Computer Architecture and Engineering
Lecture 7: Divide, Floating Point, Pentium Bug

September 17, 1997

Dave Patterson (http.cs.berkeley.edu/~patterson)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Outline of Today's Lecture

- Recap of Last Lecture and Introduction of Today's Lecture (4 min.)
- Divide (20 min.)
- Questions and Administrative Matters (2 min.)
- Floating-Point (25 min.)
- Questions and Break (5 min.)
- Pentium Bug [Patterson] (25min.)

Recap of Last Lecture: Summary

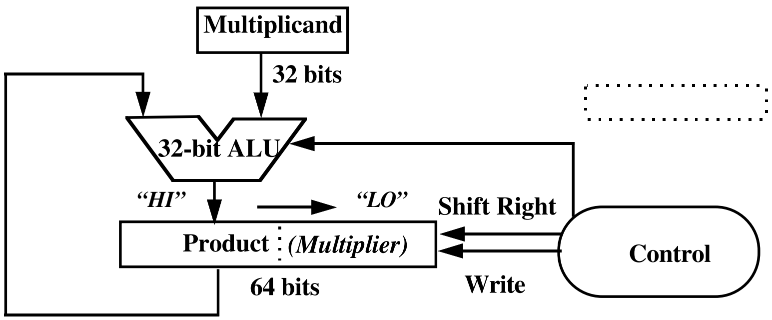
- Intro to VHDL
  - entity = symbol, architecture ~ schematic, signals = wires
  - behavior can be higher level
  - x <= boolean_expression(A,B,C,D);
- On-line Design Notebook
  - Open a window with editor, or our tool, => cut&paste
- Multiply: successive refinement to see final design
  - 32-bit Adder, 64-bit shift register, 32-bit Multiplicand Register
  - Booth's algorithm to handle signed multiplies
  - There are algorithms that calculate many bits of multiply per cycle (see exercises 4.36 to 4.39 in COD)
- Shifter: well...
- What's Missing from MIPS is Divide & Floating Point Arithmetic: Next time the Pentium Bug

Recap: VHDL combinational example

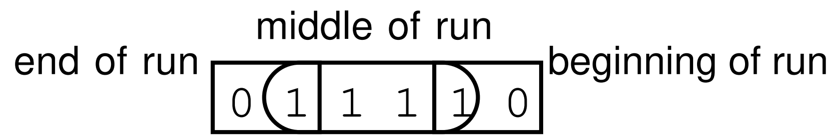
```
ENTITY nandnor is
  GENERIC (delay: TIME := 1ns);
  PORT (a,b: IN VLBIT; x,y: OUT VLBIT)
END nandnore

ARCHITECTURE behavioral OF nandnor is
BEGIN
  x <= a NOR b AFTER delay;
  y <= a NAND b AFTER delay;
END behavioral;
```

Review: MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg (shift right), (0-bit Multiplier reg)



Review: Booth's Algorithm Insight

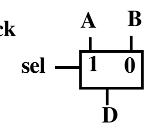


Current Bit	Bit to the Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

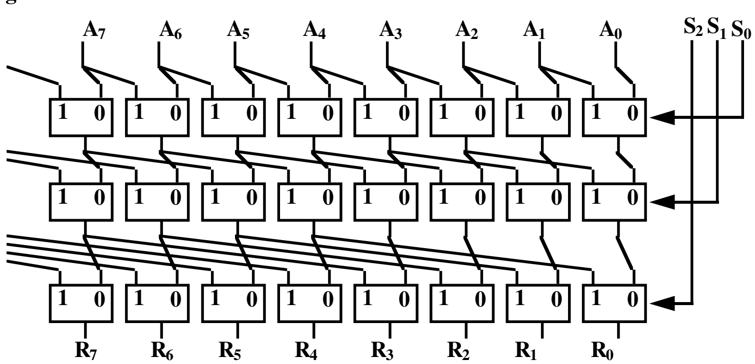
Originally for Speed since shift faster than add for his machine

Review: Combinational Shifter from MUXes

Basic Building Block



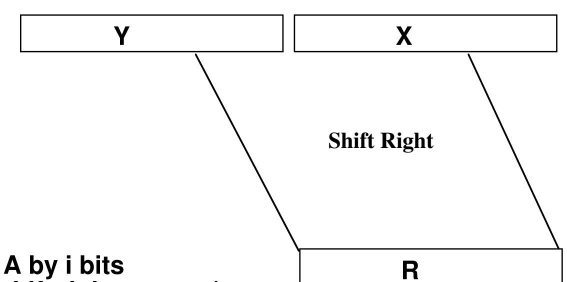
8-bit right shifter



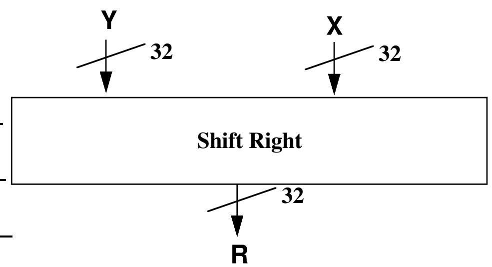
- What comes in the MSBs?
- How many levels for 32-bit shifter?
- What if we use 4-1 Muxes ?

Funnel Shifter

Instead Extract 32 bits of 64.

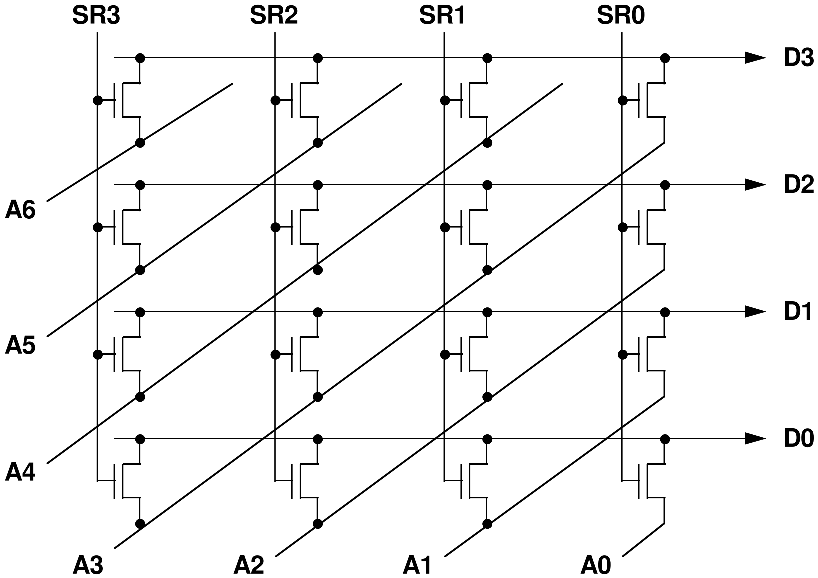


- Shift A by i bits (sa= shift right amount)
- Logical: Y = 0, X=A, sa=i
- Arithmetic? Y = _, X=_, sa=_
- Rotate? Y = _, X=_, sa=_
- Left shifts? Y = _, X=_, sa=_



Barrel Shifter

Technology-dependent solutions: transistor per switch



Divide: Paper & Pencil

```
                    1001      Quotient
               ___________
Divisor 1000  |  1001010      Dividend
                -1000
                    10
                    101
                    1010
                   -1000
                      10      Remainder (or Modulo result)
```

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

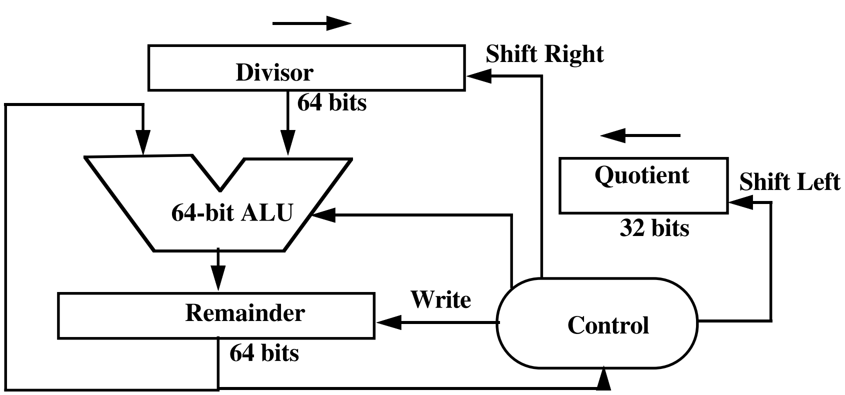
Dividend = Quotient x Divisor + Remainder

=> | Dividend | = | Quotient | + | Divisor |

3 versions of divide, successive refinement

DIVIDE HARDWARE Version 1

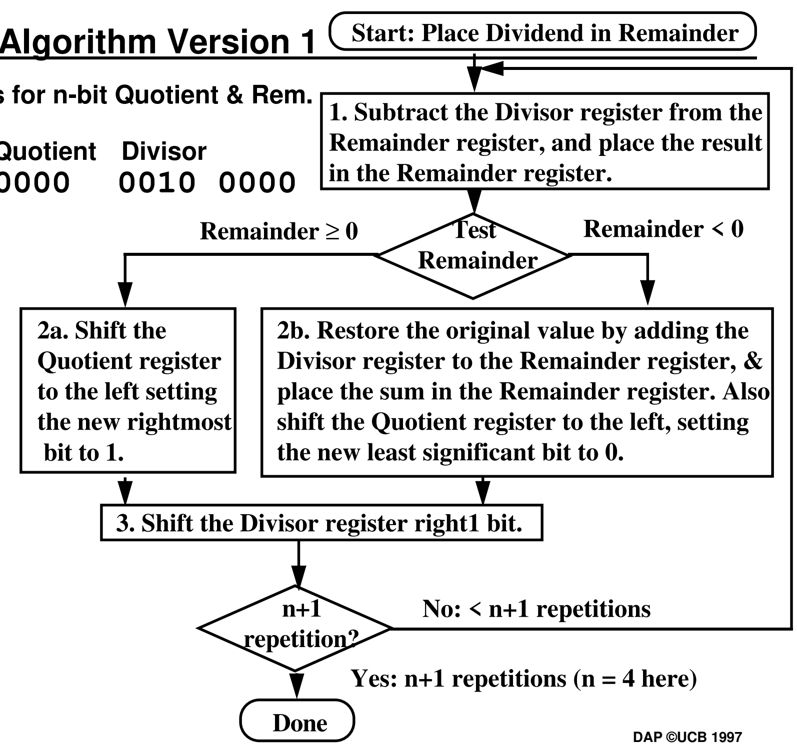
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 1

- Takes n+1 steps for n-bit Quotient & Rem.

Remainder	Quotient	Divisor
0000 0111	0000	0010 0000

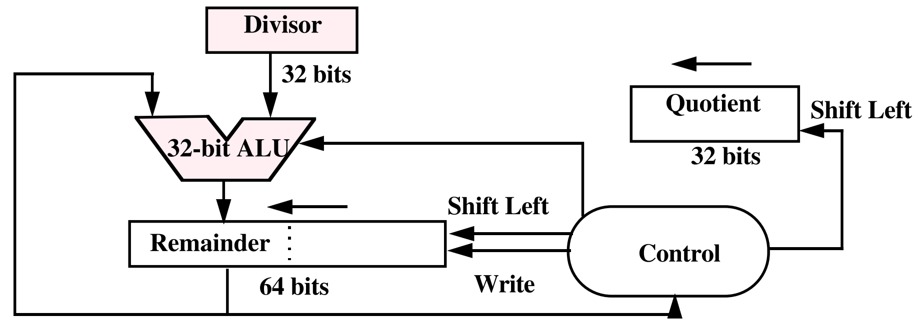


Observations on Divide Version 1

- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise too big)
  => switch order to shift first and then subtract, can save 1 iteration

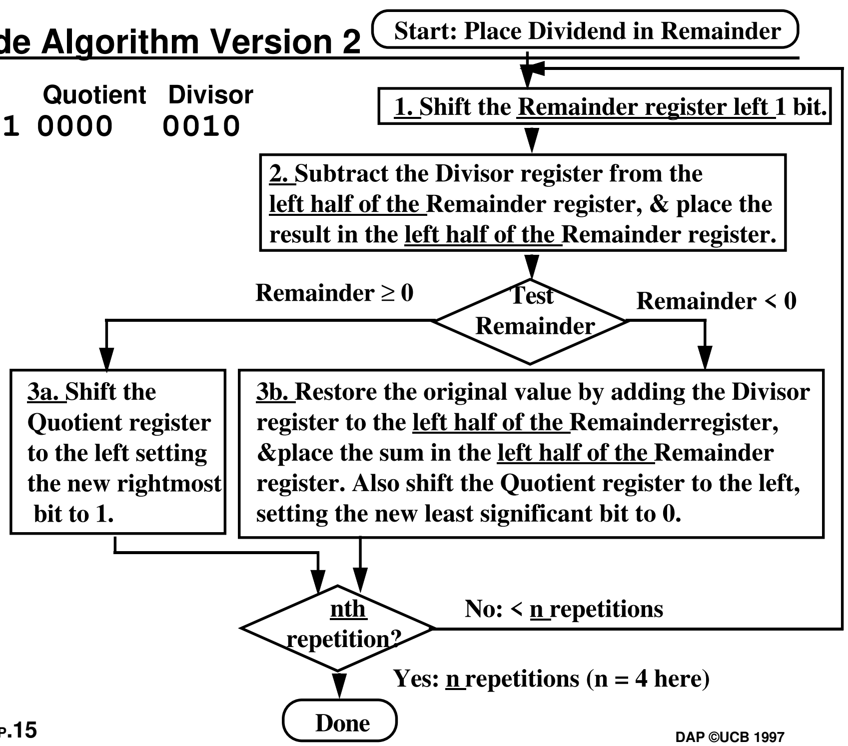
DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



Divide Algorithm Version 2

Remainder	Quotient	Divisor
0000 0111	0000	0010

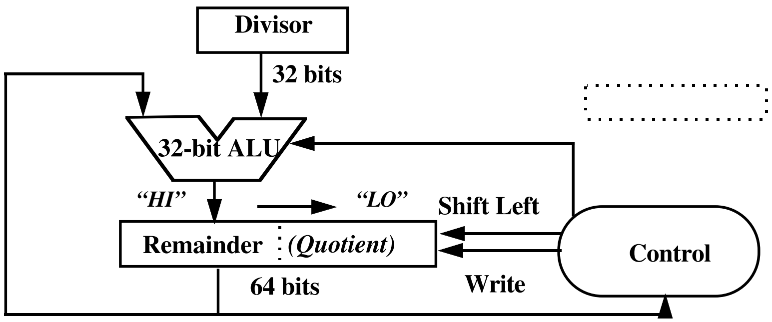


Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

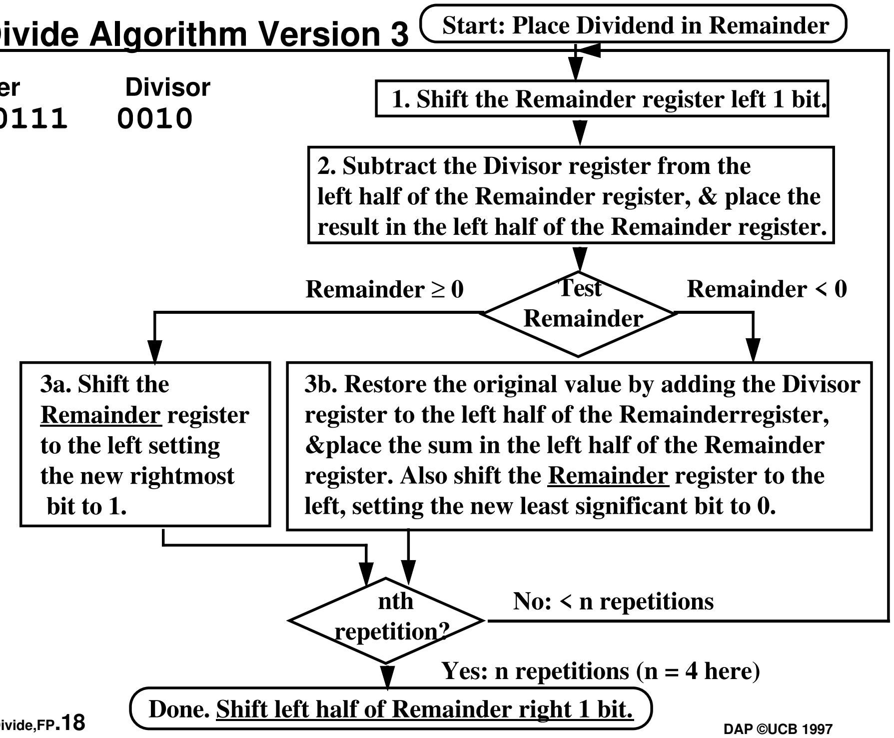
DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



Divide Algorithm Version 3

Remainder	Divisor
0000 0111	0010



Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree e.g., $-7 \div 2 = -3$, remainder $= -1$
- Possible for quotient to be too large: if divide 64-bit interger by 1, quotient is 64 bits ("called saturation")

Administrative Matters

- Midterm Wednesday 10/1
- Revise deadlines (so that can dicuss in section):
  Book exercises Fridays at noon
  Labs on Tuesdays at noon
- Finishing Chapter 4 today, moving to Chapter 5 Friday
- ALU + notebook, teams of 2 next lab

Floating-Point

- What can be represented in N bits?
- Unsigned: 0 to 2^N
- 2s Complement: -2^{N-1} to $2^{N-1} - 1$
- 1s Complement: $-2^{N-1}+1$ to $2^{N-1}-1$
- Excess M: $-M$ to $2^N - M - 1$
  - $(E = e + M)$
- BCD: 0 to $10^{N/4} - 1$
- But, what about?
  - very large numbers? 9,349,398,989,787,762,244,859,087,678
  - very small number? 0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π

Recall Scientific Notation

decimal point, exponent

$\underline{6.02} \times \underline{10}^{23}$ $\qquad 1.673 \times 10^{-24}$

Mantissa, radix (base)

Exponent: Sign, magnitude

Mantissa: Sign, magnitude

IEEE F.P. $\pm 1.M \times 2^{e - 127}$

- Issues:
  - Arithmetic (+, -, *, /)
  - Representation, Normal form
  - Range and Precision
  - Rounding
  - Exceptions (e.g., divide by zero, overflow, underflow)
  - Errors
  - Properties (negation, inversion, if $A \neq B$ then $A - B \neq 0$)

Review from Prerequisties: Floating-Point Arithmetic

Representation of floating point numbers in IEEE 754 standard:

single precision

	1	8	23
sign	S	E	M
		exponent: excess 127 binary integer	mantissa: sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M

actual exponent is
e = E - 127

$0 < E < 255$

$N = (-1)^S \, 2^{E-127} \, (1.M)$

$0 = 0\ 00000000\ 0 \ldots 0 \qquad -1.5 = 1\ 01111111\ 10 \ldots 0$

Magnitude of numbers that can be represented is in the range:

$$2^{-126}(1.0) \quad \text{to} \quad 2^{127}(2 - 2^{23})$$

which is approximately:

$$1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}$$

(integer comparison valid on IEEE Fl.Pt. numbers of same sign!)

Basic Addition Algorithm

For addition (or subtraction) this translates into the following steps:

(1) compute Ye - Xe *(getting ready to align binary point)*

(2) right shift Xm that many positions to form $Xm\ 2^{Xe-Ye}$

(3) compute $Xm\ 2^{Xe-Ye} + Ym$

if representation demands normalization, then a normalization step follows:

(4) left shift result, decrement result exponent (e.g., 0.001xx…)
right shift result, increment result exponent (e.g., 101.1xx…)
continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(5) doubly biased exponent must be corrected:

Xe = 7	Xe = 1111	= 15	= 7 + 8
Ye = -3	Ye = 0101	= 5	= -3 + 8
Excess 8	10100	20	4 + 8 + 8

extra subtraction step of the bias amount

(6) if result is 0 mantissa, may need to set the exponent to zero by special step

Extra Bits for rounding

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

How many extra bits?

IEEE: As if computed the result exactly and rounded.

Addition:

1.xxxxx	1.xxxxx	1.xxxxx
+ 1.xxxxx	0.001xxxxx	0.01xxxxx
1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy
post-normalization	pre-normalization	pre and post

- *Guard Digits*: digits to the right of the first p digits of significand to guard against loss of digits – can later be shifted left into first P places during normalization.
- Addition: carry-out shifted in
- Subtraction: borrow digit and guard
- Multiplication: carry and guard, Division requires guard

Rounding Digits

normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

E.g., B = 10, p = 3:

	0	2	1.69	= 1.6900 * $10^{2\text{-bias}}$
-	0	0	7.85	= - .0785 * $10^{2\text{-bias}}$
	0	2	1.61	= 1.6115 * $10^{2\text{-bias}}$

one round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit

IEEE Standard:

four rounding modes: round to nearest (default)
round towards plus infinity
round towards minus infinity
round towards 0

round to nearest:
round digit < B/2 then truncate
> B/2 then round up (add 1 to ULP: unit in last place)
= B/2 then round to nearest even digit

it can be shown that this strategy minimizes the mean error introduced by rounding

Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
+  0 .  0   0  X . . .  X    X  X  S
                             X  X  S
```

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0   0  X . . .  X    X  X  0
                             X  X  0
```

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0   0  X . . .  X    X  X  1
```

generates a borrow

Rounding Summary:

Radix 2 minimizes wobble in precision

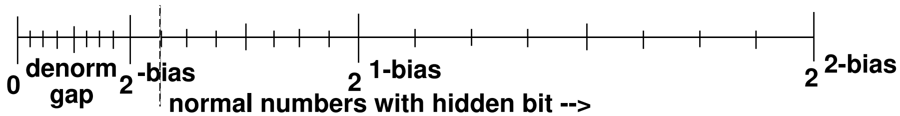
Normal operations in +,-,*,/ require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed

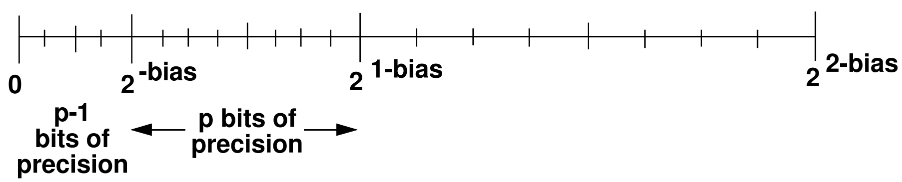
Denormalized Numbers



B = 2, p = 4

The gap between 0 and the next representable number is much larger than the gaps between nearby representable numbers.

IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike.



same spacing, half as many values!

NOTE: PDP-11, VAX cannot represent subnormal numbers. These machines underflow to zero instead.

Infinity and NaNs

result of operation *overflows*, i.e., is larger than the largest number that can be represented

overflow is not the same as divide by zero (raises a different exception)

+/- infinity | S | 1 . . . 1 | 0 . . . 0 |

It may make sense to do further computations with infinity
e.g., X/0 > Y may be a valid comparison

Not a number, but not infinity (e.q. sqrt(-4))
invalid operation exception (unless operation is = or ≠)

NaN | S | 1 . . . 1 | non-zero | ← HW decides what goes here

NaNs propagate: f(NaN) = NaN

Pentium Bug

- Pentium FP Divider uses algorithm to generate multiple bits per steps
  - FPU uses most significant bits of divisor & dividend/remainder to guess next 2 bits of quotient
  - Guess is taken from lookup table: -2, -1,0,+1,+2 (if previous guess too large a reminder, quotient is adjusted in subsequent pass of -2)
  - Guess is multiplied by divisor and subtracted from remainder to generate a new remainder
  - Called SRT division after 3 people who came up with idea
- Pentium table uses 7 bits of remainder + 4 bits of divisor = 2^{11} entries
- 5 entries of divisors omitted: 1.0001, 1.0100, 1.0111, 1.1010, 1.1101 from PLA (fix is just add 5 entries back into PLA: cost $200,000)
- Self correcting nature of SRT => string of 1s must follow error
  - e.g., 1011 1111 1111 1111 1111 1011 1000 0010 0011 0111 1011 0100 (2.99999892918)
- Since indexed also by divisor/remainder bits, sometimes bug doesn't show even with dangerous divisor value

Pentium bug appearance

- First 11 bits to right of decimal point always correct: bits 12 to 52 where bug can occur (4th to 15th decimal digits)
- FP divisors near integers 3, 9, 15, 21, 27 are dangerous ones:
  - $3.0 > d \geq 3.0 - 36 \times 2^{-22}$, $9.0 > d \geq 9.0 - 36 \times 2^{-20}$
  - $15.0 > d \geq 15.0 - 36 \times 2^{-20}$, $21.0 > d \geq 21.0 - 36 \times 2^{-19}$
- 0.333333 x 9 could be problem
- In Microsoft Excel, try (4,195,835 / 3,145,727) * 3,145,727
  - = 4,195,835 => not a Pentium with bug
  - = 4,195,<u>579</u> => Pentium with bug (assuming Excel doesn't already have SW bug patch)
  - Rarely noticed since error in 5th significant digit
  - Success of IEEE standard made discovery possible: ≈ all computers should get same answer

Pentium Bug Time line

- June 1994: Intel discovers bug in Pentium: takes months to make change, reverify, put into production: plans good chips in January 1995 4 to 5 million Pentiums produced with bug
- Scientist suspects errors and posts on Internet in September 1994
- Nov. 22 Intel Press release: "Can make errors in 9th digit ... Most engineers and financial analysts need only 4 of 5 digits. Theoretical mathematician should be concerned. ... So far only heard from one."
- Intel claims happens once in 27,000 years for typical spread sheet user:
  - 1000 divides/day x error rate assuming numbers random
- Dec 12: IBM claims happens once per 24 days: Bans Pentium sales
  - 5000 divides/second x 15 minutes = <u>4,200,000</u> divides/day
  - IBM statement: http://www.ibm.com/Features/pentium.html
  - Intel said it regards IBM's decision to halt shipments of its Pentium processor-based systems as unwarranted.

Pentium jokes

- Q: What's another name for the "Intel Inside" sticker they put on Pentiums?

  A: Warning label.

- Q: Have you heard the new name Intel has chosen for the Pentium?

  A: the Intel Inacura.

- Q: According to Intel, the Pentium conforms to the IEEE standards for floating point arithmetic. If you fly in aircraft designed using a Pentium, what is the correct pronunciation of "IEEE"?

  A: Aaaaaaaiiiiiiiiieeeeeeeeeeeee!

- TWO OF TOP TEN NEW INTEL SLOGANS FOR THE PENTIUM

  9.9999973251 It's a FLAW, Dammit, not a Bug

  7.9999414610 Nearly 300 Correct Opcodes

Pentium conclusion: Dec. 21, 1994 $500M write-off

"To owners of Pentium processor-based computers and the PC community:

We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw.

The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect.

What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns.

We want to resolve these concerns.

Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer. Just call 1-800-628-8686."

Sincerely,

Andrew S. Grove	Craig R. Barrett	Gordon E. Moore
President /CEO	Executive Vice President &COO	Chairman of the Board

Summary

- Pentium: Difference between bugs that board designers must know about and bugs that potentially affect all users
  - Why not make public complete description of bugs in later category?
  - $200,000 cost in June to repair design
  - $500,000,000 loss in December in profits to replace bad parts
  - How much to repair Intel's reputation?
- <u>What is technologists responsibility in disclosing bugs?</u>
- Bits have no inherent meaning: operations determine whether they are really ASCII characters, integers, floating point numbers
- Divide can use same hardware as multiply: Hi & Lo registers in MIPS
- Floating point basically follows paper and pencil method of scientific notation using integer algorithms for multiply and divide of significands
- IEEE 754 requires good rounding; special values for NaN, Infinity