Chapter 1
Number Systems

Agenda

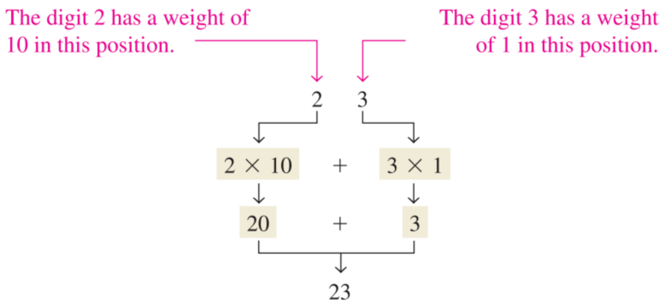
- Some preliminaries on how to convert between decimal, binary and hex.
- 8-bit unsigned number
- 8-bit signed number
- 2's complement: how to do 2'complement quickly?
- Overflow in signed arithmetic
- Multiplication
- Division

Positional Number Systems

- Positional system → "value" is based on the symbol and its position in the number
- Base-r number system
  - r unique symbols
  - represent 0 to r-1
  - e.g., decimal has 10 symbols, 0 to 9
    e.g., binary has 2 symbols, 0 to 1
    e.g., hexadecimal has 16 symbols, 0 to 9 and A to F

Base-r number systems

- The position of each digit indicates the magnitude of the quantity presented and can be assigned a weight, e.g., in decimal,



- In general, a number XYZ in base-r number system has the value of:

$$X \times r^2 + Y \times r^1 + Z \times r^0$$

- Positional systems must have the concept of "zero", e.g., in decimal, 26 or 206?

Notation systems

	written	ARM	Intel
decimal	10_{10}	10	10
hexadecimal	A_{16}	0xA	0AH
binary	1010_2	0b1010	1010B

- Computer system bases:
  2 (binary) and 16 (hexadecimal)
- Assembly language uses either
  a prefix notation system (e.g., ARM) or
  a suffix notation system (e.g., Intel)
- Syntax varies by assembly language *and* the development environment

Conversions between number systems

1) any base to decimal

   – expand the number into positional notation and evaluate

e.g., $110_2 = 1x2^2 + 1x2^1 + 0x2^0$

$= 4+2+0 = 6_{10}$

e.g. $A01_{16} = Ax16^2 + 0x16^1 + 1x16^0$

$= 10x256 + 0 + 1 = 2561_{10}$

Conversions between number systems

2) b) decimal to any base r

$\rightarrow$ *integer portion – successive divide by r*

e.g., $18_{10} = ?_2$ $= 10010_2$

18/2=9	R= 0
9/2=4	R= 1
4/2=2	R= 0
2/2=1	R= 0
1/2=0	R= 1

stop

remainder.

e.g., $2561_{10} = ?_{16}$ $= A01_{16}$

2561/16 = 160	R= 1
160/16 = 10	R= 0
10/16 = 0	R= 10 = A

stop

Conversions between number systems

3) binary ↔ hexadecimal
   4 binary digits ≡ 1 hexadecimal digit (a nibble)
   e.g. $001110110011_2 = ?_{16}$
   3 B 3

   e.g. $46A1_{16} = ?_2$

   0100011010100001

Number Representation

- A number is stored in a register or a memory location
- Assembly language fixes the length of the register and the memory location to a specific length *n*
- For ARM,
  n = 32 bits

Number Representation

1) Unsigned

– number is assumed to be non-negative and no bits are used to represent the sign

e.g., $00000001_2 = 1_{10}$

$10000001_2 = 1 \times 2^7 + 1 \times 2^0$

$= 128 + 1 = 129_{10}$

– Range of an n-bit register:

00… 00 → 11…11

Most significant bit

Least significant bit

$0 \rightarrow 2^n - 1$

Number Representation

2) Signed
   - numbers must have the following characteristics to be useful for calculations:
   1. positive and negative numbers
   2. sign test - easy way to determine if the number is positive or negative
   3. zero test - easy way to identify zero with preferably only one representation of zero.
   4. easy implementation of arithmetic operations.

Number Representation

Ways to represent signed numbers:

1. **sign-magnitude**

in general: SXX … X e.g., $00000001_2 = +1$

$10000001_2 = -1$

- both sign and magnitude are immediately obvious
- two zeros (i.e, 00000000 and 10000000)
- Addition and subtraction require different behaviors depending on the sign bit (e.g., adding two positive numbers require different operations from adding numbers with opposite signs)

Number Representation

2. Two's complement

- Positive number: Same as unsigned number representation
- Negative number:
  - Write the magnitude of the number as an unsigned number
  - Invert each bit
  - Add 1 to it
  - e.g., -127_{10} → Magnitude = 01111111
    Invert → 10000000
    +1 → 10000001
  - Quick way to do 2's complement in hex
    - Magnitude = 01111111 = 7F
    - Inversion is equivalent to subtracting F from the most (MSN) and least significant nibble (LSN):
    - MSN: F-7 = 8; LSN: F-F = 0. The result is 80.
    - 80 + 1 = 81

8-bit signed number representation

Decimal	Binary	Hex
+127	0111 1111	7F
.....	.....	.....
+2	0000 0010	02
+1	0000 0001	01
0	0000 0000	00
-1	1111 1111	FF
-2	1111 1110	FE
....	....	....
-127	1000 0001	81
-128	1000 0000	80

32-bit signed number representation

Decimal	Binary	Hex
-2,147,483,648	10000000000000000000000000000000	80000000
-2,147,483,647	10000000000000000000000000000001	80000001
-2,147,483,646	10000000000000000000000000000010	80000002
...	...	...
-2	11111111111111111111111111111110	FFFFFFFE
-1	11111111111111111111111111111111	FFFFFFFF
0	00000000000000000000000000000000	00000000
+1	00000000000000000000000000000001	00000001
+2	00000000000000000000000000000010	00000002
...	...	...
+2,147,483,646	01111111111111111111111111111110	7FFFFFFE
+2,147,483,647	01111111111111111111111111111111	7FFFFFFF

- Range of signed number:

$$-(2^{n-1}) \text{ to } (2^{n-1} - 1)$$

where n is the number of bits

Two's complement

- one zero (i.e., 00000000)
- Sign immediately obvious: Most significant bit = 1 → negative; Most significant bit = 0 → positive
- Addition and subtraction require the same operation

Addition of signed numbers

Both numbers positive:

$$\begin{array}{r} 00000111 \\ +\ 00000100 \\ \hline 00001011 \end{array} \qquad \begin{array}{r} 7 \\ +\ 4 \\ \hline 11 \end{array}$$

The sum is positive and is therefore in true (uncomplemented) binary.

Positive number with magnitude larger than negative number:

$$\begin{array}{rr} & 00001111 \\ & +\ 11111010 \\ \hline \text{Discard carry} \longrightarrow \boxed{1} & 00001001 \end{array} \qquad \begin{array}{r} 15 \\ +\ -6 \\ \hline 9 \end{array}$$

The final carry bit is discarded.

Negative number with magnitude larger than positive number:

$$\begin{array}{r} 00010000 \\ +\ 11101000 \\ \hline 11111000 \end{array} \qquad \begin{array}{r} 16 \\ +\ -24 \\ \hline -8 \end{array}$$

The sum is negative and therefore in 2's complement form.

Both numbers negative:

$$\begin{array}{rr} & 11111011 \\ & +\ 11110111 \\ \hline \text{Discard carry} \longrightarrow \boxed{1} & 11110010 \end{array} \qquad \begin{array}{r} -5 \\ +\ -9 \\ \hline -14 \end{array}$$

The final carry bit is discarded. The sum is negative and therefore in 2's complement form.

Overflow condition

- Overflow occurs when two numbers are added and the number of bits required to represent the sum exceeds the number of bits available in the representation
- Can happen when both numbers are positive or both numbers are negative
- Overflow is indicated if:
  - Correct result is out of the range between -2^{n-1} and $2^{n}-1$, where n is the number of bits (e.g., -127 and 128 for 8 bit)
  - the sum of two positive numbers is a negative number
  - the sum of two negative numbers is a positive number

$$
\begin{array}{r}
125_{10} \\
+\ 58_{10} \\
\hline
-73_{10}
\end{array}
\qquad
\begin{array}{r}
01111101 \\
00111010 \\
\hline
10110111
\end{array}
$$

Correct result = 183_{10}

→ Out of range

Subtraction

- Subtraction is a special case of addition
- $a - b = a + (-b)$
- $-b$ is obtained by taking the 2's complement of b

```
  3F       3F
- 23      +DD     (2's complement)
          11C      Carry = 1; Discard carry from result;
Result = 1C
```

$$\begin{array}{r} 3F \\ +DD \\ \hline \cancel{1}1C \end{array}$$

```
  23       23
- 3F      +C1     (2's complement)
           E4      Carry = 0
Result = E4
```

- Note that Carry = 1 if result is positive and Carry = 0 if result is negative
- This is important for establishing whether a and b is greater

Sign extension of signed number

- Positive number: e.g., extending the 8-bit number of 56_{16} to a 32-bit number will result in 00000056_{16}
- Negative number: e.g., extending the 8-bit number of 82_{16} to a 32-bit number will result in $FFFFFF82_{16}$
  - 82_{16} is $-7E_{16}$
  - With 32-bit, this number is represented as the 2'complement of $0000007E_{16} \rightarrow FFFFFF82_{16}$
- Thus, signed extension involves copying the signed bit (D7) to the upper 24 bits of the 32-bit register.
- Similar in extending to other number of bits > 8: Copy the signed bit into the additional bits in the new formats
  - 82_{16}, when extending to 16 bits, becomes $FF82_{16}$
  - 56_{16}, when extending to 16 bits, becomes 0056_{16}

Multiplication

- Multiplication of unsigned number by partial product

10×13	Multiplier $= 1101_2$		
	Multiplicand $= 1010_2$		
1010			
1101			
1010	Step 1 first multiplier bit	= **1**,	write down multiplicand
0000	Step 2 second multiplier bit	= **0**,	write down zeros shifted left
1010	Step 3 third multiplier bit	= **1**,	write down multiplicand shifted left
1010	Step 4 fourth multiplier bit	= **1**,	write down multiplicand shifted left
10000010	Step 5 add together four partial products		

- n-bit × n-bit → 2n-bit result

High-speed multiplication

FIGURE 2.3 An algorithm for multiplication

Step a. Set a counter to *n*.
Step b. Clear the 2*n*-bit partial product register.
Step c. Examine the rightmost bit of the multiplier (initially the least-significant bit). This bit is underlined in Table 2.3. If it is one, add the multiplicand to the *n* most-significant bits of the partial product.
Step d. Shift the partial product one place to the right.
Step e. Shift the multiplier one place to the right (the rightmost bit is, of course, lost).
Step f. Decrement the counter and repeat from step c until the count is 0 after *n* cycles. The product is in the partial product register.

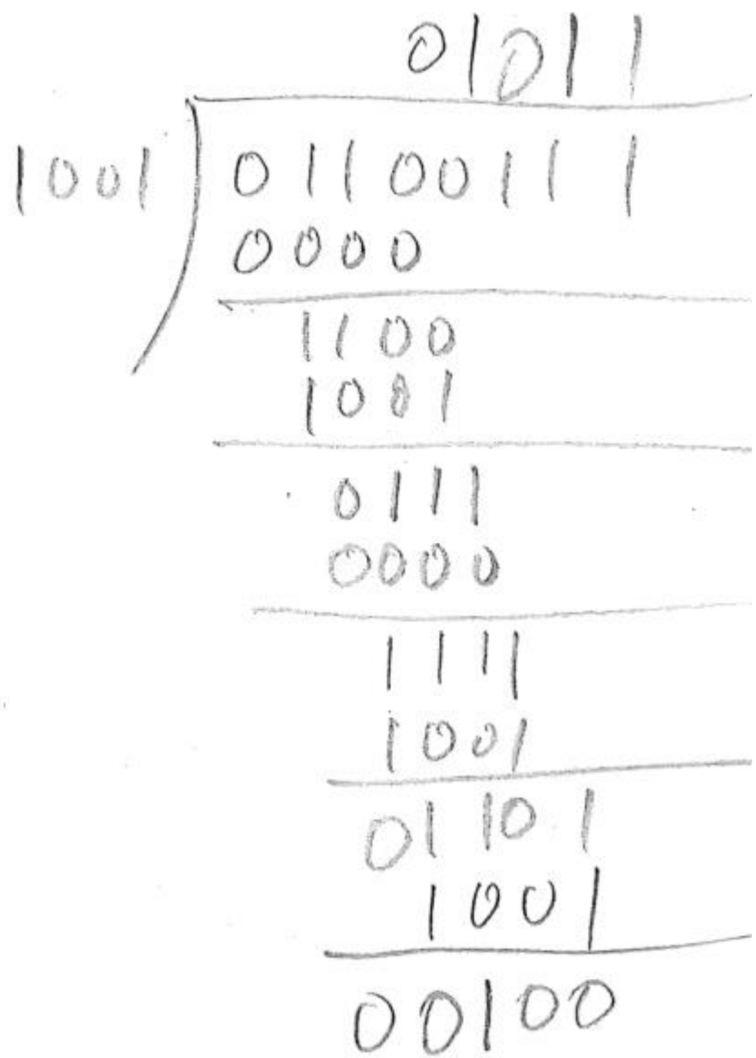
TABLE 2.3 Mechanizing Unsigned Multiplication from Figure 2.3

	Multiplier = 1101_2		Multiplicand = 1010_2		
Cycle	Step	Counter	Multiplier	Partial Product	
	a and b	4	1101	00000000	
1	c	4	110<u>1</u>	10100000	+1010
1	d and e	4	0110	01010000	
1	f	3	0110	01010000	
2	c	3	011<u>0</u>	01010000	+0000
2	d and e	3	0011	00101000	
2	f	2	0011	00101000	
3	c	2	001<u>1</u>	11001000	+1010
3	d and e	2	0001	01100100	
3	f	1	0001	01100100	
4	c	1	000<u>1</u>	10000010	+1010
4	d and e	1	0000	10000010	

10×13

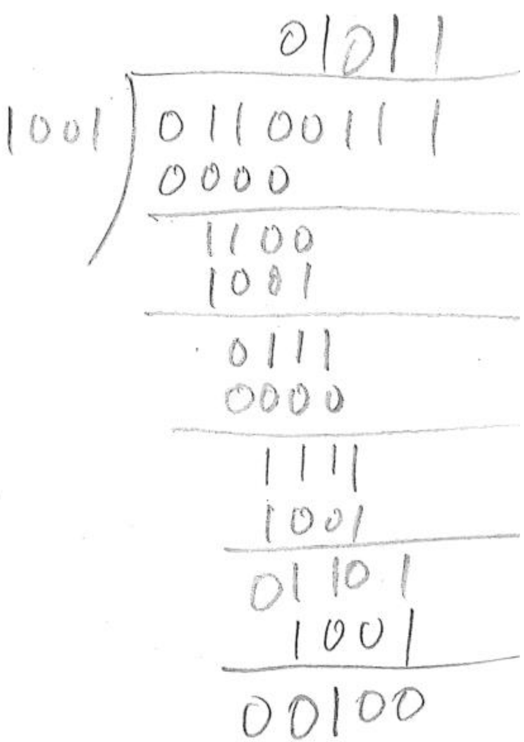
```
    1010
    1101
    1010
   0000
  1010
 1010
10000010
```

Division



- Dividend = 103_{10} = 01100111_2
- Divisor = 9_{10} = 1001_2
- Expected Result:
  - Quotient = 11_{10}
  - Reminder = 4_{10}

Restoring division



Description	Partial Dividend	Divisor	Quotient
	01100111	00001001	00000000
Align	01100111	10010000	00000000
Subtract divisor from partial dividend	-00101001	10010000	00000000
Restore divisor, shift 0 in quotient	01100111	10010000	00000000
Test for end			
Shift divisor one place right	01100111	01001000	00000000
Subtract divisor from partial dividend	00011111	01001000	00000000
Result positive—shift in 1 quotient	00011111	01001000	00000001
Test for end			
Shift divisor one place right	00011111	00100100	00000001
Subtract divisor from partial dividend	−00000101	00100100	00000001
Restore divisor, shift in 0 in quotient	00011111	00100100	00000010
Test for end			
Shift divisor one place right	00011111	00010010	00000010
Subtract divisor from partial dividend	00001101	00010010	00000010
Result positive—shift in 1 in quotient	00001101	00010010	00000101
Test for end			
Shift divisor one place right	00001101	00001001	00000101
Subtract divisor from partial dividend	00000100	00001001	00000101
Result positive—shift in 1 in quotient	00000100	00001001	00001011
Test for end			

Remainder

End: Divisor aligns with LSB of dividend

Restoring division

FIGURE 2.4 The flowchart for restoring division