The Mechanically Verified Stack Challenge

J Strother Moore
Inman Chair and Chairman

Department of Computer Sciences
University of Texas at Austin

The Premise

The formal methods community is in danger of settling into comfortable niches far smaller than our potential suggests.

To maximize the impact of our science, we must demonstrate that mechanized formal methods can dramatically decrease time-to-market while producing correct systems.

This requires

pervasive use of formal methods throughout the abstraction hierarchy.

The Challenge

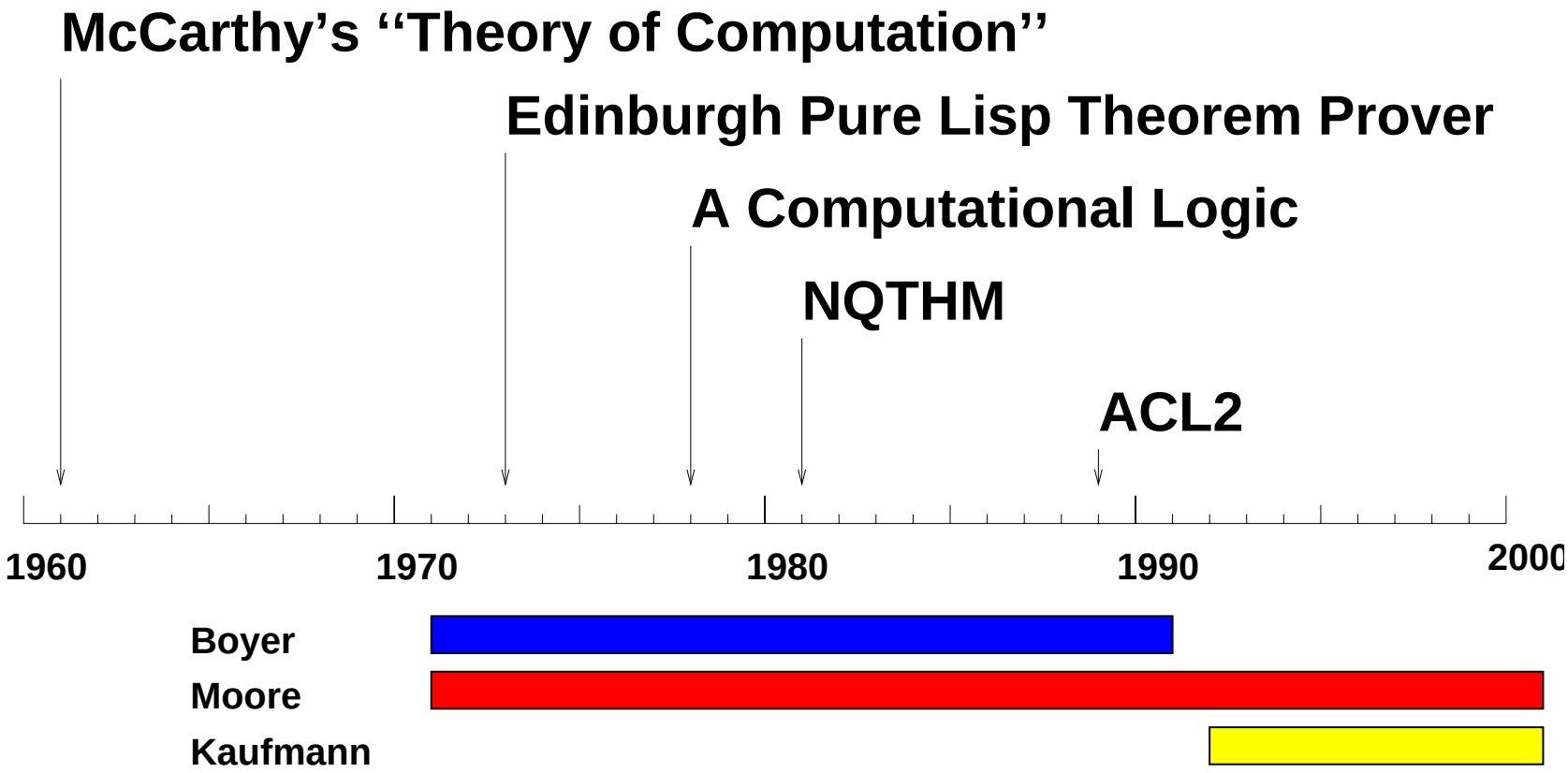
Each major group in the formal methods community should design and mechanically verify a practical embedded system, from transistors to software.

A Personal Perspective

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*

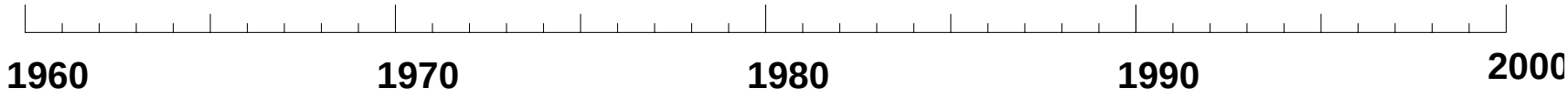
Boyer-Moore Project



simple list processing

academic math and cs

commercial applications



1970s

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

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

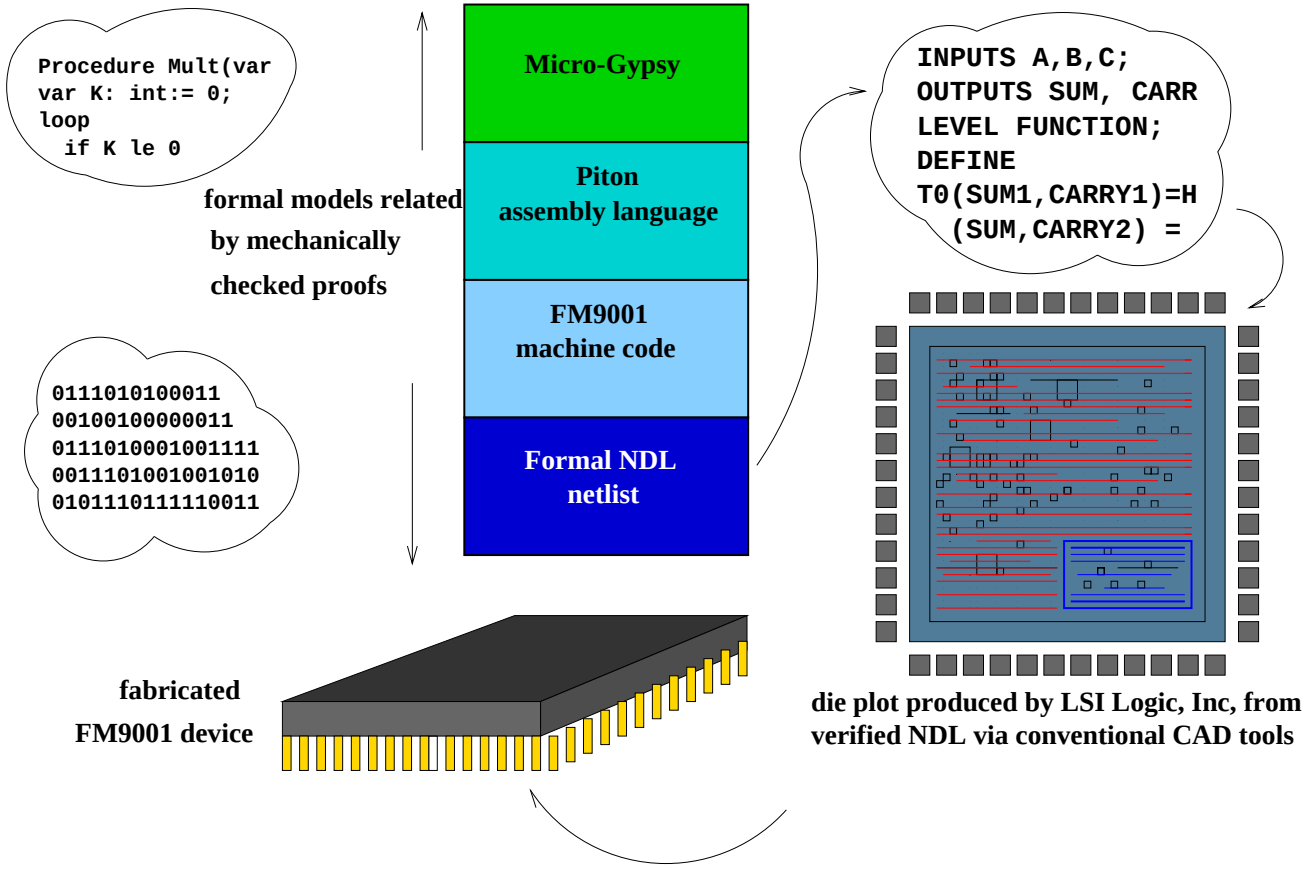
1980s Academic Math

- invertibility of RSA encryption
- undecidability of the halting problem
- Gödel's First Incompleteness Theorem (Shankar)
- Gauss' law of quadratic reciprocity (Russinoff)

1980s Academic CS

- microprocessor: gates to machine code (Hunt)
- assembler-linker-loader (Moore and Kaufmann)
- compiler (Young)

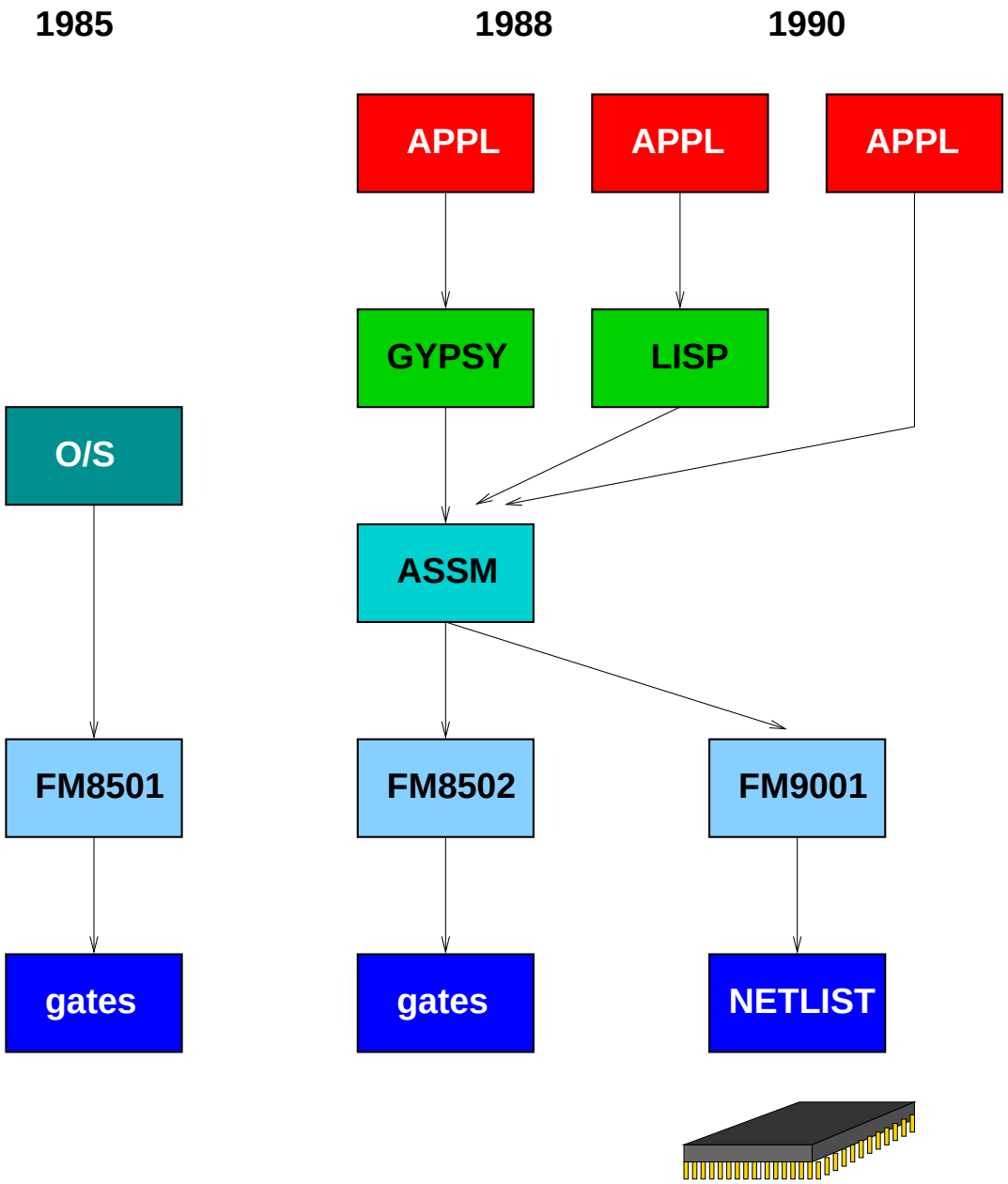
The CLI Stack



Additional Stack Components

- operating system (Bevier)
- Lisp compiler (Flatau)
- applications (Wilding)

- Applications were mechanically verified with respect to the high-level languages.
- Correct binary images were produced by mechanically verified tools.
  - `compile`
  - `assemble`
  - `link`
  - `load`


1985
1988
1990
APPL
APPL
APPL
GYPSY
LISP
O/S
ASSM
FM8501
FM8502
FM9001
gates
gates
NETLIST

What's Missing?

- The FM9001 was modeled at the gate level.

- The FM9001 had an unrealistically simple architecture: memory-mapped io, no pipeline, no speculation, no floating point, crude interrupts, no cache.

- None of the programming languages had io or floating point.

- The high-level languages were too simple to be of practical use.
  - The micro-Gypsy language had very few primitives and no dynamically allocated data types (e.g., records).
  - The Pure Lisp had automatic storage

allocation (e.g., `cons`) but no verified garbage collector.

- The compilers, assembler, linker, and loader were "cross-platform" transformers.

- No useful application programs were verified.

- The operating system was not hosted on the FM9001.

What Was Learned?

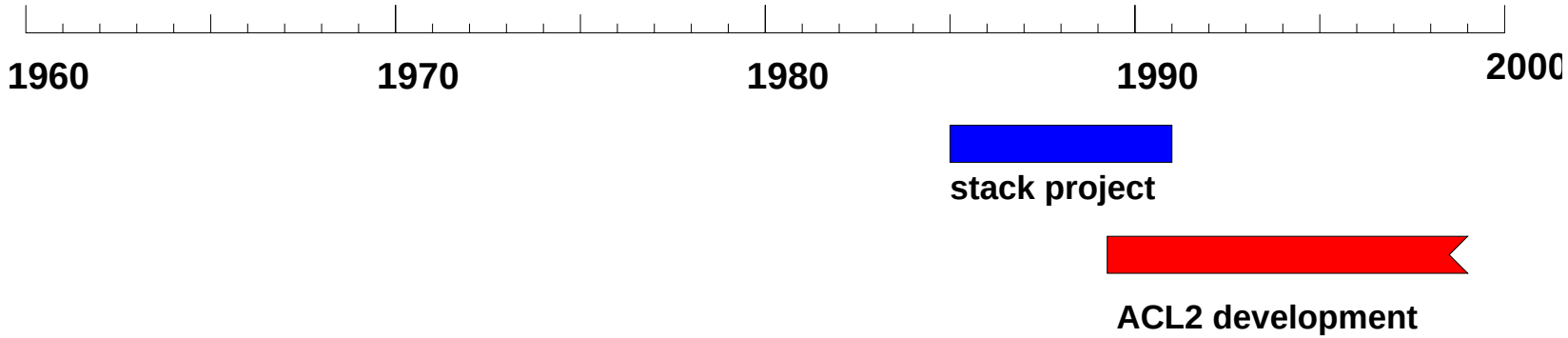
System verification does not require super-human skills.

It requires better tools than we had in 1985.

simple list processing

academic math and cs

commercial applications



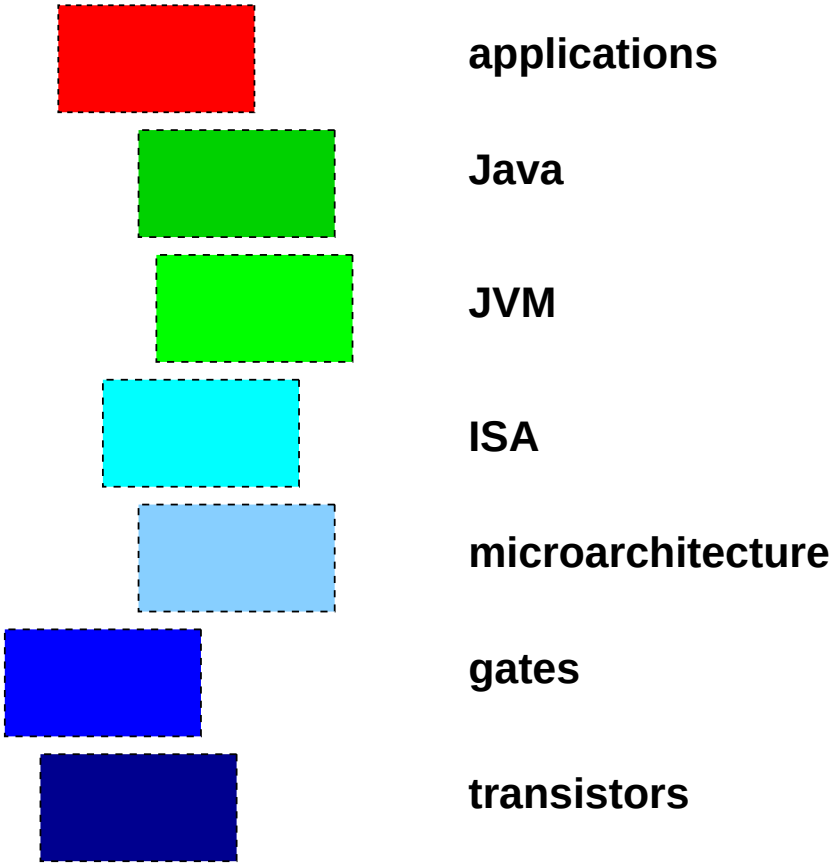
1990s

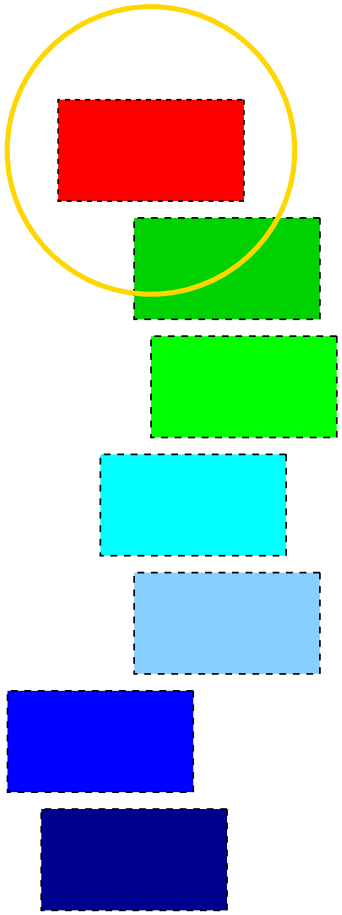
- Motorola 68020 and Berkeley C String Library
- Motorola CAP DSP
- AMD K5 FDIV
- AMD Athlon fp

- Rockwell Collins JEM1
- IBM Cryptographic coprocessor
- Union Switch & Signal and Argonne Nat'l Labs checkers
- FM9801
- JVM

- Java classes
- IBM Array Verification Tool
- Rockwell Collins CAPS

An Imaginary Stack





applications

Java

JVM

ISA

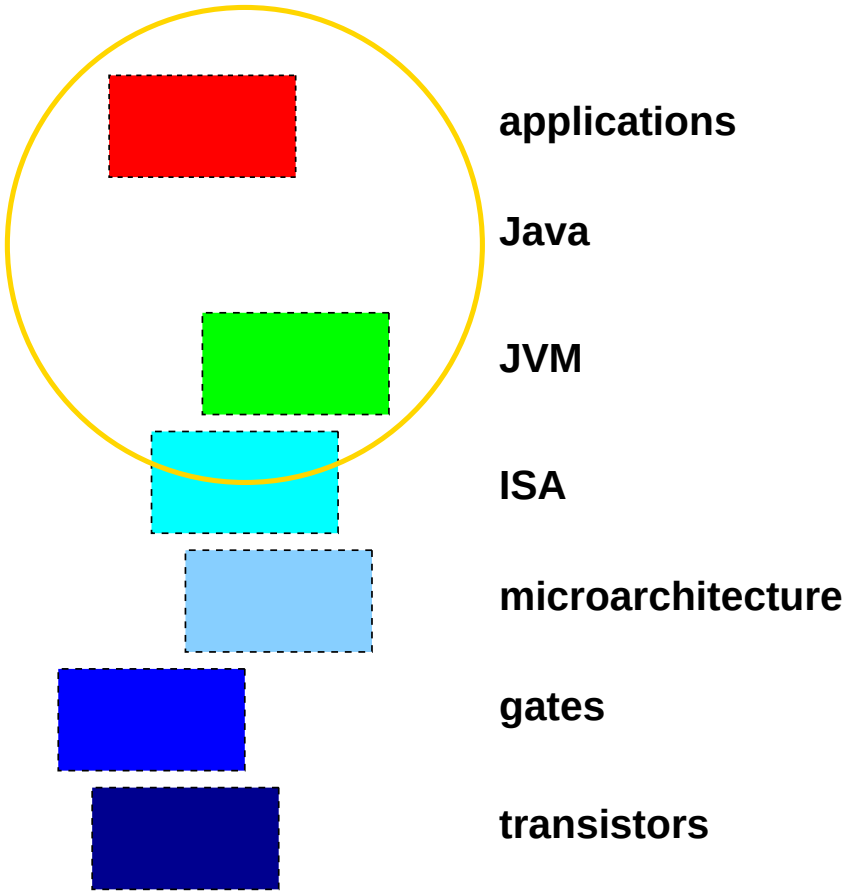
microarchitecture

gates

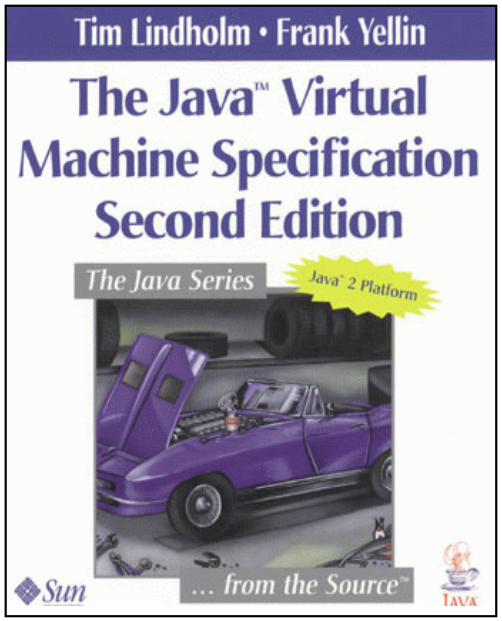
transistors

```
class Container {
    public int counter; }

class Job extends Thread {
    Container objref;
    public Job incr () {
        synchronized(objref) {
         objref.counter = objref.counter + 1; }
        return this; }
    public void setref(Container o) {
        objref = o; }
    public void run() {
        for (;;) {incr(); } } }
```

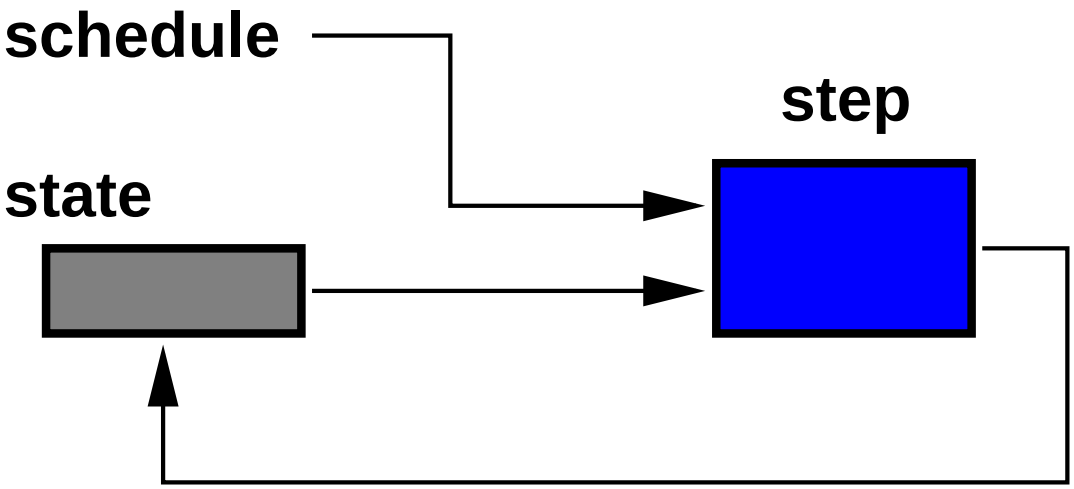
```
class Apprentice {
    public static void main(String[] args){
        Container container = new Container();
        for (;;) {Job job = new Job();
                  job.setref(container);
                  job.start(); } } }
```


applications
Java
JVM
ISA
microarchitecture
gates
transistors

Java (Golden, Krug, Liu, Moore, Porter)

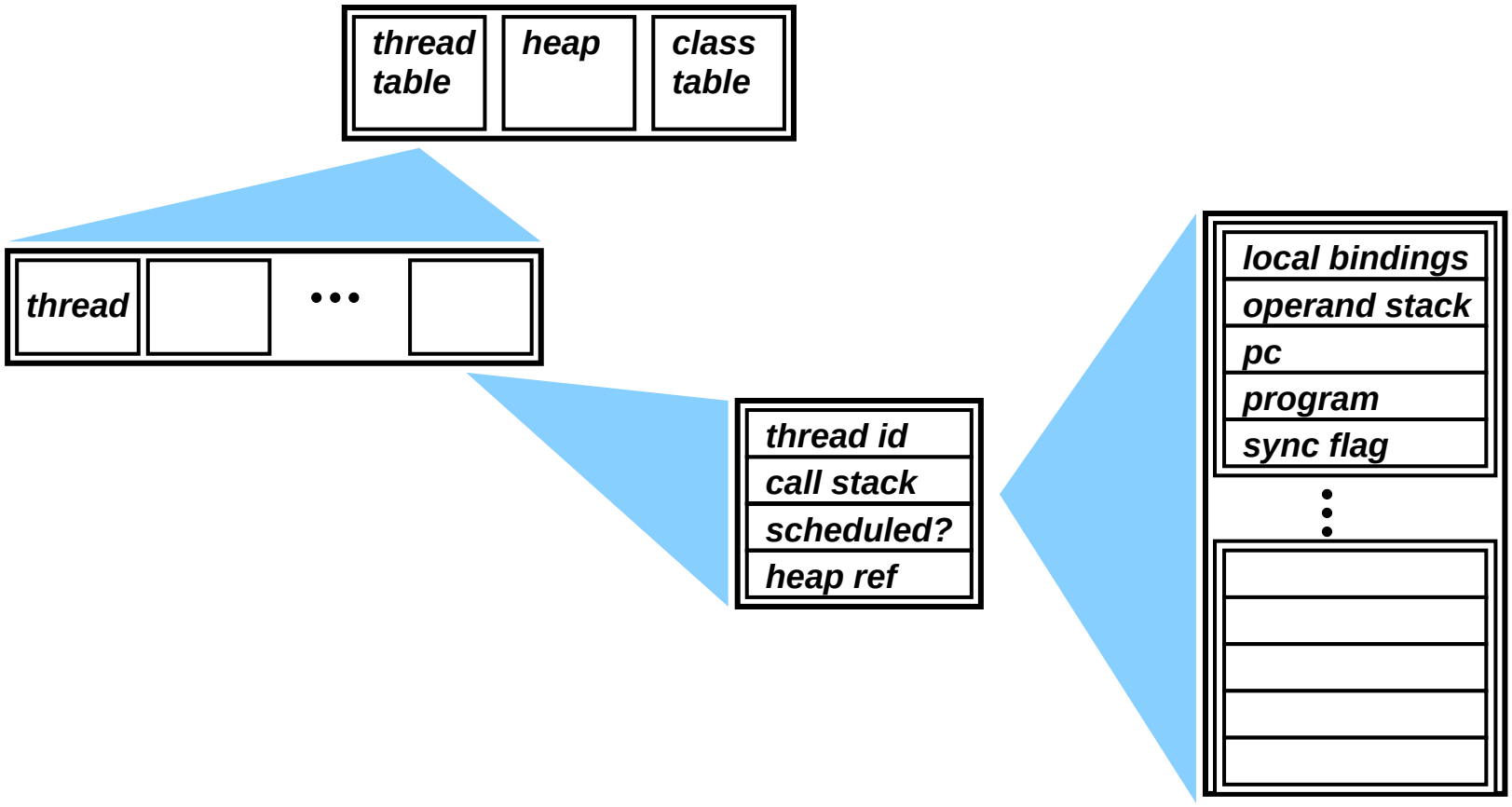


```
; JVM in ACL2
; J Moore and George Porter

(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))))
```




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

Our State: $< tt, hp, ct >$



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

```
(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))
  ...))
```

```
(defun execute-AALOAD (inst th s)
 (let* ((frame (top-frame th s))
        (index (top (stack frame)))
        (aref (top (pop (stack frame))))
        (array (deref aref (heap s))))
  (modify th s
   :pc (+ (inst-length inst) (pc frame))
   :stack
       (push (element-at index array)
             (pop (pop (stack frame)))))))
```

Some Java/JVM/ACL2 Code

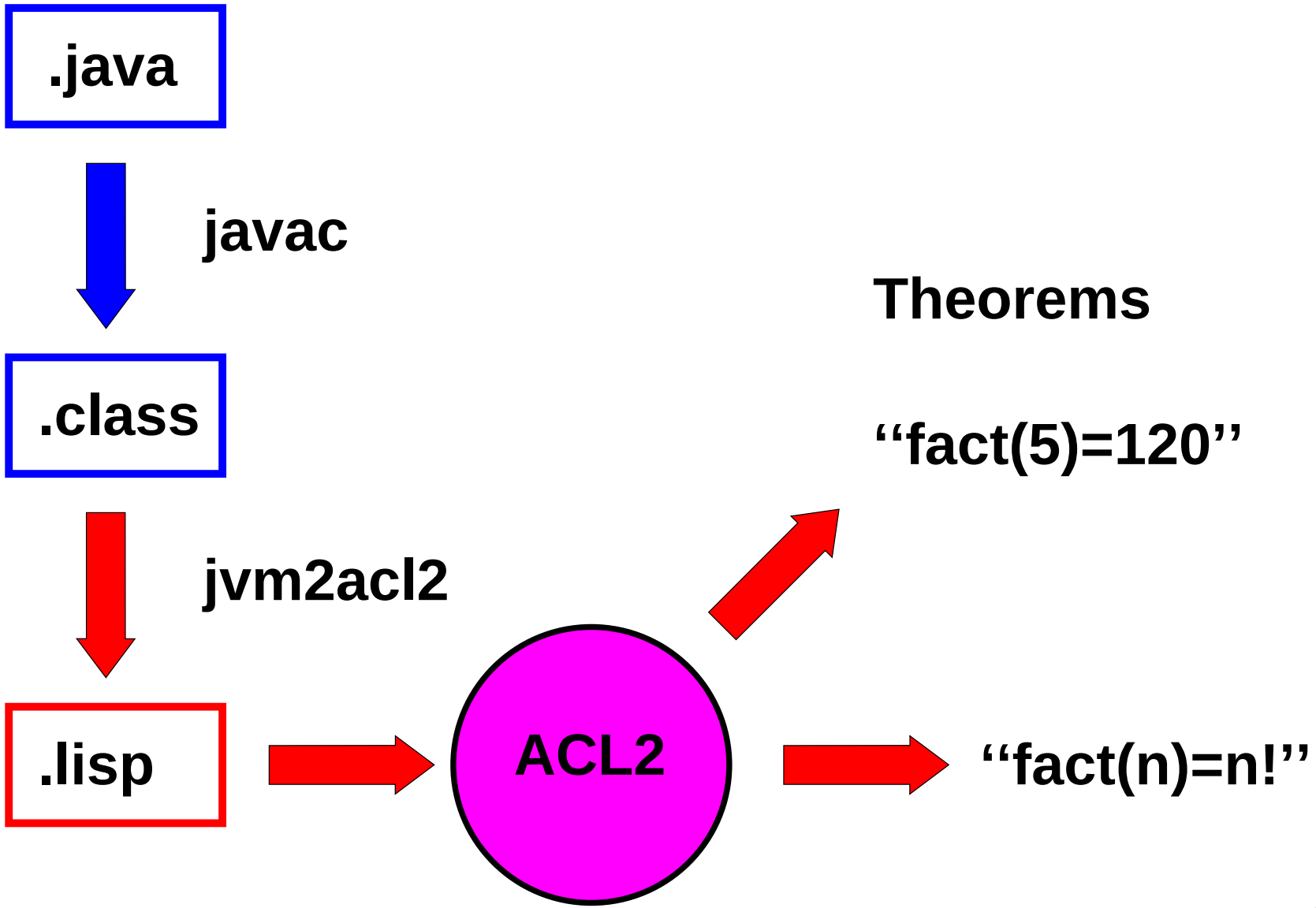
```
public static int fact(int n){
 if (n<=0) return 1;
  else return n*fact(n-1);}
```

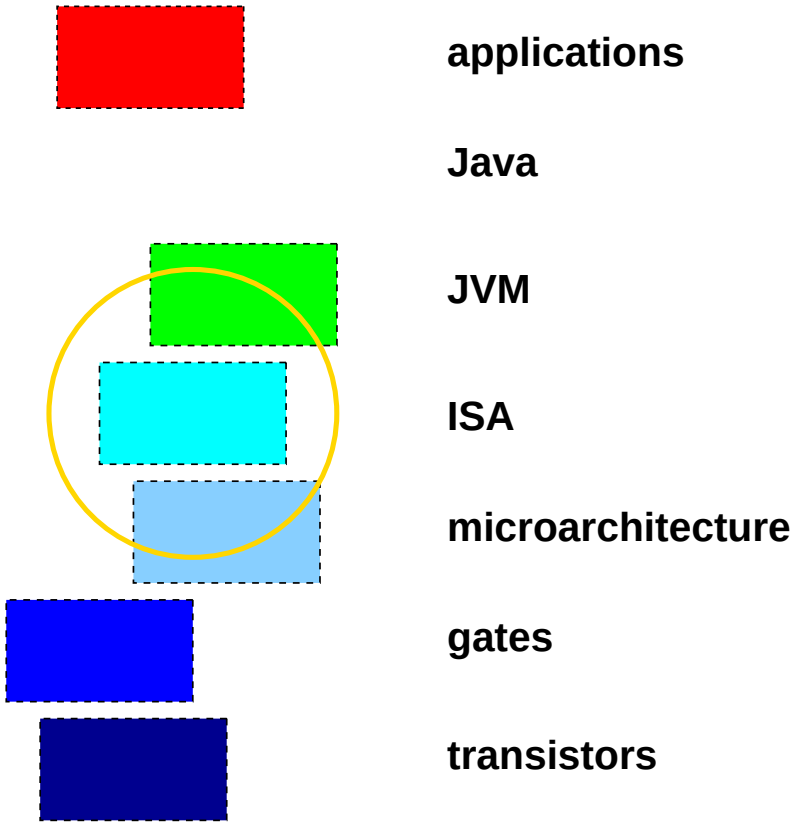
javac

```
Method int fact(int)
 0 iload_1
 1 ifgt 5
 4 iconst_1
```

jvm2acl2

```
("fact" (INT) nil
 (ILOAD_1)
 (IFGT 5)
 (ICONST_1)
```


.java
javac
.class
jvm2acl2
.lisp
ACL2
Theorems
“fact(5)=120”
“fact(n)=n!”


applications
Java
JVM
ISA
microarchitecture
gates
transistors

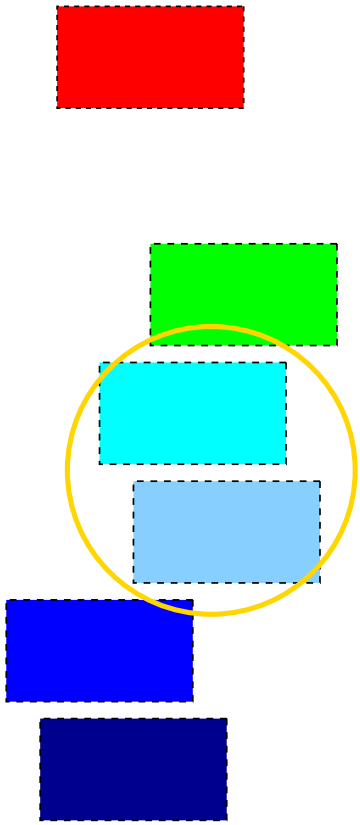
Rockwell-Collins / aJile Systems JEM1

The world's first silicon Java Virtual Machine was first modeled in ACL2.

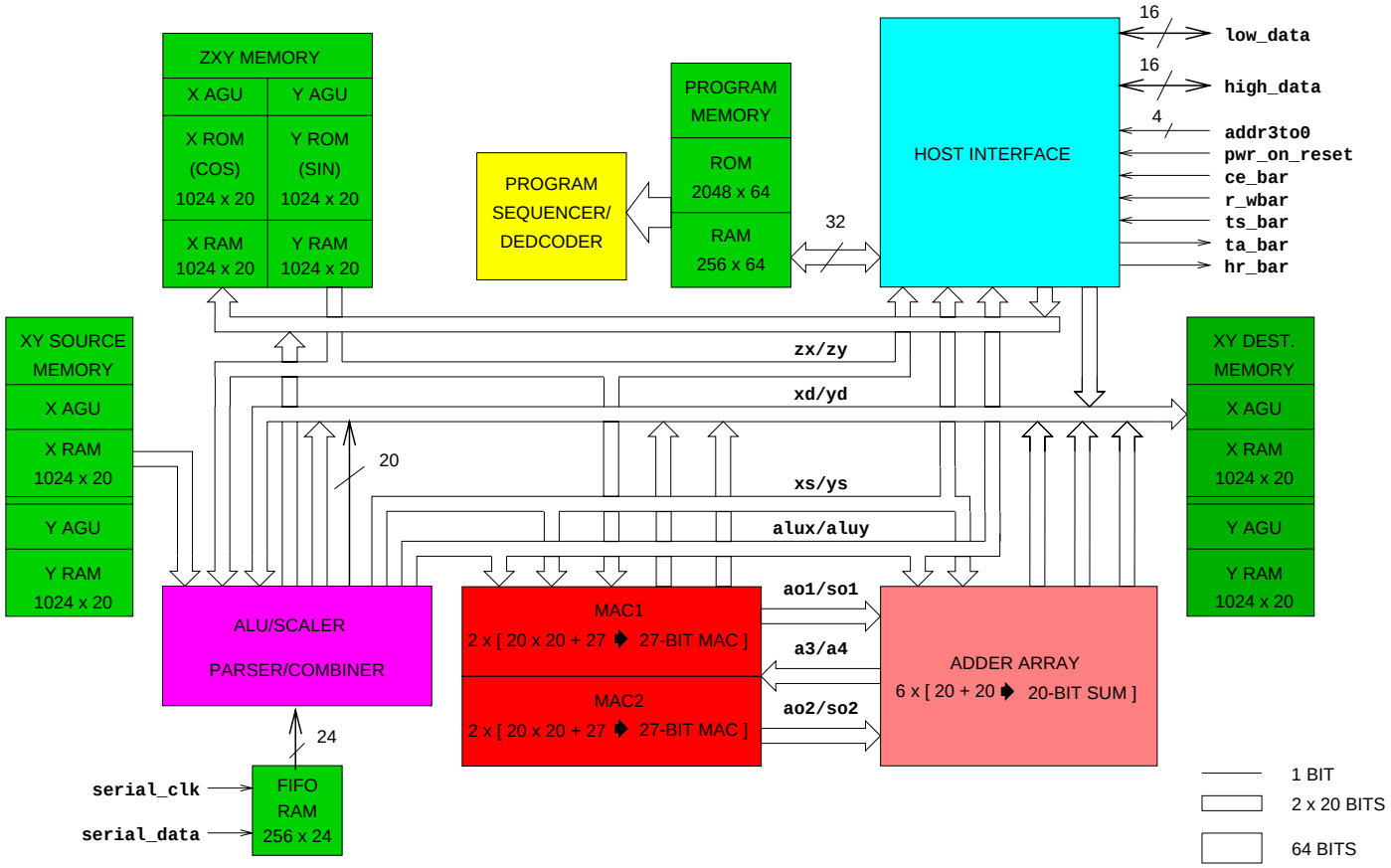
The formal model is executable.

It was used in place of a C simulator for requirements and certification testing.

It runs at 90% the speed of the C simulator.


applications
Java
JVM
ISA
microarchitecture
gates
transistors

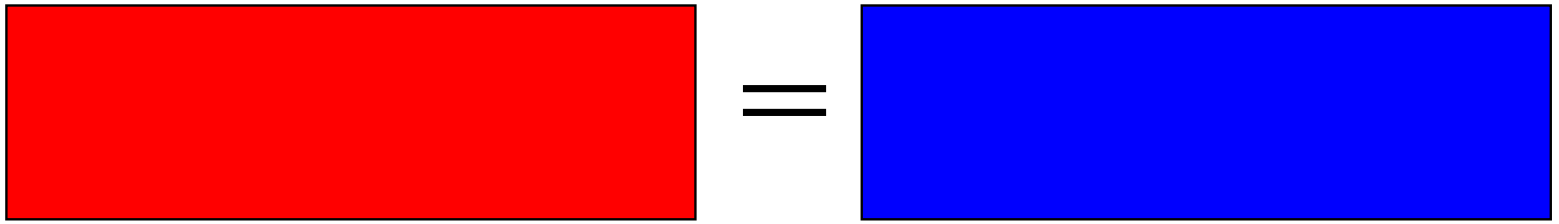
Motorola CAP DSP (Brock)



ROM containing 50 microcoded DSP algorithms

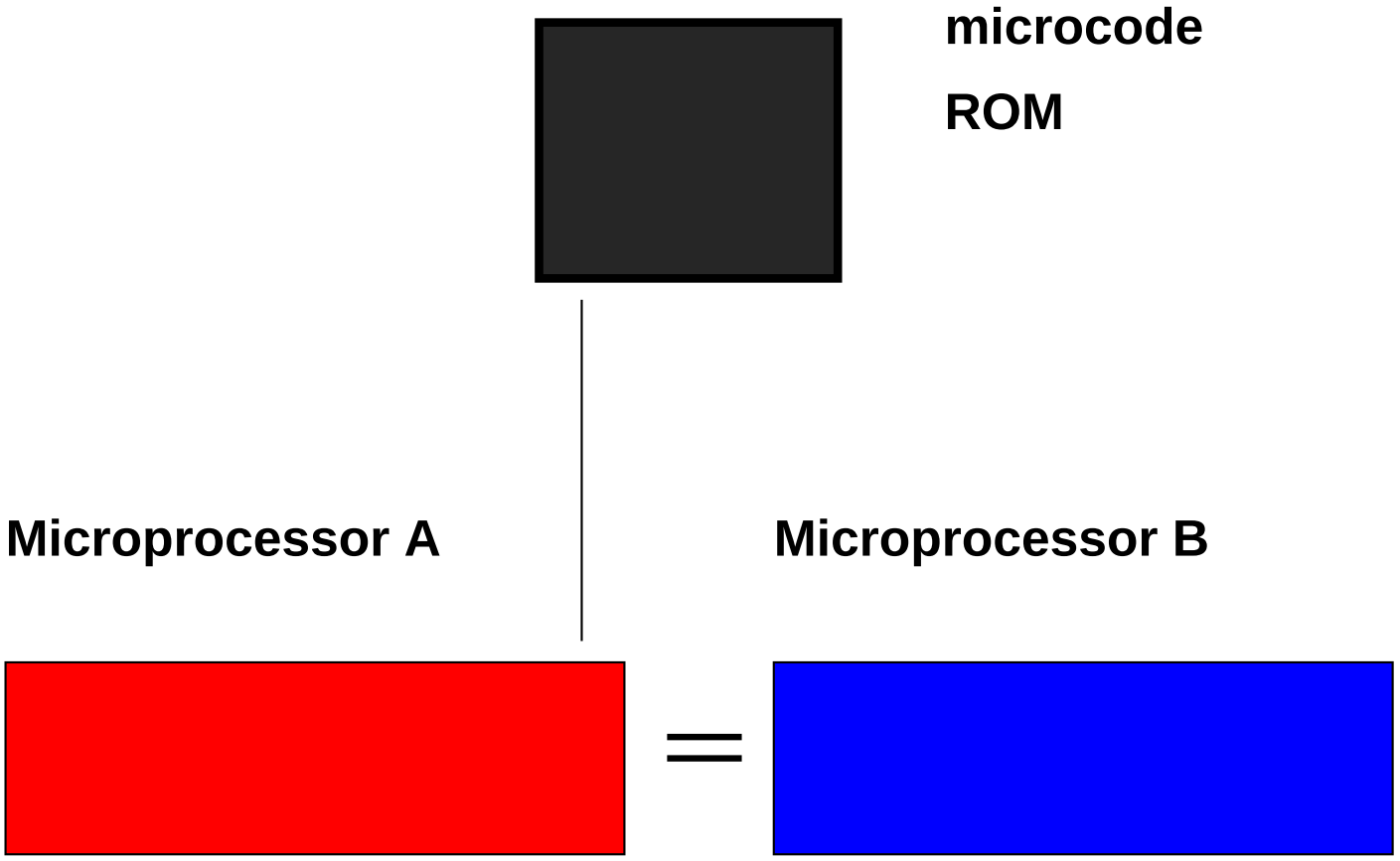
Pipelined microarchitecture

Sequential microcode ISA



(if no hazards)

Rockwell Collins Avionics



AMD K5 Algorithm FDIV$(p, d, mode)$

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]`
...	...	$=$ ...	...
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

```
      3 6.
+       -.1 7
+         .0 0 3 4
+       -.0 0 0 0 6 6
      ---------------
      3 5.8 3 3 3 3 4
   ___________________
1 2 | 4 3 0.0 0 0 0 0 0 0
      4 3 2.
      -----
        -2.
        -2.0 4
        ------
          .0 4
          .0 4 0 8
          ---------
        - .0 0 0 8
        - .0 0 0 7 9 2
        -------------
        - .0 0 0 0 0 8
```

Reciprocal Calculation:

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

Quotient Digit Calculation:

$0.083 \times 430.0000 = 35.6900000 \approx 36.000000 = q_0$

$0.083 \times \text{-}2.0000 = \text{-}.1660000 \approx \text{-}.170000 = q_1$

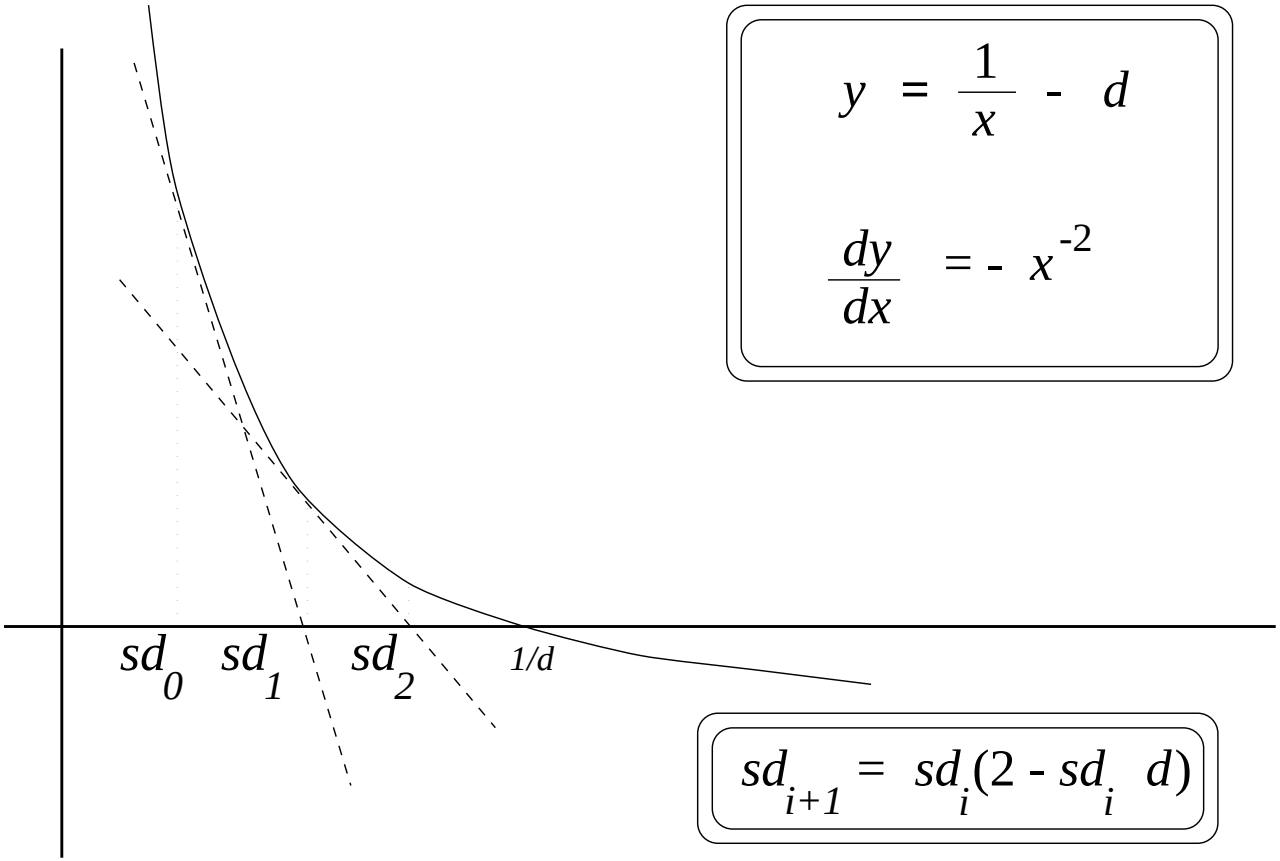
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

$0.083 \times \text{-}.0008 = \text{-}.0000664 \approx \text{-}.000067 = q_3$

Summation of Quotient Digits:

$q_0 + q_1 + q_2 + q_3 = 35.833333$

Computing the Reciprocal



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

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

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

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

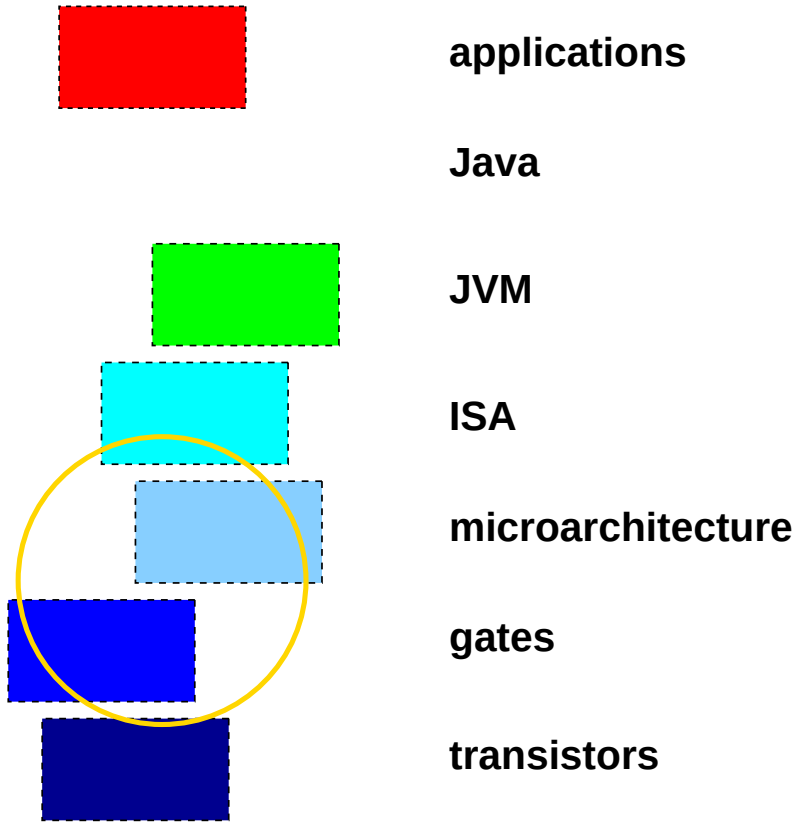
The Formal Model of the Code

```
(defun 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 ... fdiv)
       fdiv)))
```

The K5 FDIV Theorem

```
(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*)


applications
Java
JVM
ISA
microarchitecture
gates
transistors


AMD ATHLON™
ROCKETS TO 1 GHz
AMD
Athlon
PROCESSOR

All elementary floating-point operations on the **AMD Athlon** were

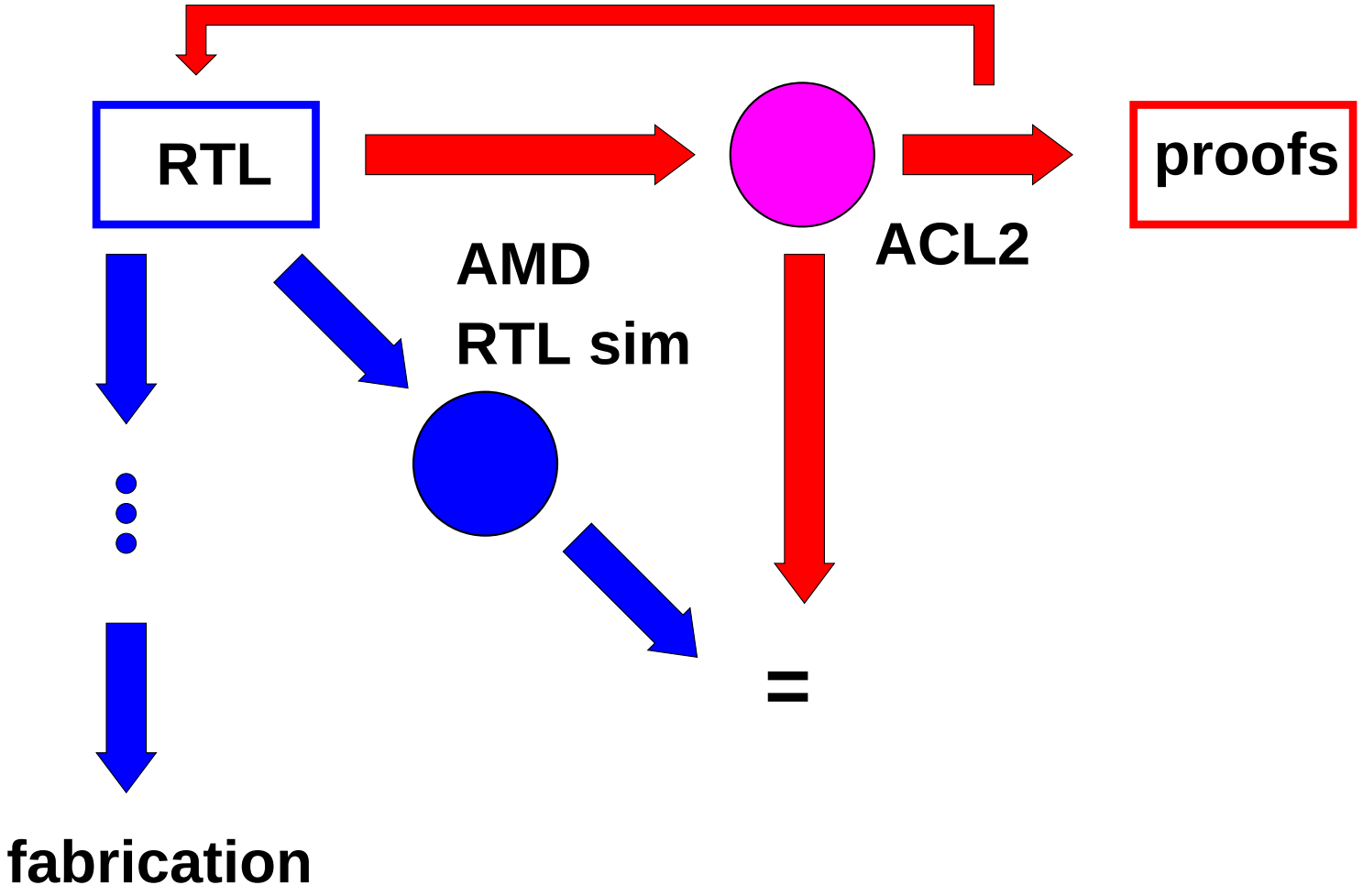
- specified in ACL2 to be IEEE compliant,
- proved to meet their specifications, and
- the proofs were checked mechanically – before fab.

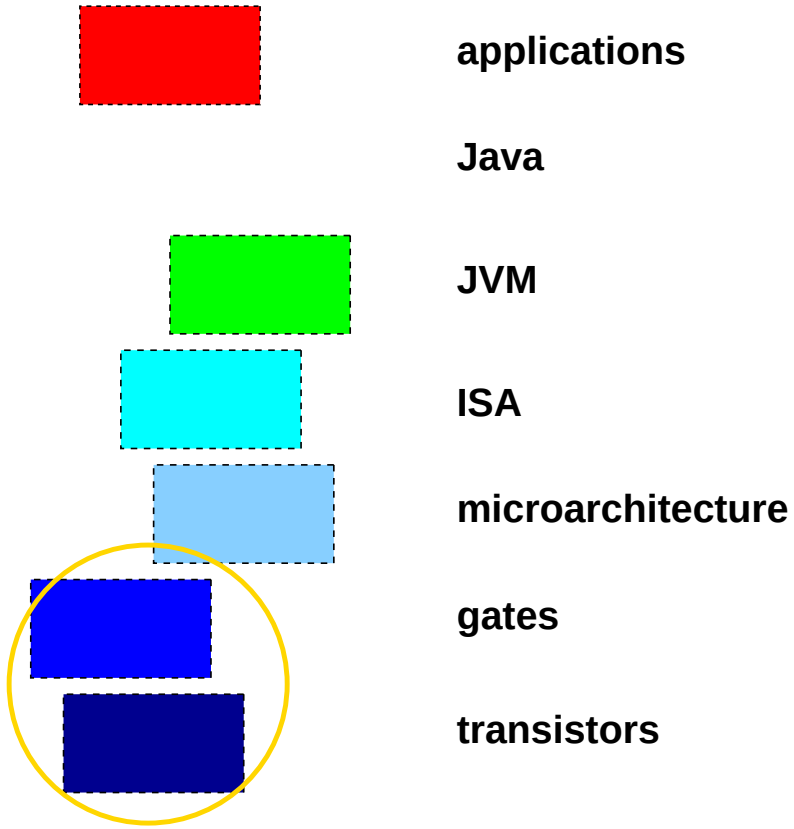
(Russinoff and Flatau)

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];
```


RTL
AMD
RTL sim
ACL2
proofs
=
fabrication


applications
Java
JVM
ISA
microarchitecture
gates
transistors

Summary of Our Current State

We have mechanically verified

- RTL for industrial designs,
- commercial microcode, and
- simple programs in a widely used programming language.

In each industrial site where ACL2 is being used, proprietary formal information has been created:

- models of artifacts,
- definition of "in-house" concepts,
- specifications, and
- lemmas and proof strategies.

Our lowest level formal models are usually

- "bit accurate,"
- "cycle accurate," and
- efficiently executable (comparable to C).

Our formal models often replace conventional simulation models:

- they are as accurate and
- they run about as fast, but
- they can be formally analyzed.

Formal models add value.

The evidence supports

- the value-added proposition,
- the claim that formal methods requires less manpower than testing, and
- the claim that formal methods reduces time-to-market.

We have certified books (lemma libraries) for

- arithmetic and bit-vectors,
- stuttering bi-simulation,
- a BDD package (60% CUDD speed), and
- a sound and complete mu-calculus model checker.

The Challenge

Each major group in the formal methods community should design and mechanically verify a practical embedded system, from transistors to software.

Why Build a Stack?

- There is still much to be learned.
- Practice makes perfect (more accurately, practice encourages mechanization).
- A stack forces you to deliver what you assume.

The Rules of the Game

- The project should produce an artifact, e.g., a chip.
- The artifact's behavior should be of interest to people not in formal methods.
- The artifact should come with a "warranty" expressed as a theorem.

- The warranty should be certified mechanically; user input and interaction are allowed but a machine must be responsible for the soundness of the claim.
- The machine used to certify the warranty should be available for others, at least, to use and test, if not inspect.

Why It is Hard

There are unsolved technical problems.

But mainly:

None of our systems allow us to move up and down the abstraction hierarchy with the ease required to build a practical system.

The Challenge Will Encourage

- more automation,
- integration of formal approaches,
- development of collaborative environments,

- identification of assumptions,
- specification of crucial system interfaces, and
- elegant designs of practical systems.