Modern Assembly Language Programming
with the
ARM processor
Chapter 7: Integer Mathematics
1 Introduction

2 Complement Math

3 Signed and Unsigned Binary Integers

4 Binary Multiplication

5 Binary Division
Binary Addition

Binary addition works exactly the same as Decimal addition. Except that the result of each column is limited to 0 or 1.

\[
\begin{array}{cccccccc}
1 & 75 & + & 19 & = & \hline \\
11 & 01001011 & + & 00010011 & \hline
01011110 & 94 & + & 19 & = & 94
\end{array}
\]
Subtracting by Adding – Base 10

This is called 10’s complement arithmetic.

<table>
<thead>
<tr>
<th>Complement Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

The 9’s complement of 56 (in three digits) is 943.
The 10’s complement of 56 in three digits is 944.
Adding the 10’s complement of $x$ is the same as subtracting $x$. 

\[
\begin{align*}
384 & \quad - 56 \\
\hline
328 & \quad =
\end{align*}
\]

\[
\begin{align*}
384 & \quad + 1 \\
\hline
1328 & \quad =
\end{align*}
\]
Subtracting by Adding – Binary

This is called 2’s complement arithmetic.*

<table>
<thead>
<tr>
<th>Complement Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
- & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1
\end{array} = \begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

The 1’s complement of 110001 (in eight bits) is 11001110.
The 2’s complement of 110001 (in eight bits) is 11001111.
Adding the 2’s complement of \( x \) is the same as subtracting \( x \).

Therefore, the 2’s complement of \( x \) is the same as \( -x \), and that is one way to store negative numbers in the computer.

\[
92_{10} = 1011100_2, \quad 49_{10} = 110001_2, \quad 43_{10} = 101011_2,
\]
Signed and Unsigned Integers

- Numbers can be interpreted by the programmer as signed or unsigned.
- The computer treats them both the same.
- Given an 8-bit integer, the programmer can consider it to hold:
  - an unsigned value between 0 and 255, or
  - a signed (two’s complement) number between −128 and +127.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01111110</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>01111111</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>10000000</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>10000001</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111110</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>11111111</td>
<td>255</td>
<td>-1</td>
</tr>
</tbody>
</table>
Base Conversion of Negative Numbers

Converting a signed 2’s complement number from binary to decimal.

1. If the most significant bit is ’1’, then
   1. Find the 2’s complement
   2. Convert the result to base 10
   3. Add a negative sign

2. Else
   1. Convert the result to base 10

<table>
<thead>
<tr>
<th>Number</th>
<th>1’s Complement</th>
<th>2’s Complement</th>
<th>Base 10</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>11010010</td>
<td>00101101</td>
<td>00101110</td>
<td>46</td>
<td>–46</td>
</tr>
<tr>
<td>1111111100010110</td>
<td>0000000011101001</td>
<td>0000000011101010</td>
<td>234</td>
<td>–234</td>
</tr>
<tr>
<td>01110100</td>
<td>Not negative</td>
<td></td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>1000001101010110</td>
<td>0111110010101001</td>
<td>0111110010101010</td>
<td>31914</td>
<td>–31914</td>
</tr>
<tr>
<td>0101001111011011</td>
<td>Not negative</td>
<td></td>
<td>21467</td>
<td></td>
</tr>
</tbody>
</table>
Base Conversion of Negative Numbers

Converting a negative number from decimal to binary.

1. Remove the negative sign
2. Convert the number to binary
3. Take the 2’s complement

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Positive Binary</th>
<th>1’s Complement</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>−46</td>
<td>00101110</td>
<td>11010001</td>
<td>11010010</td>
</tr>
<tr>
<td>−234</td>
<td>0000000011101010</td>
<td>1111111100010101</td>
<td>1111111100010110</td>
</tr>
<tr>
<td>−116</td>
<td>01110100</td>
<td>10001011</td>
<td>10001100</td>
</tr>
<tr>
<td>−31914</td>
<td>0111110010101010</td>
<td>1000001101010110</td>
<td>1000001101010111</td>
</tr>
<tr>
<td>−21467</td>
<td>0101001111011011</td>
<td>1010110000100100</td>
<td>1010110000100101</td>
</tr>
</tbody>
</table>
Addition, Subtraction, and Negation – Examples

<table>
<thead>
<tr>
<th>23</th>
<th>00010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 15</td>
<td>+ 00001111</td>
</tr>
<tr>
<td>38</td>
<td>00100110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23</th>
<th>00010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 15</td>
<td>+ 11110001</td>
</tr>
<tr>
<td>8</td>
<td>100001000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>- 23</th>
<th>11101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 15</td>
<td>+ 00001111</td>
</tr>
<tr>
<td>- 8</td>
<td>11111000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>- 23</th>
<th>11101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 15</td>
<td>+ 11110001</td>
</tr>
<tr>
<td>- 38</td>
<td>111011010</td>
</tr>
</tbody>
</table>
Long Multiplication

The result of multiplying an \( n \) bit number by an \( m \) bit number is an \( n + m \) bit number.
Long Multiplication - Signed vs Unsigned

The result depends on whether you are doing signed or unsigned multiply!

\[
\begin{array}{c c c}
73 & \times & 11011001 \\
\times & -39 & \times 01001001 \\
\hline
657 = 11111111111011001 \\
219 & 111111011001 \\
-2847 & 1111010011100001 \\
\end{array}
\]

\[
\begin{array}{c c c}
73 & \times & 11011001 \\
\times & 217 & \times 01001001 \\
\hline
511 = 00000000011011001 \\
73 & 000011011001 \\
146 & 0011011001 \\
15841 & 0011110111100001 \\
\end{array}
\]

The 2’s complement of 0011110111100001 is 1100001000011110 + 1 = 1100001000011111

You can not always use an unsigned multiply and negate the result!
Algorithm for Unsigned Multiplication – Part 1

To multiply two \( n \) bit numbers, you must be able to add two \( 2n \) bit numbers.

Assume we have \( x \) in \( r1:r0 \) and \( y \) in \( r3:r2 \)
(The high order words are in the high-order registers)

and we want to calculate \( x = x + y \)

ARM Assembly:

```
1  adds r0,r0,r2  @ add the low-order words, and
2       @ set flags in CPSR
3  adc r1,r1,r3  @ add the high-order words plus
4       @ the carry flag
```

Early ARM processors did not have a multiply instruction.

We will show how to multiply two 8-bit numbers to get a 16-bit result.

The same algorithm works for numbers of any size.
Algorithm for Unsigned Multiplication – Part 2

Given two 8-bit numbers, $x$ and $y$, where $x$ is the multiplicand and $y$ is the multiplier:

1. Extend the multiplicand $x$ to 16 bits.
2. Set a 16-bit register, $a$, to zero,
3. **while** $y \neq 0$ **do**
4. **if** $y$ is an odd number **then**
5. \[ a \leftarrow a + x \]
6. **end if**
7. **Logical** shift $y$ right one bit
8. Shift $x$ left one bit
9. **end while**
Algorithm for Unsigned Multiplication – Example

Binary multiplication is a sequence of shift and add operations.

\[ x = 01101001 \text{ and } y = 01011010 \]

<table>
<thead>
<tr>
<th>(a)</th>
<th>(x)</th>
<th>(y)</th>
<th>Next operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>00000000001101001</td>
<td>01011010</td>
<td>shift only</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00101101</td>
<td>add, then shift</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00010110</td>
<td>shift only</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00001011</td>
<td>add, then shift</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00001011</td>
<td>add, then shift</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00001011</td>
<td>add, then shift</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00000010</td>
<td>shift only</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00000001</td>
<td>add, then shift</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00000001</td>
<td>add, then shift</td>
</tr>
<tr>
<td>0000000000000000</td>
<td>00000000011010010</td>
<td>00000000</td>
<td>shift only</td>
</tr>
</tbody>
</table>

\[ 105 \times 90 = 9450 \]
Multiplication on ARM

On the ARM processor, the algorithm to multiply two 32-bit unsigned integers is very efficient:

```
    mov    r0, #0 @ r0 = low-order word of result
    mov    r1, #0 @ r1 = high-order word of result
    ldr    r2, =x @ load pointer to multiplicand
    ldr    r2, [r2] @ r2<-low-order word of multiplicand
    mov    r3, #0 @ r3<-high-order word of multiplicand
    ldr    ip, =y @ load pointer to multiplier
    ldr    ip, [ip] @ ip<-multiplier

loop:  tst    ip, #1 @ is y odd?
    addnes r0, r0, r2 @ add and set flags if y is odd
    tst    ip, #1 @ previous add may have changed flags
    adcne r1, r1, r3 @ add and use carry flag if y is odd
    lsls   r2, r2, #1 @ shift lsw of x left into carry bit
    lsl    r3, r3, #1 @ make room for the carry bit is msw
    adc    r3, r3, #0 @ add carry bit to msw of x
    lsrs   ip, ip, #1 @ shift y right
    bne    loop @ if y==0, we are done
```
Short Multiplication on ARM

If we only want a 32-bit result, we can make it even more efficient:

```assembly
1  mov  r0, #0 @ r0 is result
2  ldr  ip, =y @ ip is multiplier
3  ldr  ip, [ip]
4  ldr  r2, =x @ r2 is multiplicand
5  ldr  r2, [r2]
6  lsrs ip, ip, #1 @ shift y right carry<-lsb
7  loop:
8  6    addcs r0, r0, r2 @ add if carry is set
9  6    lsl  r2, r2, #1 @ shift multiplicand left
10   6   lsrs ip, ip, #1 @ shift y right carry<-lsb
11   6   bne  loop  @ if y==0, we are done
```

If \(x\) or \(y\) is a constant, then we don’t need the loop!
Multiplication by a Constant

Suppose we want to multiply a number \( x \) by \( 10_{10} \).
\[ 10_{10} = 1010_2 \], so we will add \( x \) shifted left 1 bit plus \( x \) shifted left 3 bits

```
1 ldr  r0, =x
2 ldr  r0, [r0] @ load \( x \)
3 lsl  r0,r0,#1 @ shift \( x \) left one bit
4 add  r0,r0,r0,lsl #2 @ shift two more bits and add
```

Now suppose we want to multiply a number \( x \) by \( 11_{10} \).
\[ 11_{10} = 1011_2 \], so we will add \( x \) plus \( x \) shifted left 1 bit plus \( x \) shifted left 3 bits

```
1 ldr  r1, =x
2 ldr  r1, [r1] @ load \( x \)
3 add  r0,r1,r1,lsl #1 @ shift one bit and add
4 add  r0,r0,r1,lsl #3 @ shift three bits and add
```
Multiplication by a Constant (continued)

Now suppose we want to multiply a number \( x \) by \( 1000_{10} \).
\[
1000_{10} = 111101000_2
\]
It looks like we need 1 shift plus 5 add/shift operations, or 6 add/shift operations... but we can do better.

```
1 ldr  r1, =x
2 ldr  r1, [r1]  @ load x
3 add  r0, r1, r1, lsl  #1  @ shift and add: r0<-x*3
4 add  r0, r0, r0, lsl  #2  @ r0<-x*3 + x*3*4 (x*15)
5 add  r0, r1, r0, lsl  #1  @ r0<-x + x*15*2 (x*31)
6 lsl  r0, #5  @ r0<-x*31*32 (x*992)
7 add  r0, r0, r1, lsl  #3  @ r0<-x*992 + x*8
```

If we inspect the constant multiplier, we can usually find a pattern to exploit that will save a few instructions.
Now suppose we want to multiply a number $x$ by $255_{10}$.

$255_{10} = 11111111_2$

It looks like we need 7 add/shift operations... but we can do it with 3.

```
1 ldr r1, =x
2 ldr r1, [r1] @ load x
3 add r0, r1, r1, lsl #1 @ shift and add: r0←x*3
4 add r0, r0, r0, lsl #2 @ r0←x*3 + x*3*4 (x*15)
5 add r0, r0, r0, lsl #4 @ r0←x*15 + x*15*16 (x*255)
```

This may be faster than a hardware multiply.

But why not multiply $x$ by 256 then subtract $x$?

```
1 @ x is currently stored in r1
2 rsb r0, r1, r1, lsl #8 @ r1 ← x*256-x
```

This is faster than a hardware multiply.
Multiplication of Large Numbers

\[
\begin{array}{c}
\begin{array}{c}
\text{a}_1 \\
\text{a}_0
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{b}_1 \\
\text{b}_0
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{Product of } a \times b
\end{array}
\]

\[=\]

\[
\begin{array}{c}
\text{a}_0 \times \text{b}_0
\end{array}
\]

\[
\begin{array}{c}
\text{a}_0 \times \text{b}_1
\end{array}
\]

\[
\begin{array}{c}
\text{a}_1 \times \text{b}_0
\end{array}
\]

\[
\begin{array}{c}
\text{a}_1 \times \text{b}_1
\end{array}
\]
Binary division is a sequence of shift and subtract operations.

\[
\begin{array}{c}
949 \\
13 \overline{12345} \\
11700 \\
645 \\
520 \\
125 \\
117 \\
8 \\
\end{array}
\]

\[
\begin{array}{c}
1110110101 \\
1101 \overline{11000000111001} \\
1101000000000 \\
1011000111001 \\
1101000000000 \\
1001001111001 \\
1101000000000 \\
1010111001 \\
1101000000000 \\
1001001 \\
110100 \\
10101 \\
1101 \\
\end{array}
\]
Algorithm for Division: Step 1

Shift divisor Left until it is greater than dividend and count the number of shifts.

94 \div 7 =

\[
\begin{array}{c}
\text{Dividend:} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\text{Divisor:} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\text{Counter:} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\text{Dividend:} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\text{Divisor:} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{Counter:} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\text{Dividend:} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\text{Divisor:} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\text{Counter:} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\hline
\text{Dividend:} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\text{Divisor:} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
\text{Counter:} & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
\text{Dividend:} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\text{Divisor:} & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\text{Counter:} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
\end{array}
\]
Algorithm for Division: Step 2

Subtract if possible, then shift to the right. Repeat while Counter $\geq 0$.

| Quotient: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Dividend: | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| Divisor:  | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Counter:  | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

- **Divisor > Dividend**: No subtract, shift 0 into Quotient, decrement Counter, shift Dividend right

| Quotient: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Dividend: | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| Divisor:  | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| Counter:  | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

- **Divisor <= Dividend**: Subtract, shift 1 into Quotient, decrement Counter, shift Dividend right

| Quotient: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Dividend: | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| Divisor:  | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Counter:  | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

- **Divisor <= Dividend**: Subtract, shift 1 into Quotient, decrement Counter, shift Dividend right

| Quotient: | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| Dividend: | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| Divisor:  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| Counter:  | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

- **Divisor > Dividend**: No subtract, shift 0 into Quotient, decrement Counter, shift Dividend right

| Quotient: | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Dividend: | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| Divisor:  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Counter:  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- **Divisor <= Dividend**: Subtract, shift 1 into Quotient, decrement Counter, shift Dividend right

| Quotient: | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Dividend: | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| Divisor:  | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| Counter:  | 0 | 0 | 0 | 0 | 0 | 0 |

- **Divisor <= Dividend**: Subtract, shift 1 into Quotient, decrement Counter, shift Dividend right

| Quotient: | 0 | 0 | 0 | 1 | 1 | 0 |
| Dividend: | 0 | 0 | 0 | 0 | 1 | 0 |
| Divisor:  | 0 | 0 | 0 | 0 | 0 | 1 |
| Counter:  | 1 | 1 | 1 | 1 | 1 | 1 |

- **Counter < 0**: We are finished. Bonus! The modulus (remainder) is in the Dividend register!
Flowchart for Division

1. **Count < 0?**
   - Yes: **Return Result**
   - No:
     - **Divisor <= Dividend?**
       - Yes: **Shift 1 into Quotient**
       - No: **Shift 0 into Quotient**

2. **Shift Divisor left until it is ≥ Dividend and count the number of shifts.**

3. **Subtract divisor from Dividend**

4. **Decrement Count and shift Divisor right**
Modified Algorithm for Division: Step 1

Instead of counting the shifts, shift a bit left in another register.

\[
\begin{array}{c}
94 \div 7 = \\
1101 \\
1111110 \\
111000 \\
100110 \\
111000 \\
1010 \\
111 \\
11
\end{array}
\]

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Modified Algorithm for Division: Step 2

Subtract if possible, then shift to the right. Repeat while Power > 0.

<table>
<thead>
<tr>
<th>Quotient:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

{ Divisor > Dividend: 
  shift Power right, shift Dividend right
}

<table>
<thead>
<tr>
<th>Quotient:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

{ Divisor ≤ Dividend: 
  Dividend -= Divisor,
  Quotient += Power, shift Power right, shift Dividend right
}

<table>
<thead>
<tr>
<th>Quotient:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

{ Divisor ≤ Dividend: 
  Dividend -= Divisor,
  Quotient += Power, shift Power right, shift Dividend right
}

<table>
<thead>
<tr>
<th>Quotient:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

{ Divisor > Dividend: 
  shift Power right, shift Dividend right
}

<table>
<thead>
<tr>
<th>Quotient:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

{ Divisor ≤ Dividend: 
  Dividend -= Divisor,
  Quotient += Power, shift Power right, shift Dividend right
}

<table>
<thead>
<tr>
<th>Quotient:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Divisor:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Power:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

{ Power = 0: We are finished. Bonus! The modulus (remainder) is in the Dividend register! }
### Division on ARM

```
1 udiv32: cmp  r1,#0  @ if divisor == zero
2    beq  qudiv32 @ exit immediately
3    mov  r2,r1 @ move divisor to r2
4    mov  r1,r0 @ move dividend to r1
5    mov  r0,#0 @ clear r0 to accumulate result
6    mov  r3,#1 @ set 'current' bit in r3
7 divstrt: cmp  r2,#0 @ WHILE ((msb of r2 != 1)
8        blt  divloop @ && (r2 < r1))
9        cmp  r2,r1 @ shift dividend left
10       lsls  r2,r2,#1 @ shift "current" bit left
11       lsls  r3,r3,#1 @ end WHILE
12      bls  divstrt @ divloop:
13      cmp  r1,r2 @ if dividend >= divisor
14     subhs r1,r1,r2 @ subtract divisor from dividend
15    addhs r0,r0,r3 @ set "current" bit in the result
16     lsr  r2,r2,#1 @ shift dividend right
17      lsrs r3,r3,#1 @ Shift current bit right into carry
18     bcc  divloop @ If carry not clear, we are done
19 qudiv32: mov pc,lr
```
Division by a Constant

In general, division is slow, but division by a constant $c$ can be simplified to a multiply by the reciprocal of $c$.

$$x \div c = x \times \frac{1}{c}$$

But we have to do it in binary using only integers.

$$x \div c = x \times \frac{2^n}{c} \times 2^{-n}$$

Multiplying by $2^n$ is the same as shifting left by $n$ bits. Multiplying by $2^{-n}$ is done by shifting right by $n$ bits. Let

$$m = \frac{2^n}{c}.$$  

We want to choose $n$ such that $m$ is as large as possible with the number of bits we are given.
Division by a Constant - Example

Suppose we want efficient code to calculate \( x \div 23 \) using 8-bit signed integer multiplication.

Find \( m = \frac{2^n}{c} \), such that \( 01111111_2 \geq m \geq 01000000_2 \).

If we choose \( n=11 \), then

\[
m = \frac{2^{11}}{23} \rightarrow \quad m = \frac{2^{11}}{23}
\]

In 8 bits, \( m \) is \( 01011001_2 \) or \( 59_{16} \).

After calculating \( y = x \times m \), it will be necessary to shift \( y \) right by 11 bits.
Division by a Constant - Example (continued)

The result for some values of \( x \) may be incorrect due to rounding error. If the divisor is positive, increment the reciprocal value by one in order to alleviate these errors.

To calculate \( 101_{10} \div 23_{10} \):

\[
\begin{array}{l}
01100101 \\
\times \quad 01011010 \\
\hline
01100101 \\
01100101 \\
01100101 \\
01100101 \\
\hline
10001110000010
\end{array}
\]

10001110000010_2 shifted right 11_10 bits is : \( 100_2 = 4_{10} \).

If the modulus is required, it can be calculated as: \( 101 - (4 \times 23) = 9 \).
On the Arm, we can divide by 23 very quickly:

@ The following code will calculate r2/23
@ It will leave the quotient in r0 and the remainder in r1
@ It will also use register r3 as a temporary variable

1. ldr r3,=0x590B2165 @ load 1/23 shifted left by 35 bits
2. smull r0,r1,r3,r2 @ multiply (3 to 7 clock cycles)
3. mov r3,r2,asr #31 @ get sign of numerator (0 or -1)
4. rsb r0,r3,r1,asr#3 @ shift right and adjust for sign
   @ now get the modulus, if needed
5. mov r1,#23 @ move denominator to r1
6. mul r1,r1,r0 @ multiply denominator by quotient
7. sub r1,r2,r1 @ subtract that from numerator
The value of $m$ can be directly computed by using the equation

$$m = \frac{2^p + \lfloor \log_2 c \rfloor - 1}{c} + 1,$$

where $p$ is the desired number of bits of precision. For example, to divide by the constant 33, with 16 bits of precision, we compute $m$ as

$$m = \frac{2^{16+5}-1}{33} + 1 = \frac{2^{20}}{33} + 1 = 31776.030303 \approx 31776 = 7C20_{16}.$$  

Therefore, multiplying a 16 bit number by $7C20_{16}$ and then shifting right 20 bits is equivalent to dividing by 33.
Uses for These Techniques

98% of computing devices are embedded.

- In 2012, the global market for embedded systems was about $1.47 trillion.
- The annual growth rate is about 14%
- Forecasts predict over 40 billion devices will be sold in 2020.

Most embedded systems are cost sensitive and use very small processors.

Some very common embedded processors are the:

- PicMicro PIC family
- Atmel AVR family,
- Intel 8051 family, and the
- Motorola 68HC11 family.

The 68HC11, 8051, AVR200+, and PIC18+ all have an 8-bit by 8-bit hardware multiply that produces a 16-bit result.

Smaller, cheaper versions of AVR and PIC have no hardware multiply at all.
Summary

- Understanding the basic mathematical operations can enable the assembly programmer to
  - work with integers of any arbitrary size
  - achieve efficiency that cannot be matched by any other language.

However!

- It is best to focus the assembly programming on areas where the greatest gains can be made.