

# Formal Methods: Practice and Pedagogy

J Strother Moore

Department of Computer Sciences  
The University of Texas at Austin

*presented at*

IX Jornadas de Enseñanza Universitaria de la Informática  
Cádiz, 9–11 July, 2003

# Formal Methods

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*

# Boyer-Moore Project

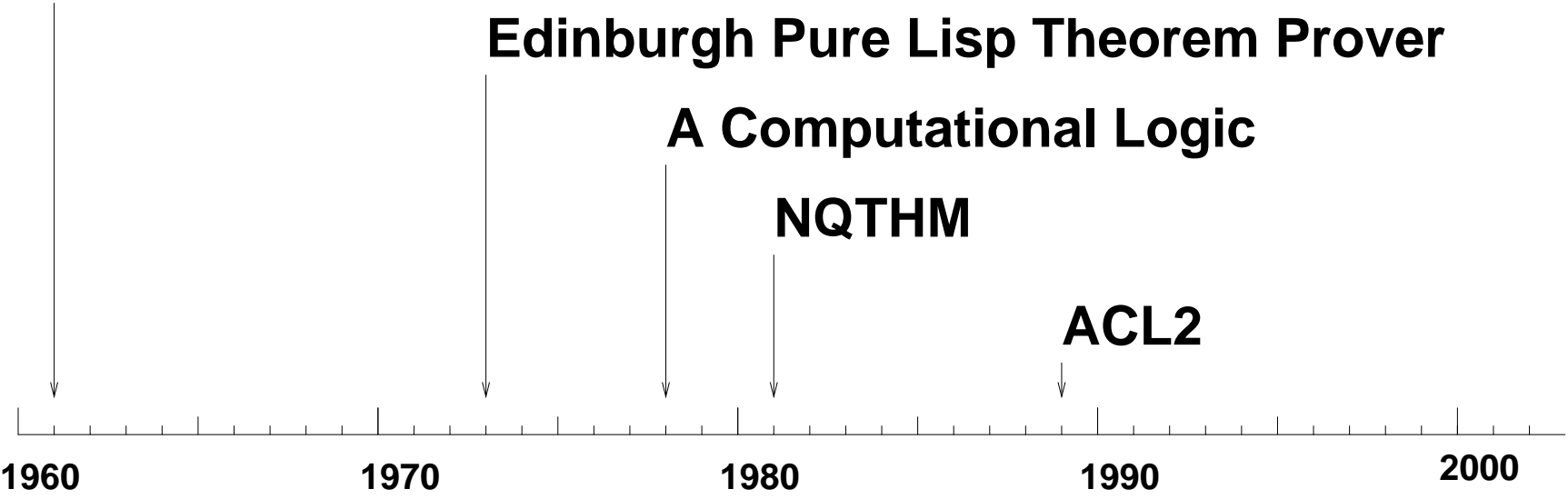
McCarthy's "Theory of Computation"

Edinburgh Pure Lisp Theorem Prover

A Computational Logic

NQTHM

ACL2



Boyer



Moore



Kaufmann



# The Expressiveness Spectrum

**Prop Calc**

**Set Theory**



**ACL2**

**BDD**

**zChaff**

**PVS**

**HOL**

**Coq**

**SMV**

**COSPAN**

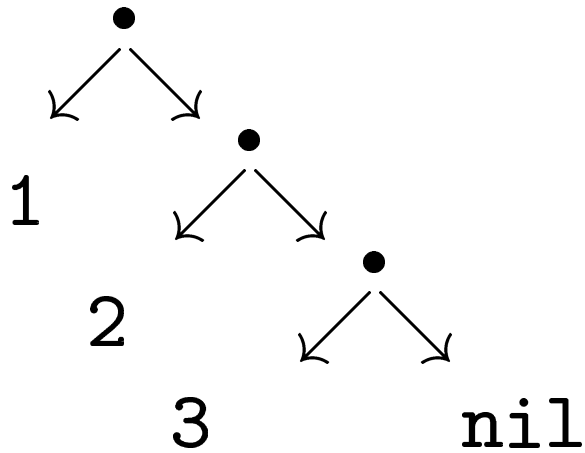
# A Classic Challenge Theorem

**Theorem:** List concatenation (“append”) is associative.

$$\text{equal } (\text{append } (\text{append } a \ b) \ c) \\ (\text{append } a \ (\text{append } b \ c))$$
$$\forall a \forall b$$
$$\text{append}(\text{append}(a, b), c)$$
$$=$$
$$\text{append}(a, \text{append}(b, c)).$$

# Examples

```
(cons 1 (cons 2 (cons 3 nil)))  
= '(1 2 3)
```



```
(append '(1 2 3) (append '(4 5 6) '(7 8 9)))  
= '(1 2 3 4 5 6 7 8 9)
```

# The Definition of append

```
(defun append (x y)
  (if (endp x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

$(\text{endp } x) \rightarrow (\text{append } x \ y) = y$

$\neg(\text{endp } x) \rightarrow (\text{append } x \ y) =$   
     $(\text{cons } (\text{car } x)$   
         $(\text{append } (\text{cdr } x) \ y))$

# A Few Axioms

$t \neq \text{nil}$

$x = \text{nil} \rightarrow (\text{if } x \ y \ z) = z$

$x \neq \text{nil} \rightarrow (\text{if } x \ y \ z) = y$

$(\text{car } (\text{cons } x \ y)) = x$

$(\text{cdr } (\text{cons } x \ y)) = y$

$(\text{endp } \text{nil}) = t$

$(\text{endp } (\text{cons } x \ y)) = \text{nil}$

```
(equal (append (append a b) c)
       (append a (append b c)))
```

(equal (append (append a b) c)  
          (append a (append b c)))

Proof: by induction on a.

(equal (append (append a b) c)  
          (append a (append b c)))

Proof: by induction on a.

Base Case: (endp a).

(equal (append (append a b) c)  
          (append a (append b c)))

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (append b c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: `(endp a)`.

```
(equal (append b c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: `(endp a)`.

```
(equal (append b c)
       (append b c))
```

(equal (append (append a b) c)  
          (append a (append b c)))

Proof: by induction on a.

Base Case: (endp a).

(equal (append b c)  
          (append b c))

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

T

(equal (append (append a b) c)  
          (append a (append b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).  
(equal (append (append a b) c)  
          (append a (append b c)))

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (cons (car a)
                  (append (cdr a) b)) c)
       (append a (append b c)))
```



```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
       (append (append (cdr a) b) c))
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
            (append (append (cdr a) b) c))
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
            (append (append (cdr a) b) c))
       (cons (car a)
          (append (cdr a) (append b c))))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
       (append (append (cdr a) b) c))
       (cons (car a)
       (append (cdr a) (append b c))))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal
  (append (append (cdr a) b) c)
  (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (append (cdr a) b) c)
       (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (append (cdr a) b) c)
       (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

T

(equal (append (append a b) c)  
 (append a (append b c)))

Proof: by induction on a.

Q.E.D.

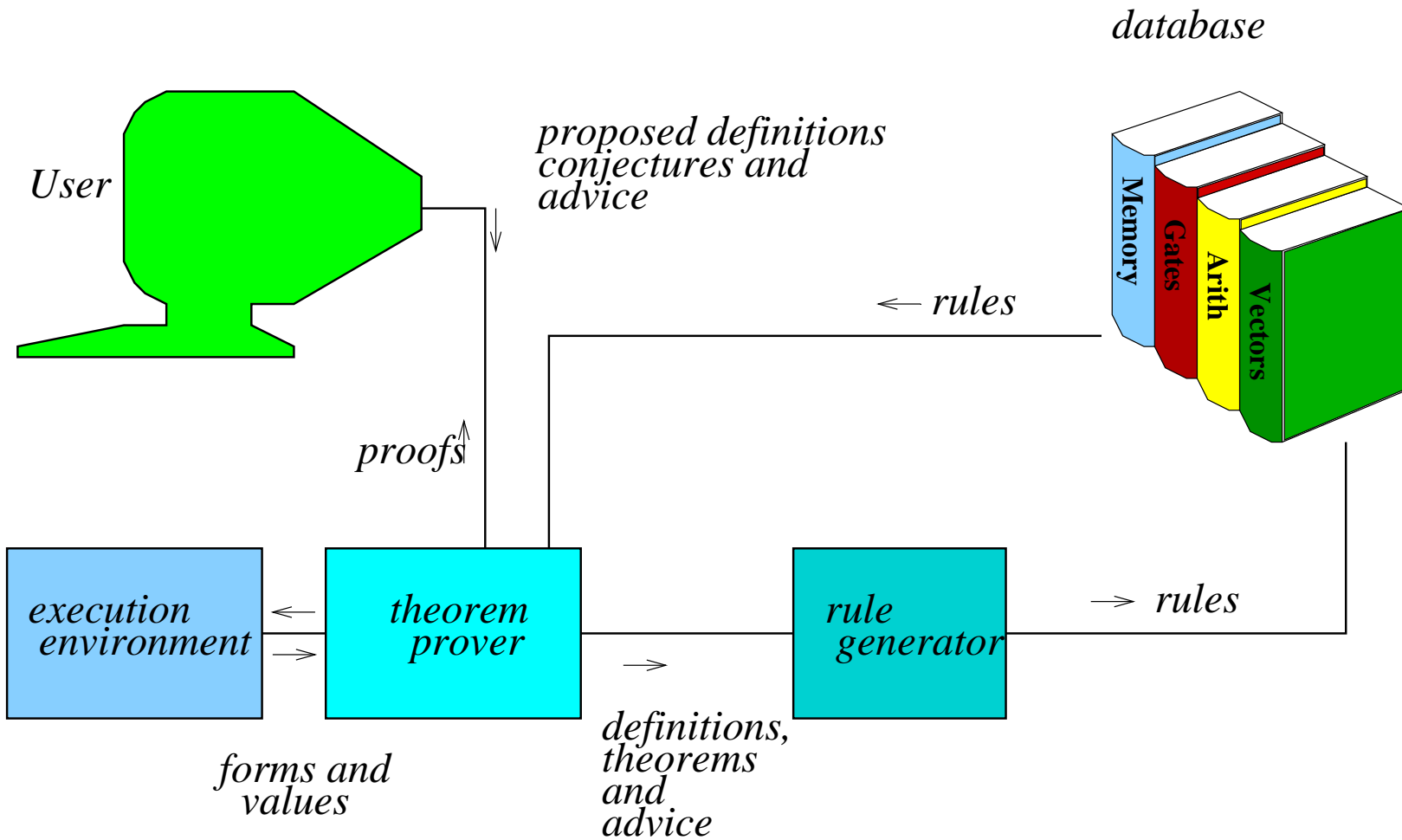
```
(defun append (x y)
  (if (endp x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Induction and recursion are duals.

## Theorem

```
(equal (append (append a b) c)
       (append a (append b c)))
```

# Demo 1



# The Need for Formal Methods in Practice

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

## AMD K5 Algorithm $\text{FDIV}(p, d, mode)$

1.  $sd_0 = \text{lo\_okup}(d)$  [exact 17 8]
2.  $d_r = d$  [away 17 32]
3.  $sdd_0 = sd_0 \times d_r$  [away 17 32]
4.  $sd_1 = sd_0 \times \text{comp}(sd_0, 32)$  [trunc 17 32]
5.  $sdd_1 = sd_1 \times d_r$  [away 17 32]
6.  $sd_2 = sd_1 \times \text{comp}(sd_1, 32)$  [trunc 17 32]
- ... .. = ... ..
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.  $fdiv = qq_1 + q_0$  *mode*

# Using the Reciprocal

$$\begin{array}{r}
 36. \\
 + \quad -17 \\
 + \quad .0034 \\
 + \quad \underline{-0.00066} \\
 35.833334 \\
 \hline
 12 \overline{) 430.000000} \\
 \underline{432.} \\
 -2. \\
 \underline{-2.04} \\
 .04 \\
 \underline{.0408} \\
 - .0008 \\
 \underline{- .000792} \\
 - .000008
 \end{array}$$

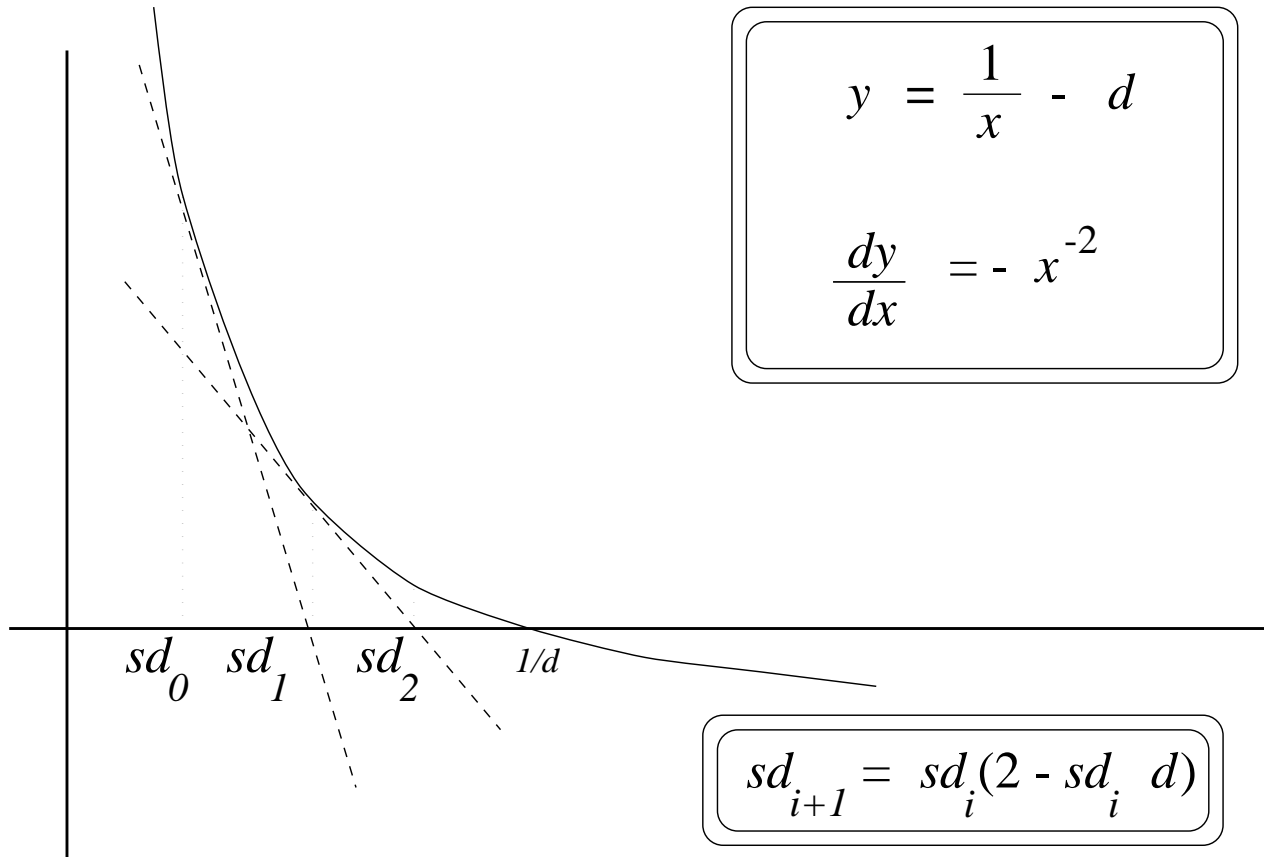
Reciprocal Calculation:  
 $1/12 = 0.083\overline{3} \approx 0.083 = sd_2$

Quotient Digit Calculation:

$$\begin{aligned}
 0.083 \times 40000 &= 35.6900000 \approx 36.000000 = q_0 \\
 0.083 \times -2.0000 &= -1.6600000 \approx -1.700000 = q_1 \\
 0.083 \times .0400 &= .0033200 \approx .003400 = q_2 \\
 0.083 \times -.0008 &= -.0000664 \approx -.000067 = q_3
 \end{aligned}$$

Summation of Quotient Digits:  
 $q_0 + q_1 + q_2 + q_3 = 35.833333$

# Computing the Reciprocal



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 <sub>2</sub>	0.1111111 <sub>2</sub>	1.0100000 <sub>2</sub>	0.11001100 <sub>2</sub>	1.1000000 <sub>2</sub>	0.10101010 <sub>2</sub>	1.1100000 <sub>2</sub>	0.10010010 <sub>2</sub>
1.0000001 <sub>2</sub>	0.1111101 <sub>2</sub>	1.0100001 <sub>2</sub>	0.11001011 <sub>2</sub>	1.1000001 <sub>2</sub>	0.10101001 <sub>2</sub>	1.1100001 <sub>2</sub>	0.10010001 <sub>2</sub>
1.0000010 <sub>2</sub>	0.1111101 <sub>2</sub>	1.0100010 <sub>2</sub>	0.11001010 <sub>2</sub>	1.1000010 <sub>2</sub>	0.10101000 <sub>2</sub>	1.1100010 <sub>2</sub>	0.10010001 <sub>2</sub>
1.0000011 <sub>2</sub>	0.1111100 <sub>2</sub>	1.0100011 <sub>2</sub>	0.11001000 <sub>2</sub>	1.1000011 <sub>2</sub>	0.10101000 <sub>2</sub>	1.1100011 <sub>2</sub>	0.10010000 <sub>2</sub>
1.0000100 <sub>2</sub>	0.1111011 <sub>2</sub>	1.0100100 <sub>2</sub>	0.11000111 <sub>2</sub>	1.1000100 <sub>2</sub>	0.10100111 <sub>2</sub>	1.1100100 <sub>2</sub>	0.10001111 <sub>2</sub>
1.0000101 <sub>2</sub>	0.1111010 <sub>2</sub>	1.0100101 <sub>2</sub>	0.11000110 <sub>2</sub>	1.1000101 <sub>2</sub>	0.10100110 <sub>2</sub>	1.1100101 <sub>2</sub>	0.10001111 <sub>2</sub>
1.0000110 <sub>2</sub>	0.1111010 <sub>2</sub>	1.0100110 <sub>2</sub>	0.11000101 <sub>2</sub>	1.1000110 <sub>2</sub>	0.10100101 <sub>2</sub>	1.1100110 <sub>2</sub>	0.10001110 <sub>2</sub>
1.0000111 <sub>2</sub>	0.1111001 <sub>2</sub>	1.0100111 <sub>2</sub>	0.11000100 <sub>2</sub>	1.1000111 <sub>2</sub>	0.10100100 <sub>2</sub>	1.1100111 <sub>2</sub>	0.10001110 <sub>2</sub>
1.0001000 <sub>2</sub>	0.1111000 <sub>2</sub>	1.0101000 <sub>2</sub>	0.11000010 <sub>2</sub>	1.1001000 <sub>2</sub>	0.10100011 <sub>2</sub>	1.1101000 <sub>2</sub>	0.10001101 <sub>2</sub>
1.0001001 <sub>2</sub>	0.1110111 <sub>2</sub>	1.0101001 <sub>2</sub>	0.11000001 <sub>2</sub>	1.1001001 <sub>2</sub>	0.10100011 <sub>2</sub>	1.1101001 <sub>2</sub>	0.10001100 <sub>2</sub>
1.0001010 <sub>2</sub>	0.1110110 <sub>2</sub>	1.0101010 <sub>2</sub>	0.11000000 <sub>2</sub>	1.1001010 <sub>2</sub>	0.10100010 <sub>2</sub>	1.1101010 <sub>2</sub>	0.10001100 <sub>2</sub>
...	...	...	...	...	...	...	...
1.0010110 <sub>2</sub>	0.11011010 <sub>2</sub>	1.0110110 <sub>2</sub>	0.10110100 <sub>2</sub>	1.1010110 <sub>2</sub>	0.10011001 <sub>2</sub>	1.1110110 <sub>2</sub>	0.10000101 <sub>2</sub>
1.0010111 <sub>2</sub>	0.11011000 <sub>2</sub>	1.0110111 <sub>2</sub>	0.10110011 <sub>2</sub>	1.1010111 <sub>2</sub>	0.10011000 <sub>2</sub>	1.1110111 <sub>2</sub>	0.10000100 <sub>2</sub>
1.0011000 <sub>2</sub>	0.1101011 <sub>2</sub>	1.0111000 <sub>2</sub>	0.10110010 <sub>2</sub>	1.1011000 <sub>2</sub>	0.10010111 <sub>2</sub>	1.1111000 <sub>2</sub>	0.10000100 <sub>2</sub>
1.0011001 <sub>2</sub>	0.1101010 <sub>2</sub>	1.0111001 <sub>2</sub>	0.10110001 <sub>2</sub>	1.1011001 <sub>2</sub>	0.10010111 <sub>2</sub>	1.1111001 <sub>2</sub>	0.10000011 <sub>2</sub>
1.0011010 <sub>2</sub>	0.1101010 <sub>2</sub>	1.0111010 <sub>2</sub>	0.10110000 <sub>2</sub>	1.1011010 <sub>2</sub>	0.10010110 <sub>2</sub>	1.1111010 <sub>2</sub>	0.10000011 <sub>2</sub>
1.0011011 <sub>2</sub>	0.1101001 <sub>2</sub>	1.0111011 <sub>2</sub>	0.10101111 <sub>2</sub>	1.1011011 <sub>2</sub>	0.10010101 <sub>2</sub>	1.1111011 <sub>2</sub>	0.10000010 <sub>2</sub>
1.0011100 <sub>2</sub>	0.1101000 <sub>2</sub>	1.0111100 <sub>2</sub>	0.10101110 <sub>2</sub>	1.1011100 <sub>2</sub>	0.10010101 <sub>2</sub>	1.1111100 <sub>2</sub>	0.10000010 <sub>2</sub>
1.0011101 <sub>2</sub>	0.1101000 <sub>2</sub>	1.0111101 <sub>2</sub>	0.10101101 <sub>2</sub>	1.1011101 <sub>2</sub>	0.10010100 <sub>2</sub>	1.1111101 <sub>2</sub>	0.10000001 <sub>2</sub>
1.0011110 <sub>2</sub>	0.1100111 <sub>2</sub>	1.0111110 <sub>2</sub>	0.10101100 <sub>2</sub>	1.1011110 <sub>2</sub>	0.10010011 <sub>2</sub>	1.1111110 <sub>2</sub>	0.10000001 <sub>2</sub>
1.0011111 <sub>2</sub>	0.1100110 <sub>2</sub>	1.0111111 <sub>2</sub>	0.10101011 <sub>2</sub>	1.1011111 <sub>2</sub>	0.10010011 <sub>2</sub>	1.1111111 <sub>2</sub>	0.10000000 <sub>2</sub>

# The Formal Model of the Code

```
(defun FDIV (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)))
    (fdiv (round (+ qq1 q0) mode)))
    (or (first-error sd0 dr sdd0 sd1 sdd1 ... fddiv)
        fddiv)))
```

# IEEE 754 Floating Point Standard

Elementary operations are to be performed as though the infinitely precise (standard mathematical) operation were performed and then the result rounded to the indicated precision.

# The K5 FDIV Theorem (1200 lemmas)

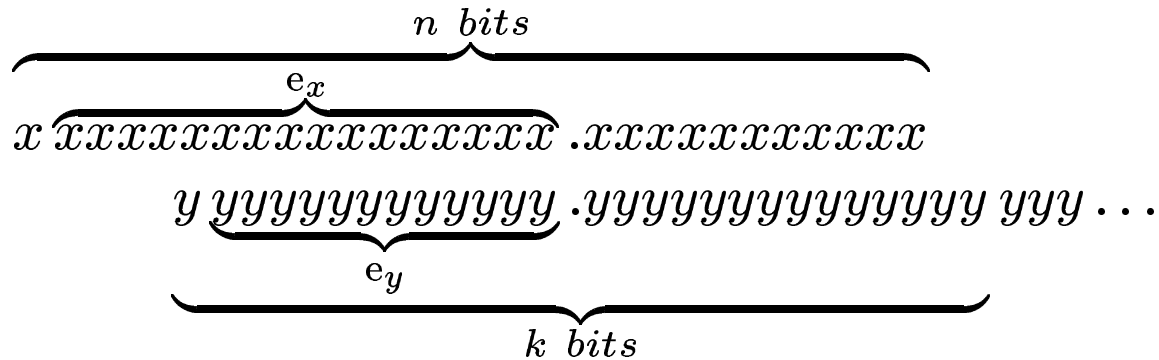
```
(defthm FDIV-divides
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
    (equal (FDIV p d mode)
           (round (/ p d) mode))))
```

(by Moore, Lynch and Kaufmann, in 1995, *before the K5 was fabricated*)

# A Lemma from the FP Books

## Lemma 7.3.2 (Sticky Plus)

Let  $x$  be a non-0 rational that fits in  $n > 0$  bits, which is to say  $\text{trunc}(x, n) = x$ . Let  $y$  be a rational whose exponent is at least two smaller than that of  $x$ ,  $1 + e_y < e_x$ . Let  $k$  be a positive integer such that  $n + e_y - e_x < k$ .



Then  $\text{sticky}(x + y, n) = \text{sticky}(x + \text{sticky}(y, k), n)$ .

# AMD Athlon 1997

All elementary floating-point operations, FADD, FSUB, FMUL, FDIV, and FSQRT, on the AMD Athlon were

- specified in ACL2 to be IEEE compliant,
- proved to meet their specifications, and
- the proofs were checked mechanically.

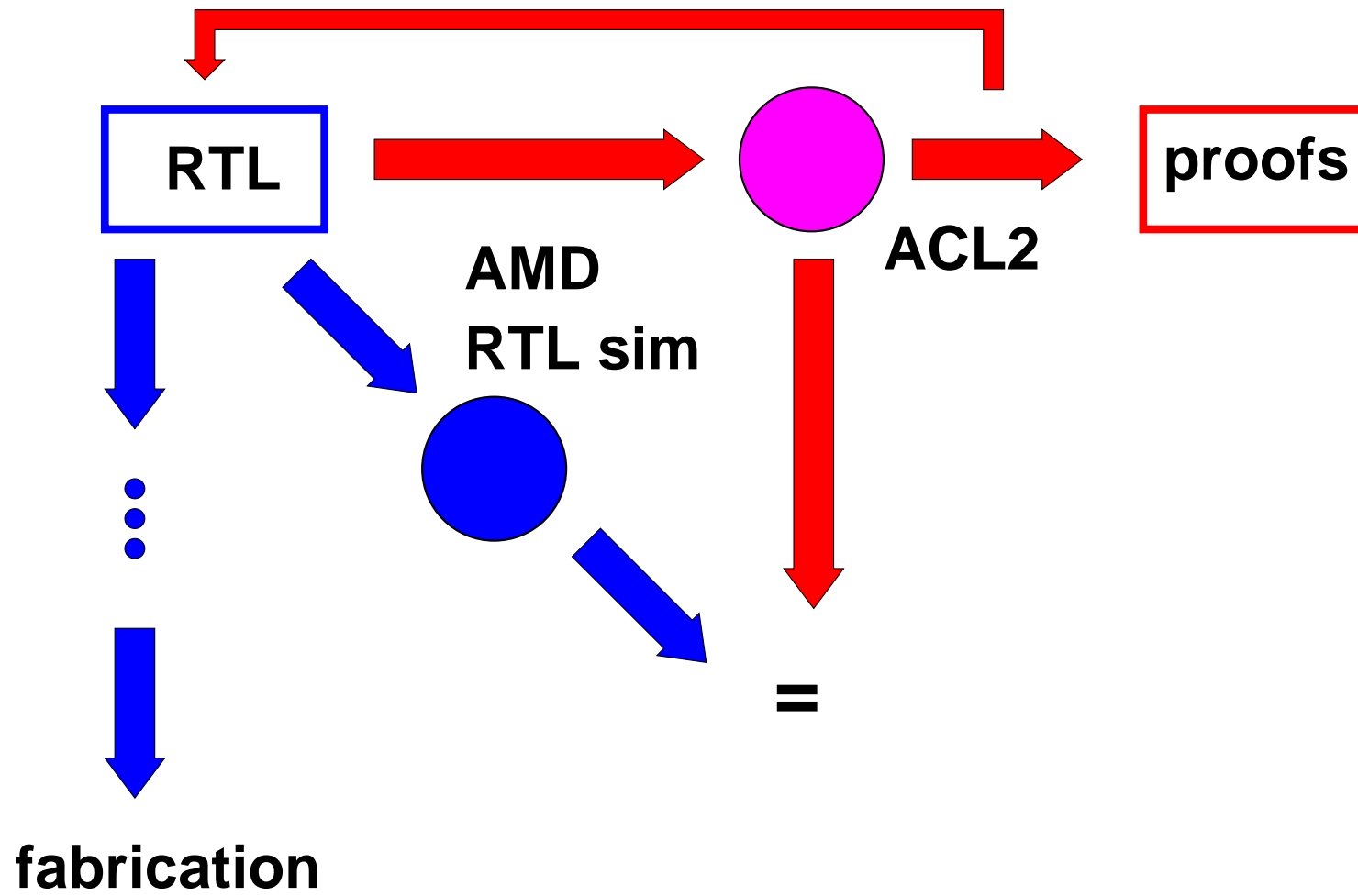
# AMD Athlon FMUL

```
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
```

```

//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];
expx[14:0] = x[78:64]; expy[14:0] = y[78:64];

```



# The Athlon FMUL Theorem

```
(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))))
```

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.

This work was done primarily by David Russinoff and Art Flatau, of AMD.

# Other Work at AMD

AMD is using ACL2 to reason about multi-processor implementations, at the algorithm level and close to the RTL level.

They have proved a progress theorem about a model hand-derived from the RTL.

They have proved correctness at the algorithm level of a mechanism related to speculative reads.

New bugs (which were undetected after simulation) have been found and fixed before tapeout.

# Other Commercial Work

- FDIV on AMD K5 (Moore-Kaufmann-Lynch)
- AMD Athlon floating point (Russinoff-Flatau)
- **Motorola 68020 and Berkeley C String Library (Yu)**
- ...

# Motorola 68020 and the C String Library

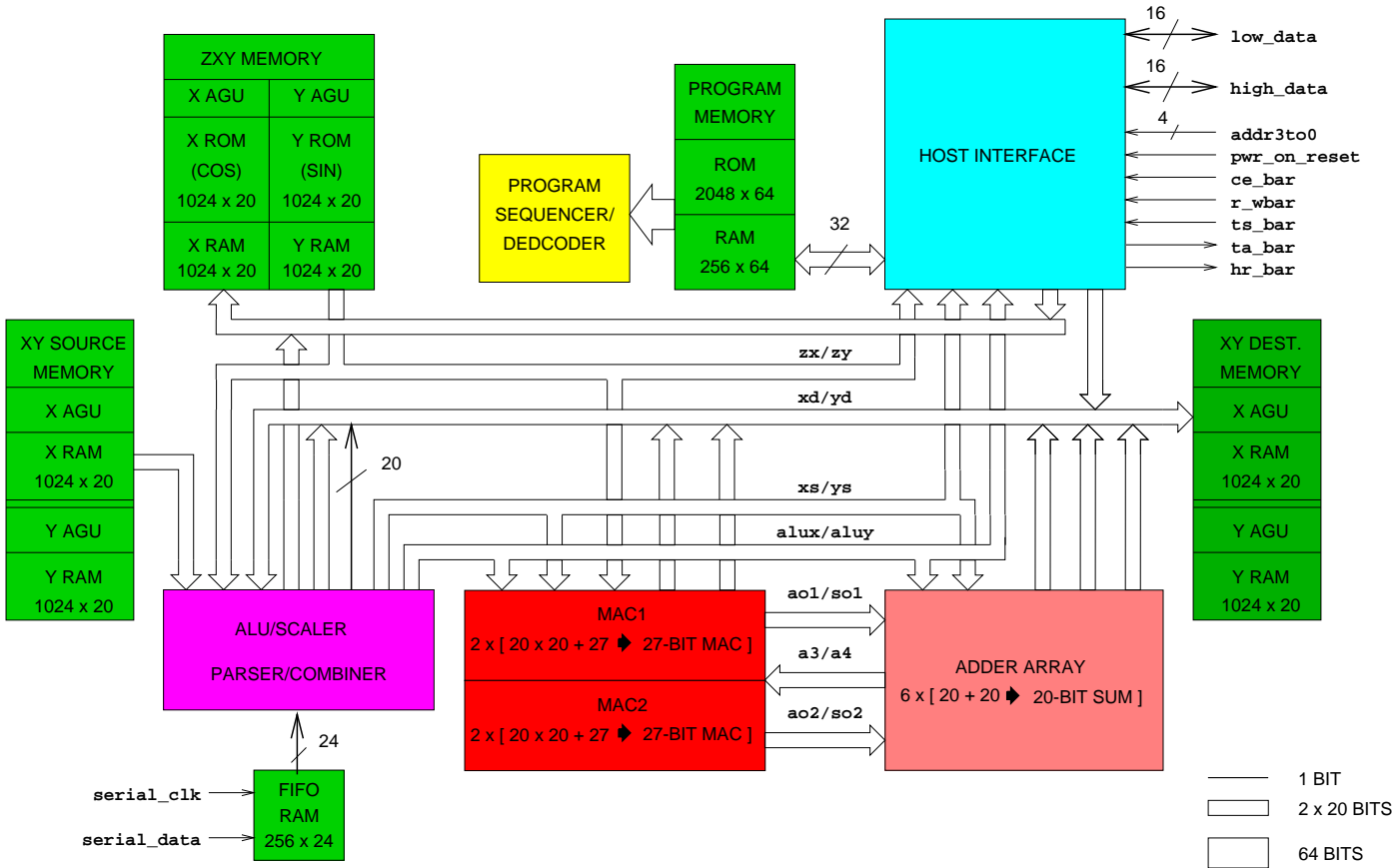
```
/* copy char from[] to to[] */
char *
strcpy(to, from)
    register char *to, *from;
{ char *save = to;
  for (; *to = *from; ++from, ++to);
  return(save);
}
gcc -o ...
```

```
0x2558 <strcpy>:      linkw fp,#0
0x255c <strcpy+4>:    moveal fp@(8),a0
0x2560 <strcpy+8>:    moveal fp@(12),a1
0x2564 <strcpy+12>:   movel  a0,d1
0x2566 <strcpy+14>:   bra  0x256c <strcpy+20>
0x2568 <strcpy+16>:   adqw  #1,a1
0x256a <strcpy+18>:   adqw  #1,a0
0x256c <strcpy+20>:   moveb  a1@,d0
```

# Other Commercial Work

- Motorola 68020 and Berkeley C String Library (Yu)
- Motorola CAP DSP (Brock)
- ...

# Motorola CAP DSP

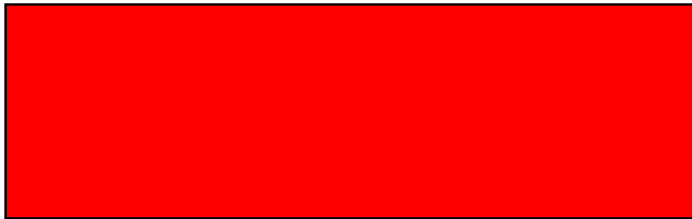




**ROM containing  
50 microcoded  
DSP algorithms**

**Pipelined  
microarchitecture**

**Sequential  
microcode ISA**



**=**



**(if no hazards)**

# Other Commercial Work

- ...
- Motorola CAP DSP (Brock)
- IBM 4758 secure co-processor (Austel)
- Union Switch and Signal safety-critical checker (Bertolli)
- ...

# IBM PCI Cryptographic Coprocessor

Top  
security  
rating

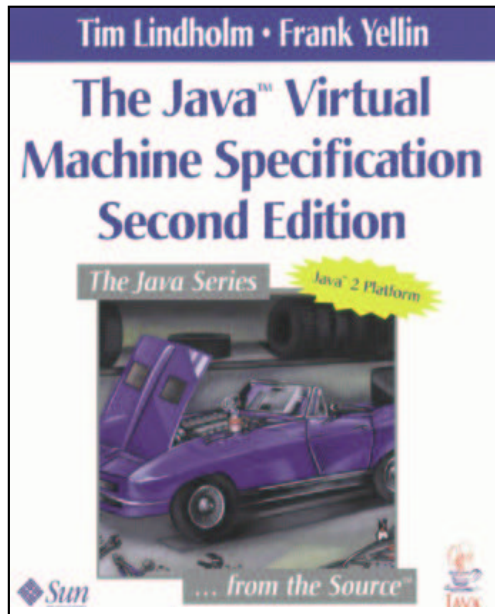


The security model was formalized in ACL2 and certain properties were proved to obtain FIPS 140-1 Level Four certification.

# Other Commercial Work

- ...
- IBM 4758 secure co-processor (Austel)
- Union Switch and Signal safety-critical checker (Bertolli)
- Rockwell Collins / aJile Systems JEM1 (Hardin-Greve-Wilding)
- Java and the JVM (UT Austin with Sun and others)

# Java and the JVM



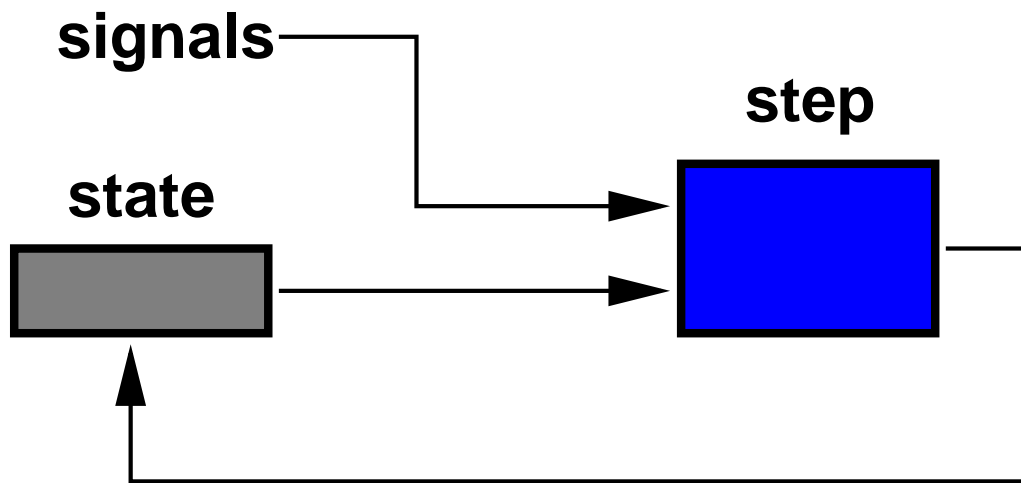
```
; JVM in ACL2

(defun make-state (tt hp ct)
  ...)

(defun step (th s)
  ...)

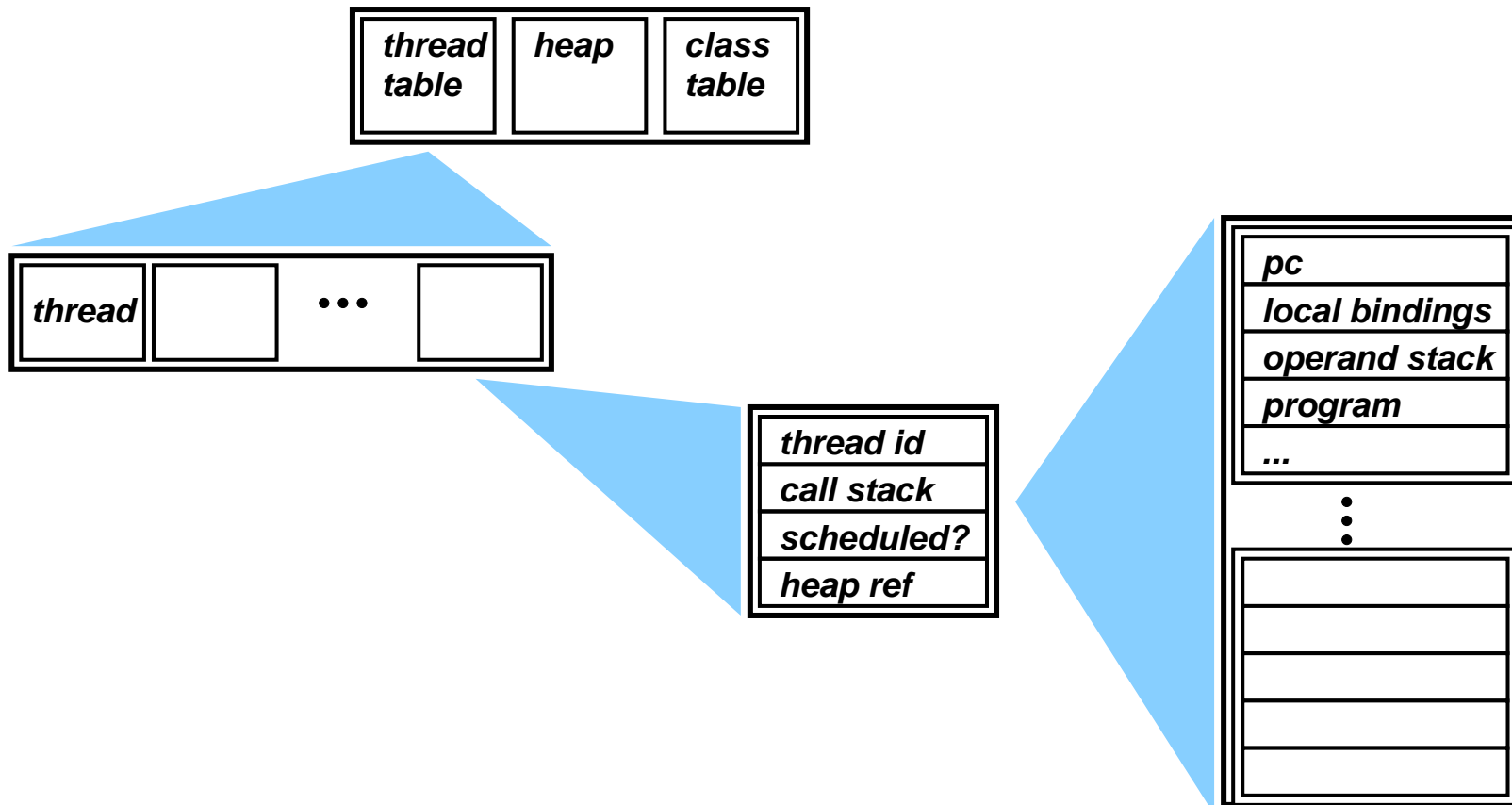
(defun run (sched s)
  (if (endp sched)
      s
      (run
       (cdr sched)
       (step (car sched) s))))
```

# Demo 2



```
(defun run (signals state)
  (if (endp signals)
      state
      (run (cdr signals)
           (step (car signals) state))))
```

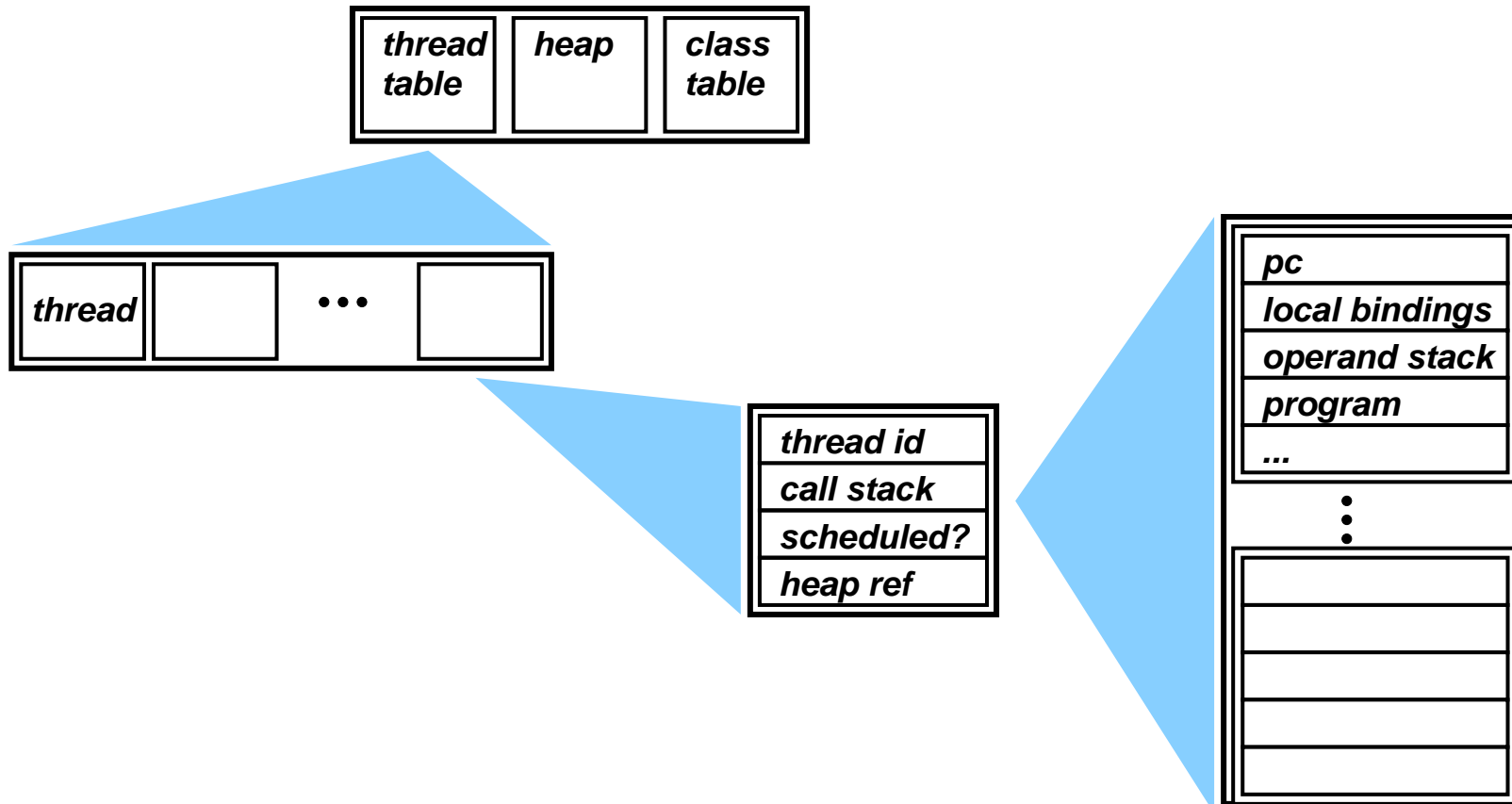
# Our State: $\langle tt, hp, ct \rangle$



```
(defun step (th s)
  (if (equal (call-stack-status th s)
            'SCHEDULED)
      (do-inst (next-inst th s) th s)
      s))
```

In our case, `th` is a thread identifier and is treated as a “signal.”

# Our State: $\langle tt, hp, ct \rangle$



```
(defun do-inst (inst th s)
  (case (op-code inst)
    (AALOAD (execute-AALOAD inst th s))
    (AASTORE (execute-AASTORE inst th s))
    (ALOAD (execute-ALOAD inst th s))
    (ALOAD_0 (execute-ALOAD_X inst th s 0))
    (ALOAD_1 (execute-ALOAD_X inst th s 1))
    (ALOAD_2 (execute-ALOAD_X inst th s 2))
    (ALOAD_3 (execute-ALOAD_X inst th s 3))
    ...))
```

# The JVM Spec from Sun

**iload\_0**

**Operation**

Load `int` from local variable 0

**Format**

`iload_0`

**Form**

26 (0x1a)

**Operand Stack**

...  $\Rightarrow$  ..., value

**Description**

The local variable at 0 must contain an `int`.

The value of the local variable at 0 is pushed onto the operand stack.

Note: `ILOAD_0`, ... `ILOAD_3` are 1-byte specializations of the 3-byte `ILOAD n`.

```

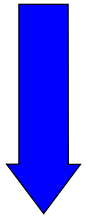
(defun execute-ILOAD (inst th s)
                                ; inst = (ILOAD n)
  (let ((n (arg1 inst)))
    (modify th s
      :pc (+ (inst-length inst)
             (pc (top-frame th s)))
      :stack (push (nth n
                     (locals (top-frame th s)))
                   (stack (top-frame th s))))))

```

# Some Java

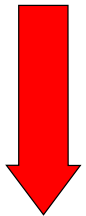
```
class Demo {  
  
    public static int fact(int n){  
        if (n>0) {return n*fact(n-1);}  
        else return 1;  
    }  
  
    public static void main(String[] args){  
        int n = Integer.parseInt(args[0], 10);  
        System.out.println(fact(n));  
        return;  
    } }  
}
```

**.java**



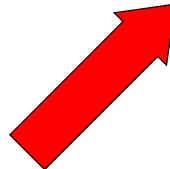
**javac**

**.class**



**jvm2acl2**

**.lisp**



**Theorems**

**“fact(5)=120”**



**“fact(n)=n!”**

# Demo 3

# Performance

The ACL2 function `run` is an *executable formal model* of the JVM.

On a 728 MHz processor, we get about 75K bytecodes/second.

With optimization, we get about 3M bytecodes/sec.

```
% java Demo 20
```

```
-2102132736
```

```
ACL2 >(acl2-Demo 20)
```

```
-2102132736
```

```
ACL2 >(! 20)
```

```
2432902008176640000
```

```
ACL2 >(int-fix (! 20))
```

```
-2102132736
```

# Conjecture

Our Java `fact` method computes the two's-complement integer represented by the low-order 32 bits of the actual factorial.

# Demo 4

# Pedagogy

What are the lessons here for undergraduate Computer Science majors?

- Mechanized formal analysis of digital artifacts is sometimes *possible* and *effective*.
- Formal specifications can serve as *prototypes* and *simulators*.
- Formal specification of correctness often requires *definition* of new concepts.

- Formal proof is facilitated by *structured code development* and *compositional reasoning*.
- There is an illuminating duality between *induction* and *recursion*.
- *Code analysis techniques* are separable from *language semantic techniques*. E.g., *operational semantics* can be used directly to support *Floyd-Hoare style code proofs*.
- *Tool support* for formal methods is available across a wide spectrum of applications.

- Formal methods is *not a panacea* but just one of the tools available to the system designer.

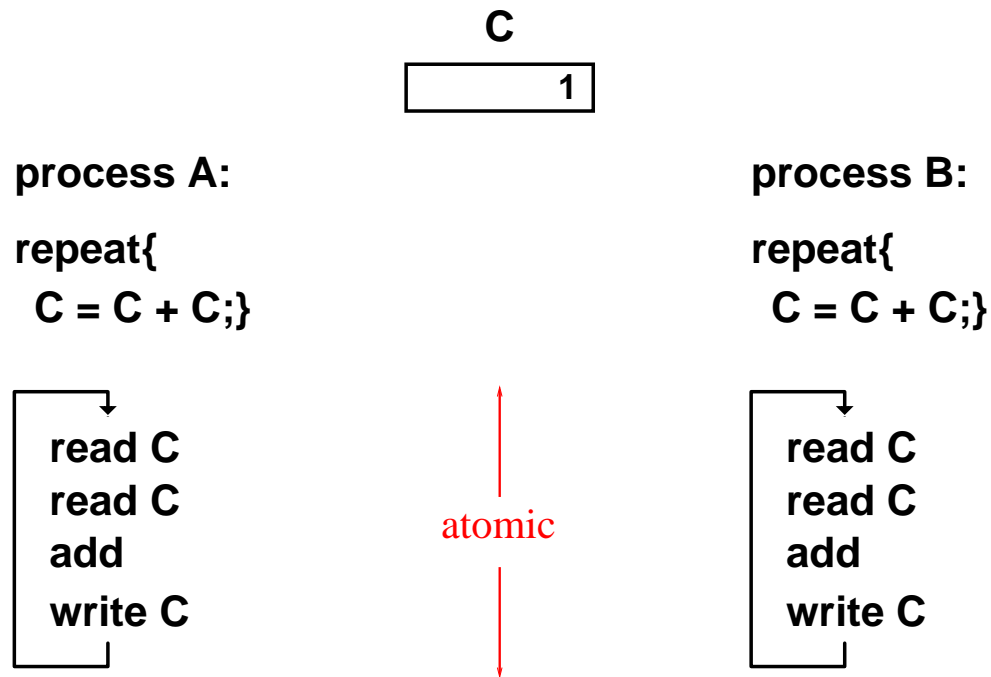
# Undergraduate “Formal Methods” Courses at UT Austin

- 313K *Logic, Sets and Functions* – introduction to mathematical logic (year 1).
- 337 *Theory in Programming Practice* – illustrative examples of the use of formal analysis in program design (year 2).
- 336 *Analysis of Programs* – code analysis and proof techniques (year 2).

- 341 *Automata Theory* – automata and formal languages (year 3).
- 378 *Computer-Aided Verification* – model checking (year 4).
- 378 *Formal Model of the JVM* – theorem proving (year 4).

There are also a regular stream of topics classes on security, distributed programming, and hardware verification.

# An Entertaining Puzzle: The Thread Game



Theorem? For every positive integer  $n$  there is an interleaving of A and B steps that produces  $C = n$ .

# References

*Computer-Aided Reasoning: An Approach*,  
Kaufmann, Manolios, Moore, Kluwer Academic  
Publishers, 2000.

*Computer-Aided Reasoning: ACL2 Case Studies*,  
Kaufmann, Manolios, Moore (eds.), Kluwer  
Academic Publishers, 2000.

<http://www.cs.utexas.edu/users/moore/acl2>