Microprocessor Engineering

Microprocessor Engineering

ŁUKASZ MAKOWSKI

Contents

Preface

Computers are useless. They can only give you answers. (Pablo Picasso)

You do *not have* to learn assembly language.
Please be aware that it is generally considered to be hard and laborious experience with no immediately visible revenue. But if you started reading this book it is possible that you *want* to or *need* to know more about computer architecture, assambly programming, microprocessors and associated electronics. Engineering experience that comes with skills in these areas can hardly be overestimated. Deeper understanding of microprocessors and related electronics is invaluable to both hardware and software engineers. Furthermore, as microprocessors became ubiquitous, it is virtually impossible to avoid them in present-day and future electrical projects.

Cars and computers are often compared as both of these inventions were very disruptive and eventually affected lives of the global population. So, let us make here a similar attempt and compare both technologies from the human point of view. The first car that used microprocessor was Cadillac Seville from 1978. Contemporary cars might have more than 100 microprocessors on board. Drivers and passengers are oblivious to this fact because these embedded systems "just work" and stay invisible unless there is some failure. Vehicular electronics, that strongly relies on microprocessors nowadays, must be highly reliable because human lives are on the stake.

Many drivers can list advertised parameters of their cars such as engine horsepower or fuel efficiency. However, most of them are not skilled in mechanical engineering and need help from professionals in case of problems with vehicle. Similarly, many people know how "fast" their computer is in terms of CPU frequency clocking and how much "memory" (RAM) it has. But it does not make them computer engineers. Did you ever opened computer case and speculated how it works? Or even maybe, did you assemble your own PC computer from individual components? Did you ever wondered how the operating system interacts with the hardware? Maybe you tried to guess how user applications such as colorful computer games or scientific simulations make appearance on the screen? I must assume here that you have enough curiosity about the topic presented in this book.

I need to ask you to acknowledge that there is no guarantee that you will learn something from this book as from any particular book about programming. Effort to code or design computing systems is creative process and as such it is skill built mainly on practice. Reading any book will not replace practical efforts which have to be taken. However, you may need a guide if you are going on a tour in the brave new world of microprocessor engineering. Therefore I hope that this handbook will help you in the efforts you are going to make and support you in your training.

To sum up above paragraphs in brief I consider that we need:

- awareness – perception of the microprocessors role and importance in the modern world,
- curiosity – inquisitiveness is the best long-term support to tackle with steep learning curve,
- commitment – personal obligation beats any amount of learning by heart.

Possibly you are aware of the fact that libraries are full of books about computers, programming and electronics. I can recommend Jeff Dunteman “Assembly Language Step-by-Step: Programming with Linux” although it is for 32-bit architecture. Perhaps you prefer to browse on-line materials where even more information (but less structured) can be found.

If there are so many books and materials already then why should we have yet another volume on such topic? Exactly because there are so many available sources! For inexperienced person it is not that easy to decide to which extent specific book corresponds to a programming environment that is in use or how much it is related to specific processor architecture. This book was based on more than 10 year of academic experience in teaching Microprocessor Engineering and even more in programming with use of many different high and low-level languages including assemblers of MOS 6502, TMS320, Intel 8051, Intel x86, Intel x86_64 (AMD 64-bit extensions), and ARM Cortex family. It is oriented towards students at Faculty of Electrical Engineering at Warsaw University of Technology and courses that are held there but might be useful for any person interested in this topic.

The book is divided into several chapters. First is dedicated to basic mathematical background which is mainly on numeral systems and logic. Part two is a discussion on key components in modern computing machinery like processors and memory. Third chapter briefly analyzes wired transmission standards that are widely implemented in contemporary microcontrollers and used in electronic systems nowadays. Fourth chapter is introduction to programming environment in Linux operating system. Fifth chapter is core of the book as it contains discussion on practical aspects of programming in x86_64 architecture .

Why Linux-based approach? There are no restrictions often found in commercial or “academic” software where only “first dose” is free of charge and some day you or someone else will have to pay for it.

Linux is vital point in free and open source software ecosystem that grown up from GNU project and liberal licenses like GPL, BSD and MIT. You may run such software as you wish, for any purpose. You may study how such programs work, and change them so they do what you wish. It assumes that source code is always available. You may redistribute such software and share it with others. You may redistribute code modified with your changes so others can use them too .

I consider GNU/Linux as the best choice for systemic education at all levels, self-training purposes, scientific research, technical experimentation, reliable applications and software development. There are numerous tools for programming purposes available on Linux, it is easy to get them and they are free. Free as in “freedom” and also free as in “free beer” . Linux is an Unix-world offspring and “Unix was designed for software development from day one, and it shows” [1].

Disclaimer

It is hardly possible to write technical textbook without references to existing products. Author's intention was to provide book with practical value hence many brands and companies are mentioned here. However, readers are asked to remember and understand that most products can be replaced with parts with similar functionality. Furthermore we endorse healthy competition in the industry which is not limited to major enterprises. There are many more companies on the market, which in spite of being smaller often specialize and provide products with notable features above the average. Good engineering practice is to analyze current market and seek for optimal solution.

All product names, trademarks and registered trademarks are property of their respective owners. All company, product and service names used in this book are for identification purposes only. Use of these names, trademarks and brands does not imply endorsement.

1. Numbers in Computing

Mathematics is the most beautiful and most powerful creation of the human spirit.
Stefan Banach

Computer processors might be perceived as extremely efficient "number crunchers" which means that they process enormous streams of information expressed only with numbers. It is worth for engineers to understand various ways by which electronic systems process these numbers.

Humans in almost every corner of the earth are used to decimal system. This is so easy and natural to count with fingers hence positional system based on powers of tens is the most common nowadays. It is well known in practice but revising it might be helpful in understanding other systems discussed here. Look at the example:

$$40375 = 4 \times 10^4 + 0 \times 10^3 + 3 \times 10^2 + 7 \times 10^1 + 5 \times 10^0 = 40000 + 0 + 300 + 70 + 5$$

In computer systems decimal numbers usually have no prefix or suffix but sometimes they are suffixed with "d": $40375d$.

Binary System

Computers do not use decimal system. They are working on *bits* where bit is unit of information. It can have only one of two possible but different values such as:

True	False
1	0
Day	Night

Binary system has only two digits: 0 and 1. It is also positional as decimal system, but the base equals 2 (that is number of digits). Binary numers sometimes are suffixed with letter "b" or prefixed with "0b". In the following example every multiplicand is expressed in decimal mode but it was omitted to simplify the equation and make it easier to follow the process:

$$0b1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 8d + 2d + 1d = 11d$$

Above example also shows how binary number is converted to decimal mode. The series of multiplicands (starting from the rightmost position) in this conversion is easy to understand, reproduce or even remember: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8129, 16384, 32768, 65536, ...

Conversion in from decimal mode to binary is a bit more tricky and there are several ways

to do it. First approach is repetitive division by 2 with remainder that indicates value of bit at specific position. Example:

$$\begin{aligned} 11d &= 2 \times 5d + 1 \\ 5d &= 2 \times 2d + 1 \\ 2d &= 2 \times 1d + 0 \\ 1d &= 2 \times 0d + 1 \end{aligned}$$

If we write remainders starting from the last one we will get binary value: **1011**b.

The above method works well for any value. But for small values simpler method might be employed which is based on subtraction. However, it requires to know (e.g. remember) weights at different digit positions. In every step a biggest possible weight is subtracted from the number that is being converted provided that result will remain positive number. At positions with corresponding weights "ones" are set leaving all other positions with "zeros". Example:

$$\begin{aligned} 11d - 8d &= 3d && \text{8d subtracted successfully so there should be ``1''} \\ 3d - 0d &= 3d && \text{4d cannot be subtracted within positive integers so ``0''} \\ 3d - 2d &= 1d && \text{2d subtracted successfully so there should be ``1''} \\ 1d - 1d &= 0d && \text{1d left so there should be ``1'' in the last position} \end{aligned}$$

This method may be much faster than previous one for small numbers. There is no threshold given how "small" the number should be as it depends on experience of the person that is doing the conversion. Range of one nibble (4 bits) is easy to achieve while full byte (8 bits) is little harder, although even 12-bit values are possible to be converted by human being without neither calculator or pen and paper at hand.

Binary system is foundation for boolean algebra. In typical computer implementations there are usually four basic operations available as presented in the table [1.1]. It is worth to remember that XOR and NOT operations are reversible. XOR is reversible because $\forall x : x \oplus x = 0$ then $x \oplus y \oplus y = x$. NOT operation is reversible because doubling negation cancels it so $\neg\neg x = x$.

Sometimes these four operations are displayed on truth tables that let easily and quickly find out the result of operation assuming known values of two arguments: x and y. They are presented in tables [1.2], [1.3], [1.4], [1.5].

In computer logic operations usually are not done on single bits but whole bytes, words and so on. In such case bits are compared pairwise and the result has length equal to number of bits compared.

Table 1.1: Four basic boolean operations.

Name	Operation	Symbol		
Conjunction	AND	$\wedge$	$x \wedge y = 1$ if $x = y = 1$	otherwise $x \wedge y = 0$
Disjunction	OR	$\vee$	$x \vee y = 0$ if $x = y = 0$	otherwise $x \vee y = 1$
Exclusive or	XOR	$\oplus$	$x \oplus y = 1$ if $x \neq y$	otherwise $x \oplus y = 0$
Negation	NOT	$\neg$	$\neg x = 1$ if $x = 0$	otherwise $\neg x = 0$

Table 1.2: Conjunction truth table.

AND $x \wedge y$	**0**	**1**
0	0	0
1	0	1

Table 1.3: Disjunction truth table.

OR $x \vee y$	**0**	1
0	0	1
1	1	1

Table 1.4: Exclusive disjunction truth table.

XOR $x \oplus y$	**0**	**1**
0	0	1
1	1	0

Table 1.5: Negation truth table.

NEG $\neg x$	**0**	**1**
	1	0

Hexadecimal Numbers

Hexadecimal mode is also positional mode so it works exactly the same way as decimal and binary modes. The base in hexadecimal mode equals to 16 so 16 digits are necessary. First five letters from latin alphabet (A-F) expand standard decimal digits 0-9 making altogether 16 digits.

Capital and small letters might be used but should not be mixed in single number. In the table below they are presented with their decimal values:

Hexadecimal number		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**...**
Decimal value		0	1	2	3	4	5	6	7	...
Hexadecimal number	...	8	9	A	B	C	D	E	F	
Decimal value	...	8	9	10	11	12	13	14	15	

Hexadecimal values are very useful and popular representation of integer numbers, memory addresses and other numbers in programming and computing machinery. One and only one of several suffixes and prefixes should be used to indicate that given number is hexadecimal value. Sometimes suffixes and prefixes are not present as it might be assumed or it is known that given number is hexadecimal. Examples:

- **0x**1245 – this is hexadecimal number prefixed with "0x" despite no letter is present in the number
- 3f78**h** – this is hexadecimal number as it is suffixed with "h" letter at the end
- **$**68000 – dollar sign was used to prefix hexadecimal values on some systems but it is rarely used nowadays as it might be confused with other meanings of the dollar sign in source code
- C0DE – we may guess it is hexadecimal value because typical digits–letters are present

Conversion from decimal mode to hexadecimal might be done in the same way as it is done between decimal and binary. In the following example a decimal value 40375 will be converted to its hexadecimal representation:

$$\begin{aligned} 40375 &= 16 \times 2523 + 7 \\ 2523 &= 16 \times 157 + 11 \\ 157 &= 16 \times 9 + 13 \\ 9 &= 16 \times 0 + 9 \end{aligned}$$

Remainders are as follows: 7, 11, 13, 9 or when represented as hexadecimal digits they are: 7, B, D, 9. Therefore 40375d = 0x9db7.

Now we try method based on subtractions knowing that $16^0 = 1, 16^2 = 256, 16^3 = 4096$. In every step we will look for the biggest value that can be subtracted that is also a product of the base mentioned earlier and term that is lesser or equal to 15 (0xf).

$$\begin{aligned} 40375 - (9 * 4096) &= 3511 \\ 3511 - (13 * 256) &= 183 \\ 183 - (11 * 16) &= 7 \\ 7d - (7 * 1) &= 0 \end{aligned}$$

Integers

The simplest, unsigned integers are simple binary representation of specific value just as it was discussed in section 1.1. Number of possible values and their ranges depends on bits used to represent the number. Characteristic data formats and their corresponding maximum unsigned numbers are shown in table 1.6. Limits of addressable space are theoretical maximums and do not consider caps such as: physical limitations of actual memory modules, hardware limitations of computer mainboard and software implementation introduced by operating systems.

Table 1.6: Unsigned integers and their limits in typical data formats.

Number of bits	Maximum unsigned	Remarks
1	1	Bit
4	15	Half-byte (nibble)
8	255	Byte
10	1023	Typical in ADC
12	4095	Typical in ADC
16	65535	Word; 64 kB addressable
32	4294967295	Double-word; 4 GB addressable
42	$2^{42} - 1$	x86_64 Intel CPU addressable space – 4 TB
48	$2^{48} - 1$	x86_64 AMD CPU addressable space – 256 TB
64	$2^{64} - 1$	Quad-word; 16 PB addressable

Signed-magnitude Integers

One bit may be designated to store information about the sign of value. If there is 0 on that bit then value is considered to be positive, otherwise it is negative. Limits of several exemplary data formats are given in table 1.7.

Table 1.7: Signed-magnitude integers and their limits in typical data formats.

Number of bits	Limits	Remarks
4	± 7	Half-byte (nibble)
8	± 127	Byte
10	± 511	Typical in ADC
12	± 2047	Typical in ADC
16	± 32767	Word
32	± 2147483647	Double-word; ± 2 billions
64	$2^{64-1} - 1$	Quad-word; ± 9 quintillions

With signed-magnitude integers there is significant problem of doubled zero as there are two possible values that represent "positive" (000..00000) and "negative" (1000..000) zeros. Sum of these two zeros results in "negative" zero.

Furthermore sum of positive and negative values that have their absolutes equal is not that straightforward. For example lets sum signed-magnitude values of -5 and +5. Assuming that they are stored in 4-bit registers their representations are: 1101 and 0101, where first (most significant) bit represents sign. Direct sum in bit pairwise order starting at the least significant results in 0010 which is $+2$ that is not the correct answer.

With signed-magnitude one needs to take sign bit into consideration first. If the bit is the same in both values then they should be added and their sign bit copied. If they have different sign bit then they should be subtracted and bit is copied from the absolutely larger value. This approach is complex and slow thus was superseded by one's complement and two's complement representations.

One's Complement and Two's Complement

In one's complement to represent negative number its positive (absolute) binary representation is bitwise inverted with NOT operation. Therefore negative numbers in one's complement can be recognized by 1 in the most significant bit. So $+5$ from previous example is still 0101 but -5 is 1010. Summing them up results in 1111 that is one of two possible representations of zero. The zero in one's complement is represented by either 0 on all bits so that it is 0^+ or 1 on all bits so it is 0^-.

Problem of doubled zero is finally solved in two's complement. In two's complement zero has no sign because it is represented only by all bits set to 0.

To convert negative (and only negative!) value from binary to two's complement one has to simply add one to one's complement of the value. Therefore $+5$ is still 0101 in two's complement. However, -5 = 1010 in one's complement so it is 1011 in two's complement. If we add these two two's complement values: 0101 and 1011 the result is **1**0000 so it is lengthier than

original but in the basis all bits will be equal to zero. This additional bit might be indication of the result equal to zero.

To convert two's complement that is negative, so one which has the most significant bit set, back to binary representation both previous operations must be performed but in reversed order. So it is necessary to subtract one and invert all bits. For example all "ones" (11..11) is a negative value which after subtraction becomes (11..110), and after inversion it is 00..001 so we conclude that it represented -1 in decimal.

Further examples are given in table 1.8. Interesting fact about two's complement is that this format can represent less positive values than negative values by one.

Two's complement might look complicated on paper but is not an issue for computing machinery. It is an ultimate solution to all problems discussed above so it is commonly implemented in contemporary processors.

Table 1.8: Some exemplary two's complement values on 8-bit register.

Value	Two's complement
+127	01111111
+32	00100000
+15	00001111
+3	00000011
+2	00000010
+1	00000001
0	00000000
-1	11111111
-2	11111110
-3	11111101
-15	11110001
-32	11100000
-127	10000001
-128	10000000

Fixed-point and Floating-point Numbers

Electronic circuitry and processors are capable of processing integer numbers quite efficiently. However, real numbers are a problem due to their possible high number of decimal places. The higher number of decimal places the larger is the precision with which the value needs to be stored and so more information it possesses. With increasing precision more bits are needed to store the real value.

There are two approaches to store real numbers in computing machinery: fixed-point and

floating-point. First one has limited precision, an arbitrary number of decimal places and so its resolution is limited. Second method provides much higher resolutions and wider range of stored values but introduces an error as it is the best possible approximation with given number of bits used to store the floating point number.

Fixed-point numbers

Fixed-point number consists of two parts: integer and fractional. With given size of register some bits are used to store integer part and remaining ones are used to store fractional part. Integer part is stored as it was described in section 1.3. Bits in fractional part have base 2 but their exponents are negative. So there is $2^{-1} = \frac{1}{2}$, $2^{-2} = \frac{1}{4}$, $2^{-3} = \frac{1}{8}$. The series ends on $2^{-n} = \frac{1}{2^n}$, where n cannot be higher than number of bits in used register. Powers of 2 with exponent down to -8 are given in table 1.9.

Table 1.9: Start of series of fractional parts in fixed-point mode.

n	x^n
-1	0.5
-2	0.25
-3	0.125
-4	0.0625
-5	0.03125
-6	0.015625
-7	0.0078125
-8	0.00390625

Fixed-point is an arbitrary data storage mode as programmers implementing it may choose how many bits should be used per each part. Example below presents fixed-point storage on 8-bit register with 3 bits used to store integer part and 5 to store fractional part. Number that we would like to store in this "3.5" format is 5.789306640625.

Converting decimal 5 to binary results in 101 and these bits will occupy upper part of the register (more significant bits). Now do repetitive subtractions, if they are possible. If subtraction was possible then corresponding fractional bit should be set.

$$
\begin{aligned}
0.789306640625 - 0.5 &= 0.289306640625 &\Rightarrow 1 \\
0.289306640625 - 0.25 &= 0.039306640625 &\Rightarrow 1 \\
0.039306640625 - 0.125 < 0 &\Rightarrow 0 \\
0.039306640625 - 0.0625 < 0 &\Rightarrow 0 \\
0.039306640625 - 0.03125 &= 0.008056640625 &\Rightarrow 1
\end{aligned}
$$

So decimal $5.789306640625 = 10111001_{3.5}$. Remainder equals to 0.008056640625 and this is the error introduced due to conversion.

Conversion back is more straightforward: $1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-5} = 5.78125$. Difference between original value and recovered one equals to remainder that we had to abandon during conversion from decimal to fixed-point.

Floating-point numbers

Floating-point number can represent highly varying range of values both very large and very small. Contemporary processors provide implementation of IEEE standard that is well established description of floating-point number formats.

Conversion between decimal value and its IEEE 754 representation is given by equation: $L_{10} = (-1)^s \, (1 + m) \, b^{e-x}$

where:

- L_{10} – decimal value
- s – one bit that represents sign
- m – mantissa that is fractional part
- b – base (radix) that in computer machinery usually is 2
- e – exponent
- x – maximum possible value of exponent (exponent bias) $x = 2^k - 2$, where k is number of exponent bits.

Range and precision depend on the number of bits alloted to mantissa and exponent. Some examples are shown in table 1.10. Effective, calculated exponent value E have to be within ranges presented in table 1.11.

Table 1.10: Some IEEE 754 modes with base 2.

Name	Bits total	Mantissa bits	Exponent bits
Half precision	16	10	5
Single precision	32	23	8
Double precision	64	52	11

.

Table 1.11: Exponent limits in popular IEEE 754 formats.

Name	e_{max}	E_{min}	$E_{max} = x$
Half precision	30	-14	+15
Single precision	254	-126	+127
Double precision	2046	-1022	+1023

Examples in half precision:

$$0\ 01111\ 0000000001 = (-1)^0 \times 2^{15-15} \times (1 + 2^{-10}) = 1.0009765625$$
$$0\ 00001\ 1000000001 = (-1)^0 \times 2^{1-15} \times (1 + 2^{-1} + 2^{-10}) = 0.0000916123$$
$$1\ 10001\ 1000000000 = (-1)^1 \times 2^{17-15} \times (1 + 2^{-1}) = -6.0$$

Examples in single precision:

$$0\ 01111111\ 01000000000000000000000 = 2^{127-127} \times (1 + 2^{-2}) = 1.25$$
$$0\ 10000010\ 11000000000000000000000 = 2^{130-127} \times (1 + 2^{-1} + 2^{-2}) = 14.0$$
$$1\ 00000001\ 00000000000000000000000 = 2^{1-127} \times 1 = -1.1755e-38$$
$$1\ 11111110\ 11111111111111111111111 \approx 2^{254-127} \times 2 = -3.40282e+38$$

If all bits of exponent are set to zero then it is a category of denormalized values. In that situation exponent used in computation has E_{min} value as given by table 1.11. Mantissa takes form of 0.m instead of regular 1.m hence there is no addition of one to the mantissa. Denormalized values are used to store extremely small numbers. Example of denormalized number:

$$0\ 00000000\ 00000000000000000000001 = 2^{-126} \times (0 + 2^{-23}) = 1e-45$$

Some other compositions of bits also have special meaning:

- +0 – all bits set to zero
- -0 – all bits set to zero except of sign bit which is set to one
- +Infinity – all exponent bits set to one, e.g.: 0 11111111 00000000000000000000000
- -Infinity – all exponent bits set to one, e.g.: 1 11111111 00000000000000000000000
- NaN – “Not a number” so result of operation such as division by zero is indicated by sign bit set to one, all exponent bits set to one, and at least one of mantissa bits set to one; e.g.: 1 11111111 00000000000000000000001

Endianness

Data of any kind in computer memory must be stored consistently and orderly. This order is known as endianness. Non-technical people also use endianness when they want to save date, which may have many formats, as for example:

- 13$^{\text{th}}$ of December, 1981
- December 13, 1981

- 1981-12-13

Computer memory might be imagined as a series of data cells. Some microcontrollers enable to access single bits but usually bytes are the smallest units available. Second order data units like words, double-words, quad-words consist of more than one byte and these bytes are stored in computer memory in specific order.

When big-endian is used then with increasing addresses bytes are stored starting from the least significant one (representing smaller values). In little endian it is otherwise so when memory is read byte by byte then data looks like bytes have reversed order. Little endian system is used in x86_64 architecture. These two endiannesses are shown on figure 1.1.

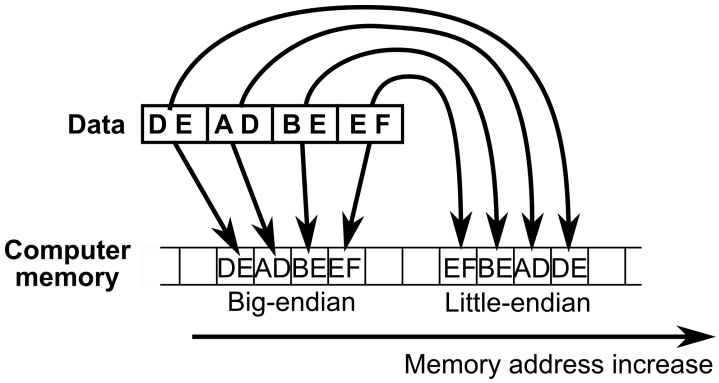

Figure 1.1 Comparison of big-endian and little-endian with one byte unit.

It is also possible to have different unit than single byte. In such case big-endian storage will have no difference to one shown in figure 1.1. However, if unit is for example word (two bytes) and 0xd1cec0de is to be stored then in memory it will be present as: 0xc0de, 0xd1ce. If the unit were one byte it would be stored as: 0xde, 0xc0, 0xce, 0xd1.

2. Electronic Components

Every time you turn on your new car, you're turning on 20 microprocessors.
ScottMcNealy

In computing machinery there are two distinctive components: processors and memory. Instructions and data are held in memory to which the processor has access. Processor retrieves instructions from memory and executes them. Some instructions work on data that is also obtained from memory. Results are temporarily stored in processor "registers" and then transmitted to memory or some other circuitry. In this section various types of processors and computer memory types will be discussed. Methods of data transmission between chips will be also briefly analyzed.

Processors

Computational processor is highly integrated circuit (**IC**). Integrated circuits with more than 1 million of transistors are categorized as Ultra-Large-Scale-of-Integration (ULSI) . Modern microprocessor is an example of such integrated circuit. It is a versatile but highly optimized and highly integrated chip used as a main component in Personal Computers (**PC**).

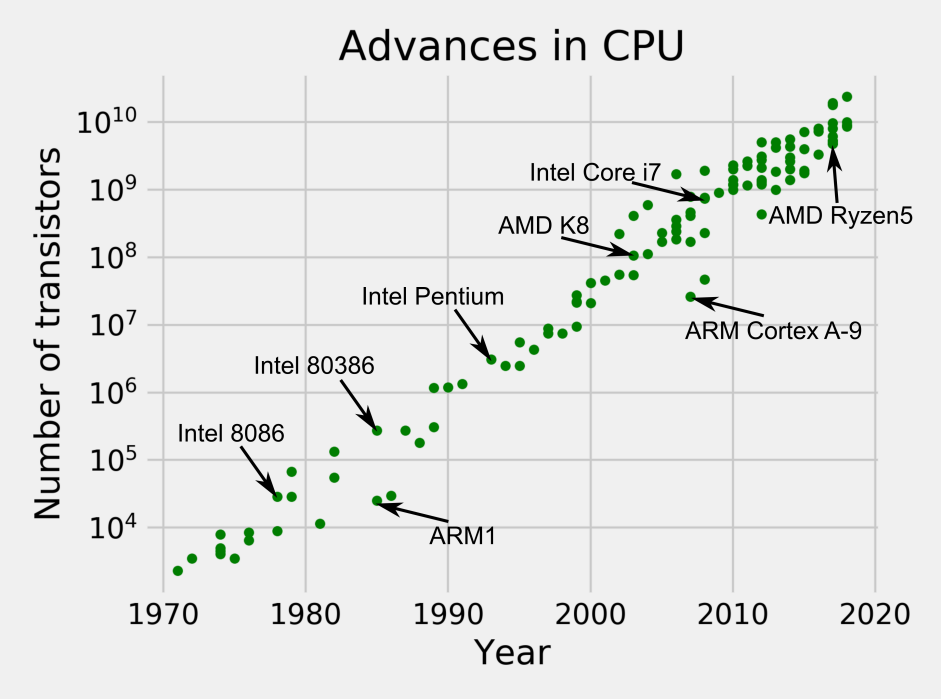

Figure 2.1 Every two years number of transistors in CPU doubles according to empirical Moore's Law.

Core computational block of processor is known as **Arithmetic-Logic Unit** (**ALU**). This block performs basic mathematical and logical operations on data that is usually provided from outside by a connection known as memory bus.

Processor cannot work on its own but only in interaction with external circuits and electronic elements. These external parts that cooperate with processor may act as temporary or permanent data storage, sensors, actuators and other chips.

Primary processor in the computer is easy to recognize and due to its complexity it is called **Central Processing Unit** or simply **CPU**. However, in single computer there are many specialized processors with different features and oriented for specific tasks. They work in parallel and only synchronize with other chips occasionally. For example there might be a chip that is controlling connection over Ethernet card and another one solely for Universal Serial Bus (USB) management. Every data storage unit such as hard disk, solid state disk, memory card reader, optical disk reader contains its own processor or processors.

When many processors of the same type are coupled and work together in parallel to perform some heavy computations they constitute a **multiprocessor system**. Typically such system is designed for heavy computational tasks and may be labeled as supercomputer. However, when supercomputers are discussed it is worth to remember the Moore's Law. It states that every year computational capability of processors is doubled. Therefore supercomputers from XXth century are comparable with todays smartwatches in terms of their computing efficiency. This development is shown in figure 2.1.

Over the years a continuous progress in scale of integration was observed so current CPUs are multicore processors as they have many ALUs on single chip. It might be also observed that typical CPUs with multicore architecture resemble multiprocessor systems in terms of their functionality.

Due to variety of applications and requirements of specific system we may distinguish different types of processors such as general purpose microprocessors, microcontrollers, video or audio processing accelerators, microcontrollers, and others. Now we will discuss details and distinctive features of these components and analyze some examples of their applications.

Microprocessor

This book is based on series of microprocessors designed by Intel that constitute **x86** family. It is a long list of backward-compatible processors for personal computers that started with Intel 8086 in year 1978.

Other companies such as AMD also manufacture processors compatible with x86 architecture. They exchange know-how, intellectual property (IP) and patents. Chips share common set of instructions, register names and other architectural features but they differ internally as every instruction is implemented inside the microprocessor in its internal *microcode*. This approach enables competitors to tackle encountered problems independently

so it stimulates progress. Thanks to common instruction set programs can be started immediately on other machine, with CPU from other manufacturer. For example lets take a stand-alone program that means it is not dependent on external libraries or operating system functions. If the program was compiled against 80386 CPU it can be started on original 386 machine, on more advanced Intel Pentium and on modern AMD Ryzen. Obviously an application compiled with the use of modern features may not work properly or even will not start on older machine.

32-bit machines that started with 80386 and lasted till Pentium 4 were developed between years 1985 – 2000. These generation of processors introduced many important features such as: memory protection and context switching (enabling multitasked operating systems), embedded coprocessor (also known as Floating Point Unit – **FPU**), internal cache memory, vector instructions (for parallel compuatational processing of large data sets) and pipelining (clever processing of many instructions at once). The biggest problem with these processors was their limited addressing space. They were able to access 4 GB of RAM as $2^{32} = 4294967296$. In practice it was not possible to have that much memory available and operating systems limited addressable memory space to 3 GB or less even though more was installed physically. Example of late 32-bit processor based on x86 architecture is shown in figure 2.2.

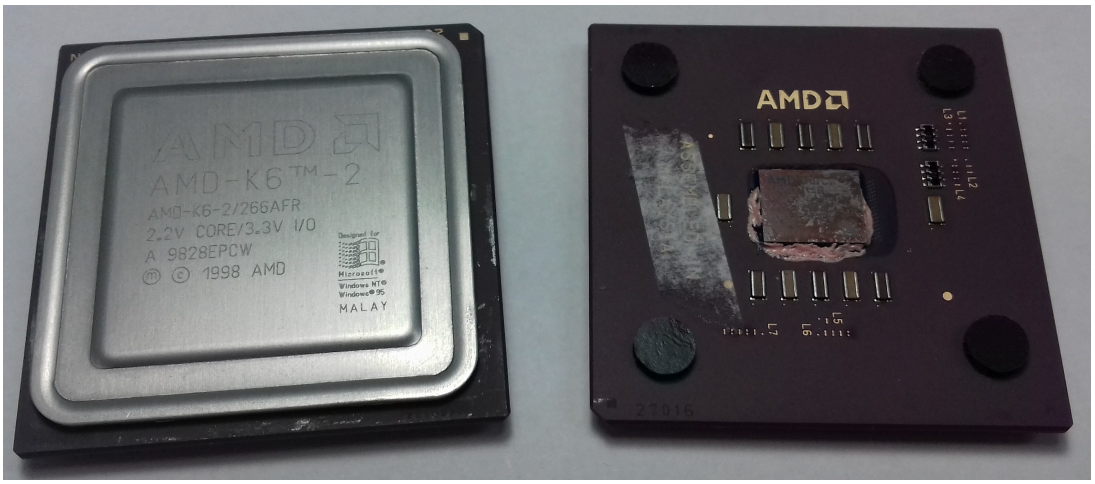

Figure 2.2: Exemplary x86 processors from the turn of XXth century with protective metal shield (AMD K6) and opened with the core exposed (AMD Duron).

Limitations of x86 microarchitecture became obvious at the end of XX^{th} century. Therefore there was a major breakthrough that was switch from 32-bit to 64-bit architecture. AMD was the first company that introduced 64-bit to x86 CPU with their AMD64 extension. 64-bit architecture means that basic CPU registers have size of 64 bits. With larger registers it becomes possible to:

- work on bigger integer numbers (in absolute terms),
- work with higer precision real numbers,

- address more memory.

Another architectural breakthrough was the consolidation of many cores in single processor. Every core might have capability of *hyperthreading* so that is works as even more (usually two) traditional processors. Therefore modern CPU can be viewed as many parallel CPUs and so it is presented in figure 2.3.

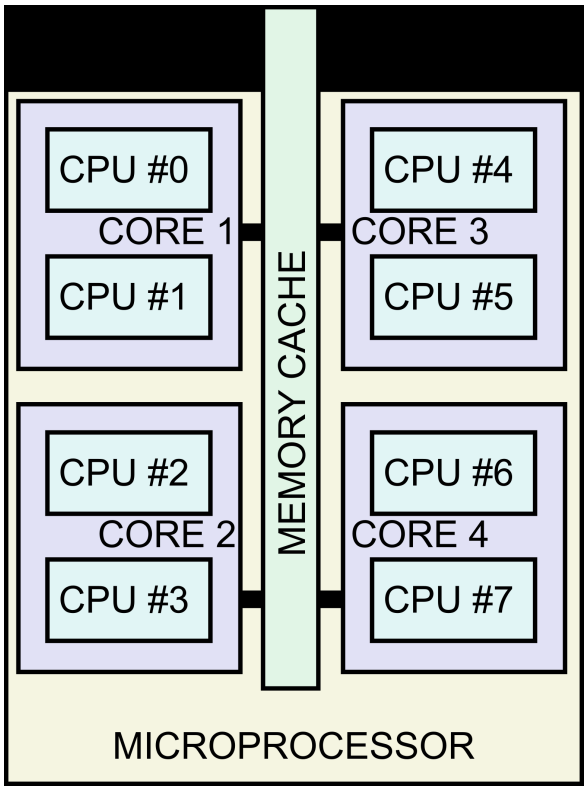

Figure 2.3 Simplified block diagram of modern multicore microprocessor.

To observe it on Linux operating system user may see the contents of special, textual file `/proc/cpuinfo` where interesting information about system microprocessor can be found. In this file every hyperthreaded core is counted as "processor" starting from 0. Usually every of these processors has the same description. In this file information about processor manufacturer ("vendor"), its model name, frequency, cache size and very detailed information coded in "flags" can be found as well. Another way to find number of CPU is to start a command line program `nproc` which simply prints number of CPUs available. More advanced and detailed information about PC can be retrieved with `dmidecode` program that also is started from command line and presents its output in simple, textual form. Program might reveal sensitive information about the machine therefore it requires root privileges to start. Similar features and programs should be available in other operating systems.

Microcontroller

Microcontrollers (**MCU**) share some features with microprocessors but their design goals and purposes are completely different. Microcontroller should have basic components that we already discussed with microprocessor: ALU and bus or buses by which it retrieves data from memory. But here starts the main difference as microprocessor internal memory is just a cache that speeds up instruction processing. In microcontroller there is internal memory that is the main memory for the device. There might be external memory of some kind although usually microcontroller-based system relies only on internal memory of MCU. This internal memory has distinctive part in which the program and constant data is stored and another one for the data that is being processed. It is unlike microprocessors where the external memory is populated both with program to execute and data which is processed by the program.

This is the key difference between microcontrollers and microprocessors that originates in two types of architecture. First type, known as von Neumann architecture has single type of memory unit and so it uses only one data bus. Even if the data bus has separate lines to transfer memory address, to control the transmission (e.g. clocking) and the data itself it is single data bus. Second type of architecture has two memory types, one for data and another for the program. Therefore it needs to have two data buses with all necessary lines. This is known as Harvard architecture. Both types are compared in figure 2.4.

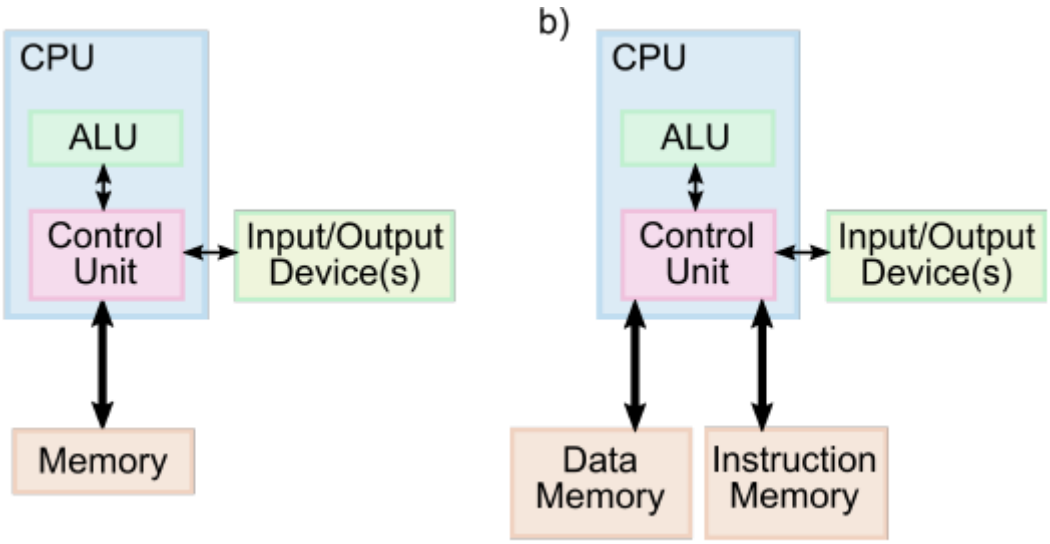

Figure 2.4: Processor architectures: a) von Neumann, b) Harvard.

Von Neumann architecture is typical for microprocessor-based systems and it is optimized for large non-volatile memory units. Design is simplified and the system is more versatile. On the other hand Harvard architecture to which microcontrollers incline is optimized for speed as both instructions of the program and data can be retrieved at the same time. Nowadays the

borderline is not that sharp as both microcontrollers and microprocessors architectures are more complex and advanced often benefiting from both approaches.

Microcontrollers are used in many electronic and embedded devices. Due to large quantity in which they are used the manufacturers design and sell series of devices that share some general set of features but differ in details such as program storage memory size (EEPROM), data memory (RAM) and package (chip physical shape and number of connections). To make this landscape even more complicated we should observe that microcontrollers might have numerous additional features such as:

- analog-to-digital converters (ADC),
- digital-to-analog converters (DAC),
- comparators,
- PWM signal generators (for motor control),
- timers,
- general purpose input-output ports (GPIO),
- communication control (e.g.: SPI, I2C, UART),
- display control.

Not all microcontrollers provide all of these features and some might be specialized in aspects not mentioned above. They also normally differ in number of lines (connections) providing specific feature. For example one type may have 2 ADC lines while the other from the series might have 3 ADC lines. Same chip might be packaged with for example 64 pins, or 100 pins, or 144 pins and so on, thus making it significant difference in number of GPIO lines and other lines available at the same time. The choice of individual microcontroller is therefore often considered to be very troublesome process with significant consequences for design of electronic device.

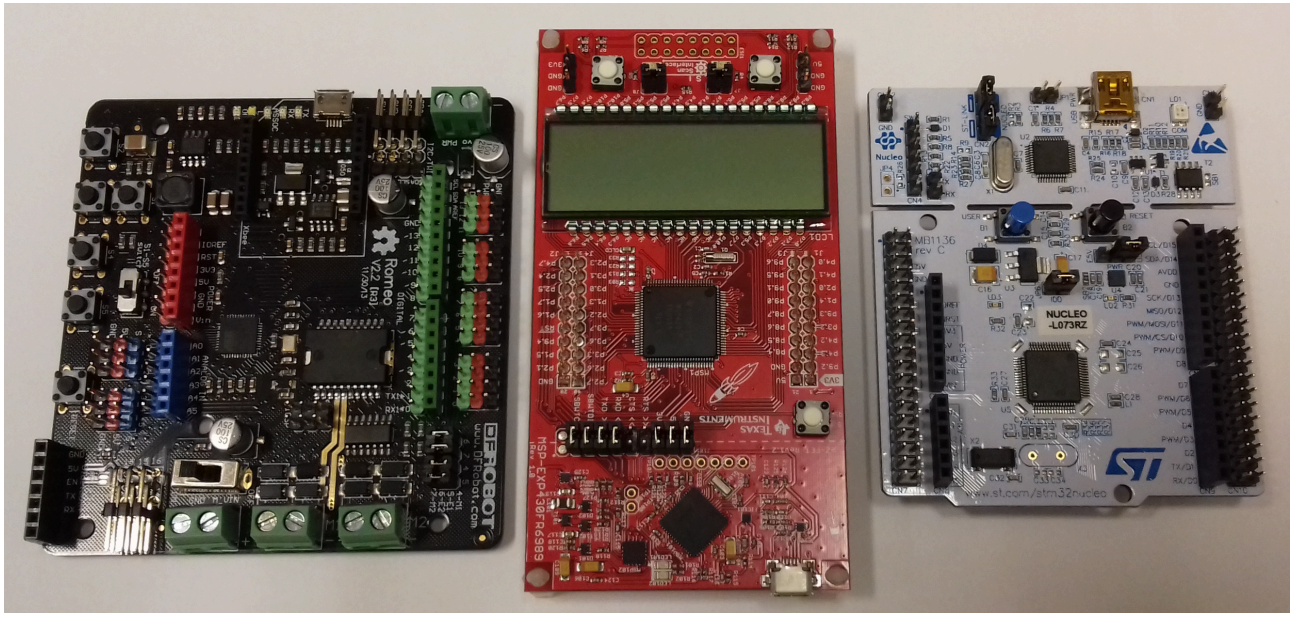

Figure 2.5: Various development boards with different processor architectures. From left to right: 8-bit ATmega32u4 by Microchip (clone of "Arduino"), 16-bit MSP430FR6989 from Texas Instruments ("Launchpad"), 32-bit STM32L073RZ from STMicroelectronics ("Nucleo").

Microcontrollers that are available in the market come with variety of "bitness" so there are 8-bit, 16-bit, 32-bit and 64-bit microcontrollers to choose from. These different architectures are shown on photograph in figure 2.5.

Not always the choice of more bits is better as price is another factor that is correlated with "bitness". For example application of 8-bit microcontroller is good enough to control simple line of Christmas lights. Popular Arduino Uno board is based on Microchip ATmega328 microcontroller that is 8-bit chip. Example of 16-bit microcontroller family are Texas Instruments MSP430 chips which can be found on inexpensive "Launchpad" development boards. On the other hand it might be necessary to use 32-bit or even 64-bit microcontroller with FPU in sophisticated measurement device. In such case one may look for chips with ARM core such as STMicroelectronics STM32 series. They are commercially available for development purposes on inexpensive boards named Nucleo or Discovery.

Specialized processors

Microprocessors are essential part of desktop computers and laptops. Microcontrollers are common in handheld devices such as mobile phones, e-book readers, remote TV controllers and numerous embedded systems. Their versatility makes them primary choice for countless electronic products. However, there are situations where it is worth to sacrifice flexibility and gain efficiency to solve one particular task. For that purposes there are specialized processors such as Graphics Processing Unit (GPU) or Digital Signal Processors (DSP) and more which

can perform just one type of task but much more efficiently than microprocessor or microcontroller could do.

Graphics Processing Unit (GPU)

GPU was formerly called "graphics accelerator" as this chip accelerates calculations performed in graphically demanding applications such as computer games or high-quality video display. Such chip might be analysed as powerful link between computer screen and CPU and computer memory.

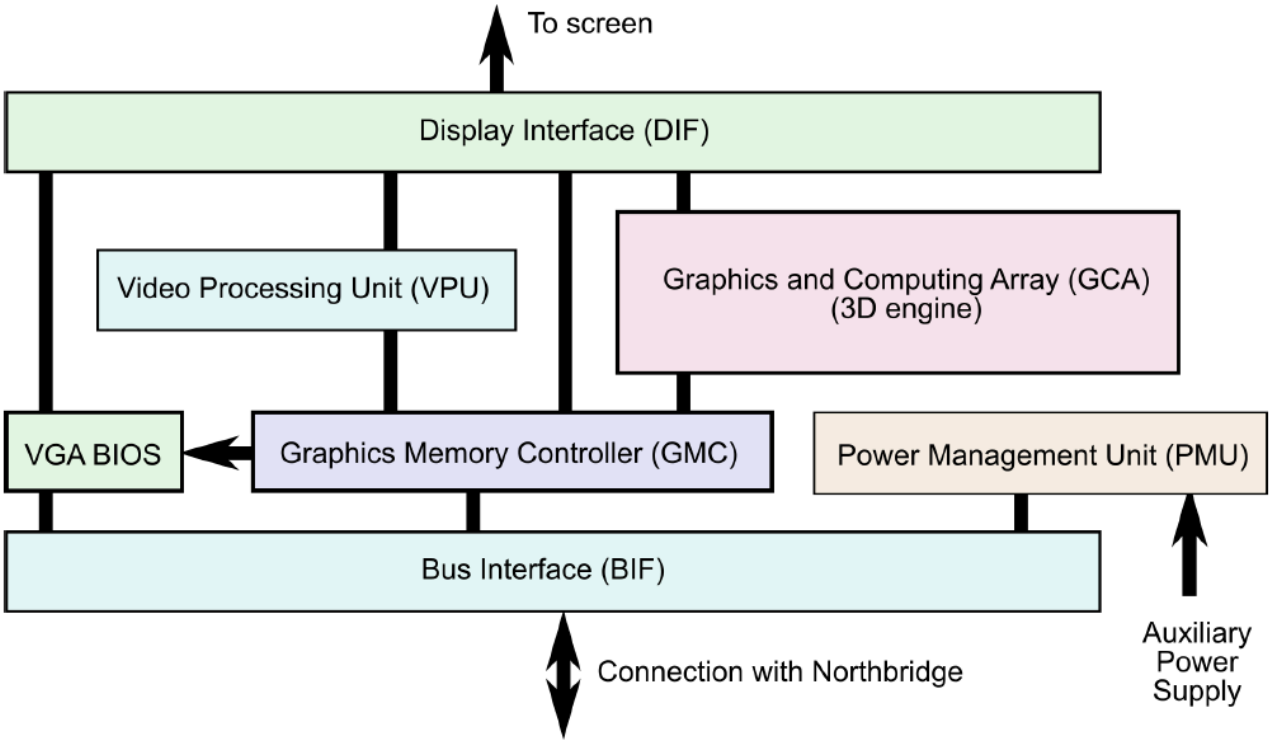

Figure 2.6: Block diagram of computer graphics card with GPU.

Now we will analyze computer video card based on GPU. Its block diagram is shown in figure 2.6. On one side it has Display Interface (DIF) unit which can generate video signals in specific standards (VGA, HDMI, DisplayPort and so on). When high resolution screens (HDTV, 4k, 6k) are in use then there is more data (more pixels) to be pushed through this channel so this link must be reliable (often synchronous) and efficient. Data can be provided to DIF from several blocks that are present in the GPU chip such as:

- generic Graphics Memory Controller (GMC) – simple graphics possibly with direct access to computer RAM but without advanced acceleration,
- Video Processing Unit (VPU) – part of GPU with optimized and implemented in hardware *media codecs* such as MPEG2, H.264, Theora, VP8 and so on,

- Graphics and Compute Array (GCA) – 3D engine that provides acceleration to: geometry calculations, rendering, texture mapping, *shaders*, more specialized graphical effects (antialiasing, SSAO, bloom, and other post-processing effects) and realistic physics of rendered objects.

Furthermore GPU are backward-compatible with older graphic controllers and provide primitive VGA BIOS that can be observed at work during computer boot up process. Finally there must be reliable Power Management Unit (PMU) as GPU cards require a lot of power – dozens of watts, that sometimes is more than the main CPU consumes. In such case often a separate and direct power connection is needed between GPU card and main power supply of the computer.

High efficiency of GPU would be wasted if it was not adequately associated with other components, particularly CPU and computer RAM memory. Therefore in modern PC the GPU is connected to CPU the same way the RAM is – through so called *Northbridge* that is very fast memory controller. Connection between Northbridge and GPU is known as Bus Interface. It is in the standard of PCI Express (**PCIe**) nowadays. On PCIe lines data transmission is clocked at very high frequency. For example version 4 of PCIe standard that was presented in 2017 allows throughput up to 16 Gbps per line that is about 2 GB/s per line. Graphic cards with GPU are usually connected over PCIe×16 which means that there are 16 lines working synchronously providing parallel transmissions so that it is possible to have throughput between GPU and RAM up to 64 GB/s. Version 5.0 of the PCIe standard is about to appear in 2019 and double these results.

Every year there are about 400 millions of GPU shipped to the market.

Digital Signal Processor (DSP)

In previous sections we focused on CPU and GPU because they are cornerstones for PC computers which are ubiquitous and so easily available for learning purposes. However, efficient processing of video data is not limited to GPU which in fact are relatively expensive solution. Similar but more versatile functionality might be gained with Digital Signal Processors. DSP have to deal with a lot of data in real-time so they are usually built in the Harvard architecture or heavily modified von Neumann architecture.

Number of 400 millions of GPU shipped to market every year might sound impressive but it is dwarfed by number of DSP shipped at the same time. The cause of that is simple: GPUs are used mainly in PC market while need for DSP is far more reaching. They are used in mobile phones, audio and video systems including TV screens, industrial machine vision, driver assistance systems, military equipment such as missile guiding systems and radars, and in many more applications.

Major companies active in the area of DSP processors are:

- Texas Instruments – manufacturing TMS320 series processors since year 1983,

- Analog Devices – manufacturing SHARC series processors since 1994 and Blackfin series processors since year 2000,
- Qualcomm – well known in the field of mobile phones where their core named Hexagon works in the background and is responsible for all "serious" tasks like hardware control including RF communication,

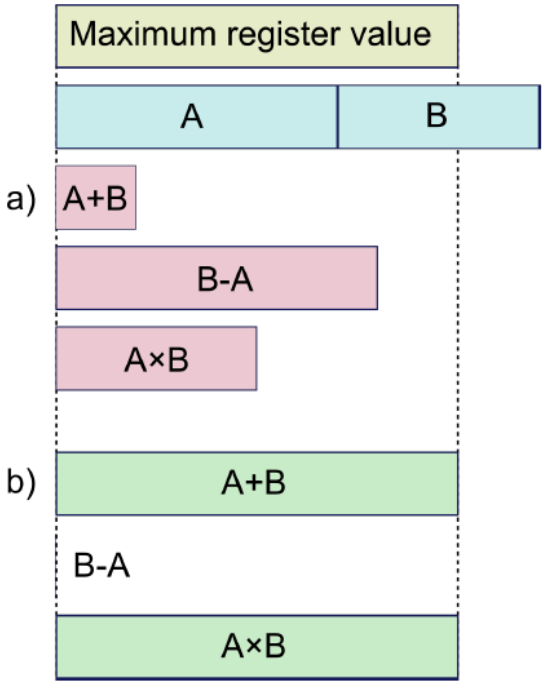

Figure 2.7: Graphical comparison of arithmetics: a) wraparound, b) saturated.

Characteristic feature of DSP processors is *saturated arithmetic*. First we should understand that any register in any type of microprocessor architecture is limited by how large or how small is the value that can be stored in it with given precision. Regular processors work in *wraparound* arithmetic in which if the result of mathematical operation exceeds these limits then a wraparound occurs. In that situation very big result might be stored as very small value and vice versa. If we would apply this approach to processing of music or video data results would be unacceptable distortions. On regular processor it is possible to proactively check by terms of software function if the result exceeds given limits and take action accordingly. But such procedure implemented in software would be hardly efficient and problematic for real-time processing of audio-video data stream in high resolution. Therefore DSP processors employ a hardware implementation of result control and if the result is about to exceed the limits then extreme value is taken as the result. Hence a wraparound never occurs.

Idea of saturated arithmetic is presented graphically in the figure 2.7. There is a register of some size and two values: A and B which are added, subtracted and multiplied. As you can see the results in group a) do not make much sense as they were obtained with wraparound

arithmetic. Section b) presents saturated arithmetic that seems to be more natural in processing of multimedia data.

System-on-Chip (SoC)

System on Chip is more complex structure than single core or multicore processor. This kind of chip integrates in one IC more cores that often are of different architectures. Despite their differences these cores are tightly coupled and each of them is doing specific task for which it was optimized to improve overall performance. Examples of such chips are:

- Texas Instruments DaVinci – used in industrial and surveillance cameras has ARM core associated with TMS320 core,
- Samsung Exynos – mostly for mobile phones or tablets includes ARM Cortex core possibly as secondary to a custom CPU, Mali GPU, Image Signal Processor (ISP) to control mobile phone camera, GSM modem, and GPS receiver,
- Qualcomm Snapdragon – includes many cores: ARM Cortex core, Hexagon DSP, Adreno GPU, ISP, GSM modem with RF frontend, short-range connectivity cores, GPS receiver, and battery charging controller,

Some of the cores existing in SoCs and listed above were discussed in previous sections while many others are specific for single purpose or task-related.

The biggest challenge in creating SoC is to efficiently interconnect all these cores. For that purpose a bus might be designed that is communication system for data transfer between components in computer architecture. Idea of such system is shown in figure 2.8.

Examples of bus architectures specific for SoC are:

- Advanced Microcontroller Bus Architecture (AMBA) is highly popular and royalty-free standard by ARM,
- Wishbone Bus by Silicore Corporation that is exceptional because it is free and open (not just royalty-free) hardware solution,
- Altera Avalon more advanced bus architecture with slave-side arbitration than enables the coexistence of many bus masters,
- CoreConnect by IBM (obsolete).

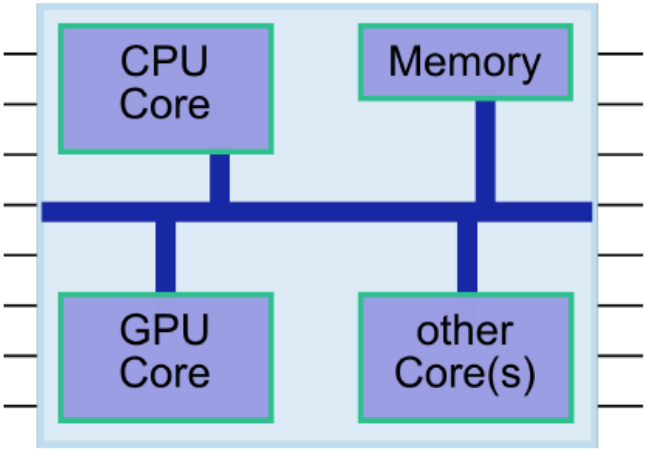

Figure 2.8: Block diagram of System-on-Chip architecture.

However, as there are more cores in single chip and many of these cores need to communicate with more than one another core (many-to-many) the interconnectivity requires more links between these chips. Therefore it becomes more challenging to outline adequate bus that is not occupying too much of chip physical area and that does not impact parasitic capacities. Therefore an alternative approach gained some popularity in recent years. This concept is named Network on Chip (NoC). In this approach cores in SoC communicate in network-like manner thanks to a routing controller that is another core embedded inside the chip. In this method cores are not connected to each other any more but through network managed by the router. However, to have positive outcome of applying NoC requires implementation of routing algorithms that have real-time performance to avoid bottlenecks and blocking (starving) of the on-chip network by cores that are more “greedy” in terms of communication.

Programmable Logic Processors

Processors discussed above might be powerful, efficient and popular in many applications. But they share one trait that sometimes is not desirable. They may be programmed and suit different functions but their internal structure is designed once and for all. In some applications it would be beneficial if the processor chip was more flexible.

Now we will discuss example of such situation in which it might be desirable or even necessary to change the internal structure of the chip after some time of its usage. Let us imagine that our company is designing and manufacturing mainboards for aircraft radar system. On the board there is specialized processor that implements in hardware a set of specific and sophisticated procedures of digital signal processing characteristic only for radar systems. Hardware implementation is necessary due to real-time and throughput constraints that could not be met if the solution was provided in purely software or hybrid manner. The

board must be reliable and robust so the company spent hefty sum of money to design and test it. Then it positively went through costly and time-consuming verification of compliance with governmental and military regulations. Our company won with competition and produced hundreds of such radars for military client. After some time a serious flaw was discovered in the processor. If it were just a regular processor then radars would be decommissioned and our company probably would go bankrupt. But our engineers were farsighted and used a programmable logic processor so we can change (reprogram) the *hardware structure* of the processor without even physical contact with the board or the processor itself.

Programmable logic processors are gaining popularity in mission-critical and safety-critical systems such as in automotive industry, defense industry, and space industry. They are also compelling way to verify new concepts in hardware design.

Field Programmable Gate Array (FPGA)

FPGA are logic chips that are intended to be programmable in the "field". The chip behavior may be changed also in remote location for example due to shift in equipment paradigm or modification of requirements that were not know during device deployment. Some FPGA have a feature which lets them to have part of them working while some other part is being updated. If we add redundancy to this then we have highly reliable yet adaptable solution for extreme requirements, like in military, telecommunications or space industry. Moreover FPGA provide and are considered as secure way to implement hardware.

FPGA are highly adaptable and may work in the place of CPU, MCU, GPU or other – more specialized processor. However, they are built as *gate arrays* therefore are more applicable to digital systems than analog systems.

FPGA chips provide low-latency signal propagation as they are naturally inclined for highly parallel mathematical operations. Therefore they are suitable for real-time data processing such as in experimental radio transceivers (e.g. Software Defined Radio, SDR), GSM Base Station Transceivers, video or radar systems.

Usually they may be supplied at low voltages such as 0,8 V ÷ 1,2 V. That is also quite desired feature in power-efficient applications where dozens or more of FPGA chips are orchestrated to work together.

FPGA internal structure is based on Configurable Logic Blocks (CLB). These blocks have many programmable internal connections. Another key idea are Look-Up Tables (LUTs) complemented with flip-flops and multiplexers. FPGA might be enhanced with non-programmable parts such as:

- internal memory blocks (volatile RAM),
- memory controllers for external memory management,
- Phase-Locked Loops (PLL) to generate highly stable high frequency signals such as clock signal,
- MCU core such as ARM Cortex for more ordinary operations.

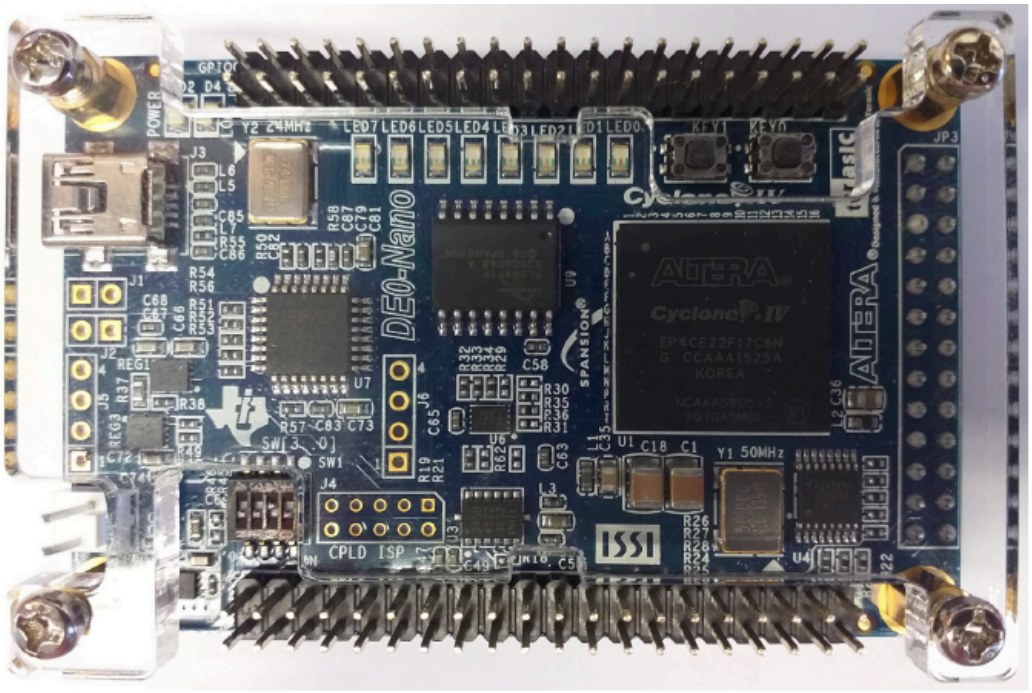

Figure 2.9: Exemplary FPGA development board with Altera chip.

Almost whole of FPGA market is occupied by four manufacturers: Xilinx which has more than half of the market share, Altera with about one-third, followed by Microsemi and Lattice . Similarly to MCU development boards there are also development boards for FPGA. Example of such board with Altera chip is shown in figure 2.9.

To program FPGA a Hardware Description Language (HDL) is used. There are two main flavors of HDL: VHDL and Verilog.

From the financial point of view FPGA have low value of NRE (Non-recurring engineering) therefore are easy to start development with them. But the cost of mass production devices using FPGA is steep as it increases proportionally to number of chips used. From this point of view there are better solutions. Therefore FPGA are often used for R&D and then design is reapplied to other chips such as Application Specific Integrated Circuit (ASIC).

Usually FPGA do not have ROM to store their program. Therefore their setup must be provided from external device or chip each time the FPGA is booted up. It might be a problem for a mature device that is massively distributed and not supposed to have significant changes. With such requirements perhaps another kind of programmable logic device should be taken into account.

Complex Programmable Logic Device (CPLD)

CPLD share many traits with FPGA, although they are fundamentally different. CPLD may have

tens of thousands programmable logic gates while FPGA may have millions of them. Therefore CPLD might be a cheaper solution to implement simpler circuits.

CPLD starts instantly after being boot up due to fact that it has internal ROM to store the program contrary to FPGA which needs to acquire its configuration and that causes delay in start time. Furthermore this downloading process which happens in FPGA-based circuits might be analyzed hence the device is more vulnerable to intellectual property infringement. Consequently CPLD are considered to be more secure than FPGA although the manufacturers of the latter kind of chips are working hard to provide safeguards.

On the other hand the feature of on-the-fly reprogramming available in many FPGA that was discussed in previous section is not available with CPLD. It might be reprogrammed but for that procedure it must be powered down. There are systems like aircraft radar which cannot be powered down yet might require occasional updates.

There are applications where one may choose between FPGA or CPLD. However, there is no restriction to use them both in the same product or even circuit board. For example CPLD might be engaged more during boot-up process to enable basic features of the device immediately after power up, and it may help to start FPGA, which will lauch more complex processing later on.

Memory

Two types of memory for computing machinery might be distinguished: *volatile* and *non-volatile*. First type requires to be constantly supplied with power to hold the data. Once the power is turned off such memory chip "forgets" all data very quickly. Therefore it is used for temporary storage of processed information that is obtained from source of information or generated (computed) by programs which run on processor. Now we will discuss volatile memory components and briefly analyze components that can store data in non-volatile way.

Volatile memory

This type of memory is more commonly known as random-access memory (**RAM**). There are two types of RAM: static and dynamic.

Static RAM (**SRAM**) is more expensive than dynamic RAM (**DRAM**) but also it is faster and is characterized with less power consumption. By "faster" we understand that time to retrieve and change the state of single memory cell is shorter – up to dozen of nanoseconds.

Thanks to these features SRAM is used inside more powerful microprocessors for the cache. Cache is build of several layers where lower levels are closer to CPU core. The closer to core the layer is the faster is access to its contents. On the other hand upper cache layers are larger and may reach size of several megabytes (MB). Lower layers of cache are per-core while upper

layers might be shared among two or more cores. Naturally new designs of microprocessors tend to have more and more cache memory thanks to improvements in manufacturing process.

When CPU is about to execute program instructions or process some data at first it is trying to get them from cache level one – L1. If the necessary data is not there then CPU is trying to find them in cache level two – L2, then level three – L3 and so on. Eventually the sought for code may or may be not found in the available cache layers so that CPU has to obtain the code and data from external memory that is usually of much slower DRAM type.

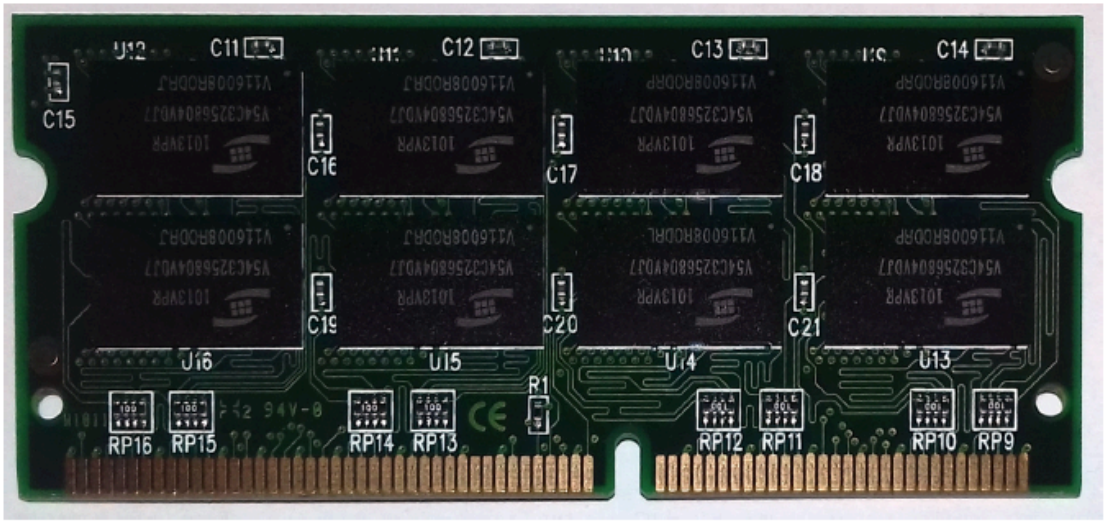

Figure 2.10: Small Outline Dual In-Line Memory Module (SO-DIMM).

The main trait of DRAM is that it is cheaper in production than SRAM. Moreover it is also "densier" so more data can be stored per physical area of chip. Therefore chips come in larger sizes measured in gigabytes (GB). Several chips form a small circuit board that is plugged into the computer mainboard. Photograph of such "memory stick" is shown in figure 2.10.

Programmable Memory

Non-volatile media storage discussed above are very common but unfortunately are not very efficient in terms of throughput. They have data transmission interfaces that are their bottlenecks. However, in electronics industry there is a need for non-volatile data storage that has as low latency as possible.

One-time Programmable ROM (PROM) was introduced in 1950s. The chip was programmed once with relatively high voltage – dozens of volts that changed the state of fuse built of semiconductor layers. This One-Time Programmable (OTP) trait is important even nowadays as it provides important security feature disabling tampering with the device after it was delivered to the customer.

Researchers at Intel investigated failures in PROM chips which lead them to conclusion that UV light might erase gate states. Thanks to that an idea of Erasable Programmable Read-Only Memory (EPROM) appeared and stayed very popular till the end of XXth century. These chips were often used to store basic embedded software known as BIOS in PC computers.

In 1970s Japanese national research institute presented Electrically Erasable Programmable Read-Only Memory (EEPROM) which simplified chip programming a lot. EEPROMs are based on floating-gate MOSFET transistors and making programming effort all-electric, without the need for complicated procedure with UV light, paved the way for modern MCUs. Actually modern electronics could not exist without EEPROM technology. EEPROM is a type of memory that can be rewritten electrically many times and preserves stored data even after power down. When chip is again powered up the stored code and data are available immediately.

As computer efficiency improved so did requirements for latency of EEPROMs increased over time also. In 1980s Toshiba answered these calls and presented flash memory that is popular nowadays in ubiquitous *USB flash drives*. There are two types of flash memory chips:

- based on NOR gate
- based on NAND gate

Table: 2.1: Comparison of NOR and NAND flash memory.

Feature	NOR flash	NAND flash
Cell size	Larger	Smaller
Read speed	Faster	Slower
Erase and write speed	Slower	Faster
Capacity	Smaller	Larger
Cost	Higher	Lower

They are compared in table 2.1 presented here. From the fact that NOR flash has larger cells one can deduce their faster read speed, lower write speed, smaller capacity and higher cost. NAND flash with their smaller cells are like opposite.

Data storage media

Non-volatile memory might take various forms: electronic chips, magnetic disks or tapes, and optical disks. Picture of various examples of storage media is shown in figure 2.11.

Figure: 2.11: Various types of data storage media. Description in text.

In upper left corner there is Compact Flash card (with white-blue sticker on it) having capacity of 2 GB. Right to it there is tiny, black Micro SD Card with capacity of 8 GB. Next to it there is handy “pendrive” with popular USB connector also providing 8 GB of space.

Hard Disk Drive (HDD) is shown below memory cards. This one was made in technology of rotating plates that still has some popularity in cheaper disks. Contemporary realizations are using solid state chips instead of rotating planes thus are known as Solid State Disks (SSD). HDD and SSD have complex but fast interface to PC mainboard. Formerly it was in IDE standard with parallel data lines while nowadays serial ATA (sATA) interface is more popular thanks to higher throughput. Popular 3.5 inch form factor that is presented in the figure 2.11 is common for HDD and SSD too. There are also 2.5 inch disks that are more popular in laptop computers where small size is more important than capacity. The disk shown on the photo has capacity of 1 TB (terabyte). In the same form factor a wide range of capacities can be found up to several TB. Historically early disks of this format had capacities in the range starting with single megabytes so their sizes might be compared with todays memory cards.

Top-right corner is occupied with Compact Disk (CD) that is example of optical storage. It provided capacity of about 650-700 MB. CD is predecessor of more modern DVD and Blu-ray

disks that have typical capacity equal to 4.7 GB and 25 GB respectively. Development of DVD and Blu-ray technologies improved capacities in both standards but nowadays popularity of optical media storage is in decline due to increased adoption of broadband Internet access. Most of optical media enables only single write for permanent storage but there also are rewritable disks such as CD-RW or DVD-RAM.

Last row is occupied with quite archaic storage media. In bottom left corner there is a magnetic tape for data archiving. It has capacity of 4 GB or 8 GB with "densier" saving. 3.5 inch floppy disk having 1.44 MB capacity is shown in the middle of bottom corner. It became popular in late XXth century as its space was large enough to store a computer game or some application. Operating systems at that time were much smaller than nowadays and a handful of floppy disks was enough to distribute their whole suite. Another format was popular before 3.5 inch floppy disks conquered the market. It was 5.25 inch floppy disk which is shown in the bottom-right corner of the figure 2.11. It was possible to record data on both sides of the 5.25 disk where one side provides "massive" capacity of about 200 kB. Yes – less than 200 kilobytes was enough to store program or game for 8-bit microcomputer or PC in 1980s!

Magnetic and optical data storage are beyond scope of this book so they will not be discussed any further.

3. Common Wired Link Standards

If you think of standardization as the best that you know today, but which is to be improved tomorrow; you get somewhere.
Henry Ford

Interfaces that can be found in electronic components might be separated into two groups:

- parallel interfaces,
- serial interfaces.

In parallel interfaces there are many lines over which data is transferred simultaneously. Intuitively the more lines the higher the throughput might be . Parallel approach works well if properly applied. But there are issues that appear with increasing speed of transmission. Firstly it is lines synchronization which is a problem that is proportional to data transmission clock rate. Secondly higher frequencies mean shorter wavelengths so even short unshielded wire or copper path on PCB might become an antenna that transmits signals without control. These signals might cause electromagnetic compatibility problems such as interference with other lines and lines intermodulation.

Parallel interfaces were once used for communication with printers – famous LPT port in PC computers. Other area of parallel data transmission domination was communication with hard disks. There were two noteworthy standards: Small Computer Systems Interface (SCSI) and Advanced Technology Attachment (ATA). SCSI was available in high-end computers such as servers and in its early days was only parallel interface. But problems with cabling led engineers to modify this standard and make it Serial Attached SCSI better known as SAS, that was also targeted at servers. Meanwhile Parallel ATA standard which was designed for typical PC computers went similar way of evolution and became a serial interface well known today as SATA.

Microprocessors do not provide sophisticated interfacing and cannot communicate with external chips on their own. That is why there are “north bridge” and “south bridge”, and other chips on the PC computer mainboards. Some of these chips might be MCUs or MCU-based Application-Specific Integrated Circuits (ASICs) which do have implementation of specific hardware interfaces. In next sections the four most common standards of communication that are widely supported by microcontrollers will be briefly discussed.

Simple serial connections

Discussion of serial connection is often started with RS-232 standard that was developed in 1962. It is prominant for being popular for almost five decades! Operational industrial,

scientific and computer networks equipment which employs this interface might be found even nowadays.

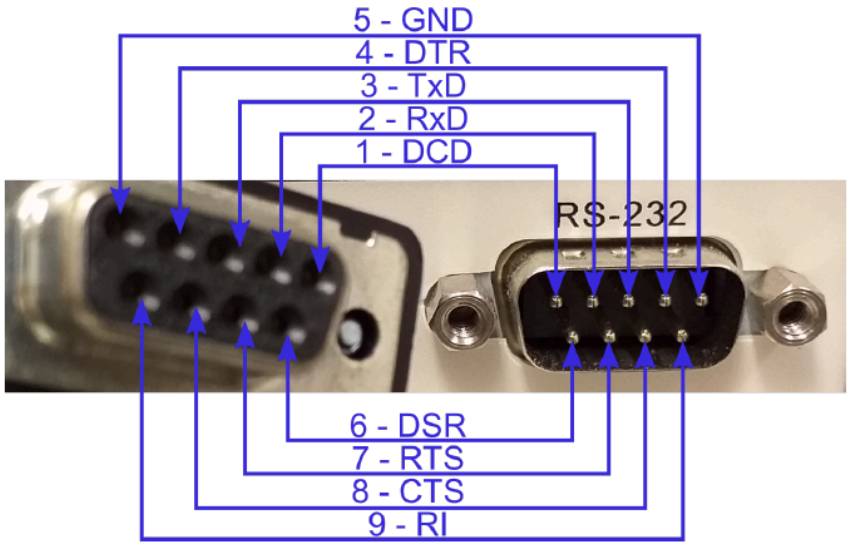

Figure 3.1: RS-232 connector (DB-9) with its lines described.

Standard originated with many flow control lines and relatively high bipolar voltages: ± 15 while up to ± 25 was considered as possible. Over the years voltages used in practice were lowered to ranges like ± 5 or ± 3. Furthermore some control lines were practically removed as considered to be not necessary any more due to the fact that more and more transmission control was done in specialized integrated circuits or in the software layer. RS-232 connector with lines described is shown in figure 3.1.

RS-232 standard spawned more rigid versions dedicated to industrial applications known as RS-422 and RS-485. On the other hand it was observed that proper handling of flow control makes it possible to minimize the number of lines to just two of them, known as RX and TX where former one is for data reception and latter one is for transmission. RXTX pair is probably the most common solution for communication that is available in microcontrollers. For example Arduino boards which are popular among hobbyists and experimenters, provide these two lines on their PCB pinout as standard. Behind the simple two-line connection there is a serial communication controller embedded inside the microcontroller chip that usually provides a buffer for transmitted data. The controller might provide asynchronous (UART) or synchronous (USART) transmission. If traditional ± 12 voltage is needed then external controller like MAX232 is used to convert low voltages to and from the microcontroller side.

One could consider that two lines for two-way communication is the limit of optimization. However, there is 1-wire standard that operates on single data line and reference ground line. So in fact there must be at least two lines but the one for transmission might provide positive voltage to the end device simultaneously with transmission. Ground line might be a shield of cable, like in coaxial cable. Hence with thin cable it is possible both to power up the end device and have communication with it. Thanks to its simplicity the 1-wire is often used in embedded

applications which need to have a remote sensor. Broadly known example of 1-wire sensor is Maxim DS18B20 temperature sensor.

Common drawback for simple serial connections is their low speed. Throughput up to 115200 bauds is theoretically available while many devices work with more typical transmission speeds such as 19200 or 9600 bauds. These can be expressed as 12.8 kB/s, 2.1 kB/s, and 1.1 kB/s respectively. Such channel capacity might be suitable for simple control protocols or basic monitoring but will not be enough for more demanding applications like digital media transmission.

Important fact about the RS-232 standard and its successors is that they require no fees to any organization. They are broadly used, well documented, have low learning curve, and are repeatedly implemented in integrated circuits. Therefore these standards or *de facto* standards still might be considered as interesting option for hardware development especially in the early stage of research.

Universal Serial Bus (USB)

USB is natural choice for popular consumer electronics because it is perceived as simple to use by everyday users. USB standard version 1.1 was established in 1998. Then it evolved in terms of data throughput, power supply requirements and complexity. Fortunately for consumers all its intricacy is well hidden within hardware specification and necessary software drivers. At time of writing this book version 4 is under development which should be presented in 2019. Technical differences between USB versions are shown in table 3.1.

Table 3.1: Comparison and evolvement of USB standard. Please note that transmission throughput is given in megabytes per second.

Feature	USB 1.1	USB 2.0	USB 3.1	USB 3.2
Maximum throughput [MB/s]	1.5	60	625	2500
Effective throughput [MB/s]	1.5	~30	~500	~1800
Supplied current [mA]	500	500	900	900
Devices on bus				

USB can connect physically separate devices but is limited by the maximum length of the cable. Theoretical maximum length of USB 2.0 cable is 5 meters while for USB 3.0 due to higher data rate it is even shorter limited to 3 meters. In practice, with good active cable a distance of several or dozen of meters might be achieved. To extend it further one may use USB hub with which 30 meters range is possible. There are also USB over Ethernet extenders thanks to which distance is limited by Ethernet specification that is about 100 meters.

Table [tab:usb] shows typical implementations of USB standard and their capability to work as power supply for connected devices . There are however special *dedicated charging ports*, which can provide higher currents and voltages than normal USB ports. Such USB 2.0 port can

provide 1.5 A while USB 3.0 port of this kind can provide 1.8 A. The drawback is that such ports are only for power supply and do not let data transmission. Furthermore there are *powered USB ports* which can provide 6 A at voltages 12 V and 24 V so maximum power of 144 W.

The basic version of USB connector has only four lines:

- **V_{BUS}** – power supply, 5V at up to 900 mA
- **D–** – data transmission differential line
- **D+** – data transmission differential line
- **GND** – ground (reference)

Figure 3.2: Popular USB 2.0 connectors – from left to right: Type A, Type B mini, Type B micro.

USB 3.0 is physically backward compatible with 2.0 as 2.0 is backward compatible with 1.0. It means that older device might be connected through new connector type although it will not benefit from additional features provided by contemporary version of the standard. Visually USB 3.0 might be distinguished from 2.0 by blue elements of the sockets and plugs. USB 3.0 may have additional lines for data transmission with transmit (SSTX) and receive (SSRX) are separated. Both of these are still differential lines so it makes four additional lines for transmission totally. USB 3.0 has also additional GND reference line. USB 3.2 uses “dual lane” approach doubling its lines so that there are 24 pins in the connector. Plugs employing this version of standard are easy to recognize as they obviously do not fit with old type sockets.

USB standard is known for having confusingly large number of connector types. Nowadays the most popular connectors are Type-A, Mini-B and Micro-B which are shown on figure 3.2. First one is used for example in memory sticks (pendrives) while two others are popular in mobile phones and cameras where smaller footprint is favored. USB 3.0 and so its connectors are gaining popularity slowly as there is not so high demand for it on consumer market. It is used in Point-of-Sale terminals, high-speed FPGA boards such as in Software Defined Radio

and mass storage devices. Usage of type-C connectors for USB 3.x is on the rise, especially in laptop computers, where it may be used for power charging thanks to additional lines.

Versioning of USB standard might be considered hard to follow by everyday consumers. Therefore there are also brand names of USB versions as shown in table 3.2.

Table 3.2: Brand names of USB versions.

USB versions	Brand name
0.9 – 1.1	FullSpeed
2.0	Hi-Speed
3.0	SuperSpeed
3.1 – 3.2	SuperSpeed+
4	Thunderbolt3 (TBA)

There are two basic types of USB products: *device* and *host*. *Host* plays role of the bus controller to which *device*-type units connect. This limits functionality of equipment hence since 2001 USB On-The-Go (OTG) is gaining popularity. OTG device can switch back and forth between *device* and *host* role, depending on circumstances. For example a smartphone can be a *device* if connected to PC computer and then it may be a *host* if it has external keyboard connected.

USB standard is supported by many microcontrollers in 8-, 16-, 32-bits architectures. Some MCUs given here should not prevent reader from searching for chip that best suits specific project requirements. One example is 8-bit Microchip ATmega32U4, prominent for being the core of several Arduino boards such as Arduino Leondardo. This chip supports behaviour of FullSpeed device and bus controller. It is available in beginner-friendly easy to solder TQFP package with only 44 pins. Microchips also manufactures 32-bit PIC32MX family of MCUs among which many chips can work as FullSpeed device, host and OTG. Many chips in this series are also available in TQFP package. STMicroelectronics is also manufacturing many MCUs with USB support in their STM32 series. Another example of chips which support USB standard are ARM Cortex-based Gecko MCUs. It is hardly possible to discuss all MCUs providing USB support therefore reader is advised to visit manufacturer's websites or use tools that support the choice such as STM32CubeMX.

Serial Peripheral Interface (SPI)

SPI is another serial interface notable in development of microcontroller-based systems. It has long tradition as it was developed in 1980 by Motorola and since then it became *de facto* industrial standard. Nowadays it is as omnipresent in microcontroller architectures as the USB is in the PC computers. Firstly because it is relatively simple to use. Secondly because there are lots of external components such as:

- sensors,
- ADC and DAC chips,
- flash data storage,
- memory chips,
- RTC chips
- displays for embedded systems.

These simple components often employ the SPI as the main or the only way for communication. Due to the popularity of this standard it is hard to find MCU which does not provide at least single SPI channel while quite often there are more available.

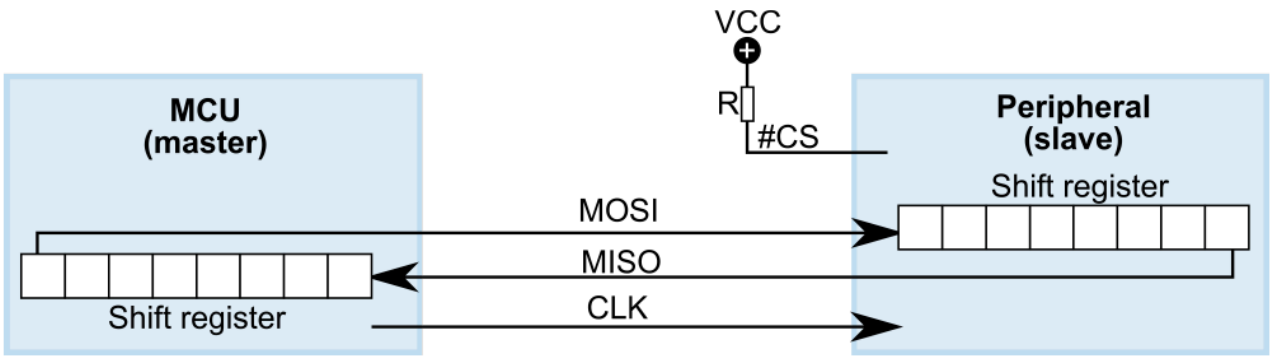

Figure 3.3: SPI communication between master and slave devices.

From architectural point of view SPI is digital, serial and synchronous communication interface. Data is transmitted in stream-like mode in both directions at the same time so it looks more like data exchange. Mechanism is based on shift registers as it is shown on block diagram in figure 3.3. Even if one side of channel does not have anything to transmit it transmits some dummy data so that equal number of bytes is transmitted in both ways so it remains synchronous. Line from master to slave is labeled as Master-Output-Slave-Input (MOSI), while the reverse direction is Master-Input-Slave-Output (MISO). In the figure 3.3 the clock (CLK) is also shown. Chip-select (CS) line in this special case of just two devices interconnected might be implemented as simple pull-up of *slave* CS line so that it is always-on.

SPI was intended and is used for very short communication like between MCU and its supporting peripherals. When one needs to employ this standard for longer transmission lines it becomes quite hard to achieve and possibly another approach should be considered. This limitation is related to synchronous nature of the SPI.

Synchronization requires clocking which is somewhat tricky as it may be done in more than one way. First of all clock is sourced by *master* device and distributed to all *slaves* hence there is only one source of reference clock signal that decides at what data rate transmissions should occur. However, *slave* devices might have some more clock limitations and the impact of line length needs to be considered as well. SPI clock rate depends on MCU system clock. In typical situations it equals to hundreds of kHz up to dozen of MHz.

Table 3.3: SPI modes.

Mode	Polarity (CPOL)	Phase (CPHA)
0	0	0
1	0	1
2	1	0
3	1	1

Data is being latched (captured) at falling or rising slope of the clocking signal while the clock signal itself might be positively or negatively polarized. Therefore there are four possibilities from which one needs to be chosen for all devices on the same SPI bus . SPI modes are shown in table 3.3.

If clock polarity (CPOL) equals 0 then such clock is idle at level low and first (leading) edge is the rising one while the trailing edge is the falling one. If clock polarity equals to 1 it means that clock idles at level high so leading edge is the falling one while trailing edge is the rising one. When clock phase (CPHA) equals to 0 then the data is asserted (already available) on the first edge of the clock signal. When clock phase equals to 1 then it leading edge signals a moment after which the data will be asserted.

The side that initiates communication is labeled as *master* while the other is known as *slave*. There is only one *master* device on single SPI bus while there may be many *slaves*. In such situation a chip-select (CS) lines needs to be used. MCU changes sets states on all CS lines so that only one peripheral *slave* device is chosen. The number of *slave* devices determines how many CS lines should be employed. Such approach is shown in figure 3.4. For clarity of the diagram there are no pull-up resistors shown on #CS lines which should be present in general to keep unused lines in known state.

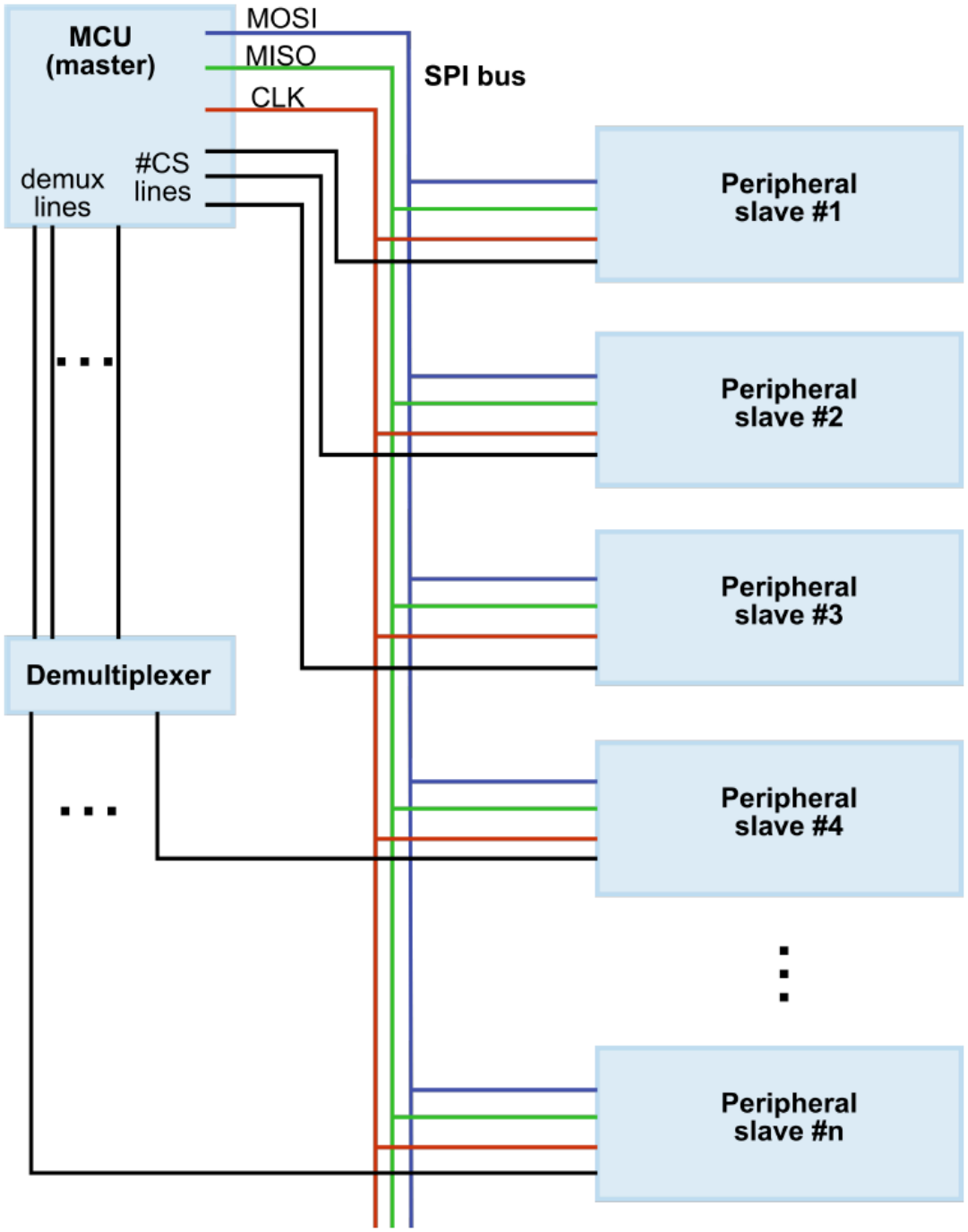

Figure 3.4: Master SPI device with many slaves.

There is no theoretical limit to number of SPI slaves but in real applications several dozens might be a practical maximum. Firstly it is due to MCU pin fan-out limit. Simply it is maximum current that can be provided by MCU to drive clocks of its many *slave* chips. Lengths of transmission lines plays significant role here as longer lines increase wear of fan-out. Then there is practical limitation of CS lines as utilizing dozens of MCU pins just for chip select lines is rather just a waste. It can be improved with use of demultiplexer so for example just 4 lines from MCU might select one of 16 *slave* peripherals. Finally physical space occupied by SPI lines

and problems with routing them on PCB becomes an obstacle that prevents using too many *slave* devices.

Inter-Integrated Circuit (I^2C)

Inter-Integrated Circuit is popular alternative to SPI which was discussed in previous section. I2C was developed by Philips in 1982 as synchronous, packet switched serial computer bus. Synchronization is based on clock line (SCL) while packets are transmitted on data line (SDA). These two lines are the only ones necessary therefore it is labeled as two-wire interface in Microchip (formerly Atmel) nomenclature. License fees for use of the standard were collected by Philips Semiconductor until it became NXP Semiconductor in 2006. Now fee is necessary for acquiring a device address only.

Table 3.4: Evolution of I2C standard.

Year	Version	Data rate	Description
1982	–	100 kbps	original version
1992	1	400 kbps	Fast mode
1998	2	3.4 Mbps	High-speed mode
2007	3	1 Mbps	Fast mode+
2012	4	5 Mbps	Ultra-fast mode

I2C was evolving over the years and as it had to compete with other standards it improved transmission speed. Versioning and possible data rates are shown in table 3.4. Version from 1982 was designed for the benefit of building control with Philips technology and was not officially standardized. Version 1 extended address space to 10 bits (1008 devices in practice) but this feature is not fully adopted and still most chips on market use only 7-bit addressing. Version 2 introduced power-saving features. Version 4 uses push-pull logic but is only an unidirectional bus. Versions presented in table 3.4 include only major versions of the I2C standard and omit minor corrections.

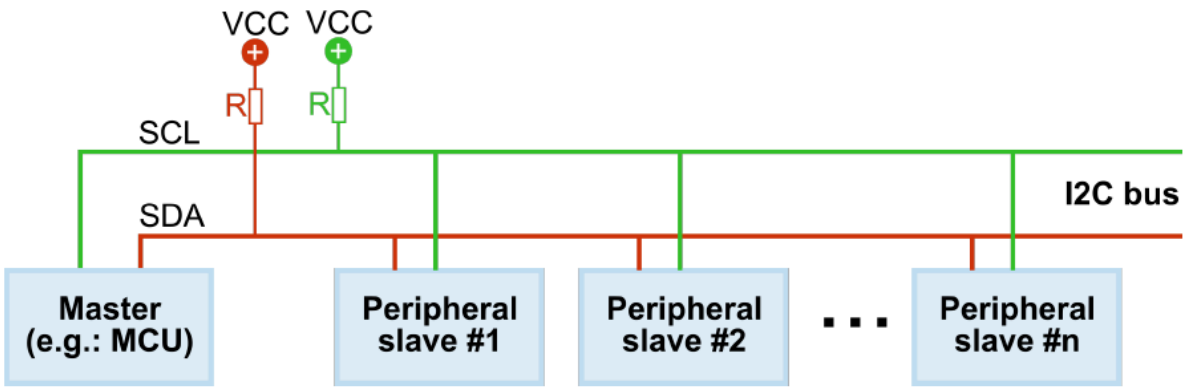

Figure 3.5: I2C bus.

Typical devices on the I2C bus are addressed with 7-bit addresses. This address is transmitted at the beginning of packet exchange so that only one specific device is active on the bus at a time. Thanks to 7-bit address space there could be 128 devices on the bus theoretically, although in practice some addresses are reserved so that up to 112 devices might co-exist on single bus. Furthermore bus length is limited by its capacitance which should not exceed 400 pF. Diagram of the bus is shown in figure 3.5. Pull-up resistors are necessary in this standard. In typical logic their values are usually in range from 1 kΩ to 10 kΩ. Lower resistance values are "strong pull-ups" which means that they are harder to change by master node but also more resilient to noise while lower resistance values are "weak pull-ups" which are easier to change but also more prone to noise.

Address of integrated circuit chip that employs I2C is applied during the manufacturing process and it must use one obtained from the NXP. If there are many chips of the same kind are to be used on single I2C bus it makes a problem as all of them will have the same address. Therefore often on the chip there is one or more pins which need to be pulled up (or down) to change the address. But even if there are 3 lines the address of chip is limited to one of eight possible values. Hence with more I2C chips, like in sophisticated, wired measurement system, it is necessary to use multiplexers to change device addresses on the fly. The drawback of this approach is increased number of lines between controller (usually MCU) and chips so that one of main benefits from using I2C that is necessity to have just two lines is lost.

I2C is widely used in PC computers to gather information from the mainboard and some of its peripherals such as temperature and fan speeds. List of component types that employ I2C standard is similar to the list of SPI devices.

4. Programming Environment

Controlling complexity is the essence of computer programming.
Brian Kernighan

Readers of this book are expected to be familiar with modern, graphical computer environment which takes form of a "computer desktop". Such GUI is also available in Linux installations typical on PC computers therefore no trouble is expected in using common tools like file system browser, text editors and web browser.

Programming environment consist of several applications which may or may be not bundled in Integrated Development Environment (IDE). There are many applications that provide features necessary in IDE. Some may be lighter, easier to use and start programming with them, contrary to others which may attempt to arrange whole programming process. Generally the more complex IDE the more initial effort is necessary to become familiar with them. There is no good or bad choice in this matter and it depends more on personal view, team experience and organizational culture. Core elements of programming environment with exemplary applications will be discussed in following sections

Code editor

Obviously to write programs one needs a source code editor. It might be simple text files editor with notepad-like style. In Linux distributions, depending on GUI that is in use, one may have **Mousepad** in XFCE4 desktop, **gedit** in GNOME desktop or **SciTE** that is based on GTK+ portable library which makes it available not only on Linux but also on different platforms. Slightly more advanced text editor is **Kate** from KDE desktop environment. Collage of these four programs is shown in figure 4.1.

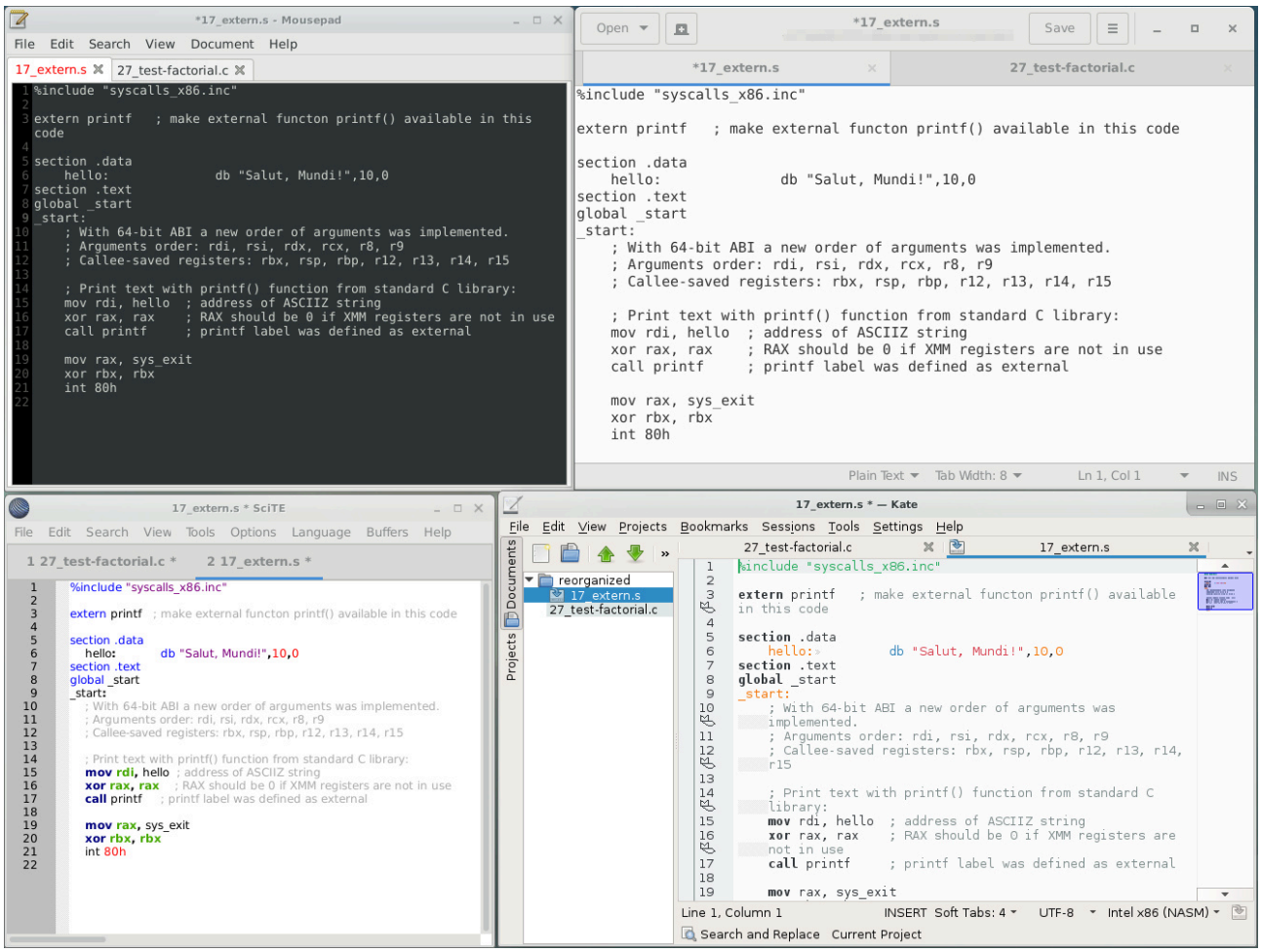

Figure 4.1: Mousepad, gedit, SciTE and Kate text editors with assembly code opened.

All of these applications provide line numbering and multiple files edition. Some may provide additional features such as auto-completion and auto-indentation. Syntax highlighting is also popular among them but not all editors recognize code written in assembly language. For the same reason the feature of code folding available in some of these editors does not work on assembly language source files.

A bit more advanced code editors let user define and maintain structure of a project that may be built of many files. Geany that is shown in figure 4.2 is example of such application. It provides syntax highlighting for variety of languages, which is not always the case for simpler editors. Editing is supported with auto-completion, auto-indentation and code folding. It provides compilation of the code and execution of resulting application directly from the GUI. Furthermore Geany may be extended with one of many available plugins.

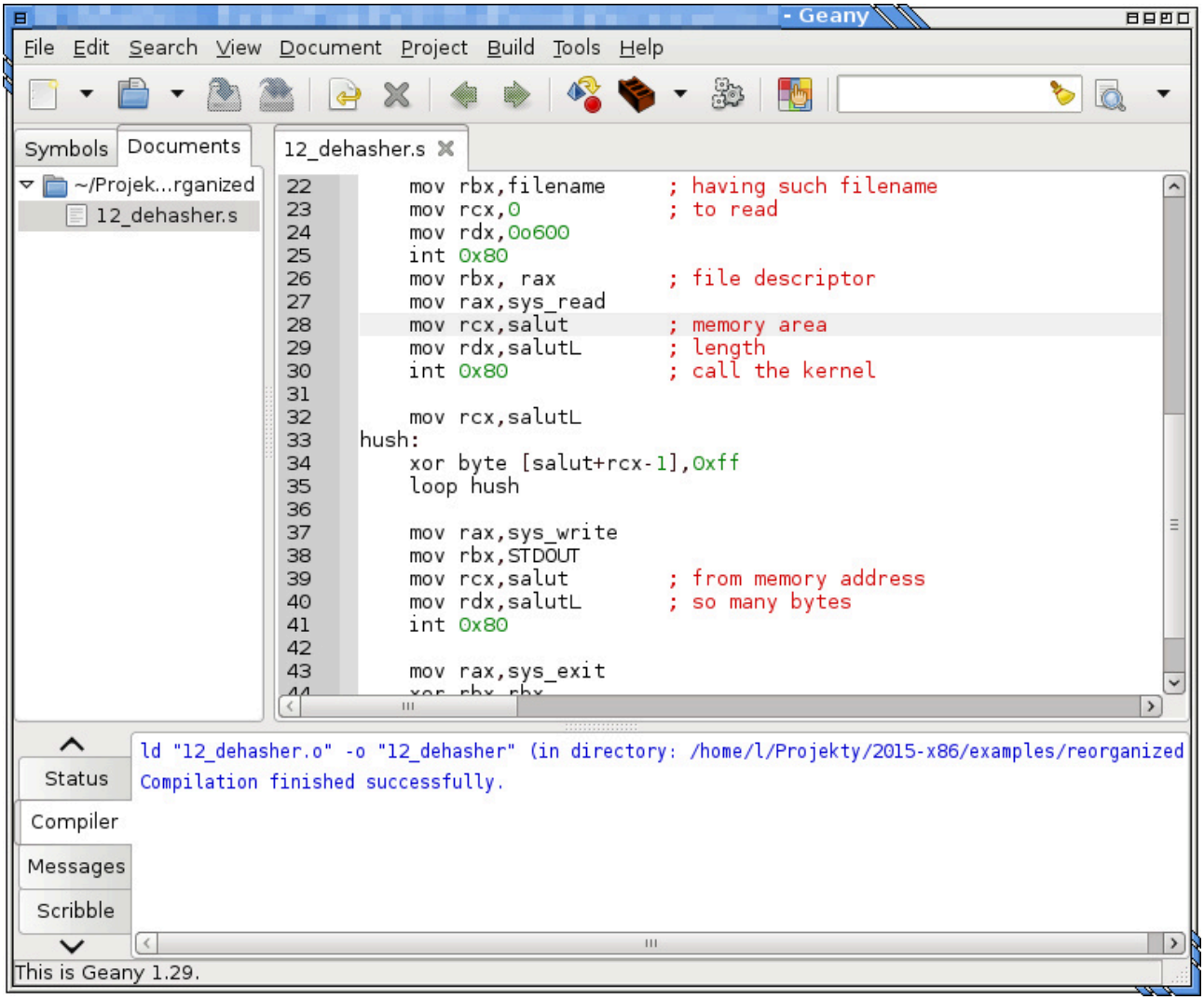

Figure 4.2: Geany code editor with single file opened.

When there is no GUI available one may use text editors that are started from command line and work in textual environment. Widely known examples in this category are: **pico**, **nano**, **vim** and **emacs**. The huge benefit of using them is that they may be started on remote machine including embedded computers with limited resources. These editors are considered to be a good choice for experienced power-users.

Compiler and Linker

To create a program one needs to have **source code file** which in majority of programming languages is just a simple text file that was created with editors such as discussed in the previous section. Source code files have names with extensions that indicate in which computer programming language they had been written. As there is ".c" for C programs, ".cpp" for C++, ".py" for Python so there is ".s" or ".asm" for code written in assembly language.

Source file is processed by **compiler** which analyses its contents lexically and grammatically. If there are no errors then compiler should generate binary file for specified machine architecture. This file is known as *relocatable object file*, although it has nothing to do with object-oriented programming. Object in this context means that this is separate piece of code and data that theoretically could be processed by machine for which the compilation happened. This kind of file is usually represented with ".o" extension.

There are many compilers available both as free (open source) and proprietary software. Some compilers are available on many platforms and architectures. Furthermore on x86 architecture there are two competing flavors of assembly coding: Intel and AT&T. Former one is considered to be easier to understand so it is more often a first choice for new programmers. Latter one is popular on UNIX-like operating systems and is default option in many debuggers. Simplicity of Intel flavor is paid with price of some ambiguity because some keywords might be confused with user variables. It is not the case in AT&T flavor where values and registers are identified with mandatory prefixes. Order of arguments is also different in both styles. Operands are read right-to-left in Intel style but left-to-right in AT&T style.

Many development tools that are popular on Linux platforms came from GNU projects such as binutils and use AT&T flavor of assembly language. Familiarity with this assembly style is not mandatory for readers of this book but might be helpful, especially during debugging of executable programs and binary files with **GDB** debugger, which is discussed later.

More experienced readers might be already aware that behind the very well known C compiler from GNU Compilers Collection that is renowned **gcc** there is assembly compiler. This compiler can be started independently by issuing command **as**, although it is rarely used this way. This book will require some use of gcc but we will avoid direct use of as.

One may consider use of **NASM** compiler which stands for Netwide Assembler . It is very popular, cross-platform and easy to use assembly compiler distributed as free open-source software on BSD license. This book is based on NASM primarily because of these features. it is so popular that most Linux distributions should have it in their package systems available for easy installation. All information about NASM syntax and the compiler documentation can be found on its main website https://www.nasm.us/. Moreover, development of its source code might be observed at its **GIT** repository https://repo.or.cz/w/nasm.git from which the latest version is available for downloads.

NASM has long-standing development history as initial release took place in year 1996. Initial authors were Simon Tatham who is author of PuTTY (famous ssh client for Windows) and Julian Hall. Project gained attention and gathered a team of developers that is nowadays lead by Hans Peter Anvin, one of Linux kernel developers. NASM versioning is a bit peculiar as the first release had version number 0.90. Despite continuous development it never reached 1.0, and currently the major version number is 2. Version 2.00 from year 2007 was the first that introduced support of x86_64 architecture. NASM is backward compatible so it still can generate 32-bit code for Unix-like machines and Windows, and even 16-bit code for DOS.

This book is based on NASM so details of its usage will be given in the next chapter, with many realistic examples.

Alternatively one may try using FASM which was written by Tomasz Grysztar. FASM first

release happened in year 2000. Since then it gained a lot of attention and popularity in low-level developers community thanks to its features: efficiency of compilation, ease of use and simplified BSD license which makes it free (open source) application. Furthermore it is cross-platform application available on Linux, Windows and BSD operating systems and it can produce executable files in many formats: MZ (DOS), PE (Windows), COFF (Unix-like) and ELF (Linux). FASM is easy to obtain and is popular package in Linux repository systems. It is distinct due to fact that the compiler program almost does not use any command line arguments because all compilation options are embedded in the code source as compiler directives.

Usage of FASM is really simple as user just needs to start it from command line and provide name of source file and optionally name of an output file:

```
> fasm myfile.asm program
```

Shortly after the above command is finished, assuming that `myfile.asm` was available in current directory, a new file will emerge with name `program`.

Basics of Linker

Linking is the last stage after the compilation necessary for preparation of the program. It builds an executable file from one or more relocatable files. Linker in Linux environment might be started explicitly as **ld** command. Some compilers have the linker embedded (like FASM) or invoke it silently (like GCC during C code compilation). Linking may also associate program with static or shared libraries that contain code necessary for this program to execute. Such more advanced linking will be discussed on practical examples in the next chapter.

Debuggers

Debugger is an application that enables its user to analyze and eventually understand how the program code is executed on-the-fly. Debuggers can start an application under test and control process of its execution. Instructions may be executed in **step**-by-step mode or until a **breakpoint** is reached. Breakpoints are set by the debugger user. Variety of views and additional tools provide insight into the program behavior. To sum up, there are several purposes for which debuggers are used:

- analyze program behavior,
- find problematic or faulty code,
- reverse-engineering.

GNU Debugger

GNU Debugger also known as **GDB** is the flagship of debuggers that are available in GNU/Linux. It is actively developed since 1986 and still new features are added. GDB supports about 30 architectures such as: x86 and x86_64, IA-64 Itanium, ARM, AVR, Alpha, Motorola 68000, MIPS, PowerPC, PA-RISC, SPARC, VAX and more.

GDB has remote debugging capability so that for example debugging application runs on PC computer to which embedded board with ARM processor is connected. Program executed on ARM platform is controlled by GDB on PC. It is indispensable feature in debugging embedded systems.

Recent version of GDB provides execution of debugged program not only in "forward direction", which is normal, typical and easy to understand but also "backwards" so that one may reverse execution of code that caused error and see how it happened.

As it was mentioned above, the GDB uses AT&T assembly flavor by default. Fortunately, it may be changed with one line in the configuration file (`.gdbinit`):

```
set disassembly-flavor intel
```

GDB comes with very limited GUI (start with `-tui`) and is generally considered as not very user-friendly. However, the purpose of the GDB is to provide excellent debugging tool so it leaves user experience to front-end applications that may use GDB as their "engine". This approach guarantees flexibility and adaptability to various development environments hence different architectures are supported consistently.

One way of adapting GDB is to enhance its own configuration as it was done in gdb-dashboard project available on GitHub: https://github.com/cyrus-and/gdb-dashboard.git. This "GDB on steroids" might be a great option as it brings almost no overhead to vanilla gdb, is easy to distribute and should work in every situation. All that needs to be done is to copy GDB config (*.gdbinit*) to user's home directory. Here please note that files with names that start with dot are "hidden" in Linux. Debug session using this config file is shown in figure 4.3.

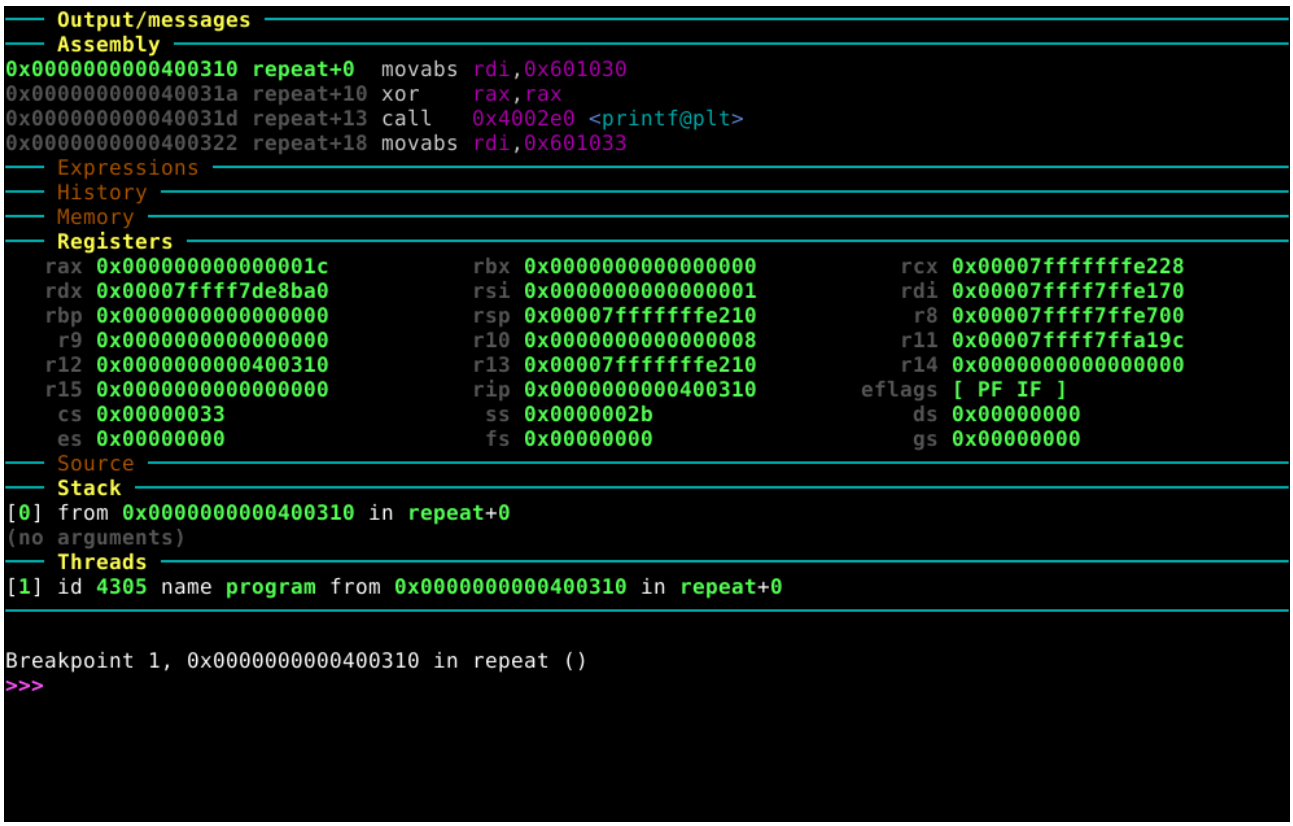

Figure 4.3: GDB with gdb-dashboard configuration.

However, gdb-dashboard will display such nice information only during debugging. So the minimum operation that must be done in DBG is to start a program under investigation, preferably with breakpoint at its beginning. Here is an example of debugging bash, popular Linux shell:

```
(gdb) b _start
Breakpoint 1 at 0x422270
(gdb) run
Starting program: /bin/bash

Breakpoint 1, 0x0000000000422270 in _start ()
```

In the above example a breakpoint is set with GDB command (`b`) at debugged program entry point `_start`. Then the program is started with `run` command but stops immediately at the declared entry point and might be analyzed in step-by-step mode.

Voltron

Voltron is an application written in Python that supports GDB with different “views”. It is

available as free software and distributed under MIT license on GitHub: https://github.com/snare/voltron. Voltron is also available as package in Debian system and possibly also in other GNU/Linux distributions so it is easy to install. After correct installation user should change GDB configuration (*.gdbinit*) and put there one line that points to installed Voltron start script:

```
source /usr/local/lib/python3.5/dist-packages/voltron/entry.py
```

Voltron is not a wrapper but a debugger front-end that uses features which already exist in GDB although not so easy to use. The GDB must start first debugging the program before Voltron will start. Otherwise there will be error message because Voltron cannot connect to GDB session. User is informed with a message that GDB is started with Voltron enabled as in this example:

```
> gdb /bin/bash
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
[...]
Voltron loaded.
Reading symbols from /bin/bash...(no debugging symbols found)...done.
```

Each of terminals shown in figure 4.4 was started independently. Then in each of them Voltron was invoked with different option to have specific view. In the left column, reading from top there are views of: memory, registers, breakpoints and stack. On the right side there is source code of debugged program opened with VIM editor, disassembled code with indicator of next instruction and GDB shell where commands may be given.

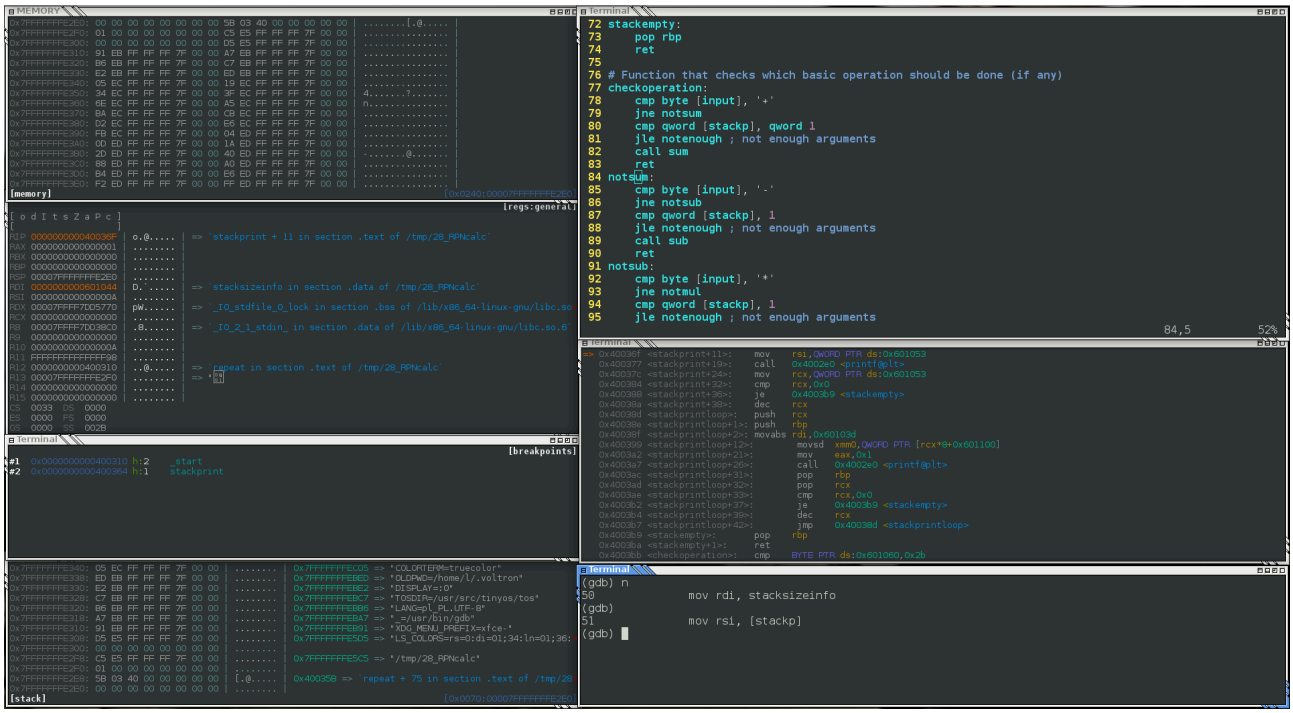

Figure 4.4: GDB session with Voltron configuration.

To see all available views users should get help message by using this command:

```
> voltron v -h
```

The list of possible views may change over time but currently there are:

- breakpoints at which continuous execution should stop – b, bp, break, breakpoints
- disassembly of binary code back to instruction mnemonics – d, dis, disasm
- CPU registers – r, reg, register, registers
- computer memory – m, mem, memory
- stack area view – s, st, stack
- backtrace view of previously called functions – t, bt, back, backtrace

The list also shows command line arguments to utilize specific view. For example to start disassembly one may simply invoke Voltron with this command:

```
voltron v d
```

However, Voltron will produce no view or only an error message unless GDB is actively debugging just like it is with gdb-dashboard GDB config.

To sum up Voltron improves usage of GDB as it makes possible to employ multiple terminals running at once on graphical desktop in windowed-like mode as it is presented in figure 4.4. However, it remains a simple text mode application so it can be used on remote machine (e.g.: server) with only limited resources and no graphics, or even no screen connected.

GDB Front-Ends Oriented for Graphical Desktop

More graphically advanced front-ends also exist such as Nemiver and KDbg. Nemiver, shown in figure 4.5 is application more oriented to Gnome Linux desktop while KDbg, shown in figure 4.6 is based on KDE desktop environment. In typical situation Linux distribution will let run any of them regardless of actually used desktop type. Both Nemiver and KDbg have similar level of functionality providing graphical fronted to commands which already exist in GDB. They require user to open executable and source code file for debugging. Then user may execute program in one of two step modes that let user analyze behavior of debugged program one instruction at a time:

- “step into” mode in which execution will step to the next instruction even if it is in a called function
- “step over” mode in which execution will immediately perform called function without showing its internals

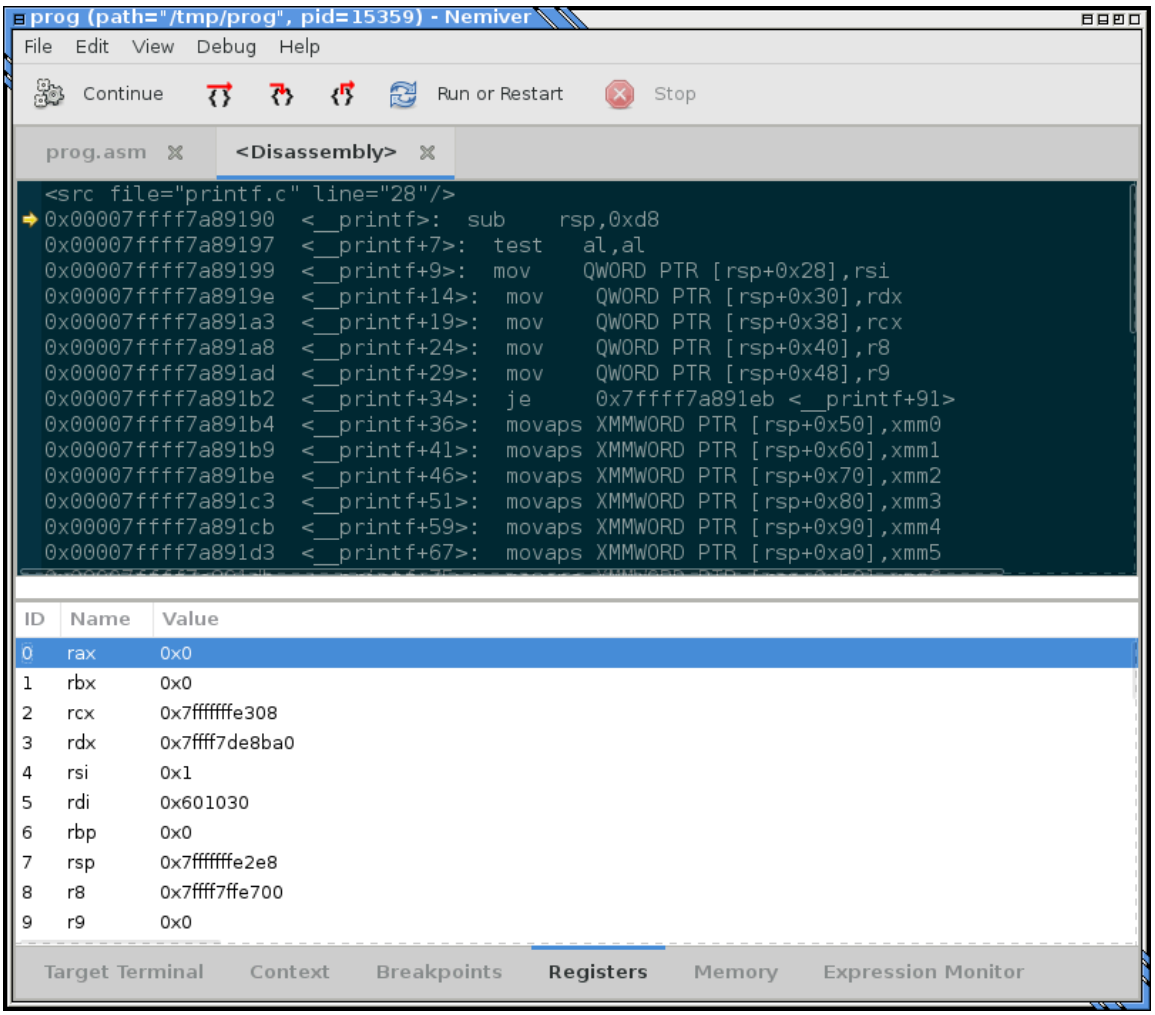

Figure 4.5: Debugging with Nemiver.

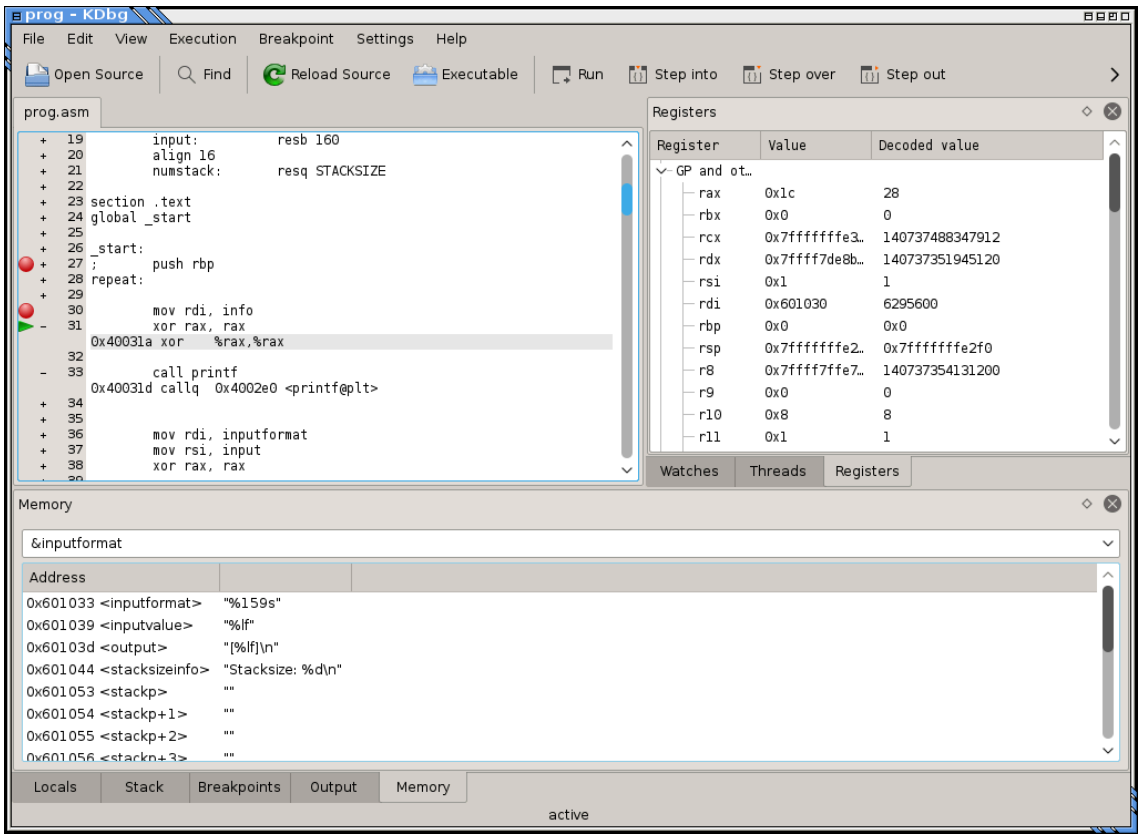

Figure 4.6: Debugging with KDbg.

Program execution in step by step mode may be very informative but also time consuming. Therefore debuggers, including gdb let user to setup breakpoints at specific positions such as: memory address, labeled instruction or function name. In front-ends like Nemiver and KDbg breakpoint is set and unset simply by clicking on the left side of code so that graphical icon like red circle will indicate breakpoint.

Radare2

Radare2 is debugger independent from GDB. It has interesting features that empower user with functions for static and dynamic code analysis and reverse-engineering. It is an open-source, free software application distributed with GNU Lesser General Public License (LGPL) license on its GitHub repository . Compiled version is available in many Linux distributions including Debian. Radare2 has impressive list of supported file formats which includes but is not limited to:

- ELF – popular in Linux and Unix-like operating systems,
- DWARF – extended debugging information
- PE – used in Microsoft Windows,

- MZ – used in Microsoft DOS,
- Java classes,
- raw binary files.

List of architectures supported by Radare2 is even longer as it supports: x86, x86_64, ARM, AVR, 8051, MIPS, PowerPC, SPARC, TMS320 and more.

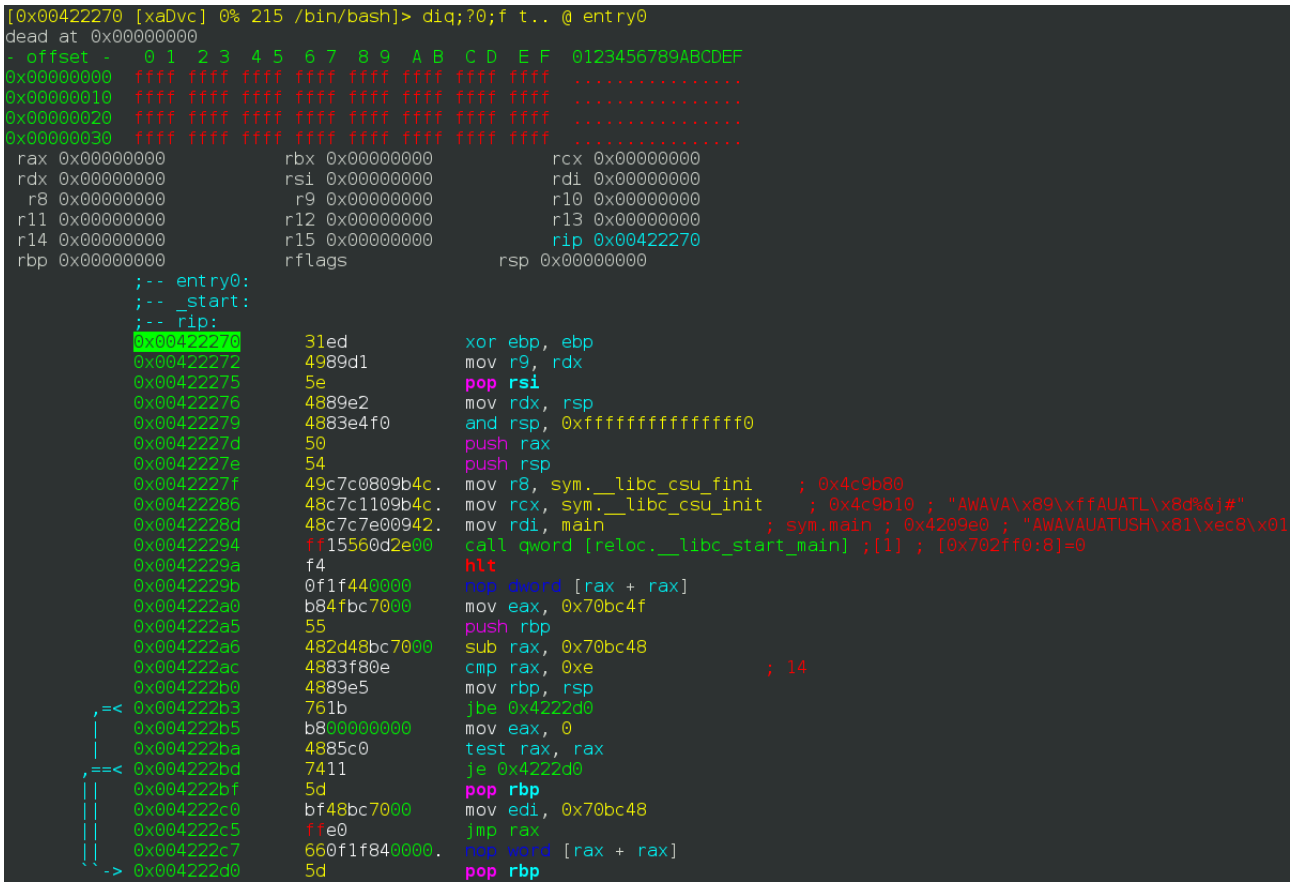

Figure 4.7: Radare2 simple session showing data in memory, CPU registers and disassembled code.

Radare2 evolved from simple hex editor in textual mode and this minimal style is still primary just as it is favored by many programmers. To start working with Radare2 in text mode one has to simply start it from command line providing name (path) of a program that is to be analyzed as in this example:

```
> r2 ./prog
```

Radare2 will start its own shell that provides set of commands. Question mark command shows help. command starts a bit more user-friendly interface which presents registers, debugging of disassembled code and data in memory. There are several views available and to switch between them one should press or . In this mode pressing shows list of available commands. To quit from the help mode press . Such session is shown in figure 4.7.

Interesting feature of Radare2 is its ability to display graph structure of the code branches. Whenever there is "if-like" statement or conditional jump then Radare2 can extract this piece of code and show it in separate frame that is connected with other blocks by True/False paths. Such view is shown in figure 4.8. To get to such view firstly one must start **r2** (the actual name of Radare2 program in the shell) with program name indented for the test as the r2 argument. Then program code must be analyzed with command: `aaa` and change the view with command: `VV`. Pressing `p` or `P` will slightly change the view, as it was mentioned above.

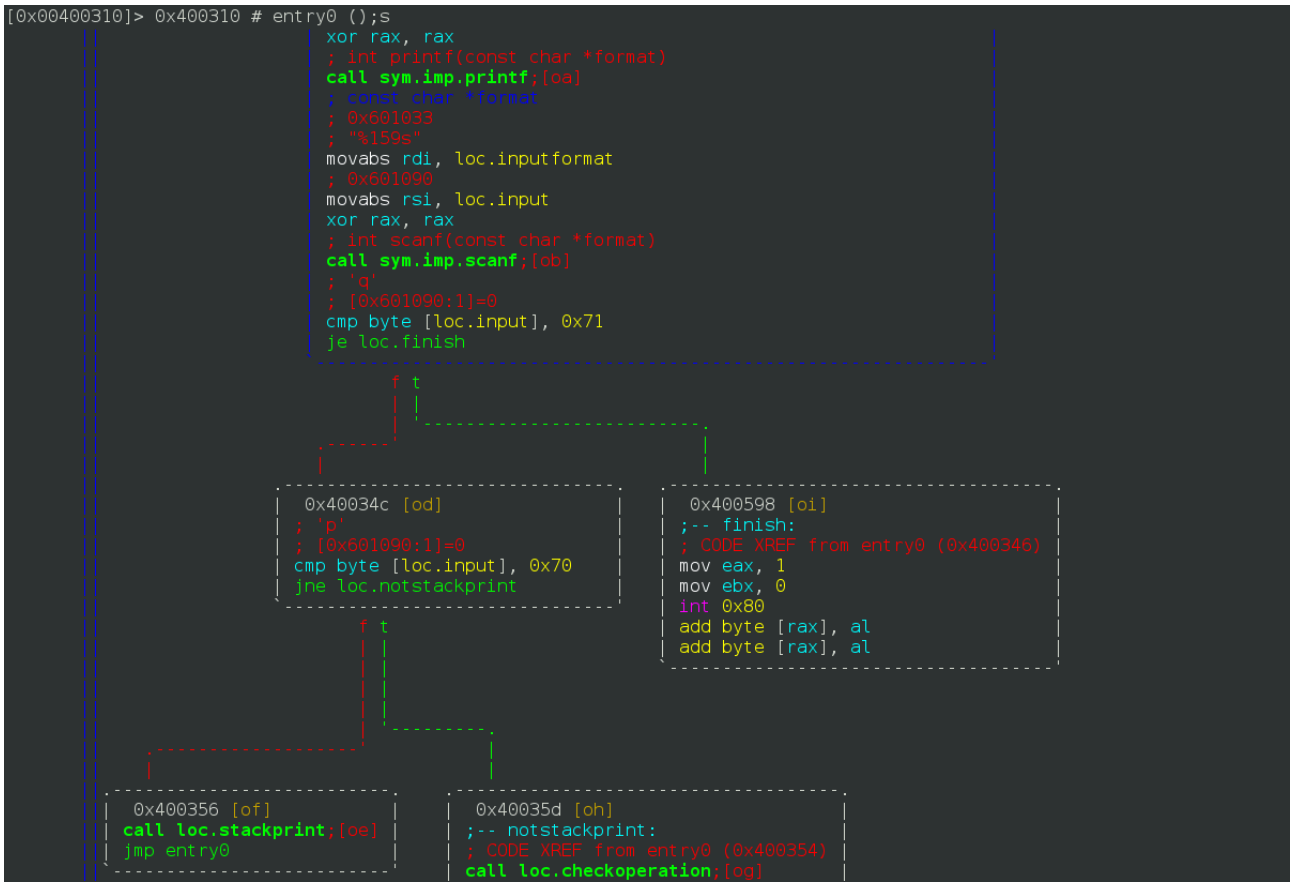

Figure 4.8: Radare2 showing structure of disassembled program.

Normally Radare2 is more oriented towards code analysis which includes disassembling. To actively debug programs with Radare2 it must be started with "-d" option as in this example:

```
r2 -d ./prog
```

Debugged program may be executed one instruction at a time in one of two step modes:

- "step into" – activated with either `s` or `F7`;
- "step over" – activated with either `S` or `F8`;

User may setup **breakpoints** either by using `db` command or in visual mode started with `Vpp`

command. In visual mode cursor must be activated with `c` and when it is placed on line at which the break in execution should occur, the user should press `F2` which enables or disables breakpoint. Command `dc` continues execution of the debugged program until any breakpoint is met.

Radare2 is highly extendable. In has embedded HTTP server so that debugging is available with modern web browser. Radare2 must be started with specific command line option to activate this mode as in the example below:

```
> r2 -c=H ./prog
```

Then user should open web browser and navigate to address http://localhost:9090/. Such session is shown in figure 4.9.

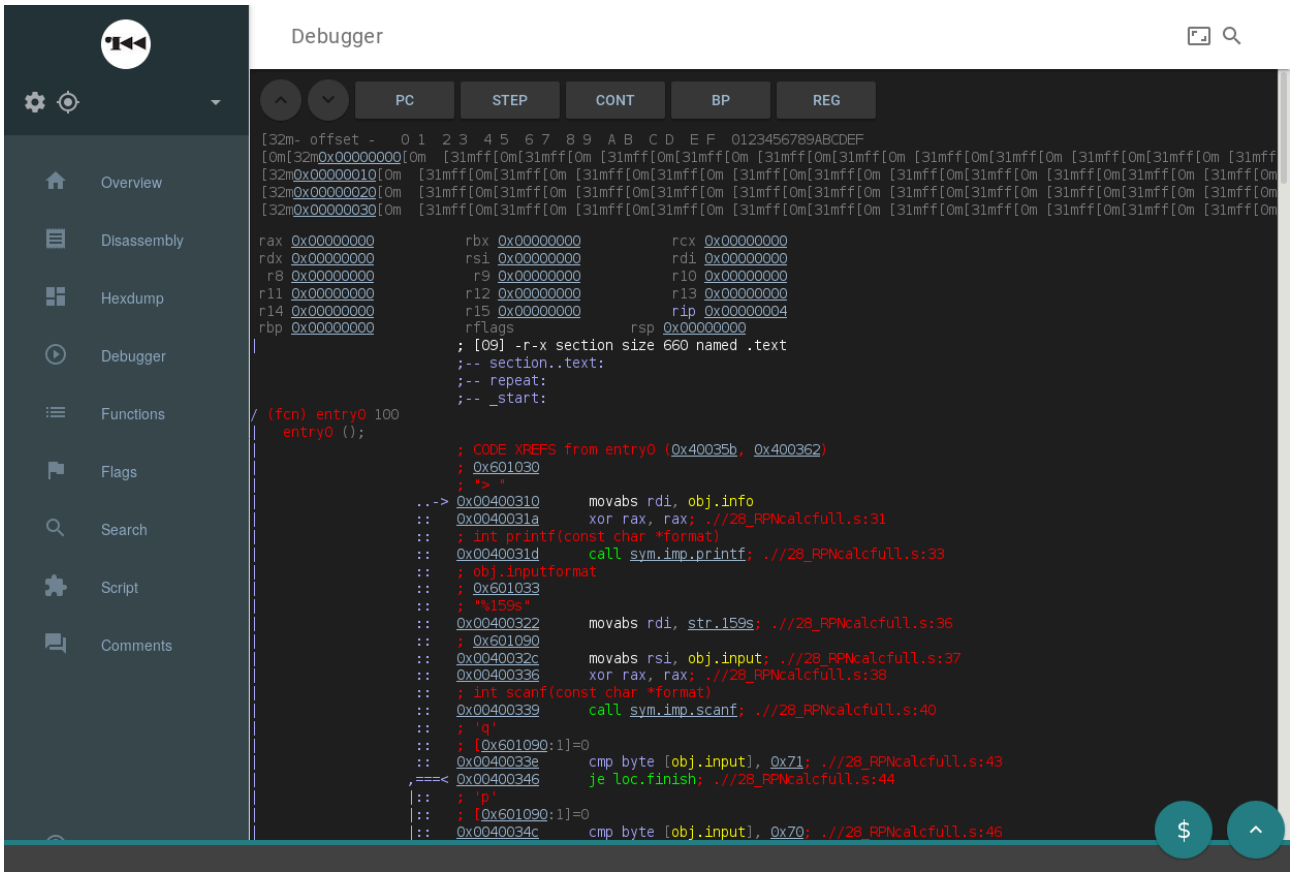

Figure 4.9: Radare2 in a web browser.

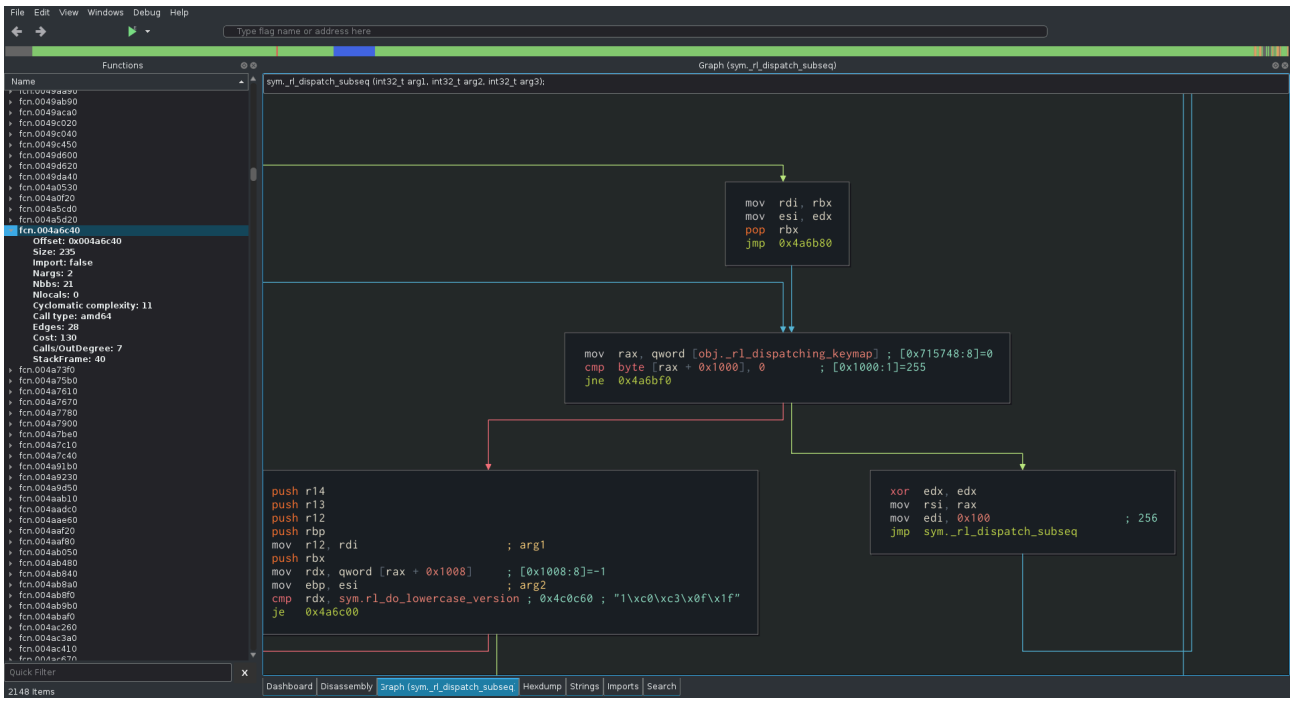

Figure 4.10: Radare2 with Cutter front-end.

For people who prefer typical desktop application but with modern GUI there is actively developed Cutter, which nicely wraps around r2. Graph view of program analyzed with cutter is shown in figure 4.10.

Version Control System

Every computer user knows that programs are versioned, usually with numbers where the higher version number the newer is an application. There is no single, standard way of software versioning. Basic way is to use single number that is good to describe major releases. But when there are improvements and patches to already existing application major and minor number are used as for example: 2.4. There might be more "layers" of changes indicated with consecutive numbers delimited with dot like: 2.6.14. If the software is as complex as full operating system then it might be quite lengthy as it can be observed in Linux:

```
> uname -vrmo
4.9.0-8-amd64 #1 SMP Debian 4.9.144-3.1 (2019-02-19) x86_64 GNU/Linux
```

To maintain versioning is responsibility of software programmer(s) or distributor. Technically it reflects development process which may be not so straightforward. Changes may bring problems, like software *bugs* and needs to be *reverted* to older, but stable stage. Sometimes there is a need to test some function before adding it to release and so one needs to make a *branch* from main code trunk. When branch is considered to be alright then it may be

merged with the trunk. Sometimes a branch never gets back to main track of development and becomes an independent application. When there is team of programmers that is working on the same source and more than one person *commits* a change to the same piece of code then these two changes make a *conflict*. This conflict needs to be *resolved* that is understood as a single, consistent change that will actually work properly. Version control system helps to maintain all these processes and more.

Over time many version control systems were developed: CVS, Subversion, Mercurial and Git. Nowadays it seems that Git is dominating probably thanks to its use in development of Linux kernel and many applications. Git itself is and free and open source software so everyone can start git servicing. Git is distributed version control system so that there is no need to have a core server, that could easily become single point of failure. Git is also distinguished thanks to GitHub, that is an acclaimed hosting service with nearly sixty millions of repositories, where half of them are free and open source. GitHub was acquired for US$7.5 billion by Microsoft in 2018.

Example how to download software from publicly available git repository is shown here:

```
git clone https://github.com/cyrus-and/gdb-dashboard.git
```

This command will create new directory to which all current files from remote repository will be downloaded.

Command-Line in Brief

Some software, especially for system administrators and programmers is distributed without GUI. Graphics might be user-friendly but are hard to port (convert) between architectures and require significant machine resources. Therefore key system tools are just command line programs due to portability, simplicity and flexibility reasons. Wrapping a command line tool with GUI or even several different types of GUI is possible and practiced often.

Even small embedded computer without screen may have command line and tools that we need to develop software. In this book we assume that some work in command line environment (CLI) with GUI applications only for code development, debugging and browsing documentation. Therefore it is worth to know a dozen of commands and several simple rules, which are discussed below to work with files and directories in CLI. Order of discussed commands is arbitrary and does reflect neither necessity nor complexity.

In following examples we assume that we start in user's home directory (/home/user) and structure of exemplary files and directories is as follows:

```
/home
 /user
  kitten.jpg
  /Desktop
```

```
   /Projects
    /Assembler
     data.bin
     info.txt
     prog.s
     prog.o
     prog
    /Python
     hello.py
```

pwd – prints current directory;

```
> pwd
/home/user
```

ls – lists files and directories in current directory; to see **l**ong list with **a**ll files and directories add optional letters after white space and hyphen; it is also possible to provide name of files or directories to be printed as a command line argument;

```
> ls
Desktop  Projects  kitten.jpg
> ls -a
.  ..  Desktop  Projects  kitten.jpg
> ls -l Projects
total 0
drwxr-xr-x 2 l l 40 Apr 23 14:59 Assembler
drwxr-xr-x 2 l l 40 Apr 23 14:59 Python
> ls -al Projects/*er
total 0
drwxr-xr-x 4 l l  80 Apr 23 14:59 .
drwxr-xr-x 5 l l 100 Apr 23 14:59 ..
drwxr-xr-x 2 l l  40 Apr 23 14:59 Assembler
drwxr-xr-x 2 l l  40 Apr 23 14:59 Python
```

cd – changes current directory; if used without arguments changes the current directory to home directory;

```
> cd
> pwd
/home/user
> cd Projects/Assembler
> pwd
/home/user/Projects/Assembler
```

mkdir – creates new directory with name given as argument;

```
> mkdir Documents
> ls
Desktop  Documents  Projects
```

rm – removes files or directories; to **r**ecursively and **f**orcibly remove directory with its subdirectories add option **-rf**; to remove more files wildcards may be used; more files or directories may be given in command line;

```
> rm kitten.jpg
> ls
Desktop  Projects
> rm -rf Projects/Python
> ls Projects
Assembler
> rm -rf Projects/Assembler/prog.o Proj*/Ass*/prog
> ls Pro*/A*
data.bin  info.txt  prog.s
```

cp – makes a copy of file given as first argument under a name given as second argument; to make a **r**ecursive copy of directory an option **-r** is used;

```
> cp kitten.jpg funny.jpg
> ls
Desktop  Projects  funny.jpg  kitten.jpg
> cp -r Projects Backup
> ls
Backup  Desktop  Projects  funny.jpg  kitten.jpg
> ls Backup
Assembler  Python
```

mv – changes name of file or directory given as first argument to new name given as second argument;

```
> mv kitten.jpg cat.jpg
> mv Projects OldProjects
> ls
Desktop  OldProjects  cat.jpg
```

touch – "touches" existing file by changing its timestamps or creates new, empty file;

```
> touch newfile
> ls
Desktop  kitten.jpg  newfile  Projects
```

chmod – changes mode of existing file; mode is set of binary flags that indicate if file is readable, writable and executable; there are three sets of these flags: one for file owner (user indicated with letter u), one for group (abbreviated as g) that is associated with particular file and one for all others (abbreviated as o) so it takes form like `rwxrwxrwx` or in octal mode 777;

```
> ls -l kitten.jpg
-rw-r--r-- 1 user user 9060 Apr 25 17:15 kitten.jpg
> chmod -r kitten.jpg
> ls -l kitten.jpg
--w------- 1 user user 9060 Apr 25 17:15 kitten.jpg
> chmod u+r kitten.jpg
-rw------- 1 user user 9060 Apr 25 17:15 kitten.jpg
> chmod 640 kitten.jpg
-rw-r----- 1 user user 9060 Apr 25 17:15 kitten.jpg
```

Reader is encouraged to experiment with commands presented above. For improved productivity it is worth to become familiar with more commands like:

- chown – changes ownership of the file
- cat – displays contents of a file;
- grep – filters files looking for specific strings using regular expressions;
- diff – compares two files, preferably textual ones;
- sort – displays sorted lines of a file;
- uniq – prints lines of file without duplicates;
- wc – counts lines, words and characters in file;
- df – displays information about disk free space, preferably with **-h** option for human-readable values;
- ps – list processes (running programs);
- kill – sends signal to process possibly ending its execution;
- man – opens manual page;

Another useful thing in CLI are redirections and pipes. Redirections change the flow of data so that output of one program might go to some file or that some file might become an input data for some program. Redirections are indicated by **>** and **<** characters where former one is for redirecting output of program to somewhere and latter one redirects data from somewhere to program's input. Pipes join output of one program with input of another program. Pipes are denoted by **|** that is vertical bar character. Here are examples:

```
> ls > files.txt
> cat files.txt
Desktop
Projects
files.txt
```

```
kitten.jpg
> grep [io][ljt] files.txt | wc -l
3
```

Having proper set of tools and just basic experience in Unix-like or Linux shell is enough to start learning assembly programming. This will be discussed in the next section.

5. Assembly Programming

We live in a society absolutely dependent on science and technology and yet have cleverly arranged things so that almost no one understands science and technology. That's a clear prescription for disaster.
Carl Sagan

Programming in assembly language shares many similarities with high level languages. It cannot be otherwise as every high level language is just an abstraction layer over the set of processor instructions that we directly use while programming in assembly language. Processor cannot execute the code directly because all that it understands are numbers. To simplify things we label some of these number in specific contexts with words or abbreviations that are easier to read hence have more meaning to programmers.

For a programmer who writes code in C language the pivotal point is the `main()` function. But from the microprocessor point of view this main function is just a labeled address in the computer memory at which some instructions can be found. Similarly, every variable, no matter how meaningful name it would have, is just a labeled memory address at which its value is stored.

In the following example a typical "hello world" program was *written* in the C language. It was then *compiled* with GCC on PC computer having GNU/Linux operating system installed. Successful compilation resulted in working *executable* program file. We will analyze steps that are taken to create executable program from the source code more closely later in this chapter.

The example program was analyzed with KDbg **debugger** as it is shown in figure 5.1. Debugger is a tool that helps to observe execution of the program in its very detailed form. The most crucial feature of debuggers is to *disassemble* numbers in computer memory back to the form with names of instructions and data which may be understand by humans. In the figure 5.1 we may see the C code interleaved with CPU instructions and their operands (arguments). At this point you do not have to worry if you do not understand it.

```
1 #include <stdio.h>
2
3 int main(void) {
  0x6b0 push   rbp
  0x6b1 mov    rbp,rsp
4         printf("Salut, Mundi!\n");
  0x6b4 lea    rdi,[rip+0x99]        # 0x754
  0x6bb call   0x560 <puts@plt>
5         return 0;
  0x6c0 mov    eax,0x0
6 }
  0x6c5 pop    rbp
  0x6c6 ret
```

Figure 5.1: Simple main() function in C language with corresponding x86_64 instructions.

Labels are stored within executable for debugging purposes but they are layer of abstraction unnecessary for electronics so they are often dropped from executables in practical situations. This cleaning process is known as *stripping* and it may significantly shrink the size of the executable file. Therefore typical program after compilation is not much more than series of instructions for CPU and some amount of data that is to be processed by these instructions. Disassembling of such programs is harder that ones with preserved debugging symbols but still possible.

In this chapter we will focus on assembly programming for x86_64 architecture that is commonly used in PC computers, Apple computers and game consoles such as Xbox One and PlayStation 4.

In 64-bit Linux programming an AMD 64 calling convention is used. List of arguments for function is passed with specific registers and in the order as it is shown in table 5.1. These registers are used to pass integer arguments and memory pointers. In case when floating-point data needs to be transmitted then **xmm** registers are used (**xmm0**, **xmm1**, **xmm2**,...). Many functions, like those from **libc** library, require the information about number of floating-point arguments passed through $\mathbf{xmm_n}$ registers. This count is provided with **rax** register. If there are no **xmm** registers in use then **rax** should be zeroed before function call. Library function is called with **call** instruction followed by address of this function that in practice is encoded with the function name (its label, like *printf*).

Table 5.1: Core of 64-bit ABI calling convention – order of function arguments.

Argument	Register
1st	rdi
2nd	rsi
3rd	rdx
4th	rcx
5th	r8
6th	r9

After function finishes its execution it may return results through registers. Main return register is **rax**, followed by **rdx**.

It should be expected that values in registers will change due to function execution because the function will operate on them. However, some of registers are callee-saved so they should be preserved across functions: **rbx**, **rsp**, **rbp**, **r12**, **r13**, **r14**, **r15**.

This calling convention is specific for x86_64 architecture. There are different registers and different calling conventions for other architectures. To see current table for all architectures supported by Linux (over 20 different processors!) one may refer to specific manual page using command:

```
> man 2 syscall
```

5.2 Numeric values of basic system calls in 64-bit Linux kernel.

Syscall name	ID
sys_read	0
sys_write	1
sys_open	2
sys_close	3
sys_stat	4
sys_fstat	5
sys_lstat	6
sys_lseek	8
sys_access	21
sys_pipe	22
sys_dup	32
sys_dup2	33
sys_nanosleep	35
sys_getpid	39
sys_exit	60
sys_fsync	74
sys_truncate	76
sys_ftruncate	77
sys_getcwd	79
sys_chdir	80
sys_rename	82
sys_mkdir	83
sys_rmdir	84
sys_creat	85
sys_link	86
sys_unlink	87
sys_chmod	90
sys_fchmod	91
sys_gettimeofday	96
sys_sysinfo	99

User applications do not directly control or communicate with hardware in modern, multi-user and multitasking operating system like Linux. Instead of that the system kernel provides functions that can perform all elemental operations like file access, process creation, timer

control and many more. These functions are known as system calls. In Linux there are over 300 system calls. The table 5.2 presents only chosen syscalls for x86_64 architecture. Number of invoked kernel function should be provided in **rax**. Linux kernel follows System V 64-bit ABI convention discussed above with exception to the 4^{th} argument. Instead of **rcx** the **r10** register is used. The kernel is invoked with **syscall** instruction. Syscall instruction does not neeed any arguments as it is configured elsewhere what should be called when CPU executes it.

If there is some less popular or non-standard hardware that needs to be controlled then the driver has to be written, which during its continuous execution becomes part of kernel, and it should provide some interface to the hardware for user programs.

To find out what specific syscall does it is advised to consult programmer's manual. For example to get information about kernel function `open`:

```
> man 2 open
```

Manual presented with this method are for C programmers but list and order of arguments are the same as in kernel syscall. It is so because the C function does not perform specific operation on its own but it invokes the kernel. Therefore list and order of arguments are consistent between the two. To quit from the displayed manual press .

To see whole list of system calls one may look into syscalls manual page:

```
> man 2 syscalls
```

Numeric values corresponding to system calls are in Linux header files. There are files like `unistd_64.h` for x86_64 architecture and similar ones for different architectures. If it is installed, for example with package management system then exemplary path is: `/ usr / src / linux-headers-4.9.0-8-amd64 / arch / x86 / include / generated / uapi / asm / unistd\_64.h`.

Another place where the list can be found is the Linux kernel itself, that is available on its website (http://kernel.org) and on Linus Torvalds github (https://github.com/torvalds/linux). The file that should be looked for is `arch / x86 / entry / syscalls / syscall\_64.tbl`.

32-bit Linux system call convention

Numbers of syscalls in 32-bit and 64-bit modes are different. Similarly to table for 64-bit system call numbers there is table for 32-bit approach. System call number is provided by **eax** register in 32-bit calling convention. Arguments are passed with **ebx**, **ecx**, **edx**, **esi** and **edi** registers. In 32-bit convention kernel call is invoked with software interrupt number 0x80 that is instruction: **int 0x80**. In this book we focus on 64-bit mode and 32-bit programs will not be discussed in details.

First Program

With remaining sections of the book we devote to practical programming on x86_64 platform with support of GNU/Linux operating system. Author of this book prefers Debian but any of its offspring such as Ubuntu, or Mint, or MX Linux might be used as well. Anyone who is already using other distribution, like: Red Hat, Fedora, openSUSE, Manjaro, Oracle, Arch, Gentoo, Tails or any other, such person should know what to do and be fine with it following examples given below.

Each section will start with simple objective that we will achieve by solving programming tasks. Now, for example we need to have a mimal working example of a program that just starts and ends properly. Such program, in file named `01-starter.asm` is really short so we may see all of it here:

```
; =====================================================
; To assemble and run:
;        nasm -felf64 01-starter.asm -o 01-starter.o
;        ld 01-starter.o -o 01-starter
;        ./01-starter
; =====================================================

section   .text
global    _start
_start:
        mov        rax, 60
        mov        rdi, 0
        syscall
```

Lines 1-6 are comments, which are started with **semicolon**. Anything in line that is after semicolon is not processed by compiler. We may put additional information for programmer in the comment. Here it is an information how to compile this code. Comment might be placed at the beginning of the line but also in the middle, after regular code. Empty lines, like line number 7 are possible and they do nothing.

Because compilation was mentioned here, let's stop at this topic for a moment and analyze the process with provided example. To compile we need to start **NASM** in command line:

```
> nasm -felf64 01-starter.asm -o 01-starter.o
```

Here NASM is informed by `-felf64` that it should compile code in the output form of **ELF64** that is Executable and Linkable Format which is supposed to be started on 64-bit architecture. Then name of file containing source code is provided (`01-starter.asm`). Finally we decide what will be the name of output file with `-o` directive. Output file should have `.o` extension.

Then linking should occur that will convert file with .o extension to an output file (hence `-o` in command line) which may be executed:

```
> ld 01-starter.o -o 01-starter
```

In Unix-like operating systems executable files have not any extensions as they are recognized due to executability flag. Program may be started by its name but as it is program in local directory and not system-wide it should employ path, that is `./` in this case:

```
> ./01-starter
```

We may analyze what is inside of each of these files with handy command file:

```
> file 01-starter*
01-starter.asm: ASCII text
01-starter: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked
01-starter.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

This is fast and easy method to find out what really is inside of a file even if it does not have extension or have a misleading one.

Above we verified that program compilation and linking are possible so it may be supposed to do what it is expected to do. Now we will analyze the remaining part of source code file to understand its behavior.

Line number 8 contains indication that specific **section** of a program starts here. Section **.text** contains instructions for the CPU. Please notice dot in the front of the section name.

```
section   .text
```

Ordinary programs have many sections although in this simple code only one was declared. More will appear automatically thanks to compilation and linking processes. Now we will analyze this aspect of the compiled program with handy **readelf** command line utility:

```
> readelf -S ./01-starter
There are 5 section headers, starting at offset 0x180:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000400080  00000080
       000000000000000c  0000000000000000  AX       0     0     16
  [ 2] .symtab           SYMTAB           0000000000000000  00000090
       00000000000000a8  0000000000000018           3     3     8
  [ 3] .strtab           STRTAB           0000000000000000  00000138
       0000000000000026  0000000000000000           0     0     1
  [ 4] .shstrtab         STRTAB           0000000000000000  0000015e
       0000000000000021  0000000000000000           0     0     1
```

```
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

You may find the .text section with which we are already familiar among other sections that were produced by compilation process. One may also use relocatable (`.o`) file obtained after compilation for **readelf** analysis instead of the final executable program. Result should be the same.

Lines 9 and 10 work together. Both are about **label** which in the assembly source code is a human-readable name for something like memory area containing CPU instruction or memory area in which variable is held. There may be many variables in the code and many spots in the code which need to be labeled. The general rule is that labels should be meaningful and that label used once cannot be used again in the same piece of code.

```
global    _start
_start:
```

From line number 9 it is known that label **_start** is declared to be **global** so it should be visible globally. Globally means that not only *inside* of this program but also from *outside*. But what does it mean *outside* in reference to the program? In this case it is simply an operating system (OS) in which the compiled program will be started. The OS needs to know where the program starts so which of its instructions should be executed first as it is the OS that lets the program to be started in the first place. Hence there is this common and predefined indicator: **_start**. However, the declaration of label global visibility is not the same as the label itself. The label is actually declared in the line 10 and may be recognized due to colon after it. Underscore in front of the label is used in "special" cases and rather should be avoided in regular labels.

We may see all labels with **nm** utility that lists all symbols from the relocatable and executable files:

```
> nm ./01-starter.o
0000000000000000 T _start
> nm ./01-starter
000000000060008c T __bss_start
000000000060008c T _edata
0000000000600090 T _end
0000000000400080 T _start
```

The main difference is clear as executable file has more symbols added by linking process. Second difference is the address at which every symbol (label) resides. In relocatable files addresses are not settled yet. These files are supposed to be able to *relocate* to different positions in the memory. If source code is raw material then relocatable file is preprocessed construction component that may be moved around. Executable file is then an established

construction which has known address. Here we see that when program starts it is placed in memory so that **_start** label points at address **0000000000400080**. Thanks to that the OS knows to which instruction it should pass execution.

Remaining lines (11-13) contain the code itself or in other words series of instructions for CPU. Only three of them are here in this program, so they should be easy enough to understand but for clarity here they are again:

```
mov         rax, 60
mov         rdi, 0
syscall
```

They must be considered as a single piece of code or a "snippet" that may be reused many times. First instruction, which is in the line 11 **mov**es value of 60 (decimal) to register **rax**. Similarly **mov** in line number 12 populates register **rdi** with 0. Finally a **syscall** instruction is performed which refers to the OS which is expected to perform an action.

In multitasking environment programs receive very small amount of time (like 1 or 10 miliseconds) in round-robin or similar manner so from the human being perspective all of them execute in parallel. Multicore CPU and properly written application also adds to this experience. The OS kernel is also piece of code that is regularly executed by CPU. There are several reasons due to which OS is invoked. One of them is voluntarily switch executed by the program which uses **syscall** instruction. Syscall is like a request from the program to the OS to perform some action. Program on its own is very limited in actions as **all** hardware is *usually* controlled by the OS or hardware drivers. Hence, even to finish the program is a work for the OS.

Functionally lines 11-13 are finishing the program by asking the OS to clean up the space after it and remove it from multitasking round-robin queue. The OS knows that this specific operation is required due to value in **rax** register when it is invoked by **syscall** instruction. So we *could* remember that 60 is kernel function that finishes the program which used it. Fortunately we do not have to memorize kernel function numbers. We may refer to table such as the table 5.2. Or even better we may place something like a table in the file alongside with the source code and let the compiler use it. We will learn this technique in the next section.

Value in register **rdi** is returned to the shell which may decide upon it, whether the program executed and finished properly or it malfunctioned for some reason. Here 0 is placed in **rdi** as it follows the most common approach to the meaning of returned values:

- 0 – properly finished,
- 1 – malfunctioned.

To sum up this particular snippet is like a well known **return 0** in the otherwise empty `main()` function in C programming language.

Thanks to above example reader should:

- become familiar with structure of assembly code used with **NASM** compiler,

- understand meaning of code section,
- understand role of labels,
- understand the process of code preparation, compilation and linking,
- be familiar with relation between program and the OS by the **syscall** instruction,
- become familiar with **readelf** and **nm** tools.

Writing Text with Kernel Function

Next step is to have some feedback from the program. The simplest way is that it displays some text onto the screen. However, in fact the program does not know what the "screen" is! It outputs data to one of file descriptors it has available. These descriptors are like handles primarily to existing files. So the program needs to have an opened file before it will write text somewhere. Well, that is easy to comprehend but how to open the display?

When every program starts it receives three file descriptors from the operating system:

- standard input, typical symbolic name STDIN, with number 0
- standard output, typical symbolic name STDOUT, with number 1
- standard errors output, typical symbolic name STDERR, with number 2

Therefore to write onto the screen usually means to write onto the standard output, with graphical desktop it will be the terminal window in which the program was started. We do not have to care how graphical desktop works or even how this terminal works. They just do thanks to layers of software and libraries that simplify our problem a lot.

The source codes we are going to analyze will get longer and longer in following sections. So they could look rather cryptic if we would see them as a whole. Therefore they will be analyzed in parts, similarly to what was done in previous section. Name of the source file is `02-writingtext.asm`.

Beginning of the next source contains comment with information about compilation:

```
; ================================================
; To assemble and run:
;        nasm -felf64 02-writingtext.asm -o 02-writingtext.o
;        ld 02-writingtext.o -o 02-writingtext
;        ./02-writingtext
; ================================================
```

We may see that there are no changes to the compilation and linking method.

In next lines we see something new, that is a compiler directive:

```
%include "include/consts.inc"
%include "include/syscalls_x86-64.inc"
```

We can recognize directive by the fact that it starts with sign. This one informs compiler that it should supersede line of code with contents of specific file so in other words to *include* it.

The first included file is which several lines are shown here:

```
; =====================================================
; consts.inc
; Constants used in examples.
; =====================================================

%define STDIN        0
%define STDOUT        1
%define STDERR        2
```

We see more compiler directives here, all starting with . These *define* that specific piece of text will be treated by compiler like it was specific number. So we do not have to remember that standard output corresponds to value of 1. We may use STDOUT instead, which should be much easier to remember and make the code easier to understand as well.

Second included file contains definitions thanks to which we may forget about numbers of syscall functions. Here are several lines of it:

```
; =====================================================
; syscalls_x86-64.inc
; Linux syscall numbers for x86-64 architecture.
; =====================================================

%define sys_read                        0
%define sys_write                       1
%define sys_open                        2
%define sys_close                       3
%define sys_newstat                     4
```

This file is lengthy with over 300 syscalls defined there. So from now on we may use descriptive names like `sys_exit` instead of memorizing and using some magic numbers.

Next lines of source file contain beginning of `.text` section that we are already familiar with:

```
section   .text
global    _start
_start:
```

They are followed by piece of code that is supposed to write message onto the screen:

```
        mov rax, sys_write
        mov rdi, STDOUT
        mov        rsi, buf
```

```
        mov         rdx, count
        syscall
```

As this fragment contains some new things it will be discussed in details. In general it is a snippet that ends with `syscall` instruction so it refers to kernel function. Therefore registers carry information what should be done. To **rax** a number of kernel function should be moved but we advanced with our approach and now we use symbolic names like `sys_exit` which are defined in one of included files. Then register **rdi** is filled with a reference to STDOUT. Then a reference to labeled message that is somewhere in the memory is moved to register **rsi**. Finally register **rdx** should contain information about the length of the message so kernel will know how many characters it should print to the output. All of these symbolic names in the executable program will be regular numbers but we can rely on compiler in this matter.

To find out why this order of arguments is used one should compare 64-bit ABI, or simply the table 5.1 with manual for system function `write()` that is available from command line:

```
> man 2 write
```

In the manual there should be line like this:

```
 ssize_t write(int fd, const void *buf, size_t count);
```

We see the order of arguments in the C function corresponds to order of arguments that was used in the assembly code and 64-bit ABI. Name of the function `write()` resembles `sys_write` kernel call. There is integer argument `fd` that relates to **rdi** register filled with file descriptor (fd) equal to `STDOUT`. Then there is a pointer to buffer named `buf` which has the same name in the assembly code and is put into the register **rsi**. Finally variable `count` provides to the `write()` function an information about length of the data that should be put onto the `STDOUT` just like it happens in assembly code by the register **rdx**.

Next lines we are quite familiar with as they finish the program properly:

```
        mov         rax, sys_exit
        xor         rdi, rdi
        syscall
```

However, there are little differences if compared with example given in section 5.1. Firstly there is no magic value moved to register **rax** any more. Constant `sys_exit` defined in included file is used instead.

Secondly the **rdi** register is not populated with 0 but there is another instruction used: **xor**. If you remember section 1.1 then you should recognize the fact that doing XOR operation on some argument with itself will result in 0. Always. So this idiomatic assembly instruction zeroes register **rdi** that in this case is both an argument and destination of the result. It is faster than moving lengthy (65-bit) value, occupies less memory and is easier to recognize as there is no doubt of the operation result. Moving 0 to register may rise questions in the future like: should it be really 0 or perhaps it should be 1 in fact?

Finally the familiar **syscall** closes the second snippet present in this code.

But it is not the end of the source file yet as there are several lines more:

```
section   .data
        buf:                db                  "Salut, Mundi!", 10
        count:                equ                   $-buf
```

This is declaration of **.data** section that obviously hold data such as message that is to be printed. The message is labeled as buf and consists of bytes that are denoted with **db** keyword (**d**ata built of **b**ytes). Texts in ASCII are just bytes therefore **db** fits the purpose. Text is written within quotation marks followed by comma and number 10.

Data structure may be just series of numbers separated by comma but whenever we can write text it is easier to be done like it was done here. However, some characters are non-printable as they have no visible glyph on the screen. They make some action on the screen like `Backspace` or `Enter`. For these characters we must refer to their ASCII values if we want them in the message. Value 10 is "new line" so it is like pressing `Enter` at the end of line. More about ASCII codes can be found in manual page:

```
> man ascii
```

Last line of code defines new label `count` that points to value that is equal to something. Hence `equ` reserved word is used here. `Count` is calculated on-the-fly during compilation and the expression `$-buf` means that from "this" memory address should be subtracted memory address where `buf` starts. "This" memory address is just after the message `buf` so the calculated difference is between the end and the beginning of the message that is to be printed so it is the message length. We do not have to worry about the actual length of the message, we may change it some other day and do not have to recalculate its length manually. All of it is done by compiler provided that the line with calculation (this `equ` value) is placed just after the message of which the length we need.

Thanks to above example reader should:

- become familiar with symbolic names that may be defined in the code,
- know the technique of common, reusable include files,
- understand how and why to avoid cryptic magic numbers with definitions of constants,
- become familiar with kernel function write,
- understand relation between C functions, 64-bit ABI and registers,
- know the **.data** section,
- know how to create memory area in the form of bytes,
- know how to calculate length of piece of memory area on-the-fly.

Data Input with Kernel Function

Single-sided communication is rather boring so in this section we will analyze program that also takes some information from the user and works with it. Source file is named `03-interact.asm`.

First ten lines of code contain comment with information about compilation and two lines that include necessary files with defined symbolic names. So it is exactly like it was in previous programs. Therefore we will omit this part here and skip to next line:

```
%define         USERNAMEMAXLEN          30
```

This is simple definition of constant that will be used in more than one place. If some value is to be used more than once then it is wise to define it once and then use the symbolic name later. If it were necessary to change the value there would be only one spot in which the change should occur instead of many. Such approach minimizes chance for errors, makes code more legible and simplifies its future maintenance and development.

Then comes **.data** section with two phrases that this program will print to the terminal:

```
section   .data
        buf:                db                "Hello, what's your name?", 10
        count:                equ                $-buf
        buf2:                db                 "I am pleased to meet you "
        count2:                equ                $-buf2
```

Firstly, we should notice that a label cannot be reused so that there are two pairs: `buf` and `buf2`, `count` and `count2`. Probably it would be better to rename these strings with something more informative like: `helloS` and `greetS`. Capital "S" as the suffix of variable name would indicate that this is textual string. I leave it as exercise for readers.

Secondly please observe that only one of these strings ends with "new line" character. The second one will let some other string join to its end in the same line.

Finally you may remember that in previous example **.data** section came after the `.text` section. Well, the order of sections in the source is not that important as the compiler and linker must setup the memory configuration anyway.

With next lines new section is introduced:

```
section .bss
        username:         resb         USERNAMEMAXLEN
```

BSS stands for "Block Started by Symbol" and it is very old name for uninitialized memory area acquired for the program on its start. This **.bss** section indicates reserved but uninitialized memory space. In this memory there may be labels thanks to which access to the section is possible. In this example `username` is the label. Then comes a keyword `resb` that corresponds

to `db` from previous example as it reserves specific number of bytes. Here is the first spot in which constant `USERNAMEMAXLEN` is used. So 30 bytes are reserved for future use.

In practice of Unix-like programming it may be expected that the **.bss** section will be filled with zeros. However, it is not guaranteed. On many platforms this section is uninitialized and may contain some random data when the program starts. Some older systems even leaked sensitive data, like passwords, due to lack of proper initialization of **.bss** section!

After both memory sections are declared in the source then comes **.text** section with its instructions, exactly like in the previous example:

```
section   .text
global    _start
_start:
        mov rax, sys_write
        mov rdi, STDOUT
        mov        rsi, buf
        mov        rdx, count
        syscall
```

The above piece of code should be clear as it simply prints message “Hello, what’s your name?” onto the screen.

Program encouraged to enter user name and waits for that with the next snippet:

```
        mov rax, sys_read
        mov rdi, STDIN
        mov rsi, username
        mov rdx, USERNAMEMAXLEN
        syscall
```

Similarity between writing and reading should be obvious. In the first case we are writing to `STDOUT` `count` number of bytes that are in the `buf` area in the **.data** section. In the second case reading occurs on `STDIN`. This is also the next spot in which `USERNAMEMAXLEN` is used to indicate number of characters (bytes in fact) that may be are stored in the `username` buffer, in the **.bss** section. If in doubt about the arguments order please consult manual for `read()` function.

In the manual it can be also found that after successful execution the `read()` function returns number of bytes that were obtained. Values are returned from kernel functions by the `rax` register. However, **rax** register is used very often and the program will continue so this number should be safely stored somewhere else, like **rbx** register:

```
        mov rbx, rax
```

If you wonder why value was stored particularly in **rbx** register then you may refer to discussion of AMD64 calling convention and list of callee-saved registers. Several registers are supposed to be protected from change in called functions. We will analyze how these functions

prevent modification of these registers in later sections of the book. Now we may assume that if our code does not affect these registers deliberately then values in them are safe.

Next piece of code simply writes the second message onto the screen – “I am pleased to meet you ” without new line character:

```
mov rax, sys_write
mov rdi, STDOUT
mov rsi, buf2
mov rdx, count2
syscall
```

There is not much to add here as it almost a duplicate of previous snippets that write message.

Next piece of code also writes a message onto the standard output but differs a little from previous invocations of kernel `sys_write`:

```
mov rax, sys_write
mov rdi, STDOUT
mov rsi, username
mov rdx, rbx
syscall
```

Lines 44 and 45 do not bring anything new. Line 46 uses label `username` that points to memory area in **.bss** section. We may expect that previous invocation of kernel `sys_read` performed correctly and some text is available in that place. Last information the kernel needs is number of bytes that it should print. User may enter texts of different lengths so it is important to store number of entered bytes as this information may is necessary now. Therefore value from `rbx`, which preserved this number of bytes through all **syscall** invocations, is retrieved now and put to **rdx** register, where it is expected to be.

Remaining part of code uses `sys_exit` exactly in a way as it was discussed in previous sections so its discussion will not be repeated here.

Session with this program may look like this:

```
> ./03-interact
Hello, what's your name?
Brave Little Penguin
I am pleased to meet you Brave Little Penguin
```

Summing up this section:

- emphasized the benefits of defined symbolic names instead of magic numbers,
- introduced **.bss** section and idea of uninitialized memory area,
- provided basic method of interaction with computer programs thanks to `sys_read` kernel function.

Functions and Stack

Next program is much lengthier than previous ones. It uses more kernel functions, but we are already familiar with this technique so it should be fairly easy to follow. However, there is also more complex topic introduced which is **stack**. Thanks to familiarity with stack we may write and use our own functions. Source discussed in this section is named `04-functions.asm`. In first lines it has comment with information about compilation and include directives, just like the previous examples, so we may skip to more interesting fragments.

Then comes a local definition of constant value and **.bss** section where it appears:

```
%define        SYSINFOLEN        128

section .bss
        sysinfobuf:                resb        SYSINFOLEN
```

This value is used just once in the code but such approach prepares the code for future expansions which could rely on this number. Prepared memory area has length of 128 bytes. It will be filled by one of kernel functions.

There is no **.data** section in this program and the next one is **.text** of which several lines we may see here:

```
section   .text
global    _start
_start:
        call mkdir
        call chdir
        call prepareinfo
        call saveinfo

        mov        rax, sys_exit
        xor        rdi, rdi
        syscall
```

It begins in a way that we are used to with the `_start` reserved label. It ends with familiar invocation of `sys_exit` kernel function. But in between there are instructions we did not see yet: `call`.

Syscall instruction passes execution to the kernel and the CPU "just knows" where the kernel resides in memory. How it is achieved is beyond the scope of this book. Similarly works `call` instruction but it remains in the user space of the code (as opposite to the kernel space) and it refers to label. We know that any instruction may be labeled and that labels are just pointers to some memory address. Hence `call` instruction passes execution to some other piece of code with given label.

Functions make code structured so easier to write, read and maintain in the future.

Furthermore functions may be reused across many program, provided that they are written and prepared properly.

The chunk of code shown above consecutively calls four functions: `mkdir`, `chdir`, `prepareinfo` and `saveinfo`. Each of these functions is piece of the discussed program as it resides in the **.text** section so we will analyze them one by one.

So the function `mkdir` starts this way:

```
mkdir:
        mov rax, sys_mkdir
        mov rdi, [rsp + 24]
```

In this piece of code there is a label `mkdir`, which we have seen already and one **mov** instruction. Obviously, **rax** register will contain number of system call that should create new directory in the filesystem. Next instruction moves some value to **rdi** register from a bit mysterious expression `[rsp + 24]`. Here we have to stop for a moment and discuss what stack is and how it works.

Stack is memory structure onto which some data may be put and may be retrieved from. Stack is often compared to pile of papers: we may put a piece of paper on top of it and get one piece of paper from the top. However, stack is implemented in a way that **it grows downward** memory addresses. Possibly it is better if you imagine a stalactite that is attached to the cave ceiling and over time it grows downward. We may also shorten it by removing some piece of it.

These two operations on the stack in computer memory are available thanks to **push** and **pop** instructions. Both of these instructions use argument which is the source in former case and destination in latter case for data that is moved to or from the stack. Information about top of the stack is stored in **rsp** register so **rsp** register is **decreased** every time **before** the **push** instruction is performed. The value by which **rsp** changes depends on architecture: in 32-bit it will increase by 4 while in 64-bit it will increase by 8. It means that every **push** and **pop** may work on only single unit of data that corresponds to architecture. Counterpart instruction **pop** reverses the push operation so it **increments rsp** by constant value **after** it retrieves 4 or 8 bytes from the stack, depending on architecture.

In between **.text** section and the **stack** there are **.bss** and **.data** sections with their known sizes. On top of them there is `heap` that grows upward like a stalagmite and so towards the stack. It is memory area that is acquired on demand, during execution of the program thanks to functions like `malloc()` well known in C programming language.

Following the cave metaphor we may say that the cave floor is the basis for a program when it loads. However, cave floor is not the maximum possible depth as we could dig deeper, possibly to the Earth core. Hence our program is not loaded at the address 0x0 but at something like 0x400080 as we observed when we analyzed the address of `_start` label in section 5.1. On the other hand cave ceiling does not limit dimensions in the other direction. So at addresses above the **stack** there is **kernel** space of the operating system.

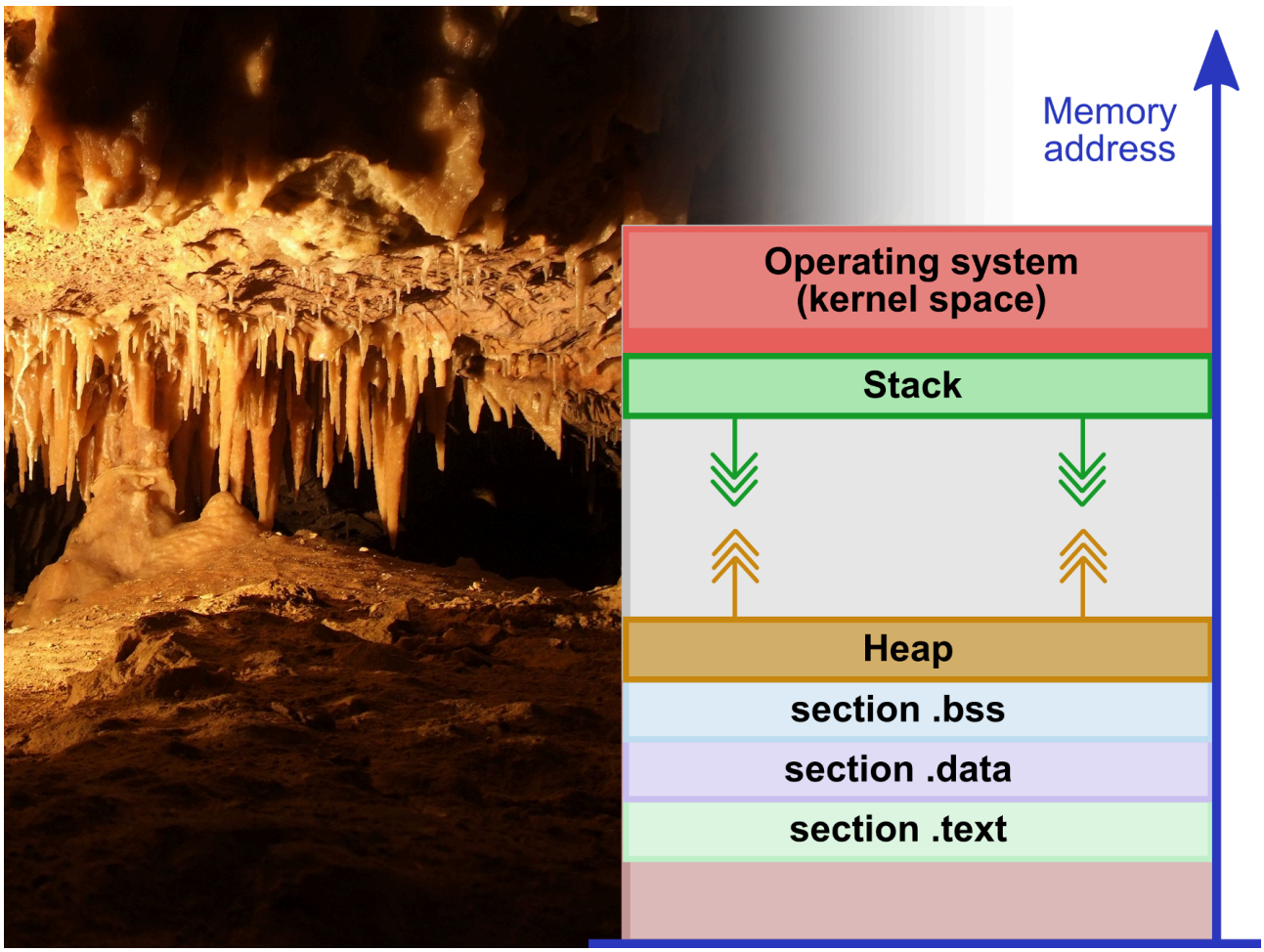

Figure 5.2: Memory structure on the background of metaphoric cave. Photo: Andrew McMillan, license: CC0.

The memory structure discussed above is shown in figure 5.2 together with its cave metaphor. Now as we understand how stack works in reference to other sections we may get back to the instruction due to which this discussion started:

```
mov rdi, [rsp + 24]
```

Moving values with **mov** instruction does not destroy data in the source. It is unlike with physical objects, such as stalactites, where once we take a piece of it then this part is actually removed. Therefore when we use stack we may copy data from it without taking it away as **pop** instruction does. The right-hand expression in the above line of code represents **relative addressing** of memory. Register **rsp** is not a direct value holder but is a pointer to memory as it is general purpose of **rsp** to point to the top of **stack**. Furthermore this memory pointer is incremented by 24 so it points to the memory address of top of the stack plus 24 bytes. But why do we add 24 and not 16, or 32, or some other value?

When **call** instruction is performed then it pushes onto the stack address of the next instruction just beyond it. It is named **return address**. Thanks to this it is possible to *return* from the called function back to the calling piece of code (which also may be a function!).

Therefore if we used relative addressing by adding +8 to **rsp** we could skip this return address as it would move pointer upward the stack – to values that were placed there earlier.

Next value on stack is number of command line arguments provided to the program. Operating system knows this value and puts it there automatically. It is like in C programming where we have:

```
int main(int argc, char * argv[]);
```

This `argc` variable is actually number of command line arguments provided to the program and is obtained from the stack. So as `argc` is obtained from the stack then maybe the same is about vector of pointers to strings known as `argv` table? Yes, it is! Perhaps you remember that `argv[0]` in C is the reference to name of the program with which it was started from the operating system. Therefore `rsp+8` points to number of arguments, `rsp+16` points to `argv[0]`, `rsp+24` points to `argv[1]` and so on...

So we found out that first command line argument of the program is available in the `mkdir` function under the address that is stored at `rsp+24`. If the architecture is 64-bit then stack can hold only 8 byte numbers. Therefore it may be only an address of the text that was provided through command line argument but not the text itself, which in general may be longer than 8 bytes (ASCII characters). Thus we need to get to the string by *dereferencing* it, just like asterisk symbol (*) does in C programming. In assembly language if we want to get to memory address pointed by some register or an relative addressing expression we indicate this request by rectangular brackets: `[rsp + 24]`. Thanks to this function `mkdir` will be able to create a new directory with name provided to the program through command line.

Remaining lines of the function look like this:

```
mov rsi, S_IRWXU | S_IRGRP | S_IXGRP
syscall
ret
```

If we consult manual of `mkdir` system function we would see that it requires two arguments: pathname made of characters and `mode`. Mode is simply set of rights that are established for new directory or file when they are created. With this example here the directory will be readable, writable and executable by the user (S_IRWXU), readable by group members (S_IRGRP) and executable by group members (S_IXGRP). Executability flag in terms of directory enables listing of its contents. Values that are represented by these constats are provided in . They are summed up logically with `OR` operation that in **NASM** is denoted with vertical bar `|`. In terms of Unix-like rights directory will be described by sequence `rwxr-x—`. It means that owner of the directory (usually it is the user who created it) may enter it, add new files and delete existing ones there, list directory contents and possibly delete it completely. Members of group to which the directory belongs may only enter the directory and see its contents. Other users will not be able to do anything with this directory.

Function finishes with **ret** instruction that retrieves address from top of the stack and

changes instruction pointer register **rip** to this value so it *returns* from the function `mkdir` to the main part of the program.

Next function invoked from the main part of the program is `chdir`:

```
chdir:
        mov rax, sys_chdir
        mov rdi, [rsp + 24]
        syscall
        ret
```

It changes current working directory to directory with given name, which is the only argument of `sys_chdir` function. Here the same directory name is used as it was used in previous function. Therefore program should create new directory and enter it. This piece of code is function hence it ends with **ret** instruction.

Next function obtains some information about operating system:

```
prepareinfo:
        mov rax, sys_sysinfo
        mov rdi, sysinfobuf
        syscall
        ret
```

This function invokes `sys_sysinfo` kernel function which populates memory area which address is given as its only argument with some information about the computer. In this example `sysinfobuf` discussed previously is used as the destination for that data.

Last function called from the main program is `saveinfo` that uses two system invocations where the first one creates new file:

```
saveinfo:
        mov rax, sys_open
        mov rdi, [rsp + 32]
        mov rsi, O_CREAT | O_WRONLY
        mov rdx, S_IRUSR | S_IWUSR | S_IRGRP
        syscall
```

File is both opened and created with `sys_open` kernel function. First argument of this kernel function, as you may see thanks to manual page of open function, is the name of a file. Here it should be provided as another argument on command line so it is available under address `[rsp + 32]` similarly to first command line argument which contained name of created directory. This address of string is moved to **rdi** register. Register **rsi** corresponds to `flags` from the manual so logical sum of `O_CREAT` and `O_WRONLY` is put there. It is easy to guess that first symbolic name indicates programmer wish to create file (if it does not exist yet) while the second one states that file should be opened (after creation) for writing only. Finally in `rdx` register mode is passed which in this case make the file available for reading and writing by

the owner and only for reading to other members of the group to which it belongs. In Unix-like style it would be a sequence: `rw-r—`. Executability flag is turned off because the file is not a program.

Remaining part of `saveinfo` function uses `sys_write` kernel call with which we are already familiar:

```
mov rdi, rax
mov rax, sys_write
mov rsi, sysinfobuf
mov rdx, SYSINFOLEN
syscall
ret
```

However, the difference between this example of writing and previous ones is that it "prints" bytes to a file, the one that was just created and opened. File descriptor of that file is available in **rax** registers just after previous syscall is finished as through **rax** register kernel returns function results. Hence before **rax** gets populated with `sys_write` number its contents (the file descriptor number) must be stored in the **rdi** register. It is very important to remember that order of function arguments and order in which registers are populated do not have to be the same.

This ends the program execution and new directory, with new file should be created. As a bonus we will analyze part of the resulting file.

```
> ./04-functions newdir datafile
> ls -l newdir/
total 4
-rw-r----- 1 l l 128 May 14 17:24 datafile
> xxd -c 8 newdir/datafile
00000000: 1c71 0000 0000 0000  .q......
00000008: 2026 0000 0000 0000   &......
00000010: a027 0000 0000 0000  .'......
00000018: 202a 0000 0000 0000   *......
00000020: 0070 dcf2 0100 0000  .p......
00000028: 0010 78da 0000 0000  ..x.....
00000030: 0030 bf12 0000 0000  .0......
00000038: 0090 b80b 0000 0000  ........
00000040: 00f0 affb 0100 0000  ........
00000048: 00f0 affb 0100 0000  ........
00000050: 1a03 0000 0000 0000  ........
00000058: 0000 0000 0000 0000  ........
00000060: 0000 0000 0000 0000  ........
00000068: 0100 0000 0000 0000  ........
00000070: 0000 0000 0000 0000  ........
```

```
00000078: 0000 0000 0000 0000  ........
```

File has binary contents hence it is pointless to see it with text editor. Instead a tool such as **xxd** that presents hexadecimal data is more suitable. Structure of this data is shown in manual of `sysinfo`.

From the manual it is known that at the beginning of this data structure there is value of type `long`, so 8 bytes on the 64-bit machine, that hold "uptime". Uptime is time that passed from the system boot up. We should remember that little-endian is used so all bytes should be written in reverse: `0000 0000 0000 711c`. Value 0x711c converted to decimal equals to 28956 decimal. This value is in seconds so after simple calculation we know that computer was started more than 8 hours ago.

Furthermore 5^{th} `long` value (counting from 1) is amount of computer memory (totalram). Here it is at **offset** 00000020 and it equals to 0x00000001f2dc7000 so 8369500160 decimal. Therefore this experiment was performed on computer which has 8 GB of RAM.

Reader is encouraged to further analyze the contents of the resulting file to practice usage of binary data, hexadecimal numbers and memory offsets.

After this section reader should:

- be introduced to topics of stack and memory organization,
- be familiar with relative addressing and dereferencing pointers,
- understand relation between stack and command-line arguments,
- know how to create new directory and file using kernel functions,
- know how to open a file and save some data array into that file,
- understand the role and how to use file flags and permissions,
- start feeling the flow of code execution made of functions.

Jumps and Branching

So far discussed examples were relatively simple, linear sequences of instructions. But in real programming regularly a decision must be made on the basis of data that is unknown until program execution. Data is usually compared with some known threshold and outcome of the comparison often takes form of simple answer like "below", "above", "equal", "not equal", "not below", or "not above". Furthermore some operations should be repeated many times until some event happens. Example of such situation is when user interacts with the program until the operation that finishes the program is requested by that person.

Branching enables alternative execution path in the program. It may occur in situations where one of branches is chosen upon the other because of information that is not known during compilation of the code. For example kernel functions like `sys_open`, `sys_read`, `sys_write` may fail to execute properly for many reasons such as: user may have no rights to open a file, there may be no space on the disk left and so on. In such situation usually a

negative value is returned via **rax** register to the code that invoked the kernel function. So far such erroneous situation was ignored in our sample programs but it should be no more.

Next example is based on source file named `05-errorcontrol.asm`. Beginning of this file is similar to previously discussed `04-functions.asm` with comment in the role of header followed by sections **.bss** and **.text**. Code from which execution of the program starts is changed a little:

```
_start:
        call openfile
        call prepareinfo
        call saveinfo
        xor         rdi, rdi
quitprog:
        mov         rax, sys_exit
        syscall
```

We can expect that this program opens a file, acquires system information and then saves this data block in the already opened file. All these three activities are performed in separate functions which all are invoked with **call** instruction. Therefore all three of them should end properly with `ret` instruction.

I am sure that you understand the difference between **call** an **syscall** but lets make it clear:

- **syscall** – invokes the **kernel**, wherever it is in the memory,
- **call** – invokes a function which resides in **user space** that is positioned at memory address given by label that is the only argument of the instruction.

New aspect is also that **rax** is populated with `sys_exit` value after zeroing of the **rdi** register and is labeled as `quitprog`. We are used to opposite order of these two instructions but the order in which registers are prepared with their specific values does not matter from the **syscall** perspective. Then one may wonder why it is done in such particular order and it should reveal itself due the following discussion.

Now we should analyze function that opens a file:

```
openfile:
        mov rax, sys_open
        mov rdi, [rsp + 24]
        mov rsi, O_WRONLY
        mov rdx, S_IRUSR | S_IWUSR | S_IRGRP
        syscall
        call verifyresult
        mov rbx, rax
        ret
```

If we compare this snippet with use of `sys_open` kernel from the previous example we should find that now to **rsp** only 24 is added. Register **rsp** points to top of the stack where return address is. Above it at address **rsp+8** there is number of command line arguments (argc). At **rsp+16** there is first argument that is the name of the program (argv[0]) so finally **rsp+24** points to first argument provided to the program via the command line. So program tries to access a file which name is given in its command line as the first argument. Potential file descriptor should be returned by the **syscall** through **rax** register.

After the syscall there is **call** to `verifyresult` function that checks if an erroneous situation occured. This function is used more than once so it will be discussed a bit later. When execution returns from the `verifyresult` then the value returned from system function is still present in **rax** (we known it thanks to examination of the `verifyresult` function) and is preserved in callee-saved register **rbx** for future use.

Second difference in reference to previous example is that the `O_CREAT` flag is not present so this program will not attempt to create new file. In other words it only tries to open file that is assumed to exists in the filesystem already.

Next function is labeled as `prepareinfo` and it is quite simple:

```
prepareinfo:
        mov rax, sys_sysinfo
        mov rdi, sysinfobuf
        syscall
        call verifyresult
        ret
```

This snippet is known from previous example except of the fact that now after the **syscall** it calls `verifyresult` function.

Finally if all went correct so far then program tries to save the data structure to a file which descriptor was held all this time in **rbx** register:

```
saveinfo:
        mov rax, sys_write
        mov rdi, rbx
        mov rsi, sysinfobuf
        mov rdx, SYSINFOLEN
        syscall
        call verifyresult
        ret
```

If file was opened successfully then **rbx** preserved its descriptor and now this value may be moved to register **rdi**, where it is expected to be by the kernel function `sys_write`. Exactly like in previous functions also in this one there is call to `verifyresult` function after the **syscall**.

Finally we should look at the function `verifyresult`:

```
verifyresult:
        cmp rax, 0
        jl        badresult
        ret
badresult:
        mov rdi, EXIT_FAILURE
        jmp quitprog
```

Because this function is called immediately after **syscall** then **rax** should contain value returned by the invoked kernel function. So that it compares **rax** register against 0 by using **cmp** instruction. According to difference between both values specific flags in the CPU will be set and due to states of these flags **conditional jumps** like **jl** used here ("jump-if-less") may happen or not. When condition is met then execution flow goes to the address given by argument of the conditional jump instruction. Otherwise next instruction after the conditional jump is executed.

So here if kernel function failed then **jl** will happen. Execution will step over the solitary **ret** instruction and will go to instruction labeled as `badresult`. This instruction prepares **rdi** with value of 1 that is hidden under the symbolic name `EXIT_FAILURE` and jumps unconditionally to the code where `sys_exit` is invoked.

Keeping all possible execution paths in mind may be problematic hence it is good moment to use tool with graphical representation of the code. Program that has this feature: radare2 and its front-end – cutter were discussed in section 4.3.4. Due to limited size of this book it is not possible to show all codes with this method but sample snapshot of code discussed in current section is shown in figure 5.3.

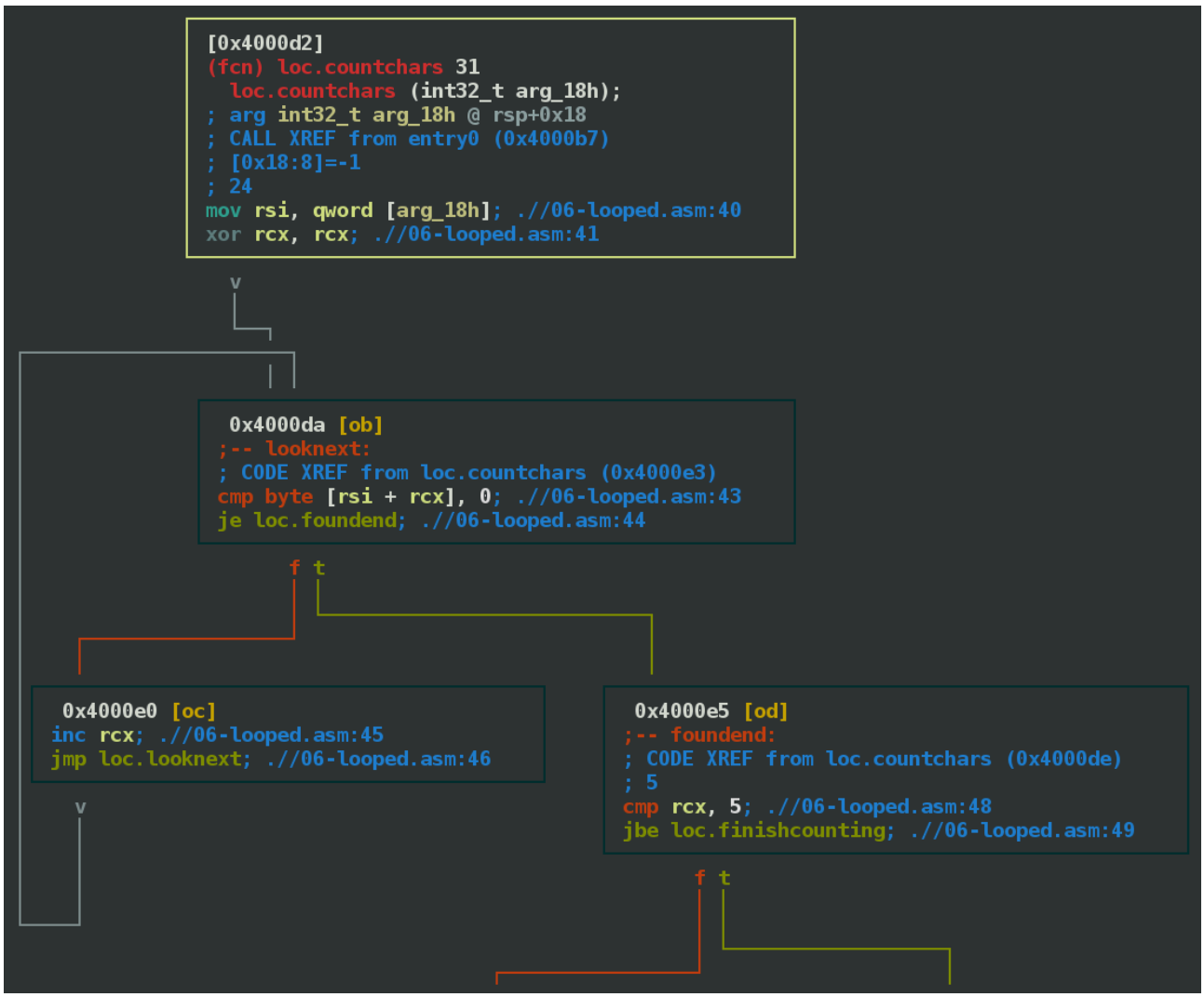

Figure 5.3: Analysis of branching in Radare2 graphical disassembler.

This program may be written in different way: invocation of `sys_exit` could be in the piece of program labeled as `verifyresult`. This modification is left for practice to the reader. Second exercise is to modify the code so that instead of **jl** the **jge** (jump-if-greater-or-equal) conditional jump would be used.

Table 5.3: Flags and register related jump instructions.

Instruction	Description	Condition
JMP	Unconditional jump	–
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE	Jump if equal	ZF = 1
JZ	Jump if zero	ZF = 1
JNE	Jump if not equal	ZF = 0
JNZ	Jump if not zero	ZF = 0
JP	Jump if parity	PF = 1
JPE	Jump if parity even	PF = 1
JNP	Jump if not parity	PF = 0
JPO	Jump if parity odd	PF = 0
JCXZ	Jump if **cx** register is 0	CX = 0
JECXZ	Jump if **ecx** register is 0	ECX = 0
JRCXZ	Jump if **rcx** register is 0	RCX = 0

Table 5.4: Jump instructions for operations on unsigned values.

Instruction	Description	Condition
JB	Jump if below	CF = 1
JNAE	Jump if not above or equal	CF = 1
JC	Jump if carry	CF = 1
JNB	Jump if not below	CF = 0
JAE	Jump if above or equal	CF = 0
JNC	Jump if not carry	CF = 0
JBE	Jump if below or equal	CF = 1 or ZF = 1
JNA	Jump if not above	CF = 1 or ZF = 1
JA	Jump if above	CF = 0 and ZF = 0
JNBE	Jump if not below or equal	CF = 0 and ZF = 0

Table 5.5: Jump instructions for operations on signed values.

Instruction	Description	Condition
JL	Jump if less	SF $\neq$ OF
JNGE	Jump if not greater or equal	SF $\neq$ OF
JGE	Jump if greater or equal	SF = OF
JNL	Jump if not less	SF = OF
JLE	Jump if less or equal	ZF = 1 or SF $\neq$ OF
JNG	Jump if not greater	ZF = 1 or SF $\neq$ OF
JG	Jump if greater	ZF = 0 and SF = OF
JNLE	Jump if not less or equal	ZF = 0 and SF = OF

Flags Register and Conditional Jumps

It was mentioned that there are lot of possible outcomes from simple comparison with **cmp** instruction. These outcomes are signaled with **FLAGS** registers which contains flags such as:

- ZF – zero flag – set to 1 if recent operation resulted in zero,
- OF – overflow flag – set to 1 if recent (mathematical) operation overflown, which means that its result is too large a positive number or too small a negative number (excluding sign bit) to fit in destination operand,
- CF – carry flag – set to 1 if recent (mathematical) operation exceeded size of destination or in other words it is high-order bit carry or borrow,
- SF – sign flag – indicates whether most recent operation resulted in positive (0) or negative (1) value,
- PF – parity flag – set to 1 if number of bits set to 1 in the lowest eight bits of resulting operation is even.

Flags are cleared when their condition is not met so if specific flag is clear (its bit set to 0) then it means that situation mentioned in the above list did not happen.

Number of possible jump-like instructions is impressive. They are given in tables: 5.3, 5.4 and 5.5.

Summing up after this section reader:

- should be able to write and use simple functions,
- understand basic difference between kernel function and functions in user space,
- be more aware of nonlinear paths of code execution,
- understand relation between **cmp** instruction, **flags** register and conditional jumps,
- should be able to list and discuss at least ten conditional jumps.

Loops

Loops in programming are iterative operations thanks to which some code may be executed many times. Each execution of code within the loop is called iteration. Code inside of the loop that is being executed is called loop body. Loops may have three general forms:

- do-while loops which first do something and then check if condition is met to start next iteration,
- while-do loops which check if condition is met and if it is then they execute their body,
- iterator-based loops that rely on some counter (iterator) that counts up or down towards some predefined limit.

From the above list it may be concluded that do-while loops execute their body at least once even if condition is not met. Two other loops check the condition first and in general it may be not met at the beginning of the loop so it will not execute even once. While-do loops and loops which are based on iterator sometimes are used interchangeably as they both represent the same idea albeit in a bit different way so in high-level programming languages there are two different keywords for them. In assembly programming the difference between them is minimal.

Loops in assembly code will be analyzed on the basis of example program 06-looped.asm. First ten lines of code resemble previous examples with header in the form of comment and includes of files that contain defined constant values. Then in this code comes new constant:

```
%define        MAXLINES        5
```

This constant is uses several times in the code. It defines maximum number of lines of text that the program is supposed to generate. It can be better explained if we look at the program output:

```
> ./06-looped ....
    ^
   ^X^
  ^X^X^
 ^X^X^X^
```

Program uses length of command line argument as an information about the requested number of lines it should generate. So if the first (and only) argument has 4 characters then program will generate four lines of ASCII text which has triangular shape. It is easy to guess that this shape is generated iteratively in a loop. Triangle is built of two strings that are defined in **.data** section:

```
section .data
        space:        times MAXLINES               db        ' '
```

```
        branch:        times MAXLINES                db        'X^'
        newline:                                        db        10
```

First string labeled simply as `space` is built of space white characters. There is **NASM** directive **times** used here that makes it possible to define some data many times so that it effectively creates a series in computer memory. We could also declare string of five spaces, but any change to required number of maximum possible lines would make it necessary to modify the code in more than one place. Thanks to the **times** directive we may avoid such problem in the future. Similarly many times in memory string "X^" will appear labeled as `branch`. After it there will be one new line character (10 decimal), which due to the fact that it is placed in **.data** section just after the string `branch` it may be considered to be part of the branch `string` and so printed on the screen together with it.

In first instructions that should be executed this program checks number of command line arguments:

```
section   .text
global    _start
_start:
        cmp qword [rsp], 2
        jne quitprogfailure
```

It compares current stack pointer, which points to number of command line arguments (like argc) against value 2. If it is not equal to 2 then it means that the program was started without any arguments or with more arguments than just one. In such case execution flow leads to setup of value returned to the operating system:

```
; erroneous quit
quitprogfailure:
        mov rdi, EXIT_FAILURE
        jmp quitprog
```

The above two-liner just sets the **rdi** register and performs unconditional jump to code that actually invokes kernel function:

```
quitprog:
        mov        rax, sys_exit
        syscall
```

We may assume that user started the program with required number of arguments (just one) so instead of the erroneous quit the program will continue:

```
        call countchars
        call printlines
        xor        rdi, rdi
quitprog:
```

```
        mov         rax, sys_exit
        syscall
```

Program will do two things in functions: it will count number of characters in the command line argument and accordingly it will print the triangle made of specific number of lines. After return from second function program will finish by using `sys_exit`. However, perceptive observer should notice that lines 26-28 are discussed again. This is the feature of assembly language that same piece of code may be reused with different input data fairly easy. In case of expected execution code will zero register **rdi** and thus finish the program by returning to the operating system information of proper execution.

Function `countchars` is more like a procedure as it receives no arguments but works on data available on the stack:

```
countchars:
        mov rsi, [rsp + 24]
        xor rcx, rcx
looknext:
        cmp byte [rsi + rcx], 0
        je foundend
        inc rcx
        jmp looknext
```

Firstly this code acquires address of string (argv[1]) that is available on the stack. Then **rcx** register is cleared up as it will have the role of loop counter and it should hold number of characters of command line argument for remaining part of the code. The string that is provided through command line is ended with byte of value 0. Code between lines 42 and 46 is the actual loop that looks for this 0 by comparison between `byte` that is at address `[rsi + rcx]` and 0. In such line we must openly declare what data unit should be used by compiler because it cannot be guessed from the context. If condition is not met then **je** instruction is simply skipped and **rcx** register is incremented by 1. Then unconditional jump repeats loop but now with already incremented **rcx** so the next byte is analyzed.

In case if byte equal to 0 is found then execution flow goes to label `foundend`:

```
foundend:
        cmp rcx, MAXLINES
        jbe finishcounting
        mov rcx, MAXLINES
finishcounting:
        ret
```

In line 48 counted number of characters is compared against maximum assumed number. If it is beyond the threshold it is changed to the maximum value given by `MAXLINES` defined constant. In both situations function ends by using **ret** instruction.

The triangle may be drawn once its number of lines is known. But before that two registers are prepared:

```
printlines:
        mov rbx, newline - 1
        mov r12, 2
```

Register **rbx** holds address of last character in the `branch` string as it is the same address as `newline` – 1. Register **r12** holds information about the number of characters from `branch` string which should be written to the screen. At the beginning it equals to 2 because one character is of which the triangle is built in the first line and the other (next one in memory) is the new line character at address `newline`.

Then there is core of the program that iteratively draws lines that together look like a triangle:

```
nextline:
        push rcx
        call drawline
        dec rbx
        dec rbx
        inc r12
        inc r12
```

Because **rcx** register may be changed during execution of incoming system calls it is preserved on the stack thanks to **push** instruction. Then in line 63 there is **call** to another procedure `drawline` that actually draws textual strings. We will get back to it in a moment but for now let us analyze remaining instructions of `printlines` procedure.

There are two decrements of **rbx** register and two increments of **r12** register. So by decreasing **rbx** we are moving pointer within branch string backwards effectively making more of it available for use. We may observe that in every line there are two more visible characters building the triangle as it grows in both left and right directions evenly. Reader may try to comment out one of this decrements and see the result. Accordingly register **r12** is increased twice so two more characters should appear.

After that value from stack is **pop**ped back to register **rcx**:

```
        pop rcx
        loop nextline
        ret
```

It should be the same number which was pushed to the stack at the beginning of the loop. It is a decreasing counter but there is no directly visible decrement of **rcx** register. Instruction **loop** looks quite innocently but it is real beast in terms of what it does. There are three operations performed by **loop** instruction:

- **rcx** (loop counter) is decremented by 1,
- current (already decremented!) value of **rcx** register is compared against 0,
- if **rcx** did not reach 0 yet then jump occurs to an address indicated by **loop** instruction argument (nextline in the above example).

A code in C resembling loop in assembly language that is based on **loop** instruction may look like this:

```
do {
 something();
} while (--rcx > 0);
```

It is imporant to observe that when **loop** instruction is executed it is assumed that **rcx** register was not changed by any other instruction inside the loop. If it is necessary to modify **rcx** inside of the loop then its value as from a counter must be preserved somewhere (e.g.: on stack) and retrieved just before the **loop** instruction. In fact this rule applies to all loops as simple it claims that counter should not be changed unless for counting loop iterations.

Another caveat about **loop** instruction is that it is very slow. It looks simple but there is alternative almost as simple. Actually all we need is to decrement **rcx** and jump on condition that **rcx**$\neq$0. Code with exactly the same effect looks like this:

```
    dec rcx
    jnz nextline
```

Why to use this? It occupies just a little more space in memory but is much faster than **loop** instruction.

Procedure ends with **ret** instruction once the above loop is finished and then execution flow goes back to already discussed line 25 from tha main part of the program.

What is left to analyze is function `drawline` that is simple double invocation of **sys_write** kernel function:

```
drawline:
        mov rax, sys_write
        mov rdi, STDOUT
        mov rsi, space
        mov rdx, rcx
        syscall
        mov rax, sys_write
        mov rdi, STDOUT
        mov rsi, rbx
        mov rdx, r12
        syscall
        ret
```

In this function it is assumed that **rcx** contains number of white spaces which should be printed. Because this register is decremented in the loop discussed above so the number of spaces is less by one every line. Second assumption is about the number of characters from `branch` string which should be printed. As it was mentioned above the pointer to position in this string is held in **rbx** register while number of characters that should appear is in **r12** register.

This function ends properly with **ret** instruction so execution goes back to line 64 which is immediately after the place from which the function was called.

Concluding, there are two methods of making loops in assembly language:

- based on loop instruction,
- based on general purpose registers, comparisons and conditional jumps.

There is no "better" or "worse" method to create a loop and programmer should use individual approach depending on specific situation.

After this section reader:

- should be more familiar with practical use of comparisons and conditional jumps,
- should be able to write loops based on comparisons and conditional jumps,
- should understand special role of **rcx** register in **loop**s,
- should be more experienced with relative addressing method,
- should be more skilled in analysis of multiple execution paths.

Logic Instructions

Our next program will be an extremely powerful cryptographic tool. Well, at least it will apply the idea of best cryptography which is one-time pad. The one-time pad method is more than century old but it is still not breakable. The idea is stunningly simple:

1. take one series of bits that represent message,
2. take another, completely random series of bits with same length as the the message,
3. "mix" these two series pair-wise using XOR operation.

Without the key it is not possible to recover the original message. Having the key and encrypted message makes it possible to use pairwise XOR operation again. Because XOR of something twice with the same argument results in the original value hence it is possible to recover the original message.

In Linux it is extremely easy to obtain random series of bits as all that is necessary is to read one of two special files available:

- `/ dev / random`

- `/ dev / urandom`

Both of these files are virtual entities provided by kernel functionality. However, there are significant differences between them.

Former one is using entropy pool built on the basis of physical events that are observed on computer, like pressing keys and moving computer mouse. Therefore it is good option for cryptographic purposes. However, entropy pool drains very quickly hence the generation of random data takes some time.

Alternatively one may use second file which is just a pseudo-random generator implemented in the kernel. It is efficient as it produces random data without delay. However, it is just a complicated mathematical formula in the kernel so it is not as reliable in terms of security as the former file.

We may create file with 100 random bytes by this method:

```
 > head -c 100 /dev/random  > rnd.key
```

For 100 bytes author had to wait more than 30 seconds but some other time 100 bytes were available immediately. This time will vary depending on machine.

Now we will analyze program named `07-hasher`. It is the longest and most complicated program so far. Its usage session may look like this:

```
 > ./07-hasher rnd.key > output.bin
Hickory dickory dock
The mouse ran up the clock
 > ls -l output.bin
-rw-r--r-- 1 l l 48 Apr 17 15:56 output.bin
 > xxd -c 8 output.bin
00000000: 7c22 07d8 cb19 b753  |".....S
00000008: 596f 21fa b9bb 71da  Yo!...q.
00000010: 072b 6860 b568 c0fa  .+h`.h..
00000018: b5cd 403d 09a4 caa7  ..@=....
00000020: a84e c55f ca4c 372f  .N._.L7/
00000028: 29f7 83fd 6bb6 17cb  )...k...
 > ./07-hasher rnd.key < output.bin
Hickory dickory dock
The mouse ran up the clock
```

Now we can analyze the code from file `07-hasher.asm` which uses uninitialized data section for data processing:

```
%define CHUNKLEN        8

section .bss
        keybuf:                 resb        CHUNKLEN
```

```
        databuf:        resb        CHUNKLEN
        fd:                         resq        1
```

Program will read data from two sources and store it in respective buffers: `keybuf` which is for random key and `databuf` which is for data that is to be encrypted or decrypted. Each buffer has the same length which in the example is 8 bytes. This number may be changed.

Program opens one file, the one with random binary key, which file descriptor is stored in `fd` variable. This part of code is simple and the method was already discussed so it will be omitted here.

Main part of the code starts with check of argument presence:

```
        cmp qword [rsp], 2
        jne badresult
```

In case of failure let it be to few or too many arguments program will be stopped.

Next comes lengthy (a bit too lengthy) loop:

```
nextchunk:
        mov rdi, STDIN
        mov rsi, databuf
        mov rdx, CHUNKLEN
        mov rax, sys_read
        syscall
        call verifyresult
        cmp rax, 0
        je quitprog

        mov rdi, [fd]
        mov rsi, keybuf
        mov rdx, rax
        mov rax, sys_read
        syscall
        call verifyresult
        cmp rax, 0
        je quitprog

        mov rcx, rax
        call hashkeybuf

        mov rdx, rax
        call storefrombuf
        jmp nextchunk
```

Inside of the loop there is reading from standard input. If everything is alright then `databuf` is

refreshed with new data. The same number of bytes is read from opened file to which reference is held in `fd` file descriptor.

It is achieved by the means of **rax** value returned from first **syscall**. The **rax** is not changed by any piece of code until it may be moved to **rdx** in line 40. It ensures that even if less bytes were available from the standard input than length of the buffer, the remaining part of the loop will rely on actual number of characters available. It is assumed here that there are always more key bytes available than there are data bytes. Without this assumption it could not be one-time pad method.

It needs to be mentioned here that the loop is too long and may be shortened by removing repeated part to external, procedure that may be **call**ed. Reader is encouraged to try it as a form of training.

Then number of characters that were read successfully is used again in ciphering of data buffer:

```
        mov rcx, rax
        call hashkeybuf
```

The actual function `hashkeybuf` that performs ciphering looks like this:

```
hashkeybuf:
        mov dl, byte [databuf + rcx - 1]
        xor byte [keybuf + rcx - 1], dl
    dec rcx
    jnz hashkeybuf
        ret
```

To **dl** register which is the lowest part of **rdx** register single byte from key buffer is moved. Next XOR operation is done on the buffer, which element length is indicated to be byte.

Please notice that **rcx** plays double role here: it is loop counter and element of relative addressing expression. Thanks to that XOR is performed in backward order – from the last till the first element of data buffer. Because loop uses **jnz** instruction then rcx will never reach 0 here. Furthermore **rcx** value provided to this procedure contains number of characters and not the maximum index which is less by one. Hence there is "-1" in the relative addressing expression. But it can be avoided by moving decrement of **rcx** before using it in relative addressing. This is another change, really simple, which should be tested by reader.

In a harder attempt this piece of code could be improved with XOR operation on whole 64-bit words assuming that input data comes in numbers that are multiples of 8. This change is also left to reader for practice.

When another chunk of data was processed then it is presented to standard output:

```
        mov rdx, rax
        call storefrombuf
```

Storing uses simple `sys_write` to `STDOUT`, which was discussed several times already so we will not go into details here.

It is assumed that the next fragment of data will be available if none of conditional jumps within the loop body worked. Otherwise unconditional jump **jmp** will start next iteration of the loop.

```
        jmp nextchunk
```

The program discussed above was example of **xor** operation which is used in one-time pad ciphering. Other logical instructions of four basic logical operations also exist and may be used in similar way.

Thanks to this section reader should:

- know how to flexibly use files from filesystem and special files like `STDOUT` or `STDIN` together in single program,
- know how to use logical instructions in code,
- become more experienced with general code flow.

Multiple Source Files

Source codes that are analyzed in this chapter get longer and longer each section. They also become harder to analyze and understand. There is a way to improve quality of code that is used in almost all programming languages. It is slicing the code into several files. Making fragments of code separated and connected only by some well defined and known interface. Usually this "glue" between functions defines way in which code is called and how it returns results.

Source code 08-multi-main.asm has in the header something that was not discussed yet:

```
extern strlen
extern strchr
extern writeout
```

These are declarations for compiler that specific symbolic names are "external" so the compiler should not bother us with errors that these labels cannot be found in the code. By using this directive we assure compiler that these labels will become available during linking, when we provide them from somewhere else. That means – from other file that will be discussed in later paragraphs of this section.

Now we analyze the "main" part further:

```
section .data
        databuf:        db        "Some string in memory.", 10, 0
```

This is sample text which will be processed in this program. It could be retrieved from external source like STDIN or file. However, such additional code that we are already familiar with, could be just confusing and make us miss the main point of this section.

Code of this program starts with preparation of arguments and invocation of already mentioned `strchr` function:

```
        mov rdi, databuf
        mov rsi, qword 's'
        call strchr                ; rax := address
```

First argument is an address to buffer which will be analyzed. Second argument is character which is sought for. After the function `strchr` finishes then **rax** points to first occurence of character provided by **rsi**.

The function `strchr` looks and works exactly like function with the same name from standard C library. But this one is implemented independently and provided alongside `strlen` and `writeout` functions in separate source file. Main part of the program uses the `writeout` to print the newly found beginning of the string and finishes its execution.

As there was **extern** keyword used in the "main" part so there are **global** directives in the second source file named `08-multi-func.asm`:

```
global strchr
global strlen
global writeout
```

We have already used this keyword many times. Thanks to this we made `_start` visible globally and so callable from external code. The method works in this program split in two files as well.

Now we may analyze three functions that are present here starting with `strlen` that is shown here complete:

```
strlen:
    xor rcx, rcx
    not rcx
    xor rax, rax
    cld
    repnz scasb
    not rcx
    dec rcx
    mov rax, rcx
    ret
```

The code shown above counts number of characters in string. Perhaps you have noticed that string `databuf` defined in file `08-multi-main.asm` is ended with byte of value 0. This is so called ASCIIZ string that is ASCII ended with zero. The trailing zero is very popular as it enables to make strings of arbitrary length. For example standard C library is using this approach. So in

here the function `strlen` walks the memory and count characters from some specific point, it this case first byte of the `databuf`, till it will find 0.

Counting is done on **rcx** register but due to applied method the register must decrement each time non-zero character is found. So it starts from maximum possible value that is logical negation of zero. Character that will end the search should be in **al** register. Because in here we are looking for 0 then simple **xor rax,rax** does the work. Last thing that needs to be prepared before search starts is status of D flag – where D stands for "direction". When it is clear then search goes forward in memory i.e. increments analyzed addresses. Otherwise it goes backwards.

Search is done in a loop that is written in line 23. This one-lines is built of two instructions. First there is **repnz** that stands for repeat while not zero. This instruction was designed primarily for analysis of textual strings but in many walks through memory it may be found useful. It comes from a group of similar instructions:

- **rep** – repeat while rcx≠0,
- **repz** – repeat while zero,
- **repe** – repeat while equal,
- **repne** – repeat while not equal,
- **repnz** – repeat while not zero.

Each of them decrements **rcx** register just like **loop** instruction does. So if **rcx** reaches 0 then it results in flag Z set and consecutively ends repetitive operation.

Companion instruction **scasb** comes from a group:

- **scas** – compares **al**, or **ax**, or **eax**, or **rax** with data in memory depending on size of provided data unit (byte, word, dword, qword); memory address is given by **rdi** (or pair **es:edi** in 32-bit programs),
- **scasb** – explicitly compares byte (register **al**)
- **scasw** – explicitly compares word (register **ax**)
- **scasd** – explicitly compares double-word (register **eax**)
- **scasw** – explicitly compares quad-word (register **rax**)

Each of these instructions increment **di/edi/rdi** register (applies to 16-bit, 32-bit and 64-bit architectures) when D flag is cleared. Value by which the memory pointer (e.g.: **rdi**) is incremented depends on which of **scas*** instructions were used.

When byte of value 0 is found then the loop ends. It is known that register **rcx** was decremented as many times as there were characters analyzed. To convert it to normal (positive) value it has to be inverted on all bits. Decrement of the inverted value comes from the fact that trailing zero should not be counted.

Function returns number of characters in a string through **rax** register.

Pair of instructions in the style **rep**–**scas** is very powerful. But could this piece of code be written in shorter and more efficient way? Perhaps you could try it, as a practical experiment.

Hint: the **repnz scasb** construct occupies only two bytes of code even on 64-bit machine but these are actually two separate instructions so you can use any of them without the other.

Next function is a bit longer so it will be analyzed in parts. So here is the beginning of `strchr`:

```
strchr:
        push rdi
        push rsi
    call strlen
```

Caller is providing two arguments to `strchr` function: pointer (address) of string to analyze and character that should be found. It is done via registers **rdi** and **rsi**, as it should be done according to 64-bit ABI. String to which the pointer leads is ASCIIZ string but its length is not known. Searching for the character should not go beyond the trailing value 0. Therefore function `strchr` relies on function `strlen`.

Both of these functions require address of a string to analyze therefore register **rdi** is preserved on the stack and then the value from stack is retrieved. Similarly the second argument that comes via **rsi** register may be modified during code execution so it is also preserved on the stack. However, retrieved value is moved to **rax**, where it is needed from the **scasb** instruction point of view.

Once the string length is known then searching may occur up to the last character of the string. The sought for character does not have to be present in the analyzed string in general. Here is this primary part of the function:

```
    mov rcx, rax
    inc rcx
    cld
    pop rax
    pop rdi
    repnz scasb
```

Because function `strlen` returns length of the string via **rax** register its value needs to be moved to proper registers. Then **rcx** is incremented so to include trailing byte with value 0. Including such possibility provides simple method to return value 0 if character was not found.

Finally the value of **rdi** is adjusted and returned via **rax**:

```
    dec rdi
    mov rax, rdi
    ret
```

Function is expected to return pointer to a found character but without decrement of **rdi** it would point to the character after that character the function was looking for.

This function could be written in better way. Try mixing code from `strlen` function with the code that searches for specific character.

Finally there is `writeout` function that looks like this:

```
writeout:
        push rdi
        call strlen
        mov rdx, rax
        pop rsi
        mov rax, sys_write
        mov rdi, STDOUT
        syscall
        ret
```

Pointer to string is provided via **rdi**. It will be needed not only to count number of characters and so to determine length of the string but also for writing that data to output. Therefore its value is protected by **push**ing to the stack. Calculated length of the string is available in **rax** register from which it is moved to **rdx** because `sys_exit` kernel function expects number of characters to be printed in rdx register (3rd argument of C function).

Writing to standard output does not need any more comments as it was discussed several times. This operation finishes execution of the function which is the last one executed by presented program.

Compilation of this program is slightly more complicated than compilations that were used in this book so far. We have to remember that all source files are compiled separately so we get as my relocatable object files as there are source files:

```
 > nasm -felf64 08-multi-main.asm -o 08-multi-main.o
 > nasm -felf64 08-multi-func.asm -o 08-multi-func.o
 > file 08-multi*o
08-multi-func.o ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
08-multi-main.o ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Further analysis of these two files brings more clarity to what they have inside:

```
 > nm *.o
08-multi-main.o:
0000000000000000 d databuf
0000000000000000 T _start
                 U strchr
                 U strlen
                 U writeout

08-multi-func.o:
0000000000000016 T strchr
0000000000000000 T strlen
000000000000002f T writeout
```

So actually there are undefined symbols (U) in the "main" part, without addresses. On the other hand the "func" part is missing the `_start` label. Only together they make whole so they have to be linked together and here we may see ability of linker at its full grace.

```
 > ld 08-multi-main.o 08-multi-func.o -o 08-multi
 > file 08-multi
08-multi: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
 > nm 08-multi
000000000060013c D __bss_start
0000000000600124 d databuf
000000000060013c D _edata
0000000000600140 D _end
00000000004000e7 t search
00000000004000b0 T _start
00000000004000f4 T strchr
00000000004000e0 T strlen
000000000040010d T writeout
 > ./08-multi
string in memory.
```

So linker (ld in this case) actually glues separate parts together. In this example there were only two such file but in regular programs the number of relocatable files might be counted in dozens.

After this section reader:

- should have profound understanding on how multiple-source programs are built,
- know relation between **global** and **extern** keywords,
- understand how rep-like instruction works,
- understand how scanning of memory with scas-like instruction works.

Standard C Library in Assembly Programs

With kernel functions it is possible to get some data from and to file descriptors. But the kernel is just giving the user controlled access to hardware. There is no higher level features like ability to actively verify input or make pretty formating of the output. Such functions are available in standard C library and in this section we will make use of them.

The program which makes background for this section should have similar behavior to program `03-interact.asm`. So it provide some introductory text, wait for user enter her/his name and then print greeting using the entered name. However, all should be done with `scanf()` and `printf()` functions known from C programming. Because code uses external functions then the compiler has to be informed about it:

```
extern scanf
extern printf
```

Then there are two NASM macros both related to maximum length of user name:

```
%define        USERNAMEMAXLEN        30
%defstr UNXLENS USERNAMEMAXLEN
```

First if them defines a number, that for example may be moved to CPU register. The other one defines series of ASCII characters so a string that may be concatenated with other strings and for example printed to the screen. Second one uses the first as its argument so one will be a number and the other its textual representation.

Strings necessary for `printf()` and `scanf()` functions are prepared within **.data** section:

```
section   .data
        intro:                db        "Hello, what's your name?", 10, 0
        welcome:        db        "I am pleased to meet you %s.", 10, 0
        scanform:        db        "%", UNXLENS, "s", 0
```

Each of these strings is ended with byte 0, so they all are ASCIIZ strings like it is used and required by functions from standard C library. Text labeled as `scanform` uses defined string and so it makes a formatting string for later use by `scanf()` function.

Memory area in which scanf() function will store the entered user name is reserved in .bss section:

```
section .bss
        username:        resb        USERNAMEMAXLEN
```

Size of the reserved space is defined in only one place so it is easy to change it in future.

Code is relatively simple as it starts with text message being printed to standard output:

```
        mov rdi, intro
        xor rax, rax
        call printf
```

Please compare library function `printf()` with corresponding kernel function `sys_write`.

In current example we do not declare any numeric representation of the function as it is called by its address seen as its name (label).

There is no need to declare `STDOUT` as the output file descriptor because basic form of `printf()` assumes it has to be so.

Pointer to buffer is given by **rdi** register that is on top of the 64-bit ABI function arguments list.

Counter indicating how many bytes should be printed was used in kernel functions but with C library functions string are ended with 0 so there is no need for that. Instead **rax** register has to be zeroed. It is so because its value is an information about number of floating-point

arguments provided to the function. According to 64-bit ABI it applies to functions with varying number of arguments such as `printf()` or `scanf()`. We will get back to this topic in section dedicated to mathematical functions.

Part of code that uses `scanf()` is slightly longer:

```
mov rdi, scanform
mov rsi, username
xor rax, rax
call scanf
```

The `scanf()` function requires formatting string and address of memory area where the data should be stored. They are provided by **rdi** and **rsi** registers.

Text that was acquired in previous snippet is used in the next **call** to `printf()`:

```
mov rdi, scanform
mov rsi, username
xor rax, rax
call scanf
```

This use of `printf()` is based on formatting string indicating use of one value that is of string type. Pointer to the string which has to be integrated is given via **rsi** register. There should be user name entered at that address thanks to previous use of `scanf()`. If reader does not remember formatting string symbols then please consult printf() manual:

```
> man 3 printf
```

Finally program should finish gracefully invoking kernel function `sys_exit`.

This program relies on external library so during compilation this fact cannot be forgotten or there will be errors and strange behavior.

```
> nasm -felf64 09-libraryfun.asm -o 09-libraryfun.o
> ld 09-libraryfun.o -o 09-libraryfun
09-libraryfun.asm:(.text+0xe): undefined reference to `printf'
09-libraryfun.asm:(.text+0x2a): undefined reference to `scanf'
09-libraryfun.asm:(.text+0x46): undefined reference to `printf'
```

Perhaps perceptive user of standard C library functions noticed requirement to link with the library by adding **-lc** argument to linker **ld**. When we add this option then linking is successful but program cannot be started properly.

```
> ld 09-libraryfun.o -o 09-libraryfun -lc
> ./09-libraryfun
bash: ./09-libraryfun: No such file or directory
```

Shell complains that the program we are trying to start seems to not exist. How can it be so if the file obviously is present in the filesystem and we may even inspect it with tools like **nm**?!

```
 > nm 09-libraryfun
0000000000600ec0 d _DYNAMIC
0000000000601000 d _GLOBAL_OFFSET_TABLE_
0000000000601065 B __bss_start
0000000000601065 D _edata
0000000000601088 B _end
00000000004002b0 T _start
0000000000601028 d intro
                 U printf@@GLIBC_2.2.5
                 U scanf@@GLIBC_2.2.5
0000000000601060 d scanform
0000000000601068 b username
0000000000601042 d welcome
```

The above analysis should give us a hint what is happening. Linker found the library so it did not complain about undefined references. But still there are two undefined symbols, both without address in memory and both from C library. It is so because the standard C library is shared one and the linking resulted in dynamically linked file, what may be verified in very simple way.

```
 > file 09-libraryfun
09-libraryfun: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically li
```

It is different to what we had so far where all programs were “statically linked”.

Interpreter in the output from file command is another name for dynamic linker. So the program was linked dynamically but we missed to provide information about the position of dynamic linker during linking phase. There is a command line option for **ld** that does it.

```
 > ld 09-libraryfun.o -o 09-libraryfun -lc --dynamic-linker /lib64/ld-linux-x86-64
```

Now it will be possible to run the program correctly assuming that on the testing system the interpreter is at given location.

One may find name of the dynamic linker thanks to **ldconfig** command line application that controls all shared libraries in the system:

```
 > /sbin/ldconfig -v | grep ld-linux
```

This program can list all shared libraries available in the system. On one of author’s computers there were more than 2000 of them but mind it is not that much in fact.

If executable program relies on shared libraries it may be investigated which ones are necessary by using **ldd** command.

```
 > ldd 09-libraryfun
```

```
        linux-vdso.so.1 (0x00007ffee4b26000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4ac6bb1000)
        /lib/ld-linux.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007f4ac6f50000)
```

Why all the fuss if we could simply consolidate library with the library statically? Well, there are several reasons for using shared libraries instead of static ones. First of all shared library is provided to all applications in the system. There may be literally dozens of thousands of them and if each would be statically linked then usage of media storage would rise significantly both in terms of occupied space and in terms of access time. Secondly shared library may be loaded to memory only once and provided to all applications so the memory footprint is limited. Finally if each of programs linked library statically then a fix to newly found problem in a library function would lead to necessity of recompilation of all programs. When library is shared then there is just one place which receives patch.

After this section reader should:

- understand differences and similarities between kernel functions and library functions,
- understand benefits and potential problems of using shared libraries,
- known how to compile source that is using shared library.

Accessing Assembly Functions from C Programs

From previous section we should remember that using external libraries is possible in assembly programs. We may assume that most of libraries were written in C language so it is like we used C functions within assembly programs. If you wonder whether it is possible the other way round too then answer is simple and obvious. Of course it is! And now we are going to test this functionality in this section. It is perhaps even easier than using shared library so this section will also introduce basics of arithmetic operations in assembly.

In the testing application we will calculate factorial of values starting from zero and going upwards. Due to finite size of CPU registers we will have to stop at some point. With more advanced code it would be possible to go beyond limits of single register but it would require significant effort in displaying results and as such is not the point of the current section.

So here is the C code for testing purposes:

```
#include <stdio.h>
extern unsigned long factorial(unsigned long);

int main(void){
    unsigned long k;
    for (k=0; k<= 22; k++)
        printf("%u! = %lu\n",
                                k,
```

```
                                (unsigned long) factorial(k)
                        );
return 0;
}
```

As you can see also in C code we have to use **extern** keyword to indicate that some functions are to be found during linking phase. Due to C rules it must be also clarified what type of data that external function expects and what type of value it will return. This is just representation that is interpreted by C code. Internally it still uses 64-bit ABI so **rdi**, **rsi** and so on for arguments and **rax** to return value. In case of more than six arguments stack has to be employed to pass them. Alternatively more arguments or returned values may be passed as memory structures to which only pointer has to be passed.

So the **factorial()** function will be iteratively called up to `k=22`. Program compilation consists of several stages as it is shown below.

```
 > nasm -felf64 -gdwarf 10-factorialfun.asm -o 10-factorialfun.o
 > gcc -c -g 10-testfactorialfun.c -o 10-testfactorialfun.o
 > gcc -g 10-factorialfun.o 10-testfactorialfun.o -o 10-testfactorialfun
```

To compile assembly part **NASM** should be used but to compile C part and link the whole program **gcc** is necessary. It is so because entry point `_start` and `main()` function are not the same. It is easy to verify with the **nm** tool that proved how invaluable it is several times already. Results will not be shown here to not clutter the book pages with about 40 symbols that are present in the executable file. Readers are encouraged to make this test in their own. So there must be some instructions executed between entry point and the `main()` function. If one is interested then analysis of the code execution with debugger brings some insight to the process which is beyond the limited scope of this book.

If code was properly compiled and program linked successfully then its results will be like it is shown below.

```
 > ./10-testfactorialfun
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
```

```
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 0
22! = 0
```

Now we may finally analyze the assembly part. There is declaration that makes `factorial` function available from the outside:

```
global factorial
```

The function is simply a label in **.text** section:

```
section .text
factorial:
```

There is not `_start` label in this piece of code as it is not a standalone program. It is a function that has to become part of a larger program.

Function starts by checking whether its argument is equal to 0 and in such case it returns 1 because $0! = 1$.

```
    cmp rdi, 0
    jne expnotzero
    mov rax, 1
    ret
```

In general situation code labeled as `expnotzero` is executed.

```
expnotzero:
    mov eax, 1
```

Product will be stored in **rax** register and it will also be used in next multiplication. So **eax** is set to 1 before the loop may start. Why not **rax** here and what happens to upper half of **rax** if only lower part – the **eax** is set to some known value? When lower part of 64-bit register is set to some value then most significant bit of it is populated to all bits in the upper part. So it preserves sign of the lower part.

The **mov** used here occupies only 5 bytes in memory where 4 of them represent value moved to **eax** and one stands for the instruction. Alternatively one could try using **xor rax,rax** in one line and then **inc rax** which effectively would result in **rax** having value of 1 . But such approach

would use 6 bytes in memory. Furthermore such code would be less clear. The value 1 used here rather should not become a symbolic name because it will always be 1 from which the factorial calculation has to start.

Argument for the factorial function is provided by **rdi** register but it may be considered as an information about how many times the loop should iterate. Therefore it is used as a loop counter and is already prepared.

Body of the loop is just few lines long:

```
loopfactorial:
    mul rdi
    cmp rdx, 0
    jnz overflow
    dec rdi
    jnz loopfactorial
    ret
```

First operation that is done here is multiplication of integer values. Instruction **mul** takes only one argument because **rax** is always used as a factor. So there has to be only second factor provided and in our case it is **rdi**. One may notice that at the beginning of loop the **rdi** holds maximum multiplicand by which the multiplication in factorial series should be done.

Product goes to pair of registers: **rdx:rax**. They make pair like they were one register with doubled size. Why is it necessary? We may find answer to this question on simple example. Let us imagine that we have very simple processor with 3-bit registers. We put decimal value 7 (0b111) to each of them. Product equals to 49 decimal or 0b110001. If we count how many bits are needed to store the result it is 6. So product requires twice as much of binary positions as the factors had. Hence such behavior of **mul** instruction that it is supported with **rdx:rax** pair. Instruction **mul** has similar behavior in 32-bit, 16-bit and 8-bit multiplications in which results are stored in **edx:eax**, **dx:ax** and **ax** respectively.

Once we understand the above explanation it becomes clear why suddenly **rdx** register is compared against 0. If it is not equal to 0 then it means that result was so large that it did not fit into **rax** only. In that case conditional jump **jnz** will execute. Otherwise code will flow decrement the loop iterator. As long as **rdi** stays above 0 then next iteration of loop may execute thanks to conditional jump in line 24.

In case of overflow simple two-liner finishes the function:

```
overflow:
        xor rax, rax
        ret
```

Function returns value by **rax** register. Factorial cannot be equal to 0 so such situation indicates overflow error.

Instruction **mul** works on unsigned integers but has a companion that works on signed integers – **imul**.

Similarly there is pair of instructions for division: **div** and **idiv**. Similarly to **mul** instructions also **div** require only one argument that is the dividend. Divisor is assumed to be in **rdx:rax** in 64-bit operation, **edx:eax** in 32-bit operation, **dx:ax** in 16-bit operation and **ax** in 8-bit operation. This guarantees backward compatibility. Because division is done on integer values then quotient goes to **rax** (or **eax**, or **ax**, or **al**) whereas remainder is stored in **rdx** (or **edx**, or **dx**, or **ah**).

For sum there is **add** instruction that has two arguments. Leftmost argument is the destination of result and also first term. Second argument is second term. When result is too high to fit into destination then **carry** flag is set. It may be used in next addition but to include this bit at the least significant position instruction **adc** should be used instead. This is "add with carry".

Similarly to addition a subtraction works thanks to **sub** instruction. First argument of this instruction is minuend and also destination for result. Subtrahend is provided as second argument. In multibyte subtraction carry flag may indicate a borrow. In such case first **sub** instruction should be followed by a **sbb** instruction(s) that subtract second argument and also value of carry flag (1 is for set).

Having this section finished reader:

- should know how to make available in C code an external function written in separate assembly source file,
- how to export functions from assembly code so that they are available in C code,
- be familiar with basic mathematical instructions that operate on integer values.

SSE Mathematical Operations

Final program in this book (`11-floats.asm`) will use some of very powerful SSE instructions. Thanks to them we will do floating point arithmetic. This program will convert temperature given in Celsius degrees to absolute Kelvin scale and to Fahrenheit degrees. Program compilation is similar to one used in previous example as it requires dynamic linking against standard C library. Properly compiled and linked program may be started so that activity shown below is possible.

```
 > ./11-floats
Please enter temperature in Celsius:
37.5
37.50'C =  310.65 K = 99.50'F
```

Code in the header has comment with info about compilation, it includes syscall definitions and declares that printf and scanf function will be available as external ones.

Then comes relatively lengthy **.data** section with several strings for `printf()` and `scanf()` functions:

```
section .data
    queryinfo:                db "Please enter temperature in Celsius:",10,0
    queryformat:        db "%lf",0
    results:                db "%.2lf'C =  %.2lf K = %.2lf'F",10,0
    toosmallerror:        db "Error: provided value is too small.",10,0
```

Please pay attention to formatting strings which all use “long float” so in other words double precision numbers. Each of such numbers occupy 64 bit so 8 bytes in memory. Some are declared further in **.data** section:

```
    align 16
    absolutezero:        dq -273.15
    CtoFmultiplier:        dq 1.8
    CtoFoffset:                dq 32.0
```

These floating point values have to be aligned so start at address that is factor of 16. All of them are quad-words so 4 double-bytes that is 8 byte per each of them. Similarly in **.bss** section there is space reserved for Celsius degrees value that should be provided by user of this program:

```
section .bss
    align 16
    tempC:                        resq 1
```

Code starts easily with print of message to the screen by the `printf()` function. There are no arguments to this simple call so it is not shown here. Further there is `scanf()` used to obtain value for conversion from the user:

```
    mov rdi, queryformat
    mov rsi, tempC
    xor rax, rax
    call scanf
```

You may observe that pointer to memory area which has to be filled with double-precision floating-point number is provided like any other pointer, via **rsi** register.

Then comes more interesting part that uses SSE instructions:

```
    movsd xmm0,[tempC]
    movsd xmm2,[absolutezero]
```

One value from memory pointed by `tempC` is loaded to lower part of register **xmm0** and another pointed by `absolutezero` is loaded to register **xmm2**.

Registers **xmm** have length of 128 bits so into one could fit two numbers that are 64-bit in size. However, in the above example there is letter “s” in the **movsd** instruction which indicates that a scalar (single number) is loaded. If we would like to load two numbers side-by-side then instead of letter “s” there should be “p” which stands for “packed”. Letter “d” in the instruction

mnemonic stands for “double precision” but there are also single precision numbers each denoted with letter “s”. Single precision numbers occupy 32 bits each, so four bytes (double word) in memory.

Therefore there are several possible options of **mov** instruction for SSE registers:

- **movss** – move scalar with single precision
- **movsd** – move scalar with double precision
- **movps** – move four single precision values
- **movpd** – move two double precision values

If there are more values to be moved then consecutive positions in memory will be used. Therefore “packed” versions of SSE instructions are very useful in vector arithmetic and signal processing. They are invaluable in computer graphics and real-time (de)compression of streamed data. So there is reason why SSE stands for Streaming SIMD Extensions. SIMD stands for “single instruction multiple data” as arithmetic instructions may be applied pairwise to all elements of packed **xmm** register.

Next there is instruction **ucomisd**:

```
ucomisd xmm0, xmm2
jb error
```

It compares two **xmm** registers, like **cmp** instruction does with integer data in regular registers: If the value that user entered is lower than absolute zero then program jumps to code labeled as `error` and finishes returning 1 to the operating system.

Otherwise it executes more instructions on **xmm** registers:

```
movsd xmm1,xmm0
subsd xmm1,xmm2
```

First one makes copy of value from register **xmm0** to register **xmm1**. Thanks to that register **xmm0** will contain the original temperature in Celsius degrees and it will be possible to print it to the screen. Second instruction subtracts double precision scalars where **xmm1** is minuend and **xmm2** is subtrahend. Difference is stored in the first of these registers. This way temperature in Kelvin scale is calculated and stored in **xmm1** register.

Then comes conversion to Fahrenheit degrees that requires multiplication by 1.8 and addition of 32.0:

```
movsd xmm2, [tempC]
mulsd xmm2, [CtoFmultiplier]
addsd xmm2, [CtoFoffset]
```

Both operations are done on **xmm2** register to which temperature in Celsius degrees is copied from memory. Instruction mulsd calculates factor of **xmm2** register and multiplier 1.8 stored

in memory. Result goes to first argument of instruction so to **xmm2** register. Then addition is performed with **addsd** instruction. Thereafter **xmm2** holds temperature in Fahrenheit degrees.

Finally results may be printed to the screen with `printf()` function:

```
mov rdi, results
mov rax, 3
call printf
```

Unlike in previous examples here we have to move value of 3 to register **rax**. It indicates that there should be three floating point values used. It is assumed that xmm registers will be used in order starting from xmm0. This is why we populated **xmm0** with Celsius degrees, **xmm1** with Kelvins and **xmm2** with Fahrenheit degrees just as it fits formatting string labeled as `results`.

Mnemonics for SSE additions, subtractions, multiplications and divisions all have suffixes similar to SSE **mov** instructions. Using them may be a bit intimidating at first but later one may consider they are easier to use than regular mathematical operations on integer values and general purpose registers. SSE has lot more specialized instructions like:

- reciprocals: **rcpps**, **rcpss**,
- square roots: **sqrtps**, **sqrtss**,
- reciprocals of square roots: **rsqrtps**, **rsqrtss**.

These few instructions are just examples of the large set available. SSE is developed since year 2000. Latest version is SSE5 from year 2009. In year 2010 Intel announced Advanced Vector Extensions (AVX) with **ymm** registers that are 256 bits long. In 2015 AVX-512 was presented that introduced 512-bit long registers **zmm**. This extension is so advanced that it provides specialized instructions for neural networks.

After this section reader:

- should understand how to use basic arithmetic operations from SSE extensions,
- know how to manage memory used for floating-point numbers,
- understand how SIMD (packed) instructions work,
- be aware of advances in SSE extensions in recent years.

Postface

It is good to have an end to journey toward, but it is
the journey that matters in the end.
Ursula K. Le Guin

Our not-too-long but not-too-short journey through microprocessor engineering ends here. Thank you for staying with me through all these pages and for turning blind eye to all inaccuracies that I surely missed in the effort of writing this book. If you wish to share your views about the book you may contact me via e-mail lukasz·makowski at ee·pw·edu·pl.

We revised fundamental aspects of numeral systems and boolean logic. We analysed hardware aspects of modern computing machinery and briefly discussed methods by which processors communicate with hardware. Then we discussed how to prepare environment in which practical aspects of programming may be learned. In the last part, and the longest one, we picked some more interesting instructions from x86-64 architecture and tested them in practical programs.

This book was not intended to cover all possible material, which is hardly feasible in any book. Its goal is to open door beyond which there is a path of increasing experience and further improvement of skill. In these final words I would like to encourage readers so that you dare to search for knowledge that takes form of bits, bytes, and electric signals. I wish you all the best in your future software and hardware projects.

Lukasz Makowski
Warszawa, May 2019