Architecture and Systems

Stefan Schwoon, ENS Cachan, Cours L3, 2014/15

October 20, 2014

This document covers part of the material of the course on *Architecture and Operating Systems* at ENS Cachan in 2014/15. It will evolve as the course continues.

1 Logical gates

Computers need to store and process information. The very first computers were based on mechanical, then electro-mechanical devices, then vacuum tubes. Since the mid-1950s, most computers are based on various types of transistors.

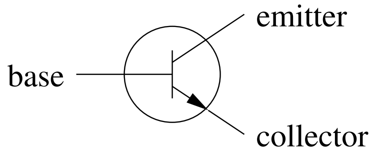

Figure 1: A transistor.

Figure 1 shows a transistor. If the voltage at the base is low, the transistor *blocks*. If the voltage at the base is "high", i.e. surpasses a certain threshold, then electric current can flow between collector and emitter.

1.1 Logical circuits

Transistors can be used to implement logical circuit. Consider Figure 2 (a). If the voltage A at the transistor base is low, then the transistor blocks, and voltage C at the output is high. If voltage A is high, then the transistor unblocks and the current at C approximates that of the ground (low). Thus C is the "inverse" of A. If we associate high voltage with value 1 or logical value *true* and low voltage with 0 or *false*, then $C = 1 - A$, or C is the logical inverse of A. Figure 2 (b) shows a block symbol for a NOT circuit.

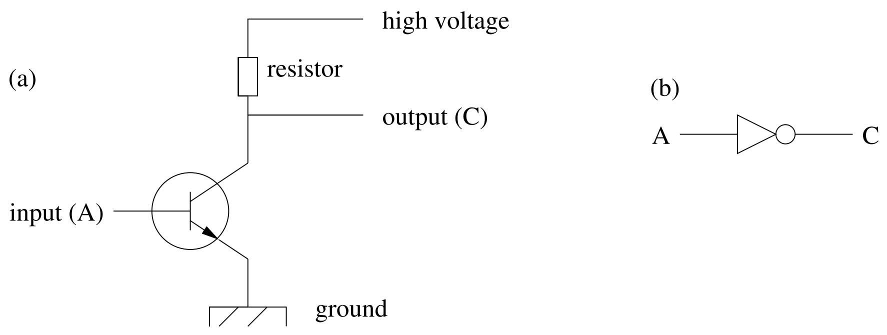

Figure 2: (a) Transistor used to realize a NOT-gate. (b) Symbol for NOT circuit.

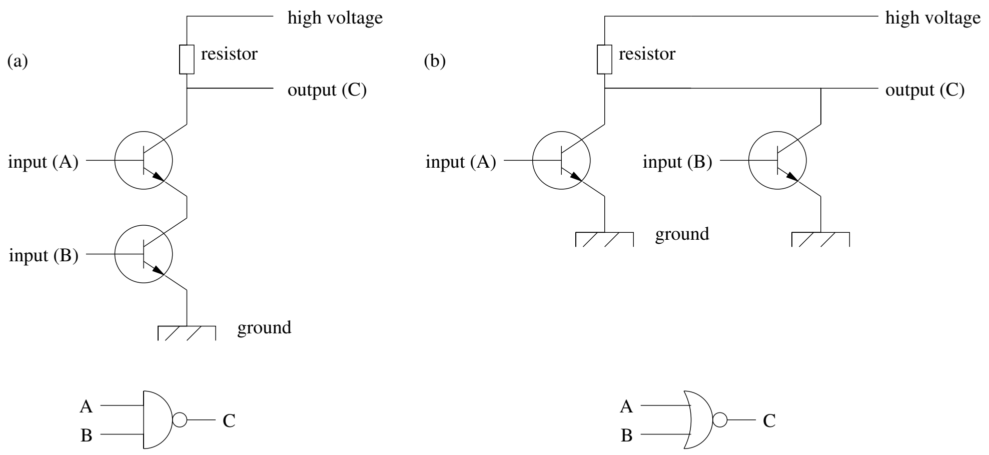

Figure 3: Circuits realizing (a) NAND gate; (b) NOR gate.

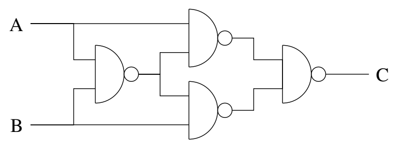

Figure 4: An XOR gate realized with NAND gates of depth 3.

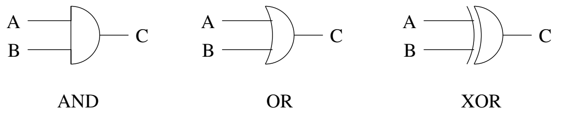

Figure 5: Symbols for additional logical gates.

Further logical functions can be realized, e.g., as shown in Figure 3, where in (a) we have $C = \neg(A \wedge B)$ and in (b) $C = \neg(A \vee B)$. Notice that either NOR or NAND are sufficient to implement any logical function. For instance, using $\bar{\wedge}$ to signify NOT-AND, we have

- $\neg A \equiv A \bar{\wedge} A$;
- $A \wedge B \equiv (A \bar{\wedge} B) \bar{\wedge} (A \bar{\wedge} B)$;
- $A \vee B \equiv (A \bar{\wedge} A) \bar{\wedge} (B \bar{\wedge} B)$.

Similar laws hold for NOR. As an example, Figure 4 shows how to implement an exclusive-or (XOR) gate using only NAND gates. Figure 5 shows some more symbols commonly used to denote logical gates.

Many different circuits can be used to implement the same logical functions. It is interesting to optimize circuits with respect to (i) their size, i.e. the number of logical gates (transistors, respectively); (ii) their depth, i.e. the length of the longest path from an input to an output, measured in terms of the number of logical gates along the way. The size of a circuit affects the cost of building it. The depth affects the speed with which outputs can be produced. For instance, the XOR gate from Figure 4 has a depth of 3. Note: An XOR gate can also be realized from NOR gates with depth 3, but only with five gates. If one allows AND and OR gates with arbitrary number of inputs (by generalizing the circuits from Figure 3), then any logical function F with n inputs can be realized with constant depth by exploiting the conjunctive/disjunctive normal form of F. However, this may lead to circuits with an enormous size and is hardly practical. The best compromise can hope for in general is to build circuits of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$.

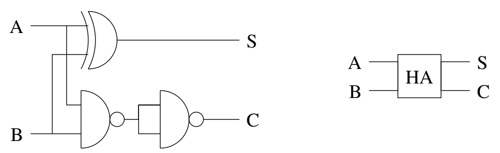

Figure 6: A half adder and its symbol.

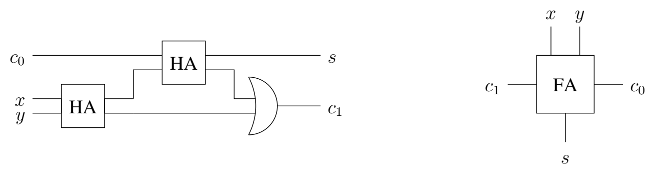

Figure 7: A full adder and its symbol.

1.2 Adder

We shall discuss how to implement arithmetic functions using logical gates. Consider Figure 6, which shows a so-called *half-adder*; the *sum* S contains the sum (modulo 2) of A and B and C the so-called *carry.* If . signifies concatenation, then we have $(C.S)_2 = A + B$, where the subscript 2 means the interpretation in binary.

Two half-adders can be combined to form a so-called *full adder*, which provides the sum and carry for three bits (two bits x, y) and a carry c_0 from another addition as in Figure 7; here we have $(c_1.s)_2 = x + y + c_0$.

Suppose that we have two 4-bit vectors x, y with $x = (x_3.x_2.x_1.x_0)_2$ and $y = (y_3.y_2.y_1.y_0)_2$. The addition of x and y can be realized by a chain of full adders, as shown in Figure 8, where $(c_4.s_3.s_2.s_1.s_0)_2 = x+y+c_0$. The symbol on the right of the figure shows a symbolic representation of such a multi-bit adder, where multiple lines of inputs/outputs are indicated by a crossed line.

If the circuit in Figure 8 is generalized to n-bit vectors, it has size and

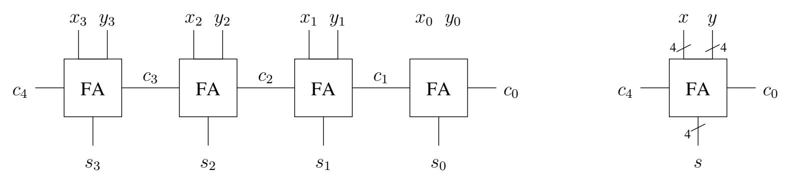

Figure 8: Adding two 4-bit numbers with a sequence of full adders.

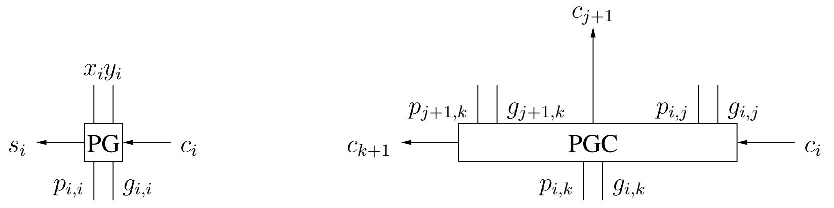

Figure 9: Symbols for propagate/generate/carry computation for 1 bit (PG) and combining two blocks (PGC).

depth $\mathcal{O}(n)$. The latter is undesirable because it would mean that addition becomes slower when a computer can process more bits. We shall discuss a realization with logarithmic depth. It is based on the principle of computing, for 'blocks' of increasing size within x and y, whether those blocks *generate* or *propagate* a carry bit. E.g., let $0 \leq i \leq j < n$, then we are interested in the block consisting of $x_j \cdots x_i$ and $y_j \cdots y_i$. Then we say that this block generates a carry if $c_{j+1} = 1$, independently of the other bits in the input. And we say that the block propagates a carry if $c_i = 1$ implies $c_{j+1} = 1$. For instance, if $i = j$, then $g_{i,j} = 1$ (the generating bit) iff $x_i + y_i \geq 2$, and $p_{i,j} = 1$ iff $x_i + y_i \geq 1$; both can be easily implemented with logical functions. Moreover, the results from two blocks can be combined, for $0 \leq i \leq j \leq k < n$, we have $g_{i,k} = g_{j+1,k} \vee (g_{i,j} \wedge p_{j+1,k})$ and $p_{i,k} = p_{i,j} \wedge P_{j+1,k}$. Also, we can compute the carry c_{k+1} from c_i by $c_{k+1} = g_{i,k} \vee (c_i \wedge p_{i,k}$.

Consider Figure 9, where the circuit PG symbolizes a full adder combined with the aforementioned computation of the propagate/carry bit for one pair of inputs x_i, y_i. Moreover, PGC denotes the combination of two blocks and the carry computation. Then an n-bit addition can be performed by the circuit shown in Figure 10 for 8 bits. This circuit has logarithmic depth if the input is generalized to n bits. Indeed, it can be seen to function in two phases, both of logarithmic depth. In the first phase, all propagate/generate bits are be computed using $\log_2 n$ PGC circuits. The second phase generates the carries by propagating them through the PGC circuits to the PG circuits. Indeed, the carries c_1, c_2, c_4, c_8 become available one step after all propagate/generate bits are available, c_3, c_6 two steps later, and c_7 three steps later Thus, the second phase also has logarithmic depth. The summation bits (not shown) are produced when all carries are available.

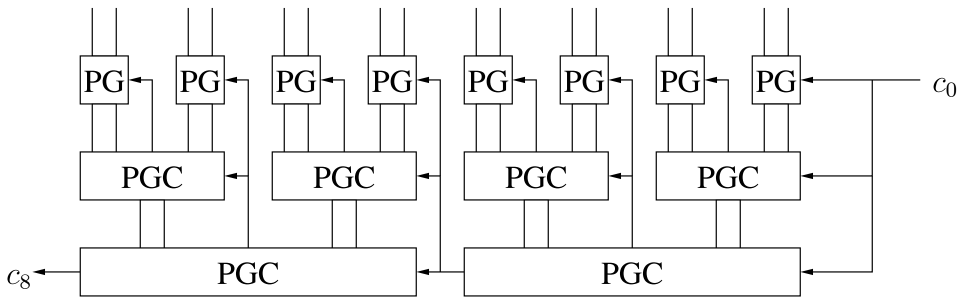

Figure 10: 8-bit adder with logarithmic depth (output of summation bits not shown for space reasons).

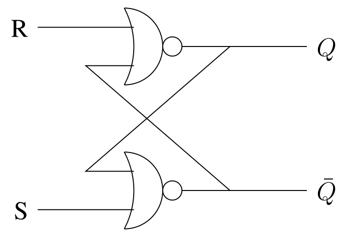

Figure 11: An RS-latch with reset (R) and set (S) inputs.

1.3 Flip-flops and registers

Flip-flops (also called *latches*, fr. *bascules*) are used to store one bit. They are realized by circuits with feedback. Figure 11 shows a so-called RS-latch. If $R = 1$ and $S = 0$ (reset), then the circuit will stabilize with the outputs $\bar{Q} = 1$ and $Q = 0$. If $R = 0$ and $S = 1$ (set), then the circuit will stabilize with $\bar{Q} = 0$ and $Q = 1$. If $R = S = 0$, the circuit is quiescent, its outputs remain as they are. Thus, a bit can be set to 1 or 0 by raising either R or S for a short time, then lowering the signal. If $R = S = 1$, then $\bar{Q} = Q = 0$, and once R and S are lowered, the outputs may oscillate; this input is thus forbidden.

Several variations of latches exist. In order to provide reliable operations in a computer, it makes sense to synchronize all circuits and allow writing operations only at specified times. This is usually done by employing an *oscillator*, that emits electric signals at regular intervals. Figure 12 shows a synchronized *D-latch*. The input C is expected to come from an oscillator, and the Schmitt trigger outputs a 1 signal only when the signal from the oscillator is in its rising flank. Thus, the signal D is translated into set/reset

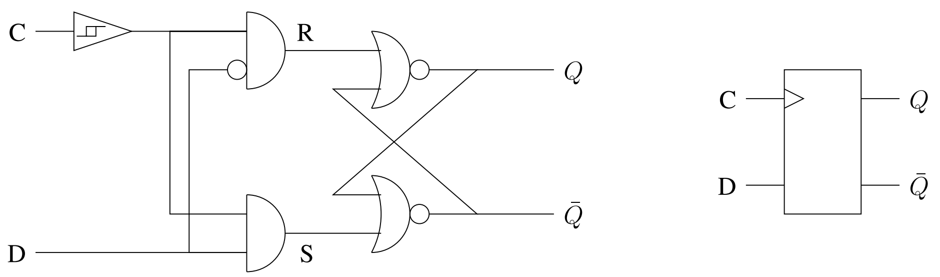

Figure 12: A D-latch with clock signal and its symbol

signals only during the rising flanks of the oscillator, i.e. at regular intervals the value of D can be written to the latch.

We can now imagine a *register* inside the CPU (processor) or a word in main memory to be realized as a vector of D-latches. Further extensions are generally used, e.g. D-latches are often extended with an *enable* signal (C and D are only taken into account if *enable* is set).

1.4 Multiplexer and decoder

This section discusses some general-purpose circuits that are useful for describing the architecture of a processor.

An (n, m)-*multiplexer*, for some $m, n \geq 1$, takes as inputs 2^n vectors $x_0, \ldots, x_{2^n-1}$, where each vector is composed of m bits, and a *selector* s, i.e. a vector of n bits. It outputs the vector x_i, where i is the value of s interpreted as a binary number. For $m = n = 1$, s is just one bit that chooses between two bits x_0 and x_1, which can be easily realized by AND and OR gates. The extension for arbitrary values of m consists merely of duplicating this circuit for different pairs of inputs. When n becomes bigger, multiple such circuits can be used in sequence, selecting first according to the lowermost bit of s, then the next most significant etc. A schematic symbol of a multiplexer is shown on the left of Figure 13.

An n-*decoder* is a circuit that takes as input an n-bit vector x. All output lines $y_0, \ldots, y_{2^n-1}$ are 0 except for x_i, where i is the value of x. Such a decoder is shown in Figure 13 on the right-hand side.

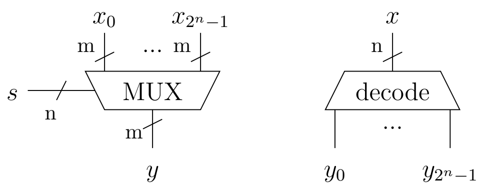

Figure 13: A multiplexer (left) and a decoder (right).

1.5 Further arithmetic-logic functions

Next to addition, other arithmetic (multiplication, division) and logical functions are necessary for the functioning of a computer. Multiplication can in principle realized using multiple additions although there exist more efficient ways to do it. Comparison between two numbers (bigger, equal, smaller) can be easily implemented using standard logical circuits.

Among the logical functions, we mention the bitwise functions (AND, OR, etc). E.g., the AND function applied to two vectors $x = x_n \cdots x_1$ and $y = y_n \cdots y_1$ is $x_n \wedge y_n \cdots x_1 \wedge y_1$, which can be realized with a constant-depth circuit that performs all AND operations in parallel.

Finally, a *d-shift* operation on a vector $x = x_n \cdots x_1$ consists of moving the contents of x by d positions to the left/right, filling non-defined bits by 0. E.g. a d-left shift operation yields the vector $x_{n-d} \cdots x_0.0^d$, effectively multiplying the binary value of x by 2^d (modulo n). For a fixed value of d, a shift operation can be implemented with a constant-depth circuit; for an arbitrary value of d (between 0 and n), it can be done with a circuit of $\mathcal{O}(n)$ depth by combining 1-shift, 2-shift, 4-shift, etc. operations with multiplexers.

2 Basic computer architecture

We can now discuss in some detail how a transistor-based computer works. Figure 14 shows a basic computer architecture with the following elements:

- the *processor* (CPU), consisting of a *control unit*, an *arithmetic-logical unit* (ALU), several registers for storing values, a *microprogramming unit* (MP), and a clock that synchronizes all actions inside the computer;

- the *main memory*;

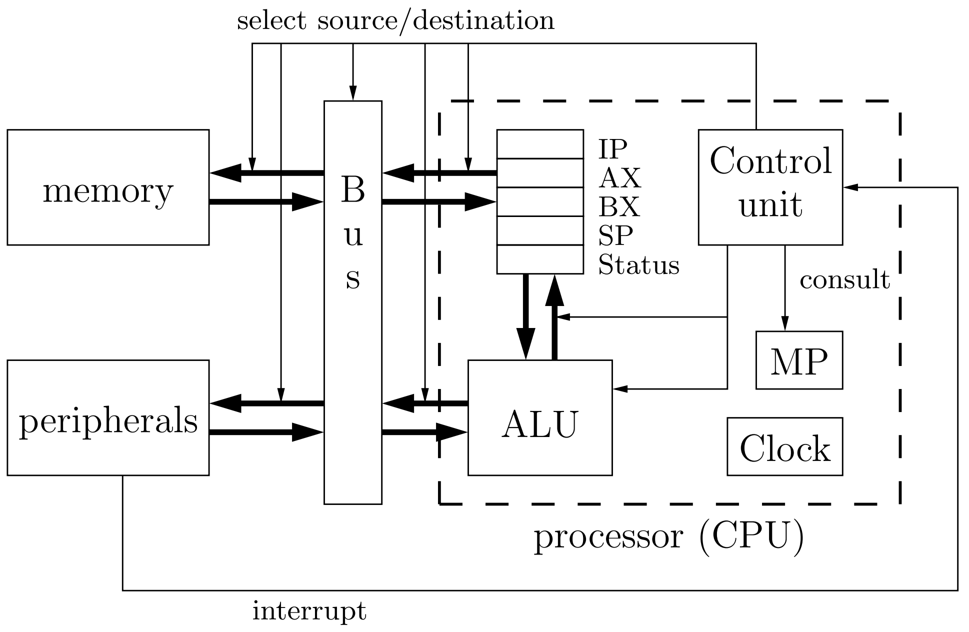

Figure 14: Basic computer architecture.

- the *peripherals*, i.e. further devices;

- the *system bus*, which serves to transfer data between the different components.

We discuss the individual components.

2.1 The memory

The memory serves to store large amounts of information, including both the instructions (*code*) and the data to be processed (this is called *von Neumann architecture*).

The memory typically consists of 2^n so-called *words* of m bits each, where m, n are some parameters. Typically, in today's computers, the basic unit of a memory is a byte consisting of 8 bits, e.g., $m = 8$. (Notice, however, that for reasons of efficiency it is often possible to obtain multiple bytes at a time, but we ignore this point in this discussion of basics.) A modern computer, even a laptop, can often store rather large amounts of memory, e.g., $n = 32$

(which corresponds to 4 GB). All storage can be implemented, e.g., using the latches discussed in Section 1.3.

A word w from address a can be obtained from memory by using a multiplexer (see Section 1.4), where a is specified by the CPU. Similarly, if a word w is to be written into memory at address a, w is obtained from the bus, a from the CPU, and a decoder (again, see Section 1.4) chooses from a the correct latches to enable the writing.

2.2 The bus

The bus serves to transfer data between different parts of the computer (CPU, memory, peripherals). This can again be realized by a couple of sufficiently dimensioned multiplexers and decoders. The source and the destination of each transfer are chosen by the CPU. To keep the bus simple, only one memory transfer is allowed at a time, so that the CPU cannot, e.g., add the contents of two memory locations immediately.

2.3 The processor

The processor is also called *central processing unit*, or CPU, for short. It contains several features:

The *registers* serve as a kind of short-term memory. They have different purposes:

- the instruction pointer (IP) contains the memory address from which the next operation (instruction) is to be obtained;
- general-purpose registers (called, e.g., AX, BX, etc in 80x86 architeture) serve to hold data;
- the *status register* is used to indicate certain events or status flags (e.g., the result of a comparison);
- the *stack pointer* (SP) points to the memory address that currently holds the top of a so-called stack (used by programming languages to implement local variables, for instance).

The *arithmetic-logical unit* (ALU) is the only part in this basic architecture that actually manipulates data. It implements the functions discussed in Section 1, e.g., addition, shift, logical and, etc. It can take one or two operands from the registers and/or the bus and write the result of an operation back to them. Operands, destination, and function to be executed are selected by multiplexers.

The *control unit* (CU) is at the core of all logical operations. For each instruction, it operates in several phases, which are synchronized by the clock signal. (The clock has connections to all parts in the computer, which are not shown in the drawing for practical reasons.) Each phase basically consists of organizing the transfers inside the computer; the CU controls the sources and destinations on the bus, the inputs and outputs of the ALU (and the function that it computes) by providing the selectors of various multiplexers on the bus and ALU, respectively.

E.g., in the first phase, the ALU transfers the value of register IP into the address selector of the memory, and it programs the bus so that it transfers a word from memory into a special instruction register inside the CPU (not shown). At the same time, the value of IP is incremented. Then, several more phases can follow. Let us consider the case where the instruction is `add ax,[0x20]`, i.e. the value of memory address $32 = (20)_{16}$ is to be added to the value of register AX and to be stored in the same.

- In the second phase, the CU programs the multiplexers connected to the bus and the registers so that the contents of AX and memory cell 32 are provided as operands to the CPU; moreover, the function *addition* is chosen by suitably programming the function selector inside the ALU.
- In the third phase, The CU instructs the ALU to transfer the result to AX, again by suitably programming the output selectors of the ALU.

The CU is then ready to process the next instruction.

In order to provide the correct selectors to all components concerned, the CU used the so-called *microprogramming unit* (MP). This is a read-only memory, which basically contains a lookup table specifying, for each instruction and each phase, the correct selection bits that must be supplied to bus/ALU/registers etc.

2.4 The peripherals and interrupt handling

Peripherals is a summary name for all other devices connected to the CPU that provide input and output. They include, for instance, the keyboard, the mouse, the display, the hard disk, printers, network connections etc. In order to communicate with these devices, the CPU arranges for words to be transferred (via the bus) to and from those devices; special CPU instructions such as `in` and `out` exist for this purpose.

While we can (in a first approximation) assume that the CPU can always send data to any peripheral, the peripherals need to obtain the attention of the CPU so that it can arrange the data transfer in the other direction. For instance, a keyboard does not have data to send all the time, but only when the user presses a key (which is rare, relative to the speed of the CPU). For this purpose, each of the peripherals possesses a wire (the so-called *interruption signal*) that is connected to the control unit (or some special register, alternatively). By raising the voltage on this wire, a peripheral signals to the CPU that its attention is required.

The behaviour of the CU is then modified as follows: Whenever a new instruction is to be executed, the CU first checks whether any interrupt signal has been raised. If this is not the case, it executes the next instruction as normal. Otherwise, it chooses a peripheral that has raised its signal and *interrupts* its own work to communicate with that device first. The current instruction pointer is saved on the stack and replaced by the address of a device-specific *interrupt handler*, obtained from a fixed part of the memory that is set aside for this purpose. The code in the interrupt handler then may then communicate with the device, organizing the transfer of data, store it in memory etc. On the communication is done, the interrupt signal is reset to 0, and normal execution resumes.

3 Data representation

This section concerns some facts about the representation of various types of data within a computer.

The following names are widely used to denominate the units of information stored and treated within a computer:

- A *bit* is the most basic piece of information, either 0 or 1. It is stored by a latch (see Section 1.3).
- A *byte* is the smallest addressable unit in the computer's memory, typically consisting of 8 bits (some historic computers had less than that).
- A *word* may denominate any vector of bits treated or transferred as a unit. E.g., a *register word* may refer to the contents of a register, or

a *bus word* the unit of data transferred on the system bus. Without addition, *word* usually means a register word.

3.1 Integers

Integers are stored in a word with a fixed size, say n bits. In C, various data types for integers exist: `char`, `short int`, `int`, `long int`, `long long int`. Their sizes are not fixed by the C standard, only their *minimal* sizes, with are 8, 16, 16, 32, 64 bits, respectively. (In the machines available in the computer room, `int` is actually 32 bits.)

In word with n bits, one can store 2^n values. Thus, these types store only a subset of the actual integers. The `unsigned` variations of these types store values from 0 to $2^n - 1$. Arithmetic operations like addition/multiplication etc., are implicitly carried out modulo 2^n.

All data types also come in `signed` variations, being able to store positive and negative integers. A simple encoding is to simply use the most significant bit as the sign. In this case, one can use n bits to store values from $-2^n - 1$ to $+2^n - 1$, with 0 having two representations.

However, the encoding used in practically all computers today is called *two's complement*. It stores values from -2^n to $2^n - 1$. Words where the most significant bit is 1 are interpreted as negative numbers, and for a positive value i (between 1 and $2^n - 1$) we can obtain $-i$ by substracting 1 and then inverting all bits. E.g., for $n = 8$, the binary representation of $i = 3$ is 00000011, and the representation of -2 is 11111101 (the inverse of the binary representation of $3 - 1 = 2$). Alternatively, $-i$ can be obtained from the *unsigned* binary representation of $2^n - i$.

Thus, if we take the binary representations of i and $-i$ and add them, we obtain 2^n, which is 0 (modulo 2^n). The two's complement therefore has the great advantage that on the binary level, all arithmetic operations are exactly the same as in the unsigned case. The computer then does not need to 'know' whether a word represents an unsigned number or one in two's complement - this becomes merely a matter of interpretation of the result.

3.2 Big-endian vs little-endian

Typically, the size of a word is bigger than that of a byte (e.g., 32 vs 8). This means that a word takes up several bytes in memory. This poses the question how the contents of the word are distributed in memory.

Big-endian means the convention of storing the most significant byte first. E.g., a word whose hexadecimal representation is $89abcdef$ is stored as four bytes, in the order 89, ab, cd, ef. *Little-endian* means the inverse, the least significant byte is stored first, in the example ef, cd, ab, 89.

This issue becomes important when data is to be exchanged between computers that might employ different standards. Whenever binary data is written into files or read/written from/to network connections, it should be converted to a so-called *network format* (a standard that happens to be big-endian). In C, the functions `ntohl` and `htonl` exist for this purpose (the name means *network to host/host to network* for *long* integers).

3.3 Floating-point numbers

For practical reasons, real numbers, like integers, are represented by words of a fixed size. Thus, only a fraction of real numbers can be represented in any given format. The most frequent standard for representing real numbers is IEEE 754, which consists of several parts. We discuss just the standard for $n = 32$ bits, corresponding to the C type `float`.

The `float` data type represents real numbers in *floating-point* notation. In general, a floating-point number is a tuple (s, m, e), where s is the *sign*, m is called the *mantissa* and e the *exponent*. Depending on s, the value of (s, m, e) is $\pm 2^e \cdot m$. For instance, $(+, 1.25, 2)$ represents the number 5.0

Naturally, not all reals (e.g., π) can be represented like this. Also, such a representation is not unique, e.g. (s, m, e) represents the same number as $(s, 2m, e - 1)$. For this reason, some normalizations are introduced. E.g., in IEEE 754, the mantissa is always in the range $[1, 2)$, which makes the representation unique for all non-zero values. In the standard for 32-bit `float`, the most significant bit is used for the sign, 8 bits for the exponent, and 23 bits for the mantissa.

The following conventions are used for interpreting these bits:

- A sign bit of 1 means *negative*.
- The exponent is computed by subtracting 127 from the aforementioned 8-bit value. Thus, the exponent has the range ± 127. (The value 128 is used for special purposes, like $\pm\infty$ or errors.)
- The most significant bit is interpreted as having value 1/2, the next bit as 1/4 etc. Finally, one adds 1 to the number thus obtained to be in

the range $[1, 2)$.

3.4 Error detection and correction

Computers store large amounts of memories. Due to electrical glitches or mechanical imperfections, the value of a bit (e.g., stored in a latch) may change unexpectedly. Likewise, when transmitting data over a network, line noise may corrupt the data being sent. We discuss some methods to guard against these errors. They consist of complementing the data bits by additional *control bits*. These control bits are never seen by the programmer or user, only used internally by the computer.

Naturally, it is impossible to guard against all possible errors. E.g., if data and control bits are corrupted at the same time, then an error may pass without notice. Thus, most schemes are meant to guard against errors under certain assumptions, e.g., that no more than one or two bits in a word may be corrupted.

Parity bit. The most basic method is that of equipping each word with a so-called *parity bit*, which indicates whether an even or odd number of data bits is 1. E.g., for a 7-bit word 0101010, the parity bit is 1 (because there is an odd number of 1s). For 0011000, the parity bit would by 0 (because there is an even number of 1s. Thus, there is always an even number of 1s if the parity bit is included in the count.

Parity bits can detect an error when a *single bit* is corrupted, but a corruption on two bits will pass unnoticed. Also, it is impossible to find out *which* bit has been corrupted.

Hamming code. We discuss a method that removes this flaw. For a word of size n, it introduces roughly $\log_2 n$ parity bits. These will allow not just to detect errors if up to two bits are corrupted, but also to repair a word if only a single bit is corrupted.

Let us consider the so-called $(7, 4)$-Hamming code, meaning that for 7 data bits, one adds four parity bits. We will number the bits (data and parity) from 1 to 11. Then, the parity bits are stored in positions 1, 2, 4, and 8. The parity bit at position p counts the parity of all positions that in whose binary representation p is set. E.g., the parity bit at position 1 is for the odd positions, the parity bit at position 2 for positions 2, 3, 6, 7, 10, 11, the parity bit at position 4 for 4, 5, 6, 7, etc.

Example: Let us consider the 7-bit data word 0101110. It is represented as an 11-bit word $p_1p_20p_4101p_8110$. We can now compute the parities, and the word becomes (with parity bits in bold): **10**0**0**101**0**110

If a single bit (data or parity) is corrupted, we can identify its position by considering the set of parity bits that have become incorrect. E.g., if the bit at position 10 changes, then the parity bits p_2 and p_8 become incorrect. We can thus deduce the position of the incorrect bit and correct its value.

If two bits (data or parity) are corrupted, we can no longer uniquely identify their positions. Worse, if the two corrupted bits are both parity (e.g., p_2 and p_8), we may deduce that a single data bit (e.g, at position 10) has been corrupted. To further guard against these types of mistakes, the Hamming code uses an additional parity bit that ranges over all data and parity bits alike.