An Overview of the DE Hardware Description Language and Its Application in Formal Verification of the FM9001 Microprocessor

Cuong Chau

ckcuong@cs.utexas.edu

Department of Computer Science

The University of Texas at Austin

September 22, 2016

Outline

Outline

Introduction

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

DE has shown to be a valuable tool in formal specification and verification of modern hardware designs [W. Hunt & E. Reeber, 2006].

Introduction

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

DE has shown to be a valuable tool in formal specification and verification of modern hardware designs [W. Hunt & E. Reeber, 2006].

In this talk, I will give an overview of the DE language, illustrate how to use it to formally specify and verify circuit designs via simple examples, and finally briefly describe its application in the FM9001 microprocessor verification.

Outline

The DE Language

DE is a hierarchical, occurrence-oriented simulator for 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 hierarchical, occurrence-oriented simulator for 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 hierarchical, occurrence-oriented simulator for 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

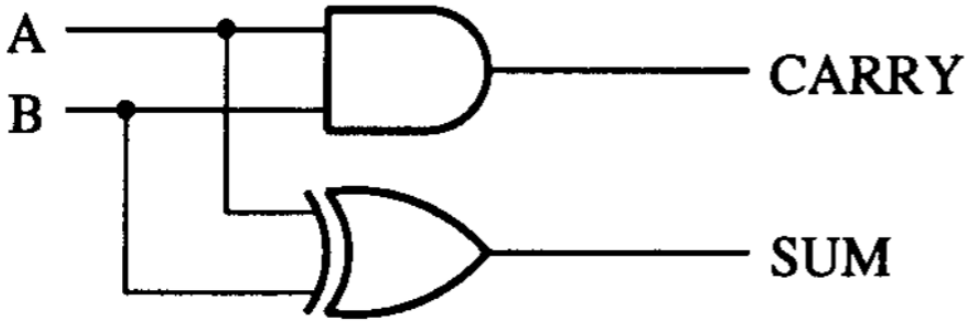
DE is a hierarchical, occurrence-oriented simulator for Mealy machines. It allows hierarchical module definition, and multiple copies of a module are identified by reference (their appearance in an occurrence).

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



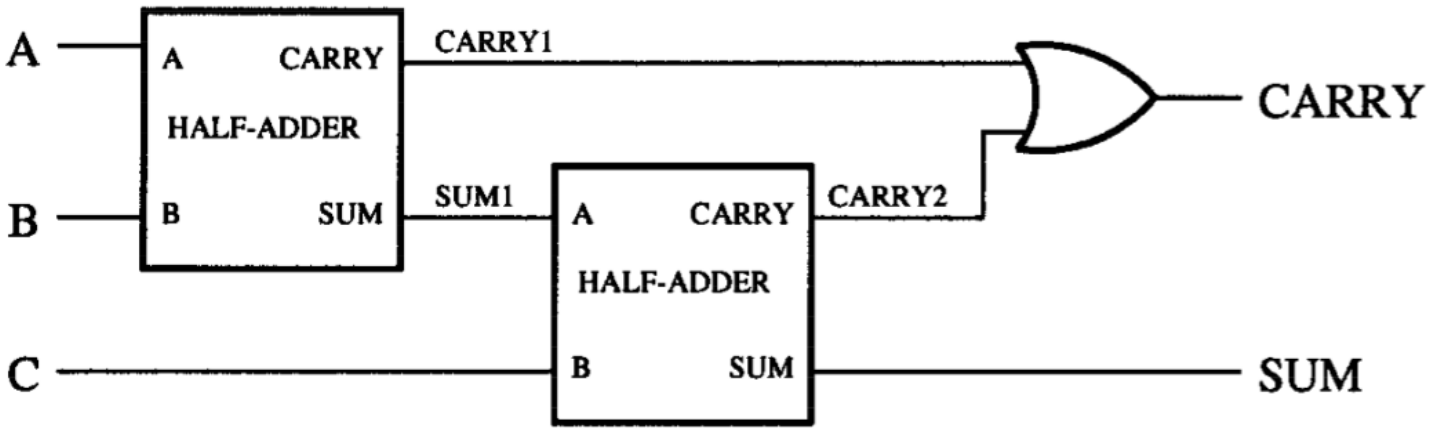

```
(defconst *half-adder*
  '((half-adder    ;; module name
     (a b)         ;; module inputs
     (sum carry)   ;; module outputs
     ()            ;; internal states
      ;; occurrences
     ((g0          ;; occurrence name
       (sum)       ;; occurrence outputs
       b-xor       ;; a primitive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

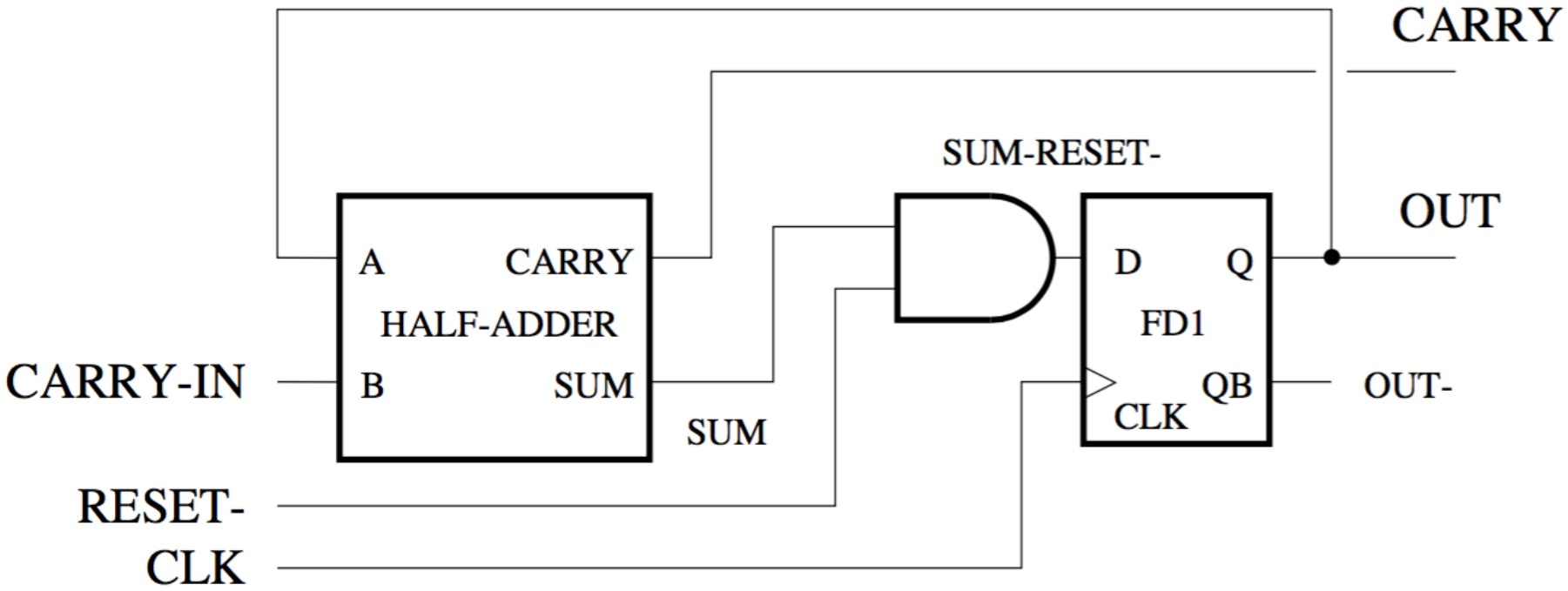

A
B
C
A
CARRY
HALF-ADDER
B
SUM
CARRY1
SUM1
A
CARRY
HALF-ADDER
B
SUM
CARRY2
CARRY
SUM

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

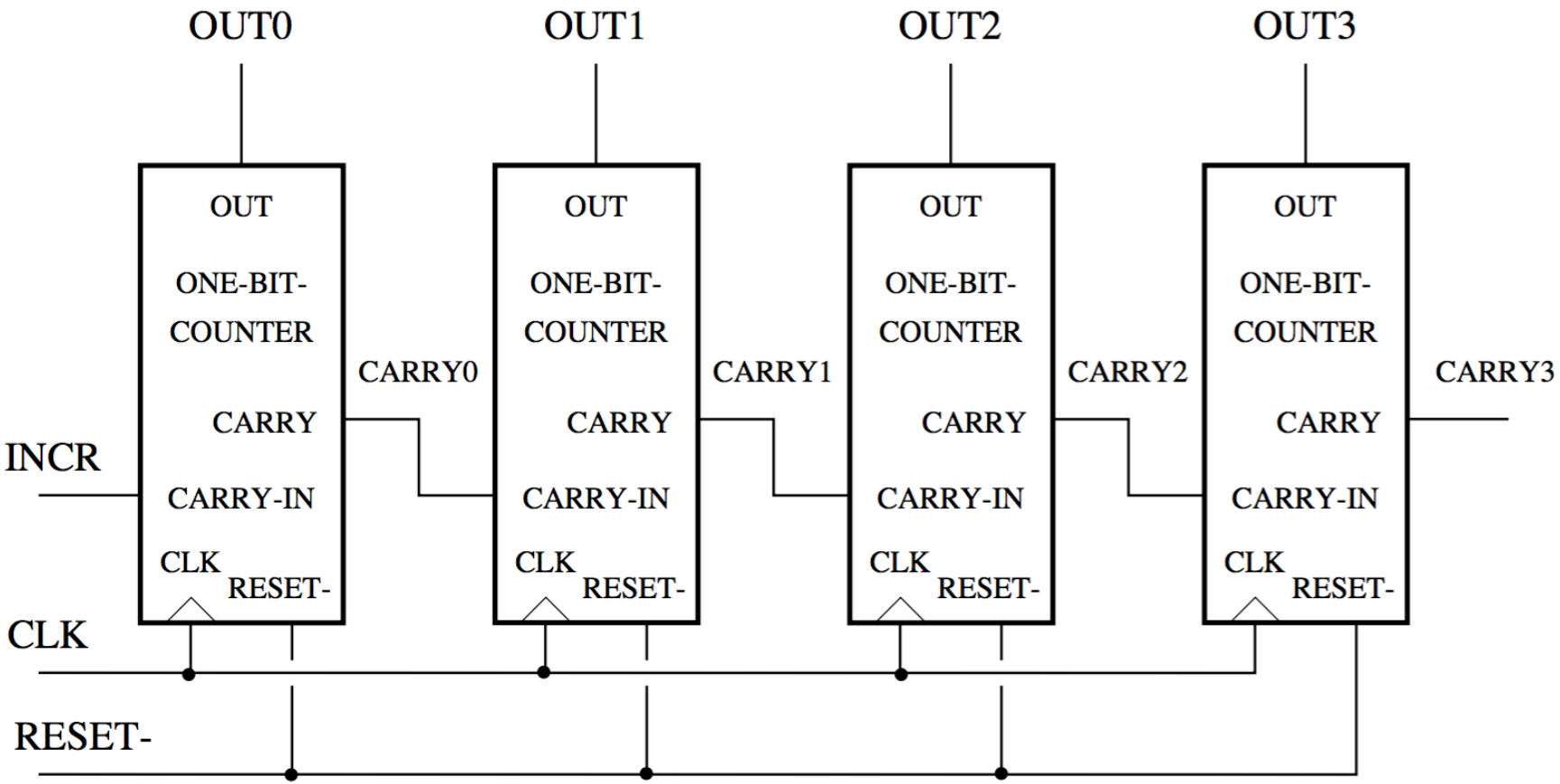


One-Bit Counter

```
(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*))
```

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*))
```

The DE Simulator

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's outputs and next state.

The DE Simulator

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's outputs and next 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 a 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 a module.

The DE Simulator

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's outputs and next 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 a 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 a module.

Demo.

Outline

Verifying DE-Specified Circuits

Each time a module is specified, there are two lemmas need to be proven: a **value lemma** specifying the module's outputs and a **state lemma** specifying the module's next state.

Verifying DE-Specified Circuits

Each time a module is specified, there are two lemmas need to be proven: 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.

Verifying DE-Specified Circuits

Each time a module is specified, there are two lemmas need to be proven: 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.

Verifying DE-Specified Circuits

Each time a module is specified, there are two lemmas need to be proven: 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.

Demo.

Circuit Generator Verification

The DE verification system can be applied to verify **circuit generators**.
Example: proving the correctness of a parameterized ripple-carry adder.

Circuit Generator Verification

The DE verification system can be applied to verify **circuit generators**.

Example: proving the correctness of a parameterized ripple-carry adder.

```
(defun v-adder (c a b)
  (declare (xargs :guard (and (true-listp a)
                              (true-listp b))))
   ;; c is a bit, a and b are bit-vectors of some length n;
   ;; this function returns a bit vector of length n+1.
  (if (atom a)
      (list (bool-fix c))
    (cons (b-xor3 c (car a) (car b))
          (v-adder (b-or3 (b-and (car a) (car b))
                          (b-and (car a) c)
                          (b-and (car b) c))
                   (cdr a)
                   (cdr b)))))
```

Circuit Generator Verification

The DE verification system can be applied to verify **circuit generators**.

Example: proving the correctness of a parameterized ripple-carry adder.

```
(defun v-adder (c a b)
  (declare (xargs :guard (and (true-listp a)
                              (true-listp b))))
   ;; c is a bit, a and b are bit-vectors of some length n;
   ;; this function returns a bit vector of length n+1.
  (if (atom a)
      (list (bool-fix c))
    (cons (b-xor3 c (car a) (car b))
          (v-adder (b-or3 (b-and (car a) (car b))
                          (b-and (car a) c)
                          (b-and (car b) c))
                   (cdr a)
                   (cdr b)))))
```

Demo.

Outline

FM9001 Microprocessor

The FM9001 is a general-purpose 32-bit microprocessor whose gate-level netlist was specified using the **DUAL-EVAL** hardware description language [B. Brock & W. Hunt, 1997].

The correctness of the FM9001 gate-level design was verified using the **NQTHM** theorem-proving system [B. Brock & W. Hunt, 1997].

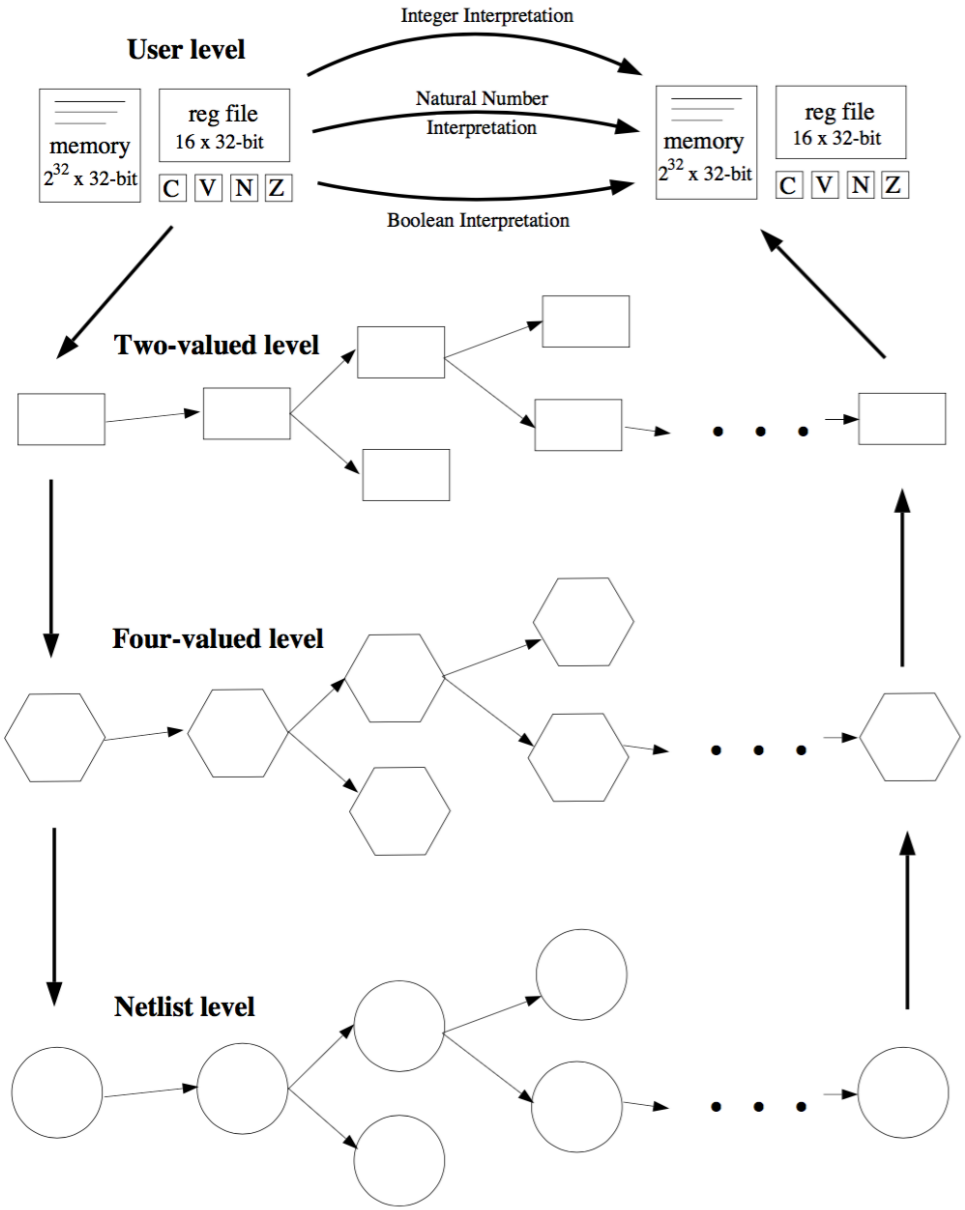
FM9001 Microprocessor

The FM9001 is a general-purpose 32-bit microprocessor whose gate-level netlist was specified using the **DUAL-EVAL** hardware description language [B. Brock & W. Hunt, 1997].

The correctness of the FM9001 gate-level design was verified using the **NQTHM** theorem-proving system [B. Brock & W. Hunt, 1997].

We have been re-specifying and re-verifying the correctness of the FM9001 design using the ACL2-based DE system.

FM9001 Specification Levels



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, 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.

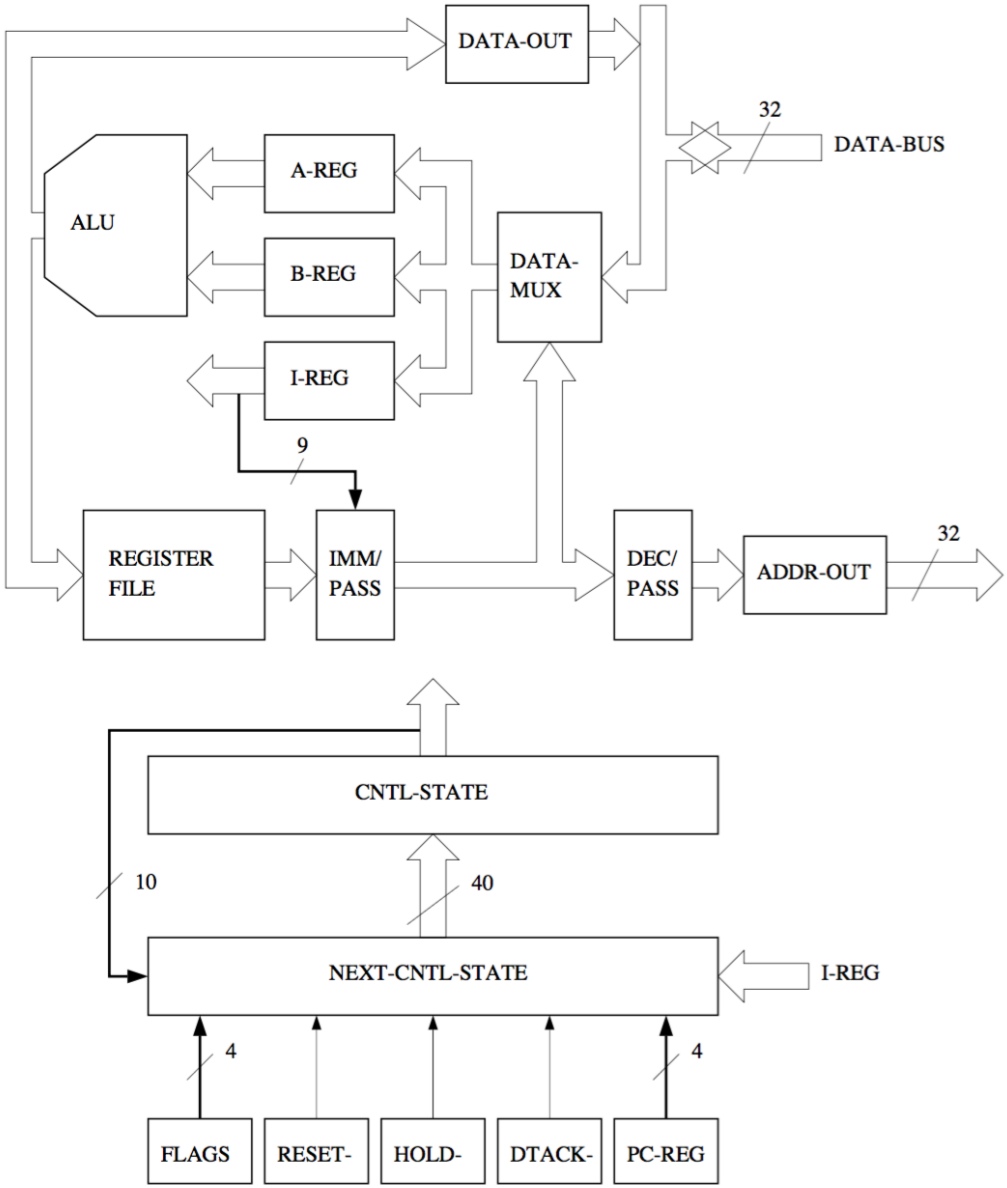
FM9001 Verification

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

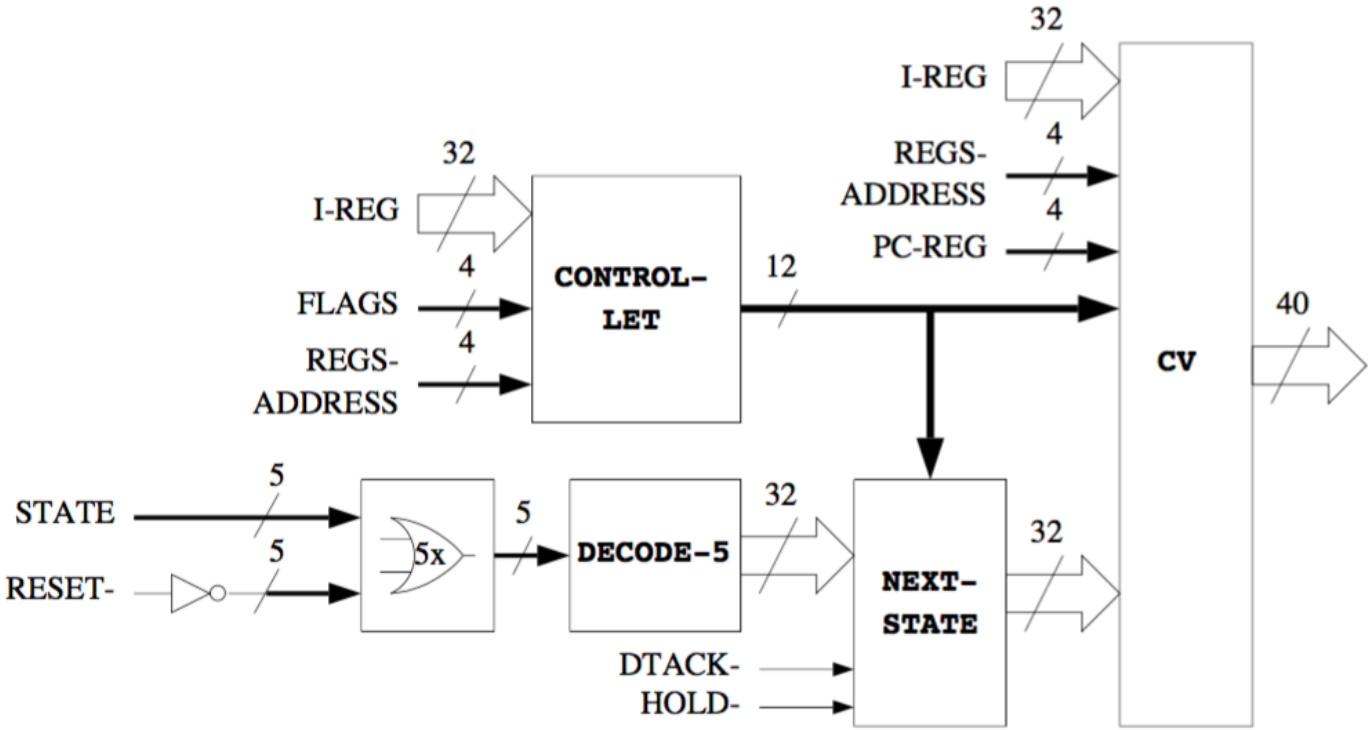
1. The FM9001 can be forced to a known state, i.e., reset, by a suitable sequence of inputs.
2. Given a set of initial conditions, the gate-level model correctly implements the high-level instruction interpreter.
3. The state at the end of the reset sequence satisfies the initial conditions for the previous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

Our result so far: proved that given a set of initial conditions, the gate-level model correctly implements the register-transfer model.

Block Diagram of the FM9001



The NEXT-CNTL-STATE module



Outline

Future Work

Finish the proof of correctness of the FM9001 gate-level design, i.e., the three major lemmas mentioned earlier.

Future Work

Finish the proof of correctness of the FM9001 gate-level design, i.e., the three major lemmas mentioned earlier.

Specify and verify the correctness of the FM9001 using the **asynchronous-circuit-oriented** formalization.

- No global clock signal.
- Local communication protocols, e.g., the link-joint interface [M. Roncken et al., 2015].
- Non-deterministic behavior due to *uncertain but bounded delays* on wires and gates.
- ...

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

The 6th International Workshop on Designing Correct Circuits (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.

Questions?