Part 4

Arithmetic Logic Unit

Negative Binary Integers

Have a positive attitude about negatives

Negative Binary Numbers

- When we write a negative number, we generally use a "-" as a prefix character
- However, binary numbers can only store ones and zeros

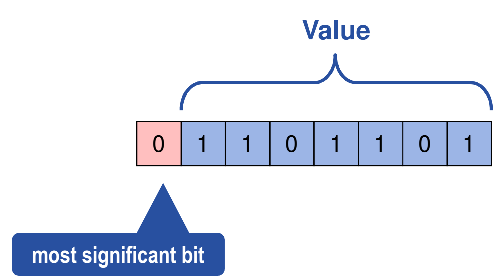
Negative Binary Numbers

- So, how we store a negative a number?
- When a number can represent both positive and negative numbers, it is called a *signed integer*
- Otherwise, it is *unsigned*

Signed Magnitude

- One approach is to use the most significant bit (msb) to represent the negative sign
- If positive, this bit will be a zero
- If negative, this bit will be a 1
- This gives a byte a range of -127 to 127 rather than 0 to 255

Signed Magnitude



Signed Magnitude: 13 and -13

Positive

0	0	0	0	1	1	0	1

Negative

1	0	0	0	1	1	0	1

Signed Magnitude Drawback #1

- When two numbers are added, the system needs to check and sign bits and act accordingly
- For example:
  - if both numbers are positive, add values
  - if one is negative subtract it from the other
  - etc…
- There are also rules for subtracting

Signed Magnitude Drawback #2

- Also, signed magnitude also can store a positive *and* negative version of zero
- Yes, there are two zeroes!
- Imagine having to write Java code like…

```
if (x == +0 || x == −0)
```


Oh noes! Two zeros?

+0

0	0	0	0	0	0	0	0

−0

1	0	0	0	0	0	0	0

1's Complement

- Rather than use a sign bit, the value can be made negative by *inverting* each bit
  - each 1 becomes a 0
  - each 0 becomes a 1
- Result is a "complement" of the original
- This is logically the same as subtracting the number from 0

Advantages / Disadvantages

- Advantages over signed magnitude
  - very simple rules for adding/subtracting
  - numbers are simply added: 5 - 3 is the same as 5 + -3
- Disadvantages
  - positive and negative zeros still exist
  - so, it's not a perfect solution

1's Complement: 13 and -13

Positive

0	0	0	0	1	1	0	1

Negative

1	1	1	1	0	0	1	0

1's Complement Has Two Zeros

+0

0	0	0	0	0	0	0	0

−0

1	1	1	1	1	1	1	1

2's Complement

- Practically all computers nowadays use *2's Complement*
- Similar to 1's complement, but after the number is inverted, 1 is added to the result
- Logically the same as:
  - subtracting the number from **2^n**
  - where *n* is the total number of bits in the integer

2's Complement Advantages

- Since negatives are subtracted from 2^n
  - they can simply be added
  - the extra carry 1 (if it exists) is discarded
  - this simplifies the hardware considerably since the processor only has to add
- The +1 for negative numbers…
  - makes it so there is only one zero
  - values range from -128 to 127

2's Complement: 13 and -13

Positive

0	0	0	0	1	1	0	1

Negative

Add 1

1	1	1	1	0	0	1	1

Just One Zero!

0

0	0	0	0	0	0	0	0

−1

1	1	1	1	1	1	1	1

Adding Binary Integers

1 + 1 = 10

Adding Binary Integers

- Computer's add binary numbers the same way that we do with decimal
- Columns are aligned, added, and "1's" are carried to the next column
- In computer processors, this component is called an *adder*

Adding Base 10 Numbers

	1	1		
	2	7	8	1
+	3	7	2	1
	6	5	0	2

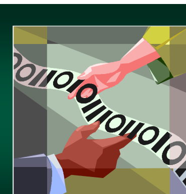
Adding Binary Example

			1	1		1	1		
182		1	0	1	1	0	1	1	0
51	+	0	0	1	1	0	0	1	1
233		1	1	1	0	1	0	0	1

Adding 2's Complement

		1	1	1	1			1	1	
91			0	1	0	1	1	0	1	1
-13	+		1	1	1	1	0	0	1	1
78			0	1	0	0	1	1	1	0

Extending Bytes



Converting from 8-bit to 16-bit and more

Extending Unsigned Integers

- Often in programs, data needs to moved to a integer with a larger number of bits
- For example, an 8-bit number is moved to a 16-bit representation

Extending Unsigned Integers

- For unsigned numbers is fairly easy – just add zeros to the left of the number
- This, naturally, is how our number system works anyway: 000456 = 456

Unsigned 13 Extended

0	0	0	0	1	1	0	1

↓

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

Extending Signed Integers

- When the data is stored in a signed integer, the conversion is a little more complex
- Simply adding zeroes to the left, will *convert a negative value to a positive one*
- Each type of signed representation has its own set of rules

2's Complement Extended Incorrectly

-13

1	1	1	1	0	0	1	1

243

↓

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

Sign Magnitude Extension

- In signed magnitude, the most-significant bit (msb) stores the negative sign
- The new sign-bit needs to have this value
- Rules:
  - copy the old sign-bit to the new sign-bit
  - fill in the rest of the new bits with zeroes – *including the old sign bit*

Sign Magnitude Extended: +77

0 1 0 0 1 1 0 1

0 0 0 0 0 0 0 0 | 0 1 0 0 1 1 0 1

Sign Magnitude Extended: -77

1 1 0 0 1 1 0 1

1 0 0 0 0 0 0 0 | 0 1 0 0 1 1 0 1

2's Complement Extension

- 2's Complement is very simple to convert to a larger representation
- Remember that we inverted the bits and added 1 to get a negative value
- Rule: copy the old most-significant bit to all the new bits

2's Complement Extended: +77

0 1 0 0 1 1 0 1

0 0 0 0 0 0 0 0 | 0 1 0 0 1 1 0 1

2's Complement Extended: -77

1 0 1 1 0 0 1 1

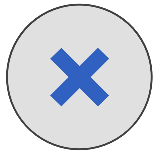
1 1 1 1 1 1 1 1 | 1 0 1 1 0 0 1 1

Multiplying Binary Numbers

11 × 11 = 1001

Multiplying Binary Numbers

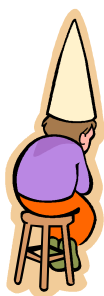
- Many processors today provide complex mathematical instructions
- However, the processor only needs to know how to add
- Historically, multiplication was performed with successive additions



Multiplying Scenario

- Let's say we have two variables: A and B
- Both contain integers that we need to multiply
- Our processor can *only* add (and subtract using 2's complement)
- How do we multiply the values?

Multiplying: The Bad Way



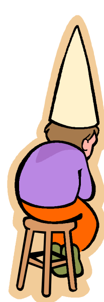
- One way of multiplying the values is to create a For Loop using one of the variables – A or B
- Then, inside the loop, continuously add the other variable to a running total

Multiplying: The Bad Way

```
total = 0;
for (i = 0; i < A; i++)
{
   total += B;
}
```


Multiplying: The Bad Way

- If one of the operands – A or B – is large, then the computation could take a long time
- This is incredibly inefficient
- Also, given that A and B could contain drastically different values – the number of iterations would vary
- Required time is not constant



Multiplying: The Best Way

- Computers can perform multiplication using long multiplication – just like *you* do
- The number of additions is then fixed to 8, 16, 32, 64 depending on the size of the integer
- The following example multiplies 2 unsigned 4-bit numbers

Unsigned Integer: 13 × 10

1 1 0 1
× 1 0 1 0

+

Unsigned Integer: 13 × 10

1 1 0 1
× 1 0 1 0

+

Unsigned Integer: 13 × 10

1 1 0 1
× 1 0 1 0

0 0 0 0

+

Unsigned Integer: 13 × 10

1 1 0 1
× 1 0 1 0

0 0 0 0

+

Unsigned Integer: 13 × 10

1 1 0 1
× 1 0 1 0

0 0 0 0
1 1 0 1

+

Unsigned Integer: 13 × 10

1 1 0 1
× 1 0 1 0

0 0 0 0
1 1 0 1

+

Unsigned Integer: 13 × 10

```
       1 1 0 1
 ×     1 0 1 0
 -------------
       0 0 0 0
     1 1 0 1
   0 0 0 0
 +
 -------------
```


Unsigned Integer: 13 × 10

```
       1 1 0 1
 ×     1 0 1 0
 -------------
       0 0 0 0
     1 1 0 1
   0 0 0 0
 +
 -------------
```


Unsigned Integer: 13 × 10

```
       1 1 0 1
 ×     1 0 1 0
 -------------
       0 0 0 0
     1 1 0 1
   0 0 0 0
 + 1 1 0 1
 -------------
```


Unsigned Integer: 13 × 10

```
          1 1 0 1
 ×        1 0 1 0
 ----------------
          0 0 0 0
        1 1 0 1
      0 0 0 0
 +  1 1 0 1
 ----------------
130 1 0 0 0 0 0 1 0
```


Multiplication Doubles the Bit-Count

- When two numbers are multiplied, the product will have twice the number of digits
- Examples:
  - 8-bit × 8-bit → 16-bit
  - 16-bit × 16-bit → 32-bit
- Often processors…
  - will store the result in the original bit-size
  - and flag an overflow if it does not fit

x86 Mathematics

Complex Math is Complex

Add & Subtract

- The Add and Subtract instructions take two operands and store the result in the second operand
- This is the same as the **+=** and **−=** operators used in Visual Basic .NET, C, C++, Java, etc…

Addition

Immediate, Register, Memory

```
ADD value, target
```

Register, Memory

Subtraction

Immediate, Register, Memory

```
SUB value, target
```

Register, Memory

Negate (2's complement)

```
NEG register
```


Example: Simple Add

Move value into RAX

```
MOV $17, %rax
ADD $2, %rax
```

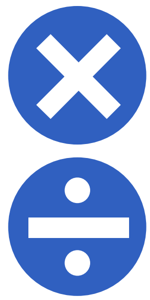
RAX += 2

Multiplication & Division

- Multiplication and division are far more complex
- The x86 treats this type of math differently than add/subtract
- It requires specific registers to be used: *RAX*, *RDX*

Multiplication Review

- Remember: when two *n* bit numbers are multiplied, result will be *2n* bits
- So…
  - two 8-bit numbers → 16-bit
  - two 16-bit numbers → 32-bit
  - two 32-bit numbers → 64-bit
  - two 64-bit numbers → 128-bit



Multiplication on the x86

- Instruction inputs are strange
  - first operand is must be stored in RAX
  - second operand must be a register (can't be a immediate)
- Result is stored into two registers
  - rax will contain the lower 8 bytes
  - rdx will contain the upper 8 bytes

x86 Multiplication

First operand

RAX

↓

IMUL *register*

↓ ↓

RDX	RAX
Upper 8 bytes	Lower 8 bytes

Multiply - Signed

```
IMUL operand
```

Register, Memory

Multiply - Unsigned

```
MUL operand
```

Register, Memory

Unsigned Multiply: 1846 by 42

```
MOV   $1846, %rax    #First operand
MOV   $42, %rbx      #Need register for MUL
MUL   %rbx           #RAX gets low 8 bytes
                     #RDX gets high 8 bytes
```

Multiplication Tips

- Even though you are just using RAX as input, both RAX and RDX will change
- Be aware that you might lose important data, and backup to memory if needed

Additional x86 Multiply Instructions

- x86 also contains versions of the IMUL instruction that take multiple operands
- Allows "short" multiplication – just stored in 1 register
- Please note: these do not exist for MUL

IMUL (few more combos)

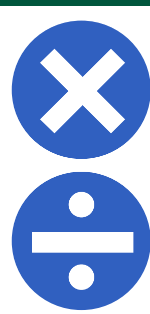
```
IMUL immediate, reg
IMUL memory, reg
IMUL reg, reg
```


Signed Multiply: 1846 by 42

```
MOV   $1846, %rax
IMUL  $42, %rax
```

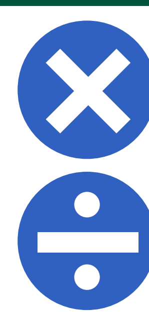

Division on the x86

- Division on the x86 is very interesting
- Like multiplication, it uses 2 registers
- The dividend (number being divided) uses two registers
  - RAX contains the lower 8 bytes
  - RDX contains the upper 8 bytes



Division on the x86

- These two registers are used for the result
- The output contains:
  - RAX will contain the quotient
  - RDX will contain the remainder



x86 Division

Upper 8 bytes — RDX | Lower 8 bytes — RAX

IDIV *register*

RDX — Remainder | RAX — Quotient

Divide - Signed

```
IDIV denominator
```

Register, Memory

Divide - Unsigned

```
DIV denominator
```

Register, Memory

Dividing Rules

- Sign Magnitude
  - create a new sign that is a XOR of the old ones
  - clear the old sign bits and expand with zeros
- 2's complement
  - numerator **must** be expanded to the destination size (twice the original)
  - this must be done beforehand
  - otherwise the result will be incorrect

Division Tips

- Even you don't store anything into RDX, you must clear it
- For signed-division…
  - RDX needs to contain *the sign-extended* value of RAX
  - this is covered in the next section

CWD (16 bit): Extend AX → DX

```
CWD
```

Convert Word Double

CDQ (32 bit): Extend EAX → EDX

CDQ

Convert Double Quad

CQO (64 bit): Extend RAX → RDX

CQO

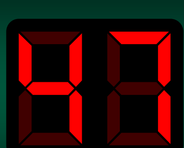
Convert Quad Oct

Divide 16-bit: -1846 by 42

```
MOV   $-1846, %ax    #AX is the dividend
MOV   $42, %bx       #Divisor
CWD                  #Extend the sign to DX
IDIV  %bx            #AX gets quotient
                     #DX gets remainder
```


Divide 64-bit: -1846 by 42

```
MOV   $-1846, %rax   #RAX is the dividend
MOV   $42, %rbx      #Divisor
CQO                  #Extend the sign to RDX
IDIV  %rbx           #RAX gets quotient
                     #RDX gets remainder
```

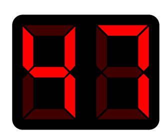



Numbers in Programs

The primitive types are pretty primitive

Primitive Data Types

- Most popular program languages hide the true nature of the computer from you
- However, most of the language's primitive data types are the same types recognized by the processor



Integer Data Types

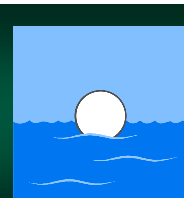
- Integer data types are stored in simple binary numbers
- The number of bytes used varies: 1, 2, 4, etc.…
- Languages often have a unique name for each – short, int, long, etc…

Floating-Point Data Type

- Floating-point numbers are usually stored using the IEEE 754 standard
- Languages often have unique names for them such as float, double, real

Floating-Point Data Type

- This is not always the case
  - some languages implement their own structures
  - e.g. COBOL
- Why?
  - some processors do not have floating-point instructions
  - or the language needs more precision and control

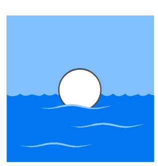


Floating Point Numbers

Real numbers are *real* complex

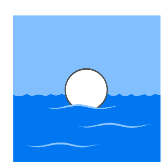
Floating Point Numbers

- Often, programs need to perform mathematics on *real* numbers
- *Floating point numbers* are used to represent quantities that cannot be represented by integers



Floating Point Numbers

- Why?
  - regular binary numbers can only store whole positive and negative values
  - many numbers outside the range representable within the system's bit width (too large/small)



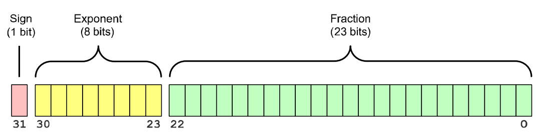
IEEE 754

- Practically modern computers use the *IEEE 754 Standard* to store floating-point numbers
- Represent by a mantissa and an exponent
  - similar to scientific notation
  - the value of a number is: *mantissa* $\times$ $2^{exponent}$
  - uses signed magnitude

IEEE 754

- Comes in three forms:
  - single-precision: 32-bit
  - double-precision: 64-bit
  - quad-precision: 128-bit
- Also supports special values:
  - negative and positive *infinity*
  - and "not a number" for errors (e.g. 1/0)

IEEE 754 Single Precision (32 bit)



Interpretation: Invalid Numbers

NaN → **1 / 0**

Naan →