Hardware Verification[a] (Illustrated with PVS)

N. Shankar
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

[a] Joint with Sree Rajan, Harald Ruess, and Mandayam Srivas. Supported by NASA, NSF Grant CCR-930044 and ARPA Contract PR8556.

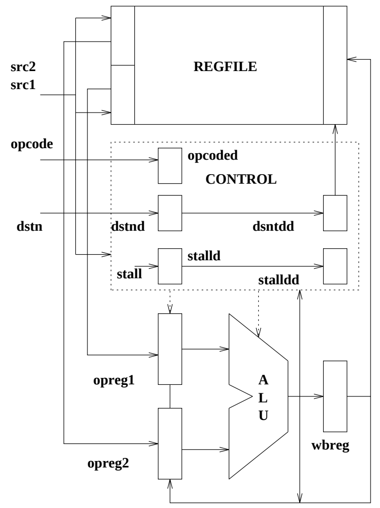
Why Verify Hardware?

- Hardware errors can be costly.
- Hardware can be cleanly specified and easily verified with the aid of machine assistance.
- For example, PVS supports hardware verification through:
  - the use of an expressive specification language
  - decision procedures for equality and arithmetic
  - Memoized rewriting
  - BDD-based propositional simplification and model checking
- Theorem provers such as Nqthm, HOL, Nuprl, and Veritas, and model checkers such as SMV and COSPAN, have been used in serious hardware verification efforts.

Outline

- This talk illustrates some basic techniques for specifying and verifying hardware.
- The first example shows the specification and verification of a simple pipelined processor.
- The second example develops the mathematics underlying the SRT algorithm for floating-point division and verifies a specific implementation.

A Pipelined Microprocessor



The Clarke Pipeline

- Example due to Clarke, et. al., 1991; a datapath oriented "benchmark" for model checkers
- Implements register-to-register operations only:
  `opcode, src1, src2, dstn`
- Three stage pipeline:
  - *read* register file and load into `opreg1` and `opreg2`
  - *compute* ALU operation and load result into `wbreg`
  - *write* back result into register file
- Uses register file bypass logic for handling data hazards
- Capable of aborting instructions in the pipeline to implement delayed branching

The Clarke Pipeline

- Controller uses internal registers (`dstnd`, `dstndd`, etc.) to save relevant information about partially executed instructions in the pipe to ensure operands for ALU operation are not stale.
- Correctness property proved: An instruction entering the pipeline at time `t` gets completed as expected at `t+3`.
  - Provided `stall` at time `t` is false,
  - the contents of destination register (`dstn(t)`) in the register file at `t+3` will be the *expected* result of the instruction entering the pipeline at `t`.
  - Note: expected result must be computed as a function of the state of the register file at `t+2` and not at `t`.

Specifying the Pipelined Processor

- We use an equational approach
- The state of every state-sensitive component in the design is modeled as a `signal`, i.e., a function from `time` to appropriately chosen `value` type.
- We define the state of a component at `t+1` as a function of the states of its inputs at earlier `t`.

Signals over Time

```
time: THEORY
  BEGIN
   time: TYPE = nat
  END time
```

```
signal[val: TYPE]: THEORY
  BEGIN
   IMPORTING time
   signal: TYPE = [time -> val]
  END signal
```

Specifying the Pipelined Processor

```
new_pipe[addr: TYPE, data: TYPE, opcodes: TYPE]
  : THEORY
  BEGIN
   IMPORTING time, signal
   t: VAR time
   %% Input wires: decoded instruction fields
      opcode: signal[opcodes]
      src1: signal[addr]
      src2: signal[addr]
      dstn: signal[addr]
   %% pipeline abort signal for control hazards
      stall: signal[bool]
   %% Register file
      regfile: signal[[addr -> data]]
   %% Internal registers %%
     % The Pipeline registers
       dstnd, dstndd: signal[addr]
       stalld, stalldd: signal[bool]
       opcoded: signal[opcodes]
    % Write back register
      wbreg: signal[data]
    % ALU input registers
      opreg1, opreg2: signal[data]
    % ALU
      aluop: [opcodes, data, data -> data]
      aluout: signal[data]
```


Registers and the Register File

```
dstnd_ax: AXIOM dstnd(t+1) = dstn(t)
dstndd_ax: AXIOM dstndd(t+1)= dstnd(t)

stalld_ax: AXIOM stalld(t+1) = stall(t)
stalldd_ax: AXIOM stalldd(t+1) = stalld(t)

opcoded_ax: AXIOM opcoded(t+1) = opcode(t)

wbreg_ax: AXIOM wbreg(t+1) = aluout(t)

regfile_ax: AXIOM regfile(t+1) =
            IF stalldd(t) THEN regfile(t)
             ELSE regfile(t)
                  WITH [(dstndd(t)) := wbreg(t)]
            ENDIF
```


The ALU

```
opreg1_ax: AXIOM opreg1(t+1) =
             IF src1(t) = dstnd(t) & NOT stalld(t)
                THEN aluout(t)
             ELSIF src1(t) = dstndd(t) & NOT stalldd(t)
                   THEN wbreg(t)
             ELSE regfile(t)(src1(t)) ENDIF

opreg2_ax: AXIOM opreg2(t+1) =
            IF src2(t) = dstnd(t) & NOT stalld(t)
               THEN aluout(t)
            ELSIF src2(t) = dstndd(t) & NOT stalldd(t)
                  THEN wbreg(t)
            ELSE regfile(t)(src2(t)) ENDIF

ALU_ax: AXIOM aluout(t) = aluop(opcoded(t), opreg1(t),
                                opreg2(t))
```


Pipeline Correctness

```
 correctness: THEOREM (FORALL t:
  NOT(stall(t)) IMPLIES
      regfile(t+3)(dstn(t)) =
        aluop(opcode(t), regfile(t+2)(src1(t)),
              regfile(t+2)(src2(t))) )

END new_pipe
```

Proved by

```
(""
 (AUTO-REWRITE-THEORY "pipe")
 (SKOSIMP*)
 (APPLY (REPEAT (ASSERT)))
 (APPLY (REPEAT (LIFT-IF :UPDATES? NIL)))
 (APPLY (THEN (BDDSIMP) (ASSERT))))
```

SRT Division

Motivation

- 5 missing transistors in (initial releases of) Pentium floating-point division hardware
- Reasonable error probability and significant error [Coe, Tang 95, Pratt '95]
- Cost $\$475,000,000$

Outline

- Fundamentals of SRT Division
- Formalization of SRT Division in PVS
- Modular Approach for the Correctness Proof

Why Verify SRT in PVS?

- Theorem proving can deal with numerical constraints better than finite-state methods.
- PVS has a high-level specification language so the algorithm can be modelled accurately.
- PVS combines decision procedures for linear arithmetic and propositional simplification (BDDs) with high-level proof strategies like induction.

Example

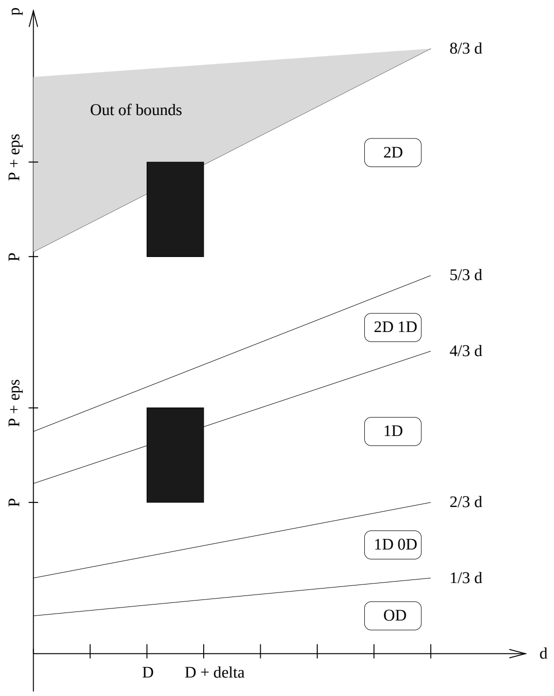
$p \;=\; 143/128 \;=\; 1.0001111$
$d \;=\; 11/8 \;=\; 1.011$
Note that $p/d \;=\; 13/16 \;=\; 0.31_4$

p_0	=	0001	.0001111	
sub d	=	1	.011	set $q_0 := 1$
$p_1/4$	=	1111	.1011111	
shift p_1	=	1110	.11111	
add d	=	1	.011	set $q_1 := \bar{1}$
$p_2/4$	=	0000	.01011	
shift p_2	=	0001	.011	
sub d	=	1	.011	set $q_2 := 1$
$p_3/4$	=	0000	.0	

Quotient:

$$q \;=\; (1.\bar{1}1)_4 \;=\; 1 - 1/4 + 1/16 \;=\; 13/16$$

Table Constraint



Numerical Basics

Floating Point Numbers $-1^s\ m\ 2^e$

Given

- Dividend mantissa p
- Divisor mantissa d
- Both in normalized form $1.xxxx_2$ (thus: $1 \leq p, d < 2$)

Compute

- Quotient mantissa q (to sufficient precision) in redundant radix-4 representation and quotient digits

$$\bar{2},\ \bar{1},\ 0,\ 1,\ 2$$

- Exponent (based on exponents of dividend and divisor)

Here restriction to fixed-point division, no pre- and postcomputations like scaling, quotient conversion or rounding

SRT Recurrence

Parameters

- Radix `r`
- Quotient range `[-a, a]`, `r/2 <= a < r`.

(typically `r = 4`, `a = 2`: adequate and efficient, since multiplication by 0, 1, and 2 is easy)

Compute

- Quotient: `q(0).q(1)q(2)...`, where
  `q(i): [-a, a]`

- Partial Remainder

```
p(0)       = p
p(i + 1)   = r * (p(i) - q(i) * d)
```

  such that for `rho := a / (r - 1)`

```
remainder_bound? :=
   | p(i + 1) / d | <= r * rho
```


Correctness of Recurrence

Theorem

```
| p(0) / d - q(0).q(1)...q(k) | <= rho/r^k
```

Remark

`q(0).q(1)...q(k - 1)` denotes the (redundant) radix-r fixed-point interpretation of the sequence of quotient digits.

Proof

Follows immediately from `remainder_bound?` and invariant

```
p(0) / d - q(0).q(1)...q(k) =
   p(k+1) / (d *r^(k+1))
```

which is proved by a simple induction on `k`

Quotient Selection

Hard part in each iteration is to determine a suitable quotient digit `q(i)`!

Characterization of legitimate quotient selections `q`

`| p(i + 1) / d | <= r * rho`

$\iff$ { recurrence }

`| p(i) - q(i) * d | <= rho * d`

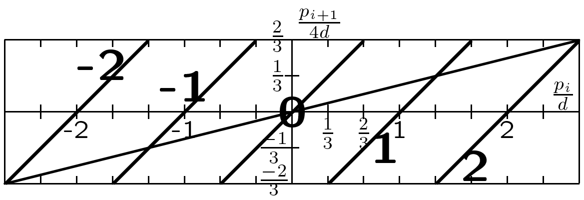
$\iff$ { algebra }

`(q(i) - rho) * d <= p(i)`
`<= (q(i) + rho) * d`

Example

for `r := 4`, `a := 2`,
and consequently `rho := 2/3` one gets several choices for some `(d, p(i))`!

Robertson Diagram [from Pratt]



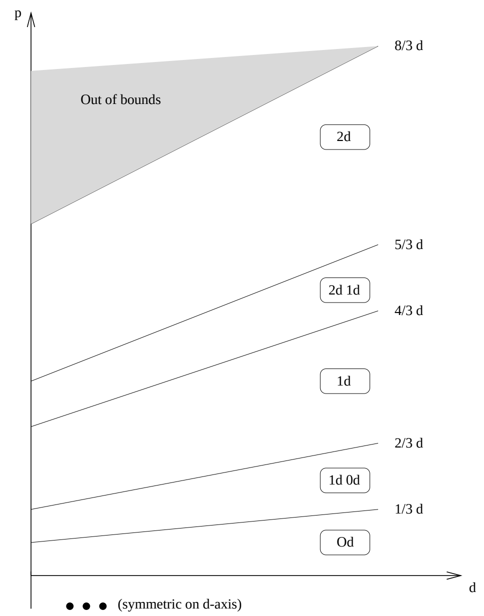
Slope of main diagonal is $1/4$.

Slope of dark q-digit lines is 1.

One step of SRT:

- compute q-digit and p_{i+1} by going vertically from $(p_i/d, y)$ position on main diagonal to suitable q-digit.
- Move horizontally to y-axis for new $p_{i+1}/4d$ value.
- Transpose about the diagonal to multiply by 4.

Constraints for Quotient Selection



Overlap

Remark: Overlap of the selection intervals permits calculating quotient digit `q` from truncated versions `P` and `D` such that

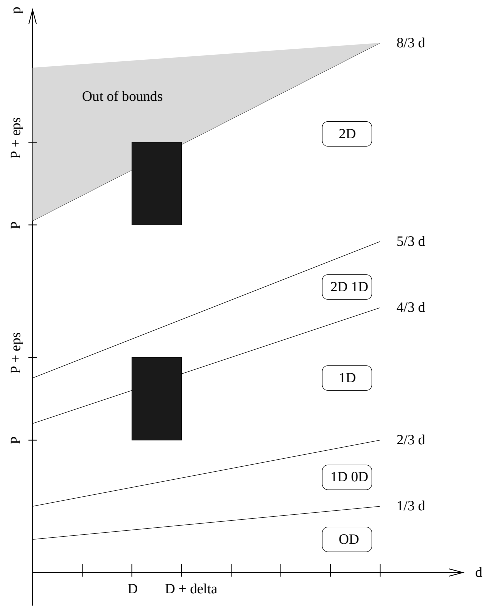
```
D <= d < D + delta

P <= p < P + eps
```

Speed-up:

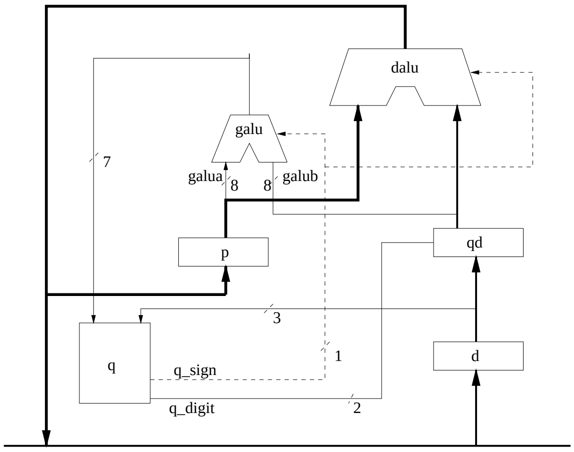
- Not all bits of the new partial remainder must be known in order to select the next quotient digit.
- Overlap full-precision calculation of the partial remainder in step `i` with the selection process of the quotient digit in step `i + 1`.

Table Constraint



Yields `lookup_legitimate?` constraint.

Division Data Path



The PVS Verification System

A very expressive specification language

- rich type system: arrays, records, predicate subtypes, etc.
- `TABLE` construct for defining functions.

Low-level decision procedures: BDDs, congruence closure, linear arithmetic.

Powerful primitive inference steps: `bddsimp`, `assert`, `do-rewrite`.

High-level strategies: `grind`, `induct-and-simplify`, etc.

Verification is decomposed into three parts

- Top level SRT correctness: Division works correctly, if quotient digits are selected according SRT invariant property
- SRT invariant proof for an abstract version of the algorithm based on a general (sufficient) condition on the legitimacy of a look-up table
- Verification of a particular data path and look-up table

Quotient Selection

Hard part in each iteration is to determine a suitable quotient digit `q(i)`!

Characterization of legitimate quotient selections `q`

```
     | p(i + 1) / d | <= r * rho
<=>  { recurrence }
     | p(i) - q(i) * d | <= rho * d
<=>  { algebra }
     (q(i) - rho) * d
     <= p(i)
     <= (q(i) + rho) * d
```

Example

for `r := 4`, `a := 2`,
and consequently `rho := 2/3` one gets several choices for some `(d, p(i))`!

Legitimacy of Quotient Selection

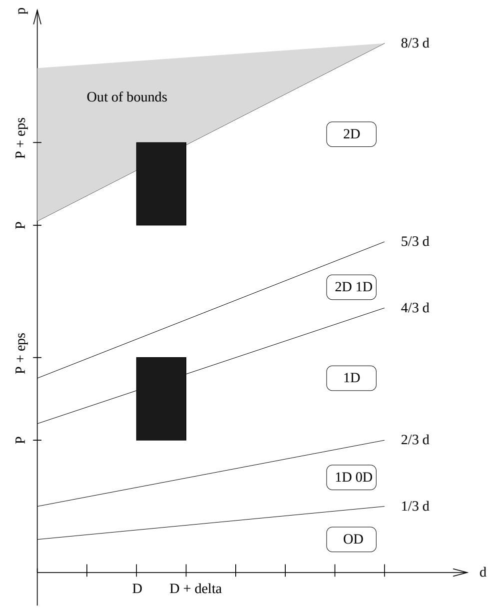
```
legitimate?(q, d, p): bool =
 (q - rho <= p / d  AND  p / d <= q + rho)

remainder_bound_equiv_legitimate: LEMMA
 (FORALL (p_new, p, q, d):
     recurrence?(p_new, p, q, d) IMPLIES
       (remainder_bound?(d, p_new)
        IFF legitimate?(q, d, p)))
```

Proof:

```
(then (skosimp*)
      (expand "legitimate?")
      (both-sides "-" "q!1")
      (both-sides "*" "r")
      (grind))
```


Table Constraint



Yields `lookup_legitimate?` constraint.

Quotient Selection Based on Approximation

Approximate version of `remainder_bound?`.

```
estimation_bound?(D, P): bool =
 (-eps - r * rho * (D + delta) < P
      & P < r * rho * (D + delta))
```

Condition on legitimate quotient selection.

```
lookup_legitimate?
   (q, D, (P: rational
           | estimation_bound?(D, P)))
   : bool =
 COND
  q = 2  -> 4/3 * (D + delta) <= P,
  q = 1  -> 1/3 * (D + delta) <= P
            & (P + eps) <= 5/3 * D,
  q = 0  -> -2/3 * D <= P
            & (P + eps) <= 2/3 * D,
  q = -1 -> -5/3 * D <= P
            & (P + eps) <= -1/3 * (D + delta),
  q = -2 -> (P + eps) <= -4/3 * (D + delta)
 ENDCOND
```


Quotient Selection (contd.)

```
remainder_bound_implies_estimation_bound
   : LEMMA
  (P <= p & p < P + eps
    AND D <= d & d < D + delta
     AND remainder_bound?(d, p))
   IMPLIES estimation_bound?(D, P)
```

Approximate quotient selection condition guarantees valid exact quotient digit.

```
lookup_is_legitimate: LEMMA
 (P <= p & p < P + eps
   AND D <= d & d < D + delta
    AND remainder_bound?(d, p)
     AND lookup_legitimate?(q, D, P))
  IMPLIES legitimate?(q, d, p)
```

PROOF

```
(grind :theories "real_props")
```

The Main Invariant for SRT Division

```
invariant: THEOREM
 remainder_bound?(d, p(0)) AND
    legitimate?(q(0), d, p(0)) AND
     (FORALL j:
       recurrence?(p(j + 1), p(j), q(j), d)
        AND P(j) <= p(j + 1)
            & p(j + 1) <  P(j) + eps
              & D <= d & d < D + delta
         AND (estimation_bound?(D, P(j))
             IMPLIES
              lookup_legitimate?(q(j + 1),
                                 D, P(j))))
 IMPLIES
   remainder_bound?(d, p(i))
    AND legitimate?(q(i), d, p(i))
```

Proof is applicable for any `delta` and `eps` as long as quotient selection technique satisfies `lookup_legitimate?` and `estimation_bound?`.

Proof of Main Invariant

Proof is by induction on `i` by means of chaining together the previously shown lemmas:

```
    legitimate?(qq(i), d, pp(i))
⇒   remainder_bound?(d, pp(i + 1))
⇒   estimation_bound?(D, PP(i))
⇒   lookup_legitimate?(qq(D, P(i)), D, PP(i))
⇒   legitimate?(qq(i + 1), d, pp(i + 1))
```


Specializing to Particular Datapath

To verify a particular data path, it is sufficient to check the following:

- Partial remainder computation satisfies the recurrence relation
- Approximations on remainder and divisor used satisfy `estimation_bound?`
- Quotient look-up table used satisfies `lookup_legitimate?` condition

Formalizing the Data Path

Signals described by uninterpreted sequences of bitvectors indexed over time

```
d                 : signal[bvec[N]]
dalu              : signal[bvec[N + 1]]
galu, galua, galub: signal[bvec[8]]
P                 : signal[bvec[7]]
D                 : signal[bvec[3]]
...
```

Arithmetic Interpretation i.e.

```
dalu(t): rational =
    fp2c[2, N - 1].val(dalu(t))
```

ALU Formalization by Constraints

```
p_eqn      : AXIOM      p(t + 1) = 4 * dalu(t)

dalu_eqn: AXIOM
  dalu(t) =
    IF q_sign(t) = 1 THEN p(t) - qd(t)
                     ELSE p(t) + qd(t) ENDIF

ulp(m:nat) = expt(2,-m)

galu_eqn: AXIOM galu(t) =
    IF q_sign(t) = 1
     THEN galua(t) - galub(t) - ulp(6)
     ELSE galua(t) + galub(t) ENDIF

galua_ineq  : AXIOM
  galua(t) <= p(t) & p(t) < galua(t) + ulp(6)

galub_ineq  : AXIOM
  galub(t) <= qd(t) & qd(t) < galub(t) + ulp(6)

galu_ineq   : AXIOM
  P(t) <= 4 * galu(t)
   & 4 * galu(t) <= P(t) + ulp(3) - ulp(4)

d_bound_by_D: AXIOM
  D(t) <= d(t) & d(t) < D(t) + 1/8
```


Some Facts About the Data Path

Recurrence

```
p(t + 1) = 4 * (p(t) - q(t) * d(t))
```

Accuracy of Estimation

```
P(t) <= p(t + 1) < P(t) + 3/16
```

Thus: `delta = 1/8, eps = 3/16`

Proofs

Combination of rewriting with decision procedures proves these facts automatically

Formalizing Taylor's Lookup Table

```
q(D: bvec[3], (P: bvec[7] | estimation_bound?(D, P))):
    subrange(-2, 2) =
LET a = -(2 - P(1) * P(0)), b = -(2 - P(1)),
    c = 1 + P(1), d = -(1 - P(1)), e = P(1)
  ...
```



```
...
IN TABLE P^(6,2), D
           |[ 000| 001| 010| 011| 100| 101| 110| 111]|
      %--------------------------------------------%
      |01010|    |    |    |    |    |    |    |  2 ||
      |01001|    |    |    |    |    |  2 |  2 |  2 ||
      |01000|    |    |    |    |  2 |  2 |  2 |  2 ||
      |00111|    |    |  2 |  2 |  2 |  2 |  2 |  2 ||
      |00110|    |  2 |  2 |  2 |  2 |  2 |  2 |  2 ||
      |00101|  2 |  2 |  2 |  2 |  2 |  2 |  2 |  1 ||
      |00100|  2 |  2 |  2 |  2 |  c |  1 |  1 |  1 ||
      |00011|  2 |  c |  1 |  1 |  1 |  1 |  1 |  1 ||
      |00010|  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 ||
      |00001|  1 |  1 |  1 |  1 |  e |  0 |  0 |  0 ||
      |00000|  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
      |11111|  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
      |11110| -1 | -1 |  d |  d |  0 |  0 |  0 |  0 ||
      |11101| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 ||
      |11100|  a |  b | -1 | -1 | -1 | -1 | -1 | -1 ||
      |11011| -2 | -2 | -2 |  b | -1 | -1 | -1 | -1 ||
      |11010| -2 | -2 | -2 | -2 | -2 | -2 |  b | -1 ||
      |11001| -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 ||
      |11000|    |    | -2 | -2 | -2 | -2 | -2 | -2 ||
      |10111|    |    |    | -2 | -2 | -2 | -2 | -2 ||
      |10110|    |    |    |    |    | -2 | -2 | -2 ||
      |10101|    |    |    |    |    |    | -2 | -2 ||
      %--------------------------------------------%
    ENDTABLE
```

Correctness of Taylor's Lookup Table

To show

- Valid indices never point to blank table entries.
  Ensured by well-formedness constraints generated by PVS typechecker.
- Every valid index yields a legitimate quotient digit.

Proofs

Simple case split on all possible values of indices `P` and `D` followed by a call of decision procedures.

Generation of Counterexamples

Experiment

- Inject a noncorrect value at a certain position
- *Rerun* the proof above

Result

```
P!1(6) = 1, P!1(5) = 1, P!1(4) = 0,
P!1(3) = 0, P!1(2) = 0
estimation_bound?(...)
D!1(2) = 0, D!1(1) = 1, D!1(0) = 0
|-------
lookup_legitimate?(...)
```

Counterexample

```
D = 010,      P = 11000
```

Remark

Most probably, a similar analysis could have detected the erroneous table entries in the Pentium FDIV table.

Putting it All Together

General theory of SRT division

Facts about Data Path

Correctness of Lookup Table

implies

```
| p /d - q(0).q(1)...q(t) | <= 2 / (3 * 4^t)
```


Related Work on Computer Arithmetic

- Bryant (1995) uses BDDs to verify one iteration of SRT division.
- Verkest, Claesen, and de Man (1994) have mechanically verified a nonrestoring division circuit.
- Leeser and O'Leary (1995) have mechanically verified a radix-2 square root algorithm.
- Clarke and Zhao (1995) use *word-level model checking* to verify Taylor's SRT circuit.
- German, Clarke, and Zhao (1996)use Analytica to verify radix-4 SRT algorithm.
- Lynch, Moore, and Kaufmann verified microcode for AMD's K5 divider using ACL2.
- Miner has formalized parts of the IEEE floating-point state in PVS.

Conclusions

- Modular approach structures the SRT proof in a nice way
- Proofs are fairly automatic
- Tight integration of decision procedures with rewriting strategies proved to be a useful workhorse
- Allows *reuse* of parts of the development for verifying other similar circuits
- Concepts of the PVS specification language like
  - predicate subtypes
  - overloading
  - tables

  proved useful for expressing high-level designs of this arithmetic circuit.