OTHER NUMBER SYSTEMS:

octal (digits 0 to 7)
group three binary numbers together and represent as base 8

3564_{10} = 110 111 101 100_2
= $(6X8^3) + (7X8^2) + (5X8^1) + (4X8^0)$
= 6754_8

hexadecimal (digits 0 to 15, actually 0 to F)
group four numbers together and represent as base 16

3564_{10} = 1101 1110 1100_2
= $(DX16^2) + (EX16^1) + (CX16^0)$
= DEC_{16}

decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

hexadecimal-to-decimal conversion

sum of powers method:
converts from least significant → most significant digit
3107_{16} = $(7X16^0) + (0X16^1) + (1X16^2) + (3X16^3)$
= 7 + 0 + 256 + 12288
= 12551_{10}

multiply and add method:
converts from most significant → least significant digit
3107_{16}

(3X16) + 1 = 48 + 1 = 49
(49X16) + 0 = 784 + 0 = 784
(784X16) + 7 = 12544 + 7 = 12551_{10}

BINARY ADDITION (UNSIGNED)

	result	carry
0 + 0 =	0	0
0 + 1 =	1	0
1 + 0 =	1	0
1 + 1 =	0	1

Example:

$$\begin{array}{rr} & 011110 \quad \leftarrow \text{carrys} \\ & 101101_2 \\ + & \underline{100111_2} \\ \text{overflow}\rightarrow\ 1 & 010100_2 \end{array}$$

HEX ADDITION

Add as if decimal numbers except:

1. if individual numbers are A thru F, convert to decimal equivalents and add
2. if sum is greater than 16, replace sum with sum-16 and carry 1
3. if sum is between 10 and 15 (decimal), replace sum by equivalent hexadecimal digit

$$\begin{array}{rr} & D_{16} \\ + & C_{16} \\ \hline 1 & 9_{16} \end{array} \quad = \quad \begin{array}{rr} & 13_{10} \\ + & 12_{10} \\ \hline & 25_{10} \end{array}$$

$$\begin{array}{rcccc} \text{CARRY'S}\rightarrow & 1 & & 1 & \\ & D & F & 6 & D \\ + & 2 & 4 & 6 & C \\ \hline \end{array}$$

```
                  1   0   3   D   9
OVERFLOW  ↑
```

REMEMBER:

decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

EXAMPLES FROM TEXT OF BINARY AND HEX ADDITION

10(a)

CARRY'S→		1	1	1	1	1	
		1	1	1	0	1	1
	+	1	0	0	1	1	1
	1	1	0	0	0	1	0
OVERFLOW	↑						

10(e)

CARRY'S→		1			1	
		1	1	0	0	1
	+	0	1	1	0	1
	1	0	0	1	1	0
OVERFLOW	↑					

11(a)

	1	2	3	5	
+	5	6	7	A	
	6	8	A	F	

11(b)

CARRY'S→		1	1	1		
		A	9	8	7	6
	+	F	D	C	A	0
	1	A	7	5	1	6
OVERFLOW	↑					

REMEMBER:

decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

fixed length binary numbers:

- computer arithmetic is performed on data stored in fixed-length memory locations, typically 8, 16 or 32 bits

BYTE	n=8 bits, $N \le 2^8 - 1 = 127_{10}$
WORD	n=16 bits, $N \le 2^{16} - 1 = 32{,}767_{10}$
LONG WORD	n=32 bits, $N \le 2^{32} - 1 = 1{,}073{,}676{,}289_{10}$

- no signed numbers (no sign bit)

An unsigned 8-bit binary number

$d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0$

example conversions between number systems

decimal	binary	hex
255	1111 1111	FF
93	0101 1101	5D

Special problems occur in manipulating fixed length numbers

DECIMAL		BINARY		HEX
201	=	1 1 0 0 1 0 0 1	=	C 9
+ 79	=	+ 0 1 0 0 1 1 1 1	=	+ 4 F
280	=	1 0 0 0 1 1 0 0 0	=	1 1 8
		↑		↑

OVERFLOWS

The number 280_{10} is larger than the maximum 8-bit number; this results in a carry beyond 8-bits in the binary representation and a carry beyond two digits in the hexadecimal representation. When doing arithmetic using fixed length numbers these carrys are potentially lost.

How do you represent negative numbers?

- signed magnitude:

  use one bit for sign, 7 bits for number

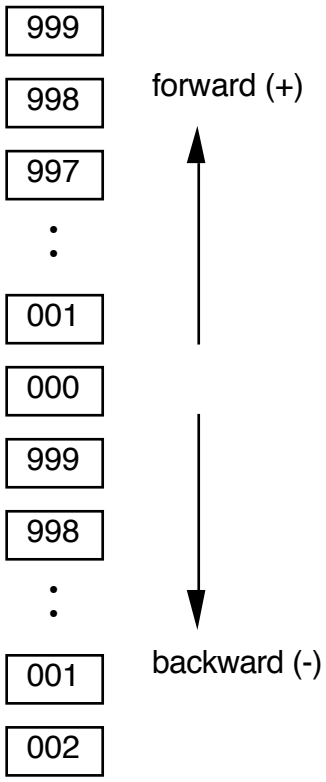
  Example: -1 (in an eight bit system) could be $1000\ 0001_2$

- 2's complement (most often used in computing)

  Example: the 2's complement number representing -1 (in an eight bit system) would be $1111\ 1111_2$ (more on this on next page)

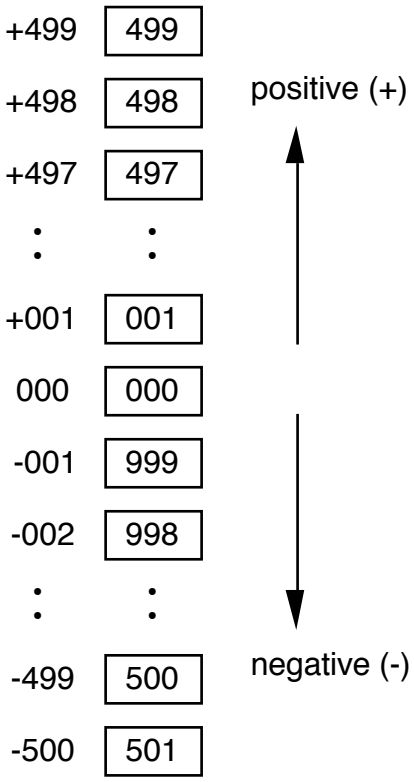
What are complement numbers?

Consider odometer numbers on a bicycle:



The problem is how to tell a +998 from a -2 (also a 998)?

NOTE: real odometers on cars don't behave this way!



The solution is to cut the number system in half, i.e. use 001 - 499 to represent positive numbers and 500 - 999 to represent negative numbers.

This is a fixed length signed number system

DECIMAL EXAMPLES

ODOMETER NUMBERS ARE USEFUL FOR BOTH ADDITION AND SUBTRACTION (COMPLEMENTS) OF SIGNED NUMBERS

- 2	+ 998
+ 3	+ 003
+ 1	1 001

In the second example, the correct result is just the +001, the overflow is ignored in fixed-length complement arithmetic.

Do subtraction as addition by using complements, i.e.

$$A - B = A + (-B)$$

IMPORTANT: It is easier to compute -B and add than to subtract B from A directly.

EXAMPLE:

- 005	→	+ 995
- 003	→	+ 997
- 008	→	1 992

Note that 995 and 997 were added in the normal fashion, the overflow was ignored, and the result is 992 which can be converted from the complement (or odometer) system back to -8, the correct answer.

signed	3-bit complement
+3	003
+2	002
+1	001
0	000
-1	999
-2	998
-3	997

-4	996
-5	995
-6	994
-7	993
-8	992

decimal	2's complement binary
+127	0 1 1 1 1 1 1 1
+126	0 1 1 1 1 1 1 0
+125	0 1 1 1 1 1 0 1
+2	0 0 0 0 0 0 1 0
+1	0 0 0 0 0 0 0 1
0	0 0 0 0 0 0 0 0
-1	1 1 1 1 1 1 1 1
-2	1 1 1 1 1 1 1 0
-127	1 0 0 0 0 0 0 1
-128	1 0 0 0 0 0 0 0

There are 256 numbers in an 8-bit fixed length 2's complement number system.

Why are these numbers called 2's complement?

$$M - N = M + (-N)$$

where -N is the complement of N

Claim: $-N = (2^8-1) + N + 1$ (this is called complementing)
$(2^8-1) + N$ produces the 1's complement
$(2^8-1) + N+1$ produces the 2's complement

Example:

DECIMAL			BINARY								
+ 1	=		0	0	0	0	0	0	0	1
- 1	=	+	1	1	1	1	1	1	1	1
0	=		1	0	0	0	1	1	0	0	0

↑

$2^8=256$

NOW WE NEED AN EASY WAY TO DO 2'S COMPLEMENT OPERATIONS.

$256 - N$	=	$2^8 - N$	=	$((2^8-1) - N) + 1$
This is complementing where N is an 8-bit fixed length binary number.				The trick is to re-arrange the operation into something easier to compute.

Now, consider $2^8-1 = 1111\ 1111_2$

	1	1	1	1	1	1	1	1
-								N
					(intermediate result)			
+								1
						(final result)		

Subtracting N from (2^8-1) amounts to flipping all the bits in N.

Example:

What is the representation of -27_{10} in 2's complement 8-bit notation?

2^8-1		1	1	1	1	1	1	1	1	always all 1's
(-)27_{10}	-	0	0	0	1	1	0	1	1	
		1	1	1	0	0	1	0	0	corresponds to flipping the bits; also known as 1's complement
add 1	+	0	0	0	0	0	0	0	1	
result		1	1	1	0	0	1	0	1	-27_{10} in 2's complement representation

This approach is necessary because:

1. Fixed length nature of stored computer numbers
2. More computationally efficient to implement a fast adder than an adder AND a subtractor.

Examples of 2's complement number operations:

Addition and subtraction in 2's complement form

Addition: To compute N1 + N2 add N1 to N2

Subtraction: To compute N1 - N2 add N1 to -N2

Examples:

DECIMAL	BINARY		HEX
11	0 0 0 0 1 0 1 1	=	0 0 0 B
+ 21	0 0 0 1 0 1 0 1	=	0 0 1 5
32	0 0 1 0 0 0 0 0	=	0 0 2 0

DECIMAL	BINARY		HEX
21	0 0 0 1 0 1 0 1	=	0 0 1 5
- 11	1 1 1 1 0 1 0 1	=	F F F 5
10	0 0 0 0 1 0 1 0	=	0 0 0 A

DECIMAL	BINARY		HEX
11	0 0 0 0 1 0 1 1	=	0 0 0 B
- 21	1 1 1 0 1 0 1 1	=	F F E B
- 10	1 1 1 1 0 1 1 0	=	F F F 6

DECIMAL	BINARY		HEX
- 11	1 1 1 1 0 1 0 1	=	F F F 5
- 21	1 1 1 0 1 0 1 1	=	F F E B
- 32	1 1 1 0 0 0 0 0	=	F F E 0

How we got the 2's complements of -11 and -21:

11		0 0 0 0 1 0 1 1	
~11	-	1 1 1 1 0 1 0 0	1's complement
+1	+	0 0 0 0 0 0 0 1	add 1
-11		1 1 1 1 0 1 0 1	2's complement representation

21		0 0 0 1 0 1 0 1	
~21	-	1 1 1 0 1 0 1 0	1's complement
+1	+	0 0 0 0 0 0 0 1	add 1
-21		1 1 1 0 1 0 1 1	2's complement representation

ALGORITHM:

1. Store N
2. Obtain ~N, the 1's complement of N by replacing (bit by bit) every 0 with a 1 and vice versa in the number's binary representation.
3. Add 1 and ignore any carry.

NOTE: This algorithm works in hex by replacing each digit x by it's hex complement, i.e. 15-x. Example: The hex equivalent of 11 is $000B, it's hex complement is then $FFF4 where each digit was computed as $F-x. Adding 1 to $FFF4 gives the result $FFF5 for the 2's complement of 11.

Example:

In binary:

N		0	0	0	0	0	1	1	0	0	1	0	0	0	1	1	1	
~N	-	1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	0	1's complement
+1	+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	add 1
-N		1	1	1	1	1	0	0	1	1	0	1	1	1	0	0	1	2's complement representation

In hex:

N		0	6	4	7	
~N	-	F	9	B	8	1's complement
+1	+	0	0	0	1	add 1
-N		F	9	B	9	2's complement representation

A calculator will always give you the 2's complement directly.

The Most Significant Bit (MSB) is the binary digit corresponding to the largest power of 2 and is always 1 for a negative 2's complement number. As a result it is often called the sign bit.

In 2's complement, -0 = +0. Technically, 0 is the complement of itself.

N		0	0	0	0	0	0	0	0	
~N	-	1	1	1	1	1	1	1	1	1's complement

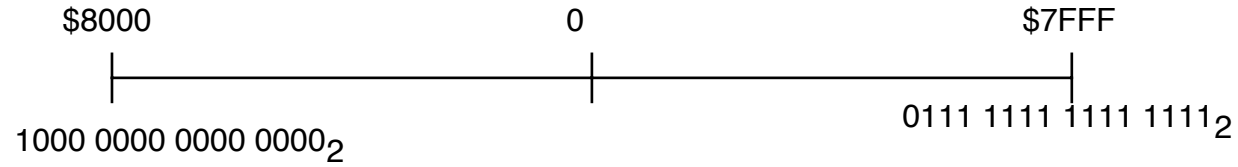
+1	+	0	0	0	0	0	0	0	1	add 1
-N		0	0	0	0	0	0	0	0	2's complement representation

Problems with fixed length arithmetic:

- overflow
- underflow

Adding signed numbers can easily exceed these limits

For a 16-bit fixed length numer system



Adding signed numbers can easily exceed these limits

First digit

```
0 0 1 1      $3 0 0 0
0 1 1 0      $6 0 0 0
             ---------
             $9 0 0 0
```

This result, from adding two positive numbers in the number system, results in $9000 which is larger than $7FFF, the largest allowed positive number. In fact, $9000 = 1001 0000 0000 0000_2 which is a negative number. The sign of the result is different from that of the operands.

RULE: If there is a sign change in adding two positive 2's complement numbers, then SIGNED OVERFLOW has occurred.

We can generalize these rules to signed addition and subtraction,

	possible overflow?	comments
positive + positive	yes	sign change, result negative
negative + negative	yes	sign change, result positive
positive + negative	no	not with fixed length numbers
negative - positive	yes	result positive
positive - negative	yes	result negative, basically adding two negative numbers
negative - negative	no	
positive - positive	no	

How about if numbers are of different length?

decimal 2's complement binary

	3 bit	4 bit	8-bit
+3	0 1 1	0 0 1 1	0 0 0 0 0 0 1 1
+2	0 1 0	0 0 1 0	0 0 0 0 0 0 1 0
+1	0 0 1	0 0 0 1	0 0 0 0 0 0 0 1
0	0 0 0	0 0 0 0	0 0 0 0 0 0 0 0
-1	1 1 1	1 1 1 1	1 1 1 1 1 1 1 1
-2	1 1 0	1 1 1 0	1 1 1 1 1 1 1 0
-3	1 0 1	1 1 0 1	1 1 1 1 1 1 0 1
-4	1 0 0	1 1 0 0	1 1 1 1 1 1 0 0

SIGN EXTENSION:
To extend a 2's complement number to a greater number of binary digits, you just continue the sign bit to the left.

Examples:

Extend $9B to 16-bits
$9B = 1001 1011_2 = 1111 1111 1001 1011_2 = $ FF 9B

Extend $5F to 16-bits
$5F = 0101 1111_2 = 0000 0000 0101 1111_2 = $ 00 5F

Adding two 2's complement numbers of different length:

```
4 3 A 0          3 A 0
                 4
    9 B          F F 9 B  ←need to sign extend.
                          You can't just add zeros
---------        ---------
? ? ? ?        1 4 3 3 B
```

What is $FF 9B and $9B? Claim: $ 9B = -101_{10}

101	0 0 6 5	0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1
~101	F F 9 A	1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 0
+1	0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
-101	F F 9 B	1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1

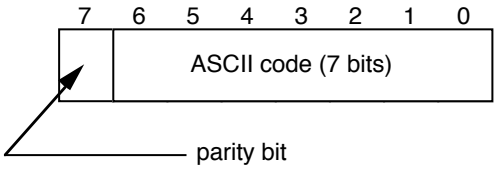
Note that the 8-bit $9B and the 16-bit $FF9B both represent -101 in their respective number systems.

CHARACTER REPRESENTATION

ASCII American Standard Code for Information Interchange

This code defines the following characters:

letters	A,B,...,Z,a,b,...,z
digits	0,1,2,3,4,5,6,7,8,9
special characters	+,*,/,@,$,<space>, ...
non-printing characters	bell,line feed (LF), carriage return (CR),...



ASCII uses 8 bits to represent characters. Actually, only 7 bits are used to uniquely define the character and the 8-th bit (called the parity bit) is used for error detection. When used, the value of the parity bit depends upon the numbers of 1's in bits 0-7. For odd parity, bit 8 is set to make the total number of 1's in the byte an odd number such as 1 or 7. For even parity, bit 8 is set to make the total number of 1's in the byte an even number such as 0, 2 or 8.

Some useful ASCII character codes:

character	ASCII code (in hex)
/	2F
0	30
1	31
2	32
8	38
9	39
:	3A
;	3B
@	40
A	41
B	42
Z	5A
[	5B
\	60
a	61

z	7A
{	7B

etc.

LOGIC OPERATORS

- used for decision making, conditional tests, etc.
- OR, AND, XOR, NOT

OR designated as A+B If A or B is 1, then A+B=1 else A+B=0

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

AND designated as AB If A and B is 1, then AB=1 else AB=0

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

XOR designated as A B Used to differentiate which bits are the same and which bits are different between two binary numbers.

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

NOT designated as ~A If A is 1, then ~A=0; If A is 0, then ~A=1

A	~A
0	1
1	0

Logic functions can be combined.

$\bar{f}$ = NOT (f) = ~f

Consider the logic function $\overline{\overline{A}\bullet\overline{B}}$

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}\bullet\overline{B}$	$\overline{\overline{A}\bullet\overline{B}}$	A + B
0	0	1	1	1	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	0	1	1

Note that the last two columns are the same.

DeMorgan's theorems:

$$\overline{A\bullet B} = \overline{A} + \overline{B}$$

$$\overline{A + B} = \overline{A}\bullet\overline{B}$$

Basically, these theorems say that we can generate all logic functions with combinations of OR and NOT.

Representation of numbers in binary:

(Ford & Topp call this the expanded form representation of a number)

$10_2 = 1\times2^1 + 0\times2^0 = 2_{10}$

$11_2 = 1\times2^1 + 1\times2^0 = 2 + 1 = 3_{10}$

How can you convert numbers from decimal to binary?

Subtraction of powers method:

Example: Convert $N = 217_{10}$ to $(D_7D_6D_5D_4D_3D_2D_1D_0)_2$

You can represent up to 256 using eight bits.

Want $N = 217_{10} = D_7\times2^7 + D_6\times2^6 + D_5\times2^5 + D_4\times2^4 + D_3\times2^3 + D_2\times2^2 + D_1\times2^1 + D_0\times2^0 = (D_7D_6D_5D_4D_3D_2D_1D_0)_2$

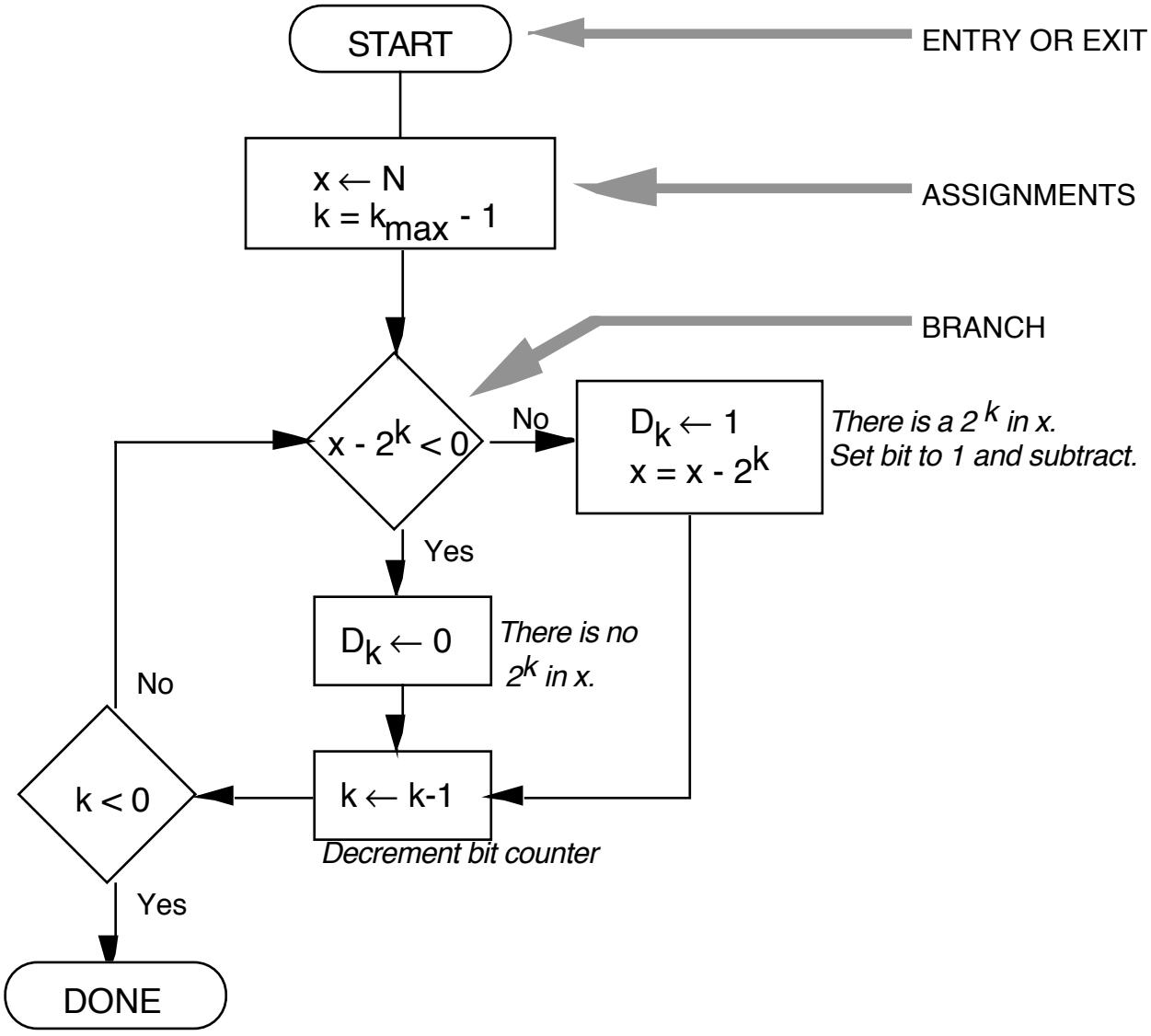
Test each bit (starting from the most significant)

$217_{10} - 2^7$	$= 217_{10} - 128_{10} = 89_{10} > 0$	$\rightarrow D_7=1$
$89_{10} - 2^6$	$= 89_{10} - 64_{10} = 25_{10} > 0$	$\rightarrow D_6=1$
$25_{10} - 2^5$	$= 25_{10} - 32_{10} < 0$	$\rightarrow D_5=0$
$25_{10} - 2^4$	$= 25_{10} - 16_{10} = 9_{10} > 0$	$\rightarrow D_4=1$
$9_{10} - 2^3$	$= 9_{10} - 8_{10} = 1_{10} > 0$	$\rightarrow D_3=1$
$1_{10} - 2^2$	$= 1_{10} - 4_{10} < 0$	$\rightarrow D_2=0$
$1_{10} - 2^1$	$= 1_{10} - 2_{10} = < 0$	$\rightarrow D_1=0$
$1_{10} - 2^0$	$= 1_{10} - 1_{10} = 0$	$\rightarrow D_0=1$

Therefore, $217_{10} = 11011001_2$

FLOWCHART (representing the subtraction of powers method)

N = STARTING NUMBER (the number to convert)
k_{max} = # of binary digits (8)



In pseudo code the same algorithm can be documented as:

```
x = number_tobe_conv
k = number_digits - 1
(* number_digits = 8 do loop starts at 7 *)

WHILE k≥0 do
      BEGIN
      if (x-2^k)<0 THEN d(k)=0
          ELSE
          BEGIN
          d(k) = 1
          x = x-2^k
          END
      k=k-1
      END
```

trace loop:
setup:
x=217
k=7
looping:
k=7
    $217-2^7=89$
    d(7)=1
    x=89
k=6
    $89-2^6=25$
    d(6)=1
    x=25
k=5
    $25-2^5=-7$
    d(5)=0
k=4
    $25-2^4=9$
    d(4)=1
    x=9
k=3
    $9-2^3=1$
    d(3)=1
    x=1
k=2
    $1-2^2=-3$
    d(2)=0
k=1
    $1-2^1=-2$
    d(1)=0
k=0
    $1-2^0=0$
    d(0)=1
    x=0

Even Odd decimal-to-binary conversion method:

Example:
Convert N = 217_{10} to $(D_7D_6D_5D_4D_3D_2D_1D_0)_2$

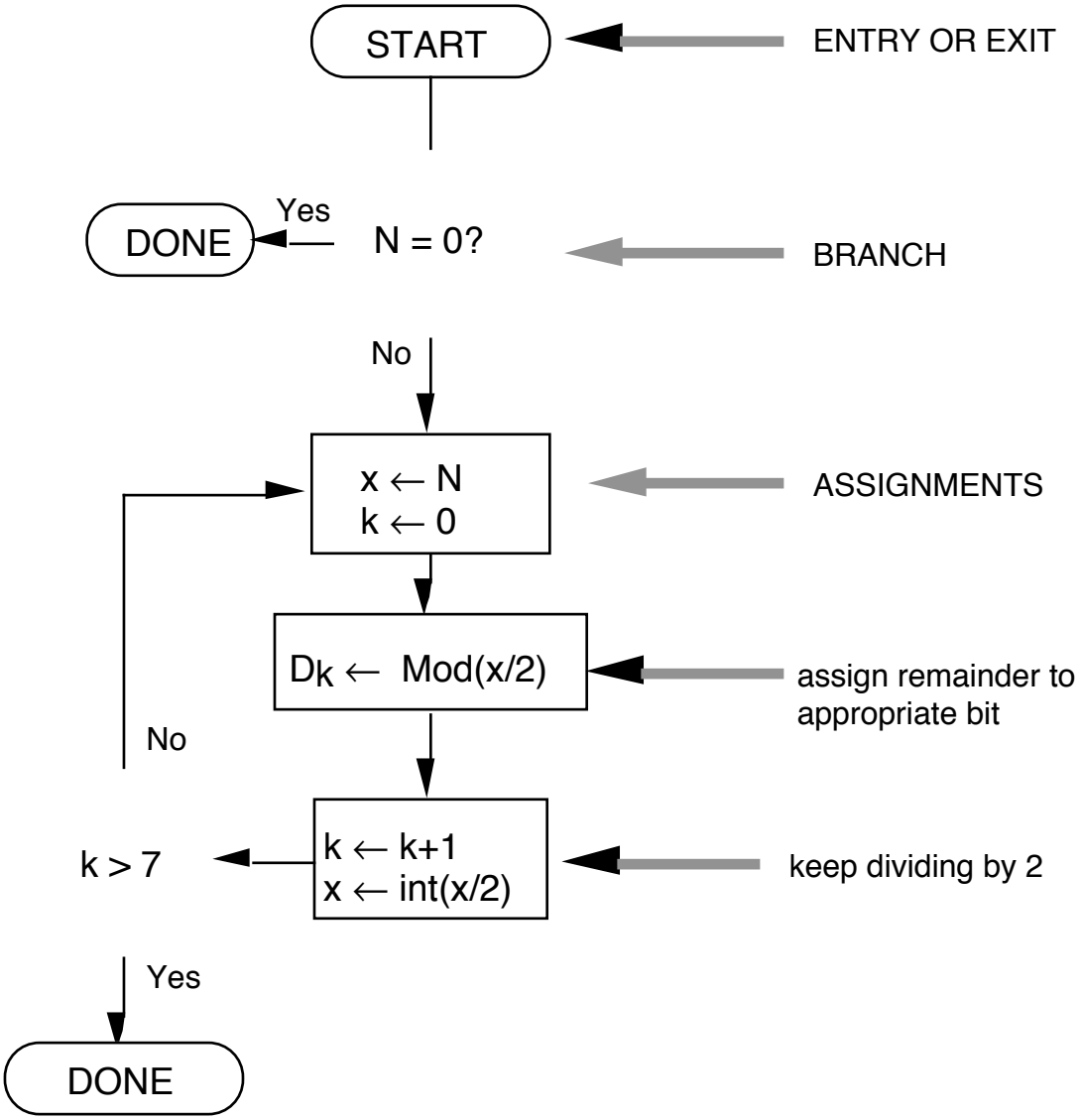
Test each bit (starting from the least significant)

$217_{10}/2$	= 108 with remainder=1	$\rightarrow D_0=1$
$108_{10}/2$	= 54 with remainder=0	$\rightarrow D_1=0$
$54_{10}/2$	= 27 with remainder=0	$\rightarrow D_2=0$
$27_{10}/2$	= 13 with remainder=1	$\rightarrow D_3=1$
$13_{10}/2$	= 6 with remainder=1	$\rightarrow D_4=1$
$6_{10}/2$	= 3 with remainder=0	$\rightarrow D_5=0$
$3_{10}/2$	= 1 with remainder=1	$\rightarrow D_6=1$
$1_{10}/2$	= 0 with remainder=1	$\rightarrow D_7=1$

Therefore, $217_{10} = 11011001_2$

FLOWCHART (representing the even-odd method)

N = STARTING NUMBER
kmax = # of binary digits (8)



In pseudo code the same algorithm can be documented as:

```
N = number_tobe_conv
x = quotient

IF N≥0 THEN
      BEGIN
      x=N
      k=0
      WHILE k<8 DO
          BEGIN
          d(k) = mod(x,2)
          x = int(x/2)
          k=k+1
          END
      END
```

trace loop:	setup:
	x=217
	k=0
looping:	
k=1	
d(0) = mod(217,2) = 1	
x=int(217,2) = 108	odd
k=2	
d(1) = mod(108,2) = 0	
x=int(108,2) = 54	even
k=3	
d(1) = mod(54,2) = 0	
x=int(54,2) = 27	even
k=4	
d(1) = mod(27,2) = 1	
x=int(27,2) = 13	odd
k=5	
d(1) = mod(13,2) = 1	
x=int(13,2) = 6	odd
k=6	
d(1) = mod(6,2) = 0	

```
    x=int(6,2) = 3                  even
k=7
    d(1) = mod(3,2) = 1
    x=int(3,2) = 1                  odd
k=8
    d(1) = mod(1,2) = 1
    x=int(1,2) = 0                  odd
```

How about fractions?

No one said we couldn't have negative powers of two!

$$\begin{aligned} 0.1101_2 &= 0\times2^0 + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4} \\ &= 1\times0.5 + 1\times.25 + 0\times.125 + 1\times.0625 \\ &= .5 + .25 + 0 + .0625 \\ &= 0.8125_{10} \end{aligned}$$

Binary $\rightarrow$ decimal conversion is always exact

Decimal $\rightarrow$ binary conversions are NOT always exact

Example:

0.42357_{10} @ $0.011011000..._2$ (binary numbers typically do not terminate)

Example: (using subtraction of powers conversion):

$0.42357_{10} - 2^{-1} = .42357-.5 < 0 \Rightarrow d_{-1}=0$

$0.42357_{10} - 2^{-2} = .42357-.25=.17357 \Rightarrow d_{-2}=1$

$0.17357_{10} - 2^{-3} = .17357-.125=.04857 \Rightarrow d_{-3}=1$

$0.04857_{10} - 2^{-4} = .04857-.0625 < 0 \Rightarrow d_{-4}=0$

$0.04857_{10} - 2^{-5} = .04857-.03125 = 0.01732 \Rightarrow d_{-5}=1$

$0.01732_{10} - 2^{-6} = .01732-.015625 = 0.001695 \Rightarrow d_{-6}=1$

$0.001695_{10} - 2^{-7} < 0 \Rightarrow d_{-7}=0$

$0.001695_{10} - 2^{-8} < 0 \Rightarrow d_{-8}=0$

$0.001695_{10} - 2^{-9} < 0 \quad \Rightarrow \quad d_{-9}=0$