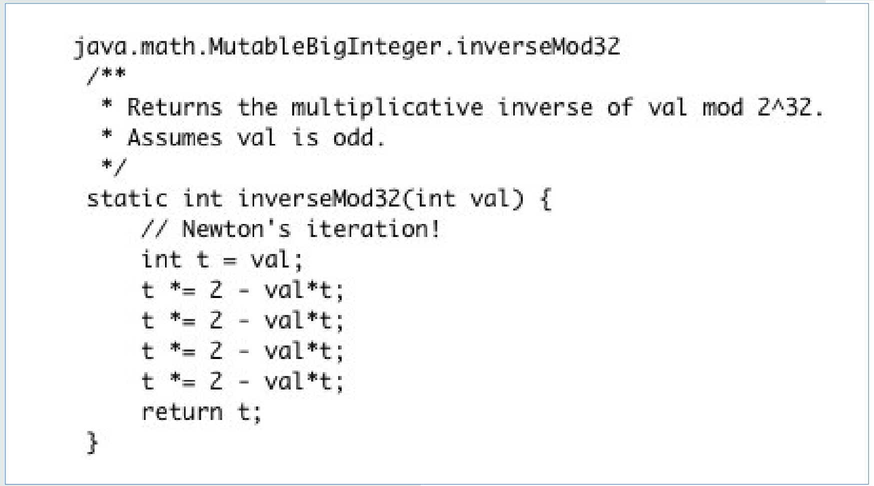
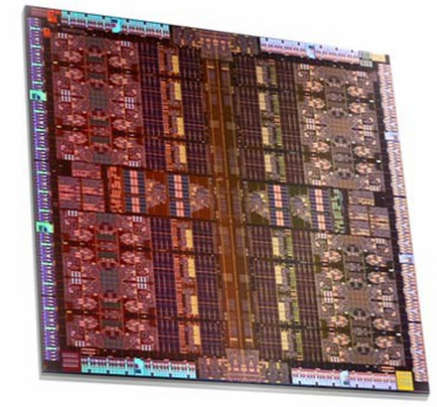
Theorem Proving with ACL2 for Industry Artifacts

Dmitry Nadezhin
Oracle Labs

June 14, 2016



```
java.math.MutableBigInteger.inverseMod32
 /**
  * Returns the multiplicative inverse of val mod 2^32.
  * Assumes val is odd.
  */
 static int inverseMod32(int val) {
     // Newton's iteration!
     int t = val;
     t *= 2 - val*t;
     t *= 2 - val*t;
     t *= 2 - val*t;
     t *= 2 - val*t;
     return t;
 }
```



Demand for Formal Verification

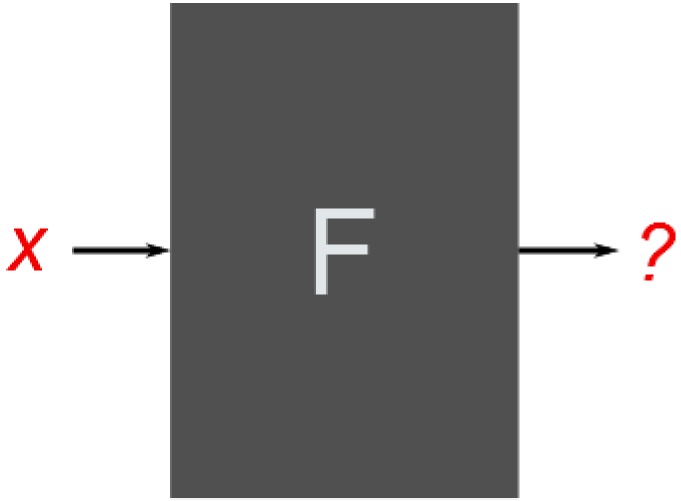
- *Use formal verification to increase confidence in correctness of Oracle designs*
  - Reduce number of bugs that escape to silicon

- Project started in summer 2013 after request for help from SPARC™ Architect
  - “Catching errors is getting harder”
  - “Fixing errors is becoming costlier”
    - E.g., Intel 1995 Pentium Fdiv bug resulted in a quarterly statement charge of $500M
    - Each extra tape-out, due to undetected bugs, costs $$$ and time to market

Introducing Formal Verification

- Formal Verification: rigorous and automated analysis that demonstrates that an implementation satisfies its specification for all inputs
- Main technique: symbolic simulation
  - Simulation of the circuit using symbolic inputs instead of concrete values
  - F(0) = 5, F(1) = 8, F(2) = 11, F(3) = 14, …

    vs
  - F(x) = 3*x + 5 for all 64-bit unsigned numbers x
- Two uses of symbolic simulation:
  - Model Checking
  - Theorem Proving

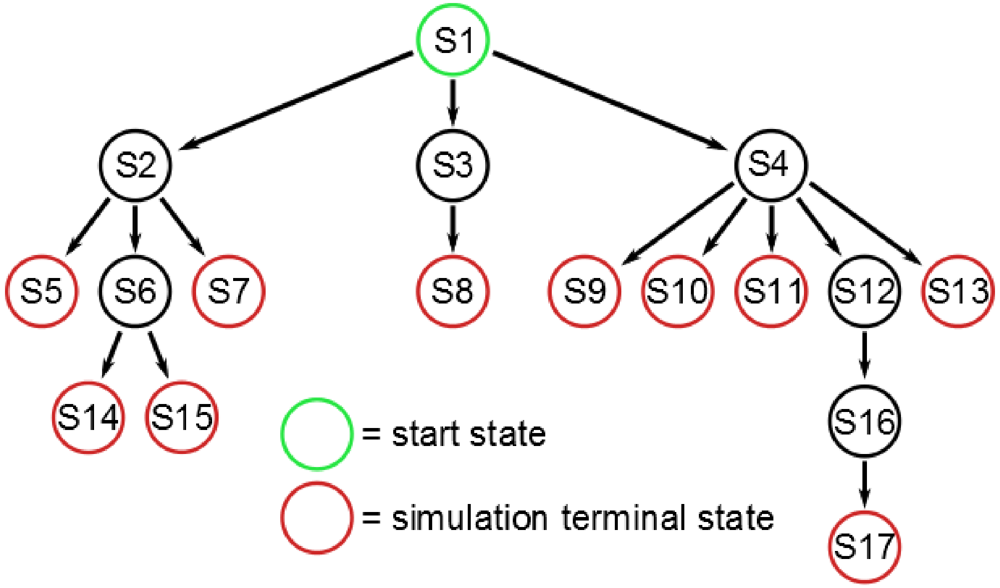


Agenda

- Why Formal Verification at Oracle?
- ACL2 basics
- Hardware verification
- Software verification

Introducing Model Checking

- Model Checking: stepping a design from one set of states to the next set of possible states, checking that user-provided properties always hold…
  … until you visit all states or run out of time
- Applications: coherency protocols, distributed algorithms
  - Typical example: “this buffer never overflows”
- Pros: automatic
- Cons: limited scalability



Model Checking vs Theorem Proving

- Oracle uses model checking for proving properties with modest state space
- Model checking is insufficient for verification of units with complex data path
- Oracle has a theorem proving group which is collaboration between Oracle Labs and Microelectronics
- We use ACL2 as our main tool

Why ACL2 ?

- ACL2 Prover
  - Programming language written in subset of Lisp
  - Theorem prover written in ACL2
    - Proof engine used at AMD, IBM, Centaur, Motorola, Intel
    - 2005 ACM Software System Award
  - Maintained at Univ. of Texas with help from community
- ACL2 Books (~5500)
  - A “book” is a library of functions and lemmas
    - Arithmetic, bitops, RTL, proof and definition utilities
  - Includes a Verilog parser and hardware symbolic simulator
- Support Tools: SAT solvers, waveform viewer
- Robert Boyer, J Moore, then Matt Kaufmann
- http://www.cs.utexas.edu/~moore/acl2/

ACL2 Basics

- Lisp data types
- Programming
- Logic
- Proving
- Theorems become rules

Lisp Data Types - atoms

- Integers: 5, -3, #x100
- Rationals: 1/2
- Complex rationals: #c(1 2)
- Characters: #\A
- Strings: “Hello”
- Symbols: NIL, T, +, IF, FOO, X

Lisp Data Types - conses

- (x . y)
- Example: ((1 . #\A) . (“Hello” . NIL))
- List
  - Example: (A . (B . (C . NIL)))
  - Abbreviated as (A B C)
- Association list
  - Example: ((A . 1) . ((B . 2) . ((C . 3) . NIL)))
  - Abbreviated as ((A . 1) (B . 2) (C . 3))

Programming

- `(+ 2 5)`
- `(defun sqr(x) (* x x))`
- `(sqr 5)`
-
  ```
  (defun sum1(n)
     (declare (xargs :measure (if (zp n) 0 n)))
     (if (zp n) 0 (+ n (sum1 (- n 1)))))
  ```
- `(sum1 100)`

Reasoning Using Rewriting

- `(sqr x)  ==>  (* x x)`

- (defrule square-of-sum
```
(defrule square-of-sum
    (equal (sqr (+ a b))
           (+ (sqr a) (* 2 a b) (sqr b))))
```

- `(sqr (+ a b))   ==> (+ (sqr a) (* 2 a b) (sqr b))`

- `(in-theory (disable sqr))`

- `:use (:Instance sqr (x (+ a b)))`

Induction

-
```
(defruled sum1-thm
    (implies (natp n)
             (equal (sum1 n)
                    (* 1/2 n (+ n 1)))
:enable sum1
:induct (sum1 n))
```

- Proof obligations generated by :induct:

```
  (IMPLIES (AND (NOT (ZP N)) (:P (+ -1 N)))
                (:P N))
  (IMPLIES (ZP N) (:P N)))
```

Using Lemmas

-
  ```
  (defun sum2 (i n0)
     (declare (xargs :measure (if (zp i) 0 i)))
     (if (zp i) 0 (+ (+ 1 n0 (- i)) (sum2 (- i 1) n0))))
  ```

-
  ```
  (defruled sum2-as-sum1-lemma
      (implies (and (natp i) (natp n) (<= i n))
                    (equal (sum2 i n)
                           (- (sum1 n) (sum1 (- n i))))))
  ```

-
  ```
  (defruled sum2-as-sum1
      (equal (sum2 n n) (sum1 n))
      :use (:instance sum2-as-sum1-lemma (i n))
  ```

Ordinals < ε_0

- The ordinals less than ε_0 can be represented by finite rooted trees.
- $\omega^p m + n$, where m is positive integer, p and n are ordinals
- (make-ord (p m n) ((p . m) . n)))
- $\omega m + n \leftarrow$ (make-ord 1 m n)
- (defun ack (m n)

```
  (declare (xargs :measure (make-ord 1 (+ (nfix m) 1) (nfix n))))
  (if (zp m)
      (+ n 1)
      (if (zp n)
          (ack (- m 1) 1)
          (ack (- m 1) (ack m (- n 1))))))
```

First-Order Classic Logic

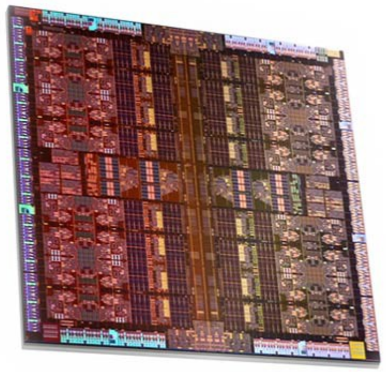
- (defruled excluded-middle (or be (not be)))

-
  ```
  (defun-sk exists-twin-prime (n)
    (exists x
            (and (integerp x)
                 (> x n)
                 (primep x)
                 (primep (+ x 2)))))
  ```

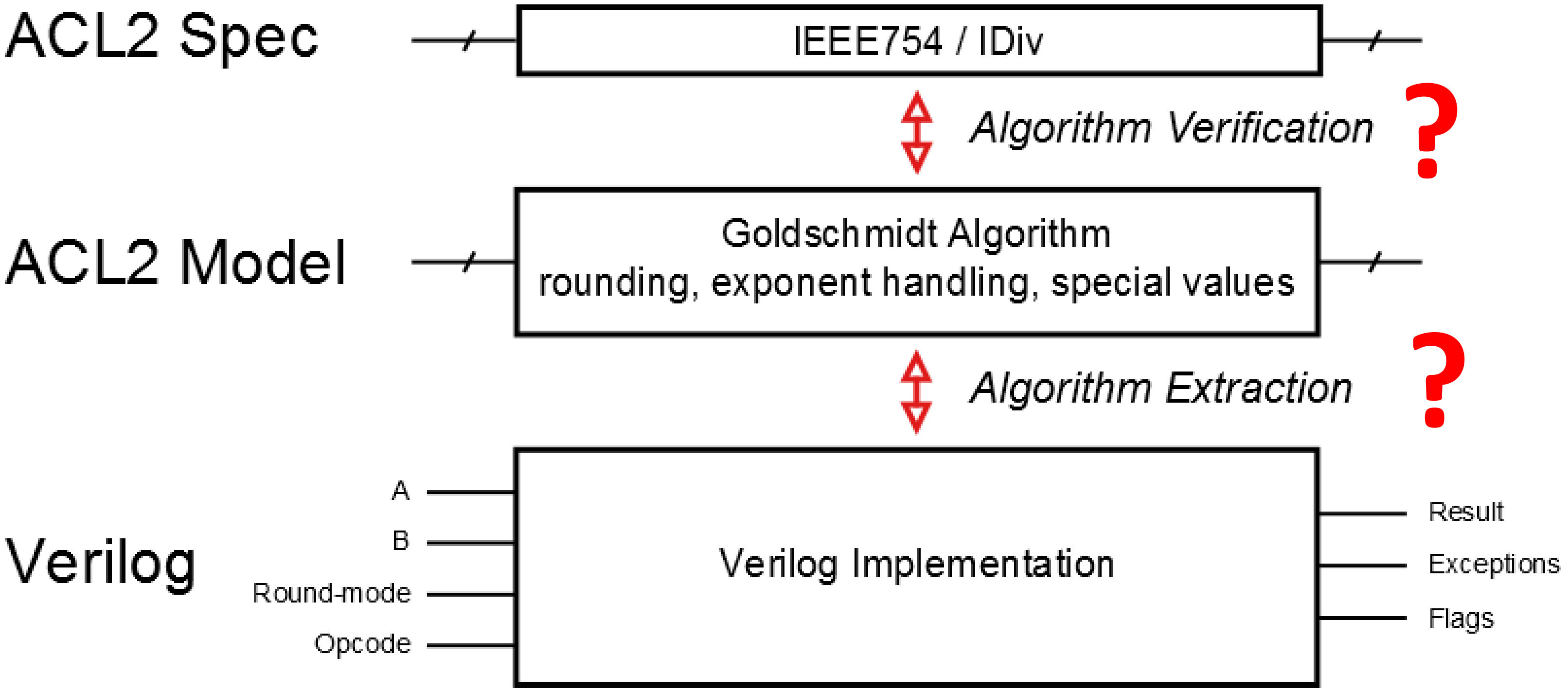
-
  ```
  (defun-sk twin-primes-infinite ()
     (forall n (exists-twin-prime n)))
  ```

Formal Verification of Divide and Square Root Circuits

- New implementations on SPARC^{TM} core
- 32/64-bit floating-point division and square root
  - fdivd
  - fdivs
  - fsqrtd
  - fsqrts
- 32/64-bit integer divide
  - udivx
  - sdivx
  - udiv
  - sdiv

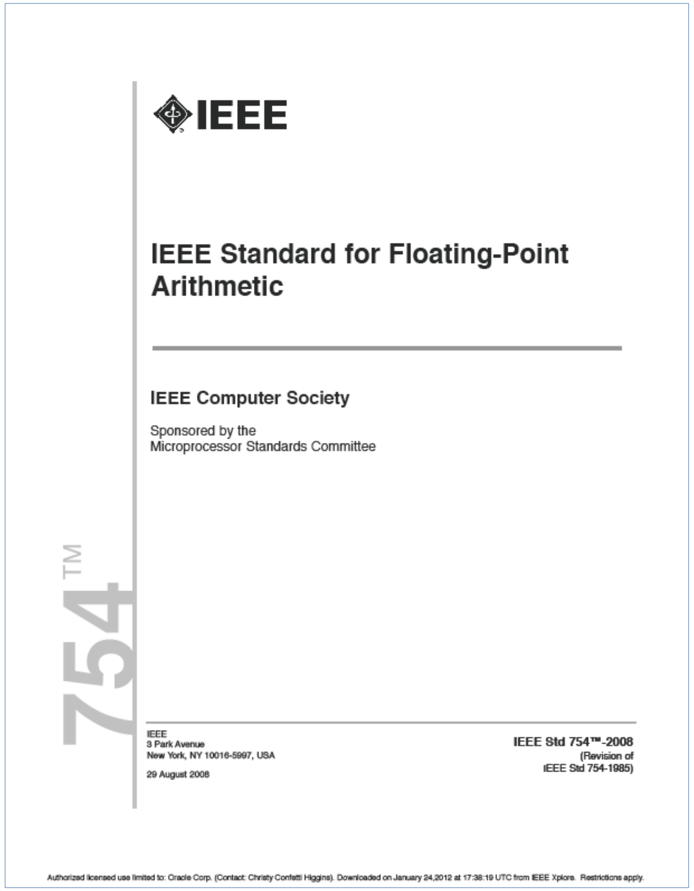


Our Proof Goal and Strategy



Specification

- IEEE754 Standard on Floating-Point Arithmetic
  - 80-page document written in English
  - Our ACL2 specification includes
    - *div*, *sqrt*, *add*, *mul*, and fused *mul-add*
    - all special values (+/- 0, +/-Infinity, NaNs)
    - all exception flags
    - denormals
    - four rounding modes
    - customization for NaN values
- Validated specifications against 9.5M test vectors from Oracle’s test suite



IEEE

754™

IEEE Standard for Floating-Point Arithmetic

IEEE Computer Society

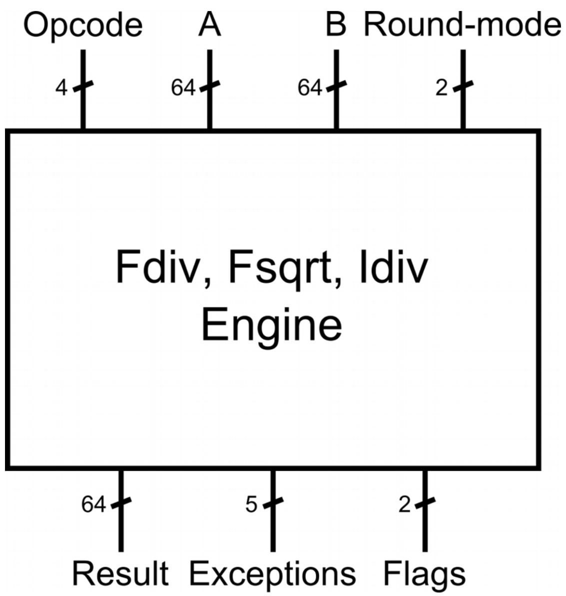
Sponsored by the
Microprocessor Standards Committee

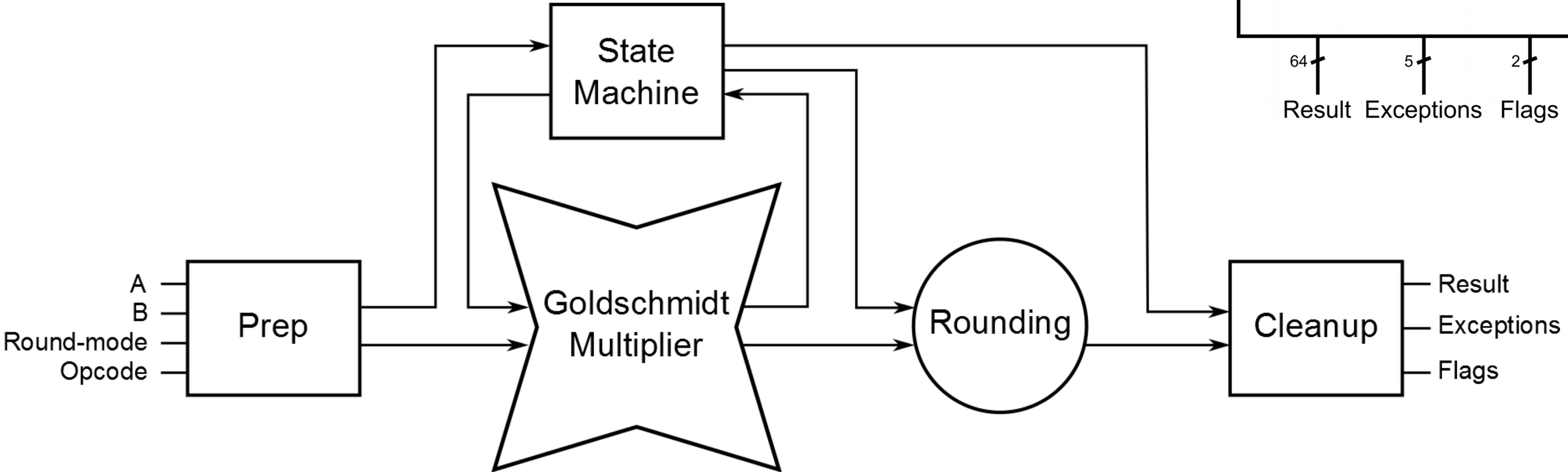
IEEE
3 Park Avenue
New York, NY 10016-5997, USA

29 August 2008

IEEE Std 754™-2008
(Revision of
IEEE Std 754-1985)

Verilog Implementation





ACL2 Model – Code List

- Code list has some primary inputs
- Code list is a sequence of instructions
- Each instruction computes new value by applying an operation to operands
- Each operand is either primary input or result of a previous instruction
- Example:
  - Inputs: in_0, in_1, in_2
  - $x_0 = in_0 * in_1$
  - $x_1 = x_0 + in_2$
  - $x_2 = x_0 – in_2$
  - $x_3 = x_1 * x_2$
- No loops. Limited branching: selection among results of a few code lists

Encoding Code Lists in ACL2

- inp is a list of primary inputs
- Selection function for each primary input
- Each instruction is a function of inp

- inp is (list in0 in1 in2)
- (defun in0 (inp) (nth 0 inp))
- (defun in1 (inp) (nth 1 inp))
- (defun in2 (inp) (nth 2 inp))
- (defun x0 (inp) (* (in0 inp) (in1 inp)))
- (defun x1 (inp) (+ (x0 inp) (in2 inp)))
- (defun x2 (inp) (- (x0 inp) (in2 inp)))
- (defun x3 (inp) (/ (x1 inp) (x2 inp)))

Code List of Bit-Vectors

- Bit vector of n bits is represented in ACL2 by a natural $0 <= bv < 2^n$
- Arithmetic operations +,-,*
- Operation part-select selects subvector from bit vector
- (part-select :high 63 :low 32 x)
- floor $((x \bmod 2^{64}) * 2^{-32})$

- Example: multiplier 32 x 32 → 32
  - (part-select :high 63 :low 32 (* x y))

Algorithm Extraction

- Use hardware-related ACL2 tools developed by ACL2 community
  - Verilog parsing - VL
  - Symbolic simulation – STV (Symbolic Trajectory Evaluation)
    - Control signals are concrete
    - Data signals are symbolic

The Goldschmidt Division Algorithm

- Input: A in [1,2), B in [1,2)
- Output: approximation of A/B
- T = table_lookup(B)
- $d_0 = B * T;$
- $n_0 = A * T;$
- **for** (**int** i = 0; i < MAX; i++) {
  /** **invariant** n_i/d_i == A/B d_i --> 1 */
- $r_i = 2 – d_i;$
- $d_{i+1} = d_i * r_i;$
- $n_{i+1} = n_i * r_i;$
- }
- **return** $n_{MAX};$

$$\frac{A}{B} = \frac{A \times T \times r_0 \times r_1 \times r_2 \; ..}{B \times T \times r_0 \times r_1 \times r_2 \; ..} \longrightarrow \frac{Q}{1}$$

Error Analysis of Goldschmidt Algorithm

- Error analysis is a crucial part of complete proof
  - If error in computed approximation is “small enough,” then the rounding step will return the correct IEEE 754 result
- Precise error analysis provides opportunity for improvement
  - Error analysis may permit optimization of the lookup tables, and thereby reduction of chip area or power consumption or latency

Error Analysis

- T from table lookup is an approximation for 1/B
- u is the negation of relative error in T: $u = (1/B - T)/(1/B) = 1 - B*T$
- $d_0 = B*T = 1-u$
- $n_0 = A*T = A*T$
- $r_0 = 2 – d_0 = 1+u$
- $d_1 = d_0*r_0 = 1 – u^2$
- $n_1 = n_0*r_0 = A*T*(1+u)$
- $r_1 = 2 – d_1 = 1 + u^2$
- $n_2 = n_1*r_1 = A*T*(1+u+u^2+u^3)$
- $A/B = A*T /(1-u) = A*T*(1+u+u^2+u^3+u^4+u^5+ \dots)$
- $error_2 = n_2 – A/B = A*T*(-u^4-u^5-\dots)$

Error Analysis and Finite Hardware Precision

- Fixed-point operations, each multiplication result is truncated from 2M bits to M bits
- Each rounding error ed_i, en_i is in interval $(-2^{-M},0]$
- $d_0 = B*T = 1-u + ed_0$
- $n_0 = A*T = A*T + en_0$
- $r_0 = 2 – d_0 = 1+u - ed_0 + er_0$
- $d_1 = d_0*r_0 = 1 – u^2 + (1-u)*(-ed_0+er_0) + (1+u)*ed_0+ed_0*(-ed_0+er_0) + ed_1$
- $n_1 = n_0*r_0 = A*T*(1+u) + A*T*(-ed_0+er_0)+(1+u)*en_0+en_0*(-ed_0+er_0) + en_1$
- . . .
- Total-error $= n_2 – A/B = A*T*(-u^4-u^5-…) + …$
- Make canonical multivariate polynomial for total error above (exactly)
- Evaluate it in interval arithmetic

Multivariate Polynomials

- Fixed list of variables: u, A, T, en_0, ed_0 …
- Polynomial is represented by a list of terms
- Term is a product of a rational coefficient and a monomial
  - Example: 3/7 * u^2*A*T
  - Represented as ((2 1 1 0 0) . 3/7)
- Operations on polynomials: +, scale, -, *
- Point evaluation of polynomial at point vector
- Interval evaluation of polynomial at interval vector
- Theorems:
  - Point evaluation of a sum is a sum of point evaluations
  - If point vector is in interval vector, then point evaluation is in interval evaluation

Global Error Bounds

- T = table_lookup(B)
- table_lookup is a step function.
  - table_lookup(B) = T_i when B in $[B_i, B_{i+1})$
- Relative error in T is given by u: u = 1 – B*T
- u in $(1 - B_{i+1}{*}T_i,\ 1 - B_i{*}T_i]$ when B in $[B_i, B_{i+1})$
- Do interval evaluation of error polynomial for each segment $[B_i, B_{i+1})$

- First we coded this in Java using interval library JInterval
- Error bounds were inside tolerance, though table segments were too small
- We suggested smaller table with larger segments and still good error bounds
- Designers accepted the table temporarily, we continued ACL2 proofs
- Finally ACL2 proofs confirmed error bounds

Verification and Improvements

- We proved correctness of computation of significands using the Goldschmidt algorithm
- We also proved correctness of rounding, exponent handling, exception flags
- In summary we proved that the ACL2 model satisfies the IEEE 754 specification
  - ACL2 model = Floating-point divide implementation
  - ACL2 model = Floating-point square root implementation
- Furthermore:
  - The formal verification resulted in significant reduction of lookup tables
  - Formal verification effort also resulted in simplification of square root implementation and its proof

Formal Verification of JDK methods

- Java or JVM ?
- Which methods ?
- JVM models in ACL2
- A small method
- Transcendental functions

Java or JVM ?

- Should we trust Java compiler ?
- Multiple languages: Java, Scala, Kotlin, Jython, Ruby
- Classes generated on the fly

- JVM class files

Which Methods ?

- Easy specification, difficult proof
- Math methods
- java.math.BigInteger
- java.lang.Math
- java.lang.StrictMath

JVM Models in ACL2

- Defensive Java Virtual Machine - Richard M. Cohen 1997
- http://www.computationallogic.com/software/djvm/
- JVM M5 - J Strother Moore and George Porter
- https://github.com/acl2/acl2/blob/master/books/models/jvm/m5/m5.lisp
- JVM M6 – Hanbing Liu
- https://github.com/haliu/M6

- Floating-point instructions are not implemented in any of them
- Choose M5 because it is in official ACL2 repository

Small Method java.math.MutableBigInteger.inverseMod32

```
/**
 * Returns the multiplicative inverse of val mod 2^32.  Assumes val is odd.
 */
static int inverseMod32(int val) {
    // Newton's iteration!
    int t = val;
    t *= 2 - val*t;
    t *= 2 - val*t;
    t *= 2 - val*t;
    t *= 2 - val*t;
    return t;
}
```

Specification of inverseMod32 in Terms of JVM M5

- To prove that result after execution of inverseMod32 by JVM
  - Using thread th, starting in state s, and odd input value val
  - (val * result) **mod** 2^{32} = 1

-
  ```
  (defrule |inverseMod32 correct|
    (implies
      (and (poised-to-invoke-inverseMod32 th s val)
             (integerp val) (oddp val))
      (equal (int-fix (* val (top (stack (top-frame th (run (repeat th 37) s))))))
               1)))
  ```

result after execution on JVM

Proof of inverseMod32

- Define $\text{defect}_i = (1 – \text{val} * t_i)\ \mathbf{mod}\ 2^{32}$
- $\text{defect}_0 = (1 – \text{val} * \text{val})\ \mathbf{mod}\ 2^{32}$
- $\text{defect}_0\ \mathbf{mod}\ 2^3 = 0$
- $\text{defect}_{i+1} = (1 – \text{val} * t_{i+1})\ \mathbf{mod}\ 2^{32} = (1 – \text{val} * t_i * (2 – \text{val} * t_i))\ \mathbf{mod}\ 2^{32}$
  $= (1 – 2 * \text{val} * t_i + (\text{val} * t_i)^2)\ \mathbf{mod}\ 2^{32} = \text{defect}_i^{\,2}\ \mathbf{mod}\ 2^{32}$
- $\text{defect}_1\ \mathbf{mod}\ 2^6 = 0$
- $\text{defect}_2\ \mathbf{mod}\ 2^{12} = 0$
- $\text{defect}_3\ \mathbf{mod}\ 2^{24} = 0$
- $\text{defect}_4 = 0$

Transcendental Functions in JDK

- Portable – sin(x) returns the same result on all platforms
- William Kahan coined the term “The table maker's dilemma” for the unknown cost of rounding transcendental functions
- sin(x) in [l,u], where l and u are adjacent floating-point numbers
- Which of l and u must the method sin(x) return?
- Correct rounding says “nearest” - too costly, JDK declines this
- java.lang.Math says “any if them” - not portable
- java.lang.StrictMath says “the same as C library Fdlibm 5.3” - portable though a little arbitrary

What Is the Meaning of Fdlibm Functions

- C code
- Compilation to LLVM
- Compilation to specific ISA like X64

- Parse C by libclang and write FdlibmTranslit.java
- Compile C to llvm
- Compile FdlibmTranslit.java to FdlibmTranslit.class
- Prove equivalence of LLVM and FdlibmTranslit.class

- Designers write Fdlibm.java manually
- Prove equivalence of FdlibmTranslit.class and Fdlibm.class

Conversion of libclang Tree to FdlibmTranslit.java

- A few Java helper methods
- static int[] __AMP(double x) - view double as a pair of 32-bit integers
- static double __HI(double x, int high)
- static int compareUnsigned(int x, int y)

- Libclang tree contains types. It is easy to write tree patterns which modify code
- (ui >> 16)   → (ui >>> 16)
- (ui > 0x100) → Integer.compareUnsigned(x, 0x100) > 0
- *(((int *) (& d)) + 1)   → _AMP(d)[1]
- Lab: S1; S2; if (p) goto Lab; → Lab: for(;;) { S1; S2; if (p) continue Lab; break; }

Prove Equivalence of Fdlibm.llvm and FdlibmTranslit.class

- Function in LLVM is a control flow graph.
- Its nodes are basic blocks
- A basic block contains a list of instructions
- Each basic block has predecessors and successors

- We can build control flow graph from bytecode of JVM method
- Control flow graphs are almost the same except
- Jump chains: LLVM has empty basic blocks, bytecode resolves them
- Translation of simple condition expressions (p ? 1 : -1) – Cselect instruction in LLVM

- LLVM has unbounded number of registers, JVM has local stack and local variables

Acknowledgements

- Jo Ebergen for his mentorship and for help with these slides
- J Moore, Matt Kaufmann, Warren Hunt Jr. for ACL2
- David Rager, Austin Lee, Ben Selfridge, Cuong Chau for team work
- David Russinoff, Jared Davis, Sol Swords, Hanbing Liu – for their ACL2 books

ORACLE®