Evaluatable, High-Assurance Microprocessors

David Greve

Matthew Wilding

{dagreve,mmwildin}@rockwellcollins.com

Rockwell Collins, Inc.

Advanced Technology Center

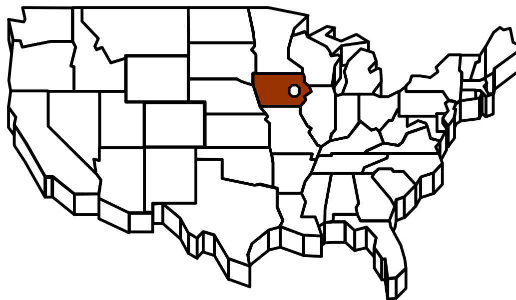
Cedar Rapids, Iowa

NSA HCSS research conference

March 2002

Rockwell Collins

- Advanced Communication and Aviation Equipment
  - Air Transport, Business, Regional, and Military Markets
  - $2.5 Billion in Sales
- Headquartered in Cedar Rapids, IA
  - 17,000 Employees Worldwide



Advanced Technology Center

Air Transport/BRS
Displays
SATCOM
Flight Guidance Systems
Data Management Systems

Military
Joint Strike
JTIDS
KC-135
GPS

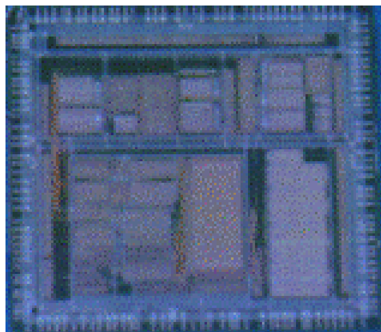
Commercial Systems

Government Systems

Advanced Technology Center

- The **Advanced Technology Center** (**ATC**) identifies, acquires, develops and transitions value-driven technologies to support the continued growth of Rockwell Collins.
- The **Advanced Computing Systems** department addresses emerging technologies for high assurance computing systems with particular emphasis on embedded systems.
- The **Formal Methods Center of Excellence** applies mathematical tools and reasoning to the problem of producing high assurance systems.

CAPS: Collins Adaptive Processing System



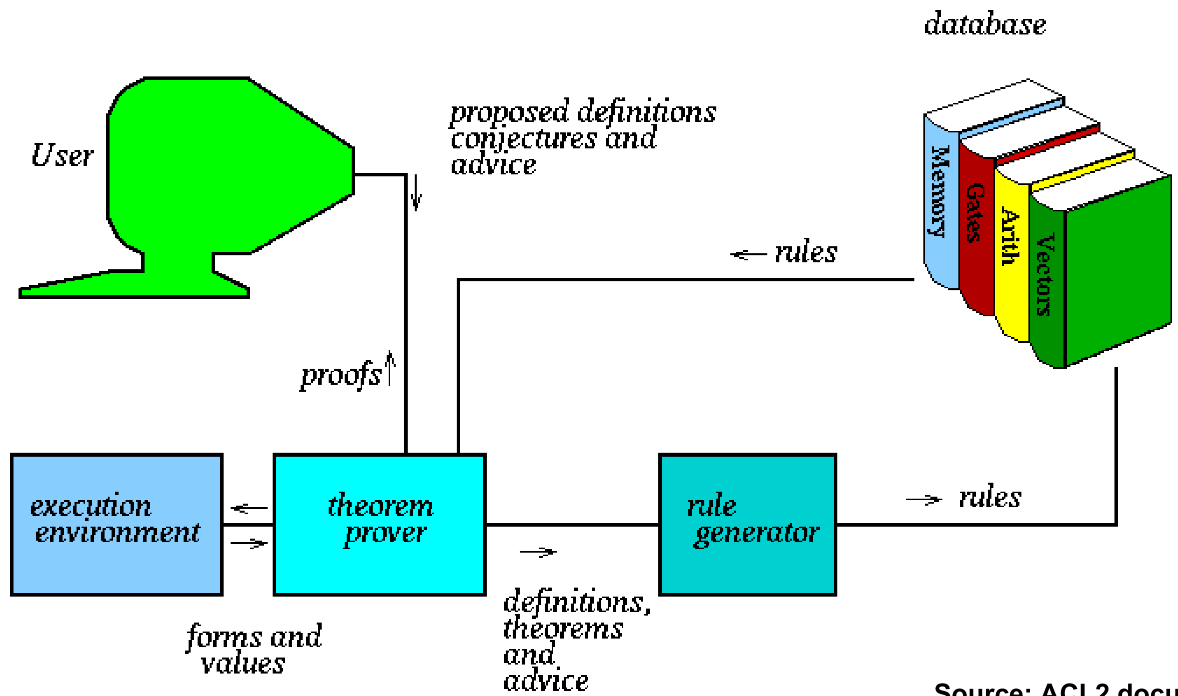
CAPS is a processor family for critical applications

- RC processors have a 30+ yr history
- stack-based, Java bytecode-like
- microcoded, stack and instruction caches
- verified through extensive process of “walkthrus”
- microarchitecture simulator: 5K LOC
- Some CAPS family members used in ultra-critical applications

Background: ACL2

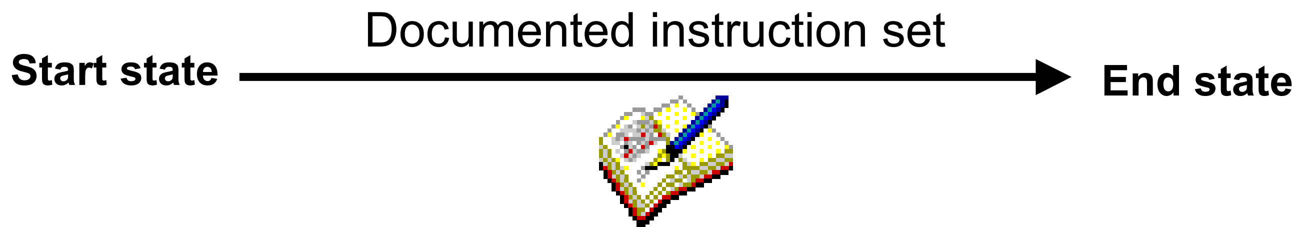
- ACL2 is a system for modeling and mathematical reasoning.
  - One of a number of available “theorem provers”
  - ACL2 homepage at the University of Texas at Austin
- The logic of ACL2 is a subset of Common Lisp
  - basically, the *functional* (or *applicative*) part of standard Common Lisp
- ACL2 documented in 2 books and an extensive webpage
- Some interesting applications outside Rockwell Collins:
  - communication protocol correctness
  - AMD K5 and Athlon floating-point operation implementation
    - numerical analysis
    - pipeline disentanglement
  - CLI “stack” (Nqthm)
  - A verified BDD package
  - proofs in mathematics

Background: ACL2 (cont.)

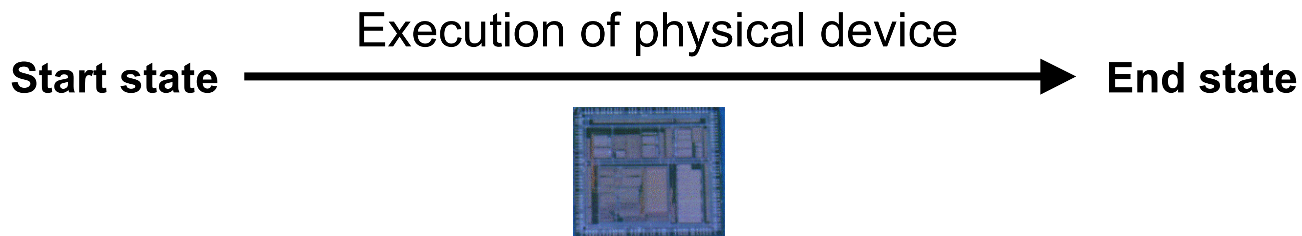


Source: ACL2 documentation

Formal Informal Microprocessor Correctness



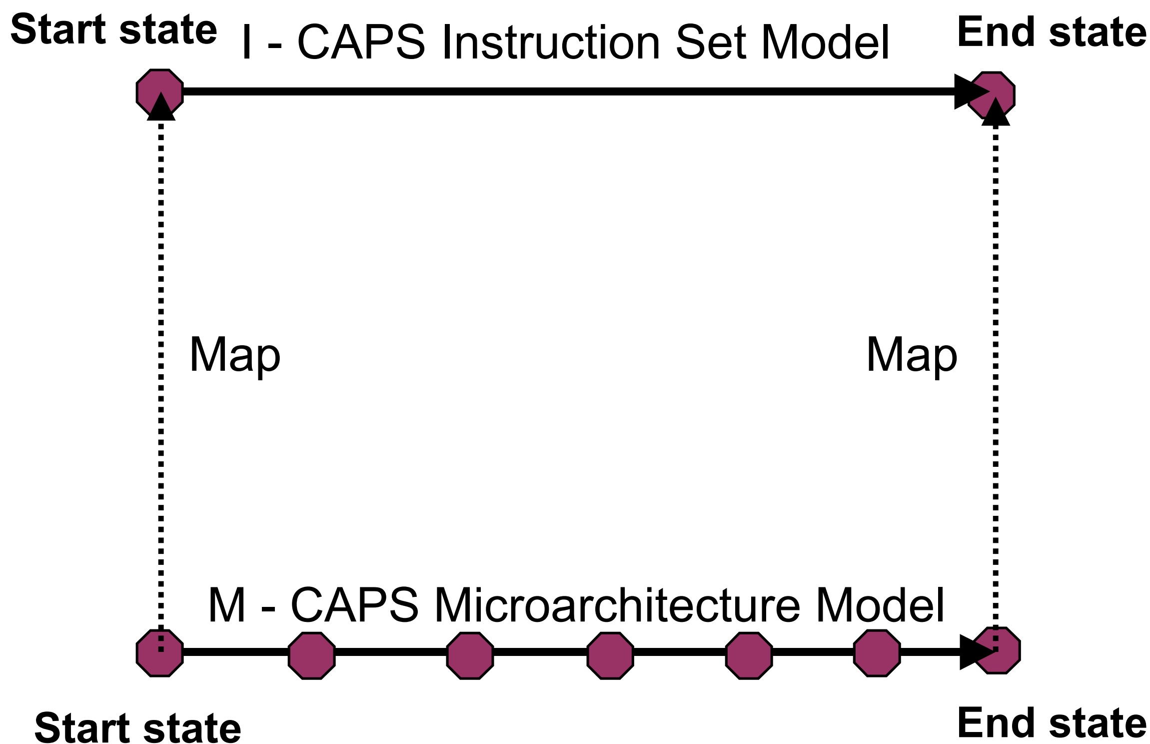
"Works the same as?"



Formal Formal Microprocessor Correctness

Our formalization of the informal notion of correctness poses two evaluation challenges:

- **Does the theorem formalize the informal notion of correctness?**
  - Is I "right"?
  - Is M "right"?
  - Is Map "right"?
- **Is it really a theorem?**



"Formal" proofs in a "formal" process
Executable Formal Models for Validation and Specless Verification" - DASC 2000

Microarchitectural EFMs

- We build executable formal models (EFMs) of our machines.

  Where some write imperative code

  ```
  state.alu.z := 1;
  state.alu.pc := state.alu.pc + 1
  ```

  we write applicative code

  ```
  (let ((st (update-nth (alu.z) 1 st)))
    (let ((st (update-nth (alu.pc) (1+ (alu.pc st)) st)))
  ```

Two papers about our EFM approach

What, Why

Efficient Simulation of Formal Processor Models (FMSD, May 2001)

How

High-Speed, Analyzable Simulators (ACL2 Case Studies, July 2000, KAP)

- Based on preexisting CAPS simulator
  - roughly 5K LOC, 337 elements - integers and arrays - in state
  - microcycle simulator let-expressions 300 x 4 = 1200 deep
  - No performance degradation vs. C - no boxing!

- Imperative-speed execution now built into provers
  - initial RC EFMs used homegrown tools
  - STOBJs, starting with ACL2 2.4
  - PVS has added execution with destructive optimization

Rockwell Collins

The READER

The reader is a Common Lisp/ACL2 macro that expands imperative-looking programs into EFMs.

```
(DEFUN CAPS_BUS_ALU (ST)
  (DECLARE …)
  (CAPS *STATE->STATE*
             (ALU. F_PREV = (ALU. F))
             (CASE (MSQ. FN)
                     ((0)
                      (ALU. F = (OR (ALU. S) (ALU. R)))
                      (ALU. COUT = 0)
                      (ST_. J0 = 1))
                     ((1)
                      (ALU. F = (OR (ALU. S) (~ (ALU. R))))
                      (ALU. COUT = 0)
                      (ST_. J0 = 0))
                     ...
```

We exploit ACL2's support of a real programming language a lot.

The macro expansion has

- single-threaded access to a list containing the state,
- declarations so GCL compiles efficiently, and
- state accessing using "update-nth" and "nth" functions.

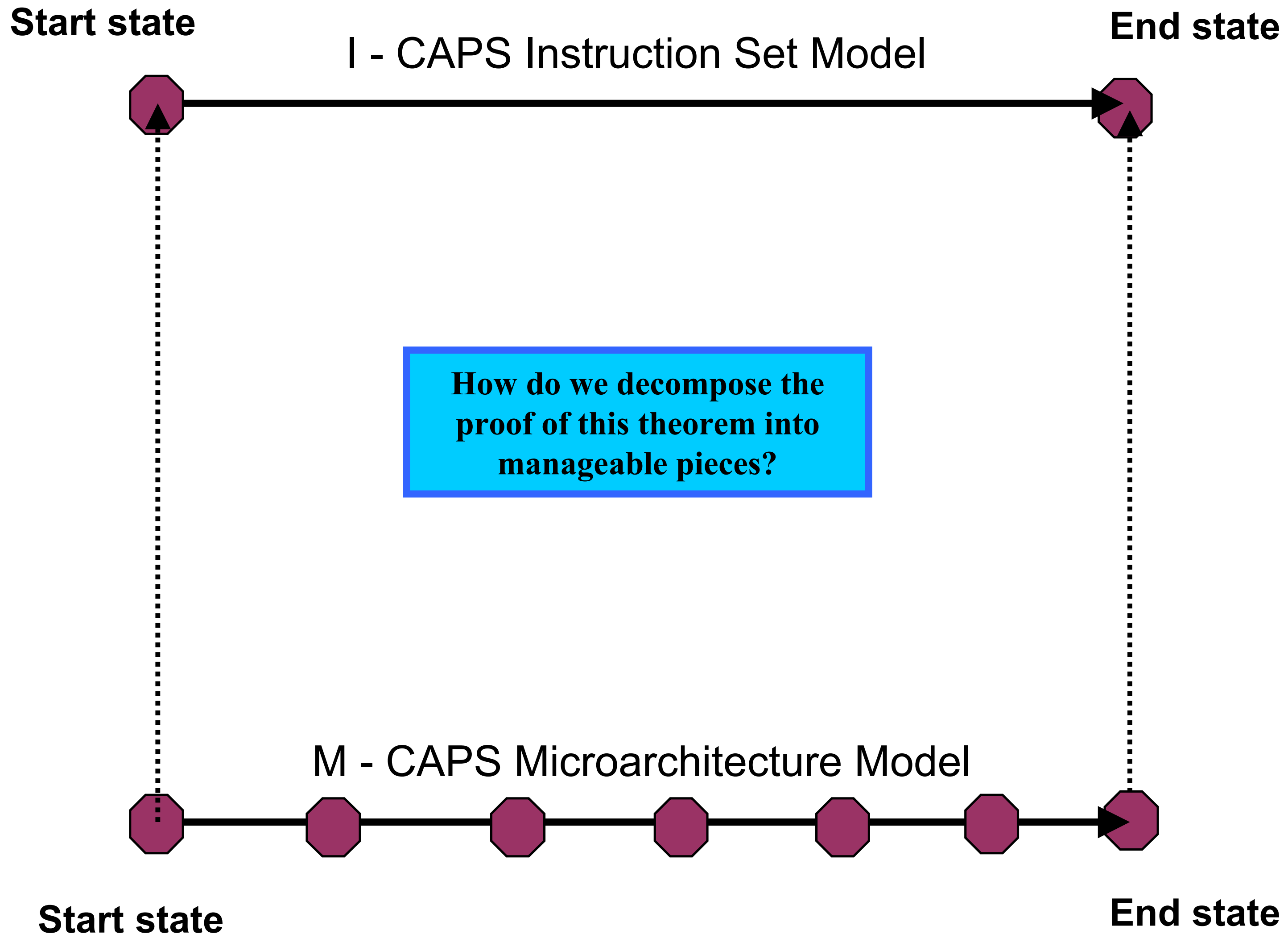
CAPS Microarchitecture Model

The ACL2 CAPS uarch model replaces the C model in the CAPS microcode simulator. The replacement is not observable to users.

CAPS ACL2 uarch model passes 3-hr standard CAPS regression test!

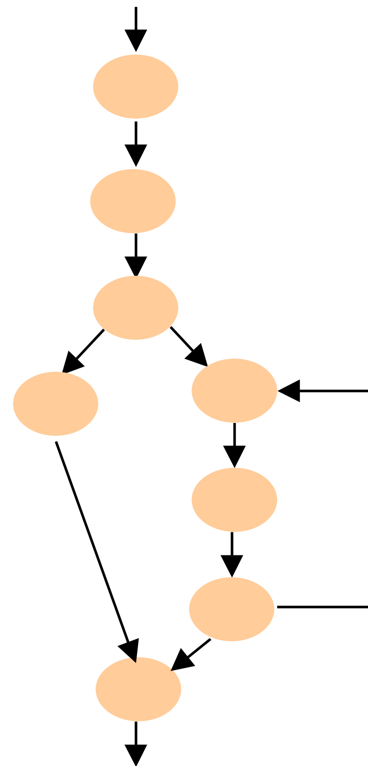
High-speed, formal models provide for evaluatability (looks like C, passes regression tests, integrated into dev process, proofs checked)

CAPS Correctness Theorem



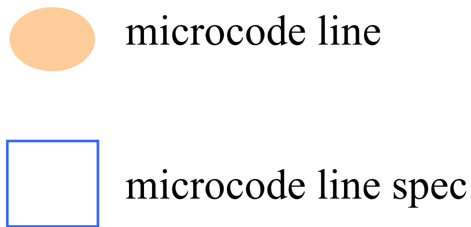
Decomposing the Proof

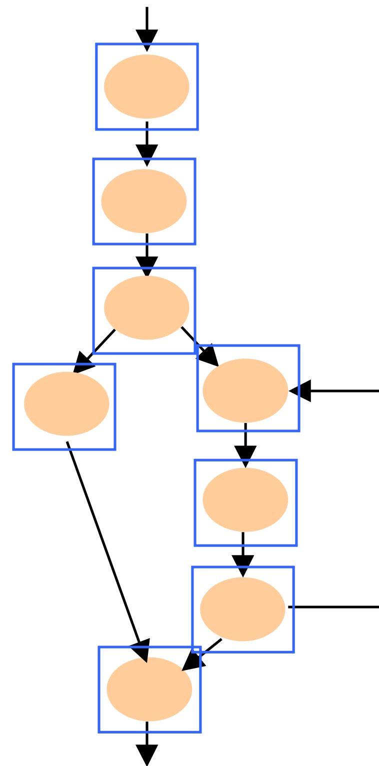
Microcode sequences can be specified and verified in steps.



Decomposing the Proof

Microcode sequences can be specified and verified in steps.

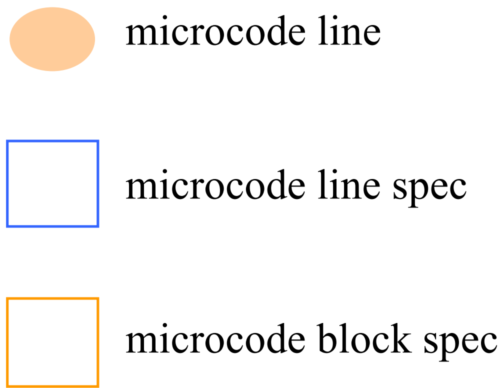


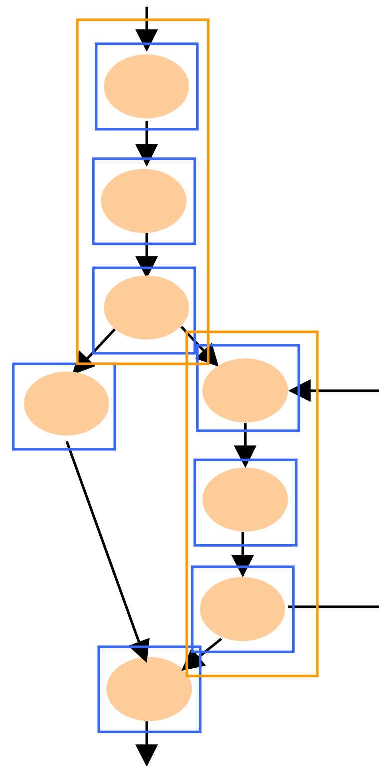


Instruction microcode implementation

Decomposing the Proof

Microcode sequences can be specified and verified in steps.

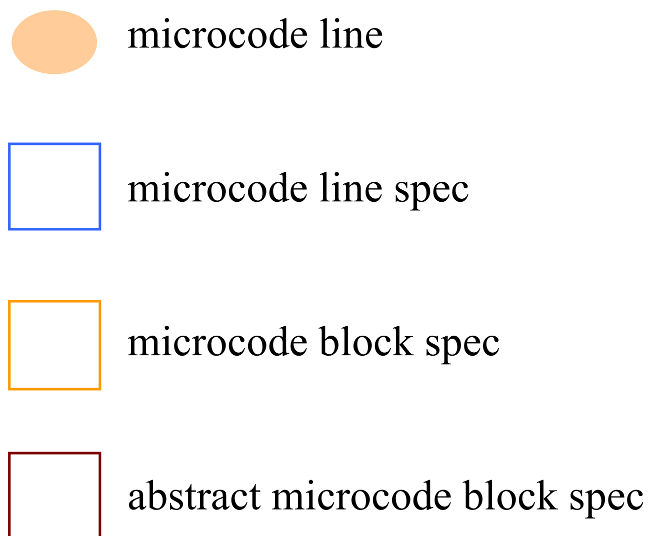


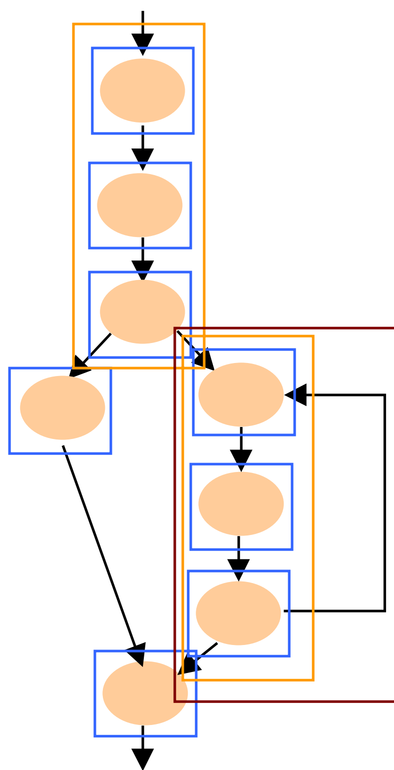


Instruction microcode implementation

Decomposing the Proof

Microcode sequences can be specified and verified in steps.

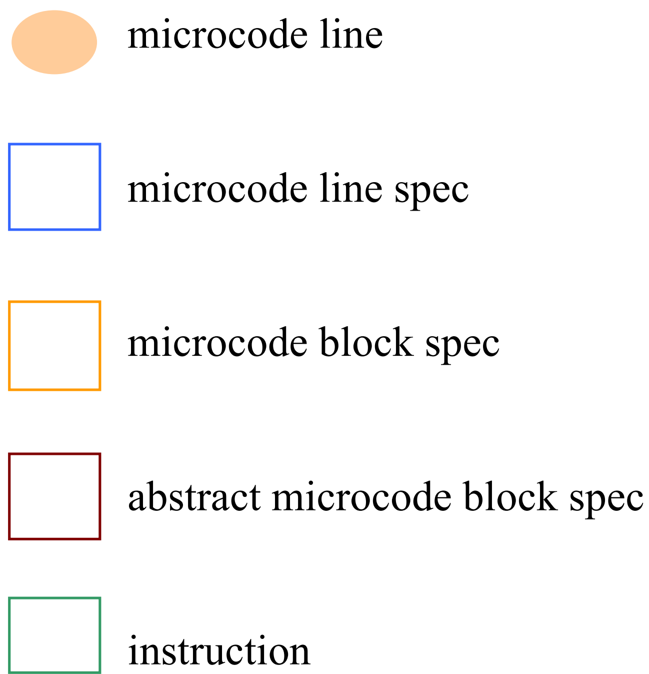


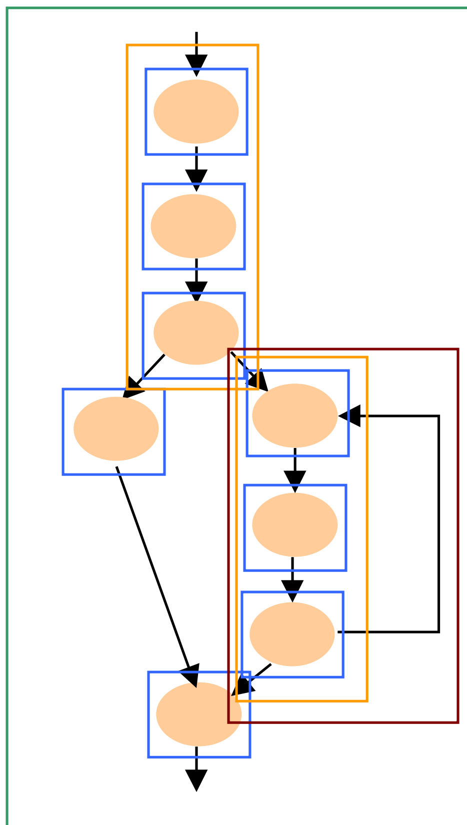


Instruction microcode implementation

Decomposing the Proof

Microcode sequences can be specified and verified in steps.

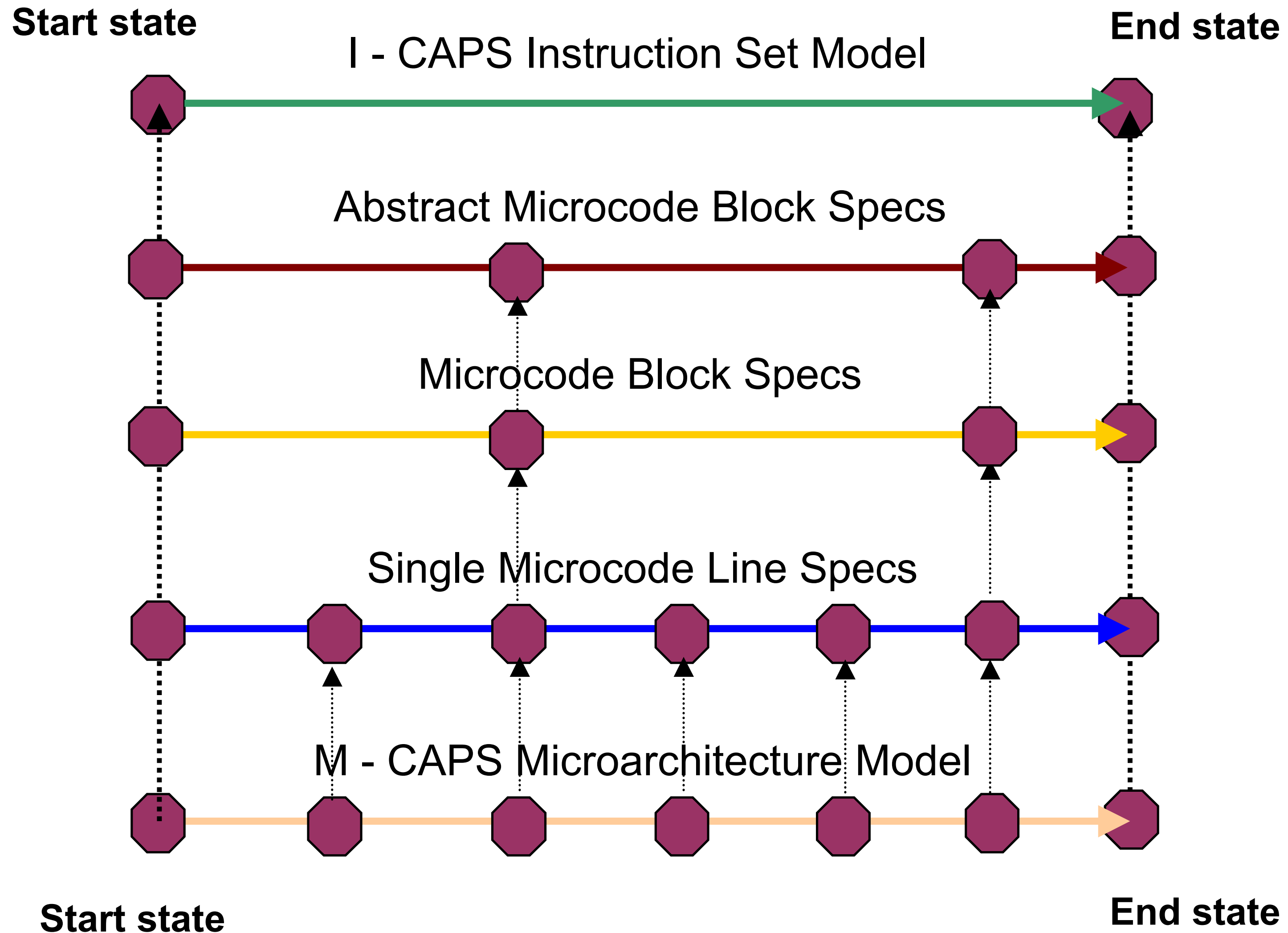




Instruction microcode implementation

Rockwell Collins

Proving the CAPS Correctness Theorem



Rockwell Collins

MAXOF2

We illustrate this proof architecture with a simple CAPS instruction. maxof2 pops 2 stack values and pushes the greater.

Goal theorem:

```
(defthm maxof2-works
  (implies
   (and
    (caps_init_uinstp (uaddr::maxof2) st)
    (goodocc st)
    (stk-adjusted (op::maxof2) st)
    (st-p st)
    (normal-operation st)
    (cache-loaded st))
   (equal (CAPS::map (m st (clock::maxof2 st)) CAPS::st)
     (CAPS::i (op::maxof2) (CAPS::map st CAPS::st)))))
```

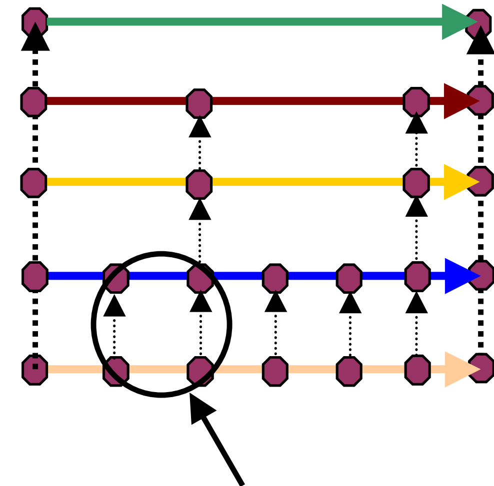
A relevant definition used in I:

```
(defun op-maxof2 (st)
  (declare (xargs :stobjs (st)))
  (CAPS *state->state*
       (POP TS2)
       (POP TS1)
       (TSr = (? (> (logext 16 TS1) (logext 16 TS2))
                 TS1 TS2))
       (PUSH TSr)
       (pc = (& (1+ (pc)) #xffffff))
        st))
```

Three or four 64-bit words of microcode are executed by the CAPS machine for maxof at locations 12F, 41C, 41D, and 41E.

Specifying a Line of Microcode

```
(defun line::x041c (st)
    (let ((st0 st))
    (^
     (V    = (logext 16 (uC::V  )))
     (VM1  = (logext 16 (uC::VM1)))
     (uadr = (uC::IF?=> (! (ext. mode)) (uaddr::ill_inst)))
     (skv  = (> VM1 V))

     (st = (m-step st))
     (st = (base-state st0 st))

     (st = (MACRO (misc-regs :sxv skv)))
     (st = (sequence uadr st))
     (return st))))
```



One line of microcode

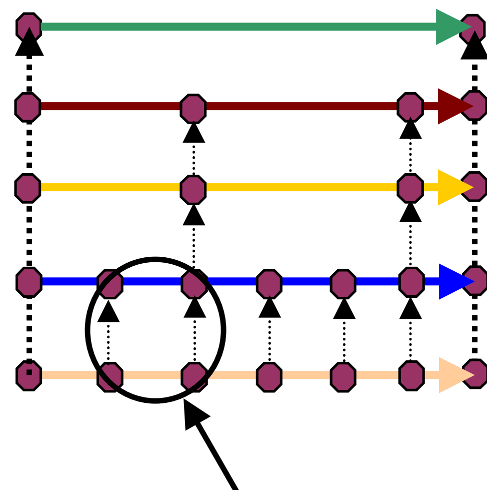
We only specify interesting parts of how a line of microcode changes the machine's state. Other parts are specified to work as defined by m.

Theorems for a Line of Microcode

(line::prove :uaddr #x041c)

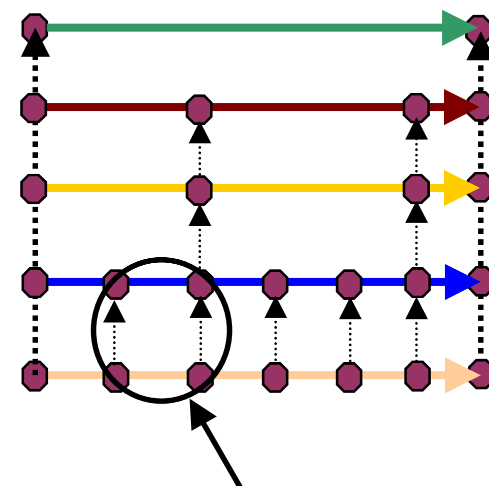
generates many needed lemmas, such as:

```
(DEFTHM LINE::X041C-OPERATION
          (IMPLIES (AND (ST-P ST)
                        (GOODOCC ST)
                        (CAPS_INIT_UINSTP 1052 ST)
                        (NORMAL-OPERATION ST)
                        (MAPPED-MICROCYCLE ST))
                   (AND (ST-P (LINE::X041C ST))
                        (NORMAL-OPERATION (LINE::X041C ST))
                        (GOODOCC (LINE::X041C ST)))))

(DEFTHM LINE::X041C-UINST-1
          (IMPLIES (AND (ST-P ST)
                        (GOODOCC ST)
                        (CAPS_INIT_UINSTP 1052 ST)
                        (NORMAL-OPERATION ST)
                        (MAPPED-MICROCYCLE ST))
                   (AND (CAPS_INIT_UINSTP 1053 (LINE::X041C ST))
                        (MAPPED-MICROCYCLE (LINE::X041C ST)))))

(DEFTHM LINE::X041C-EXECUTION
          (IMPLIES (AND (CAPS_INIT_UINSTP 1052 ST)
                        (ST-P ST)
                        (NORMAL-OPERATION ST)
                        (GOODOCC ST)
                        (MAPPED-MICROCYCLE ST))
                   (EQUAL (M ST (CLOCK::X041C ST))
                          (LINE::X041C ST))))
```



Proof of a Line of Microcode

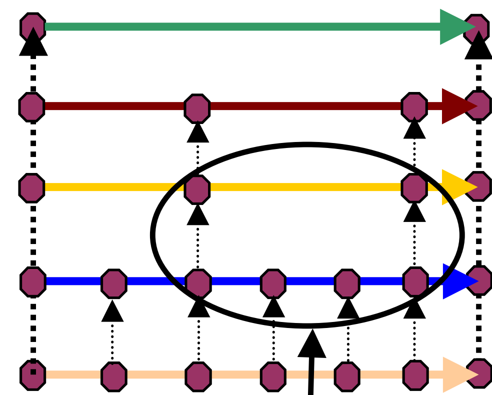
- Even after decomposing the proof, programming ACL2 to prove these kinds of theorems is a big job!
  - Super-IHS
    - We have proved hundreds of rules in our strategy for simplifying microprocessor operation expressions.
      - moving bits around - easy
      - arithmetic - easy
      - arithmetic and bit-vector - hard
  - update-nth equality
  - thousands of rules get automatically generated and proved related to state updates and references
- ACL2 theorem prover itself enhanced to integrate efficient nth-update-nth reasoning into simplifier.



See J Moore's CAV'01 paper for nu-rewriter details

Specifying a Block of Microcode

```
(defblock maxof2
  :uaddr (uaddr::maxof2)
  :ep? t
  :raw? t
  :body (line::x012f
         line::x041c
         (caps.alu.sxv.q
          (1 line::x041d
             :map
           )
          (0 line::x041d
             line::x041e
             :map)))
```



One block of microcode

`Defblock` generates what is needed to specify and verify a block of microcode, like **`line::prove`**.

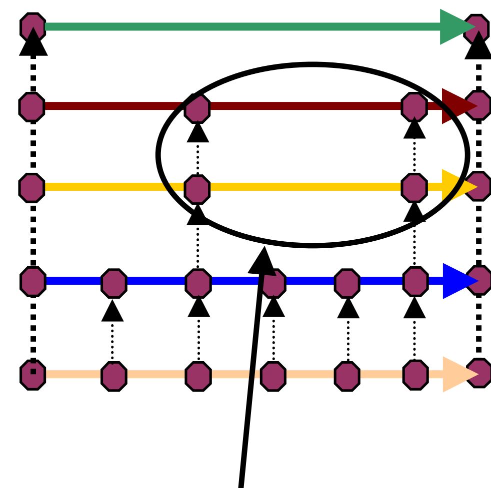
- clock function
- spec function
- correctness theorems

Abstract Microcode Block Specification

Abstract specs can be very helpful.

- Practically speaking, they are required for blocks containing loops in order to eliminate recursion over state.
- Like at the microcode-line level, these specs benefit from identifying interesting elements and specifying irrelevant elements using the lower-level model.

(`MAXof2` is too simple to benefit from a more abstract spec for the execution of its microcode.)



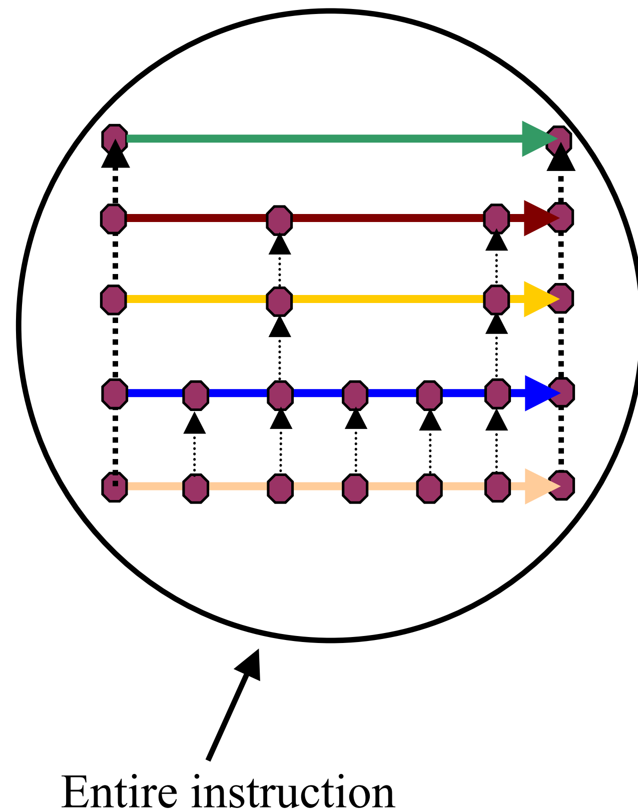
Microcode block abstraction

Rockwell Collins

MAXOF2 works!

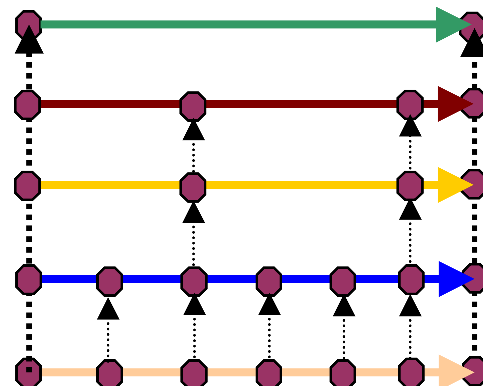
We put the pieces together to prove the main theorem:

```
(defthm maxof2-works
  (implies
   (and
    (caps_init_uinstp (uaddr::maxof2) st)
    (goodocc st)
    (stk-adjusted (op::maxof2) st)
    (st-p st)
    (normal-operation st)
    (cache-loaded st))
   (equal (CAPS::map (m st (clock::maxof2 st)) CAPS::st)
      (CAPS::i (op::maxof2) (CAPS::map st CAPS::st)))))
```



New Stuff

- Executable formal models (EFMs)
- A “reader” that greatly simplifies writing analyzable applicative code that runs with imperative speed.
- Proof decomposition
  - **similar in some ways to other proof decomposition challenges**
  - **definition of level by using lower levels**

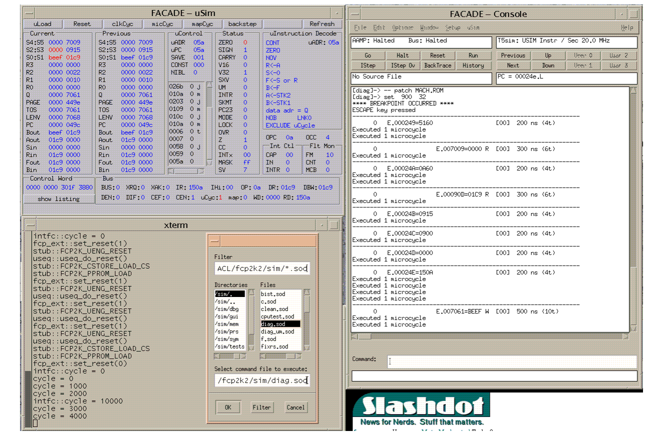


- Nu-rewriter (JSM)
- Proof automation
  - **Books of theorems that constitute a strategy (of course!)**
  - **Theorems generated from state description**
  - **Code that supports the proof decomposition process**
    - **Theorem-generating macros**

Rockwell Collins

Summary

At Rockwell Collins we are...



- writing software that is evaluatable - and fast!
- modeling microprocessor microarchitectures,
- proving correctness using a theorem prover, and
- exploring how to use this in a certification context.