Little (but Hard) Theorems About Big Systems: Some Case Studies

J Strother Moore
Department of Computer Sciences
University of Texas at Austin

Marktoberdorf Summer School 2004

Lecture 6

The Plan

- Trivial Hardware
- AMD Hardware
- Other Commercial Applications
- Conclusion

Trivial Hardware Demo

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.

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]
 ...  ...  $= \ldots$                                 ...
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.  $fdiv = qq_1 + q_0$                              $mode$
```

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
        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:

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

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

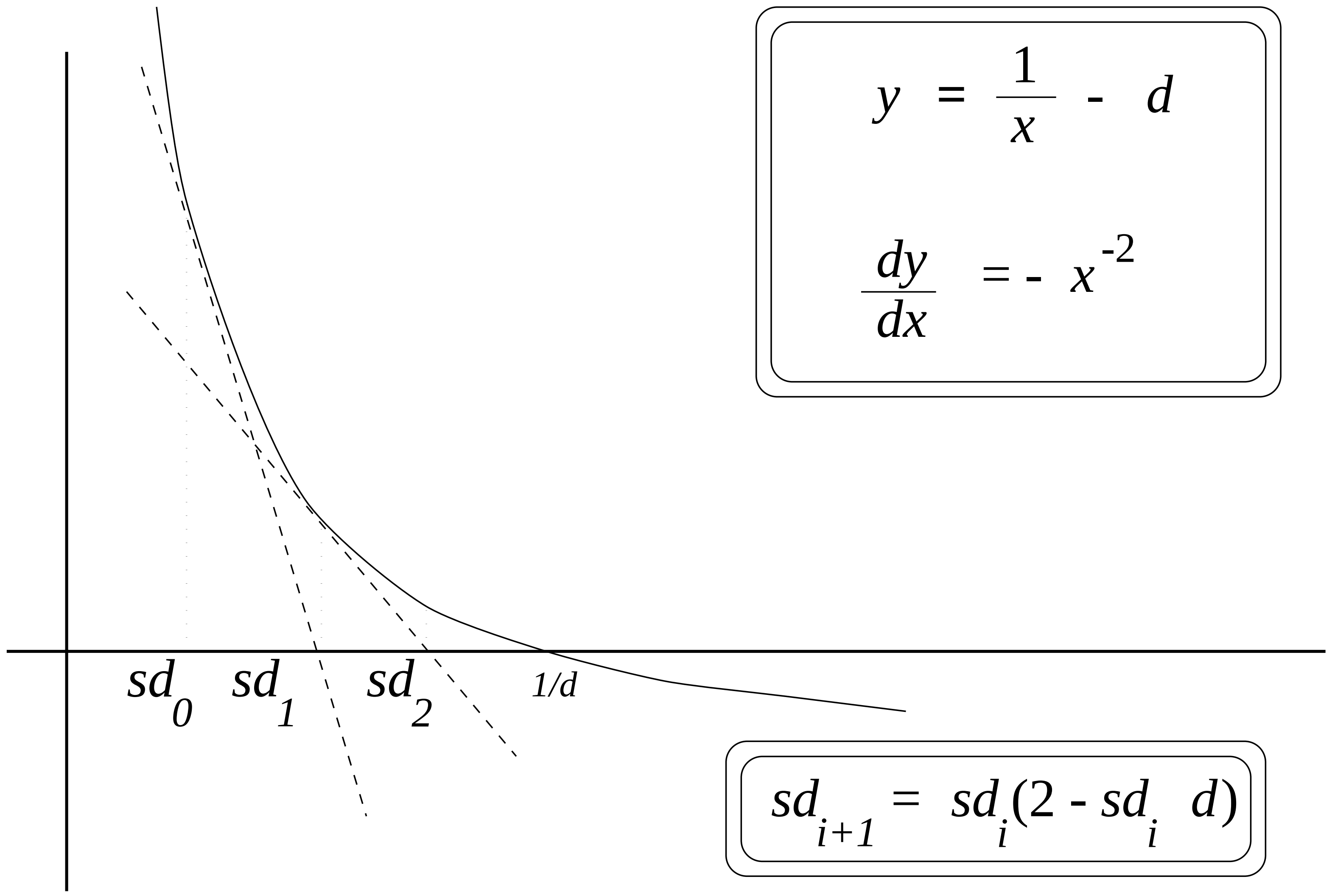
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

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

Summation of Quotient Digits:

$q_0 + q_1 + q_2 + q_3 = 35.833333$

Computing the Reciprocal



$$\boxed{y = \frac{1}{x} - d \qquad \frac{dy}{dx} = -x^{-2}}$$

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

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 K5 FDIV

```
(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 (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*)

AMD Athlon 1997

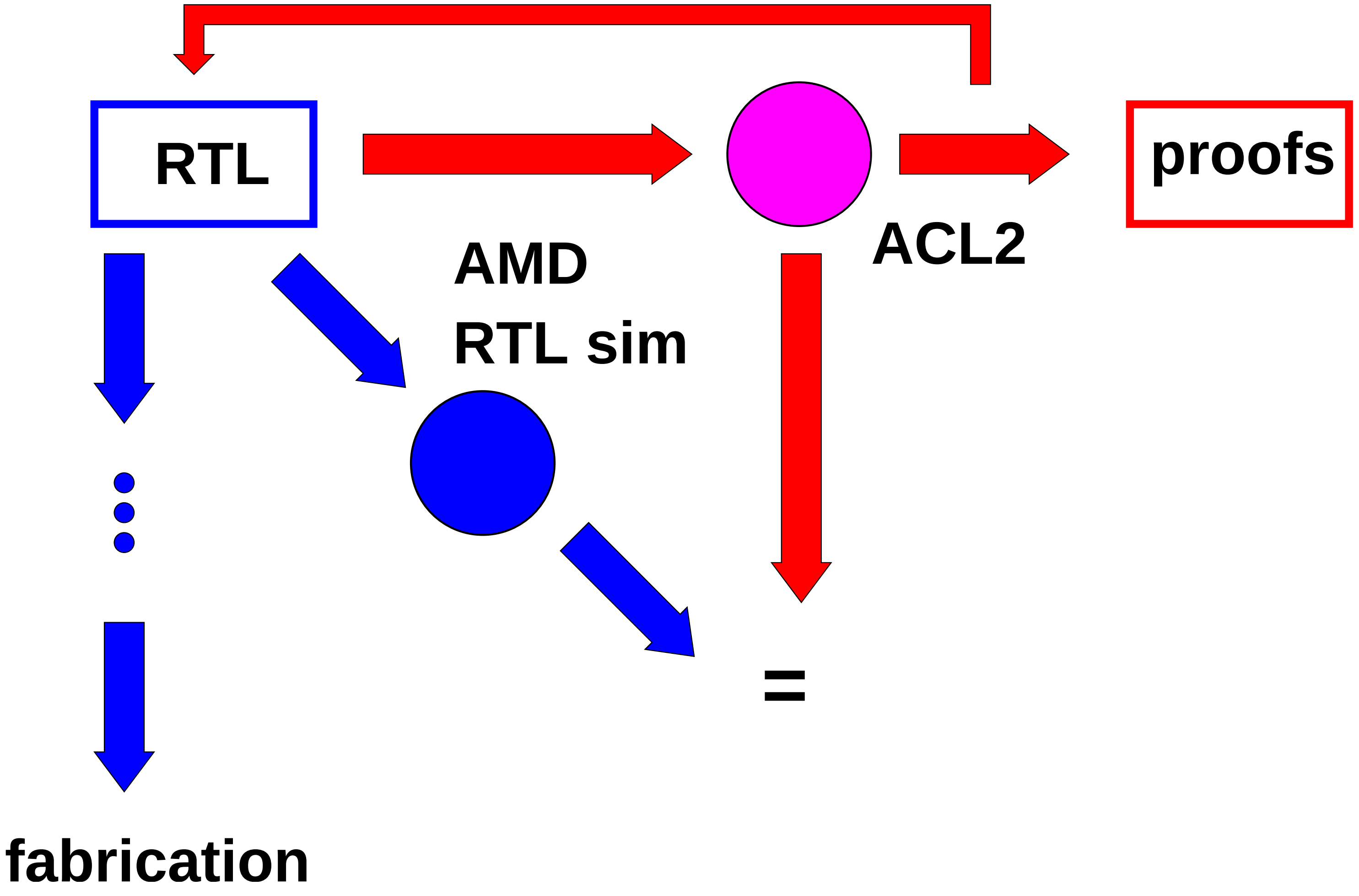
All elementary floating-point operations,
`FADD`, `FSUB`, `FMUL`, `FDIV`, and `FSQRT`,
on the AMD Athlon were

- 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];
```


RTL
AMD
RTL sim
ACL2
proofs
=
fabrication

Other Work at AMD

AMD is experimenting (struggling) with ACL2 to reason about the bus unit on a future AMD microprocessor.

The unit consists of 1 million lines of RTL, which translates to 1.5 million lines of ACL2.

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.

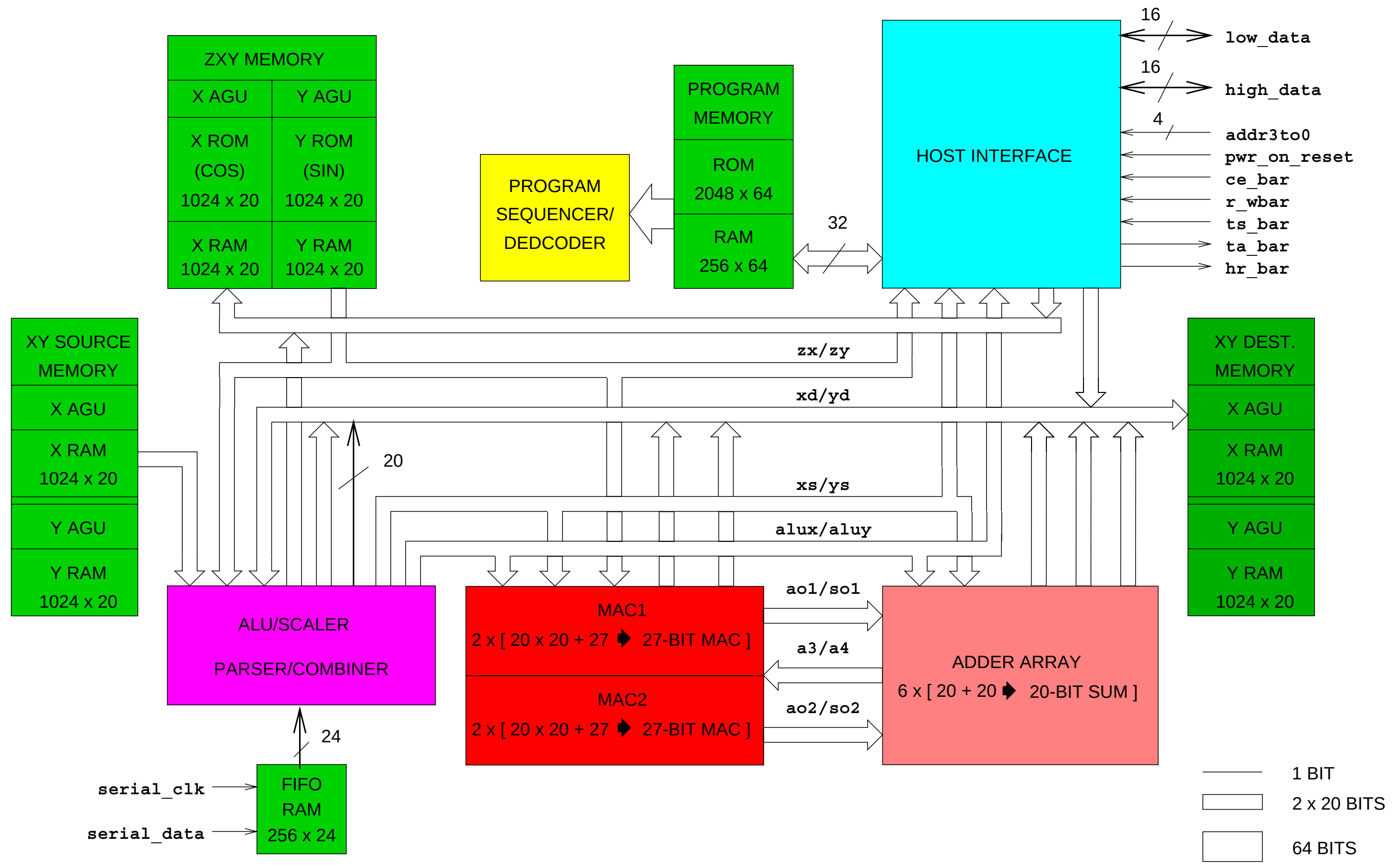
New bugs (which were undetected after simulation) have been found and fixed before tapeout.

Commercial Applications of ACL2

- JVM bytecode programs (sequential and multi-threaded) (Moore)
- JVM bytecode verifier (Liu)
- AMD modules for K5, Athlon, Opteron, etc. (Russinoff, Sumners, Kaufmann, Flatau)
- IBM Power 4 `FDIV` and `FSQRT` (Sawada)

- Motorola CAP DSP (Brock, Hunt, Moore)
- Rockwell Collins microarchitectural equivalence (Greve, Wilding)
- Rockwell Collins / aJile Systems JEM1 (Hardin, Greve, Wilding)
- Rockwell Collins AAMP-7 separation kernel (Greve, Wilding)

Motorola CAP DSP



Features of CAP Microarchitecture

separate program and data memories

252 programmer-visible data and control registers

6 independently addressable data and parameter memories

data memories are logically partitioned into ‘source’ and ‘destination’ memories; the sense of the memories may be switched under program control

the arithmetic unit includes 4 multiplier-accumulators and a 6-adder array

64-bit instruction word, which in the arithmetic unit is further decoded into a 317-bit, low-level control word

instruction set includes no-overhead looping constructs; as many as 10 different registers are involved in the determination of the next program counter

a single instruction can simultaneously modify over 100 registers

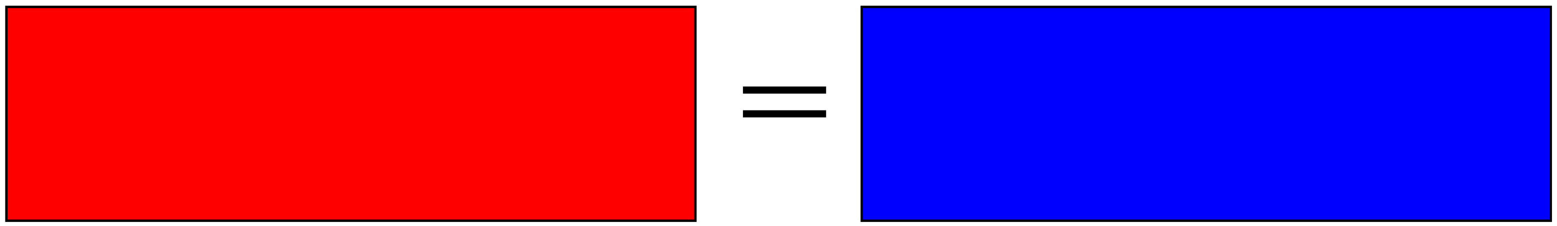
the 3-stage instruction pipeline contains many hazards visible to the programmer.

A Hint of the Main Theorem

ROM containing
50 microcoded
DSP algorithms

Pipelined
microarchitecture

Sequential
microcode ISA



(if no hazards)

Summary and Conclusion

We've seen proofs at many levels in the system hierarchy

- basic recursive functions (`app`, `rev`)
- arithmetic (AMD FPU)
- algorithms (e.g., `qsort`)
- hardware description languages (`eval-net`, AMD RTL)

- processor architecture (CAP DSP, et al)
- machine/assembly code (`m0`, `m6` – the JVM)
- microcode (CAP DSP, Rockwell AAMP-7)
- Java (via `javac` and bytecode proofs)
- operating system/separation kernel (Rockwell AAMP7)

- safety and progress properties of multi-threaded Java classes

On Hardware versus Software

Every ACL2 proof is an example of software verification.

E.g., one view of our JVM work: We are proving theorems about 10,000 lines of Lisp.

Whether you *like* Lisp or not you must admit: it is a widely used, commercially-supported, ANSI standard programming language.

On Lisp as a Logic

If you think that these projects would be easier in another logic, try it!

On Why ACL2 is Successful

Its logic is a functional programming language:

- clean semantics
- often adequate expressive power
- executable (dual-use) models

- industrial strength
- directly supported by other tools (Emacs, many GUI debuggers, profilers, etc).

Do not lightly dismiss these advantages.

We use Lisp as both an object logic and a meta-language.

Having a formally supported meta-language is very powerful.

We support proofs with a high degree of automation.

We make the user guide the system by posing theorems.

This is hard to learn and at first makes you feel powerless.

But it has big advantages when you want to do big proofs.

On Theorem Proving versus Other Techniques

My personal goal is to make it possible, with an acceptable amount of input from the user, to prove mechanically the important properties of practical hardware and software.

We can ignore complexity only when that complexity is irrelevant to the properties of interest.

Many of the other lectures are about about automatic techniques for proving certain kinds of properties.

The “ultimate general-purpose verification system” will have to employ those techniques, as needed.

But that system will also have to face number theory, sequences, trees, recursion and induction.

Those basic problems will not go away, regardless of improvements to logics and programming languages.

So there is plenty of work to do!

On What Remains

There are two hard problems in theorem proving and you have encountered them both in the simple exercises:

- finding the right formula to prove, and
- inventing the right concepts and lemmas to decompose the proof.

Acknowledgments

Little of what you have seen would have been possible without the creative work of many colleagues and students over three decades.

Thank you for giving me the chance to tell you about some of it.

J Strother Moore
Marktoberdorf, 2004