CS 271
Computer Architecture & Assembly Language

Lecture 7

Binary Arithmetic, Byte Ordering

Float point representation

1/25/22, Tuesday

Oregon State University

Odds and Ends

- Grading 1 done
- Program 2 past due

Lecture Topics:

- Binary Arithmetic
- Byte Ordering
- Floating-point Representation

Binary Arithmetic
Byte Ordering

Arithmetic Operations

- The following examples use 8-bit twos-complement operands
  - Everything extends to 16-bit, 32-bit, n-bit representations.
  - What is the range of values for 8-bit operands? 2^8

unsigned: $0 \sim 2^8 - 1 \rightarrow 0 \sim 255$

signed: $-2^{8-1} \sim 2^{8-1} - 1 \rightarrow -128 \sim 127$

- The usual arithmetic operations can be performed directly in binary form with n-bit representations.

Binary Addition

- Specify result size (bits)
- Binary addition table:

+	0	1
0	0	1
1	1	10

- Use the usual rule of add and carry
  - With two operands, the carry bit is never greater than 1
  - 0+0+1=01, 0+1+1=10, 1+0+1=10, 1+1+1=11
- Example:

```
  00101101
+ 00011110
  01001011
```

- How does overflow occur?

Binary Subtraction

- Use the usual rules
  - Order of operands
  - Borrow and subtract
- Example:

```
    00101101
-   00011110
    --------
    00001111
```

- … or negate and add (-00011110 = 11100010)
- Example:

```
    00101101
+   11100010
    --------
    00001111
```

Verification

- Perform operation on binary operands
- Convert result to decimal
- Convert operands to decimal
- Perform operation on decimal operands
- [convert result to binary]
- Compare results

Binary Multiplication

- Usual algorithm

```
   45
×   5
-----
  225
```

```
        00101101
×            101
----------------
        00101101
+     00101101
----------------
      0011100001
```

128 + 64 + 32 + 1

= 225

Binary Multiplication

- Repeated addition

$$\begin{array}{r} 45 \\ \times \quad 5 \\ \hline \end{array}$$

$\rightarrow 45 + 45 + 45 + 45 + 45$

Binary Multiplication

45 x 5

- … or shift left (and add leftovers, if multiplier is not a power of 2)
- Check for overflow

45 x 2 = 90

45 | 101101

90 | 1011010

00101101

x 10

001011010

45 x 5

= 45 x 4 + 45

111

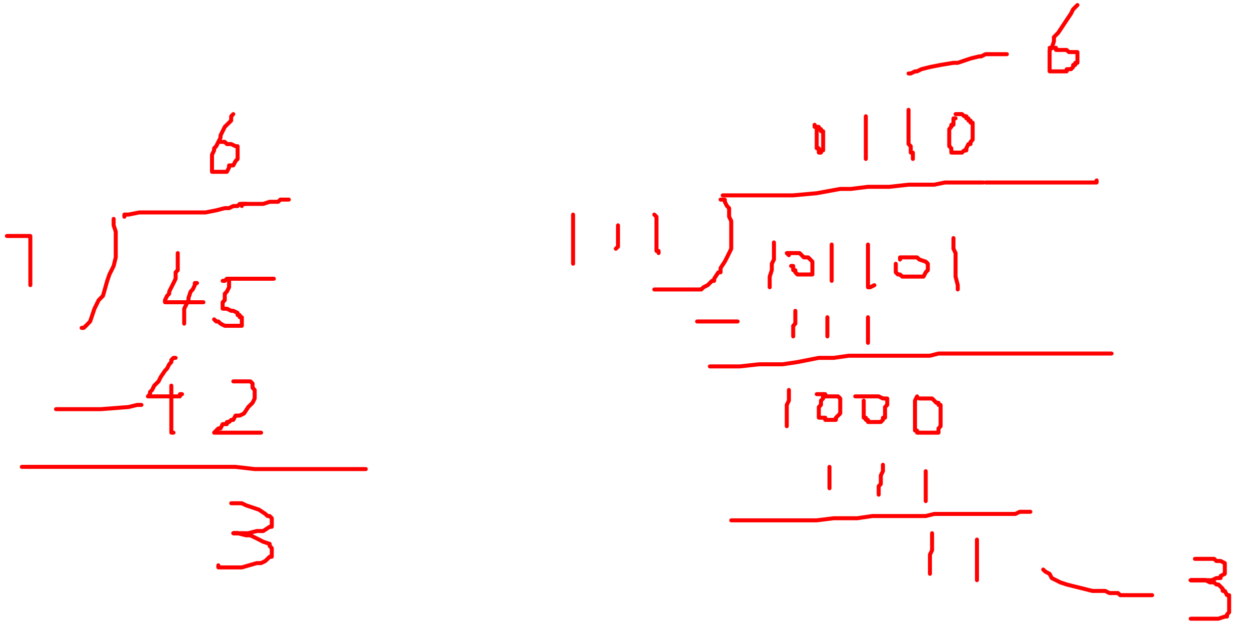
10110100

+ 101101

11100001

Binary Division

- Usual algorithm



Binary Division

- Repeated subtraction
  - count … until remainder is less than divisor

Binary Division

- If divisor is a power of 2, shift right and keep track of dropped bits
- Check for remainder

Arithmetic Operations

- Note: all of the integer arithmetic operations can be accomplished using <u>only</u>:
  - Add
  - Complement

- Addition: √
- Subtraction: complement and add
- Multiplication: repeated add
- Division: repeated subtract
- Comparison: non-destructive subtract

Byte-ordering

- When it takes more than one byte to represent a value
- Big-endian
  - Bytes are ordered left → right (most significant to least significant) in each word
  - Use in Motorola architectures (Mac) and others
- Little-endian
  - Bytes are ordered least significant to most significant in each word
  - Used in Intel architectures
- For both schemes
  - Within each byte, bit values are stored left → right (as usual)
  - Each character is one byte
  - Strings are stored in byte order
- Problem: communicating between architectures

Byte-ordering (big-endian)

- Example 32-bit integer: -1234

11111111	11111111	11111011	00101110
Byte3	Byte2	Byte1	Byte0

- Big-endian (big end first)
- Memory addresses

1004	1005	1006	1007
11111111	11111111	11111011	00101110
FF	FF	FB	2E
Byte3	Byte2	Byte1	Byte0

Byte-ordering (little-endian)

- Example 32-bit integer: -1234

11111111	11111111	11111011	00101110
Byte3	Byte2	Byte1	Byte0

- Little-endian (little end first)
- Memory addresses

1004	1005	1006	1007
00101110	11111011	11111111	11111111
2E	FB	FF	FF
Byte0	Byte1	Byte2	Byte3

Communication

- Internet Communication must be consistent across architectures.
- Network order is always big-endian

- More about this in your networking courses (CS/ECE 372)

Floating-point Representation

Floating-point values

- “decimal” means “base ten”
- “floating-point” means “a number with an integral part and a fraction part”
  - Sometimes called “real”, “float”
- Generic term for “decimal point” is “radix point”

Converting floating-point (decimal ←→ binary)

- Place values:

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

- Integral part Fraction part

- Example: 4.5 (decimal) = 100.1 (binary)

Converting floating-point (decimal ←→ binary)

- Example: 6.25 = 110.01
- Method:
  - 6 = 110 (Integral part: convert in the usual way)
  - .25 x 2 = 0.5 (Fraction part: successive multiplication by 2)
  - .5 x 2 = 1.0 (Stop when fraction part is 0)

- 110.01

Converting floating-point (decimal ←→ binary)

- Example: 6.2 ≈ 110.001100110011…
- Method:
  - 6 = 110 (Integral part: convert in the usual way)
  - .2 x 2 = 0.4
  - .4 x 2 = 0.8 (Fraction part: successive multiplication by 2)
  - .8 x 2 = 1.6 (Stop when fraction part repeats <u>or size is exceeded</u>)
  - .6 x 2 = 1.2
  - .2 x 2 = 0.4
- 110.0011 0011 0011 …

Floating-point: Internal Representation

- Some architecture handle the integer part and the fraction part separately
  - Slow

- Most use a completely different representation (IEEE standard) and a separate ALU (Floating-Point Unit)
  - Faster operations
  - For 32-bit representation:
    - Range of values is approximately -3.4×10^{38} … $+3.4 \times 10^{38}$
    - Limited precision approximately -1.4×10^{-45}… $+1.4 \times 10^{-45}$

IEEE 754 Standard

- Single-precision (32-bit)
- Double-precision (64-bit)
- Extended (80-bit)
- 3 parts
  - 1 sign bit
  - “biased” exponent (single: 8 bits,
double: 11 bits,
extended: 16 bits)
  - Normalized mantissa (single: 23 bits,
double: 52 bits,
extended: 63 bits)

32-bit Examples

- 6.25 in IEEE single precision is

sign exp mantissa

- 0 10000001 10010000000000000000000000
- 0100 0000 1100 1000 0000 0000 0000 0000
- =0x40C80000

- 6.2 in IEEE single precision is
- 0 10000001 10001100110011001100110
- 0100 0000 1100 0110 0110 0110 0110 0110
- =0x40C66666

32-bit Example: 6.25

- 6.25 in IEEE single precision is
- 6.25 (decimal) = 110.01 (binary)
- Move the radix point until a single 1 appears on the left, and multiply by the corresponding power of 2
- $=1.1001 \times 2^2$
- So the sign bit is 0 (positive)
- The "biased" exponent is 2 + 127 = 129 = 10000001
- And the "normalized" mantissa is 1001 (drop the 1, and zero-fill)
- 0 10000001 10010000000000000000000
- 0100 0000 1100 1000 0000 0000 0000 0000
- =0x40C80000

32-bit Example: 6.2

- 6.2 in IEEE single precision is
- 6.2 (decimal) = 110.001100110011… (binary)
- Move the radix point until a single 1 appears on the left, and multiply by the corresponding power of 2
- =1.10001100110011… x 2^2
- So the sign bit is 0 (positive)
- The “biased” exponent is 2 + 127 = 129 = 10000001
- And the “normalized” mantissa is 10001100110011… (drop the 1)
- 0 10000001   10001100110011001100110
- 0100 0000   1100   0110 0110 0110 0110 0110
- =0x40C66666
- Note that this representation is not exactly equal to 6.2

32-bit Example: 0xC1870000

- What decimal floating-point number is represented by 0xC1870000?
- 1100 0001 1000 0111 0000 0000 0000 0000
- 1 10000011 00001110000000000000000

- … so the sign is negative
- … the "unbiased" exponent is 131 – 127 = 4
- … and the "unnormalized" mantissa is 1.00001110000000000000000 (insert the 1 left of the radix point)
- Move the radix point 4 places to the right → 10000.111
- -10000.111 = -16.875

- We will continue to program the integer unit for now

- Floating-pointing programming … later

if(a==b) ✗ a, b are floats

if(a-b < 0.0000001) ✓