FIT 1047

Introduction to computer systems, networks and security

MONASH University

Today will be about data

- How can we represent numbers? Integers, signed integers, 1's complement, 2's complement, floating point
- How can we represent characters? Latin alphabet, ASCII, extended ASCII, Unicode
- Error detection

Lets look at the content of computer memory:

00111100110100100101010000011101
11011000010010100101010000111001
00100101001000010111001010100100
00111000010101011010100101001001
11111111110000110100010100100100

One word might have 32 bits
(64 in most modern PCs)
00111100110100100101010000011101
11011000010010100101010000111001
00100101001000010111001010100100
00111000010101011010100101001001
11111111110000110100010100100100

What does it actually represent?
11011000010010100101010000111001

- a number
- part of a long number
- a few characters
- program code
- pointer to a different memory location
- random bits
- … something else….

Integers

With n bit in a word we can express 2^n numbers.

With 3 bits we should be able to count from 0 to $2^3-1=7$

With 5 bits we should be able to count from 0 to $2^5-1=31$

Obviously, we could just represent binary by using one bit for each digit.

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Adding binary numbers is simply this:

0+0 = 0

0+1 = 1

1+0 = 1

1+1 = 10

Adding longer numbers looks like this:

1001 (9)

0101 (5)

Adding longer numbers looks like this:

1001 (9)
0101 (5)

1
---0

Adding longer numbers looks like this:

```
1001 (9)
0101 (5)
 _1_
--10
```

Adding longer numbers looks like this:

1001 (9)
0101 (5)
 1
-110

Adding longer numbers looks like this:

```
1001 (9)
0101 (5)
 1
1110 (14)
```

Integers in computers

Why is this intuitive representation of binary numbers not used?

Some requirements:

- Efficient use of space
- Negative and positive integers
    - Split number-space, half positive/half negative
- Efficient realisation in hardware
- Easy recognition of overflow

Intuitive approach

Use the most significant bit (the first bit from the left) for the sign. Let 0 be positive and 1 be negative.

This approach is called sign and magnitude.

The table now looks like this.

Decimal	Positive
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

The sign and magnitude representation has drawbacks:

- Two zeros (one of them negative?).
- Difficult and inefficient to implement (think about the process for adding or subtracting and determining if the result is positive, negative or too big -> overflow)

Instead of separating sign and magnitude one could use the complete number.

The so-called 1's complement of a binary number is derived by flipping all bits.

Now, if we define numbers with a leading 0 as positive (as before), we can define the flipped number as negative.

This results in the following table.

1's complement

Decimal	Positive	Negative	1's comp.
0	000	-0	111
1	001	-1	110
2	010	-2	101
3	011	-3	100

Now, subtraction is just adding the 1's complement and add the carry bit.

Example: 2-1 is 2+(-1) = 010 + 110 = 1000

add carry bit results in 001

Also 1's complement has drawbacks:

- still two representations of zero
- need to calculate carry bit and overflow

Another representation is derived by flipping all bits and then adding one bit, just discarding any carry bit.

This so-called 2's complement has got some nice properties.

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100

Decimal	Positive	Negative	1's comp.	2's comp.
0	000	-0	111	n.a.
1	001	-1	110	111
2	010	-2	101	110
3	011	-3	100	101
4	n.a.	-4	n.a.	100


COMPTEUR D'EXCURSION
20"/K
00002
24"/K

Some properties of 2's complement

1. Negative 1 is always 111...1
2. The smallest negative number is 100...0
3. The largest positive number is 011...1

Try some arithmetics:

2+1 = 3

010 + 001 = 011

$3 - 1 = 3 + (-1) = 2$

011 + 111 = 010 (ignore carry bit)

What about negative results?

2-4 = 2 + (-4) = -2

010 + 100 = 110

What about overflows?

3+2 = 5

011 + 010 = 101

Would be correct in an unsigned system.

Here it is not correct, because 101 = -3

Simple rule:

If two positive numbers (leading 0) add up to a negative number -> Overflow

Lets try negative numbers

-4-3 = -7

100 + 101 = 1001

Ignoring carry bit would result in 001, which is obviously wrong.

Simple rule:

If adding two negative numbers (leading 1) results in no carry bit on adding place n-1 -> Overflow.

Example for no overflow with negative numbers:

```
  111 (-1)
  110 (-2)
  1___
 ~~1~~101 (-3)
```

Example for overflow with negative numbers:

111 (-1)

100 (-4)

no carry bit on place n-1

1011 (3?? Overflow!)

Lets look at the same example in an 8-bit system:

11111111 (-1)

11111100 (-4)

11111____

~~1~~11111011 (-5 no overflow)

Summary for signed integers

- 2's complement has advantages
- Only one representation of zero
- Addition and subtraction done with simple adders (efficient hardware realisation)
- Easy overflow detection

Floating point numbers

We can do a lot with 64 bit integers. And we can combine words in memory to create much longer integers.

So, why do we need another representation?

Example

Speed of light is roughly 300,000 km per second.

Compute distance travelled at speed of light in 0.00015 seconds.

Using integers, would require to work with long numbers.

Scientific notation

(significand multiplied by base 10 to the power of some exponent):

Speed of light:

3×10^5 km/sec

Look at a time of:

1.5×10^{-4} sec

Now, computing this is as easy as 3x1.5 and 5-4 resulting in

4.5×10^1 km

Thus, in 0.00015 seconds, light will travel 45 kilometers.

Floating point numbers are essentially scientific notation using base 2

We can use it to efficiently compute with very large and very small numbers.

Why do we need integers, then?

Well, scientific notation has a precision issue.
1/3 in base 10 is 0.3333333333....
2/3 in base 10 is 0.6666666666....
1/3 + 2/3 is 0.99999999.....

But: 1/3 + 2/3 is 1

Also, computers only have a fixed length for storing numbers, thus rounding is necessary, which introduces errors.

A binary example

IEEE 754 64bit (double precision) floating point uses 52 bits for the significand and 11 bits for the exponent plus 1 sign bit.

1/10 in binary is 0.0001100110011001100....

in double precision
1.1001100110011001100110011001100110011001100110011001101
x 2^(-4)

transferring back to decimal, we get

0.1000000000000000055511151231257827021181583404541015625

The error can be quite small (e.g. 2^{-53}), but it adds up....

Summary floating point

- For many applications precision is good enough (e.g. 3D graphics calculations)
- Other applications need 100% precision (e.g. rounding errors can make a difference for money)

Note: Higher programming languages let you work using decimal. Nevertheless, knowledge about number representations is often really essential. Rounding errors or conversion errors can have grave consequences.

Video: first Ariane V start in 1996

Representation of characters

Input and output for human interaction requires to use characters.

These also need to have a binary representation.

We need to represent "the" alphabet (upper and lower case), numbers, and other symbols used in writing.

One also needs additional values for things like line feed, blanks, start of text, end of text, end of transmission, etc.


AND THAT'S NOT EVEN THE WORST PART! THE WORST PART IS THAT—
U+202e
...NEVE T'NDID YEHT—
?LLEH EHT TAHW...
...UOY DID WOH
.ELOHSSA...

Initially, codes only represented letters needed by the English language.

The American Standard Code for Information Interchange (ASCII) became an official standard in 1967

ASCII only uses 7 bits with the 8s bit supposed to be used as parity bit. This bit is turned on or off if the sum of the other 7 bits is even or odd. A very simple way of error detection.

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

Original 7 bit ASCII cannot used to represent characters such as ä,ö,ü,ß,Ä,Ö,Ü used in German, á,é,ç used in French.

Therefore, the unused 8th bit was used to build extended character sets.

For example the German umlaut ü is DC Hex or 11011100 in the 'Latin 1 Western European' set.

Depending on the character set, extensions can represent mathematical symbols, characters of various languages or even special characters such as ©.

Even the extended ASCII character set for converting binary numbers into characters is very restricted (max 256 Symbols).
Some language have more than 12,000 Symbols!

The logical way is to use multi-byte character sets.
A 2-byte character set can represent 2 to the power of 16 characters (more than 65,536 characters)

Unicode is a 16-bit alphabet that is divided into character types and character sets (code pages). Thus, by loading the particular code page, one can decide on the fly which language to use.

Furthermore, an extension mechanisms to 21 bits would allow for as additional million characters.

Unicode codespace

- Latin, Greek, Cyrillic, etc.
- Dingbats, mathematical, etc.
- Chinese, Japanese and Korean phonetic symbols and punctuation
- plus various other types of symbols (e.g. emoticons)

All-in-all, Unicode defines 220 blocks of related characters.

Shapes Miscellaneous Symbols Dingbats Miscellaneous Mathematical Symbols-A Supplemental Arrows-A Braille Patterns Supplemental Arrows-B Miscellaneous Mathematical Symbols-B Supplemental Mathematical Operators Miscellaneous Symbols and Arrows Glagolitic Latin Extended-C Coptic Georgian Supplement Tifinagh Ethiopic Extended Cyrillic Extended-A Supplemental Punctuation CJK Radicals Supplement Kangxi Radicals Ideographic Description Characters CJK Symbols and Punctuation Hiragana Katakana Bopomofo Hangul Compatibility Jamo Kanbun Bopomofo Extended CJK Strokes Katakana Phonetic Extensions Enclosed CJK Letters and Months CJK Compatibility CJK Unified Ideographs Extension A Yijing Hexagram Symbols CJK Unified Ideographs Yi Syllables Yi Radicals Lisu Vai Cyrillic Extended-B Bamum Modifier Tone Letters Latin Extended-D Syloti Nagri Common Indic Number Forms Phags-pa Saurashtra Devanagari Extended Kayah Li Rejang Hangul Jamo Extended-A Javanese Cham Myanmar Extended-A Tai Viet Meetei Mayek Extensions Ethiopic Extended-A Meetei Mayek Hangul Syllables Hangul Jamo Extended-B High Surrogates High Private Use Surrogates Low Surrogates Private Use Area CJK Compatibility Ideographs Alphabetic Presentation Forms Arabic Presentation Forms-A Variation Selectors Vertical Forms Combining Half Marks CJK Compatibility Forms Small Form Variants Arabic Presentation Forms-B Halfwidth and Fullwidth Forms Specials Linear B Syllabary Linear B Ideograms Aegean Numbers Ancient Greek Numbers Ancient

Supplement Latin Extended Additional Greek Extended General Punctuation Superscripts and Subscripts Currency Symbols Combining Diacritical Marks for Symbols Letterlike Symbols Number Forms Arrows Mathematical Operators Miscellaneous Technical Control Pictures Optical Character Recognition Enclosed Alphanumerics Box Drawing Block Elements Geometric Shapes Miscellaneous Symbols Dingbats Miscellaneous Mathematical Symbols-A Supplemental Arrows-A Braille Patterns Supplemental Arrows-B Miscellaneous Mathematical Symbols-B Supplemental Mathematical Operators Miscellaneous Symbols and Arrows Glagolitic Latin Extended-C Coptic Georgian Supplement Tifinagh Ethiopic Extended Cyrillic Extended-A Supplemental Punctuation CJK Radicals Supplement Kangxi Radicals Ideographic Description Characters CJK Symbols and Punctuation Hiragana Katakana Bopomofo Hangul Compatibility Jamo Kanbun Bopomofo Extended CJK Strokes Katakana Phonetic Extensions Enclosed CJK Letters and Months CJK Compatibility CJK Unified Ideographs Extension A Yijing Hexagram Symbols CJK Unified Ideographs Yi Syllables Yi Radicals Lisu Vai Cyrillic Extended-B Bamum Modifier Tone Letters Latin Extended-D Syloti Nagri Common Indic Number Forms Phags-pa Saurashtra Devanagari Extended Kayah Li Rejang Hangul Jamo Extended-A Javanese Cham Myanmar Extended-A Tai Viet Meetei Mayek Extensions Ethiopic Extended-A Meetei Mayek Hangul Syllables Hangul Jamo Extended-B High Surrogates High Private Use Surrogates Low Surrogates Private Use Area CJK Compatibility Ideographs Alphabetic Presentation Forms Arabic Presentation Forms-A Variation Selectors Vertical Forms Combining Half Marks CJK Compatibility Forms Small Form Variants Arabic Presentation Forms-B Halfwidth and Fullwidth Forms Specials Linear B Syllabary Linear B Ideograms Aegean Numbers Ancient Greek Numbers Ancient Symbols Phaistos Disc Lycian Carian Old Italic Gothic Ugaritic Old Persian Deseret Shavian Osmanya Cypriot Syllabary Imperial Aramaic Phoenician Lydian Meroitic Hieroglyphs Meroitic Cursive Kharoshthi Old South Arabian Avestan Inscriptional Parthian Inscriptional Pahlavi Old Turkic Rumi Numeral Symbols Brahmi

Tibetan Myanmar Georgian Hangul Jamo Ethiopic Ethiopic Supplement Cherokee Unified Canadian Aboriginal Syllabics Ogham

Runic Tagalog Hanunoo Buhid Tagbanwa Khmer Mongolian Unified Canadian Aboriginal Syllabics Extended Limbu Tai Le New Tai

Lue Khmer Symbols Buginese Tai Tham Balinese Sundanese Batak Lepcha Ol Chiki Sundanese Supplement Vedic Extensions

Phonetic Extensions Phonetic Extensions Supplement Combining Diacritical Marks Supplement Latin Extended Additional Greek

Extended General Punctuation Superscripts and Subscripts Currency Symbols Combining Diacritical Marks for Symbols Letterlike

Symbols Number Forms Arrows Mathematical Operators Miscellaneous Technical Control Pictures Optical Character Recognition

Enclosed Alphanumerics Box Drawing Block Elements Geometric Shapes Miscellaneous Symbols Dingbats Miscellaneous

Mathematical Symbols-A Supplemental Arrows-A Braille Patterns Supplemental Arrows-B Miscellaneous Mathematical

Symbols-B Supplemental Mathematical Operators Miscellaneous Symbols and Arrows Glagolitic Latin Extended-C Coptic

Georgian Supplement Tifinagh Ethiopic Extended Cyrillic Extended-A Supplemental Punctuation CJK Radicals Supplement Kangxi

Radicals Ideographic Description Characters CJK Symbols and Punctuation Hiragana Katakana Bopomofo Hangul Compatibility

Jamo Kanbun Bopomofo Extended CJK Strokes Katakana Phonetic Extensions Enclosed CJK Letters and Months CJK Compatibility

CJK Unified Ideographs Extension A Yijing Hexagram Symbols CJK Unified Ideographs Yi Syllables Yi Radicals Lisu Vai Cyrillic

Extended-B Bamum Modifier Tone Letters Latin Extended-D Syloti Nagri Common Indic Number Forms Phags-pa Saurashtra

Devanagari Extended Kayah Li Rejang Hangul Jamo Extended-A Javanese Cham Myanmar Extended-A Tai Viet Meetei Mayek

Extensions Ethiopic Extended-A Meetei Mayek Hangul Syllables Hangul Jamo Extended-B High Surrogates High Private Use

Surrogates Low Surrogates Private Use Area CJK Compatibility Ideographs Alphabetic Presentation Forms Arabic Presentation

Forms-A Variation Selectors Vertical Forms Combining Half Marks CJK Compatibility Forms Small Form Variants Arabic

Presentation Forms-B Halfwidth and Fullwidth Forms Specials Linear B Syllabary Linear B Ideograms Aegean Numbers Ancient

Greek Numbers Ancient Symbols Phaistos Disc Lycian Carian Old Italic Gothic Ugaritic Old Persian Deseret Shavian Osmanya

Cypriot Syllabary Imperial Aramaic Phoenician Lydian Meroitic Hieroglyphs Meroitic Cursive Kharoshthi Old South Arabian

Avestan Inscriptional Parthian Inscriptional Pahlavi Old Turkic Rumi Numeral Symbols Brahmi Kaithi Sora Sompeng Chakma

Sharada Takri Cuneiform Cuneiform Numbers and Punctuation Egyptian Hieroglyphs Bamum Supplement Miao Kana Supplement

Byzantine Musical Symbols Musical Symbols Ancient Greek Musical Notation Tai Xuan Jing Symbols Counting Rod Numerals

Mathematical Alphanumeric Symbols Arabic Mathematical Alphabetic Symbols Mahjong Tiles Domino Tiles Playing Cards

Enclosed Alphanumeric Supplement Enclosed Ideographic Supplement Miscellaneous Symbols And Pictographs Emoticons


AND THAT'S NOT EVEN THE WORST PART! THE WORST PART IS THAT—
U+202e
...NEVE T'NDID YEHT—
?LLEH EHT TAHW...
...UOY DID WOH
.ELOHSSA...

What is UTF-8 and UTF-16?

Unicode is the standard for computers to display and manipulate text while UTF-8 and UTF-16 are two of the many mapping methods for Unicode.

- UTF = Unicode Transfer Format
- UTF-8 uses one byte, UTF-16 two bytes per character. Thus, UTF-8 is more efficient for ASCII text

Error detection and correction

There might be errors in the transmission of messages.

Single bits might flip, parts of a word might be missing, etc.

Error detection and correction

- Parity
- Checksum
- CRC – Cyclic Redundancy Check

Parity

What does parity mean?

Parity is just another fancy word for equality

- Needs one additional parity bit
- Decide on even or odd for the complete number
- Set parity bit to 0 or 1 so that number of 1s is even (for even parity), or odd (for odd parity)

Example for even parity:

0	1	0	1	1	1	0	

Example for even parity:

0	1	0	1	1	1	0	

Calculate parity bit to get an even number of 1s:

Example for even parity:

0	1	0	1	1	1	0	

Calculate parity bit to get an even number of 1s:

0	1	0	1	1	1	0	0

Example for even parity:

0	1	0	1	1	1	0	

Calculate parity bit to get an even number of 1s:

0	1	0	1	1	1	0	0

One bit error:

0	1	0	1	1	0	0	0

Example for even parity:

0	1	0	1	1	1	0	

Calculate parity bit to get an even number of 1s:

0	1	0	1	1	1	0	0

One bit error:

0	1	0	1	1	0	0	0

Two errors:

0	0	0	1	1	0	0	0

Checksum

- Parity was just about counting. Odd or even.
- Checksums need a bit more processing power

Lets look at a message
43 52 43 30 31 30

1. Pick a number size we want to divide by and agree on it. Lets use 16.
2. Add all numbers. Results in 229
3. Divide sum by the number agreed on. 229 / 16 = 14.3125
4. Only take the remainder. 0.3125 means a remainder of 5.
5. Send the checksum with the message and check.

Checksum example:

43	52	43	30	31	30	5

Checksum example:

43	52	43	30	31	30	5

One error:

42	52	43	30	31	30	4 □ 5

Checksum example:

43	52	43	30	31	30	5

Two errors:

42	52	43	30	30	30	3 □ 5

Checksum example:

43	52	43	30	31	30	5

Another two errors:

42	52	43	30	32	30	5 = 5

Cyclic Redundancy Check
CRC

Instead of adding up the number, concatenate into one big number:

435243303130	

Divide by a previously established number (we use 16):and take the remainder:

435243303130 / 16	10

Divide by a previously established number (we use 16) and take the remainder:

435243303130 / 16	10

- In binary, CRCs can work over several bytes.
- The standardised number for division is a polynomial in the ring of polynomials over the finite field GF(2).
- Bits of the message are coefficients of a polynomial.
- The standardised number is the generator polynomial.
- Bits of the CRC are the coefficient of the polynomial derived by dividing the message polynomial by the generator polynomial.

Many standardised CRC codes of different lengths exist.

CRC codes do not provide security!

An attacker could just manipulate the message and compute a new CRC code.

Important: CRC is not a security measure. CRCs are about errors (safety) not malicious attacks (security).

Tutorials next week

- Some exercises with numbers (understand why 2's complement works)
- Use a tool to simulate first logic circuits

(content of next week Monday's lecture)

Next lecture: Monday 8 March 3pm - 4pm

- Bring some device (Wifi and browser) for MARS

(please contact me if you have any problems with this).