OR`, `F-NOT`, `F-XOR`) are monotonic.

Monotonicity of DE

Given two states `st1` and `st2`, the relation `st1` $\leq$ `st2` can be loosely interpreted that `st2` may differ from `st1` only by replacing `X` values in `st1` with any values. We call `st1` approximates `st2`.

Monotonicity of DE

Given two states `st1` and `st2`, the relation `st1` $\leq$ `st2` can be loosely interpreted that `st2` may differ from `st1` only by replacing `X` values in `st1` with any values. We call `st1` approximates `st2`.

`st1` $\leq$ `st2`

$\Rightarrow$

`(de fn ins st1 netlist)` $\leq$ `(de fn ins st2 netlist)`

Monotonicity of DE

Given two states `st1` and `st2`, the relation `st1` $\leq$ `st2` can be loosely interpreted that `st2` may differ from `st1` only by replacing `X` values in `st1` with any values. We call `st1` approximates `st2`.

`st1` $\leq$ `st2`

$\Rightarrow$

`(de fn ins st1 netlist)` $\leq$ `(de fn ins st2 netlist)`

$\Rightarrow$

`(run fn ins-seq st1 netlist)` $\leq$ `(run fn ins-seq st2 netlist)`

Monotonicity of DE

Given two states `st1` and `st2`, the relation `st1` $\leq$ `st2` can be loosely interpreted that `st2` may differ from `st1` only by replacing `X` values in `st1` with any values. We call `st1` approximates `st2`.

`st1` $\leq$ `st2`

$\Rightarrow$

`(de fn ins st1 netlist)` $\leq$ `(de fn ins st2 netlist)`

$\Rightarrow$

`(run fn ins-seq st1 netlist)` $\leq$ `(run fn ins-seq st2 netlist)`

If `(run fn ins-seq st1 netlist)` contains no `X` value, then
`(run fn ins-seq st1 netlist)` $=$ `(run fn ins-seq st2 netlist)`

Monotonicity of DE

Given two states st1 and st2, the relation st1 $\leq$ st2 can be loosely interpreted that st2 may differ from st1 only by replacing X values in st1 with any values. We call st1 approximates st2.

st1 $\leq$ st2
$\Rightarrow$
(de fn ins st1 netlist) $\leq$ (de fn ins st2 netlist)
$\Rightarrow$
(run fn ins-seq st1 netlist) $\leq$ (run fn ins-seq st2 netlist)

If (run fn ins-seq st1 netlist) contains no X value, then
(run fn ins-seq st1 netlist) = (run fn ins-seq st2 netlist)

If st1 is contains only X values, and (run fn ins-seq st1 netlist) is the desired **reset state**, then this state can be reached from any state st2.

State Approximation

The state approximation notion is changed under our proposed representation of the memory model.

State Approximation

The state approximation notion is changed under our proposed representation of the memory model.

Below is the ACL2 version of the state approximation definition introduced in the previous work.

```
(defun s-approx (s1 s2)
  (cond ((or (consp s1) (consp s2))                    ;; (1)
         (if (consp s1)
             (if (consp s2)
                 (and (s-approx (car s1) (car s2))
                      (s-approx (cdr s1) (cdr s2)))
               nil) nil))
        ((or (ramp s1) (ramp s2)) ...)                 ;; (2)
        ((or (romp s1) (romp s2)) ...)                 ;; (3)
        ((or (stubp s1) (stubp s2)) ...)               ;; (4)
        (t ...)))
```

State Approximation

The state approximation notion is changed under our proposed representation of the memory model.

Below is the ACL2 version of the state approximation definition introduced in the previous work.

```
(defun s-approx (s1 s2)
  (cond ((or (consp s1) (consp s2))                        ;; (1)
         (if (consp s1)
             (if (consp s2)
                 (and (s-approx (car s1) (car s2))
                      (s-approx (cdr s1) (cdr s2)))
               nil) nil))
        ((or (ramp s1) (ramp s2)) ...)                     ;; (2)
        ((or (romp s1) (romp s2)) ...)                     ;; (3)
        ((or (stubp s1) (stubp s2)) ...)                   ;; (4)
        (t ...)))
```

Memory cells are defined as CONSES: cases (2), (3), and (4) in the above definition will never be satisfied. They are all subsumed in case (1).

State Approximation

We change the state approximation definition by rearranging the order of cases to (2), (3), (4), and (1).

```
(defun s-approx (s1 s2)
  (cond ((or (ramp s1) (ramp s2)) ...)                     ;; (2)
        ((or (romp s1) (romp s2)) ...)                     ;; (3)
        ((or (stubp s1) (stubp s2)) ...)                   ;; (4)
        ((or (consp s1) (consp s2))                        ;; (1)
         (if (consp s1)
             (if (consp s2)
                 (and (s-approx (car s1) (car s2))
                      (s-approx (cdr s1) (cdr s2)))
               nil) nil))
        (t ...)))
```

Monotonicity of DE

We need the following property in order to establish the monotonicity property for DE.

```
(implies (s-approx s1 s2)
         (s-approx (cdr s1) (cdr s2)))
```

Monotonicity of DE

We need the following property in order to establish the monotonicity property for DE.

```
(implies (s-approx s1 s2)
         (s-approx (cdr s1) (cdr s2)))
```

The above property holds when we impose a constraint on the **value of each memory cell** that it must be a four-valued vector.

Monotonicity of DE

We need the following property in order to establish the monotonicity property for DE.

```
(implies (s-approx s1 s2)
         (s-approx (cdr s1) (cdr s2)))
```

The above property holds when we impose a constraint on the **value of each memory cell** that it must be a four-valued vector.

- This constraint does not affect the correctness proofs for FM9001 since the FM9001 specification enforces a restriction that only bit vectors are stored in memory.

Monotonicity of DE

We need the following property in order to establish the monotonicity property for DE.

```
(implies (s-approx s1 s2)
         (s-approx (cdr s1) (cdr s2)))
```

The above property holds when we impose a constraint on the **value of each memory cell** that it must be a four-valued vector.

- This constraint does not affect the correctness proofs for FM9001 since the FM9001 specification enforces a restriction that only bit vectors are stored in memory.

We establish the monotonicity property for DE with stricter hypotheses: **the structures of states and netlist must be syntactically well-formed.**

Outline

Conclusion

We successfully verify the correctness the FM9001 microprocessor design.

- This work provides a library of verified hardware circuit generators that can be applied when reasoning about the synthesis of hardware circuits using DE.
- We also verify guards for the DE system.

Conclusion

We successfully verify the correctness the FM9001 microprocessor design.

- This work provides a library of verified hardware circuit generators that can be applied when reasoning about the synthesis of hardware circuits using DE.
- We also verify guards for the DE system.

This work is also a contribution to ACL2 for two reasons.

- First, it moves into the ACL2 regression suite one of the most important theorems proved by Nqthm.
- Second, it is the first step toward porting the entire Computational Logic verified stack [Bevier et al.:1989, Moore:1996] from Nqthm to ACL2.

References

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

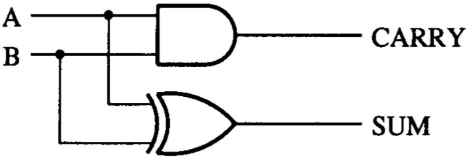
B. Brock & W. Hunt (1997)
The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor
Formal Methods in System Design, 11, 71 – 104.

W. R. Bevier and Hunt, Jr., W. A. and J S. Moore and W. D. Young (1989)
Special Issue on System Verification
Journal of Automated Reasoning, 5(4), 409 – 530.

J S. Moore (1996)
Piton: A Mechanically Verified Assembly-Level Language
Automated Reasoning Series, Kluwer Academic Publishers.

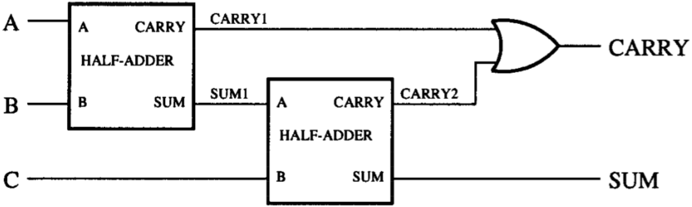
Questions?

Half-Adder



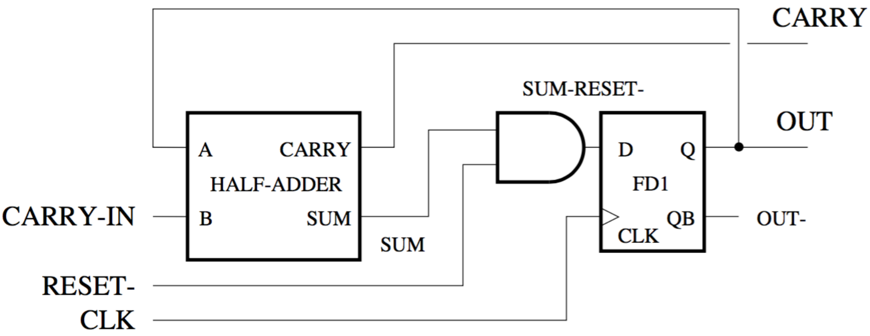

```
(defconst *half-adder*
  '((half-adder     ;; module name
     (a b)          ;; module inputs
     (sum carry)    ;; module outputs
     ()             ;; internal states
     ;; occurrences
     ((g0           ;; occurrence name
       (sum)        ;; occurrence outputs
       b-xor        ;; primitive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

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

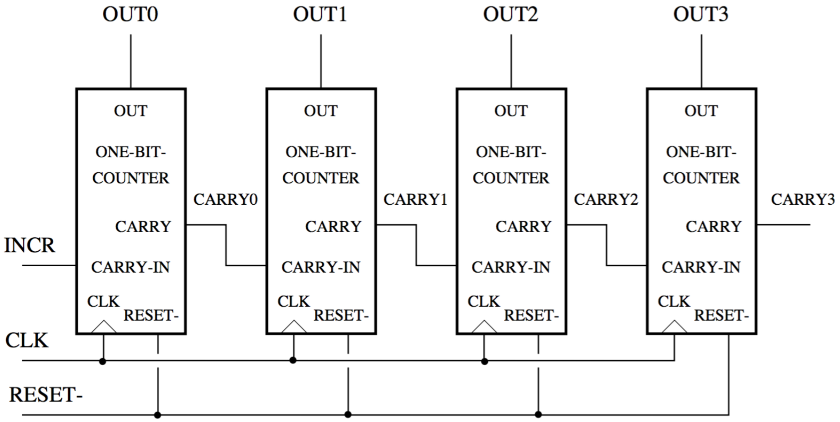


One-Bit Counter

```
(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*))
```

Four-Bit Counter



Four-Bit Counter

```
(defconst *four-bit-counter*
  (cons
   '(four-bit-counter
     (clk incr reset-)
     (out0 out1 out2 out3)
     (h0 h1 h2 h3)
     ((h0 (out0 carry0) one-bit-counter (clk incr   reset-))
      (h1 (out1 carry1) one-bit-counter (clk carry0 reset-))
      (h2 (out2 carry2) one-bit-counter (clk carry1 reset-))
      (h3 (out3 carry3) one-bit-counter (clk carry2 reset-))
      ))

   *one-bit-counter*))
```