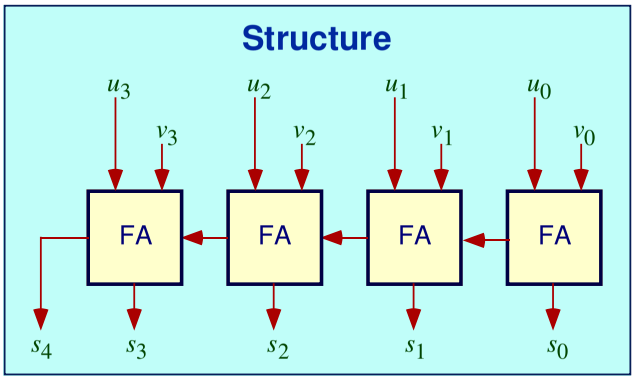
Implementing Computer Arithmetic

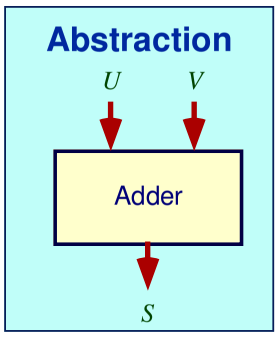
Randal E. Bryant

CS 15-347
March 14, 1996

- Fast Addition
- Pentium Division

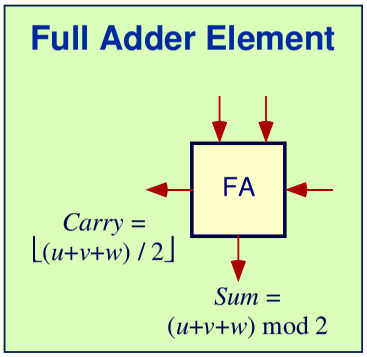
Conventional Adder Design



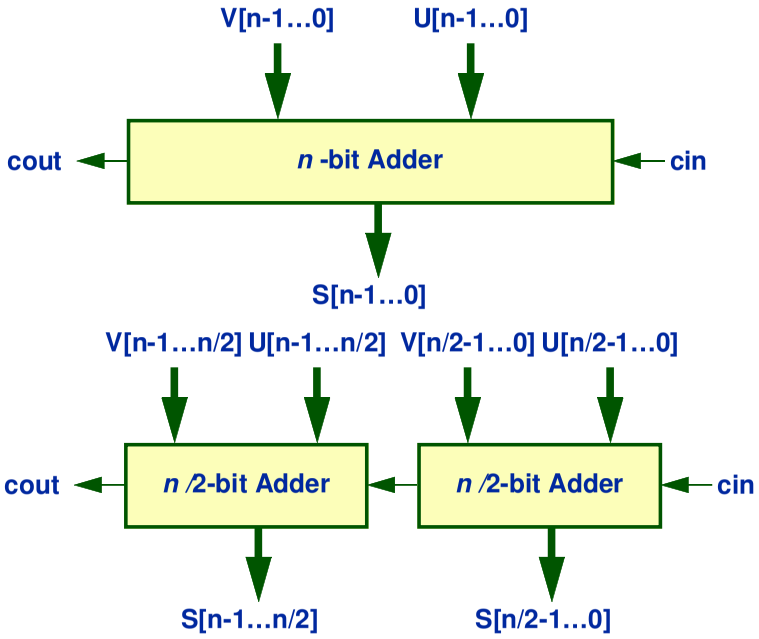


Shortcoming

- Must propagate carries across all cells
- Can improve with more costly hardware



Divide & Conquer?



- This is not an improvement

Carry Function Reformulation

Carry Function

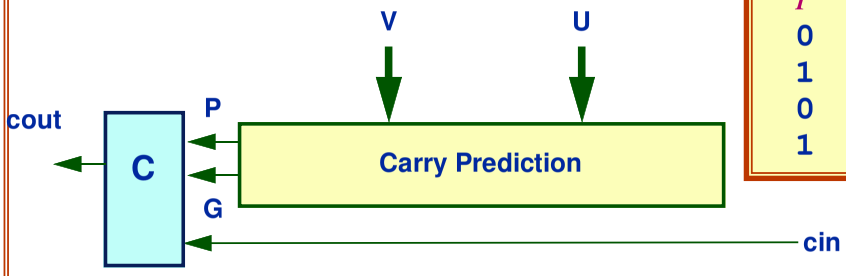
u	*v*	*cin*	*cout*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Alternate Formulation

u	*v*	*cout*	
0	0	0	kill
1	0	*cin*	propagate
0	1	*cin*	propagate
1	1	1	generate

- When both local inputs 0, no carry
- When one is 0, the other is 1, propagate carry input
- When both are 1, then generate a carry

Block Level Carry

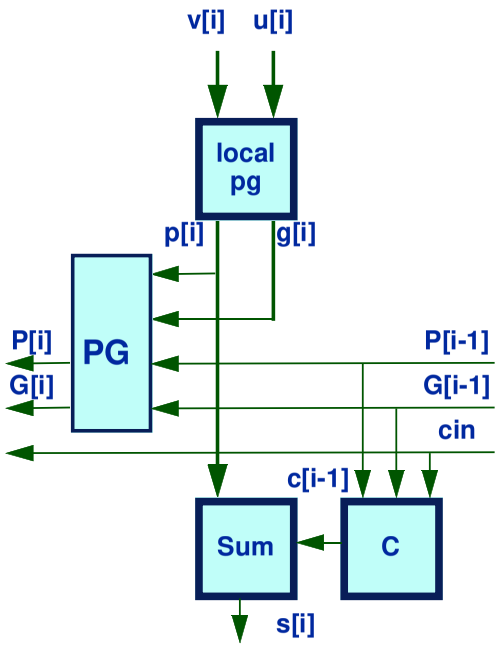


C Block

P	*G*	*cout*	
0	0	0	kill
1	0	*cin*	propagate
0	1	1	generate
1	1	–	impossible

- P indicates that carry propagates through block
- G indicates that block generates carry

Reformulated Adder Cell



local pg

u[i]	v[i]	p[i]	g[i]
0	0	0	0
0	1	1	0
1	0	1	0
0	1	0	1

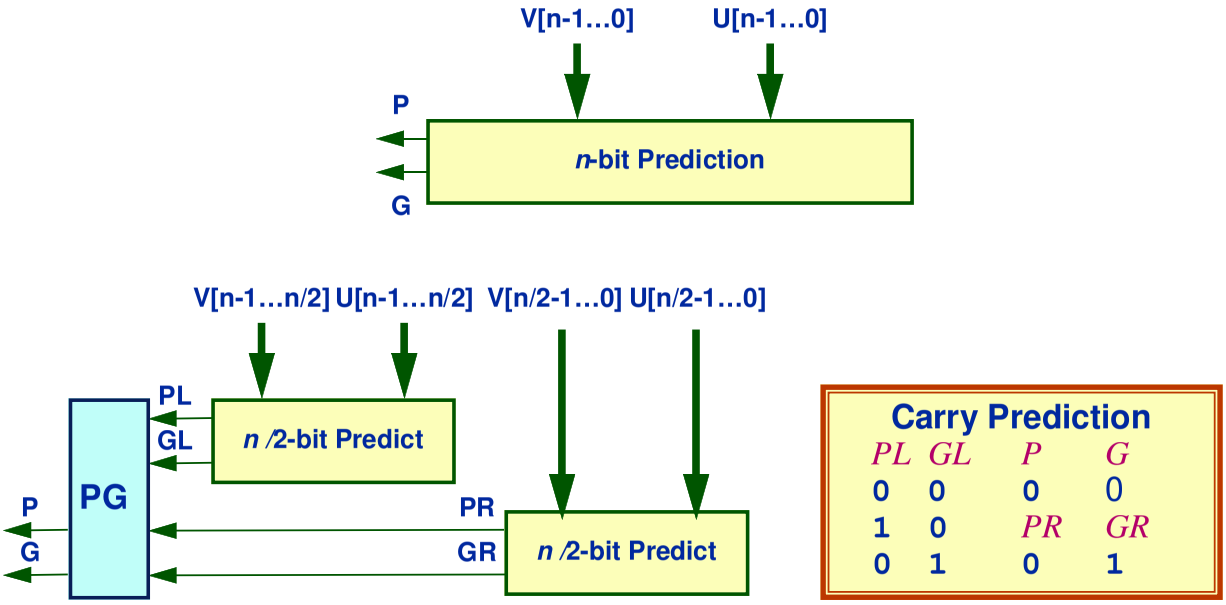
Carry Prediction: PG

p[i]	g[i]	P[i]	G[i]
0	0	0	0
1	0	P[i-1]	G[i-1]
0	1	0	1

Sum

p[i]	c[i-1]	s[i]
0	0	0
0	1	1
1	0	1
1	1	0

D & C Carry Prediction

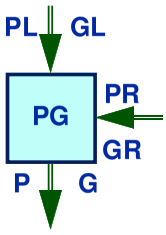


Carry Prediction

PL	*GL*	*P*	*G*
0	0	0	0
1	0	*PR*	*GR*
0	1	0	1

- Divide & conquer yields shallow circuit

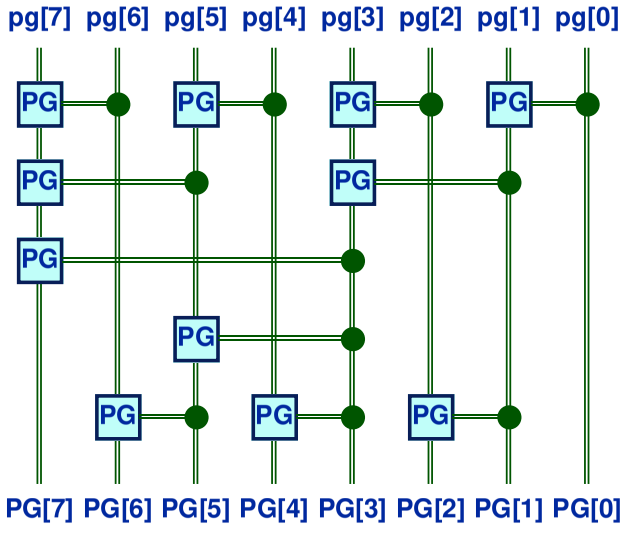
Carry Prediction Tree



- O(n) complexity
- O(log n) depth

Generalization

- Parallel Prefix
- Any associative operation



Division, Pentium Style

Summary

- **Pentium is Intel's most advanced microprocessor**
  - 3.3 Million transistors
  - In 42% of all personal computers sold Dec. '94
- **Until recently, all had error in floating point division hardware**
  - 5 missing transistors, fixed with change to single mask
- **Disclosed largely via Internet**
- **Intel ultimately offered replacements to everyone**
  - $475 Million charge from 4Q94 revenue

Outline

- **Brief chronology**
- **Technical details**
- **Overall impact**

Numerical Basics

Floating Point Numbers

- **Numerical Form:** $-1^s\ m\ 2^e$
  - Mantissa m either 24, 53, or 64 bits

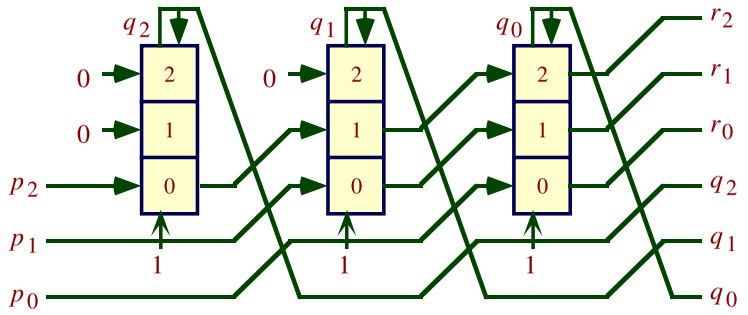
Given

- **Dividend mantissa** p
- **Divisor mantissa** d
- **Normalized forms:** $\mathtt{1.xxxx}_2$
  - Range Constraint: $1 \le p, d < 2$

Compute

- **Quotient mantissa** q
  - To sufficient precsion
- **Exponent**
  - Based on exponents of dividend & divisor

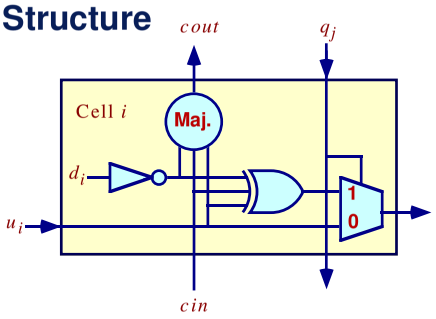
Radix 2 Divider Circuit



Operation

- Each stage generates 1 bit of quotient
- Conditionally subtracts divisor
- Partial remainder shifted

Cell Structure



Radix 4 Division

Conventional "Restoring" Algorithm

- Select quotient digit q_i
  - 0, 1, 2, or 3
- Subtract $q_i d$ from p
- Shift p left

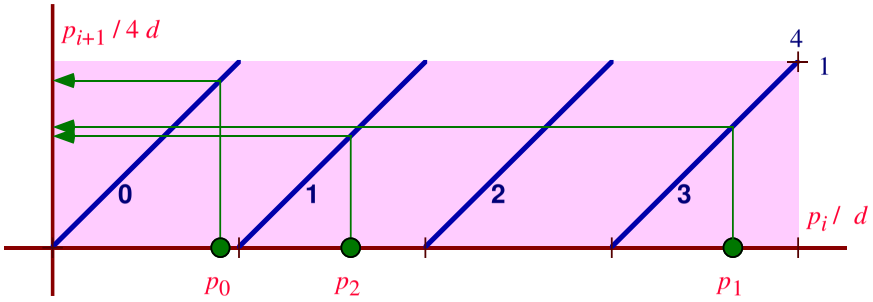
Divisor d	
X 1	1.301
X 2	3.202
X 3	11.103

- Must consider all digits to select digit q_i
- Must form "awkward multiple" $3d$

Example

p_0	1.2130	q_1	0
$-q_1 d$	- 0.		
	1.2130		
p_1	12.1300	q_2	3
$-q_2 d$	-11.1030		
	1.0210		
p_2	10.2100	q_3	2
$-q_3 d$	- 3.2020		
	1.0020		
	10.0200		

Visualization of Digit Selection



Interpretation

- Horizontal axis indicates scaled partial remainder
  - In range [0, 4)
- Vertical axis indicates result of subtracting quotient digit
  - Unique choice of digit
  - In range [0, 1)

SRT Division

History

- Radix 4 version due to Atkins (1968)
- Commonly used by others, first time for Intel

Goals

- Predict quotient digit based on incomplete information about partial remainder and divisor
- No awkward multiples

Key Ideas of Method

- Quotient Digits –2, –1, 0, +1, +2
  - Redundancy allows imperfect digit prediction
  - Power of two multiples
    - » Implement with shift and/or complement

SRT Iteration Step

Given

- **Partial Remainder** p_i
  - Two's complement representation
  - Can bound range $-16/3 < p_i < 16/3$
- **Divisor** d
  - Range Constraint: $-8/3\ d \leq p_i \leq 8/3\ d$

P

sxxx.xxxxxx ... x_2

D

1.xxxxxx ... x_2

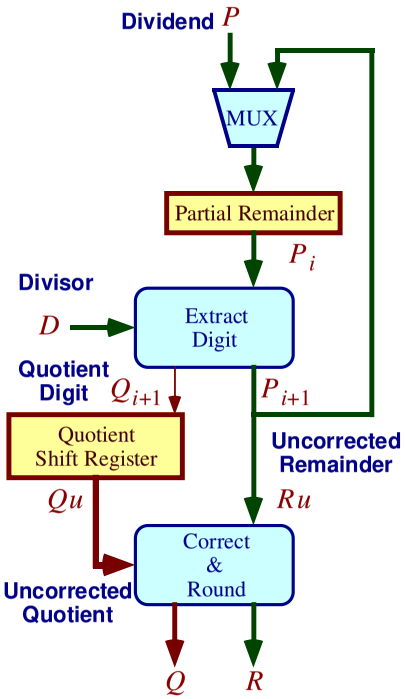
Select

- **Quotient Digit** q_{i+1}
  - By table lookup
  - Using truncated values P and D
- **Compute** $p_{i+1} = 4\ [p_i - q_{i+1}\ d\ ]$

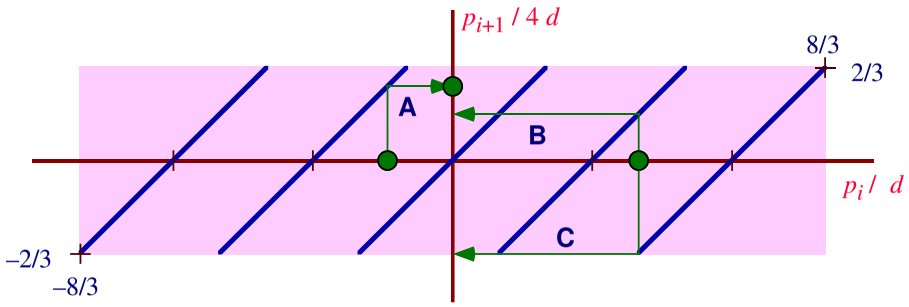
SRT Divider Circuit

Registers

- **Hold partial remainder**
  - Initially loaded with dividend
- **Divisor (not shown)**
  - Unchanged for entire computation
- **Quotient shift register**
  - Accumulates digits from iterations



Quotient Digit Selection



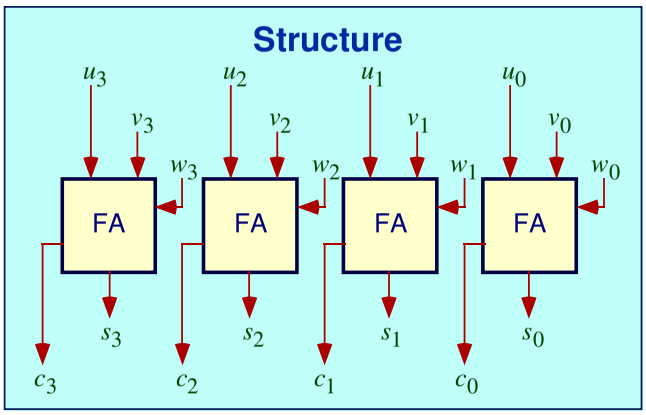
Examples

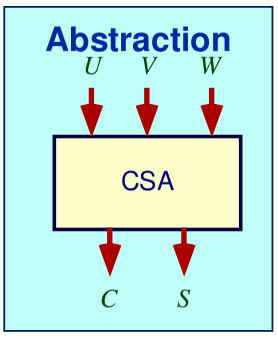
	p_i	q_{i+1}	p_{i+1}	
A	$-7/15\ d$	-1	$4\ [8/15]\ d$	$= 32/15\ d$
B	$4/3\ d$	$+1$	$4\ [1/3]\ d$	$= 4/3\ d$
C	$4/3\ d$	$+2$	$4\ [-2/3]\ d$	$= -8/3\ d$

Invariant Preserved

$$-8/3\ d \le p_{i+1} \le 8/3\ d$$

Carry Save Adders





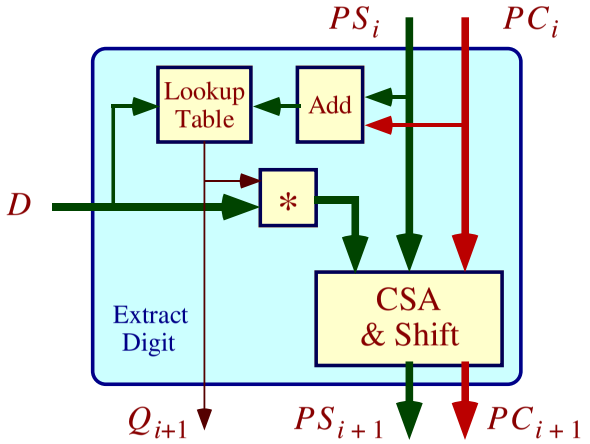
$$S + 2C = U + V + W$$

- Rearrange elements to eliminate carry chain
- Three input words
- Two output words
- 3 to 2 reducer

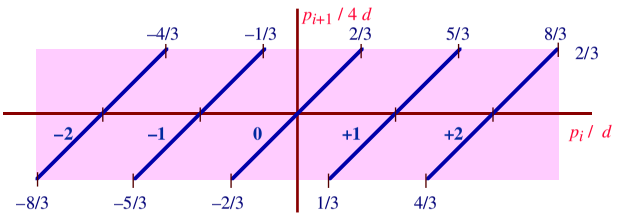
SRT Division Step

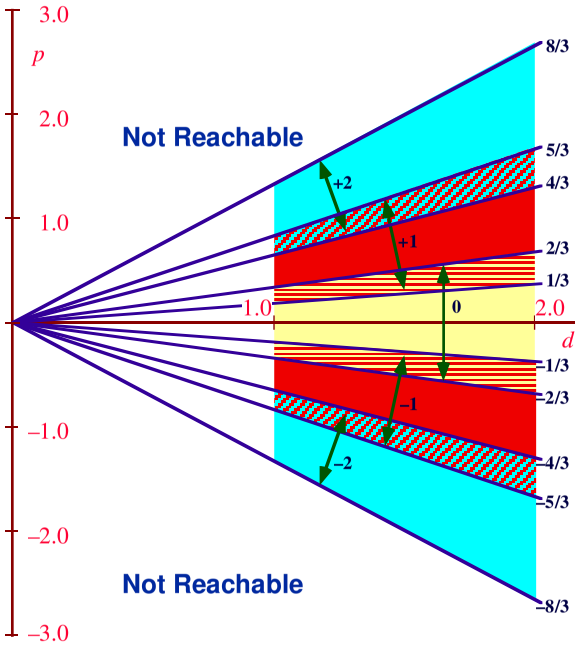
Elements

- **Carry Save Adder to subtract weighted divisor**
- **7-bit adder to get estimate of partial remainder**
  - High order bits
- **Lookup table to determine quotient digit**
  - 4 “interesting” bits of divisor
    - » Leading bit always 1
  - Implemented as PLA
  - Where Intel made mistake
- **Shift/complementer to weight divisor**
  - Multiply by quotient digit



Lookup Table Design





Digit Selection

- **Depends on ratio of d and p**
- **Redundancy leads to overlapping regions**
  - May choose either digit

Discretization of P & D

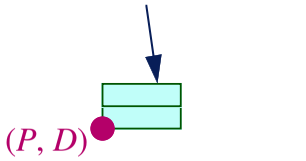
Values

- **Least Resolvable Unit (LRU)**
  - Weight of least significant bit
- **Partial Remainder** p_i
  - Approximate by P
  - Potentially underestimates by two LRUs
  - One each from pc and ps
- **Divisor** d
  - Approximate by D
  - Potentially underestimates by one LRU

P `sxxx.xxxxxx ...`

D `1.xxxxxx ...`

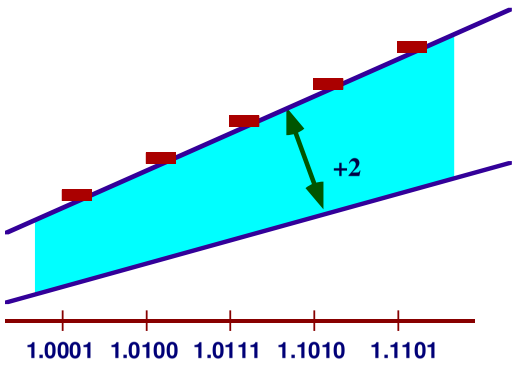
Range of Values for (p_i, d)



Compensating for Inaccuracy

- **Redundancy enables valid digit selection for all cases**
  - As long as have enough bits of p_i and d

Intel's Table Errors

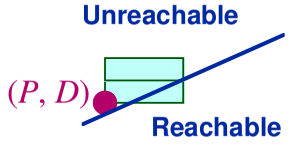


Actual Bug

- **Erroneous entries for 5 cases**
- **Table generates 0 instead of +2**

Occurs Only under Marginal Conditions

- (P, D) **appear to be out of range**
- D **must under-estimate** d
- P **must closely estimate** p_i
  - Unlikely with carry-save format



Example of Error

Correct Behavior

p_0	0100.111000	q_1	+2
$+ - q_1 d$	1100.010010		
	0001.001010		
p_1	0100.101000		

Incorrect Behavior

p_0	0100.111000	q_1	0
$+ - q_1 d$	0000.000000		
	0100.111000		
p_1	0011.100000		

Leading digits drop off end

Divisor	d
X 1	0001.110111
X 2	0011.101110
X -1	1110.001001
X -2	1100.010010

- Once hit bad table entry, cannot recover
- Contrary to some statements in press
  - Not "Self Correcting" !

When Can Error Occur?

Requires at Least 7 Iterations

- Hard to get to bad positions
- Worst case error 6 X 10^{-5}

Only for "At Risk" Divisors

- Stay in single column for entire division
- Remaining part of divisor should have lots of 1's
  - To ensure that D underestimates d
- Coe's Example
  - $d = 3145727_{10}$ 10111111111111111111
  - Correct result: 1.33382044913624100
  - Pentium's result: 1.33373906890203759

At Risk!

- Nicely's Example
  - $d = 824633702441_{10}$ 10111111111111111111011100000101001

Table Realization

Programmed Logic Array

- Encodes sum-of-products logic expression
- Coding generated automatically
  - Given complete table
  - With ESPRESSO program
- Mask patterns generated automatically

Likely Source of Error

- Specified entries as "Don't Cares"
  - Since appear to be out of range
- PLA optimizer chose to assign 0's

Table

u	*v*	*c*	*Carry*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

PLA

u	*v*	*c*	*Carry*
–	1	1	1
1	–	1	1
1	1	–	1

How Serious is the Error?

Intel's Claims

- Most applications don't do much division
  - 1000 per day
- Many applications can tolerate occasional errors
  - E.g., image generation
- Maximum error fairly small
  - Once every 1.5 X 10^9 divides
- Statistical likelihood remote
  - Once every 27,000 years

IBM's Counterclaims

- Even spreadsheet users do lots of divides
  - Up to 4.2 Million per day
- Error more likely for numbers just below integer
  - Up to once every 10^8 divides

Other Concerns

Error is not Random

- Get same result every time on every Pentium

Numbers Used in Practice NOT Uniformly Distributed

- E.g., `4.999999 / 14.999999`
  - gives `0.33332922` instead of `0.33333329`
  - Vaughan Pratt's "bruised integers"

Error is Hard to Work Around

- Can shut off floating point
  - Machine then runs slower than 486
- Can recompile with workaround divide subroutine
  - But most users only have binaries

Bad Numerical Properties

- E.g, `4.999999 / 15.0 > 4.999999 / 14.999999`
  - Problem for iterative methods

Who is Affected?

Average PC User

- Not likely

Heavy Users of Numerical Software

- Significant subset of Pentium customers
- Concern for numerical accuracy very high
  - Single inaccurate result can be magnified by later steps
- Ethical/legal responsibilities
  - Negligent to knowingly use flawed system
  - Would you buy a bridge from a civil engineer who had used a Pentium?

Why Didn't Intel Discover it?

Standard Steps in Verifying Design

- **Simulate many cases on high-level software model**
  - Probably had correct P-D table entries
- **Simulate/emulate final logic design**
  - Hardware emulators costing $Millions
  - Run complete chip model at ~ 100Hz
    - » < 10^{-6} X real time
  - Not clear which version of table was used
- **Run tests on initial production chips**
  - Feasible to run billions of tests
  - Should have caught error here

Observations

- **Hard to test all aspects of such a complex system**
  - But this is a weak excuse

System Considerations

Microprocessor Complexity Increasing at Rapid Pace

- **Pentium has features historically found in high-end mainframes**
  - Hardware support for multiplication, division, square root, and transcendental functions
  - Sophisticated instruction pipeline
- **Exercising all possible system behaviors especially difficult**
- **Future generations even more complex**
  - Speculative execution

Must Get it Right First Time

- **Cannot send field service to replace boards**
- **Typically have more people working on verification than on design**

Ubiquitous Systems of Unprecedented Complexity

- **Industry's headaches caused by own successes**