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

\[
\begin{array}{c}
1 \\
+ \\
2 \quad 7 \quad 8 \quad 1 \\
\hline \\
3 \quad 7 \quad 2 \quad 1 \\
\hline \\
6 \quad 5 \quad 0 \quad 2
\end{array}
\]

Adding Binary Example

\[
\begin{array}{c}
182 \\
+ \\
51 \\
\hline \\
233
\end{array}
\]

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.

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: 13 and -13

- 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 #1

- When a number can represent both positive and negative numbers, it is called a signed integer.
- Otherwise, it is unsigned.
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...

```java
if (x == +0 || x == -0)
```

Oh noes! Two zeros?

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
```
Practically all computers 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.

Since negatives are subtracted from $2^n$, they can simply be added. The extra carry 1 (if it exists) is discarded, simplifying 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.

In reality, processors don't know (or care) if a number is unsigned or signed. The hardware works the same either way. It's your responsibility to keep track if it's signed/unsigned.

In many cases, you must use the correct instruction - based on whether you are treating the data as signed or unsigned. With great programming power comes great responsibility.
Adding 2’s Complement

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Multiplying Binary Numbers

\[11 \times 11 = 1001\]

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

```c
total = 0;
for (i = 0; i < A; i++)
{
    total += B;
}
```
Multiplying: The Bad Way

- If A or B is large, then it 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 multiply by using long multiplication – just like you do
- Number of additions is fixed to 8, 16, 32, 64 depending on the size of the integer
- The following example multiplies 2 unsigned 4-bit numbers

Unsigned Integer: $13 \times 10$

\[
\begin{array}{c}
1 & 1 & 0 & 1 \\
\times & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
+ \\
0 & 0 & 0 & 0 \\
\end{array}
\]

Unsigned Integer: $13 \times 10$

\[
\begin{array}{c}
1 & 1 & 0 & 1 \\
\times & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
+ \\
1 & 1 & 0 & 1 \\
\end{array}
\]
Unsigned Integer: $13 \times 10$

<p>| 1 | 1 | 0 | 1 |</p>
<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Unsigned Integer: $13 \times 10$

<p>| 1 | 1 | 0 | 1 |</p>
<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- When two numbers are multiplied, the product will have **twice** the number of digits
- Examples:
  - 8-bit x 8-bit $\rightarrow$ 16-bit
  - 16-bit x 16-bit $\rightarrow$ 32-bit
- Processors can:
  - fit the result in the original bit-size *(and raise an overflow if it does not fit)*
  - ...or store the new double-sized number

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...
Subtraction

\[ \text{SUB target, value} \]

Immediate, Register, Memory

Register, Memory

Negate (2's complement)

\[ \text{NEG register} \]

Example: Simple Add

Move value into RAX

\[ \text{MOV rax, 17} \]
\[ \text{ADD rax, 2} \]

RAX += 2

x86 Multiplication

Complex Math is Complex

Multiplication & Division

- The x86 treats multiplication quite differently than add/subtract
- Why? Intel was designed as a business processor and high-precision math is paramount

Multiplication Review

- Remember: when two \( n \) bit numbers are multiplied, result will be \( 2n \) bits
- So...
  - two 8-bit numbers \( \rightarrow \) 16-bit
  - two 16-bit numbers \( \rightarrow \) 32-bit
  - two 32-bit numbers \( \rightarrow \) 64-bit
  - two 64-bit numbers \( \rightarrow \) 128-bit
Multiplication on the x86

- Intel stores the product into two registers
  - RAX will contain the lower 8 bytes
  - RDX will contain the upper 8 bytes
- This maintains the high-precision result
- Instruction inputs are strange
  - first operand is **must** be stored in RAX
  - second operand **must** be a register or memory

### Multiply - Signed

- IMUL operand
- Register or Memory only

### Multiply - Unsigned

- MUL operand
- Register or Memory only

### Signed Multiply: 1846 by 42

```
MOV rax, 1846    #First operand
MOV rbx, 42      #Need register for MUL
IMUL 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
Over time, designers requested a low-precision version of multiplication. Intel added "short" IMUL instructions that store into a single register. Please note: these do not exist for MUL.

Additional x86 Multiply Instructions

Signed Multiply: 1846 by 42

This works, but could cause an overflow.

Signed Multiply: 1846 by 42

This works, but could cause an overflow.

Extending Unsigned Integers

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

Extending Unsigned Integers

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

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

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

### 2's Complement Incorrectly Done

-13: 11110011

243: 00000000 11110011

### 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
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

2's Complement Extended: -77
2's Complement Extended: -77

\[ \begin{array}{cccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\downarrow & & & & & & & \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} \]

x86 Division

Complex Math is Complex

Division on the x86

- Division on the x86 is very interesting
- Since multiplication stores into two registers, divide uses these as the numerator
- Numerator is fixed as:
  - RAX contains the lower 8 bytes
  - RDX contains the upper 8 bytes

Division on the x86

- These two registers are also used for the result
- The output contains:
  - RAX will contain the quotient (the whole number)
  - RDX will contain the remainder

x86 Division

\[
\begin{array}{cccc}
\text{Upper 8 bytes} & \text{Lower 8 bytes} \\
\hline
\text{RDX} & \text{RAX} \\
\hline
\text{IDIV denominator} \\
\text{RDX} & \text{RAX} \\
\text{Remainder} & \text{Quotient} \\
\end{array}
\]

Divide - Signed

IDIV denominator

Register or Memory only
Divide - Unsigned

\[ \text{DIV denominator} \]

Register or Memory only

Dividing Rules

- For 2's complement systems...
  - numerator must be expanded to the destination size (twice the original)
  - this must be done beforehand
  - otherwise the result will be incorrect

On the Intel...

- You must setup RDX before you divide
- For unsigned: store 0 in it
- For signed-division:
  - RAX needs to be sign-extended into RDX
  - there are special instructions

Sign Extend Example

\[
\begin{array}{c|c|c}
\text{RAX} & 0 & 1 0 0 1 1 0 1 \\
\text{RDX} & 0 & 1 0 0 1 1 0 1 \\
\end{array}
\]

Sign Extend Example 2

\[
\begin{array}{c|c|c}
\text{RAX} & 1 & 0 1 1 0 0 1 1 \\
\text{RDX} & 1 & 0 1 1 0 0 1 1 \\
\end{array}
\]
Sign Extend Example 2

CWD (16 bit): Extend AX → DX

CDQ (32 bit): Extend EAX → EDX

CQO (64 bit): Extend RAX → RDX

Divide 64-bit: -1846 by 42

MOV rax, -1846    #RAX is the dividend
MOV rbx, 42       #Divisor
CQO               #Sign extend to RDX
IDIV rbx          #RAX gets quotient
                  #RDX gets remainder