Exercises from the book

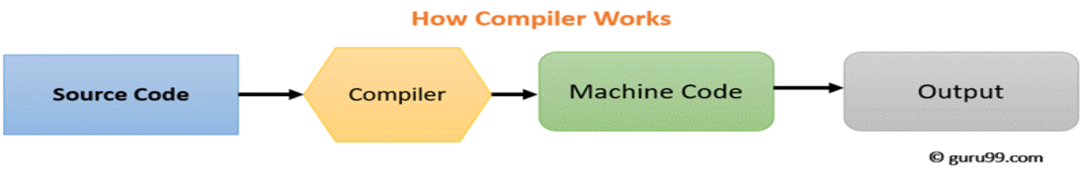
Essentials of computing systems

https://doi.org/10.21814/uminho.ed.33

Exerc. 1.1: Describe the following terms with your own words: **(a)** Compiler; **(b)** Interpreter; **(c)** Virtual machine.

a) A compiler is a computer program that takes as input the text of a program written in a high-level language like C, called the “source code” and outputs binary code, or machine code, that can be execute by the computer.

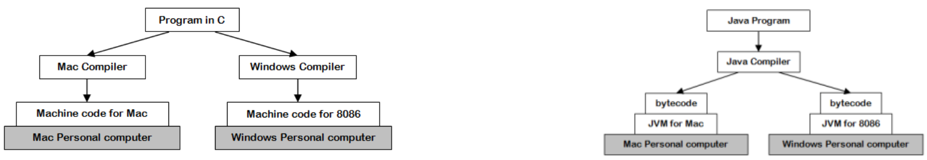
b) An interpreter executes the machine code as it reads and translates the source code (for instance Python). The compiler does the whole translation beforehand and does not execute the program. An interpreter is more flexible but is slower than a compiled program.



See https://www.guru99.com/difference-compiler-vs-interpreter.html for a more comprehensive comparison.

c) A Virtual Machine (VM) is way of being able to run the same compiled program on different architectures and be more efficient than an interpreter. Different hardware has different compilations of source code into machine code. So, to run the same program on different architectures and if we do not want to use an interpreter, we can use a VM specific for that architecture. This is the approach taken by Java and C#. A Java or C# compiler translates the source code into “bytecode”. This is binary code BUT not yet machine code because machine code is specific to the architecture. The bytecode is run by a virtual machine. Each different architecture has its own version of the VM so the SAME bytecode can be translated to the specific machine code.

This may seem like the interpreter, but it is much faster due to the nature of the source code taking much more effort to translate to machine code than the bytecode. Languages that use VMs are faster than interpreted ones and can run the same “compiled program” on different architectures.



From http://www.edu4java.com/en/concepts/compiler-interpreter-virtual-machine.html

Exerc. 1.2: Write a small program in a given programming language. Compile it and try to calculate the ratio of source code statements to the machine language instructions generated by the compilation process. Add different types of statements to the high-level program, one at a time, and check how the machine language program is affected.

```
// https://godbolt.org/
void dummy()
{
  // Uncomment the following lines, one by one,
  // waiting for the compiler to update the output.

  // int a = 2;
  // int b = 5;
  // int c = 0;
  // c = a + b;

  // change type of variable a from int to long
}
```

Using https://godbolt.org/ present several versions of a small C function (see below). The source code looks different. How about the machine translation?

```
int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) {
        fact *= i;
    }
    return fact;
}


int factorial(int n) {
    if (n <= 0) return 1;
    else return n * factorial(n-1);
}
```

Exerc. 1.3: On a **big-endian** computer, a 32-bit integer with value

00010010 00110100 01010110 01111000

is about to be stored in the memory at location 132,104. Indicate which memory cells are affected and which values are stored in each one.

132103	
132104	00010010
132105	00110100
132106	01010110
132107	01111000
132108	

Exerc. 1.4: Consider that part of the memory of a **little-endian** computer contains the values shown in the figure. Indicate the value of a 32-bit integer if it is read from the memory location 4365.

4362	0100 0011
4363	0111 0000
4364	0000 0011
4365	0001 0010
4366	1111 1111
4367	0000 0000
4368	0000 1111

00001111 00000000 111111 00010010

Exerc. 1.5: In a stored-program computer, both the instructions and the data of a program are located in the main memory while it is executed. What are the possible implications if a program accidentally modifies the value that is stored in a memory cell that is related to an instruction?

The program will not behave as expected. It can crash or produce incorrect output. Sometimes this is not an accident. Some viruses have its code encrypted. When loaded, the first instructions decode the main virus program, by modifying memory cells with the encrypted code.

Sometimes this happens because of external factors. See “Soft Error” (not to be confused with software error) in https://en.wikipedia.org/wiki/Soft_error

---

Exerc. 1.6: In a factory, the production process of a given product goes through four steps: preparation, assembly, testing, and packaging. Those steps take the following times, in seconds, to be executed: preparation (20), assembly (30), testing (35), and packaging (35). Calculate the time needed to produce 1000 replica of the product by:

(a) a single person.
120 * 1000 = 120000

(b) four persons working in a pipeline.
Time to output the first item = 120
Output time per item after the first one: 35
Total processing time = 120 + 999 x 35 = 35085

Exerc. 2.1: To encode Roman numbers (from 1 to 899), the following binary encoding for the symbols has been proposed: I (01), V (100), X (00), L (101), C (110), D (111). Indicate whether this encoding is valid and, if so, what Roman number is represented by the binary pattern **111101000101**.

Answer: Yes, it is valid because each symbol has a distinct encoding and the symbols I,V,X,L,C and D are enough to represent the numbers 1 to 899 (DCCCXCIX).

111 101 00 01 01

111	101	00	01	01
D	L	X	I	I

DLXII = 562 (https://www.calculatorsoup.com/calculators/conversions/roman-numeral-converter.php)

Exerc. 2.2: Decode the following ASCII string: 1010101 0101110 0100000 1001101 1101001 1101110 1101000 1101111.

Answer:
ASCII table: https://www.sciencebuddies.org/science-fair-projects/references/ascii-table

1010101	0101110	0100000	1001101	1101001	1101110	1101000	1101111
U	.		M	i	n	h	o

U. Minho

Exerc. 2.3: A digital image has 128x128 pixels. Each pixel in the image stores information related to three channels (Red, Blue, Green). If each channel is capable of distinguishing 256 different tones, indicate the size in bytes of the image.

Answer:
Number of pixels: 128 x 128 = 16384
Each pixel needs 3 bytes (needs to represent 3 numbers ranging from 0 to 255 and to represent a number between 0 and 255 we need 8 bits = 1 byte)
Size of picture = 16384 x 3 = 49152 Bytes

Exerc. 2.4: An image occupies 192 kibibytes and has dimensions of 256x512 pixels. Each pixel is represented by three unsigned integer values, which indicate the intensity of each channel (Red-Green-Blue) in that pixel. Indicate, in binary and decimal, the maximum value that can be assigned to each of these integers, if they all have the same size.

Answer:
1 KiB (kibibyte) = 1024 bytes (book page 17)
Size of picture file: 192 x 1024 = 196608 bytes
Number of pixels in image: 256x512 = 131072
Number of bytes available for each pixel: 196608 / 131072 = 1,5 bytes
One byte = 8 bits. 1,5 bytes = 8 + 4 = 12 bits
So, for each value of the RGB, we have 12 / 3 = 4 bits available.
With 4 bits we can go from 0000 to 1111 (binary) ⇔ 0 to 15 (decimal)

Exerc. 2.5: The CYMK* subtractive colour system is formed by Cyan, Magenta, Yellow and Black and works due to the absorption of light, as the colours that are seen come from the part of the light that is not absorbed. Each pixel is represented by four 6-bit patterns that indicate the intensity in each channel. Indicate how many different colours a pixel can have, assuming that the "00000-" ("000000" and "000001") patterns cannot be used.

Answer: with 6 bits we have 2^6 = 64 possible values for each of the components but we cannot use 2 of them ("000000" and "000001") so we have 62 for each component. The color is obtained from the combination of the four values, so we have 62x62x62x62 = 62^4 = 14776336 possible colors.

Exerc. 2.6: The SCB system for evaluating football players consists of three parameters: Strength, Courage and Braveness. Each parameter is represented by a binary pattern (of 7 bits each) that indicates the respective intensity. Indicate the number of different valid assessments with this system, if the "1111111" and "1111110" patterns represent evaluations that are still unknown or invalid, respectively.

Answer: For each of the three parameters (with 7 bits) we have 2^7 possibilities minus two ("1111111" and "1111110"). So, for each possibility we have 2^7-2 = 128-2 = 126 possible assessments. Since we use three dimensions (Strength, Courage and Braveness) the total number of possible assessments will be 126x126x126 = 126^3 = 2000376.

Exerc. 2.7: Calculate the size in kB of a sound file, if the recording lasts exactly 2 minutes and it is sampled using a sampling rate of 50 kHz and a sample resolution of 8 bits. What is the size in KiB?

Answer:
2 minutes = 120 seconds
50 kHz = 50 cycles/sec * 1000 = 50 000
Number of samples = 120 x 50000 = 6 000 000 samples
Each sample = 8 bits = 1 byte
Size of file = 6 000 000 x 1 byte = 6 000 000 bytes
6 000 000 bytes = 6 000 000 / 1000 Kb = 6000 Kb
6 000 000 bytes = 6 000 000 / 1024 KiB ≈ 5860 KiB (5859,375)

Exerc. 2.8: Calculate the sample resolution of a sound file with 4.5kB, if the recording lasts one minute with a sampling rate of 50 Hz.

Answer:
50 Hz = 50 cycles/sec
Number of samples taken in 1 minute (60 seconds) = 60 x 50 = 3000
Size of one sample = size of file 4500 bytes / 3000 samples = 1,5 bytes = 8+4 bits = 12 bits

Exerc. 2.9: Consider that a typical daily newspaper page contains 3700 Unicode characters including white spaces.

(a) How many bytes are needed to encode a 32-page edition of that daily newspaper, if it includes on average 50 photos (1.2MiB each one)?

(b) How many mebibytes are needed to store all the numbers of the newspaper published in a year?

(c) If a library contains 1250 different daily newspapers, which have on average 50 years of publication, how many pebibytes are stored there?

Answer:
UTF-8 is based on 8-bit code units. Each character is encoded as 1 to 4 bytes. Book page 19: “The majority of the common-use characters fit into the first 64k code patterns, which just require two bytes (16 bits)”. We will assume that on average 1 Unicode character equals 2 bytes. So, One page = 3700 characters = 3700 x 2 = 7400 bytes.

a) Encode a 32 page edition:
32 x 7400 + 50 x 1,2 x 1024^2 = 236800 + 62914560 = 63 151 360 bytes

b) From a) one year will be (365 x 63151360) / 1024^2 mebibytes = 23050246400 / 1048576 = 21982 MiB

c) One paper per year = 21982 MiB
50 years x 1250 titles x 21982 MiB = 1373875000 Mib x 1024^2 = 1440612352000000 bytes
1440612352000000 / 1024^5 = 1,28 PiB

Exerc. 3.1: As the figure shows, bank cheques in Portugal have 10 boxes to indicate the amount to be paid. What are the minimum and the maximum values that can be written in a bank cheque?

Answer:

Minimum: 0,01
Maximum: 99999999,99

Exerc. 3.2: Represent the following decimal numbers as binary numbers:
(a) 131; (b) 511; (c) 888; (d) 4096.

Answer:

a) 131	b) 511	c) 888	d) 4096
$131_{10} = 10000011_2$	111111111	1101111000	1000000000000

a) Divide by the base 2 to get the digits from the remainders

Division by 2	Quotient	Remainder (Digit)	Bit #
(131)/2	65	1	0
(65)/2	32	1	1
(32)/2	16	0	2
(16)/2	8	0	3
(8)/2	4	0	4
(4)/2	2	0	5
(2)/2	1	0	6
(1)/2	0	1	7

$= (10000011)_2$

b)

Division by 2	Quotient	Remainder (Digit)	Bit #
(511)/2	255	1	0
(255)/2	127	1	1
(127)/2	63	1	2
(63)/2	31	1	3
(31)/2	15	1	4
(15)/2	7	1	5
(7)/2	3	1	6
(3)/2	1	1	7
(1)/2	0	1	8

$= (111111111)_2$

c)

Division by 2	Quotient	Remainder (Digit)	Bit #
(888)/2	444	0	0
(444)/2	222	0	1
(222)/2	111	0	2
(111)/2	55	1	3
(55)/2	27	1	4
(27)/2	13	1	5
(13)/2	6	1	6
(6)/2	3	0	7
(3)/2	1	1	8
(1)/2	0	1	9

$= (1101111000)_2$

d)

Division by 2	Quotient	Remainder (Digit)	Bit #
(4096)/2	2048	0	0
(2048)/2	1024	0	1
(1024)/2	512	0	2
(512)/2	256	0	3
(256)/2	128	0	4
(128)/2	64	0	5
(64)/2	32	0	6
(32)/2	16	0	7
(16)/2	8	0	8
(8)/2	4	0	9
(4)/2	2	0	10
(2)/2	1	0	11
(1)/2	0	1	12

$= (1000000000000)_2$

Exerc. 3.3: What is the largest natural number that can be represented with
(a) 5, (b) 10, (c) 18, and (d) 32 bits?

Answer:
a) $(11111)_2 = (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = (31)_{10}$

b) $1111111111_2 =$
$1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1023_{10}$

c) $111111111111111111_2 = 1 \times 2^{17} + 1 \times 2^{16} + 1 \times 2^{15} + 1 \times 2^{14} + 1 \times 2^{13} + 1 \times 2^{12} + 1 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 262143_{10}$

d) $11111111111111111111111111111111_2 = 1 \times 2^{31} + 1 \times 2^{30} + 1 \times 2^{29} + 1 \times 2^{28} + 1 \times 2^{27} + 1 \times 2^{26} + 1 \times 2^{25} + 1 \times 2^{24} + 1 \times 2^{23} + 1 \times 2^{22} + 1 \times 2^{21} + 1 \times 2^{20} + 1 \times 2^{19} + 1 \times 2^{18} + 1 \times 2^{17} + 1 \times 2^{16} + 1 \times 2^{15} + 1 \times 2^{14} + 1 \times 2^{13} + 1 \times 2^{12} + 1 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 4294967295_{10}$

Exerc. 3.4: List all the digits and their binary representation in base 13.

Answer:

0	0000_2
1	0001_2
2	0010_2
3	0011_2
4	0100_2
5	0101_2
6	0110_2

7	0111_2
8	1000_2
9	1001_2
A	1010_2
B	1011_2
C	1100_2

Exerc. 3.5: Convert the following binary numbers to hexadecimal:
(a) 101111101101; (b) 1001110110;
(c) 11111111111; (d) 10100011110.

Answer:
HINT: Convert every 4 binary digits (from bit0) to hex digit.

(a) 101111101101 = 1011 1110 1101 = B E D = BED

(b) 1001110110 = 10 0111 0110 = 2 7 6 = 276

(c) 11111111111 = 111 1111 1111 = 7 F F = 7FF

(d) 10100011110 = 101 0001 1110 = 5 1 E = 51E

Exerc. 3.6: Convert the following hexadecimal numbers to binary:
(a) BEEF; **(b)** 1000.FF; **(c)** ABC.DEF; **(d)** DAC.34.

Answer:

(a) BEEF = B E E F = 1011 1110 1110 1111 = 1011111011101111

(b) 1000.FF = 1 0 0 0 . F F = 0001 0000 0000 0000 . 1111 1111 = 1000000000000.11111111_2

(c) ABC.DEF = 1010 1011 1100 . 1101 1110 1111 = $101010111100.110111101111_2$

(d) DAC.34 = 1101 1010 1100 . 0011 0100 = 110110101100. 00110100_2

Exerc. 3.7: Convert the following decimal numbers to base 5:
(a) 77 **(b)** 131 **(c)** 511 **(d)** 1000.

Answer:

(a) 302_5

Division	Quotient	Remainder (Digit)	Digit #
77/5	15	2	0
15/5	3	0	1
3/5	0	3	2

= $(302)_5$

(b) 1011_5

Division	Quotient	Remainder (Digit)	Digit #
131/5	26	1	0
26/5	5	1	1
5/5	1	0	2
1/5	0	1	3

= $(1011)_5$

(c) 4021_5

511/5	102	1	0
102/5	20	2	1
20/5	4	0	2
4/5	0	4	3

= $(4021)_5$

(d) 13000_5

1000/5	200	0
200/5	40	0
40/5	8	0
8/5	1	3
1/5	0	1

= $(13000)_5$

Exerc. 3.8: Convert the following base-9 numbers to decimal: **(a)** 66; **(b)** 123; **(c)** 317; **(d)** 800.

Answer:

(a) $66_9 = 6 \times 9^1 + 6 \times 9^0 = 60_{10}$

(b) $123_9 = 1 \times 9^2 + 2 \times 9^1 + 3 \times 9^0 = 102_{10}$

(c) $317_9 = 3 \times 9^2 + 1 \times 9^1 + 7 \times 9^0 = 259_{10}$

(d) $800_9 = 8 \times 9^2 + 0 \times 9^1 + 0 \times 9^0 = 648_{10}$

Exerc. 3.9: Convert the following numbers from the given base to the indicated bases:
(a) 66_{10} to bases 2, 7 and 9
(b) $13F.4_{16}$ to bases 10 and 12
(c) 1110010.1_2 to bases 3, 4 and 7
(d) $AB7_{13}$ to bases 2, 6, and 8.

Answer:

a) 66_{10}

1000010_2

66/2	33	0
33/2	16	1
16/2	8	0
8/2	4	0
4/2	2	0
2/2	1	0
1/2	0	1

= $(1000010)_2$

123_7

Division	Quotient	Remainder (Digit)
66/7	9	3
9/7	1	2
1/7	0	1

= $(123)_7$

73_9

66/9	7	3
7/9	0	7

= $(73)_9$

b) $13F.4_{16}$

$13F.4_{16} = 319.25_{10}$

$13F.4_{16} = 1 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 + 4 \times 16^{-1} = 319.25_{10}$

$13F.4_{16} = 227.3_{12}$

Base 16 to decimal calculation:
$13F.4_{16} = 1 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 + 4 \times 16^{-1} = 319.25_{10}$

Decimal to base 12 calculation:
Multiply the number with the destination base raised to the power of decimals of the result up to 6 digits resolution: **$floor(319.25 \times 12^1) = 3831$**

Divide by the base to get the digits from the remainders:

Division	Quotient	Remainder (Digit)	Digit #
3831/12	319	3	0
319/12	26	7	1
26/12	2	2	2
2/12	0	2	3

$= (227.3)_{12}$

(c) 1110010.1_2 to bases 3, 4 and 7

11020.111111_3

Base 2 to decimal calculation:
$1110010.1_2 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 114.5_{10}$

Decimal to base 3 calculation:
Multiply the number with the destination base raised to the power of decimals of the result up to 6 digits resolution:
$floor(114.5 \times 3^6) = 83470$

Divide by the base to get the digits from the remainders:

83470/3	27823	1
27823/3	9274	1
9274/3	3091	1
3091/3	1030	1
1030/3	343	1
343/3	114	1
114/3	38	0
38/3	12	2
12/3	4	0
4/3	1	1
1/3	0	1

1302.2_4

Base 2 to decimal calculation:
$1110010.1_2 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 114.5_{10}$

Decimal to base 4 calculation:
Multiply the number with the destination base raised to the power of decimals of the result up to 6 digits resolution:
$floor(114.5 \times 4^1) = 458$

Divide by the base to get the digits from the remainders:

458/4	114	2
114/4	28	2
28/4	7	0
7/4	1	3
1/4	0	1

$= (1302.2)_4$

222.333333_7

Base 2 to decimal calculation:
$1110010.1_2 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 114.5_{10}$

Decimal to base 7 calculation:
Multiply the number with the destination base raised to the power of decimals of the result up to 6 digits resolution:
$floor114.5 \times 7^6 = 13470810$

Divide by the base to get the digits from the remainders:

13470810/7	1924401	3
1924401/7	274914	3
274914/7	39273	3
39273/7	5610	3
5610/7	801	3
801/7	114	3
114/7	16	2
16/7	2	2
2/7	0	2

(d) AB7_{13} to bases 2, 6, and 8.

11100110000_2	12304_6	3460_8
Base 13 to decimal calculation: $AB7_{13} = 10 \times 13^2 + 11 \times 13^1 + 7 \times 13^0 = 1840_{10}$ Decimal to base 2 calculation: Divide by the base to get the digits from the remainders:	Base 13 to decimal calculation: $AB7_{13} = 10 \times 13^2 + 11 \times 13^1 + 7 \times 13^0 = 1840_{10}$ Decimal to base 6 calculation: Divide by the base to get the digits from the remainders:	Base 13 to decimal calculation: $AB7_{13} = 10 \times 13^2 + 11 \times 13^1 + 7 \times 13^0 = 1840_{10}$ Decimal to base 8 calculation: Divide by the base to get the digits from the remainders:

1840/2	920	0
920/2	460	0
460/2	230	0
230/2	115	0
115/2	57	1
57/2	28	1
28/2	14	0
14/2	7	0
7/2	3	1
3/2	1	1
1/2	0	1

1840/6	306	4
306/6	51	0
51/6	8	3
8/6	1	2
1/6	0	1

1840/8	230	0
230/8	28	6
28/8	3	4
3/8	0	3

Exerc. 3.10: A given computer is equipped with 1,073,741,824 bytes of memory. Why was this odd number chosen?

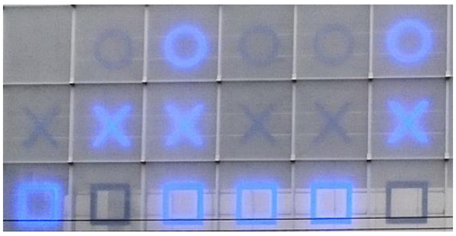
Answer:

Let's see if 1073741824 is a power of 2. If $\log_2(n)$ is integer than n is a power of 2, else not. $\log_2(1073741824) = 30$ so $1073741824 = 2^{30}$

This number was chosen probably because this computer uses 30 bits to address 2^{30} memory cells. With 30 bits we can address each memory byte individually.

Exerc. 3.11: The St. Galen train station, in Switzerland, is equipped with a binary electronic clock to indicate the time of day (hours, minutes, seconds) on three lines. At what time the photograph was taken?

Answer:



Hours	01001	9
Minutes	011001	25
Seconds	101110	46

Exerc. 3.12: A 32-bit signed integer on a little-endian computer contains the numerical value of 3. If it is transmitted to a big-endian computer byte by byte and stored there, with byte 0 in byte 0, byte 1 in byte 1, and so on, what is its numerical value on the big-endian machine if read as a 32-bit unsigned integer?

Answer:

Little Endian Byte Order: **The least significant byte** (the "little end") of the data **is placed at the byte with the lowest address**. The rest of the data is placed in order in the next bytes in memory.

Big Endian Byte Order: **The most significant byte** (the "big end") of the **data is placed at the byte with the lowest address**. The rest of the data is placed in order in the next bytes in memory.

Number 3 as a 32 bit signed integer on the the little endian computer will be interpreted like:
$3_{10} = 11_2 = 00000000\ 00000000\ 00000000\ 00000011_2$

How this number is stored CORRECTLY on both architectures:

Little Endian

Memory position X	00000011
X +1	00000000
X + 2	00000000
X + 3	00000000

Big Endian

Memory position X	00000000
X +1	00000000
X + 2	00000000
X + 3	00000011

Transmitting from little endian to big endian and storing the bytes using the same order is the same as reversing the interpretation of the bits. Big endian stored number will be interpreted like this:

Little Endian

Memory position X	00000011
X +1	00000000
X + 2	00000000
X + 3	00000000

Big Endian without changing order

Memory position X	00000011
X +1	00000000
X + 2	00000000
X + 3	00000000

Or, writing from **most significant byte to least**, according to memory position, we get:
$00000011\ 00000000\ 00000000\ 00000000_2 = 50331648_{10} = 300\ 0000_{16}$
Very different from 3.

Exerc. 3.13: As of 2018, Iceland had about 23,000 registered footballers (male and female). Calculate the minimum number of bits that allows representing this value with an integer encoded in signal and amplitude.

Answer:

1 bit for signal
How many bits for amplitude? We have to represent 23000 different athlets.

Log2(23000) bits for amplitude = 14.489 Since it must be an integer number, we need 15 bits for amplitude. To represent with signal and amplitude we need 1 + 15 = 16 bits

Exerc. 3.14: Monaco had, in 2013, 37,831 inhabitants. Calculate the minimum number of bits that are required to encode this value as a signed integer. What is the answer if the value is encoded as an unsigned integer?

Answer:

To represent 37831 in binary we need Log2(37831) = 15.2 => 16 bits
As a signed integer: 1 for sign + 16 for amplitude = 17
Unsigned: 0 for sign + 16 for amplitude = 16

Exerc. 3.15: A given company has 19 employees, who are paid every two weeks. It is necessary to register the number of half hours that each employee worked in each workday (Monday to Friday). For health reasons, the law does not permit an employee to work more than 12h in a day. Indicate the minimum number of bits needed to represent this information for two weeks.

Answer:

Assuming we just need to record the work done be each employee every day. It is implicit that we already have a data structure for each employee.

Maximum work time per day = 12h
Each employee can work from 0 to 24 half hours per day.
To record a number from 0 to 24 we need Log2(24) = 4.5 => 5 bits
Two work weeks = 10 days. Total bits = 19 x 10 x 5 = **950 bits**

Assuming we record the work done be each employee every day and need to record the employee id

To identify each employee, we need to record a number from 0 to 18 Log2(19) = 4.2 => 5 bits

Maximum work time per day = 12h
Each employee can work from 0 to 24 half hours per day.
To record a number from 0 to 24 we need Log2(24) = 4.5 => 5 bits

To record this information for 19 employees for 10 days (2 weeks) we need:
19 x 10 x 10 = **1900 bits**

Assuming we just want to record the total number of half hours done in 2 weeks per employee

To identify each employee, we need to record a number from 0 to 18 Log2(19) = 4.2 => 5 bits

Maximum number of half hours worked in 2 weeks per employee: 24 x 10 = 240
Log2(240) = 7.9 => 8 bits

To process the two-week salary, we need:
19 x (5 + 8) = **247 bits**

Exerc. 3.16: An European institute aims to assign a code to each of its members. To this end, it was decided to use the format AA / HHHHH-BB, with A being a capital letter, H being a hexadecimal digit and B being a base 2 digit. The two letters indicate which of the 51 affiliated countries the member belongs to (e.g., BE for Belgium, PO for Portugal, LX for Luxembourg). The binary digits are used to encode the type of membership of the member with the Association (00: junior member, 01: regular member, 10: senior member). Indicate, for a given country, the maximum number of members that this code allows to register.

Answer:

For each country we can have 0 up to $FFFFF_{16}$ members in each one of the 3 classes (junior, regular and senior). In total we can address/identify 3 x FFFFF members. Log2(FFFFF) = 20.
Maximum number of registered members for on country: 3×2^{20}

Exerc. 3.17: In 2014, the Spy's Gangnam Style video reached 2,147,483,648 views on YouTube. However, the number presented was negative. Explain why this happened and suggest the simplest solution to overcome it. To solve this issue, YouTube made an internal change that now allows the counters to go up to 9,223,372,036,854,775,807.

Answer:

$2147483648_{10} = 10000000\ 00000000\ 00000000\ 00000000_2$
This is a 32 bit number. If youtube showed a negative value we know that the representation was a 32 bit signed integer. The new maximum (9223372036854775807) tells us that they changed to a 64 bit signed integer because Log2(9223372036854775807) = 63.
If they used an unsigned 64 bit integer the maximum would be:
2^{64} = 18 446 744 073 709 551 616

Exerc. 3.18: Find the value of X, so that: **(a)** $23_x = 10101_2$; **(b)** $4X_7 = 35_9$.

Answer:

(a) $23_x = 10101_2 = 21_{10}$
From $23_x = 21_{10}$, we know that the last digit in base X is 3. To find X we divide 21 by X and the first reminder must be 3 (and the second 2). Let's try X = 9:

Division	Quotient	Remainder (Digit)	Digit #
21/9	2	3	0
2/9	0	2	1

$= (23)_9$

$2X^1+3X^0 = 21 \Leftrightarrow 2X = 18 \Leftrightarrow X = 9$

(b) $4X_7 = 35_9 = 32_{10}$ Since the base is 7 we just need to convert 32_{10} to base 7: 44_7
X = 4 Or $4x7^1+Xx7^0 = 32 \Leftrightarrow x = 32- 28 = 4$

Exerc. 3.19: Add the following natural numbers:
(a) $110011_2 + 10101_2$; **(b)** $129B_{12} + 239_{12}$; **(c)** $CBA_{16}+987_{16}$.

Answer:
Natural numbers = 1, 2, 3, 4, 5, … so we know they are all positive.

(a) $110011_2 + 10101_2 = 1001000_2 = 72_{10}$

(b) $129B_{12} + 239_{12} = (2135 + 333)_{10} = 2468_{10} = 1518_{12}$

(c) $CBA_{16} + 987_{16} = (3258 + 2439)_{10} = 5697_{10} = 1641_{16}$

Exerc. 3.20: Indicate the ten's-complement of the following decimal numbers:
(a) 1236; **(b)** 90037; **(c)** 111122.

Answer:

The "Ten's-complement" is the number we add to make 10.
For a 4 digit number is the number we add to have 10000

(a) 1236 Ten's-complement = 10000 – 1236 = 8764

(b) 90037 Ten's-complement = 100000 – 90037 = 9963

(c) 111122 Ten's-complement = 1000000 – 111122 = 888878

Exerc. 3.21: Indicate the two's-complement of the following binary numbers:
(a) 0011100_2; **(b)** 1100110011_2; **(c)** 00000001_2; **(d)** 11100000001_2.

Answer:

How to calculate two's-complement for a binary number?

1. Find the one's complement by inverting 0s & 1s of a given binary number.
2. Add 1 to the one's complement to get the two's complement.

	Binary number	one's-complement	two's-complement
(a)	0011100	1100011	1100100
(b)	1100110011	0011001100	0011001101
(c)	00000001	11111110	11111111
(d)	11100000001	00011111110	00011111111

Exerc. 3.22: Write the 8-bit sign-magnitude, one's-complement, two's-complement representations for decimal numbers: **(a)** +18; **(b)** +121; **(c)** -33; **(d)** -100.

For the **8-bit sign-magnitude** we first convert the number to binary and then set the first bit to 0 if the number is positive or 1 if negative.

One's-complement: A negative number needs to be converted to its complement (by flipping 0s to 1s and vice versa), which should have a '1' in the MSB. **Positive numbers, which have a '0' in the MSB, are used as is, i.e., they are not converted to their complements**.

Two's complement is just one's complement incremented by 1. To find the two's complement of a binary number, one just needs to flip bits and add 1. Again, positive numbers, which have a '0' in the MSB, are used as is, i.e., they are not converted to their complements.

	8-bit sign-magnitude	one's-complement	two's-complement
(a) +18	00010010	00010010	00010010
(b) +121	01111001	01111001	01111001
(c) -33	10100001	-(00100001) = 11011110	11011110 + 1 = 11011111
(d) -100	11100100	-(01100100) = 10011011	10011011 + 1 = 10011100

Exerc. 3.23: Calculate the value of the 10-bit binary number 10110 00111_2 in the following representations:
(a) sign-magnitude; **(b)** one's-complement; **(c)** two's-complement. **(d)** excess-511.

Answer:

(a) sign-magnitude – the first bit tells us the number is negative. The remaining bits are the magnitude. 011000111 = 199 so, **Sign-magnitude 10110 00111_2 = -199**

(b) one's-complement. If this number is in one's-complement form, we know it is negative (from the most significant bit.)
Revert from one-complement 1011000111 -> 0100111000 -> 312 but we know it is negative so **10110 00111_2 = -312**

(c) two's-complement. To revert from two's complement we subtract 1 and then flip the bits:
1011000111 -> 1011000110 -> 0100111001_2 = 313_{10} but we know it is a negative number.
So, 1011000111_2 = -313_{10}

(d) excess-511. In an excess-b representation, an n-bit pattern, whose unsigned integer value is V (0 ≤ V < 2n) represents the signed integer V-b, where b is the bias (or offset) of the numeral system. The representable numeric values range from -b to 2^n-1-b. In this case b = 511.

Our number 1011000111_2 = 711_{10} We know that this bit pattern represents V which in turn represents the number V-b that we will call X:

V-b = X ⇔ 711-511 = X ⇔ X = 200. In excess-511, the number 10110 00111_2 = **200**$_{10}$

Exerc. 3.24: Represent the number -233 in the following 10-bit representations:
(a) sign-magnitude; **(b)** one's-complement; **(c)** two's-complement; **(d)** excess-511.

Answer:
(a) sign-magnitude -233_{10} = 10**11101001**$_2$
(1110 1001_2 = 233_{10})

(b) one's-complement with 10 bits:
-233_{10} = -0011101001_2 -> **1100010110_2**

(c) two's-complement
-233_{10} -> one'e complement 1100010110_2 -> two's complement (add 1) **1100010111_2**

(d) Represent **-**233 in excess-511 with 10-bit representation.
-233 + 511 = 278_{10} = 0100010110_2

Exerc. 3.25: Perform binary subtraction by taking the two's-complement of the subtrahend:
(a) 100110_2-111_2; **(b)** 100110_2-10000_2; **(c)** 1010101_2-11_2; **(d)** 1000001_2-1000000_2.

Answer:

1. In the first step, find the 2's complement of the subtrahend.
2. Add the complement number with the minuend.
3. If we get the carry by adding both numbers, we discard this carry and the result is positive else take 2's complement of the result which will be negative.

(a) 100110 - 111 ⇔ 100110 – 000111 ⇔ 100110 + 111001 = [~~1~~]011111 = 011111 = 11111
Two's complement of 000111 = 111000 + 1 = 111001

(b) 100110 - 10000 ⇔ 100110 – 010000 ⇔ 100110 + 110000 = [~~1~~]010110 = 10110
Two's complement of 010000 = 101111 + 1 = 110000

(c) 1010101 – 11 ⇔ 1010101 + 1111101 = ~~1~~1010010 = 1010010

(d) 1000001 – 1000000 ⇔ 1000001 + 1000000= ~~1~~0000001 = 1
(complement of 1000000 -> 0111111 + 1 -> 1000000 then add and drop carry bit)

Exerc. 3.26: Add the following pairs of unsigned binary numbers, explicitly indicating the carries:

(a)	**(b)**	**(c)**	**(d)**
11010	111010	1001111010	1101011
+ 1010	+ 101010	+ 1011010	+ 1011000

Answer:

(a)	(b)	(c)	(d)
11 1	111 1	1111 1	1111
11010	111010	1001111010	1101011
+ 1010	+ 101010	+ 1011010	+ 1011000
100100	1100100	1011010100	11000011

Exerc. 3.27: Add the following pairs **of 8-bit two's-complement** numbers, explicitly indicating situations of overflow:

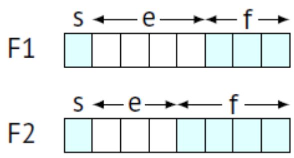
(a)	**(b)**	**(c)**	**(d)**
1001 1010 + 1000 1010	0111 1010 + 0110 1010	1101 1101 + 1110 1101	0110 1011 + 0101 1000

Answer:
From page 35 of class manual: "an addition overflows whenever the signs of the addends are the same (both numbers are either positive or negative) and the sign of the sum is different from the addends' sign." In other words:
"In two's complement, If the sum of two positive numbers yields a negative result, the sum has overflowed. If the sum of two negative numbers yields a positive result, the sum has overflowed. Otherwise, the sum has not overflowed."

(a)	(b)	(c)	(d)
10011010 + 10001010 100100100	01111010 +01101010 11100100	11011101 + 11101101 111001010	01101011 + 01011000 11000011
With 8 bits we have Overflow: adding 2 negatives we have a positive	Overflow: adding 2 positive returns a negative	Result: 11001010 Dropped last carry	Overflow: adding 2 positive returns a negative

Exerc. 3.28

Exerc. 3.28: Consider two floating-point formats F1 and F2, with 8 bits, based on all the principles presented in Section 3.6, namely normal numbers, subnormal numbers, special values, etc.



(a) Indicate the mathematical expressions that can be used to calculate the normal numbers in both formats.
(b) For each format, indicate the bit patterns and the respective decimal value for i) the smallest positive subnormal number, ii) the largest subnormal number, iii) the smallest positive normal number, iv) one, and the v) largest normal number.
(c) Calculate the decimal values of the following bit patterns for the F1 format: i) 10110011, ii) 01111010, iii) 10010001, iv) 00000011, v) 11000001.
(d) Represent in the F1 format, the following values: i) -111.01_3, ii) 128_{10}, iii) 111.01_{10}, iv) $-18C_{16}$, v) 0.005_8.
(e) Convert the following numbers represented in the F1 format into the F2 format: i) 00110011, ii) 11101001, iii) 00010000, iv) 11001110, v) 10000010. Overflow must be represented by $\pm\infty$, underflow by ± 0 and the roundings must be made to the closest value.

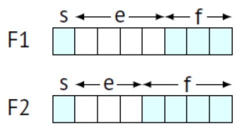
Answers:

The exponent is encoded in an excess format. The bias value is a number near the middle of the range of possible values that is selected to represent zero. The bias typically equals 2^{k-1}-1, where k is the number of bits in the exponent. The actual exponent is found by subtracting the bias from the stored exponent.

The mantissa is always normalized (1.xyz…) so it always starts with a 1. Because of that we can "ignore" that 1 in the representation. This bit is also the only one that is to the left of the binary point. So, the only part of the mantissa that needs to be represented in the bit pattern is its fractional part.

(a) Indicate the mathematical expressions that can be used to calculate the normal numbers in both formats.

Equation 3.6 page 37:
$V = (-1)^S \times (1 + f) \times 2^{e-b}$
The bias typically equals 2^{k-1}-1, where k is the number of bits in the exponent.



General formula for **normal** numbers:
$V = -1^S \times (1.0 + 0.M) \times 2^{e-bias}$

F1: $b = 2^{k-1}-1 = 2^3-1 = \mathbf{7}$
$V = (-1)^S \times (1 + f) \times 2^{e-7}$

F2: $b = 2^{k-1}-1 = 2^2-1 = \mathbf{3}$
$V = (-1)^S \times (1 + f) \times 2^{e-3}$

3.28 (b) For each format, indicate the bit patterns and the respective decimal value for…

Book page 38:
"The all-zeros exponent is reserved to represent subnormal numbers and zero. A subnormal number (or denormalised number) is a non-zero number with magnitude smaller than the smallest positive normal number. Its exponent value is fixed to be 1-bias and the mantissa M is restricted by the condition $0 \leq M < 1$ (there is no leading 1). So, for the all-zeros e exponent, the value V of a subnormal number is given by the following equation: $V = (-1)^S \times f \times 2^{e-b}$

A floating-point number may be recognized as subnormal whenever its exponent is the least value possible. For the mantissa the interpretation is that if the exponent is non-minimal, there is an implicit leading 1, and if the exponent is minimal, there isn't, and the number is subnormal.

when converting a number to binary excess format, offset is added to the original number and when retrieving original number, it's subtracted.

3.28 (b) i) the smallest positive subnormal number

0 for the signal, 0000 for the exponent (smallest possible) and the smallest possible mantissa with 3 digits and not being zero: 001

0	0	0	0	0	0	0	1

$V = (-1)^0 \times 2^{-2} \times 2^{0-7} = 2^{-9} = 1 / 512$
NOTE: In the mantissa we write 2^{-2} instead of 2^{-3} because this is a subnormal number.

For F2

0	0	0	0	0	0	0	1

$V = (-1)^0 \times 2^{-3} \times 2^{0-3} = 2^{-6} = 1 / 64$

3.28 (b) ii) the largest subnormal number

0 for the signal, 0000 for the exponent (smallest possible) and the largest possible mantissa with 3 digits: 111

0	0	0	0	0	1	1	1

$V = (-1)^0 \times (2^0 + 2^{-1} + 2^{-2}) \times 2^{0-7} = (4\times2^{-2} + 2\times2^{-2} + 1\times2^{-2}) \times 2^{-7} = (4\times2^{-9} + 2\times2^{-9} + 1\times2^{-9}) = 7\times2^{-9}$
$V = 7 / 512$

0	0	0	0	1	1	1	1

$V = (-1)^0 \times (2^0 + 2^{-1} + 2^{-2} + 2^{-3}) \times 2^{0-3} = (8\times2^{-3} + 4\times2^{-3} + 2\times2^{-3} + 1\times2^{-3}) \times 2^{-3} =$
$= (8\times2^{-6} + 4\times2^{-6} + 2\times2^{-6} + 1\times2^{-6}) = 15\times2^{-6}$
$V = 15 / 64$

3.28 (b) iii) the smallest positive normal number

0 for the signal, 0001 for the exponent (smallest possible and normal) and the smallest possible mantissa with 3 digits: 000

0	0	0	0	1	0	0	0

For normal number: $-1^S \times (1.0 + 0.M) \times 2^{e-bias}$
Exponent = e – bias = 1 – 7 = -6
$V = (-1)^0 \times (1 + 0) \times 2^{1-7} = 2^{-6}$
$V = 1 / 64$

0	0	0	1	0	0	0	0

Exponent = e – bias = 1 – 3 = -2
$V = (-1)^0 \times (1 + 0) \times 2^{1-3} = 2^{-2}$
$V = 1 / 4$

3.28 (b) iv) One

We know that the mantissa (for normal numbers) starts always with 1 (not represented). So, for the number 1, the mantissa bits will all be zero.

Now the exponent. We know the offset is 7 (see (a)). We want to store the number 0 in the exponent. So, we add 0 + 7 = 7 = 0111_2 If we stored 0 directly it would be a subnormal number!

0	0	1	1	1	0	0	0

For normal number: $-1^S \times (1.0 + 0.M) \times 2^{e\text{-}bias}$
V = $(-1)^0 \times (1 + 0) \times 2^{7-7}$ = 1x1x1
V = 1

For F2 we just need to represent the exponent. The bias is 3 (see (a)) So exponent = 0 + 3 = 011_2

0	0	1	1	0	0	0	0

V = $(-1)^0 \times (1 + 0) \times 2^{3-3}$ = 1x1x1
V = 1

3.28 (b) v) the largest normal number.

We will use the largest possible value for the mantissa (all 1s) and the largest for the exponent. Exponents of all zeros or all ones are reserved for subnormal numbers or special values. Largest exponent with 4 bits = 1110 (we cannot use 1111 and 0 on any other position would be smaller)

0	1	1	1	0	1	1	1

For normal number: $-1^S \times (1.0 + 0.M) \times 2^{e\text{-}bias}$
Exponent = 1110_2 = 14 ; F1 bias = 7
V = $(-1)^0 \times (1 + 2^{-1} + 2^{-2} + 2^{-3}) \times 2^{14-7} = (1+1/2+1/4+1/8) \times 2^7 = 1.875 \times 128 = 240$
V = 240

0	1	1	0	1	1	1	1

Exponent = 110_2 = 6 ; F2 bias = 3
V = $(-1)^0 \times (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}) \times 2^{6-3} = (1+1/2+1/4+1/8+1/16) \times 2^3 = 1.9375 \times 8 = 15.5$

3.28 (c) Calculate the decimal values of the following bit patterns for the F1 format: Exponent 4 bits, Mantissa 3 bits

3.28 (c) i) 10110011

1	0	1	1	0	0	1	1

V = $(-1)^1 \times (1 + 0\text{x}2^{-1} + 2^{-2} + 2^{-3}) \times 2^{6-7} = -1 \times (1.375) \times ½ = -0.687500$

3.28 (c) ii) 01111010

0	1	1	1	1	0	1	0

NaN
All exponent bits are 1 and mantissa not zero. Table 3.4, page 39.

3.28 (c) iii) 10010001

1	0	0	1	0	0	0	1

V = $(-1)^1 \times (1 + 2^{-3}) \times 2^{2-7} = -1 \times (1.125) \times 2^{-5} = -0.035156$

3.28 (c) iv) 00000011

0	0	0	0	0	0	1	1

SUBNORMAL V = $(-1)^0 \times (2^{-1} + 2^{-2}) \times 2^{0-7} = 0.75 \times 2^{-7} = 0.005859$

3.28 (c) v) 11000001

1	1	0	0	0	0	0	1

V = $(-1)^1 \times (1 + 2^{-3}) \times 2^{8-7}$ = -1 x (1.125) x 2 = -2,25

(d) Represent in the F1 format, the following values: i) -111.01_3, ii) 128_{10}, iii) 111.01_{10}, iv) $-18C_{16}$, v) 0.005_8.

3.28 (d) F1 format: Exponent 4 bits, Mantissa 3 bits

Answers:

3.28 (d) i) -111.01_3

Base 3 to decimal:

$(-111.01)_3 = -[(1 \times 3^2) + (1 \times 3^1) + (1 \times 3^0) + (0 \times 3^{-1}) + (1 \times 3^{-2})] = -13.11111_{10}$

To convert fraction to binary, start with the fraction in question and multiply it by 2 keeping notice of the resulting integer and fractional part. Continue multiplying by 2 until you get a resulting fractional part equal to zero. Then just write out the integer parts from the results of each multiplication.

0.11111 x 2 = **0** + 0.22222
0.22222 x 2 = **0** + 0.44444
0.44444 x 2 = **0** + 0.88888
0.88888 x 2 = **1** + 0.77776
0.77776 x 2 = **1** + 0.55552
0.55552 x 2 = **1** + 0.111
0.111 x 2 = **0** + 0.222
0.222 x 2 = **0** + 0.444
0.444 x 2 = **0** + 0.888
0.888 x 2 = **1** + 0.776
0.776 x 2 = **1** + 0.55
....
Until we reach a form where the fraction is zero, something like 0.abc x 2 = 1 + **0**
The 1s and 0s On the right side of the equal sign are collected to form our binary fraction. In our example we have: 0.**00011100011** ...
So:
$111.01_3 = -1101.00011100011010101_2$

We must encode this binary number into F1 format: Exponent 4 bits, Mantissa 3 bits
Normalize:
$-1101.00011100011010101_2 = -1.10100011100011010101_2 \times 2^3$

Signal: -1 => signal bit = 1
Our exponent = 3 + bias = 3 + 7 = 10 = 1010_2
Our mantissa = ~~1~~**101**00011100011010101_2

1	1	0	1	0	1	0	1

3.28 (d) ii) 128_{10}
$128_{10} = 10000000_2 = 1.000 \times 2^7$
Signal bit = 0 (positive number)
Our exponent = 7 + 7 = 14 = 1110
Our mantissa = 0000

0	1	1	1	0	0	0	0

3.28 (d) iii) 111.01_{10}

$111.01_{10} = 1101111.0000001010001111011_2 = 1.101111000000101 \times 2^6$
Signal bit = 0 (positive number)
Exponent = 6 + 7 = 13 = 1101
Mantissa = ~~1~~.101

0	1	1	0	1	1	0	1

3.28 (d) iv) $-18C_{16}$
F1 format: 4 bits exponent, 3 mantissa, bias 7

$-18C_{16} = -0001\ 1000\ 1100 = -110001100 = -1.10001100 \times 2^8$

Signal bit = 1 (negative number)
F1 exponent = 8 + 7 = 15 = 1111_2 **OVERFLOW** since max in excess-7 = 14

Because the value is less than the smaller negative integer (biggest in absolute value) we should represent -infinity. By definition s=1; e=all 1 and m= all 0 (table 3.4 from the book)

1	1	1	1	1	0	0	0

Value = -infinity

3.28 (d) v) 0.005_8

$0.005_8 = 0.005_8 = 0 \times 8^0 + 0 \times 8^{-1} + 0 \times 8^{-2} + 5 \times 8^{-3} = 0.009765625_{10}$

Convert decimal 0.009765625 to binary:

0.009765625 x 2 = 0 + 0.01953125
0. 01953125 x 2 = 0 + 0.0390625
0.0390625 x 2 = 0 + 0.078125
0.078125 x 2 = 0 + 0.15625
0.15625 x 2 = 0 + 0.3125
0.3125 x 2 = 0 + 0.625
0.625 x 2 = 1 + 0.25
0.25 x 2 = 0 + 0.5
0.5 x 1 = 1 + 0

$0.009765625_{10} = 0.000000101_2 = 1.01 \times 2^{-7}$ (normalized)

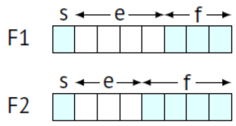
Signal bit = 0 (positive)
Exponent = -7 + 7 = 0000 => SUBNORMAL
Since this is a subnormal number we will use all mantissa bits
Mantissa = 101

0	0	0	0	0	1	0	1

Converting 00000101 to decimal, considering signal, 4 exponent digits and 3 mantissa digits we will get: 0.009766 (the original was 0.09765625)

3.28 (e) Convert the following numbers represented in the F1 format into the F2 format:
i) 00110011, ii) 11101001, iii) 00010000, iv) 11001110, v) 10000010.

Answers:
F1 format: Exponent 4 bits, Mantissa 3 bits
F2 format: Exponent 3 bits, Mantissa 4 bits



F1: $b = 2^{k-1}-1 = 2^3-1 = \mathbf{7}$
$V = (-1)^S \times (1 + f) \times 2^{e-7}$

F2: $b = 2^{k-1}-1 = 2^2-1 = \mathbf{3}$
$V = (-1)^S \times (1 + f) \times 2^{e-3}$

3.28 (e) i) 0 0110 011

Exponent bits = 0110 Exponent value (decimal) = 6
$V_{F1} = (-1)^0 \times (1 + 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3}) \times 2^{6-7} = 1.375\times1/2 = 0.6875$ (decimal)

Convert to binary. To avoid the decimal separator, multiply the decimal number with the base raised to the power of decimals (4 in this example) in result: $0.6875\times2^4 = 11$ Faster then the method we used in exercise 3.28 (d)

$11_{10} = 01011_2$ To get the original value (with decimal point) we reverse the process:
$01011_2 \times 2^{-4} = 0.1011_2$ Normalizing: $= 0.1011_2 = 1.011 \times 2^{-1}$

F2 Exponent value = -1 + F2_bias = -1 + 3 = 2 = $10_2 = 010_2$ (F2 demands 3 bits for exponent)
Mantissa = ~~1.~~0110 (we need 4 bits for F2 mantissa)

F1 0 0110 011 = F2 **0 010 0110**

3.28 (e) ii) 1 1101 001

Exponent = 1101 = 13 decimal
$V_{F1} = (-1)^1 \times (1 + 0\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}) \times 2^{13-7} = -1\times(1.125)\times64 = -72$
$72 = 1001000 = 1.001000 \times 2^6$

F2 exponent = 6 + F2_bias = 6 + 3 = 9 = 1001_2
We cannot represent the exponent 1001_2 with 3 bits.
Since the number is negative, we store -infinity. All exponent bits as 1 and the mantissa as zero: 11110000 (table 3.4 from the book)

3.28 (e) iii) 0 0010 000

Exponent = 0010 = 2 decimal
$V_{F1} = (-1)^0 \times (1 + 0x2^{-1} + 0x2^{-2} + 0x2^{-3}) \times 2^{2-7} = 2^{-5} = 0.03125$

Convert to binary. To avoid the decimal separator, multiply the decimal number with the base raised to the power of decimals (5 in this example) $0.03125 \times 2^5 = 1$
$1_{10} = 1_2$ Reverse the multiplication we did: $1_2 \times 2^{-5} = 0.00001_2$
Normalizing = $0.00001_2 = 1_2 \times 2^{-5} = 1.0_2 \times 2^{-5}$

F2 exponent = exponent + F2_bias = -5 + 3 = -2 In excess representation we can't have negative numbers. Lets try to **represent it as a subnormal number: $V = -1^s \times f \times 2^{1-bias}$**

By definition, for subnormal numbers, e=1-bias ⇔ e = 1-3 = -2 (for F2 format)
Move the decimal point so our exponent = -2 => Our number = $1.0_2 \times 2^{-5} = 0.010_2 \times 2^{-2}$
Answer: 0 000 0010 (exponent = 000, mantissa = 0010 in subnormal we take all bits)

3.28 (e) iv) 1 1001 110

$V_{F1} = (-1)^1 \times (1 + 1x2^{-1} + 1x2^{-2} + 0x2^{-3}) \times 2^{9-7} = -1x1.75x4 = -7$
$7_{10} = 111_2$ Normalizing $1.11x2^2$
F2 exponent = 2 + F2_bias = 2+3 = 5 = 101_2
Mantissa ~~1~~.11
F2 format: 1 101 1100

3.28 (e) v) 1 0000 010

Exponent bits = 0000 (subnormal number). Book page 38: *A subnormal number (or denormalised number) is a non-zero number with magnitude smaller than the smallest positive normal number.* ***Its exponent value is fixed to be 1-bias*** *[…] Value = $(-1)^s{}_x f_x 2^{1-bias}$*

Subnormal number => exponent = 000
F1 bias = 7
V_{F1} (subnormal) = $(-1)^1 \times (0x2^0 + 1x2^{-1} + 0x2^{-2}) \times 2^{1-7} = -(0.5) \times 2^{-6} = -0.5/64 = -1/128 = -1x2^{-7}$
V_{F1} (subnormal) = $-1x2^{-7} = 0.0078125_{10} = -0.0000001_2$ =

To store a subnormal number in F2 (bias = 3):
Signal = 1 (negative number)
F2 exponent = 000 (3 bits all zero) ⇔ e+3=0 ⇔ e = -3

To make e = -3 we rewrite the value: $V_{F2} = -0.0000001_2 = -0.0000001_2 \times 2^0 = -0.0001_2 \times 2^{-3}$

Mantissa = 0000 4 most significant bits. Since this is a subnormal number we do not assume the form (1+f) for the mantissa. The value for subnormal numbers is *Value = $(-1)^s{}_x f_x 2^{1-bias}$*

F2 format: 1 000 0000

Exerc. 3.29: Write a C program that calculates the decimal value for a bit pattern that represents a floating-point number. The inputs, provided through the command line, are: the bit pattern (sequence of non-separated 0s and 1s), the number of bits of the exponent e, and the number of bits of the mantissa f. If no pattern is provided, the program lists a pair (bit pattern, decimal value) for all possible 2^{1+e+f} bit patterns.

```
/*
    Exercise 3.29 of the book https://doi.org/10.21814/uminho.ed.33
*/
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

void convert2decimal(char bitPattern[], int nBitsExponent, int nBitsMantissa)
{
    if ((1 + nBitsExponent + nBitsMantissa) != strlen(bitPattern))
    {
        printf("Invalid bit pattern length: %d != %d",
                    (int)strlen(bitPattern), 1 + nBitsExponent + nBitsMantissa);
        return;
    }

    int signBit = (int)(bitPattern[0] - '0');

    char* exponent = (char *)malloc(nBitsExponent + 1);
    memcpy(exponent, bitPattern + 1, nBitsExponent);
    exponent[nBitsExponent] = 0;

    char* mantissa = (char *)malloc(nBitsMantissa + 1);
    memcpy(mantissa, bitPattern + 1 + nBitsExponent , nBitsMantissa);
    mantissa[nBitsMantissa] = 0;

    int exponentValue = 0;
    for (int i = 0; i < nBitsExponent; i++)
        exponentValue += (int)(exponent[i] - '0') * pow(2, nBitsExponent - 1 - i);

    int count_ones = 0; int count_zeros = 0;
    for (int i = 0; i < nBitsExponent; i++)
    {
        if (exponent[i] == '1')
            count_ones++;    // exponent all ones represents infinity or NaN
        else                 // table 3.4 https://doi.org/10.21814/uminho.ed.33
            count_zeros++;   // exponent all zeros represent subnormal numbers and zero
    }

    int mantissaNonZeroBitCount = 0;
    for (int i = 0; i < nBitsMantissa; i++) mantissaNonZeroBitCount += (int)(mantissa[i] - '0');

    if(count_ones == nBitsExponent)
    {
        printf("%s\t", bitPattern);
        if (mantissaNonZeroBitCount == 0)
            printf("%sInfinity\n", (signBit == 1) ? "-" : "+");
        else
            printf("NaN\n");
        return;
    }

    // In normal numbers the first bit of the mantissa is 1 x 2^0 = 1
    double mantissaValue = (count_zeros != nBitsExponent) ? 1 : 0;
    for (int i = 0; i < nBitsMantissa; i++)
    {
        if(count_zeros != nBitsExponent) // normal number
            mantissaValue += (int)(mantissa[i] - '0') * pow(2, -1 - i);
        else
            mantissaValue += (int)(mantissa[i] - '0') * pow(2, - i);
    }

    int bias = (int)pow(2, nBitsExponent - 1) - 1;
    double valueBase10 = pow(2, exponentValue - bias) * mantissaValue;
    if (signBit == 1) valueBase10 = -valueBase10;
```

```
    printf("%s\t%f\t%s\n", bitPattern, valueBase10,
    (count_zeros == nBitsExponent && mantissaNonZeroBitCount != 0) ? "(Subnormal number)" : "");
}

void printSeries(int nBitsExponent, int nBitsMantissa)
{
    int nBits = 1 + nBitsExponent + nBitsMantissa;  // sign + exponent bits + mantissa bits
    char bitPattern[nBits + 1];                     // +1 for null terminator
    bitPattern[nBits] = 0;                          // null terminator at end of pattern

    unsigned long long maxValue = (unsigned long long)pow(2, nBits-1) - 1;
    for (int sign = 0; sign <= 1; sign++)
        for(unsigned long long iterator = 0; iterator <= maxValue; iterator++)
            {
                for(int i = 0; i < nBits; i++)
                {
                    bitPattern[nBits-i-1] = (iterator & (1 << i)) ? '1' : '0';
                }
                bitPattern[0] = (char)(sign + '0');
                convert2decimal(bitPattern, nBitsExponent, nBitsMantissa);
            }
}

int main(void)
{
    char bitPattern[256];
    int nBitsExponent; int nBitsMantissa; int go = 1;
    char c;
    while (go)
    {
        printf("Enter number of bits for exponent: ");
        scanf("%d", &nBitsExponent);

        printf("Enter number of bits for mantissa: ");
        scanf("%d", &nBitsMantissa);

        printf("Enter bit pattern. Enter * to list all possible values: ");
        scanf("%s", bitPattern);

        if (bitPattern[0] == '*')
            printSeries(nBitsExponent, nBitsMantissa);
        else
            convert2decimal(bitPattern, nBitsExponent, nBitsMantissa);

        printf("\nAnother? (y/n): ");
        scanf(" %c",&c);
        go = (c == 'y');
    }
    return 0;
}
```

Some useful links:

- https://ncalculators.com/digital-computation/1s-2s-complement-calculator.htm
- https://www.rapidtables.com/convert/number/decimal-to-binary.html
- https://www.rapidtables.com/convert/number/binary-to-decimal.html
- https://www.rapidtables.com/convert/number/binary-to-hex.html
- https://www.rapidtables.com/convert/number/hex-to-binary.html
- https://www.rapidtables.com/convert/number/base-converter.html
- https://projects.klickagent.ch/prozessorsimulation/?converter=true
- https://www.omnicalculator.com/math/binary-subtraction (with steps)
- https://www.h-schmidt.net/FloatConverter/IEEE754.html
- https://trekhleb.dev/blog/2021/binary-floating-point/ (only for normal numbers)
- https://indepth.dev/posts/1019/the-simple-math-behind-decimal-binary-conversion-algorithms

Exerc. 4.1: After the execution of instruction **movl $721, %ebx**, what are the decimal values for the contents of registers bh and bl?

Data Registers

The IA-32 processors provides four 32-bits data registers, they can be used as:

- Four 32-bits registers (EAX, EBX, ECX, EDX)
- Four 16-bits registers (AX, BX, CX, DX)
- Eight 8-bits registers (AL, AH, BL, BH, CL, CH, DL, DH)

32-bits registers (31...0)	Bits 31...16	Bits 15...8	Bits 7...0
EAX		AH	AL
EBX		BH	BL
ECX		CH	CL
EDX		DH	DL

The data registers can be used in most arithmetic and logical instructions. But when executing some instructions, some registers have special purposes.

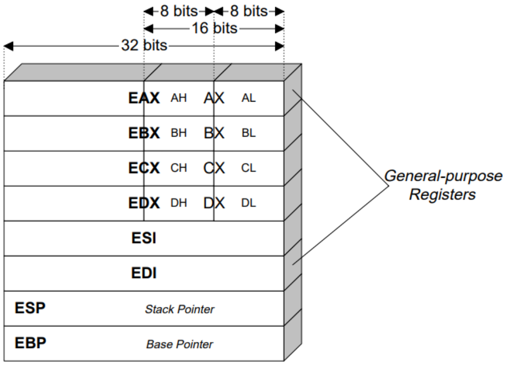


Figure 1. The x86 register set.

https://www.cs.dartmouth.edu/~sergey/cs258/tiny-guide-to-x86-assembly.pdf

Answer:
%ebx contains 721_{10} = 00000010 11010001
%ebh = 2_{10}
%ebl = 209_{10}

Exerc. 4.2: Consider that the following values are stored at the indicated memory addresses and registers. All values are represented in hexadecimal.

address	value	address	value	register	value
110	FF	118	13	eax	110
111	0	119	0	ebx	A
112	0	11A	0	ecx	1
113	0	11B	0	edx	3
114	AB	11C	55		
115	0	11D	0		
116	0	11E	0		
117	0	11F	0		

See also table 4.2 from the book!

(a) Calculate the values for the indicated operands:
i) %eax, ii) 0x114, iii) $0x118,
iv) (%eax), v) 4(%eax), vi) 9(%eax,%edx), vii) 280(%ecx,%edx), viii) 0xFC(,%edx,8),
ix) 2(%eax,%ebx).

Answer:

i) %eax = 110_{16} Contents of %eax

ii) 0x114 = AB_{16} Contents of memory at cell 0x114

iii) $0x118 = 118_{16} immediate value (constant)

iv) (%eax) = M[Rb] = M[110_{16}] = FF

v) 4(%eax) = M[110 + 4_{16}] = M[114_{16}] = AB

vi) 9(%eax,%edx) = M[Rb +Ri +Imm] = M[110_{16} + 3_{16} + 9_{16}] = M[11C] = 55_{16}

vii) 280_{10}(%ecx,%edx) = M[Rb +Ri +Imm] = M[1_{16} + 3_{16} + 118_{16}] = M[11C] = 55_{16}
280_{10}=118_{16}

viii) 0xFC(,%edx,8) = M[Ri x s + Imm] = M[3x8 + FC] = M[114_{16}] = AB

ix) 2(%eax,%ebx) = M[Rb + Ri + Imm] = M[110_{16} + A + 2] = M[11C] = 55_{16}

4.2 (b) For each instruction, indicate the result and where it is stored:

addl %eax, %ebx
addl (%eax), %ecx
subl 4(%eax), %edx
andl $43, (%eax,%edx,4)
decl %edx
incl 8(%eax)
imull %eax, %ebx
sall 2, %ebx

address	value	address	value	register	value
110	FF	118	13	eax	110
111	0	119	0	ebx	A
112	0	11A	0	ecx	1
113	0	11B	0	edx	3
114	AB	11C	55		
115	0	11D	0		
116	0	11E	0		
117	0	11F	0		

Answer:
Table 4.4 IA32 instructions for arithmetically operating the data and Table 4.6 The IA32 instructions for logically operating the data.

addl %eax, %ebx	%ebx = 110 + A = $\mathbf{11A_{16}}$
addl (%eax), %ecx	%ecx = M[110] + 1 = FF + 1 = $\mathbf{100_{16}}$
subl 4(%eax), %edx	%edx = 3 - M[110 + 4] = 3 – AB = 3 - 171_{10} = $\mathbf{-168_{10}}$
andl $43, (%eax,%edx,4)	(%eax,%edx,4) points to memory cell 110 + 3x4 = 110 + C = 11C M[11C] = 55 55_{16} & 43_{10} ⇔ 01010101 & 00101011 = 1 (operates in 4 bytes) **M[11C] = 1** (4 bytes)
decl %edx	**%edx = 2**
incl 8(%eax)	M[110 + 8] = M [118] = 13 **M [118] =** $\mathbf{14_{16}}$ **(with 4 bytes)**
imull %eax, %ebx	Book Pag. 55: "The single-operand form of the IMUL instruction executes a signed multiply of a byte, word, or double-word by the contents of the al, ax, or eax registers and stores the product in the ax, dx:ax or edx:eax registers, respectively. The two-operand form of IMUL executes a signed multiply of a register or memory word or double-word by a register word or double-word and stores the product in that register word or long word." %eax (= 110) x %ebx = 110 x A = $\mathbf{AA0_{16}}$ (stored in %ebx)
sall 2, %ebx	arithmetic left shift double-word %ebx = %ebx << 2 = A << 2 = 1010 << 2 = 101000 = $\mathbf{28_{16}}$

Exerc. 4.3: Complete the targets of the instructions. See page 62 of book
40F780: 75 03 jne XX

jne -> Jump if note qual / jump if not zero
"The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are program counter relative. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. **These offsets can be encoded using one, two, or four bytes**. A second encoding method is to give an "absolute" address, using four bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations."[1]

40F780: 75 03 **jne XX** (XX = destination)
Destination = Next memory position + 1 (byte 03) + 03 = 40F781 + 4 = **40F785**

8318A1: 0F 85 F1 FE FF FF **jne XX**
"**offsets can be encoded using one, two, or four bytes**" (see introductory text) so our jmp instruction is 0F 85 (2 bytes)
The last 4 bytes are F1 FE FF FF **but** IA32 is little endian so the offset is: FF FF FE F1
FFFFFEF1 is negative (signal bit is 1)
F F F F F E F 1
1111 1111 1111 1111 1111 1110 1111 0001
in two's complement:
0000 0000 0000 0000 0000 0001 0000 1111
0 0 0 0 0 1 0 F

FFFFFEF1 is negative in two's complement. FFFFFEF1 = -10F
Destination = Next memory position + 5 (bytes 85 F1 FE FF FF) - 10F = 8318A2 + 5 - 10F = $\mathbf{831798_{16}}$

[1] http://gec.di.uminho.pt/Discip/IA32_gas/csapp-concepts-AnV1.pdf

Exerc. 4.4: Indicate the addresses of the instructions.

x : 77 20 jne 300834
300834 = (x+1) + 1 (byte 20) + 20 ⇔ x = 300834 -22 = **300812**

x : EB E8 jmp 854FA2
854FA2 = (x+1) + 1 (byte E8) + E8 ⇔ x = 854FA2 - EA = **854EB8**

Exerc. 4.5: Consider the following C program, named sc1.c:

```c
#include <stdio.h>
int a, b, c;
int main()
{
    scanf("%d", &a);
    b = a * 2;
    c = b - a;
    printf("%d %d\n", b, c);
}
```

Generate the assembly code for this program, with the following command line:

```
gcc -m32 -O0 -S -o sc1-0.s sc1.c
```

(a) Identify how each C instruction/constructor was translated into assembly code
(b) Repeat the previous question, but first replace the scanf instruction with a=10

Answer:

(a) Identify how each C instruction/constructor was translated into assembly code. Here we will use the output from https://godbolt.org/. In class show and explain the result of

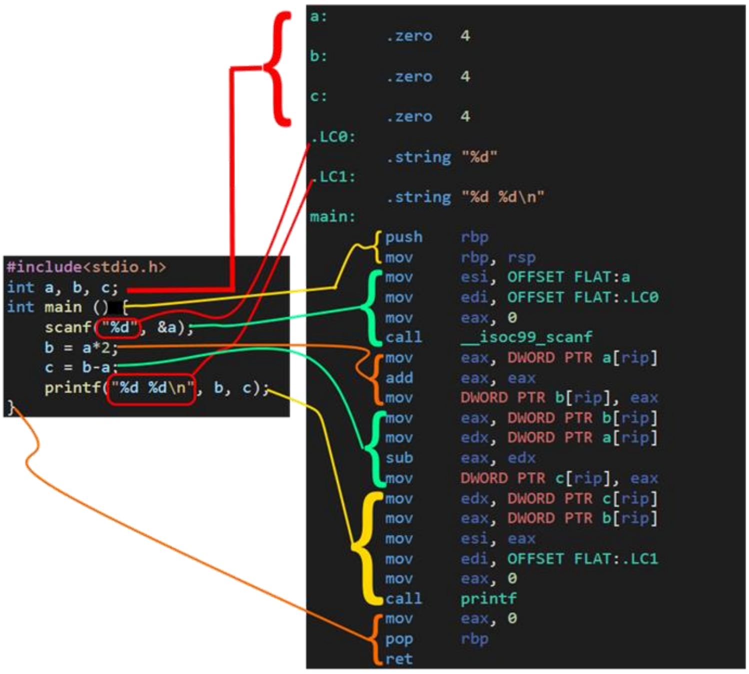
gcc -m32 -O0 -S -o sc1-0.s sc1.c

If you get the error "fatal error: bits/libc-header-start.h: No such file or directory" That's because you are compiling in a 64 bit machine and gcc was installed with only 64 library files. To support 32 bit execute:

sudo apt install gcc-multilib

if the problem persists try:

sudo apt install libc6-dev-i386



(b) Repeat the previous question, but first replace the scanf instruction with:
a=10;

```
main:
        push    rbp
        mov     rbp, rsp
        mov     esi, OFFSET FLAT:a
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    __isoc99_scanf
        mov     eax, DWORD PTR a[rip]
        add     eax, eax
        mov     DWORD PTR b[rip], eax
        mov     eax, DWORD PTR b[rip]
        mov     edx, DWORD PTR a[rip]
        sub     eax, edx
        mov     DWORD PTR c[rip], eax
        mov     edx, DWORD PTR c[rip]
        mov     eax, DWORD PTR b[rip]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

```
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR a[rip], 10
        mov     eax, DWORD PTR a[rip]
        add     eax, eax
        mov     DWORD PTR b[rip], eax
        mov     eax, DWORD PTR b[rip]
        mov     edx, DWORD PTR a[rip]
        sub     eax, edx
        mov     DWORD PTR c[rip], eax
        mov     edx, DWORD PTR c[rip]
        mov     eax, DWORD PTR b[rip]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
```

Exerc. 4.6: Consider the following C program, named sc2.c:

```c
#include<stdio.h>
int i=10, j, k, l;
int main ()
{
    scanf("%d", &j);
    if (i<j)
        k = i+j;
    else
        k = i-j;
    l=3*k;
}
```

Generate the assembly code for this program, with the following command lines:
gcc -m32 -O0 -S -o sc2-0.s sc2.c

4.6 (a) Identify how each C instruction/constructor was translated into assembly code.

```c
#include<stdio.h>
int i=10, j, k, l;
int main ()
{
    scanf("%d", &j);
    if (i<j)
        k = i+j;
    else
        k = i-j;
    l=3*k;
}
```

```
i:
        .long   10
j:
        .zero   4
k:
        .zero   4
l:
        .zero   4
.LC0:
        .string "%d"
main:
        push    rbp
        mov     rbp, rsp
        mov     esi, OFFSET FLAT:j
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    __isoc99_scanf
        mov     edx, DWORD PTR i[rip]
        mov     eax, DWORD PTR j[rip]
        cmp     edx, eax
        jge     .L2
        mov     edx, DWORD PTR i[rip]
        mov     eax, DWORD PTR j[rip]
        add     eax, edx
        mov     DWORD PTR k[rip], eax
        jmp     .L3
.L2:
        mov     eax, DWORD PTR i[rip]
        mov     edx, DWORD PTR j[rip]
        sub     eax, edx
        mov     DWORD PTR k[rip], eax
.L3:
        mov     edx, DWORD PTR k[rip]
        mov     eax, edx
        add     eax, eax
        add     eax, edx
        mov     DWORD PTR l[rip], eax
        mov     eax, 0
        pop     rbp
        ret
```

`int i=10, j, k, l;`	`i:`
	`.long 10`
	`j:`
	`.zero 4`
	`k:`
	`.zero 4`
	`l:`
	`.zero 4`
	`.LC0:`
	`.string "%d"`

`int main ()`
`{`
	`main:`
	`push rbp`
	`mov rbp, rsp`

`scanf("%d", &j);`	`mov esi, OFFSET FLAT:j`
	`mov edi, OFFSET FLAT:.LC0`
	`mov eax, 0`
	`call __isoc99_scanf`

`if (i<j)`	`mov edx, DWORD PTR i[rip]`
	`mov eax, DWORD PTR j[rip]`
	`cmp edx, eax`
	`jge .L2`

`k = i+j;`	`mov edx, DWORD PTR i[rip]`
	`mov eax, DWORD PTR j[rip]`
	`add eax, edx`
	`mov DWORD PTR k[rip], eax`
	`jmp .L3`

`else`
`k = i-j;`
	`.L2:`
	`mov eax, DWORD PTR i[rip]`
	`mov edx, DWORD PTR j[rip]`
	`sub eax, edx`
	`mov DWORD PTR k[rip], eax`

`l=3*k;`	`mov edx, DWORD PTR k[rip]`
	`mov eax, edx`
	`add eax, eax`
	`add eax, edx`
	`mov DWORD PTR l[rip], eax`

`}`	`mov eax, 0`
	`pop rbp`
	`ret`

(b) Identify which modifications occur if the constant 3 in the instruction “l=3*k” is replaced by 4, 7, 9, 24, and 39.

Answer:

l=3*k;

```
.L3:
        mov     edx, DWORD PTR k[rip]
        mov     eax, edx
        add     eax, eax
        add     eax, edx
        mov     DWORD PTR l[rip], eax
```

l=4*k;

```
.L3:
        mov     eax, DWORD PTR k[rip]
        sal     eax, 2
        mov     DWORD PTR l[rip], eax
```

l=7*k;

```
.L3:
        mov     edx, DWORD PTR k[rip]
        mov     eax, edx
        sal     eax, 3
        sub     eax, edx
        mov     DWORD PTR l[rip], eax
```

l=9*k;

```
.L3:
        mov     edx, DWORD PTR k[rip]
        mov     eax, edx
        sal     eax, 3
        add     eax, edx
        mov     DWORD PTR l[rip], eax
```

l=24*k;

```
.L3:
        mov     edx, DWORD PTR k[rip]
        mov     eax, edx
        add     eax, eax
        add     eax, edx
        sal     eax, 3
        mov     DWORD PTR l[rip], eax
```

l=39*k;

```
.L3:
        mov     eax, DWORD PTR k[rip]
        imul    eax, eax, 39
        mov     DWORD PTR l[rip], eax
```

Exerc. 4.7: Consider the following program written in C and analyze the assembly code generated by the gcc compiler with -O0 option.

```c
#include<stdio.h>
int array[100], sum=0;
int main () {
    int i;
    for (i=0; i<100; i++)
        scanf("%d", &array[i]);
    for (i=0; i<100 && array[i]>0; i++)
        sum += array[i];
}
```

Answer:

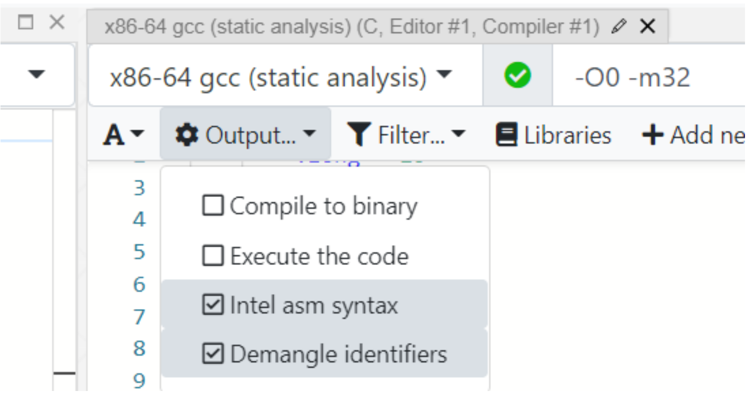
CDQE - Convert Doubleword to Quadword
CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

test performs a bit-wise logical AND of the two operands.
SF, ZF and PF flags are set according to the result.


```c
#include<stdio.h>
int array[100], sum=0;
int main () {
    int i;
    for (i=0; i<100; i++)
        scanf("%d", &array[i]);
    for (i=0; i<100 && array[i]>0; i++)
        sum += array[i];
}
```


```
array:
        .zero   400                         int array[100]
sum:
        .zero   4                           int sum=0;
.LC0:
        .string "%d"
main:
        push    rbp                         int main () {
        mov     rbp, rsp                    int i;
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 0        i=0;
        jmp     .L2                         Goto L2 unconditionally
.L3:                                        Start of scanf("%d", &array[i]);
        mov     eax, DWORD PTR [rbp-4]      eax <- i
        cdqe                                Expand eax to 64 bit rax
        sal     rax, 2                      Multiply by 4
        add     rax, OFFSET FLAT:array      Add start of array
        mov     rsi, rax                    rsi <- address of array
        mov     edi, OFFSET FLAT:.LC0       edi <- addr. Of str "%d"
        mov     eax, 0                      eax <- 0 before call
        call    __isoc99_scanf              Call scanf
        add     DWORD PTR [rbp-4], 1        i++
.L2:
        cmp     DWORD PTR [rbp-4], 99       i<100 ?
        jle     .L3                         Jump to L3 if <= 99
        mov     DWORD PTR [rbp-4], 0        i = 0 start of 2d for
        jmp     .L4                         Goto L4 (unconditionally)
.L6:
        mov     eax, DWORD PTR [rbp-4]      eax <- i
        cdqe                                Expand eax to 64 bit rax
        mov     edx, DWORD PTR array[0+rax*4]   edx <- array[i]
        mov     eax, DWORD PTR sum[rip]     eax <- sum
        add     eax, edx                    eax <- sum + array[i]
        mov     DWORD PTR sum[rip], eax     Save result in var sum
        add     DWORD PTR [rbp-4], 1        i++
.L4:
        cmp     DWORD PTR [rbp-4], 99       i<100 ?
        jg      .L5                         Jmp to L5 if > 99 else exit f.
        mov     eax, DWORD PTR [rbp-4]      eax <- i
        cdqe                                Expand eax to 64 bit rax
        mov     eax, DWORD PTR array[0+rax*4]   eax<-array[i]
        test    eax, eax                    array[i] > 0 ?
        jg      .L6                         Goto L6 if array[i] > 0 else exit f.
.L5:
        mov     eax, 0                      eax <- 0
        leave                               mov %ebp, %esp
                                            pop %ebp
        ret
```

NOTE: To get 32 bit output on https://godbolt.org/ set the following options:



Exerc. 4.8: Consider the following C program and compile it into IA-32 assembly code with the gcc compiler.

```
#include <stdio.h>
int main() {
   int n;
   scanf("%d", &n);
   if (n%2!=1)
      badDec2bin(16);
   else
      badDec2bin(44);
}
```

```
int badDec2bin (int n) {
   int c;
   for (c=16; c>=0; c--) {
      if (n»c & 16)
         printf("1");
      else
         printf("0");
   }
}
```

4.8 (a) Build with the maximum detail the stack frame for function badDec2bin, indicating the size and the position of each element.

Calling a function

Before executing a function, a program pushes all of the parameters for the function onto the stack in the reverse order that they are documented. Then the program issues a call instruction indicating which function it wishes to start. **The call instruction does two things:**

1. First it pushes the address of the next instruction, which is **the return address**, onto the stack.
2. Then, it modifies the instruction pointer %eip to point to the start of the function."

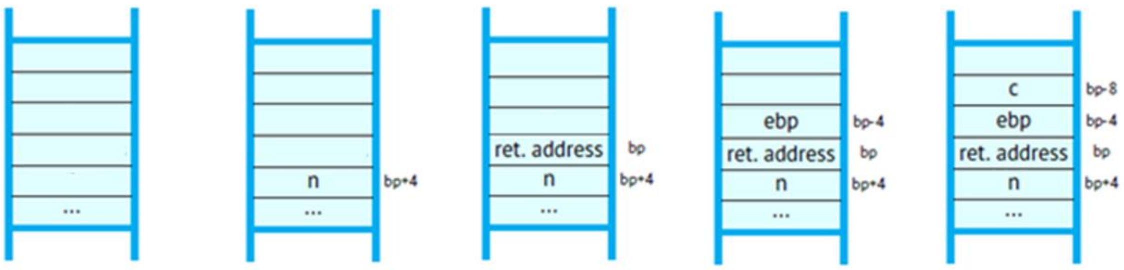
During the setup, the following two instructions are carried out immediately:

1. **push %ebp**
   Save the current base pointer register. The base pointer is a special register used for accessing function parameters and local variables. **The stack frame is delimited by two pointers: %ebp serves as the pointer pointing to the bottom of the stack frame and %esp serves as the pointer pointing to the top of the stack frame**. Once the current function (i.e. callee) is done, we need to resume the execution of the caller function. This means that we need to **restore the caller's base pointer register %ebp when we are done with callee function**. Thus, we need to save the current base pointer register, which is the caller's for the future caller stack frame restoration.

2. **mov %ebp %esp**
   Once we save the caller's %ebp, we can now setup current stack frame's %ebp. The reason for this is that we must be able to access the function parameters that were pushed earlier onto the stack by the caller function as fixed indexes from the base pointer. We cannot use stack pointer directly for accessing parameters because the stack pointer can move while the function is executing. At this point, the stack looks like this (4 bytes per memory position):

```
...            local variables
Caller's %ebp  <--- -4(%ebp)
Return address <--- (%ebp)
Argument 1     <--- 4(%ebp)
Argument 2     <--- 8(%ebp)
...
Argument N     <--- N*4(%ebp)
```

stack frame for function badDec2bin:



4.8 (b) Identify how the C instructions in highlighted lines have been translated into assembly code by the gcc compiler with -O0 option.

```c
int badDec2bin(int n)
{
    int c;
    for (c = 16; c >= 0; c--)
    {
        if (n >> c & 16)
            printf("1");
        else
            printf("0");
    }
}
```

```asm
badDec2bin:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 16
        jmp     .L2
.L5:
        mov     eax, DWORD PTR [rbp-4]
        mov     edx, DWORD PTR [rbp-20]
        mov     ecx, eax
        sar     edx, cl
        mov     eax, edx
        and     eax, 16
        test    eax, eax
        je      .L3
        mov     edi, 108
        call    putchar
        jmp     .L4
.L3:
        mov     edi, 48
        call    putchar
.L4:
        sub     DWORD PTR [rbp-4], 1
.L2:
        cmp     DWORD PTR [rbp-4], 0
        jns     .L5
        nop
        leave
        ret
```

4.8 (b) Identify how the C instructions in highlighted lines have been translated into assembly code by the gcc compiler with -O0 option.

```c
int main()
{
    int n;
    scanf("%d", &n);
    if (n % 2 != 1)
        badDec2bin(16);
    else
        badDec2bin(44);
}
```

The CDQ (**Convert Doubleword to Quadword**) instruction extends the sign bit of EAX into the EDX register

```asm
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        lea     rax, [rbp-4]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    __isoc99_scanf
        mov     eax, DWORD PTR [rbp-4]
        cdq
        shr     edx, 31
        add     eax, edx
        and     eax, 1
        sub     eax, edx
        cmp     eax, 1
        je      .L7
        mov     edi, 16
        call    badDec2bin
        jmp     .L8
.L7:
        mov     edi, 44
        call    badDec2bin
.L8:
        mov     eax, 0
        leave
        ret
```

Exerc. 4.9: Complete the C program based on the respective assembly code.
See this tutorial first: https://www.youtube.com/watch?v=wy3e52A7Lu8

```
int cmpXY (int x, int y){
       int val = ... ;
       if ( ... ){
              if ( ... )
                     val = ... ;
              else
                     val = ... ;
       }
       else
              if (x ... )
                     val = ... ;
              return val;
}
```

```
cmpXY: ...
       movl 12(%ebp), %eax
       movl 8(%ebp), %ecx
       movl $0, -4(%ebp)
       movl 8(%ebp), %edx
       cmpl 12(%ebp), %edx
       je LBB5
       jle LBB3
       movl $1, -4(%ebp)
       jmp LBB4
LBB3:  movl $2, -4(%ebp)
LBB4:  jmp LBB8
LBB5:  cmpl $10, 8(%ebp)
       jle LBB8
       movl $3, -4(%ebp)
LBB8:  movl -4(%ebp),  %eax
       ...
```

ESP register is the current stack pointer and EBP is the base pointer for the current stack frame

```
int cmpXY (int x, int y){
       int val = 0 ;
       if ( x != y ){
           if ( x > y )
               val = 1 ;
           else
               val = 2 ;
       }
       else
           if (x > 10 ) (*)
               val = 3 ;
        return val;
}
```

```
cmpXY: ...
       movl 12(%ebp), %eax       EAX <- [EBP+12] = y
       movl 8(%ebp), %ecx        ECX <- [EBP+8]  = x
       movl $0, -4(%ebp)         val <- 0 val is in [EBP-4]
       movl 8(%ebp), %edx        EDX <- [EBP+8] = x
       cmpl 12(%ebp), %edx       compare x with y
       je LBB5                   goto LBB5 if equal continue otherwise
       jle LBB3                  goto LBB3 if x less or equal y
       movl $1, -4(%ebp)         val <- 1
       jmp LBB4                  goto LBB4
LBB3:  movl $2, -4(%ebp)         val <- 2
LBB4:  jmp LBB8                  goto LBB8
LBB5:  cmpl $10, 8(%ebp)         compare x with 10
       jle LBB8                  goto LBB8
       movl $3, -4(%ebp)         val <- 3
LBB8:  movl -4(%ebp), %eax       EAX <- val
       ...
```

`if ( x > y ) or if ( x >= y )` would output the same code:

```
cmpl 12(%ebp), %edx         compare x with y
je LBB5                     goto LBB5 if equal
jle LBB3                    goto LBB3 if less or equal
```

Note: In all exercises in this chapter, assume that a word is four bytes, and that the memory is addressed at the byte level

Exerc. 5.1: Consider four computers with different caches:

- **C1**: direct mapping, 2^{20} words of main memory, cache with 32 blocks, cache block with 16 words.
- **C2**: direct mapping, 2^{32} bytes of main memory, cache with 1024 blocks, cache block with 32 words.
- **C3**: fully associative mapping, 2^{16} words of main memory, cache with 64 blocks, cache block with 32 words.
- **C4**: fully associative mapping, 2^{24} words of main memory, cache with 128 blocks, cache block with 64 words.

Exerc. 5.1 (a) Calculate the number of blocks that exist in the main memory.

Problem data: C1: direct mapping, **2^{20} words of main memory**, cache with 32 blocks, cache **block with 16 words**.

If memory has 2^n addressable words and each block has k words than number of blocks in main memory M = $2^n/k$ (book page 77)

K = 16 = 2^4
M = $2^{20}/ 2^4$ = $\mathbf{2^{16}}$

Problem data: C2: direct mapping, **2^{32} bytes of main memory**, cache with 1024 blocks, cache **block with 32 words**.
From the exercise text we should assume a word = 4 bytes so, memory 2^{32} bytes = 2^{30} words
K = 32 = 2^5
M = $2^{30} / 2^5$ = $\mathbf{2^{25}}$

Problem data: C3: fully associative mapping, 2^{16} words of main memory, cache with 64 blocks, cache block with 32 words.
K = 32 = 2^5
M = $2^{16} / 2^5$ = $\mathbf{2^{11}}$

Problem Data: C4: fully associative mapping, 2^{24} words of main memory, cache with 128 blocks, cache block with 64 words.
K = 64 = 2^6
M = $2^{24} / 2^6$ = $\mathbf{2^{18}}$

Exerc. 5.1 (b) Draw the format of a memory address as seen by each cache, indicating the sizes of the tag, block (when applicable), and offset fields.

Book page 79:
"The direct mapping function can be implemented using the main memory address. For the purpose of cache access, each main memory address can be viewed as consisting of three fields: the **o LSBs identify a unique word or byte within a memory block (o = $\log_2 K$)**. The next b bits specify part of the block number (b = $\log_2 m$). The remaining t bits are saved in the tag (t = n - b - o).

Problem data: C1: direct mapping, 2^{20} words of main memory, cache with 32 blocks, cache block with 16 words.

Memory = 2^{20} words = 2^{22} bytes
Memory physical address bits = $\log2(2^{22})$ = 22 bits

Block size = 16 words = 16x4 bytes = 2^6 bytes
Block offset = log2(2^6) = 6 bits

Number of blocks in main memory = $2^{22}/2^6 = 2^{16}$
Number of Block bits = log2(2^{16}) = 16

22 memory address bits	
16 block number bits	6 block offset bits

Cache size = 32 blocks x 16 words = 2^5 x 2^6 bytes = 2^{11} bytes
Number of lines = blocks in cache = cache size / block size = 2^{11} / 2^6 = 2^5
Block line number bits = log2(2^5) = 5

Tag number bits = physical address bits – (line number bits + offset bits)
Tag number bits = 22 - (5 + 6) = 11

22 memory address bits		
16 block number bits		6 block offset bits
11 tag bits	5 Block line number bits	6 block offset bits

Offset, o: 6 bits
Block, s: 5 bits
Tag: 11 bits

Problem data: C2: direct mapping, 2^{32} bytes of main memory, cache with 1024 blocks, cache block with 32 words.

Memory = 2^{32} bytes
Memory physical address bits = log2(2^{32}) = 32 bits

Block size = 32x4 bytes = 2^7 bytes
Block offset = log2(2^7) = 7 bits

Number of blocks in main memory = $2^{32}/2^7 = 2^{26}$
Number of Block bits = log2(2^{25}) = 25

32 memory address bits	
25 block number bits	7 block offset bits

Cache size = 1024 blocks x 2^7 bytes = 2^{10} x 2^7 bytes = 2^{17} bytes
Number of lines/blocks in cache = cache size / block size = 2^{17} / 2^7 = 2^{10}
Block number bits = log2(2^{10}) = 10

Tag number bits = physical address bits – (line number bits + offset bits)
Tag number bits = 32 – (10+7) = 15

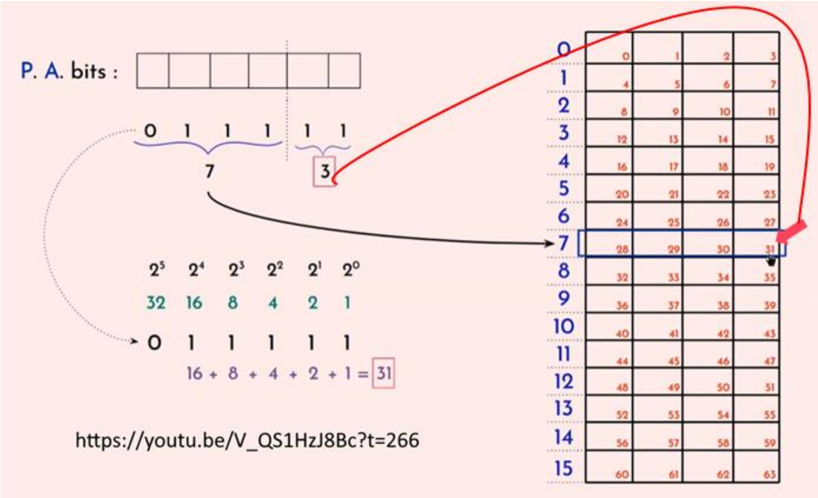
32 memory address bits		
25 block number bits		7 block offset bits
15 tag bits	10 Block line number bits	7 block offset bits

Offset, o: 7 bits
Block, s: 10 bits
Tag: 15 bits

Exerc. 5.1 (b)

Fully associative mapping:

- Tag bits hold block index (index of block in main memory)
- The higher order n bits of a memory address are the tag bits of the block in the cache. The tag bits are used to find if that memory block is in the cache
- Tag bit size = block index bit size
- The offset part indicates where, in the block, is this memory address



Problem data: C3: fully associative mapping, 2^{16} words of main memory, cache with 64 blocks, cache block with 32 words.

Memory = 2^{16} words = 2^{18} bytes
Physical Address bits = log2(2^{18}) = 18 = tag bits + block offset bits
Block size = 32 words = 32 x 4 bytes = 2^7 bytes
Block offset bits = log2(2^7) = 7
Tag = 18 – 7 = 11
Or…
Number of blocks in main memory = $2^{18}/2^7$ = 2^{11} blocks
Number of Block bits = log2(2^{11}) = **11 = Tag bit size**

Tag: 11 bits, offset: 7 bits

Problem data: C4: fully associative mapping, 2^{24} words of main memory, cache with 128 blocks, cache block with 64 words.

Memory = 2^{24} words = 2^{26} bytes
Memory physical address bits = log2(2^{26}) = 26 bits
Block size = 64 words = 64x4 bytes = 2^8 bytes => block offset bits = log2(2^8) = 8
Number of blocks in main memory = $2^{26}/2^8$ = 2^{18}
Number of Block bits = log2(2^{18}) = 18 = Tag bit size
Physical Address bits = tag bits + block offset bits
26 = 18 + block offset bits ⇔ **block offset bits = 8** (see block size above)

Tag: 18, offset: 8

Exerc. 5.1 (c) Indicate the cache block to where is mapped the memory reference $3DB63_{16}$ in C1 and $13463FA_{16}$ in C2. Specify in each case the tag value.

C1: direct mapping, 2^{20} words of main memory, cache with 32 blocks, cache block with 16 words.

$3DB63_{16}$
0011 1101 1011 0110 0011

From exercise b) we have

Offset: 6 bits
Block: 5 bits
Tag: 11 bits

22 memory address bits 00001111011 01101 100011		
11 tag bits 00001111011	5 Block number bits 01101	6 block offset bits 100011

Cache block = 01101_2 = 13
Tag = 00001111011_2

C2: direct mapping, 2^{32} bytes of main memory, cache with 1024 blocks, cache block with 32 words.

$13463FA_{16}$
000000010011010 0011000111 1111010

32 memory address bits		
15 tag bits 000000010011010	10 Block number bits 0011000111	7 block offset bits 1111010

Cache block = 0011000111= 199
Tag = 000000010011010

Exerc. 5.1 (d) Calculate the size in bytes of each cache. IGNORE validity bit
Note: In all exercises in this chapter, assume that a word is four bytes, and that the memory is addressed at the byte level

- **C1**: direct mapping, 2^{20} words of main memory, cache with 32 blocks, cache block with 16 words.
- **C2**: direct mapping, 2^{32} bytes of main memory, cache with 1024 blocks, cache block with 32 words.
- **C3**: fully associative mapping, 2^{16} words of main memory, cache with 64 blocks, cache block with 32 words.
- **C4**: fully associative mapping, 2^{24} words of main memory, cache with 128 blocks, cache block with 64 words.

Cache CAPACITY = number of sets x number of blocks per set x bytes per block
Cache SIZE = cache capacity + cache lines x tag bits
Number of lines = number of blocks in cache (book page 77)

C1 Cache capacity = 1 set x 32 blocks x 16 words = 2^5 x 2^4 x 2^2 bytes = 2^{11} bytes = **2048 bytes**
In each cache line we have 16 words plus 11 bits for the tag. 11 bits = 11/8 bytes
(Tag bits from 5.1b)
We have 32 cache lines (= number of cache blocks)
C1 cache size = cache capacity + 32 * 11/8 = 2048 + 44 = **2092**

C2 cache capacity = 1 x 1024 blocks x 32 words = 2^{10} x 2^5 x 2^2 bytes = 2^{17} bytes = **131072 bytes**
C2 cache lines = 1024
C2 cache size = cache capacity + cache lines x tag bits = 131072 + 1024 x 15 / 8 =
= 131072 + 1920 = **132992**
(Tag bits from 5.1b)

C3: fully associative mapping, 2^{16} words of main memory, cache with 64 blocks, cache block with 32 words.

C3 fully associative mapping => there is only one set
C3 Cache capacity = 1 x 64 blocks x 32 words = 2^6 x 2^5 x 2^2 bytes = 2^{13} bytes = **8192 bytes**
C3 cache lines = 64
From 5.1b we know that tag = 11 bits
C3 cache size = cache capacity + cache lines x tag bits = 8192 + 64 x 11 / 8 = 8192+88 = **8280**

C4 fully associative mapping => there is only one set
C4 Cache capacity = 1 x 128 blocks x 64 words = 2^7 x 2^6 x 2^2 bytes = 2^{15} bytes = **32768 bytes**
From 5.1b we know that tag = 18 bits
C4 cache size = cache capacity + cache lines x tag bits = 32768 + 128 x 18 / 8 = 32768 + 288 =
= **330656**

Exerc. 5.2: Consider a computer with a memory with **128Mib** words. Blocks are 64 words in length and the cache consists of 32Kib blocks. For a 2-way set associative cache mapping scheme, illustrate the format for a main memory address, including the fields and their sizes.

- A 2-way set associative cache mapping means we divide the cache in sets of 2 lines per set
- The memory blocks are mapped to a set in cache using the same logic as for direct mapping
- Inside each set, the mapping is associative mapping

1 MiB = 1024 x 1024 bytes
Memory size = 128 x 4 words = 128 x 4 x 1024 x 1024 bytes = 2^7 x 2^2 x 2^{10} x 2^{10} = 2^{29} bytes
Block size = 64 words x 4 = 2^6 x 2^2 = 2^8 bytes
Block offset: 8 bits
Number of blocks in memory = memory size / block size = 2^{21} Blocks
Number of bits to identify a memory block: 21

1 Kib = 1024 bytes = 2^{10} bytes
Number of lines = cache size / block size = 32Kib / 2^8 = 2^5 x 2^{10} / 2^8 = 2^{15} / 2^8 = 2^7 lines
Number of sets = number of lines / 2 (two-way) = 2^6 sets => we need 6 bits for set index
Set: 6 bits the least 6 significant bits of block index in main memory
Tag bits are the most significant bits of the block index. In this case 21-6 = **15 Msb**
Block offset: 8 bits, the least significant 8 bits of the Physical Address

29 memory address bits		
15 tag bits	6 set index bits	8 block offset bits

Answer: tag 15, set 6, offset 8

Exerc. 5.3: A 2-way set associative cache consists of four sets. The main memory contains 2Kib 8-word blocks.

Memory = 2Kib x 8 x 4 bytes = 2 x 2^{10} x 2^3 x 2^2 = 2^{16} bytes
Block size = memory size / number of blocks => it is given as being 8-word
Block size = 8 words x 4 = 2^3 x 2^2 = 2^5 bytes
Block offset: 5 bits

Number of blocks in memory = memory size / block size = 2^{16} / 2^5 = 2^{11} Blocks
Number of bits to identify a memory block (block index): 11

problem states we have 4 sets => set = 2 bits
Set: 2 bits the least 2 significant bits of block index in main memory

16 memory address bits		
11 block index bits		5 block offset bits
tag bits	2 set index bits	5 block offset bits

Tag bits = 11 - 2 = 9

Exerc. 5.3 (b) Compute the hit ratio for a program that loops three times from locations 8 to 55 in main memory. Assume that all instructions occupy four bytes.

hit ratio = cache hits / (cache hits + cache misses) ⬄ hit ratio = cache hits / cache requests
Memory block size = 32 bytes
Block 0 contains addresses 0 -> 31
Block 1 contains addresses 32 -> 63

Our program loops from 8 to 55 three times, Instructions are 4 bytes.
Memory requests highlighted:

Block	Memory Address
0 0-31	0
	4
	8
	12
	16
	20
	24
	28
1 32-63	32
	36
	40
	44
	48
	52
	56
	60

Memory requests for one loop: (55 – 8) / 4 = 47 / 4 = 11.75 => 12
Total memory requests = 3 x 12 = 36

First loop has 2 cache misses (cache starts invalid)
Loops 2 and 3 are all cache hits

hit ratio = cache hits / (cache requests)
hit ratio = (36-2) / 36 = 34 / 36 = 17 / 18
hit ratio = 17 / 18

Exerc. 5.4: A computer, using a set associative cache, has 2^{16} words of main memory and a cache of 32 blocks, and each cache block contains eight words.

(a) What is the format of a memory address as seen by a **2-way set associative cache**, i.e., what are the sizes of the tag, set, and word fields?

Memory size = 2^{16} x 4 bytes = 2^{16} x 2^2 = 2^{18} bytes => **Physical address bits = 18**
Block size = memory size / number of blocks => it is given as being 8-word
Block size = 8 words x 4 = 2^3 x 2^2 = 2^5 bytes
Block offset: 5 bits
Number of blocks in main memory = 2^{18} / 2^5 = 2^{13}

Number of sets?
Cache size = 32 x 8 words x 4 bytes = 2^5 x 2^3 x 2^2 = 2^{10} bytes
Number of cache lines = cache size / block size = 2^{10} / 2^5 = 2^5

The cache is 2-way (given) so every set is going to contain 2 lines
Number of sets = Number of cache lines / set size = 2^5 / 2^1 = 2^4 => **Set bits = 4**

Physical address bits = tag bits + set bits + block offset bits
Tag bits = 18 - 4 - 5 = 9

<table>
<tr><td colspan="3">18 memory address bits</td></tr>
<tr><td colspan="2">13 block index bits</td><td>5 block offset bits</td></tr>
<tr><td>9 tag bits</td><td>4 set index bits</td><td>5 block offset bits</td></tr>
</table>

Exerc. 5.4 (b) Repeat the previous question if the cache is 4-way set associative.
From a) we have:

Physical address bits = 18
Block offset: 5 bits

The cache is 4-way (given) so every set is going to contain 4 lines
Number of sets = Number of cache lines / set size = $2^5 / 2^2 = 2^3$ => **Set bits = 3**

Physical address bits = tag bits + set bits + block offset bits
Tag bits = 18 - 3 - 5 = 10

18 memory address bits		
13 block index bits		5 block offset bits
10 tag bits	3 set index bits	5 block offset bits

Exerc. 5.5: A computer uses a memory address word size of 8 bits. This computer has a 16-byte **direct-mapped cache** with 4 bytes per block. The format of a memory address as seen by the cache is the following:

4 bits	2 bits	2 bits
t	b	o

← 8 bits →

While running a program, the computer accesses several memory locations, according to the following sequence: **6D, B9, E3, 16**, **E3**, 4E, 4F, 14, 91, A4, A5, A7, A9, 98, and 99 (in hexadecimal). The memory addresses of **the first four accesses have been loaded into the cache blocks** as shown below. The contents of the tag are shown in binary and the cache contents are simply the hexadecimal addresses whose contents are stored at each cache location.

block	tag				
0	1110	(E3)	(E2)	(E1)	(E0)
1	0001	(17)	(16)	(15)	(14)
2	1011	(BB)	(BA)	(B9)	(B8)
3	0110	(6F)	(6E)	(6D)	(6C)

Exerc. 5.5 (a) What is the hit ratio for the memory reference sequence given above?
See "Exercise_5.5.ppsx" to watch the evolution of the cache shown below in its final state.

Request	Hit	Address	tag	block	offset
6D	0	0110 1101	0110	11	01
B9	0	1011 1001	1011	10	01
E3	0	1110 0011	1110	00	11
16	0	0001 0110	0001	01	10
E3	1	1110 0011	1110	00	11
4E	0	0100 1110	0100	11	10
4F	1	0100 1111	0100	11	11
14	1	0001 0100	0001	01	00
91	0	1001 0001	1001	00	01
A4	0	1010 0100	1010	01	00
A5	1	1010 0101	1010	01	01
A7	1	1010 0111	1010	01	11
A9	0	1010 1001	1010	10	10
98	0	1001 1000	1001	10	00
99	1	1001 1001	1001	10	01

hit ratio = cache hits / cache requests
we have 15 cache requests.
Hit ratio = 6 / 15 = 2 / 5

Exerc. 5.5 (b) What memory blocks are in the cache after the last address has been accessed?

		Block offset			
block	**tag**	**11**	**10**	**01**	**00**
00	**1001**	**(93)**	**(92)**	**(91)**	**(90)**
01	**1010**	**(A7)**	**(A6)**	**(A5)**	**(A4)**
10	**1001**	**(9B)**	**(9A)**	**(99)**	**(98)**
11	**0100**	**(4F)**	**(4E)**	**(4D)**	**(4C)**

Exerc. 5.6: Consider a byte-addressable computer with 24-bit addresses, a cache capable of storing a total of 64KiB of data, and blocks of 32 bytes. **Show the format of a 24-bit memory address for the following mapping functions**: (a) direct, (b) fully associative, and (c) 16-way set associative

Exerc. 5.6 (a) **direct**

24 memory address bits		
block number bits		block offset bits
tag bits	Block line number bits	block offset bits

Physical address bits: 24
Cache size = 64 KiB = 64 x 1024 = 2^6 x 2^{10} = 2^{16} bytes
Block size = 32 bytes = 2^5 Bytes => **block offset bits, o = 5**
Number of blocks in physical memory = 2^{24} / 2^5 = 2^{19} => block number bits = 19

Number of lines = blocks in cache (direct) = cache size / block size = 2^{16} / 2^5 = 2^{11}
Block line number bits, s = log2(2^{11}) = 11

Physical address bits = tag bits + block number bits + block offset bits
Tag bits = 24 – 11 – 5 = 8

Exerc. 5.6 (b) fully associative

24 memory address bits	
block number bits (tag)	block offset bits

Physical address bits: 24
Cache size = 64 KiB = 64 x 1024 = 2^6 x 2^{10} = 2^{16} bytes
Block size = 32 bytes = 2^5 Bytes => **block offset bits, o = 5**
Number of blocks in physical memory = 2^{24} / 2^5 = 2^{19} => block number bits = 19
Tag bits = 19

Exerc. 5.6 (c) 16-way

Already calculated:

Physical address bits: 24
Cache size = 64 KiB = 64 x 1024 = 2^6 x 2^{10} = 2^{16} bytes
Block size = 32 bytes = 2^5 Bytes => **block offset bits = 5**
Number of blocks in physical memory = 2^{24} / 2^5 = 2^{19} => block number bits = 19

Number of sets?
Number of cache lines = cache size / block size = 2^{16} / 2^5 = 2^{11}
The cache is 16-way (given) so every set is going to contain 16 lines = 2^4 lines
Set bits, s = log2(2^4) = **4**

Physical address bits = tag bits + set bits + block offset bits
Tag bits = 24 – 4 – 5 = 15

24 memory address bits		
19 block index bits		5 block offset bits
15 tag bits	4 set index bits	5 block offset bits

Useful links:

- Direct memory mapping cache
  - https://www.youtube.com/watch?v=V_QS1HzJ8Bc
  - https://www.youtube.com/watch?v=OxaYvJquPe0
- Associative mapping cache
  - https://www.youtube.com/watch?v=uwnsMaH-iV0
  - https://www.youtube.com/watch?v=OGDEsD3hdbk
- Set associative cache
  - https://www.youtube.com/watch?v=KhAh6thw_TI
  - https://www.youtube.com/watch?v=ejTCm7eHsM8