Number System for CS2100

WANG Xiyu

September 14, 2024

Contents

1 Data Representation

Data representation depends on the data type. For example, the same binary '01000110' represents 70 as an integer, but 'F' as a character. The most basic unit for internal data representation is bit, which is either 0 or 1. A byte contains 8 bits, a nibble contains 4 bits etc. Multiples of bytes such as 1 byte, 2 bytes and 4 bytes are 'words', depending on the specific computer architecture.

Since N bits can represent up to 2^N values, to represent M values,

$$\lceil \log_2 M \rceil$$

bits are required.

2 Decimal Number System

A weighted-positional number system with base (or radix) 10. Consists of symbols = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

$$(a_n a_{n-1} ... a_0 . f_1 f_2 ... f_m)_{10} =$$

$$(a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + ... + (a_0 \times 10^0) + (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + ... + (f_m \times 10^{-m})$$

3 Other Number Systems

In C, prefix 0 suggests octal based number, e.g. 032 represents $(32)_8$ Prefix 0x for hexadecimals. e.g. 0x32 represents $(32)_16$

In QTSpim, a MIPS simulator, prefix 0x represents hexadecimals too.

In Verilog, prefix 8'b represents 8-bit binary values, e.g. 8'b11110000 represents binary 11110000; prefix 8'h for hexadecimals, and 8'd for decimals.

Base-R to Decimal Conversion

$$(a_n a_{n-1} ... a_0 . f_1 f_2 ... f_m)_r =$$

$$((a_n \times r^n) + (a_{n-1} \times r^{n-1}) + ... + (a_0 \times r^0) + (f_1 \times r^{-1}) + (f_2 \times r^{-2}) + ... + (f_m \times r^{-m}))_{10}$$

Decimal to Base-R Conversion

- Whole numbers: repeatedly divide by R until 0;
- Fractions: repeatedly multiple by R until 1;

For example, $(0.3125)_{10} \implies (.0101)_2$

$$0.3125 \times 2 = 0.625 \implies \text{Carry } 0$$

$$0.625 \times 2 = 1.25 \implies \text{Carry } 1$$

$$0.25 \times 2 = 0.50 \implies \text{Carry } 0$$

$$0.5 \times 1 = 1 \implies \text{Carry } 1$$

Startng from the most significant bit to the least.

The general strategy for manual base conversion is to convert to decimals first.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

// Function to convert a character to its corresponding integer value
```

```
int charToValue(char c) {
    if (isdigit(c)) return c - '0';              // '0'-'9' -> 0-9
    else if (isalpha(c)) return toupper(c) - 'A' + 10; // 'A'-'Z' -> 10-35
    return -1; // Error case: character is not valid
}

// Function to convert an integer value to its corresponding character
char valueToChar(int value) {
    if (value >= 0 && value <= 9) return value + '0'; // 0-9 -> '0'-'9'
    else if (value >= 10 && value <= 35) return value - 10 + 'A'; // 10-35 ->
        'A'-'Z'
    return '?'; // Error case
}

int toDecimal(const char *input, int baseFrom) { // ensure original input not
    modified
    int res = 0;
    int power = 1;    // initialised to baseFrom^0
    for (int i = strlen(input) - 1; i >= 0; i--) {
        int value = charToValue(input[i]);      // convert char representation to
            int value
        if (value < 0 || value >= baseFrom) {
            printf("Error: Invalid digit '%c' for base %d\n", input[i], baseFrom);
            exit(1);
        }
        res += value * power;
        power *= baseFrom;
    }
    return res;
}

void fromDecimal(int input, int baseTo, char *result) {
    int i = 0;
    while (input > 0) {
        int remainder = input % baseTo;
        result[i++] = valueToChar(remainder);
        input /= baseTo;
    }
    result[i] = '\0';
    // Reverse the result string
    for (int j = 0; j < i / 2; j++) {
        char temp = result[j];
        result[j] = result[i - j - 1];
        result[i - j - 1] = temp;
    }
}
```

```
void integerConversion(const char *input, int baseFrom, int baseTo, char
    *result) {
    if (baseFrom < 2 || baseTo < 2 || baseFrom > 36 || baseTo > 36) {
        printf("Error: Base must be between 2 and 36.\n");
        exit(1);
    }
    // Convert from the source base to decimal
    int decimalValue = toDecimal(input, baseFrom);
    // Convert from decimal to the target base
    if (decimalValue == 0) {
        result[0] = '0';
        result[1] = '\0';
    } else {
        fromDecimal(decimalValue, baseTo, result);
    }
}

int main() {
    return 0;
}
```

4 Binary to Octal/Hex Conversion Shortcut

Binary to Octal

Partition in group of 3 since $2^3 = 8$

$$(10111011001.101110)_2 = (10\ 111\ 011\ 001\ .\ 101\ 110)_2 = (2731.56)_8$$

In reverse, represent each digit to 3 bit.

$$(2731.56)_8 = (10\ 111\ 011\ 110\ .\ 101\ 110)_2$$

Binary to hexadecimals

Similarly, partition in groups of 4.

$$(5D9.B8)_{16} = (101\ 1101\ 1001\ .\ 1011\ 1000)_2$$

A = 1010; C = 1100; F = 1111;

5 ASCII Code

Decimal	Hex	Octal	Symbol	Name
0	00	000	NUL	Null character
1	01	001	SOH	Start of Heading
2	02	002	STX	Start of Text
3	03	003	ETX	End of Text
4	04	004	EOT	End of Transmission
5	05	005	ENQ	Enquiry
6	06	006	ACK	Acknowledgment
7	07	007	BEL	Bell
8	08	010	BS	Backspace
9	09	011	HT	Horizontal Tab
10	0A	012	LF	Line Feed
11	0B	013	VT	Vertical Tab
12	0C	014	FF	Form Feed
13	0D	015	CR	Carriage Return
14	0E	016	SO	Shift Out
15	0F	017	SI	Shift In
16	10	020	DLE	Data Link Escape
17	11	021	DC1	Device Control 1
18	12	022	DC2	Device Control 2
19	13	023	DC3	Device Control 3
20	14	024	DC4	Device Control 4
21	15	025	NAK	Negative Acknowledgment
22	16	026	SYN	Synchronous Idle
23	17	027	ETB	End of Trans. Block
24	18	030	CAN	Cancel
25	19	031	EM	End of Medium
26	1A	032	SUB	Substitute
27	1B	033	ESC	Escape
28	1C	034	FS	File Separator
29	1D	035	GS	Group Separator
30	1E	036	RS	Record Separator
31	1F	037	US	Unit Separator
32	20	040	(space)	Space
33	21	041	!	Exclamation mark
34	22	042	"	Double quote
35	23	043	#	Number sign
36	24	044	$	Dollar sign
37	25	045	%	Percent sign
38	26	046	&	Ampersand
39	27	047	’	Single quote
40	28	050	(	Left parenthesis
41	29	051	)	Right parenthesis
42	2A	052	*	Asterisk

Decimal	Hex	Octal	Symbol	Name
43	2B	053	+	Plus sign
44	2C	054	,	Comma
45	2D	055	-	Hyphen-minus
46	2E	056	.	Period
47	2F	057	/	Slash
48	30	060	0	Digit 0
49	31	061	1	Digit 1
50	32	062	2	Digit 2
51	33	063	3	Digit 3
52	34	064	4	Digit 4
53	35	065	5	Digit 5
54	36	066	6	Digit 6
55	37	067	7	Digit 7
56	38	070	8	Digit 8
57	39	071	9	Digit 9
58	3A	072	:	Colon
59	3B	073	;	Semicolon
60	3C	074	<	Less-than sign
61	3D	075	=	Equals sign
62	3E	076	>	Greater-than sign
63	3F	077	?	Question mark
64	40	100	@	At sign
65	41	101	A	Uppercase A
66	42	102	B	Uppercase B
67	43	103	C	Uppercase C
68	44	104	D	Uppercase D
69	45	105	E	Uppercase E
70	46	106	F	Uppercase F
71	47	107	G	Uppercase G
72	48	110	H	Uppercase H
73	49	111	I	Uppercase I
74	4A	112	J	Uppercase J
75	4B	113	K	Uppercase K
76	4C	114	L	Uppercase L
77	4D	115	M	Uppercase M
78	4E	116	N	Uppercase N
79	4F	117	O	Uppercase O
80	50	120	P	Uppercase P
81	51	121	Q	Uppercase Q
82	52	122	R	Uppercase R
83	53	123	S	Uppercase S
84	54	124	T	Uppercase T
85	55	125	U	Uppercase U
86	56	126	V	Uppercase V

Decimal	Hex	Octal	Symbol	Name
87	57	127	W	Uppercase W
88	58	130	X	Uppercase X
89	59	131	Y	Uppercase Y
90	5A	132	Z	Uppercase Z
91	5B	133	[	Left square bracket
92	5C	134	\	Backslash
93	5D	135	]	Right square bracket
94	5E	136	^	Caret
95	5F	137	_	Underscore
96	60	140	`	Grave accent
97	61	141	a	Lowercase a
98	62	142	b	Lowercase b
99	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Left curly brace
124	7C	174	|	Vertical bar
125	7D	175	}	Right curly brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

ASCII Table with MSB and LSB Headers

MSBs \ LSBs	000	001	010	011	100	101	110	111
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
001	BS	HT	LF	VT	FF	CR	SO	SI
010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
011	CAN	EM	SUB	ESC	FS	GS	RS	US
100	(space)	!	"	#	$	%	&	’
101	(	)	*	+	,	-	.	/
110	0	1	2	3	4	5	6	7
111	8	9	:	;	<	=	>	?
1000	@	A	B	C	D	E	F	G
1001	H	I	J	K	L	M	N	O
1010	P	Q	R	S	T	U	V	W
1011	X	Y	Z	[	\	]	^	_
1100	‘	a	b	c	d	e	f	g
1101	h	i	j	k	l	m	n	o
1110	p	q	r	s	t	u	v	w
1111	x	y	z	{	|	}	~	DEL

6 Signs

Negative Numbers

Unsigned numbers consists of only non-negative values, while signed numbers include all values. There are 3 common ways to represent signed binary number:

- Sign-and-Magnitude
- 1s Complement
- 2s Complement

6.1 Sign-and-Magnitude

0 for + and 1 for -. The first bit of an 8-bit binary number represents the sign, the remaining 7 bits for magnitude.

$$00110100 \implies +110100_2 = +52_{10}$$

$$10010011 \implies -10011_2 = -19_{10}$$

The largest value an 8-bit binary using Sign-and-Magnitude can represent is

$$01111111 = +127_{10}$$

Smallest

$$11111111 = -127_{10}$$

There will be 2 zero, 0^+ and 0^-

$$00000000 = +0_{10}$$

$$10000000 = -0_{10}$$

To negate a number, just invert the sigb bit.

$$10000101_{sm} negate \implies 00000101_{sm}$$

6.2 1s Complement

Given a decimal number x which can be expressed as an n-bit binary number, its negated valye can be obtained in 1s-complement representation using:

$$\boldsymbol{-x = 2^n - x - 1}$$

Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

$$-00001100_2 = 2^8 - 12 - 1 = 243 = 11110011_{1s}$$

Which is, to invert every bit in the binary representation.

$$(14)_{10} = (00001110)_2 = (00001110)_{1s}$$

$$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$$

The largest value an 8-bit binary using 1s-complement can represent is

$$01111111 = +127_{10}$$

Smallest

$$10000000 = -127_{10}$$

There will be 2 zero, 0^+ and 0^-

$$00000000 = +0_{10}$$

$$11111111 = -0_{10}$$

Range (for n bits):

$$\boldsymbol{[-(2^{n-1} - 1), 2^{n-1} - 1]}$$

The most significant bit still represents the sign: 0 for + and 1 for -

6.3 2s Complement

2's complement is the default integer representation scheme in C. Given a decimal number x which can be expressed as an n-bit binary number, its negated valye can be obtained in 2s-complement representation using:

$$\boldsymbol{-x = 2^n - x} \text{ (in decimal)}$$

Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 2s-complement is:

$$-00001100_2 = 2^8 - 12 = 244 = 11110100_{2s}$$

Which is, to invert every bit in the binary representation, then + 1. The largest value an 8-bit binary using 1s-complement can represent is

$$01111111 = +127_{10}$$

Smallest

$$10000000 = -128_{10}$$

Zero

$$00000000 = +0_{10}$$

Range (for n bits):

$$\boldsymbol{[-2^{n-1}, 2^{n-1} - 1]}$$

6.4 Conversion between 1's complement and 2's complement

1's complement and 2' complement etc in this section are meant to represent negative numbers, therefore if a integer is positive, the binary representation of it in its 1's complement is the same as its 2's complement. **Conversion between 1's and 2's Complement:**

One's complement and two's complement are methods of representing negative binary numbers. To convert from one's complement to two's complement, you simply add 1 to the one's complement representation. Conversely, to convert from two's complement to one's complement, you subtract 1.

- **1's Complement to 2's Complement:** Add 1 to the binary number.
- **2's Complement to 1's Complement:** Subtract 1 from the binary number.

```
int twos_to_ones(int x) {
    if (x >= 0) {
        return x;      // if positive, 1's comp is the same as 2's comp,
    }                  // no need to change.
    return ~(-x);      // if negative, invert bit which yields x - 1,
}                      // representing the subtraction of 1
                       // when converting 2's comp to 1's comp

int ones_to_twos(int x) {
    if (x >= 0) {
        return x;
    }
    return ~(x - 1);      // if negative, invert bits and add 1
}
```

For example, to convert a negative number in one's complement to two's complement, you add 1 to the least significant bit (LSB). To convert a negative number in two's complement to one's complement, you subtract 1 and invert all the bits.

4-bit system Comparision

Value	Sign-and-Magnitude	1's Comp.	2's Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Table 3: Comparison of Binary Representations

Value	Sign-and-Magnitude	1's Comp.	2's Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

Table 4: Binary Representations of Negative Values in Different Formats

Excess-n representation

Translate all bit representation by n to evenly distribute positive and negative numbers. Example: Excess-4 representaton on 3-bit numbers:

Ex-4 Representation	Signed values
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

Table 5: Excess-4 representaton on 3-bit numbers

As the table shown, original binary representation from 0 to 7 is mapped to signed integer representations from -4 ro 3 by a translation of 4. Thus called Excess-4 representation.

7 Addtion and subtraction

2s Complement on Additon and subtraction

Algorithm for Addition of integers A and B:

1. Perform binary addition on the two numbers;
2. Ignore carry out of the most significant bit;
3. Check for overflow, when the carry in and carry out of the most significant bit are different. This may only happen when A and B are of the same sign.

A - B = A + (-B)

Overflow

Due to the fixed range of signed numbers, overflow may happen when addition or subtraction result goes beyond the range. This can be easily detected when the sign of the result is different from both A and B.

Example: 4-bit 2s-complement system has range of $-8_{10} to 7_{10}$

$$0101_{2s} + 0110_{2s} = 1011_{2s}$$

$$5_{10} + 6_{10} = -5_{10}$$

By comparing the sign bit it's easy to tell.

1s Complement on Addition/Subtraction

Algorithm:

1. Perform binary additon on A and B;
2. If there is a carry out of the MSB, add 1 to the result.
3. Check for overflow.

8 Real Number Representation

Computer hardwares only have finite number of bit, therefore precision in represention fractions has a certain limit. There are 2 common ways to represent real numbers, fixed-point representation and IEEE 754 Floating-Point Representation.

Fixed-point representation

In fixed-point representation, fixed numbers of bits are allocated for the integer and fraction parts. This leads to a fixed level of precision.

$$011010.11_{2s} = 26.75_{10}$$

$$111110.11_{2s} = -000001.01_{2} = -1.25_{10}$$

If only 2 bits are allocated for fraction, the smallest unit this scheme can represent will be 2^{-2} or 0.25.

IEEE 754 Floating-Point Representation

This scheme uses scientific number expression that allows for the representation of very large or very small number, with variable precision. Such number representation consists of 3 components,

1. 1 bit representing sign

2. 8 bits representing exponent in excess-127 (11 bits in 64 bits, or double precision, in excess-1023)
3. 23 bits mantissa (52 bits in 64 bits, or double precision)

Algorithm:

1. Identify the sign, 0 for positive, 1 for negative on the sign bit.
2. Normalise the binary representation with an implicit leading bit 1.
3. Store the fraction part of normalised representation in mantissa, starting from the most significant bit and fill the remaining of the 23 bits with 0.
4. Store the exponent on 2 from the normalised expression, in excess-127.

Examples:

$$-6.5_{10} = -110.1_2 = -1.101_2 \times 2^2$$

$$\text{Sign bit} = 1$$

$$\text{Exponent} = 2 + 127 = 129_{10} = 10000001_2$$

$$\text{Mantissa} = 10100000000000000000000$$

$$\implies -6.5_{10} = 11000000110100000000000000000000_2 = C0D00000_{16}$$

$$0xC3380000_{16} = 11000011001110000000000000000000_2$$

$$\implies 11000011001110000000000000000000$$

$$\implies -1.0111_2 \times 2^{10000110_2} = -1.0111_2 \times 2^7$$

$$\implies -10111000_2 = -184$$

Rounding

The most important thing you must know about floating point arithmetic is that associativity is never preserved (even when the operation is theoretically associative), i.e.

$$(A \text{ op } B)opC \neq A \text{ op } (B \text{ op } C)$$

Rounding destroys associativity! Rounding is selecting a representable number as the result.

$$numbers \in (1.0000...000 \text{ (23 bits decimals)} , 1.000...0001 \text{ (23 bits decimals)})$$

Cannot be represented.

Examples

$00110011_{1s} - 00101010_{1s} = 00110011_{1s} + (-00101010_{1s}) = 00110011_{1s} + 11010101_{1s}$
$= 100001000$ add the carry-out 1 to the end:
$= 00001001_{1s}$