Number Systems, Computer Arithmetic and Character Systems

Joannah Nanjekye

July 09, 2024

Decimal System

Consider a whole number: 62_{10}

- $62_{10} = (60 \times 10^{1}) + (2 \times 10^{0})$
- $62_{10} = 111110_{2}$

Consider a fraction number: 0.456_{10}

- $0.456_{10} = (4 \times 10^{-1}) + (5 \times 10^{-2}) + (6 \times 10^{-3})$
- $0.81_{10} = 0.110011_{2}$

The right had side is the Least Significant Bit while the left hand side is the Most Significant Bit

Binary System

- $111110_2 = (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 62_{10}$
- $1001.101 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$

Hexadecimal System

- Binary digits are grouped into sets of four bits, called a nibble
- $00010001_2 = 11_{16}$
- $41_{16} = 00100001_2$
- $41_{16} = (4 \times 16^1) + (1 \times 16^0) = 65_{10}$

$0000 = 0$	$0100 = 4$	$1000 = 8$	$1100 = C$
$0001 = 1$	$0101 = 5$	$1001 = 9$	$1101 = D$
$0010 = 2$	$0110 = 6$	$1010 = A$	$1110 = E$
$0011 = 3$	$0111 = 7$	$1011 = B$	$1111 = F$

Hexadecimal System

- Why the hexadecimal notation is preferred:
  - It is more compact than binary notation
  - In most computers, binary data occupy some multiple of 4 bits, and hence some multiple of a single hexadecimal digit
  - It is extremely easy to convert between binary and hexadecimal notation

Integer Representation

- Consider an 8-bit word
- It can represent positive numbers from 0 to 255
- And negative numbers from -127 to +128
- This is because the the Most Significant Bit will represent the sign

Sign Magnitude Representation

- In an n-bit word, the right most n - 1 bits hold the magnitude of the integer:
  - +18 = 00010010
  - -18 = 10010010
- Limitations of the sign magnitude representation
  - Arithmetic has to consider both the sign and magnitude
  - There are two representations of zero (+0 and -0)

twos Complement

- +3 = 00000011
- +2 = 00000010
- +1 = 00000001
- +0 = 00000000
- -1 = 11111111
- -2 = 11111110
- -3 = 11111101

Integer Arithmetic: Negation

- In twos complement negation is achieved in two steps:
  - compute the Boolean complement of each bit of the integer
  - Treat he result as an unsigned binary integer, add 1
- +18 = 00010010
- bit wise complement = 11101101
- 11101101 + 1 = 11101110 = -18
- Negation is referred to as the **twos complement operation**
- There are two special cases 0 and 128 for an 8-bit representation:
  - 0 = 00000000 twos complement + 1 = [1]00000000 = 0 (discard carried 1)
  - +128 = 10000000 twos complement + 1 = 10000000 = -128

Integer Arithmetic: Addition

- Change negative numbers to twos complement
- Treat positive numbers as unsigned
- The result is in twos complement form
- Whether it is positive or negative

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Integer Arithmetic: Addition and Overflow

- The result may be larger than the word size
- This is called an **overflow**
- When adding two numbers, if they are both positive or both negative, an overflow occurs if and only if the result has the opposite sign

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Integer Arithmetic: Subtraction

Subtraction is achieved by performing addition. The number being subtracted (subtrahend) should be converted to twos complement before being added to the minuend

```
      0010 =  2
     +1001 = −7
      1011 = −5

(a) M = 2 = 0010
    S = 7 = 0111
   −S =     1001
```

```
      0101 =  5
     +1110 = −2
     10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   −S =     1110
```

```
      1011 = −5
     +1110 = −2
     11001 = −7

(c) M = −5 = 1011
    S =  2 = 0010
   −S =      1110
```

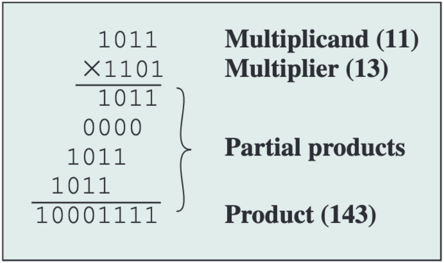
```
      0101 = 5
     +0010 = 2
      0111 = 7

(d) M =  5 = 0101
    S = −2 = 1110
   −S =      0010
```

```
      0111 = 7
     +0111 = 7
      1110 = Overflow

(e) M =  7 = 0111
    S = −7 = 1001
   −S =      0111
```

```
      1010 = −6
     +1100 = −4
     10110 = Overflow

(f) M = −6 = 1010
    S =  4 = 0100
   −S =      1100
```

Unsigned: Multiplication

- Generate partial products which are summed to get the final product
- Each successive partial product is shifted one position to the left relative to the preceding partial product
- The multiplication of two n-bit binary integers results in a product of up to 2n bits in length (e.g., 11 * 11 = 1001)



Unsigned: Multiplication

There are two approaches:

- perform a running addition on the partial products (saves storage)
- For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required

C	A	Q	M		
0	0000	1101	1011	**Initial values**	
0	1011	1101	1011	**Add**	**First cycle**
0	0101	1110	1011	**Shift**	
0	0010	1111	1011	**Shift**	**Second cycle**
0	1101	1111	1011	**Add**	**Third cycle**
0	0110	1111	1011	**Shift**	
1	0001	1111	1011	**Add**	**Fourth cycle**
0	1000	1111	1011	**Shift**	

Signed: Multiplication

One alternative is to:

- convert both multiplier and multiplicand to positive numbers
- perform the multiplication
- compute the twos complement of the result

We can use Booth's algorithm instead.

```
     0111
   × 0011     (0)
 11111001     1-0
 0000000      1-1
 000111       0-1
 00010101     (21)
```

(a) $(7) \times (3) = (21)$

```
     0111
   × 1101     (0)
 11111001     1-0
 0000111      0-1
 111001       1-0
 11101011     (-21)
```

(b) $(7) \times (-3) = (-21)$

```
     1001
   × 0011     (0)
 00000111     1-0
 0000000      1-1
 111001       0-1
 11101011     (-21)
```

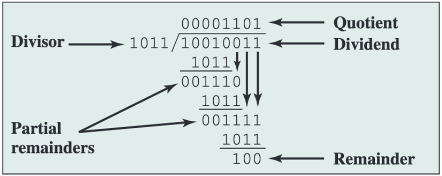
(c) $(-7) \times (3) = (-21)$

```
     1001
   × 1101     (0)
 00000111     1-0
 1111001      0-1
 000111       1-0
 00010101     (21)
```

(d) $(-7) \times (-3) = (21)$

Unsigned: Division

- Examine the bits of the dividend from left to right
- Stop when the bits examined correspond to number greater than or equal to the divisor
- Before the condition, place 0s in the quotient from left to right
- After the condition, place a 1 in the quotient and the divisor is subtracted from the partial dividend



Signed: Division

1. Compute he twos complement of the divisor
2. Express the dividend as a 2n-bit positive number, e.g 4-bit 0111 becomes 00000111
3. Shift the dividend left 1 bit position
4. Subtract the divisor from the dividend
5. If the result is positive, MSB = o, then dividend is 1
6. If the result is negative, MSB = 1, then dividend is 0
7. Repeat steps 2 through 4 as many times as there are bit positions in the dividend

Signed: Division

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Limitations of the twos Complement

- Very large and very small numbers can not be represented
- The fractional part of the quotient in a division of two large numbers could be lost

The Scientific Notation

- Move the decimal point to a convenient location
- Use the exponent of 10 to keep track of that decimal point
- This approach can be used on binary numbers as follows:

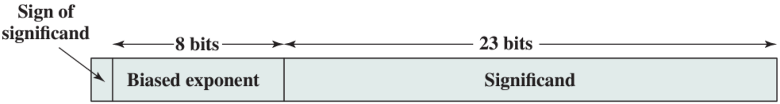
$$\pm S \times B^{\pm E}$$

Three fields are used to store the binary word of a number:

- Sign: plus or minus
- Significand S
- Exponent E
- Base B

Floating-point Representation

- The leftmost bit stores the sign of the number
- The exponent value is stored in the next 8 bits
- The representation used is known as a biased representation
- Bias = $(2^{k-1} - 1) = 2^7 - 1 = 127$, k is the number of bits
- The final portion of the word (23 bits in this case) is the significand



$$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$$
$$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$$
$$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$$
$$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$$

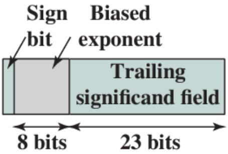
IEEE Binary Floating-point Representation

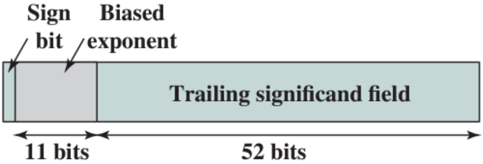
Defines the following format:

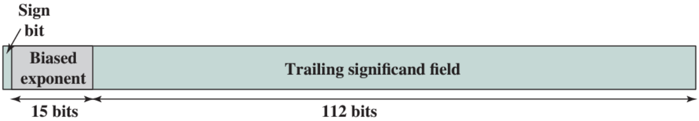
- **Arithmetic format:** all the mandatory operations defined by the standard are supported by the format
- **Basic format:** covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic
- **Interchange format:** fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage

Each of the formats have bit lengths of 32, 64, and 128 bits and exponents of 8, 11, and 15 bits

IEEE Binary Floating-point Representation







Floating-point Arithmetic

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$

$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$

$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$

$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$

$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

Addition and Subtraction

- Check for zeros
- Align the significands
- Add or subtract the significands
- Normalize the result

Guard Bits

- The ALU loads the exponent and significand before a floating point operation
- The length of the register is almost always greater than the length of the significand plus an implied bit
- The register contains additional bits, called guard bits
- To pad out the right end of the significand with 0s

$$
\begin{array}{rl}
x = & 1.000.....00 \times 2^{1} \\
\underline{-y} = & \underline{0.111.....11} \times 2^{1} \\
z = & 0.000.....01 \times 2^{1} \\
= & 1.000.....00 \times 2^{-22}
\end{array}
$$

(a) Binary example, without guard bits

$$
\begin{array}{rl}
x = & .100000 \times 16^{1} \\
\underline{-y} = & \underline{.0FFFFF} \times 16^{1} \\
z = & .000001 \times 16^{1} \\
= & .100000 \times 16^{-4}
\end{array}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{array}{rl}
x = & 1.000.....00\ 0000 \times 2^{1} \\
\underline{-y} = & \underline{0.111.....11\ 1000} \times 2^{1} \\
z = & 0.000.....00\ 1000 \times 2^{1} \\
= & 1.000.....00\ 0000 \times 2^{-23}
\end{array}
$$

(b) Binary example, with guard bits

$$
\begin{array}{rl}
x = & .100000\ 00 \times 16^{1} \\
\underline{-y} = & \underline{.0FFFFF\ F0} \times 16^{1} \\
z = & .000000\ 10 \times 16^{1} \\
= & .100000\ 00 \times 16^{-5}
\end{array}
$$

(d) Hexadecimal example, with guard bits

Rounding

- This is where extra bits in a floating-point format number are removed
- To generate a number that is close to the original number
- Alternative approaches to rounding:
  - **Round to nearest:** rounded to the nearest representable number
  - **Round toward $+\infty$:** rounded up toward plus infinity
  - **Round toward $-\infty$:** rounded up toward minus infinity
  - **Round toward 0:** rounded toward zero

Character Systems

- We deal with both symbolic alphabetic and numeric data
- Systems to encode characters as binary numbers are required for a computer
- Common systems include:
  - ASCII (American Standard Code for Information Interchange)
  - EBCDIC (Extended Binary Coded Decimal Interchange Code)
  - UNICODE (A unique number is provided for each character)

ASCII

"One" = O = 0x4F, n = 0x6E, e = 0x65

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL