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

$$\boxed{\begin{array}{rcr} {\scriptstyle 1} & & {\scriptstyle 1\,1} \\ 75 & & 01001011 \\ +\;\;19 & = & +\;\;00010011 \\ \hline 94 & & 01011110 \end{array}}$$

Subtracting by Adding – Base 10

This is called 10’s complement arithmetic.

Complement
Table

0	9
1	8
2	7
3	6
4	5
5	4
6	3
7	2
8	1
9	0

$$\begin{array}{r} 384 \\ -\quad 56 \\ \hline 328 \end{array} \quad = \quad \begin{array}{r} 384 \\ 943 \\ +\qquad 1 \\ \hline 1\,|\,328 \end{array}$$

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.

Subtracting by Adding – Binary

This is called 2's complement arithmetic.*

Complement
Table

0	1
1	0

$$\begin{array}{r} 0\,1\,0\,1\,1\,1\,0\,0 \\ -\;\; 0\,0\,1\,1\,0\,0\,0\,1 \\ \hline 0\,0\,1\,0\,1\,0\,1\,1 \end{array} \quad = \quad \begin{array}{r} 0\,1\,0\,1\,1\,1\,0\,0 \\ 1\,1\,0\,0\,1\,1\,1\,0 \\ +\;\; 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline 1\,\big|\,0\,0\,1\,0\,1\,0\,1\,1 \end{array}$$

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.

---

[*] $92_{10} = 1011100_2$, $49_{10} = 110001_2$, $43_{10} = 101011_2$,

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 $+127$.

Binary	Unsigned	Signed
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111110	126	126
01111111	127	127
10000000	128	-128
10000001	129	-127
⋮	⋮	⋮
11111110	254	-2
11111111	255	-1

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

Number	1’s Complement	2’s Complement	Base 10	Negative
11010010	00101101	00101110	46	−46
1111111100010110	0000000011101001	0000000011101010	234	−234
01110100	Not negative		116	
1000001101010110	0111110010101001	0111110010101010	31914	−31914
0101001111011011	Not negative		21467	

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

Base 10	Positive Binary	1's Complement	2's Complement
−46	00101110	11010001	11010010
−234	0000000011101010	1111111100010101	1111111100010110
−116	01110100	10001011	10001100
−31914	0111110010101010	1000001101010110	1000001101010111
−21467	0101001111011011	1010110000100100	1010110000100101

Addition, Subtraction, and Negation – Examples

$$\begin{array}{rr} & 23 \\ + & 15 \\ \hline & 38 \end{array} = \begin{array}{rr} & 00010111 \\ + & 00001111 \\ \hline & 00100110 \end{array}$$

$$\begin{array}{rr} & 23 \\ - & 15 \\ \hline & 8 \end{array} = \begin{array}{rr} & 00010111 \\ + & 11110001 \\ \hline & 1|00001000 \end{array}$$

$$\begin{array}{rr} - & 23 \\ + & 15 \\ \hline & -\ 8 \end{array} = \begin{array}{rr} & 11101001 \\ + & 00001111 \\ \hline & 11111000 \end{array}$$

$$\begin{array}{rr} - & 23 \\ - & 15 \\ \hline - & 38 \end{array} = \begin{array}{rr} & 11101001 \\ + & 11110001 \\ \hline & 1|11011010 \end{array}$$

Long Multiplication

The result of multiplying an n bit number by an m bit number is an $n+m$ bit number

$$\boxed{\begin{array}{r} 1\,0\,1 \\ \times\quad 8\,9 \\ \hline 9\,0\,9 \\ 8\,0\,8 \\ \hline 8\,9\,8\,9 \end{array} \quad = \quad \begin{array}{r} 0\,1\,1\,0\,0\,1\,0\,1 \\ \times\quad 0\,1\,0\,1\,1\,0\,0\,1 \\ \hline 0\,1\,1\,0\,0\,1\,0\,1 \\ 0\,1\,1\,0\,0\,1\,0\,1 \\ 0\,1\,1\,0\,0\,1\,0\,1 \\ 0\,1\,1\,0\,0\,1\,0\,1 \\ \hline 0\,0\,1\,0\,0\,0\,1\,1\,0\,0\,0\,1\,1\,1\,0\,1 \end{array}}$$

Long Multiplication - Signed vs Unsigned

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

$$\begin{array}{r} 7\,3 \\ \times\quad -3\,9 \\ \hline 6\,5\,7 \\ 2\,1\,9 \\ \hline -2\,8\,4\,7 \end{array} \quad = \quad \begin{array}{r} 1\,1\,0\,1\,1\,0\,0\,1 \\ \times\quad 0\,1\,0\,0\,1\,0\,0\,1 \\ \hline 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,0\,1\,1\,0\,0\,1 \\ 1\,1\,1\,1\,1\,1\,1\,0\,1\,1\,0\,0\,1 \\ 1\,1\,1\,1\,0\,1\,1\,0\,0\,1 \\ \hline 1\,1\,1\,1\,0\,1\,0\,0\,1\,1\,1\,0\,0\,0\,0\,1 \end{array}$$

$$\begin{array}{r} 7\,3 \\ \times\quad 2\,1\,7 \\ \hline 5\,1\,1 \\ 7\,3 \\ 1\,4\,6 \\ \hline 1\,5\,8\,4\,1 \end{array} \quad = \quad \begin{array}{r} 1\,1\,0\,1\,1\,0\,0\,1 \\ \times\quad 0\,1\,0\,0\,1\,0\,0\,1 \\ \hline 0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0\,1\,1\,0\,0\,1 \\ 0\,0\,0\,0\,0\,1\,1\,0\,1\,1\,0\,0\,1 \\ 0\,0\,1\,1\,0\,1\,1\,0\,0\,1 \\ \hline 0\,0\,1\,1\,1\,1\,0\,1\,1\,1\,1\,0\,0\,0\,0\,1 \end{array}$$

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:

```
      adds    r0,r0,r2    @ add the low-order words, and
                          @ set flags in CPSR
      adc     r1,r1,r3    @ add the high-order words plus
                          @ 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:

```
Extend the multiplicand $x$ to 16 bits.
Set a 16-bit register, $a$, to zero,
while $y \neq 0$ do
    if y is an odd number then
        $a \leftarrow a + x$
    end if
    Logical shift $y$ right one bit
    Shift $x$ left one bit
end while
```

Algorithm for Unsigned Multiplication – Example

Binary multiplication is a sequence of shift and add operations.

$x = 01101001$ and $y = 01011010$

a	x	y	Next operation
0000000000000000	0000000001101001	01011010	shift only
0000000000000000	0000000011010010	00101101	add, then shift
0000000011010010	0000000110100100	00010110	shift only
0000000011010010	0000001101001000	00001011	add, then shift
0000010000011010	0000011010010000	00000101	add, then shift
0000101010101010	0000110100100000	00000010	shift only
0000101010101010	0001101001000000	00000001	add, then shift
0010010011101010	0011010010000000	00000000	shift only

$105 \times 90 = 9450$

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:

```
        mov     r0, #0   @ r0 is result
        ldr     ip, =y   @ ip is multiplier
        ldr     ip,[ip]
        ldr     r2, =x   @ r2 is multiplicand
        ldr     r2,[r2]
        lsrs    ip,ip,#1 @ shift y right carry<-lsb
loop:
        addcs   r0,r0,r2 @ add if carry is set
        lsl     r2,r2,#1 @ shift multiplicand left
        lsrs    ip,ip,#1 @ shift y right carry<-lsb
        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

```
        ldr     r0, =x
        ldr     r0,[r0]         @ load x
        lsl     r0,r0,#1        @ shift x left one bit
        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

```
        ldr     r1, =x
        ldr     r1,[r1]         @ load x
        add     r0,r1,r1,lsl #1 @ shift one bit and add
        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}.
$1000_{10} = 1111101000_2$
It looks like we need 1 shift plus 5 add/shift operations,
or 6 add/shift operations… but we can do better.

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

Multiplication by a Constant (continued)

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.

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

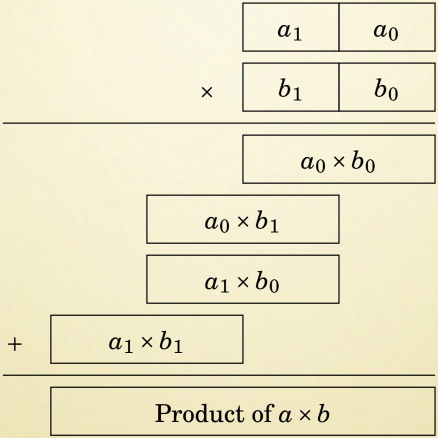
This may be faster than a hardware multiply.

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

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

This *is* faster than a hardware multiply.

Multiplication of Large Numbers



a_1 a_0

$\times$ b_1 b_0

$a_0 \times b_0$

$a_0 \times b_1$

$a_1 \times b_0$

$+$ $a_1 \times b_1$

Product of $a \times b$

Long Division

Binary division is a sequence of shift and subtract operations.

$$
\begin{array}{r}
949\\
13\overline{)12345}\\
\underline{11700}\\
645\\
\underline{520}\\
125\\
\underline{117}\\
8
\end{array}
$$

$$
\begin{array}{r}
1110110101\\
1101\overline{)11000000111001}\\
\underline{1101000000000}\\
1011000111001\\
\underline{110100000000}\\
100100111001\\
\underline{11010000000}\\
1010111001\\
\underline{110100000}\\
100011001\\
\underline{11010000}\\
1001001\\
\underline{110100}\\
10101\\
\underline{1101}\\
1000
\end{array}
$$

Algorithm for Division: Step 1

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

$94 \div 7 =$

$$\begin{array}{r}
1101\\
111\overline{)1011110}\\
\underline{111000}\\
100110\\
\underline{11100}\\
1010\\
\underline{111}\\
11
\end{array}$$

Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	0	0	0	1	1	1
Counter:	0	0	0	0	0	0	0	0
Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	0	0	1	1	1	0
Counter:	0	0	0	0	0	0	0	1
Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	0	1	1	1	0	0
Counter:	0	0	0	0	0	0	1	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	0	1	1
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

Algorithm for Division: Step 2

Subtract if possible, then shift to the right. Repeat while Counter >= 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	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	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	0	1

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

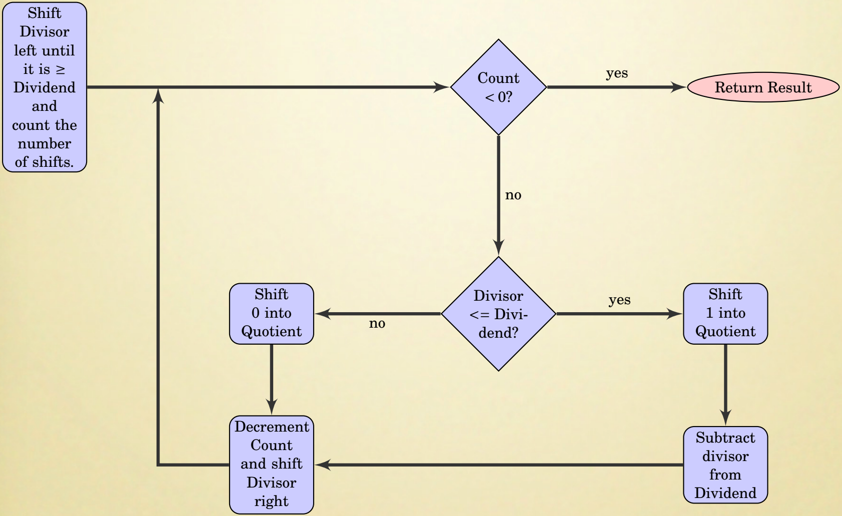
Quotient:	0	0	0	0	0	1	1	0
Dividend:	0	0	0	0	1	0	1	0
Divisor:	0	0	0	0	0	1	1	1
Counter:	0	0	0	0	0	0	0	0

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

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

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

Flowchart for Division



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

Count < 0?

yes

Return Result

no

Divisor <= Dividend?

yes

Shift 1 into Quotient

Subtract divisor from Dividend

no

Shift 0 into Quotient

Decrement Count and shift Divisor right

Modified Algorithm for Division: Step 1

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

$94 \div 7 =$

$$\begin{array}{r} 1101 \\ 111\overline{)1011110} \\ \underline{111000} \\ 100110 \\ \underline{11100} \\ 1010 \\ \underline{111} \\ 11 \end{array}$$

Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	0	0	0	1	1	1
Power:	0	0	0	0	0	0	0	1

Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	0	0	1	1	1	0
Power:	0	0	0	0	0	0	1	0

Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	0	1	1	1	0	0
Power:	0	0	0	0	0	1	0	0

Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	0	1	1	1	0	0	0
Power:	0	0	0	0	1	0	0	0

Dividend:	0	1	0	1	1	1	1	0
Divisor:	0	1	1	1	0	0	0	0
Power:	0	0	0	1	0	0	0	0

Modified Algorithm for Division: Step 2

Subtract if possible, then shift to the right. Repeat while Power > 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
Power:	0	0	0	1	0	0	0	0

Divisor > Dividend:
shift Power right, 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
Power:	0	0	0	0	1	0	0	0

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

Quotient:	0	0	0	0	1	0	0	0
Dividend:	0	0	1	0	0	1	1	0
Divisor:	0	0	0	1	1	1	0	0
Power:	0	0	0	0	0	1	0	0

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

Quotient:	0	0	0	0	1	1	0	0
Dividend:	0	0	0	0	1	0	1	0
Divisor:	0	0	0	0	1	1	1	0
Power:	0	0	0	0	0	0	1	0

Divisor > Dividend:
shift Power right, shift Dividend right

Quotient:	0	0	0	0	1	1	0	0
Dividend:	0	0	0	0	1	0	1	0
Divisor:	0	0	0	0	0	1	1	1
Power:	0	0	0	0	0	0	0	1

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

Quotient:	0	0	0	0	1	1	0	1
Dividend:	0	0	0	0	0	0	1	1
Divisor:	0	0	0	0	0	0	1	1
Power:	0	0	0	0	0	0	0	0

Power = 0: We are finished. Bonus! The modulus (remainder) is in the Dividend register!

Division on ARM

```
udiv32: cmp   r1,#0      @ if divisor == zero
        beq   qudiv32    @   exit immediately
        mov   r2,r1      @ move divisor to r2
        mov   r1,r0      @ move dividend to r1
        mov   r0,#0      @ clear r0 to accumulate result
        mov   r3,#1      @ set "current" bit in r3
divstrt:cmp   r2,#0      @ WHILE ((msb of r2 != 1)
        blt   divloop
        cmp   r2,r1      @ && (r2 < r1))
        lslls r2,r2,#1   @  shift dividend left
        lslls r3,r3,#1   @  shift "current" bit left
        bls   divstrt    @ end WHILE
divloop:cmp   r1,r2      @ if dividend >= divisor
        subhs r1,r1,r2   @   subtract divisor from dividend
        addhs r0,r0,r3   @   set "current" bit in the result
        lsr   r2,r2,#1   @ shift dividend right
        lsrs  r3,r3,#1   @ Shift current bit right into carry
        bcc   divloop    @ If carry not clear, we are done
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

$$m = \frac{2^{11}}{23} \rightarrow$$

$$
\begin{array}{r}
1011001 \\
10111\,\overline{)\,100000000000} \\
10111000000 \\
\hline
1001000000 \\
101110000 \\
\hline
11010000 \\
10111000 \\
\hline
11000 \\
10111 \\
\hline
1
\end{array}
$$

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 $101_{10} \div 23_{10}$:

$$
\begin{array}{r}
0\,1\,1\,0\,0\,1\,0\,1 \\
\times\quad 0\,1\,0\,1\,1\,0\,1\,0 \\
\hline
0\,1\,1\,0\,0\,1\,0\,1 \\
0\,1\,1\,0\,0\,1\,0\,1 \\
0\,1\,1\,0\,0\,1\,0\,1 \\
0\,1\,1\,0\,0\,1\,0\,1 \\
\hline
1\,0\,0\,0\,1\,1\,1\,0\,0\,0\,0\,0\,1\,0
\end{array}
$$

10001110000010_2 shifted right 11_{10} bits is : $100_2 = 4_{10}$.

If the modulus is required, it can be calculated as: $101 - (4 \times 23) = 9$

Division by a Constant on ARM

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
ldr   r3,=0x590B2165 @ load 1/23 shifted left by 35 bits
smull r0,r1,r3,r2    @ multiply (3 to 7 clock cycles)
mov   r3,r2,asr #31  @ get sign of numerator (0 or -1)
rsb   r0,r3,r1,asr#3 @ shift right and adjust for sign
                     @ now get the modulus, if needed
mov   r1,#23         @ move denominator to r1
mul   r1,r1,r0       @ multiply denominator by quotient
sub   r1,r2,r1       @ subtract that from numerator
```

Formula for Finding Reciprocal

The value of m can be directly computed by using the equation

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

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

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

Therefore, multiplying a 16 bit number by $7\mathrm{C}20_{16}$ 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.