Computer Arithmetic

Introduction to Computer Architecture
David Black-Schaffer

Contents

• Binary numbers
• Addition, carries, and multiplication
• Signed numbers
• Fixed point
• Floating point

Material that is not in this lecture

Readings from the book
  • Division (3.4)

The book has excellent descriptions of this topic.
Please read the book before watching this lecture.
The reading assignment is on the website.

(Don’t forget: the assigned reading may include details or bits and pieces
that I don’t cover in the lecture. You’re responsible for that as well on the
exam.)
Binary numbers
(And some other useful bases)

Why use binary numbers?

- Computer use binary numbers because:
  - Easy to build circuits: 1=1V, 0=0V (in the past 3.3V or 5V)
  - Easy to build complex circuits with switches (transistors)

- You could use multiple voltage levels...
  - 1=1V, 2=2V, 3=3V, etc.
  - But noise would kill you
  - Digital logic is noise tolerant:
    - No noise: 1 + 0 \rightarrow 1
    - With noise: 0.9 + 0.4 \rightarrow 1, not 1.3
  - Analog circuits carry noise through:
    - 1.4V + 3.4V \rightarrow 4.8V (closer to 5 than 4!)

- What's interesting about computer arithmetic is how much we can do with a limited number of bits

Radix numbers (binary)

Decimal (Base = 10)

<table>
<thead>
<tr>
<th>100's place</th>
<th>10's place</th>
<th>1's place</th>
<th>1/10 place</th>
<th>1/100 place</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_0</td>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
</tr>
</tbody>
</table>

Binary (Base = 2)

<table>
<thead>
<tr>
<th>2^n place</th>
<th>2's place</th>
<th>1's place</th>
<th>1/2 place</th>
<th>1/4 place</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_0</td>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
</tr>
</tbody>
</table>

Number\_decimal = \sum_{n=0}^{\infty} d_n \times 10^n

Number\_binary = \sum_{n=0}^{\infty} d_n \times 2^n

\begin{align*}
2.25 &= 2 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} \\
   &= 2 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} \\
   &= 2 \times 2^0 + 0 \times 2^{-1} + 2 \times 2^{-2} + 1 \times 2^{-3} \\
   &= 10.011_{\text{binary}}
\end{align*}
Useful bases for computers

- **Base-10, decimal**
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Human-readable

- **Base-2, binary**
  - Digits: 0, 1
  - Machine-readable

- **Base-8, octal**
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7

- **Base-16, hexadecimal**
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

**Decimal:** 10, 11, 12, 13, 14, 15

**Examples**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
</tbody>
</table>

Octal and hex are exactly 2 and 3 bits per digit.

**LSBs and MSBs**

- **LSB** = Least Significant Bit
- **MSB** = Most Significant Bit

**Example:**

1 1 1 1 1 1 1 1 0 0 0 1

MSB – largest value digit

LSB – lowest value digit
### Binary Addition

<table>
<thead>
<tr>
<th>Binary Addition</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0 + 0 = 0)</td>
<td>1011000, = 180&lt;sub&gt;10&lt;/sub&gt;</td>
</tr>
<tr>
<td>(0 + 1 = 1)</td>
<td>+ 110011, = 55&lt;sub&gt;10&lt;/sub&gt;</td>
</tr>
<tr>
<td>(1 + 1 = 10)</td>
<td>11110001, = 243&lt;sub&gt;10&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

### Binary Multiplication

<table>
<thead>
<tr>
<th>Binary Multiplication</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0 \times 0 = 0)</td>
<td>101100   multiplicand</td>
</tr>
<tr>
<td>(0 \times 1 = 0)</td>
<td>x 1011   multiplier</td>
</tr>
<tr>
<td>(1 \times 0 = 0)</td>
<td>101100   Partial products</td>
</tr>
<tr>
<td>(1 \times 1 = 1)</td>
<td>101100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shift and add</th>
<th>111100100 product</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 111100100</td>
<td>111100100 product</td>
</tr>
</tbody>
</table>

**Addition, carries, and multiplication**
Multiplication designs

- **Parallel Multiplication**
  - N x N multiplication produces 2N output bits
  - Question: How does this work with 32 bit registers in MIPS?
  - Answer: Special H and L registers for the two 32 bit halves
- **Serial Multiplication**
  - One adder used repeatedly
  - Lots and lots of adders
  - Double precision product
  - Multiplication takes a lot of area: can trade off area and time

Serial multiplication 1

- **Multiplicand**
- **Multiplier**
- **Partial products**
- **Product**

Serial multiplication 2

- **Multiplicand**
- **Multiplier**
- **Partial products**
- **Product**

Check LSB and add if 1
Serial multiplication 3

0010 multiplicand
0000 multiplier
Partial products
00001010 product

Shift multiplicand left and multiplier right

Serial multiplication 4

0010 multiplicand
0000 multiplier
Partial products
00001010 product

Check LSB and add 1

Serial multiplication 5

0010 multiplicand
0000 multiplier
Partial products
00001010 product

Shift multiplicand left and multiplier right
Serial multiplication 6

\[
\begin{array}{c}
0010 \times 0101 \\
\hline
0000 \quad 1010 \\
\hline
0000 \quad 0000 \\
\end{array}
\]

Check LSB and add if 1

Serial multiplication 7

\[
\begin{array}{c}
0010 \times 0101 \\
\hline
0000 \quad 1010 \\
\hline
0000 \quad 0000 \\
\end{array}
\]

Shift multiplicand left and multiplier right

Serial multiplication 8

\[
\begin{array}{c}
0010 \times 0101 \\
\hline
0000 \quad 1010 \\
\hline
0000 \quad 0000 \\
\end{array}
\]

Check LSB and add if 1
Serial multiplication 9

- Serial multiplication does each partial product one after another.
- Each step is shifting and adding if the LSB is one.
- Only need 1 adder, but many steps

Parallel multipliers

- Build \( n \) adders and everything at the same time
- Faster, but much larger

Building multipliers

- Multipliers used to be a real problem to build
- These days the question is how fast, not how big

---

© 2012 David Black-Schaffer
Design tradeoffs in multipliers

- Normalized to 1992 design. (E.g., what would it be like if you built it in the 1992 technology)
- Absolute. (E.g., how it did in its own technology)

Tradeoffs change: today area (gates) is free, but speed and design are expensive.

Summary of addition and multiplication

- Calculations are the same as base 10
  - E.g., carries and multiplication
- Addition can be implemented very quickly in circuits
- Multiplication is much larger, but can be done fast
- Subtraction with unsigned numbers is messy
  (see book)
- Division is always messy
  (see book)
- Next step: signed numbers
Signed numbers

How to represent signed numbers

- Three standard representations
  - Signed magnitude
  - One's complement
  - Two's complement

- In all of these the MSB is also the sign bit
  (1=negative)

- No one uses one's complement so we will skip it
- Everyone uses two's complement for integers
- Signed magnitude is used in floating point

Signed magnitude

- Simplest idea: use the first bit to indicate +/-
The problems with signed magnitude

- We must handle sign and magnitude separately
  - If A negative and B negative, A+B -> negative A + B
  - If A positive and B positive, A+B -> positive A + B
- Far more messy for subtraction:

<table>
<thead>
<tr>
<th>Adding</th>
<th>Subtracting</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A+B]</td>
<td>[A-B]</td>
</tr>
<tr>
<td>(A) + (B)</td>
<td>(A) + (B)</td>
</tr>
<tr>
<td>(A) + (B)</td>
<td>(A) + (B)</td>
</tr>
<tr>
<td>(A) + (B)</td>
<td>(A) + (B)</td>
</tr>
<tr>
<td>(A) + (B)</td>
<td>(A) + (B)</td>
</tr>
<tr>
<td>(A) + (B)</td>
<td>(A) + (B)</td>
</tr>
</tbody>
</table>

Two's complement

- Represent negative numbers with their complements
  - Complement of A -> A' = 2^n - A
  - E.g., for 4 bits (n=4): Complement of 0001 = 2^4 - 0001 = 10000 - 0001 = 1111
  - 1111 = -1, which is the complement of +1
- Two's complement negation:
  - Invert and add 1
- From earlier lecture:
  - Basic idea: biggest digit is negative, all others are positive
  - 1000 = -1*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = -8
  - Numbers range from -2^(n-1) to 2^(n-1)-1

Two's complement subtraction

- A-B = A + (-B)
- A + (1B + 1) Two's complement: -B = 1B+1
- A + (1B + 1)
  - To compute A-B
    - Invert B (change all 1s to 0s and 0s to 1s)
    - Add 1 and B
    - Add in 1
  - Example:
    \[
    \begin{align*}
    5_{10} - 6_{10} & = -1_{10} \\
    0101, -0110 & = 1111_2 \\
    0101, +1001, +1 & = 1111_2
    \end{align*}
    \]
Two's complement examples

<table>
<thead>
<tr>
<th>Question: Did we get an overflow?</th>
<th>Answer: No, 8 and +3 are in the valid range.</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3 0011</td>
<td>-2 1110</td>
</tr>
<tr>
<td>+4 0100</td>
<td>+ -6 +1010</td>
</tr>
<tr>
<td>+7 0111</td>
<td>-8 1000</td>
</tr>
<tr>
<td>+ -7 +1001</td>
<td>+ -3 +1101</td>
</tr>
<tr>
<td>-3 1101</td>
<td>+3 0011</td>
</tr>
</tbody>
</table>

• Numbers range from \(-1 \times 2^{n-1}\) to \(2^{n-1} - 1\)
• 4-bit number ranges is: -8 to +7

Overflow in two's complement

• Different from unsigned numbers (just carry out)
• Overflow means the number cannot be represented
• In two's complement addition:
  • Numbers of opposite signs cannot overflow
  • Overflow if numbers are of the same sign and result is of a different sign
  • Rule: carry in to the sign bits ≠ carry out of the sign bits
• In both examples, carry in to the sign bits == carry out

Carry in to the sign bits == carry out from the sign bits, no overflow

Comparing numbers

<table>
<thead>
<tr>
<th>Unsigned operands</th>
<th>Two's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Z: equality/inequality</td>
<td>• Z: equality/inequality</td>
</tr>
<tr>
<td>• C=0: A=B</td>
<td>• C: no meaning</td>
</tr>
<tr>
<td>• C=1: A&gt;B</td>
<td>• $S$ and $O$</td>
</tr>
<tr>
<td>• $S$: no meaning</td>
<td>• $S \text{ XOR } O = 0: A=B$</td>
</tr>
<tr>
<td>• $O$: no meaning</td>
<td>• $S \text{ XOR } O = 1: A=B$</td>
</tr>
</tbody>
</table>

• To compare A and B, do $A-B$
• Then test the following: $ZERO, CARRY, SIGN, OVERFLOW$
Signed number summary

- Sign magnitude has:
  - Two zeros (waste of bits)
  - Complicated rules for subtraction

- Two's complement has:
  - One zero
  - Simple rules for subtraction

- All signed integers are in two's complement

- But what about non-integers?

Non-integers: fixed and floating point numbers
Non-integer numbers

- In decimal we use a decimal point: $12.75_{10} = 1\times10^1 + 2\times10^0 + 7\times10^{-1} + 5\times10^{-2}$
- In binary we can use a binary point or a radix point: $12.75_{10} = 1\times2^4 + 2\times2^3 + 7\times2^1 + 5\times2^0$

- This is fixed-point: $abc.def = a\times2^n + b\times2^{n-1} + c\times2^{n-2} + d\times2^{n-3} + e\times2^{n-4} + f\times2^{n-5}$
- Where do we put the binary point?
  - For example, if we have 6 bits to work with:
    - xxx.xyyy $\rightarrow$ max = 111.1111, min = 000.0001
    - xxxx.y $\rightarrow$ max = 11111.11, min = 00000.0  

Math is almost the same in fixed-point

<table>
<thead>
<tr>
<th>011.010</th>
<th>011.010</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 010.011</td>
<td>\times 001.110</td>
</tr>
<tr>
<td>111.101</td>
<td>000000</td>
</tr>
<tr>
<td>Addition and subtraction (two's complement) are the same.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0001011011100</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 101.110</td>
</tr>
</tbody>
</table>

To convert back to 3 bit + 3 bit fixed point:
1) Put in the binary point
2) Round the result

But there's a problem...

- Fixed point works great if we know the range
  - E.g., we know the numbers will range from +16 to -16 or 1.5 to 0 $\rightarrow$ can choose the right binary point
  - Common for signal processing (DSPs in cell phones, etc.)
  - But we usually don’t know how large our numbers will be

- Floating point solves this:
  - Use some of the bits to choose the binary point
  - Example: if we have 20 bits, use 4 bits to specify where to put the binary point
    - 0000 0000 0000 | 0000
    - 0000 0000 0000 | 0000
    - 0000 0000 0000 | 0000
    - 1100 (12.5)
    - 0010 (4.2)
Floating point

- Use some of the bits to choose the binary point
  - Have large or small numbers by moving the binary point
  - More efficient use of the bits!

1.23456 x 2^78

"Significant digits" determine the precision. Mantissa or Fraction

"Implied base" which is usually 2. We don’t store the base, but we agree upon it.

The IEEE floating point standard

- Developed ~1980 to standardize floating point math, make it portable, and maximize quality
- This is very hard to get right: GPUs only got standardized in 2008

(-1)^s x (1.f) x 2^(e-127)

(Which is not at all obvious.)

Floating point details

- Sign s
  - The whole number has a sign: positive / negative
- Exponents e
  - Unsigned value, but we bias it by -127 to get positive and negative values
  - Makes comparison easy (just look at the magnitude)
- Fraction f
  - Integer value
  - Regarded as a binary number with the binary point to the left of the MSB

(-1)^s x (1.f) x 2^(e-127)
Floating point examples

Normalization

- We have an implied 1 before the fraction
  - This forces only a single representation of each number
  
  \((-1)^3 \times (1.f) \times 2^{(6 - 127)}\)

- Example without normalization:
  - 3 bit fraction: \((-1)^2 \times 0.f_f_f \times 2^2\)

- How can we represent 2?
  
  \(- (\frac{1}{2})^3 \times 0.100 \times 2^{(3-127)} = \frac{1}{8} \times 0 = 2 \)
  
  \(- (\frac{1}{2})^3 \times 0.010 \times 2^{(3-127)} = \frac{1}{8} \times 4 = 2 \)
  
  \(- (\frac{1}{2})^3 \times 0.001 \times 2^{(3-127)} = \frac{1}{8} \times 16 = 2 \)

- This is a waste of bits to represent the same number many different ways!

Question:
With a 33-bit mantissa and no normalization, how many ways are there to represent each number?

Answer:
23. That's a lot of waste.
Floating point number spacing

- Example 7-bit floating point number:
  - 1 bit sign, 3 bits exponent, 3 bits fraction = \((-1)^0 \times 0.110\) 2

  Non-uniform coverage (linear scale)

  Logarithmic coverage (log-log scale)

  Question: Why are the numbers distributed logarithmically?

  Answer: 2^n is a logarithmic spacing!

Floating point addition

- Example: 2.1 \times 10^{12} + 9.2 \times 10^{10}

  1. Re-write to match exponents by shifting the mantissas:
     2.1 \times 10^{12} = 0.21 \times 10^{13}

  2. Add mantissas:
     \(2.1 \times 10^{12} + 0.92 \times 10^{12}\)

  3. Round to fit in the number of bits we have:
     \(2.19 \times 10^{12}\)

  What did we do?
  - Shift (multiply) the smaller number to match the exponents
  - Add the mantissas
  - Round at the end to fit in to the number of bits

  Much more complex than integer math

  May lose precision of smaller numbers when added.
  (Fast if f(t), but big implications for numerical stability when dividing.)

Floating point multiplication

- Example: 2.1 \times 10^{12} \times 9.2 \times 10^{10}

  1. Multiply mantissas:
     \(2.1 \times 9.2 = 19.32\)

  2. Add exponents:
     \(12 + 10 = 22\)

  3. Round (and shift the decimal place) to fit in the number of bits we have:
     \(1.9 \times 10^{23}\)

  What did we do?
  - Multiply the mantissas and add the exponents
  - Shift (normalize) to put the decimal point in the right place
  - Round at the end to fit in to the number of bits

  Much simpler than floating point addition

  Still needs a big multiplier (23 bits for floats, 53 for doubles)
Losing precision in floating point

- Consider:
  - Big + Small = Big
  - \(2.3 \times 10^{20} + 9.2 \times 10^{5} = 2.3 \times 10^{20} + 0.0000000000000092 \times 10^{20} \approx 2.3 \times 10^{20}\)
- Now what happens if I later subtract Big?
  - Do I get Small back?
  - \((\text{Big} + \text{Small}) - \text{Big} = ?\)
  - \((\text{Big} + \text{Small}) - \text{Big} = \text{Big} - \text{Big} = 0\)
  - No, I get something very close to zero.
- Now what happens if I try to divide by the result?
  - \(x/0 = \ldots \text{bad}\)
  - Order of operations matters!
  - \((\text{Big} - \text{Big}) + \text{Small} = \text{Small}\)
- Good news: There are lots of truly excellent libraries that do the right thing for you.
  - This is why you do not write your own linear algebra code or FFT code

Non-integer number representation summary

- Fixed Point: xxxx yyy
  - Just the integer maths, but you specify how many bits for the fraction
  - Good if you know the range of your numbers beforehand
- Floating Point: (-1)^s \times \text{mantissa} \times 2^{\text{exponent}}
  - The fractional point is determined by the exponent
  - Mantissa = the fractional part
  - Exponent = determines where the fractional point is
  - Sign = positive or negative number (mantissa and exponent are not 2's complement)
  - Non-linear scale (more accuracy with smaller numbers)
  - Addition is complicated (need to shift to match exponents)
  - Multiplication is simple (just add exponents and multiple mantissas)
  - Division is not something you ever want to try to implement
- The real truth:
  - There are lots of nasty issues with floating point numbers. You can read about them on Wikipedia, but they will probably give you a headache.