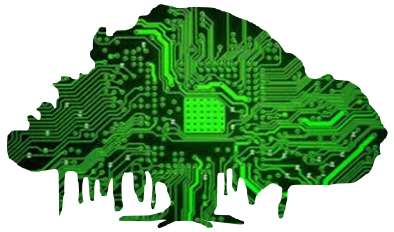


Lecture 2 – Binary numbers and representations

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

Chapter 1 (second half)

Complements of numbers

- Complement operations are run on a single number in any given base
- Complements are used in digital computers to simplify the subtraction operation and for logical manipulation
- Simplifying operations leads to simpler, less expensive circuits to implement the operations
- There are two types of complements for each base-*r* system:

1. The radix complement [r's complement] – called the 10's complement in decimal, 2's complement in binary and so on
2. The diminished radix complement [(r-1)'s complement] – called the 9's complement in decimal, 1's complement in binary and so on

Diminished radix complement

- Given a number *N* in base *r* having *n* digits, the (*r* - 1)'s complement of *N* , i.e., its diminished radix complement, is defined as $(r^n - 1) - N$
- For decimal numbers, the 9's complement of *N* is $(10^n - 1) - N$
- In this case, $10^n - 1$ is a number represented by *n* 9s
- For example, if *n* = 4, we have 10^4 = 10,000 and 10^4 - 1 = 9999
- It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9
- Examples:
  - 1242
  - 9981

Diminished radix complement

- For binary numbers, the 1's complement of *N* is $(2^n - 1) - N$.
- Again, $(2^n - 1)$ is a binary number represented by *n* 1s
- For example, if *n* = 4, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus, the 1's complement of a binary number is obtained by subtracting each digit from 1
- However, when subtracting binary digits from 1, we can have either 1 - 0 = 1 or 1 - 1 = 0, which causes the bit to change from 0 to 1 or from 1 to 0, respectively
- Therefore, **the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's.**
- Examples:
  - 11100101
  - 10000

Radix complement

- The *r*'s complement of an *n*-digit number *N* in base *r* is defined as $r^n - N$ for $N \neq 0$ and as 0 for $N = 0$
- Comparing with the (*r* - 1)'s complement, we note that the r's complement is obtained by adding 1 to the (*r* - 1)'s complement, since $r^n - N = [(r^n - 1) - N] + 1$
- Thus, the 10's complement of decimal 2389 is 7610 + 1 = 7611 and is obtained by adding 1 to the 9's complement value
- The 2's complement of binary 101100 is 010011 + 1 = 010100 and is obtained by adding 1 to the 1's-complement value
- Examples:
  - $(66772)_{10}$
  - $(10011)_2$

Some notes on Complements

- If the original number *N* contains a radix point, the point should be removed temporarily in order to form the *r*'s or (*r* - 1)'s complement
- The radix point is then restored to the complemented number in the same relative position
- Example: 9's complement and 10's complement of $(9782.314)_{10}$
- It is also worth mentioning that **the complement of the complement restores the number to its original value**
- To see this relationship, note that the *r*'s complement of *N* is $r^n - N$, so that the complement of the complement is $r^n - (r^n - N) = N$ and is equal to the original number
- *(r-1)*'s complement of *N* is $r^n - 1 - N$, so that the complement of the complement is $(r^n - 1) - (r^n - 1 - N) = N$ and is equal to the original number

Subtraction with Radix complements

- The usual method of borrowing taught in elementary school for subtraction is less efficient when subtraction is implemented with digital hardware
- Lets assume we have to perform M-N in base r
- Here is the algorithm using Radix complement:
  1. Take radix complement of N: $r^n - N$
  2. Add this to M: $r^n - N + M = r^n + (M - N) = r^n - (N - M)$
  3. If you get a carry in the (n+1)th digit, then the result is positive, discard the carry and you are done
  4. If you ***do not*** get a carry in the $(n+1)^{th}$ digit, then the result is **negative**. Take the radix complement of the number to get the answer, then put a negative sign
- 10's complement subtraction:
  - $(9812)_{10} - (3142)_{10}$
  - $(1423)_{10} - (7336)_{10}$

Subtraction with Diminished radix complements

- The usual method of borrowing taught in elementary school for subtraction is less efficient when subtraction is implemented with digital hardware
- Lets assume we have to perform M-N in base r
- Here is the algorithm using Diminished radix complement:
  1. Take diminished radix complement of N: $r^n - 1 - N$
  2. Add this to M: $r^n - 1 - N + M = r^n + (M - N - 1) = (r^n - 1) - (N - M)$
  3. If you get a carry in the $(n+1)^{th}$ digit, then the result is positive, ***add the carry to the result*** and you are done
  4. If you ***do not*** get a carry in the $(n+1)^{th}$ digit, then the result is **negative**. Take the diminished radix complement of the number to get the answer, then put a negative sign
- 9's complement subtraction:
  - $(6552)_{10} - (3145)_{10}$
  - $(2142)_{10} - (9667)_{10}$

Binary subtraction with complements

- Perform the following subtractions using 2's complement method:
- $(110001)_2 – (010100)_2$
- $(010110)_2 – (100)_2$
- $(10)_2 – (100000)_2$
- $(100001)_2 – (110100)_2$

- Perform the following subtractions using 1's complement method:
- $(110001)_2 – (010100)_2$
- $(100100)_2 – (011101)_2$
- $(1)_2 – (10100)_2$
- $(11010)_2 – (110111)_2$

Subtraction using complements

Radix Subtraction

- Find Radix Complement of Y
- Add Y complement to X

Extra Leading Digit

- Drop extra digit

No Extra Digit

- Take Radix Complement
- Attach Negative

Reduced Radix Subtraction

- Find Reduced Radix Complement of Y
- Add Y complement to X

Extra Leading Digit

- Drop extra digit
- Add extra digit to result

No Extra Digit

- Take Reduced Radix Complement
- Attach Negative

Representing negative binary

- In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign
- Because of hardware limitations, computers must represent everything with binary digits
- It is customary to represent the sign with a bit placed in the leftmost position of the number
- The convention is to make the sign bit 0 for positive and 1 for negative
- This can be done using:
    1. Signed magnitude representation
    2. Signed complement representation
        1. Signed 1's complement representation
        2. Signed 2's complement representation

Signed magnitude representation

- In this notation, the number consists of a magnitude and a symbol (+ or -) or a bit (0 or 1) indicating the sign
- This is similar to the representation of signed numbers used in ordinary arithmetic
- For example, the string of bits 01001 represents +9, and 11001 represents -9 in signed magnitude representation
- In signed-magnitude, -9 is obtained from +9 by changing only the sign bit in the leftmost position from 0 to 1
- Weird: +0 is represented as 0000 and minus 0 is represented as 1000. So two representations for zero – inefficient and may cause errors

Signed complement representation

- When arithmetic operations are implemented in a computer, it is more convenient to use a different system, referred to as the *signed complement* system, for representing negative numbers
- In this system, a negative number is indicated by its complement
- Whereas the signed-magnitude system negates a number by changing its sign, the signed-complement system negates a number by taking its complement
- Since positive numbers always start with 0 (plus) in the leftmost position (in all representations), it follows that the complement will always start with a 1, indicating a negative number
- In signed-1's-complement, -9 is obtained by complementing all the bits of +9, including the sign bit
- The signed-2's-complement representation of -9 is obtained by taking the 2's complement of the positive number, including the sign bit

Reading and Writing signed complements

- Write into memory the following numbers in signed 2's complement prestation in 4 bits:
- +3
- -7
- 0

- We read these numbers from memory knowing its in signed 2's complement prestation in 4 bits, what are the numbers?:
- $(1100)_2$
- $(1111)_2$
- $(0000)_2$
- $(1000)_2$

Interpretations for 4 bit binary numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+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
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Signed addition

- Here is some magic: if the numbers are represented in memory in 2's complement form, we just need to add the two numbers, the sign takes care of itself!
- The result is also in 2's complement representation
- The sign bit is to be included in the addition and if there is a carry, it is discarded
- Examples in 4-bit signed 2's complement representation:

1. 3+1
2. 1+(-7)
3. (-8)+5
4. 7+(-3)

- In order to obtain a correct answer, we must ensure that the result has a sufficient number of bits to accommodate the sum
- If we start with two *n*-bit numbers and the sum occupies *n* + 1 bits, we say that an overflow occurs

Signed subtraction

- Subtraction of two signed binary numbers when negative numbers are in 2's-complement form is simple and can be stated as follows:
  - Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit)
  - A carry out is discarded
- This works because: $M - N = M + (-N)$
- Examples in 4-bit signed 2's complement representation:

1. 3-5
2. 6-2
3. 1-7

Binary codes

- Any discrete element of information that is distinct among a group of quantities can be represented with a binary code (i.e., a pattern of 0's and 1's)
- The codes must be in binary because, in today's technology, only circuits that represent and manipulate patterns of 0's and 1's can be manufactured economically for use in computers
- However, it must be realized that binary codes merely change the symbols, not the meaning of the elements of information that they represent
- If we inspect the bits of a computer at random, we will find that most of the time they represent some type of coded information rather than binary numbers
- An *n*-bit binary code can represent up to 2^n distinct elements of the set that is being coded

Binary Coded Decimal (BCD)

- The code most commonly used for the decimal digits is the straight binary assignment and is called *binary-coded decimal* (BCD)
- A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2
- The 10 decimal digits form such a set
- A binary code that distinguishes among 10 elements must contain at least four bits, but 6 out of the 16 possible combinations remain unassigned

Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Binary Coded Decimal (BCD)

- A number with *k* decimal digits will require 4*k* bits in BCD
- Decimal 396 is represented in BCD with 12 bits as 0011 1001 0110, with **each group of 4 bits representing one decimal digit**
- A BCD number greater than 10 looks different from its equivalent binary number, even though both contain 1's and 0's
- Moreover, **the binary combinations 1010 through 1111 are not used and have no meaning in BCD**

Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD addition

- We can add two BCD digits as follows:

1. Add the corresponding binary
2. If the result is less than 1001, you are done
3. If the result is more, add six to it. Now, the last four digits represent the correct BCD and the carry can be carried forward

- Example:
  - $(392)_{BCD} + (479)_{BCD}$

Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Signed BCD

- The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary
- We can use either the familiar signed-magnitude system or the signed-complement system
- The sign of a decimal number is usually represented with four bits to conform to the four-bit code of BCD
- It is customary to designate a plus with 0000 and a minus with the BCD equivalent of 9, which is 1001

Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

ASCII code

- Many applications of digital computers require the handling not only of numbers, but also of other characters or symbols, such as the letters of the alphabet
- The standard binary code for the alphanumeric characters is the American Standard Code for Information Interchange (ASCII), which uses seven bits to code 128 characters
- The seven bits of the code are designated by *b*1 through *b*7, with *b*7 the most significant bit
- The letter *A*, for example, is represented in ASCII as 1000001
- The ASCII code also contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions
- The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters, such as %, *, and $

ASCII code

$b_4b_3b_2b_1$	$b_7b_6b_5$ = 000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	∧	n	~
1111	SI	US	/	?	O	–	o	DEL

Parity bit

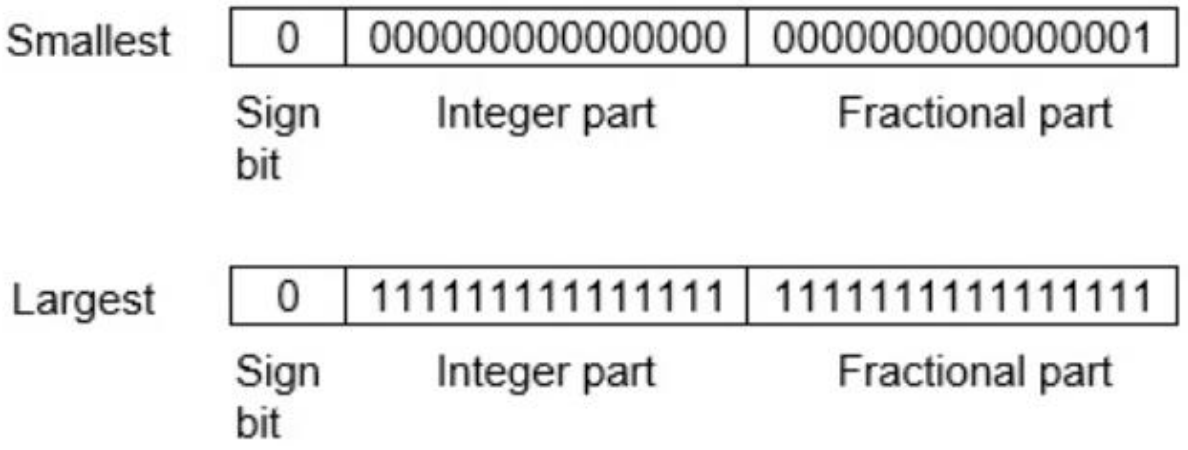
- ASCII is a seven-bit code, but most computers manipulate an eight-bit quantity as a single unit called a *byte*
- Therefore, ASCII characters most often are stored one per byte
- The extra bit is sometimes used for other purposes, depending on the application
- To detect errors in data communication and processing, an eighth bit is sometimes added to the ASCII character to indicate its parity
- A *parity bit* is an extra bit included with a message to make the total number of 1's either even or odd
- ASCII 'A': 1000001. With even parity: 'A': 01000001. If we get a communication that is 11000001, then we can discard it

Representing fractions (real numbers)

- We need to operate with fractions all the time
- This means we need a method to store/represent them in binary
- The simplest way is to have a “fixed” point representation where the binary point is assumed to be fixed at a certain location
- For example, for an 4-bit system, if given fixed-point representation is II.FF, then you can store minimum value is 00.01 (0001) and maximum value is 11.11 (1111)
- Remember the point is not actually stored – it is assumed to be there
- There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field – which means we can also have signed magnitude fixed point numbers and signed complement fixed point numbers

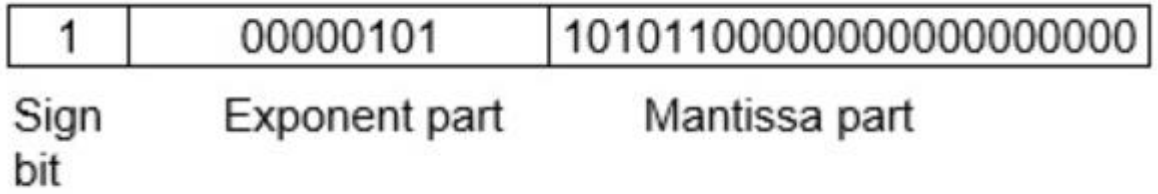
Fixed point representation

- The advantage of using a fixed-point representation is performance and ease of arithmetic
- The disadvantage is relatively limited range of values that they can represent
- So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy
- The smallest positive number is $2^{-16} \approx 0.000015$, and the largest positive number is ≈ 32768,



Floating point representation

- This representation does not reserve a specific number of bits for the integer part or the fractional part
- Instead it reserves a certain number of bits for the number (called the mantissa) and a certain number of bits to say where within that number the decimal place sits (called the exponent)
- We convert the number to be stored as $N = M * r^e$ and store M and e as binary
- Clearly, a large mantissa and small exponent can give both high precision and high range
- For instance, using 32-bit format: the 1 bit sign bit, 8 bits for signed exponent, and 23 bits for the fractional part. Smallest is 1.18×10^{-38} and the largest is 3.40×10^{38}



$-53.5=(-110101.1)_2=(-1.101011)\text{x}25$

Next week's Lab

- Introduction
- Arduino – download and install the "Arduino IDE"
  - Uses simple C based coding system
- Everyone brings a laptop
- Not gate operation
  - Convert binary 1 to 0 and 0 to 1 and make some LEDs light up
  - Binary 0 and 1 are represented by 0 V and +5 V respectively