Machines Reasoning about Machines

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Office: TAY 4.140A
Email: moore@cs.utexas.edu

http://www.cs.utexas.edu

Wishful Thinking ...

If we had some exact language ... or at least a kind of truly philosophic writing, in which the ideas were reduced to a kind of alphabet of human thought, then all that follows rationally from what is given could be found by a kind of calculus, just as arithmetical or geometrical problems are solved.

— *Leibniz (1646–1716)*

Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program.

— *John McCarthy, "A Basis for a Mathematical Theory of Computation," 1961*

or ...

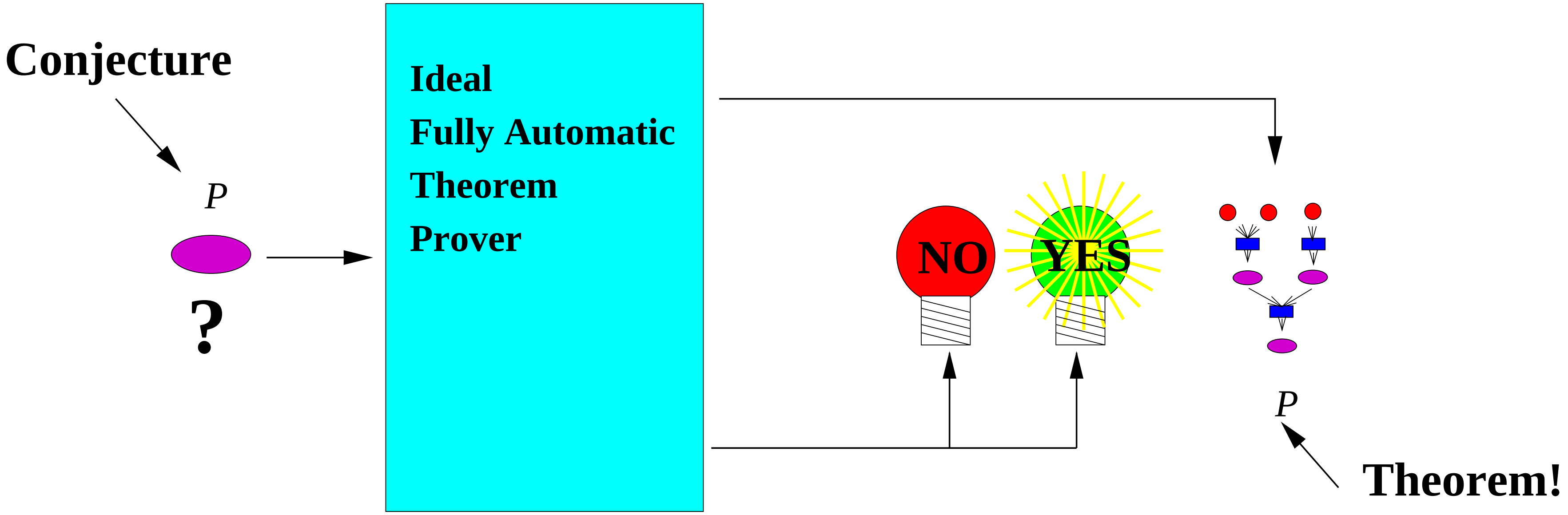
Necessity?

An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results

— *NY Times, "Circuit Flaw Causes Pentium Chip to Miscalculate, Intel Admits," Nov 11, 1994*

Intel Corp. last week took a $475 million write-off to cover costs associated with the divide bug in the Pentium microprocessor's floating-point unit — *EE Times, Jan 23, 1995*

Can Machines Reason?



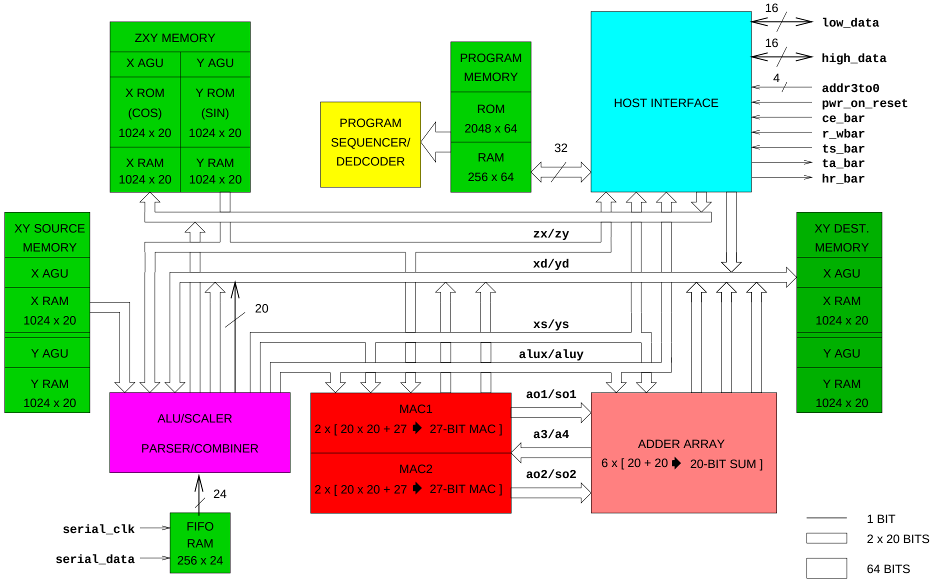
Conjecture P must be written in some *formal language*.

Axioms express the primitive truths of the system.

Rules of Inference describe how to derive new truths from old ones.

Theorems are truths, derived from axioms by rules of inference.

To reason about machines, we must have a language in which to describe the intended operation of machines.



$$
machine(s,n) = \begin{cases} s, & \text{if } n = 0; \\ machine(step(fetch(s), s), n-1), & \text{otherwise.} \end{cases}
$$

We use a functional programming language: Lisp. Instead of writing

$$machine(s, n) = \begin{cases} s, & \text{if } n = 0; \\ machine(step(fetch(s), s), n-1), & \text{otherwise.} \end{cases}$$

we write

```
(defun machine (s n)
  (if (zp n)
      s
      (machine (step (fetch s) s) (- n 1))))
```

A Trivial Theorem

```
(defun machine (s n)
  (if (zp n)
      s
      (machine (step (fetch s) s) (- n 1))))

(defthm sequential-execution
 (implies (and (integerp i)
               (integerp j)
               (<= 0 i)
               (<= 0 j))
          (equal (machine s (+ i j))
                 (machine (machine s i) j))))
```

Automatic Proof. Induction on `i`. **Q.E.D.**

A Less Trivial Example?

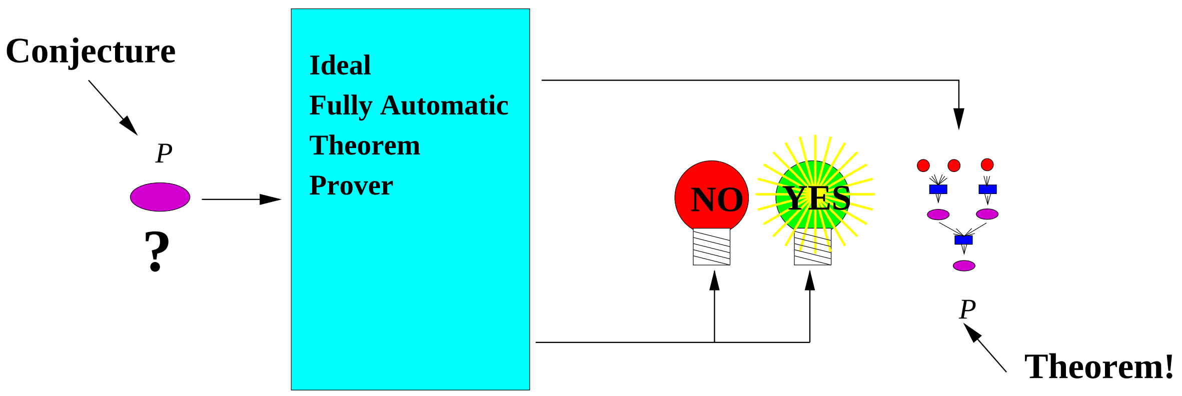
Axiom.
$0 + n = n$

Axiom.
$s(m) + n = s(m + n)$
...
Axiom.
$n^0 = 1$

Axiom.
$n^{s(m)} = n \times n^m$

Conjecture. *P*
$i > 2 \rightarrow x^i + y^i \neq z^i.$

— *posed by Fermat, 1637; proved by Wiles, 1995; not proved by machine!*



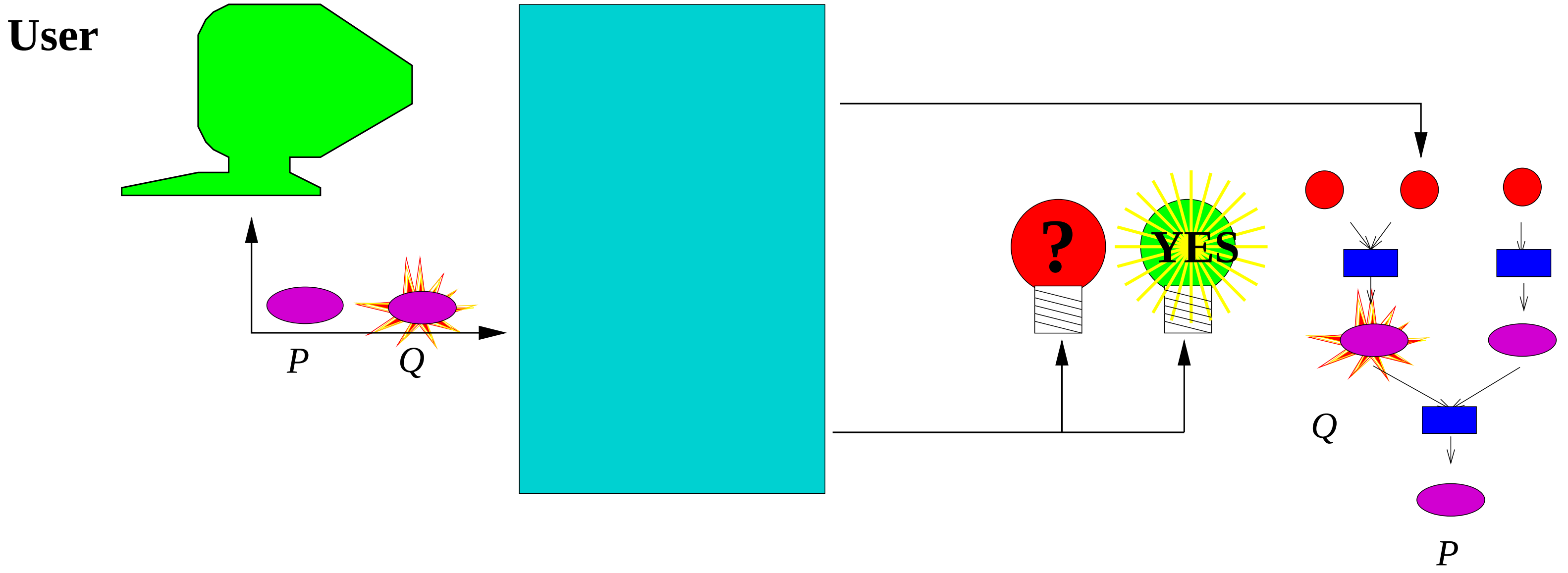
Yeah, right.

Elevate them guns a little lower.

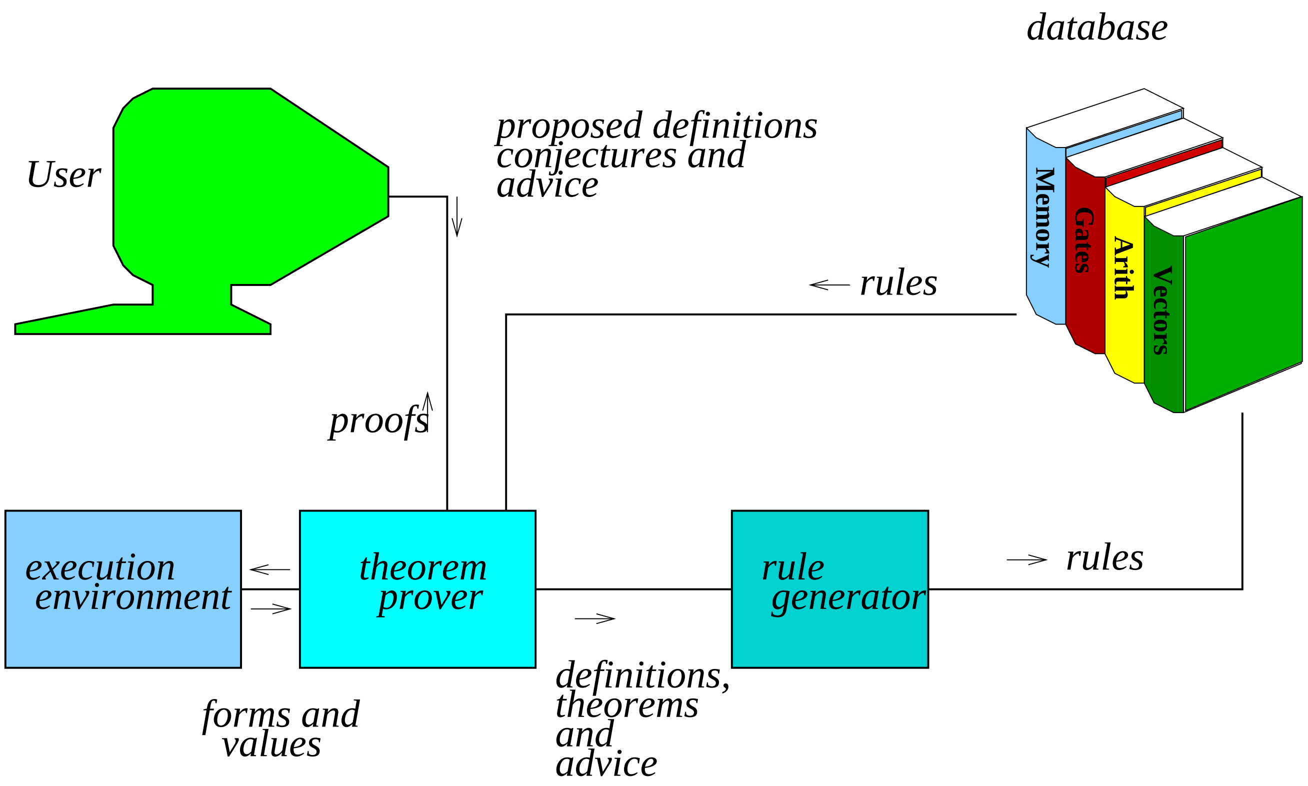
— *Andrew Jackson, at the Battle of New Orleans, Jan, 1815.*

In order for a program to be capable of learning something it must first be capable of being told it.

— *McCarthy, "Programs with Common Sense" (aka "The Advice Taker"), 1959*



System Architecture



Proving Theorems about Lisp Functions

We are concerned with proving theorems in a first-order theory of lists, ...

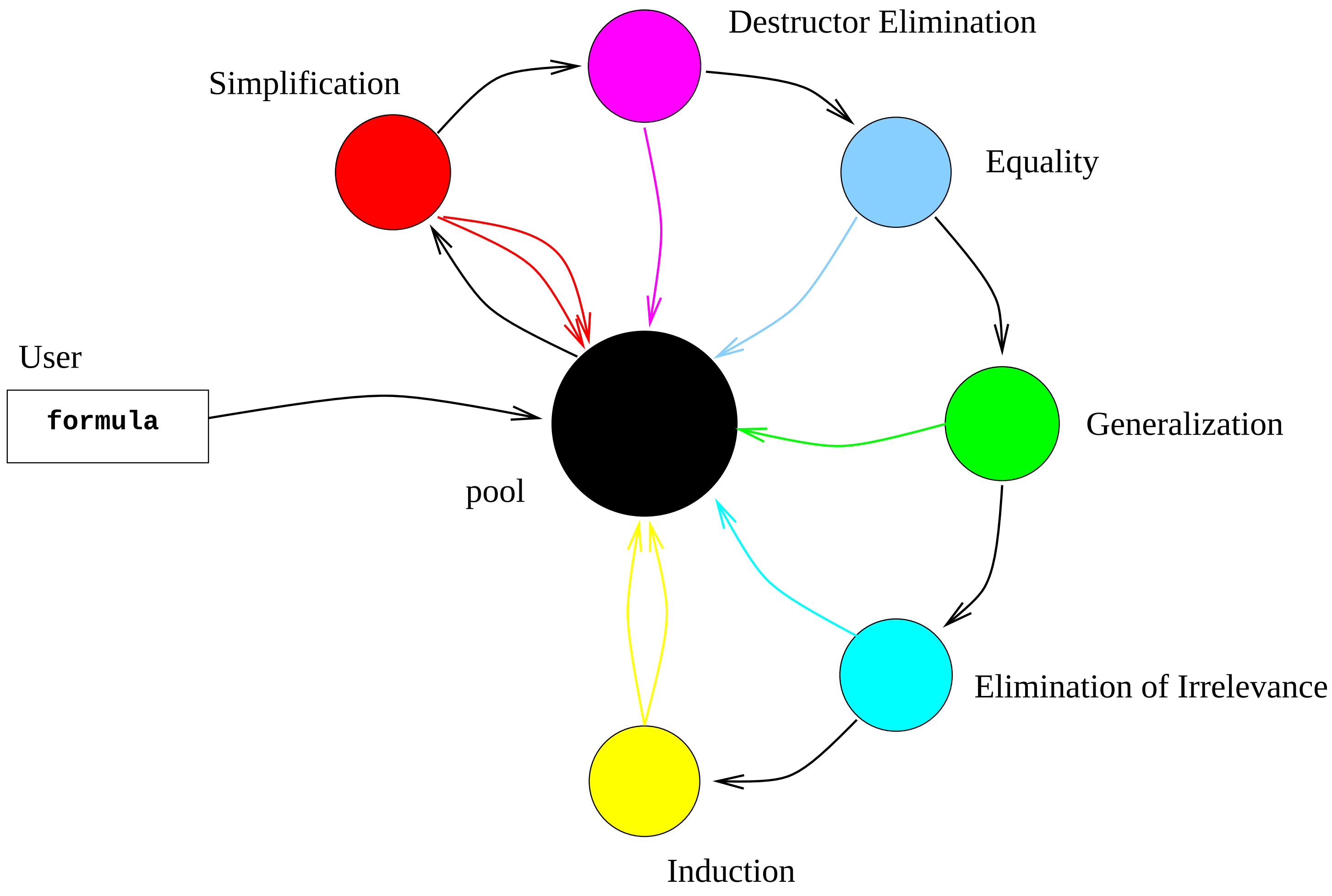
We use a subset of Lisp as our language because recursive list processing functions are easy to write in Lisp and because theorems can be naturally stated in Lisp...

It is intuitively clear that evaluation and induction are complements.

We employ a Lisp interpreter to "run" our theorems and a heuristic which produces induction formulas from information about how the interpreter fails. We combine with the induction heuristic a set of simple rewrite rules of Lisp and a heuristic for generalizing the theorem being proved.

— *Boyer and Moore, "Proving Theorems about Lisp Functions," 1973*

A Theorem Prover



A Simple Lisp Theorem, circa 1973

```
ACL2 !>(defthm associativity-of-app
               (equal (app (app a b) c)
                      (app a (app b c))))

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes
are suggested by this conjecture.  Subsumption reduces that number to two.
However, one of these is flawed and so we are left with one viable candidate.

We will induct according to a scheme suggested by (APP a b).  If
we let (:P a b c) denote *1 above then the induction scheme we'll use is

(AND (IMPLIES (NOT (CONSP a)) (:P a b c))
     (IMPLIES (AND (CONSP a) (:P (CDR a) b c))
              (:P a b c))).

This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT a) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized by
E0-ORDINALP).  When applied to the goal at hand the above induction scheme
produces the following two nontautological subgoals.
```

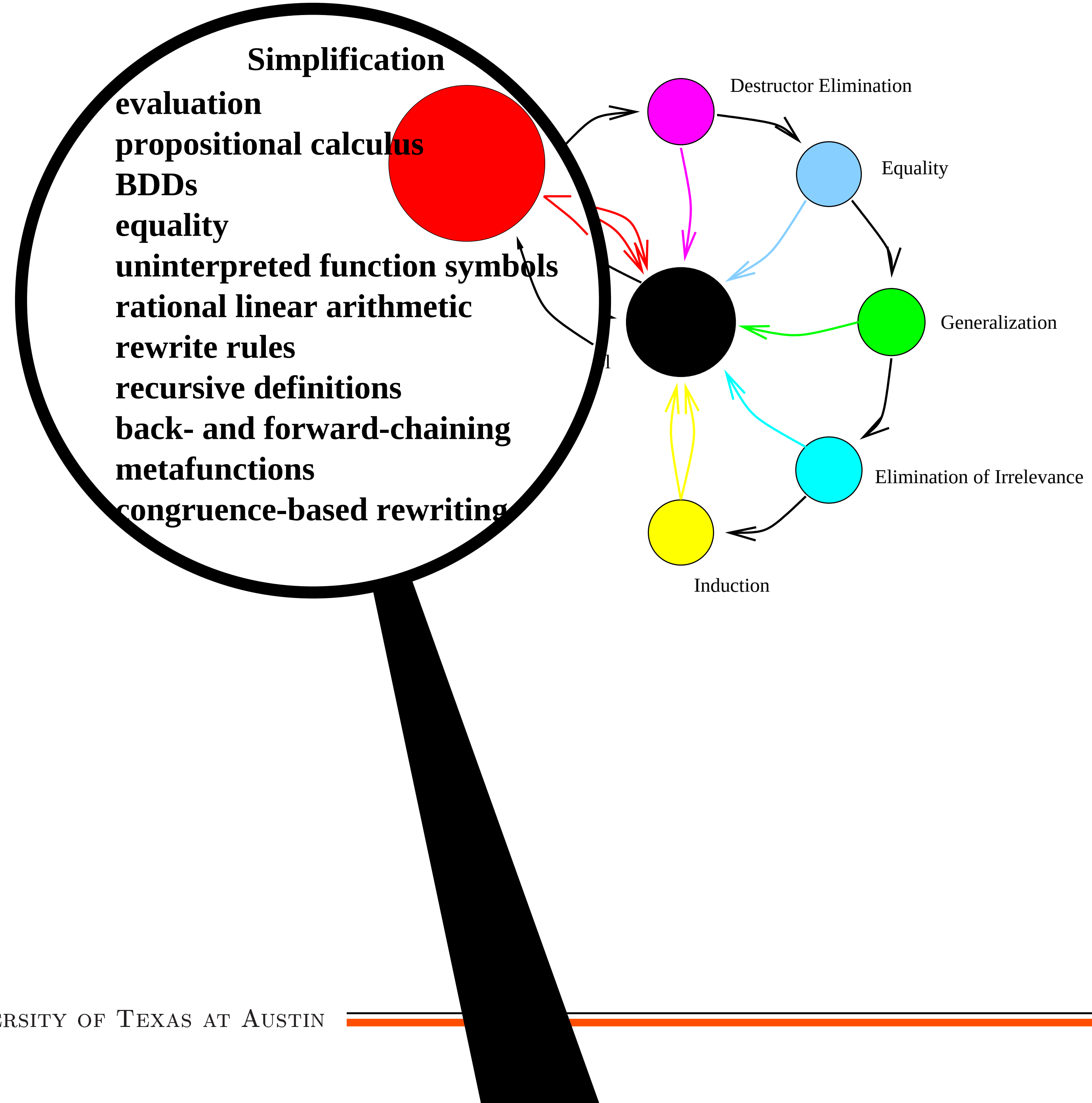
```
Subgoal *1/2
(IMPLIES (NOT (CONSP a))
         (EQUAL (APP (APP a b) c)
                (APP a (APP b c)))).
But simplification reduces this to T, using the :definition APP and
primitive type reasoning.

Subgoal *1/1
(IMPLIES (AND (CONSP a)
              (EQUAL (APP (APP (CDR a) b) c)
                     (APP (CDR a) (APP b c))))
         (EQUAL (APP (APP a b) c)
                (APP a (APP b c)))).
But simplification reduces this to T, using the :definition APP, the
:rewrite rules CDR-CONS and CAR-CONS and primitive type reasoning.

That completes the proof of *1.
Q.E.D.

Summary
Form:  ( DEFTHM ASSOCIATIVITY-OF-APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CDR-CONS)
        (:REWRITE CAR-CONS)
        (:DEFINITION APP))
Warnings:  None
Time:  0.05 seconds (prove: 0.00, print: 0.03, other: 0.02)
 ASSOCIATIVITY-OF-APP
```

The ACL2 Project

ACL2 stands for "A Computational Logic for Applicative Common Lisp." It is

- a functional programming language based on Common Lisp,
- a first-order mathematical logic, and
- a mechanical theorem prover.

ACL2 was written by Matt Kaufmann and J Strother Moore and is the successor to the Boyer-Moore theorem prover. It has been under continuous development since 1989.


Simplification
evaluation
propositional calculus
BDDs
equality
uninterpreted function symbols
rational linear arithmetic
rewrite rules
recursive definitions
back- and forward-chaining
metafunctions
congruence-based rewriting
Destructor Elimination
Equality
Generalization
Elimination of Irrelevance
Induction

Floating Point Division Microcode for AMD K5 (1995)

Algorithm divide$(p, d, mode)$

1.	sd_0	$= \text{lookup}(d)$	`[exact 17 8]`
2.	d_r	$= d$	`[away 17 32]`
3.	sdd_0	$= sd_0 \times d_r$	`[away 17 32]`
4.	sd_1	$= sd_0 \times \text{comp}(sdd_0, 32)$	`[trunc 17 32]`
5.	sdd_1	$= sd_1 \times d_r$	`[away 17 32]`
6.	sd_2	$= sd_1 \times \text{comp}(sdd_1, 32)$	`[trunc 17 32]`
7.	dh	$= d$	`[trunc 17 32]`
8.	dl	$= d - dh$	`[exact 17 32]`
9.	p_0	$= p$	`[exact 17 64]`
10.	ph_0	$= p_0$	`[trunc 17 32]`
11.	q_0	$= sd_2 \times ph_0$	`[away 17 24]`
12.	qdh_0	$= q_0 \times dh$	`[exact 17 64]`
13.	qdl_0	$= q_0 \times dl$	`[exact 17 64]`
14.	pt_1	$= p_0 - qdh_0$	`[exact 17 64]`
15.	p_1	$= pt_1 - qdl_0$	`[exact 17 64]`
...	...	$= \ldots$	...
29.	q_3	$= sd_2 \times ph_3$	`[trunc 17 24]`
30.	qq_2	$= q_2 + q_3$	`[sticky 17 64]`
31.	qq_1	$= qq_2 + q_1$	`[sticky 17 64]`
32.	$divide$	$= qq_1 + q_0$	$mode$

Using the Reciprocal

$$
\begin{array}{rr}
 & 36. \\
+ & -.17 \\
+ & .0034 \\
+ & -.000066 \\
\hline
 & 35.833334 \\
\hline
12 \,\big) & 430.0000000 \\
 & 432. \\
\hline
 & -2. \\
 & -2.04 \\
\hline
 & .04 \\
 & .0408 \\
\hline
 & -\ .0008 \\
 & -\ .000792 \\
\hline
 & -\ .000008
\end{array}
$$

Reciprocal Calculation:

$$1/12 \;=\; 0.083\overline{3} \approx 0.083 = sd_2$$

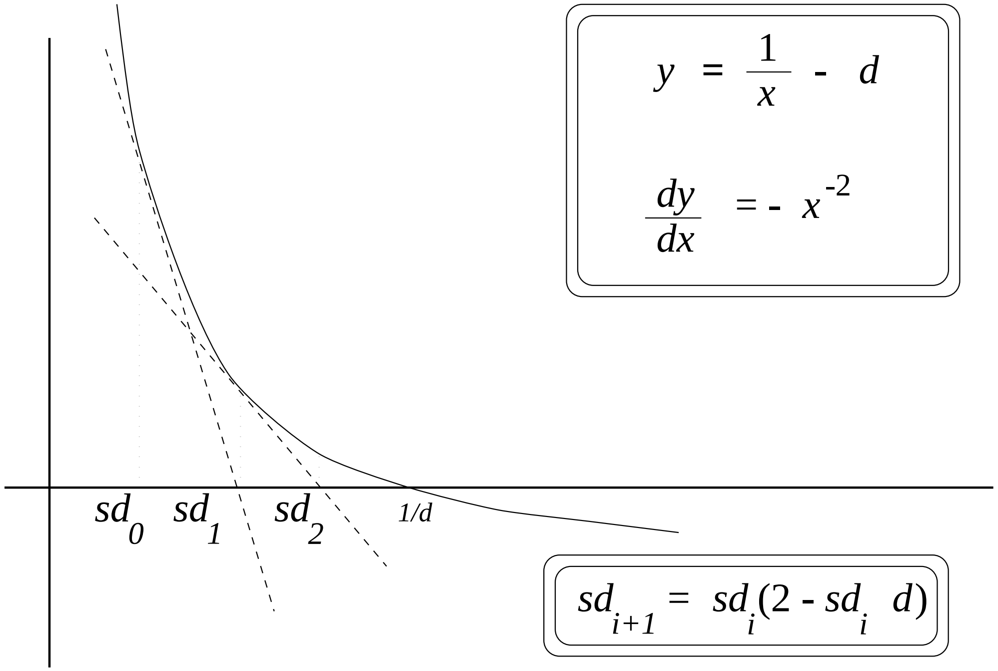
Quotient Digit Calculation:

$$0.083 \times 430.0000 = 35.6900000 \approx 36.000000 = q_0$$
$$0.083 \times -2.0000 = -.1660000 \approx -.170000 = q_1$$
$$0.083 \times .0400 = .0033200 \approx .003400 = q_2$$
$$0.083 \times -.0008 = -.0000664 \approx -.000067 = q_3$$

Summation of Quotient Digits:

$$q_0 + q_1 + q_2 + q_3 = 35.833333$$

Computing the Reciprocal



$$y = \frac{1}{x} - d$$

$$\frac{dy}{dx} = -\,x^{-2}$$

$$sd_{i+1} = sd_i(2 - sd_i\ d)$$

The Lookup Table

top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse
1.0000000_2	0.11111111_2	1.0100000_2	0.11001100_2	1.1000000_2	0.10101010_2	1.1100000_2	0.10010010_2
1.0000001_2	0.11111101_2	1.0100001_2	0.11001011_2	1.1000001_2	0.10101001_2	1.1100001_2	0.10010001_2
1.0000010_2	0.11111011_2	1.0100010_2	0.11001010_2	1.1000010_2	0.10101000_2	1.1100010_2	0.10010001_2
1.0000011_2	0.11111001_2	1.0100011_2	0.11001000_2	1.1000011_2	0.10101000_2	1.1100011_2	0.10010000_2
1.0000100_2	0.11110111_2	1.0100100_2	0.11000111_2	1.1000100_2	0.10100111_2	1.1100100_2	0.10001111_2
1.0000101_2	0.11110101_2	1.0100101_2	0.11000110_2	1.1000101_2	0.10100110_2	1.1100101_2	0.10001111_2
1.0000110_2	0.11110100_2	1.0100110_2	0.11000101_2	1.1000110_2	0.10100101_2	1.1100110_2	0.10001110_2
1.0000111_2	0.11110010_2	1.0100111_2	0.11000100_2	1.1000111_2	0.10100100_2	1.1100111_2	0.10001110_2
1.0001000_2	0.11110000_2	1.0101000_2	0.11000010_2	1.1001000_2	0.10100011_2	1.1101000_2	0.10001101_2
1.0001001_2	0.11101110_2	1.0101001_2	0.11000001_2	1.1001001_2	0.10100011_2	1.1101001_2	0.10001100_2
1.0001010_2	0.11101101_2	1.0101010_2	0.11000000_2	1.1001010_2	0.10100010_2	1.1101010_2	0.10001100_2
…	…	…	…	…	…	…	…
1.0010110_2	0.11011010_2	1.0110110_2	0.10110100_2	1.1010110_2	0.10011001_2	1.1110110_2	0.10000101_2
1.0010111_2	0.11011000_2	1.0110111_2	0.10110011_2	1.1010111_2	0.10011000_2	1.1110111_2	0.10000100_2
1.0011000_2	0.11010111_2	1.0111000_2	0.10110010_2	1.1011000_2	0.10010111_2	1.1111000_2	0.10000100_2
1.0011001_2	0.11010101_2	1.0111001_2	0.10110001_2	1.1011001_2	0.10010111_2	1.1111001_2	0.10000011_2
1.0011010_2	0.11010100_2	1.0111010_2	0.10110000_2	1.1011010_2	0.10010110_2	1.1111010_2	0.10000011_2
1.0011011_2	0.11010011_2	1.0111011_2	0.10101111_2	1.1011011_2	0.10010101_2	1.1111011_2	0.10000010_2
1.0011100_2	0.11010001_2	1.0111100_2	0.10101110_2	1.1011100_2	0.10010101_2	1.1111100_2	0.10000010_2
1.0011101_2	0.11010000_2	1.0111101_2	0.10101101_2	1.1011101_2	0.10010100_2	1.1111101_2	0.10000001_2
1.0011110_2	0.11001111_2	1.0111110_2	0.10101100_2	1.1011110_2	0.10010011_2	1.1111110_2	0.10000001_2
1.0011111_2	0.11001101_2	1.0111111_2	0.10101011_2	1.1011111_2	0.10010011_2	1.1111111_2	0.10000000_2

The Formal Model of the Code

```
(defun divide (p d mode)
 (let* ((sd0    (eround (lookup d)                  '(exact 17 8)))
        (dr     (eround d                           '(away 17 32)))
        (sdd0   (eround (* sd0 dr)                  '(away 17 32)))
        (sd1    (eround (* sd0 (comp sdd0 32))      '(trunc 17 32)))
        (sdd1   (eround (* sd1 dr)                  '(away 17 32)))
        (sd2    (eround (* sd1 (comp sdd1 32))      '(trunc 17 32)))
        ...
        (qq2    (eround (+ q2 q3)                   '(sticky 17 64)))
        (qq1    (eround (+ qq2 q1)                  '(sticky 17 64)))
        (divide (round   (+ qq1 q0)                 mode)))
       (or (first-error sd0 dr sdd0 sd1 sdd1 sd2 dh dl p0 ph0 ... divide)
           divide)))
```

The K5 FDIV Theorem

```
(defthm divide-divides
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
           (equal (divide p d mode)
                  (round  (/ p d) mode))))
```

Less formally, if p and d are floating point numbers, $d \neq 0$, and $mode$ is a rounding mode, then the algorithm produces the result of rounding p/d to a floating point number as specified by $mode$.

This work was done by Moore, Lynch and Kaufmann, in 1995, *before the K5 was fabricated*.

AMD Athlon

```
module FMUL;                    // sanitized from AMD Athlon(TM)
                                // by David Russinoff and Art Flatau

//**************************************************************
// Declarations
//**************************************************************

//Precision and rounding control:

`define SNG   1'b0          // single precision
`define DBL   1'b1          // double precision
`define NRE   2'b00         // round to nearest
`define NEG   2'b01         // round to minus infinity
`define POS   2'b10         // round to plus infinity
`define CHP   2'b11         // truncate
```

```
//Parameters:

input x[79:0];              //first operand
input y[79:0];              //second operand
input rc[1:0];              //rounding control
input pc;                   //precision control
output z[79:0];             //rounded product

//**************************************************************
// First Cycle
//**************************************************************

//Operand fields:

sgnx = x[79]; sgny = y[79];                           //signs
expx[14:0] = x[78:64]; expy[14:0] = y[78:64];   //exponents
sigx[63:0] <= x[63:0]; sigy[63:0] <= y[63:0];   //significands
...
```

The "Athlon" FMUL Theorem

The RTL for FMUL (above) was mechanically translated into the definition of a function `fmul` and the following theorem was proved by David Russinoff and Art Flatau of Advanced Micro Devices, Inc.

```
(defthm correctness-of-fmul
    (let ((ideal (rnd (* (hat x) (hat y))
                      (mode rc)
                      (precision pc)))
          (z (fmul x y rc pc)))
      (implies (and (normal-encoding-p x (extfmt))
                    (normal-encoding-p y (extfmt))
                    (member rc (list 0 1 2 3))
                    (member pc (list 0 1))
                    (repp ideal (extfmt)))
               (and (normal-encoding-p z (extfmt))
                    (= (hat z) ideal))))
  :hints ...)
```

Reasoning about the Athlon

All elementary floating-point operations on the AMD Athlon were specified in ACL2 and the proofs were checked mechanically.

All elementary floating-point operations on the AMD Athlon were specified in ACL2 and the proofs were checked mechanically.

All elementary floating-point operations on the AMD Athlon were specified in ACL2 and the proofs were checked mechanically.

The ACL2 proofs uncovered bugs that had remained hidden through hundreds of millions of test cases in RTL simulators.

The bugs were fixed and the new RTL verified *before* the Athlon was fabricated.

Accuracy of the Models

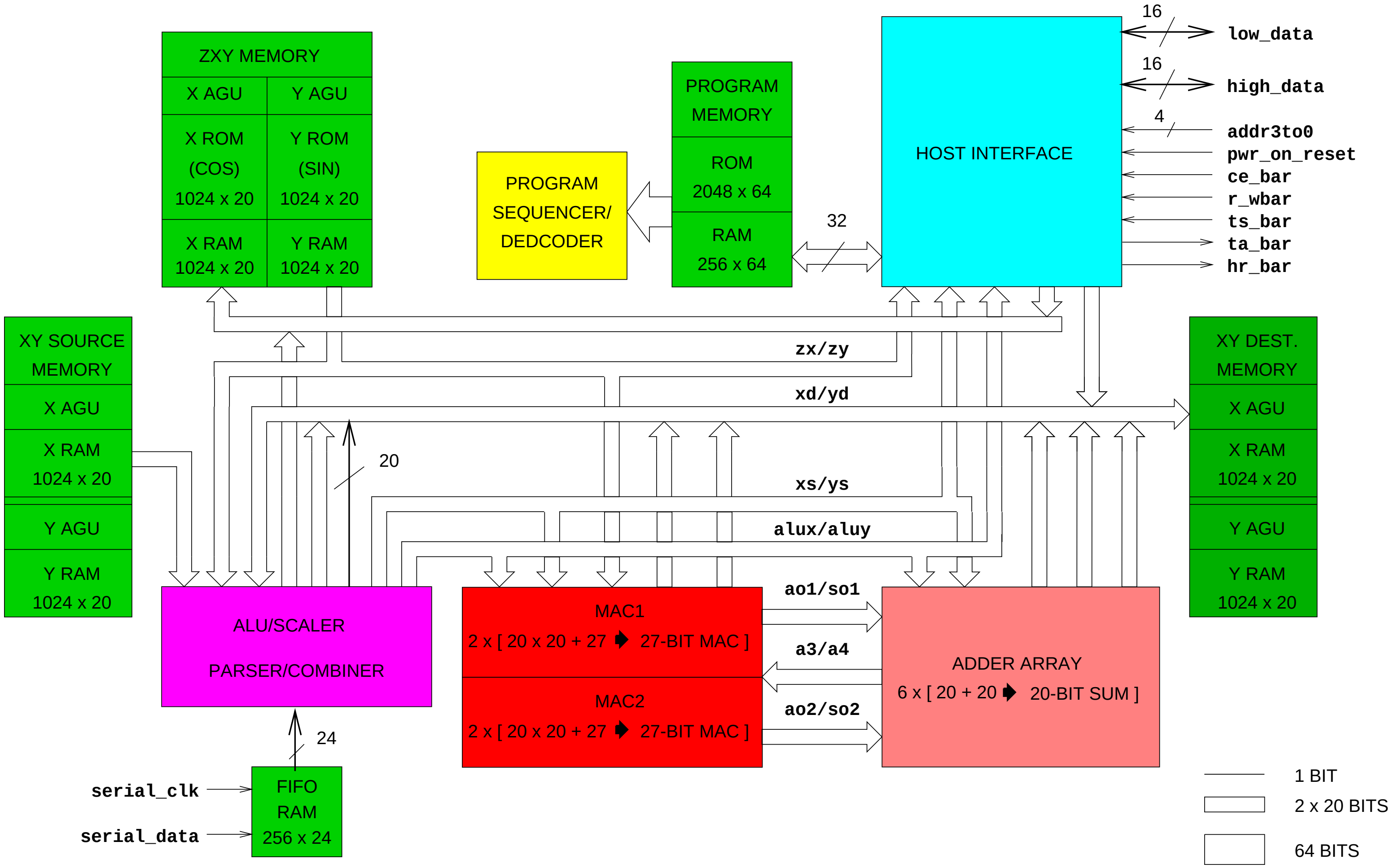
How do we know the model is accurate? That is, how do we know that the translation from RTL to ACL2 was accurate?

We cannot prove that the ACL2 `fmul` and the RTL `fmul` are "the same" because the "semantics" of AMD's RTL is not formal: it is implied by the behavior of the RTL simulator.

The translation has been tested extensively.

- 80 million floating point test vectors were run through the RTL simulator (on the FSQRT model) and through the ACL2 translation.
- All 80 million tests produced exactly the same bit vectors.

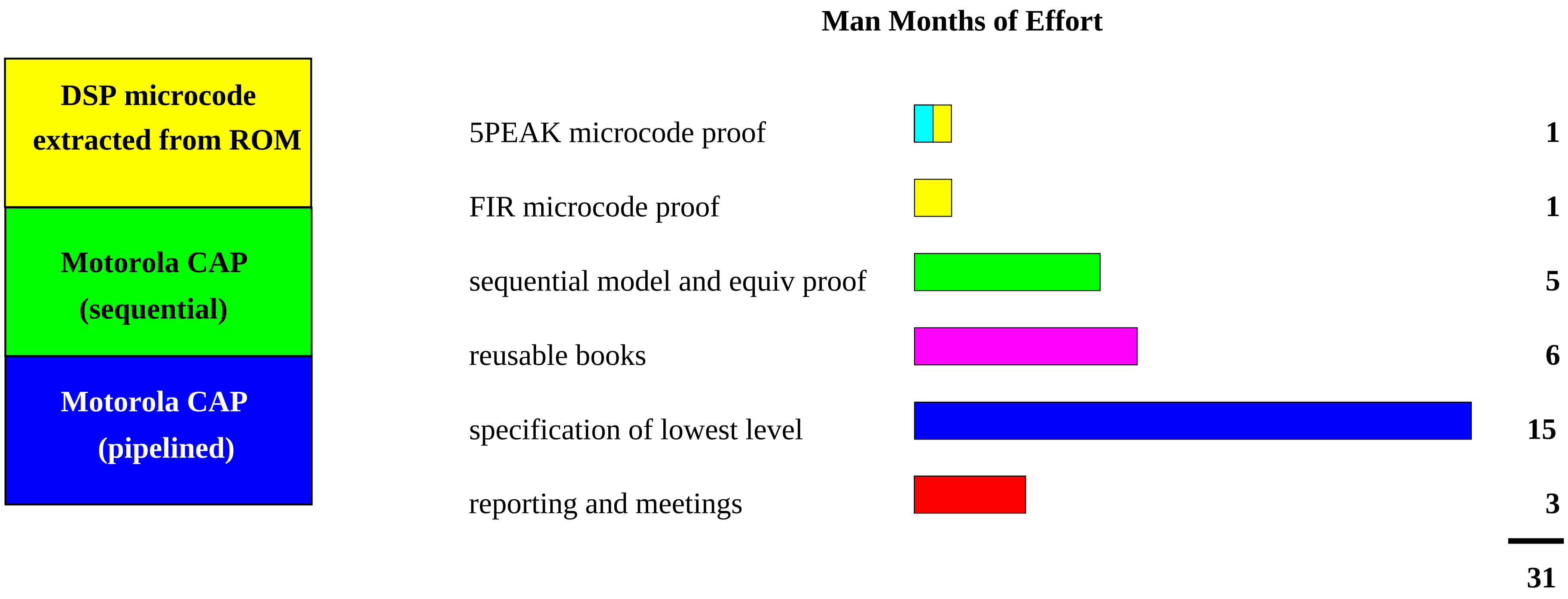
Motorola CAP Digital Signal Processor



Results (Brock)

- a formal model (including the pipeline)
- a "sequential model" (with the pipeline abstracted away)
- a bit-accurate and cycle-accurate simulator several times faster than Motorola's SPW model
- definition of a "hazard detection" predicate
- proof that "no hazard $\rightarrow$ pipelined $=$ non-pipelined"
- proof that over 50 microcode programs extracted from ROM have no hazards
- proof that 2 microcode programs are correct

Task Breakdown

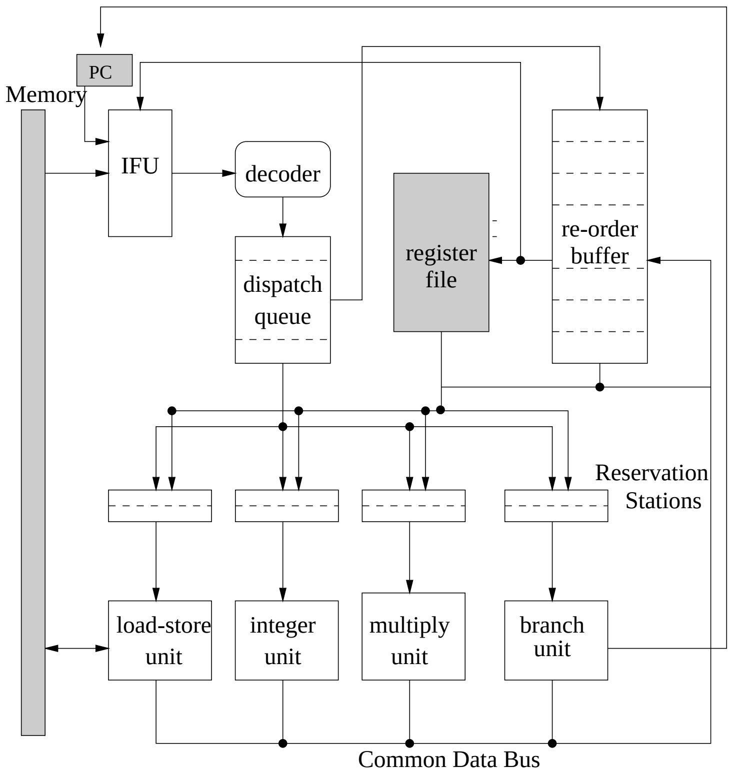


The modeling and theorem-proving work were carried out concurrently with the processor design.

Multiple Issue Out-of-Order Execution Model

Jun Sawada (a recent UT graduate now at IBM) verified the correspondence between a conventional ISA model and a pipelined microprocessor with

- multiple,
- out-of-order instruction issue, with
- a reorder buffer,
- speculative execution and
- exceptions.



An IBM Secure Co-Processor

Vernon Austel and Sean Smith at IBM Research verified a security model for

- IBM 4758 secure coprocessor (http://www.ibm.com/security/cryptocards),
- which is a small computer inside a container that should prevent an attacker from reading or modifying its data using unauthorized means.
- The 4758 has been evaluated by an independent commercial evaluator according to the FIPS 140-1 criteria and has achieved level four in both hardware and software.

Rockwell Collins Avionics

ACL2 is being used to formalize and analyze Rockwell Collins avionics microprocessors (Greve, Hardin, Wilding).

- World's first silicon Java Virtual Machine (JEM1) was formalized in ACL2 during the design.
- Formal model was used as a simulator to test design requirements.
- ACL2 model executes at C speeds.
- Several ongoing Rockwell projects are using ACL2 to analyze new designs.

Istituto per la Ricerca Scientifica e Tecnologica

A safety-critical software component was verified with ACL2.

- Union Switch and Signal has a translator (compiler) to build trainborne control systems.
- The compiler runs on COTS platforms.
- An online checker is used to certify the correctness of the translation.
- The checker was verified with ACL2.

Java Byte Code

A class description

```
("Cons" ("Object")                ; superclass chain
        ("car" "cdr")             ; fields
                                  ; methods
        (("cons" (X Y)            ;  method name and parameters
           (NEW "Cons")           ;  byte-coded method body
           (STORE TEMP)
           (LOAD TEMP)
           (LOAD X)
           (PUTFIELD "Cons" "car")
           (LOAD TEMP)
           (LOAD Y)
           (PUTFIELD "Cons" "cdr")
           (LOAD TEMP)
           (XRETURN))
         ("insert" (E X)
           (LOAD X)
           (IFEQ 18)
```

```
(LOAD X)
(GETFIELD "Cons" "car")
(LOAD E)
(SUB)
(IFLE 5)
(LOAD E)
(LOAD X)
(INVOKESTATIC "Cons" "cons" 2)
(XRETURN)
(LOAD X)
(GETFIELD "Cons" "car")
(LOAD E)
(LOAD X)
(GETFIELD "Cons" "cdr")
(INVOKESTATIC "Cons" "insert" 2)
(INVOKESTATIC "Cons" "cons" 2)
(XRETURN)
(LOAD E)
(LOAD X)
...
```

A JVM Theorem

```
(defthm isort-spec
  (implies (and (standard-hyps s0)
                (equal (next-inst s0)
                       '(invokestatic "cons" "isort" 1)))
           (let* ((x0 (top (stack (top-frame s0))))
                  (heap0 (heap s0))
                  (n0 (isort-clock x0 heap0))
                  (s1 (tjvm s0 n0))
                  (x1 (top (stack (top-frame s1))))
                  (heap1 (heap s1)))
             (implies (ok-refp x0 heap0)
                      (let ((list0 (deref* x0 heap0))
                            (list1 (deref* x1 heap1)))
                        (and (ordered list1)
                             (perm list1 list0)))))))
```

Less formally, if the method `isort` is invoked on a heap reference to a linked list, the result is a heap reference to an ordered permutaton of that list.

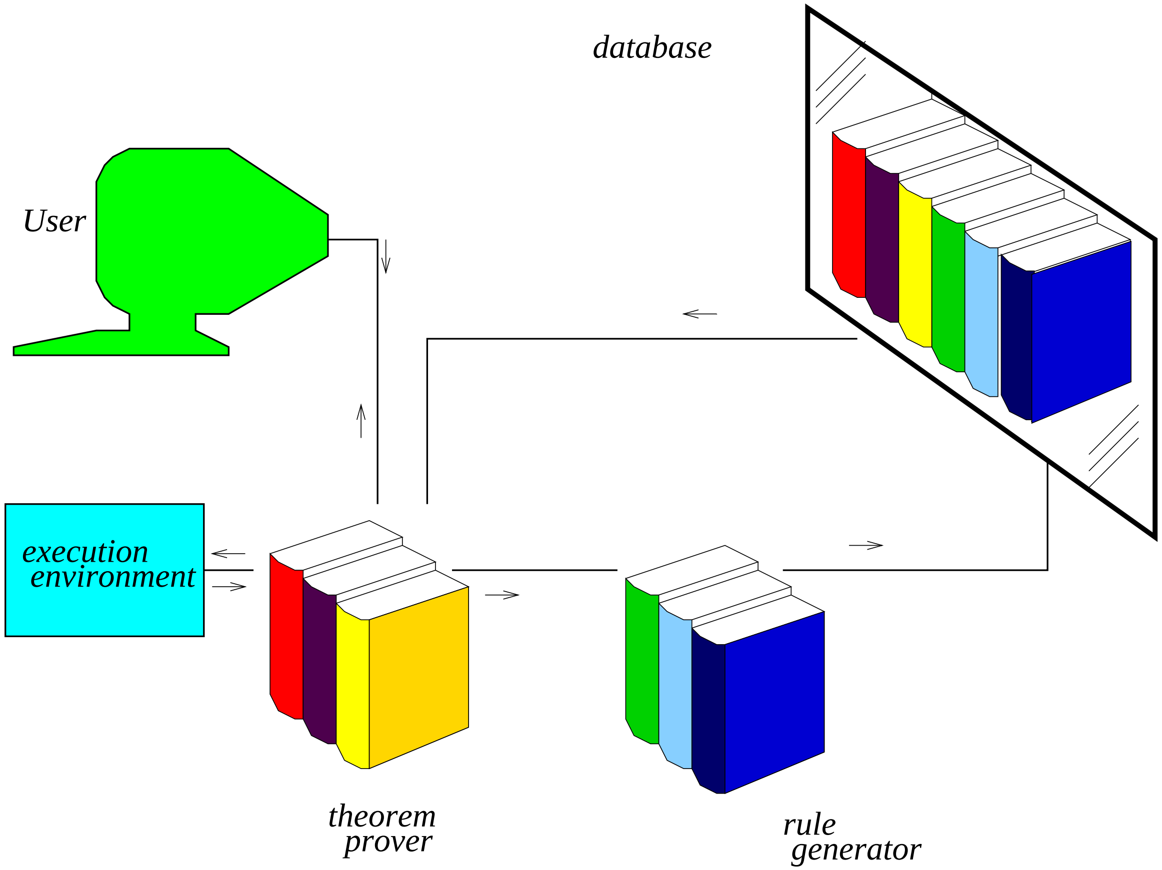
JVM Threads

A UT undergraduate (George Porter) has built a model of the JVM, including threads.

- I typed in the conjecture that the execution of one thread doesn't affect the private state of another (i.e., the non-heap part).
- ACL2 failed to prove the conjecture. The formula printed showed that it was not a theorem: A thread can create another thread!

Moral: Even when theorems aren't proved, models and proof attempts are informative.

An Amazing Fact



This opens up exciting possibilities in metatheoretic reasoning.

About ACL2

The ACL2 home page: http://www.cs.utexas.edu/users/moore/acl2.

Distributed without fee under the GNU General Public License.

Books about ACL2:

- How to use ACL2:
  Computer-Aided Reasoning: An Approach, Kaufmann, Manolios, Moore, Kluwer Academic Publishers, 2000.

- What is being done with it:
  Computer-Aided Reasoning: ACL2 Case Studies, Kaufmann, Manolios, Moore (eds.), Kluwer Academic Publishers, 2000.