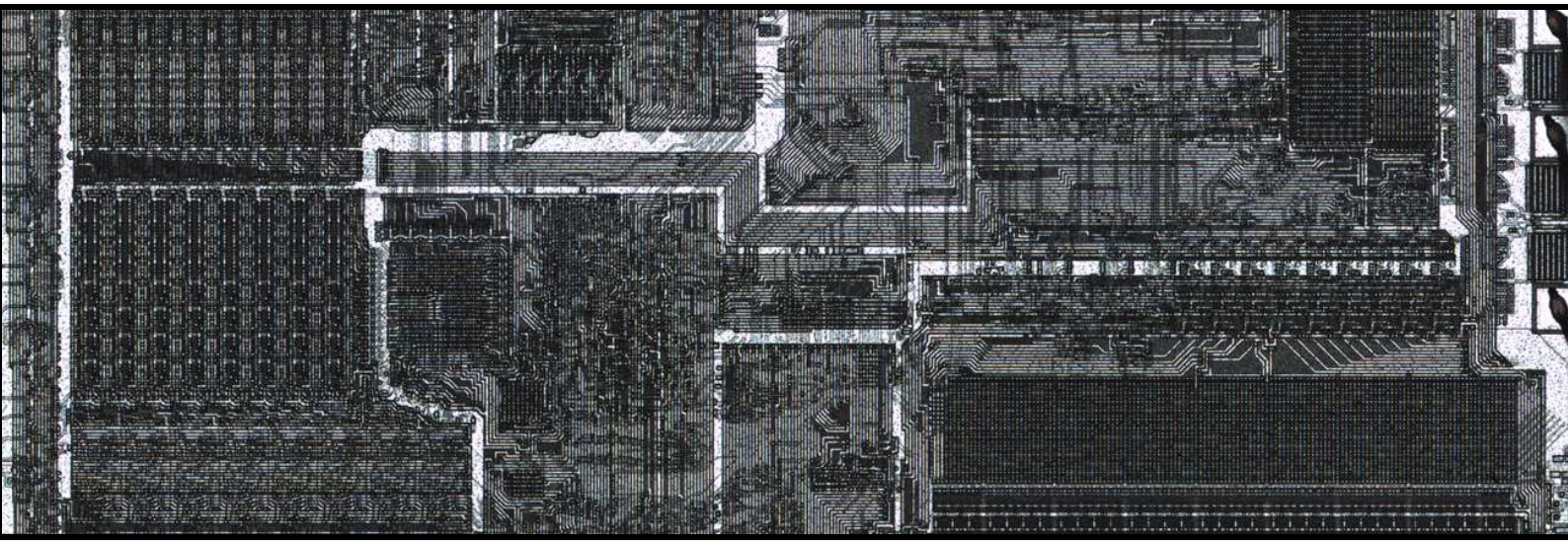
Alessandro Pellegrini • Bruno Ciciani

# Computing Architectures

A Bottom-Up Approach



Alessandro Pellegrini • Bruno Ciciani

# Computing Architectures

A Bottom-Up Approach

First edition: – – –

Printed in the World

# Contents

# 1 Introduction to Microprocessors

The current technology involving computers requires its users to understand both the hardware and the software that they will be working with. In fact, hardware and software coexist at various levels, and being able to understand their interaction allows to design applications which are *efficient* from different perspectives. While in the past decades extreme importance was given to the *performance* of an application, namely "how fast" the program was able to transform some input into the expected output, or "how fast" a system was able to respond to user interactions, current trends require applications to be efficient as well as from the point of *network latency* (when dealing with distributed systems or web-based applications) or even from an *energy* perspective. Very often, having a clear understanding of the interactions that the software we write has with the underlying hardware can help a lot in the design of applications which embrace all of the efficiency aspects.

Many times developers of applications forget that, in the end, their software will be run by a *microprocessor*. In fact, at the heart of any computing application, we find a *central processing unit* (the CPU), which *interprets* programs. Although different implementations of different CPUs can end up with an extremely varied efficiency (again, in terms of both performance and energy), the same theoretical foundations apply, and their effects have become pervasive in every aspect of our society. There are automatic systems which monitor road traffic, and allow us to reroute to minimize the time spent in traffic; satellite applications which are able to forecast the upcoming weather, so that we can safely organize our holidays; extremely advanced mobile phones which give us access to any sort of information in any place; ATM machines which give us money any time we need (at least if we have some to withdraw…); lightweight laptop computers which allow us to work anywhere (this was really science fiction only 40 years ago). And this is just to mention few applications which we encounter in our daily life.

More in general, applications of computer systems can be grouped into several classes. *Desktop computing* is likely the class which we know better. This is an application of computer systems which give a large number of possibilities (office production, gaming, music, …) to a single user or to a small group of users, keeping the cost somewhat reduced. *Server computing* is at the opposite side, offering large-scale machines which target the execution of programs requiring a lot of processing power, or

a lot of memory to process large data sets. To this class of computing systems users often have only access through a networked interface. *Embedded computing*, on the other hand, comprises very small devices which require a very reduced amount of power to run, but are often tight to a specific (or a small group) of applications. This is usually what we find in cars' engine control units, or in portable media players. Nevertheless, the current *miniaturization trend* has given us embedded systems which cost always less and are more and more *general-purpose*, such as Arduino or Raspberry PI, just to mention some of them. This has lead to the *Mobile computing* paradigm as well, where the always more powerful mobile phones and/or tablets can be used for purposes different from the originally-designed one. Finally, a recent trend in computing is *Cloud computing*, which tries to reduce the cost associated with the maintenance of a cluster of server machines, adopting the *pay-as-you-go* model, where the user pays only for the amount of time that a given machine is used. Cloud computing has been made possible as well by advancements in software, due to the fact that the machines offered to the users are not real machines, rather they are virtualized by a set of software components, often referred to as *hypervisors*. Anyway, all these application would not exist if microprocessors weren't at their heart, in some place.

This brief overview of modern computing applications is inevitably incomplete, but tells that computer science is a quickly evolving field. This quick evolution comes from both software and technological (that is, at the level of the hardware) advancements. For example, over the past 45 years, the total number of transistors available on a microchip has doubled every 18–24 months, a trend which is known as *Moore's law*. This law is just an observation by Gordon E. Moore, the co-founder of Intel, of a trend which was taking place in the 1960's, but it has not significantly changed yet. Think, for example, that the hardware equipment used by the Apollo 11 to send the first man on the moon was exaggeratedly less powerful than a common mobile phone that we have in our hands.

This yielded a proportional increase in a single processor's clock speed which, in the past decades, was bringing an enhancement in the computing speed which users were receiving for free, whenever they were upgrading their hardware. Improvements in algorithms and/or code optimizations were not strict requirements to be pursued, specifically for common and non-mission-critical[1] applications, because—as the time was passing by—the software was just working more and more efficiently.

This quick evolution was related to the fact that microprocessor developers have been always adopting new technologies to produce CPUs. In particular, newer technologies were involving smaller components. Having a smaller component is a benefit from several points of view, as it allows to cram more components in a single area unit: therefore, we can either add new functionalities in the same component, or make the original units faster. Or we can combine both benefits at the same time. Figure 1.1 shows this scaling: in particular, it shows the minimum size of a transistor feature (in $\mu m$) over the years.

A *transistor* can be seen as a "switch" which can be either *on* or *off*, and its state is controlled by an electric signal. By the data shown in Figure 1.1, we can see that the size of a transistor has become extremely small, on the order of $0.01\mu m$. This means that a single chip, created from one slice of silicon, can host up to several billions of transistors, and is therefore called *very large-scale integrated*

[1] A mission-critical application is an application whose failure may compromise some goal-directed activity, which could possibly involve fatalities. An example of a mission-critical application is a navigational system for a spacecraft.

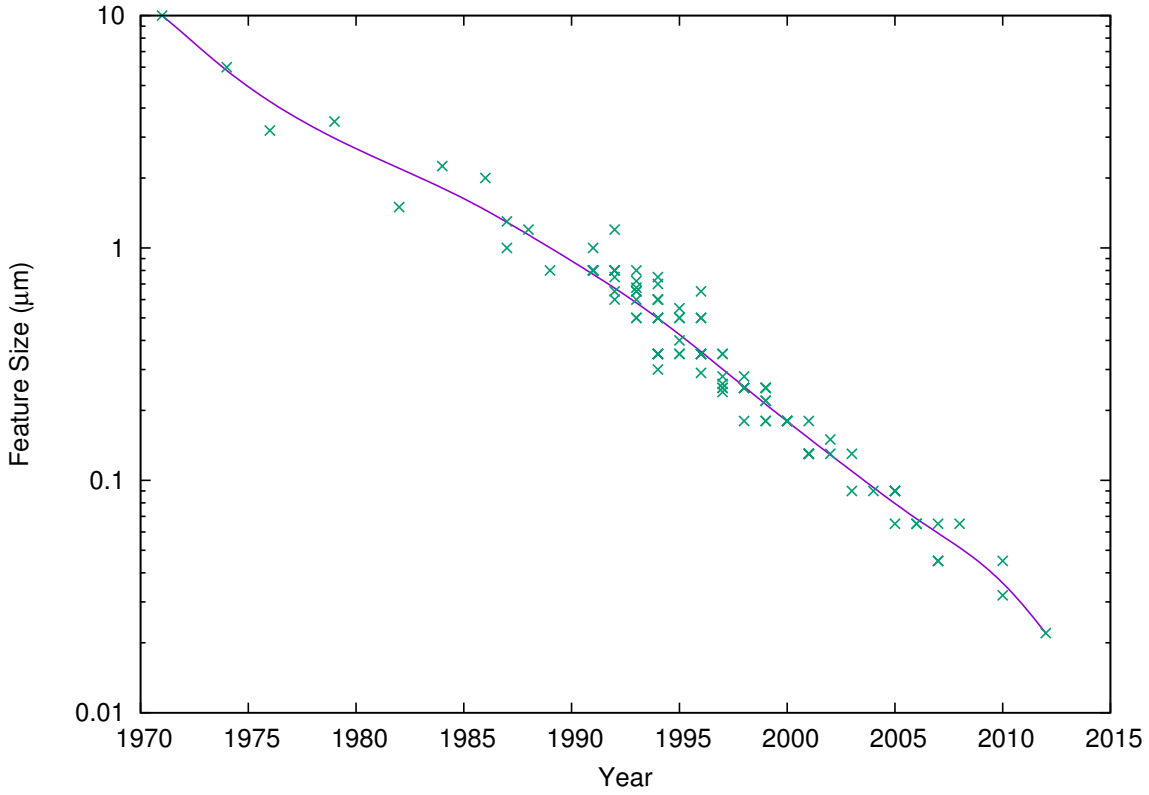


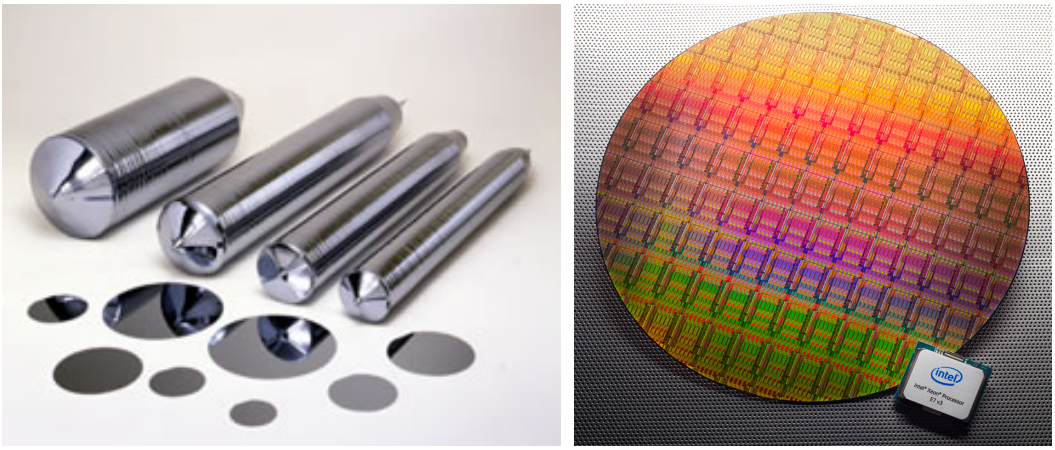

**Figure 1.1:** Technology Scaling: the minimum size of a transistor feature over time.

*circuit* (VLSI).

A chip is produced from silicon, a substance found in sand. Silicon is not a good electricity conductor, and is therefore called a *semi-conductor*. To transform a piece of silicon in a chip, several mechanical and chemical steps are carried out. In the beginning, the silicon is organized as a *monocrystalline silicon ingot* or *boule* (see Figure 1.2(a)), which consists of pure silicon, where the crystal lattice is continuous, unbroken, and free of any grain. It is cut into slices of few hundred microns, usually using a diamond saw, which are called *wafers* (see again Figure 1.2(a)). Wafers are then processed chemically, according to a specific pattern, to deposit substances to create conductors (using microscopic copper wires), insulators, or transistors. This process is called *doping*, or more informally *printing*, since the process is also called *photography*.

A processed wafer is shown in Figure 1.2(b). In the optimal case, this wafer can be cut into different 84 Intel Xeon CPUs. Nevertheless, even a small defect in the silicon crystal, or the presence of small particles of dust on the surface of the wafer during its processing, or even the smallest imperfection in the pattern used to print the wafer can produce *defects*, which will prevent the final chip from working properly, due to the effects on the local electronic properties of the material. Therefore, once a wafer is printed, a first testing occurs, which allows to identify these defective chips, so that they are later discarded in the productive process. Looking at Figure 1.2(b), it is interesting to note that the edge of the wafer contains "incomplete" chips. These are of course not expected to work properly, but are anyhow printed because this makes the creation of the final pattern easier.

The wafer is then cut into different *dies*, which is what we usually refer to as a chip. After this



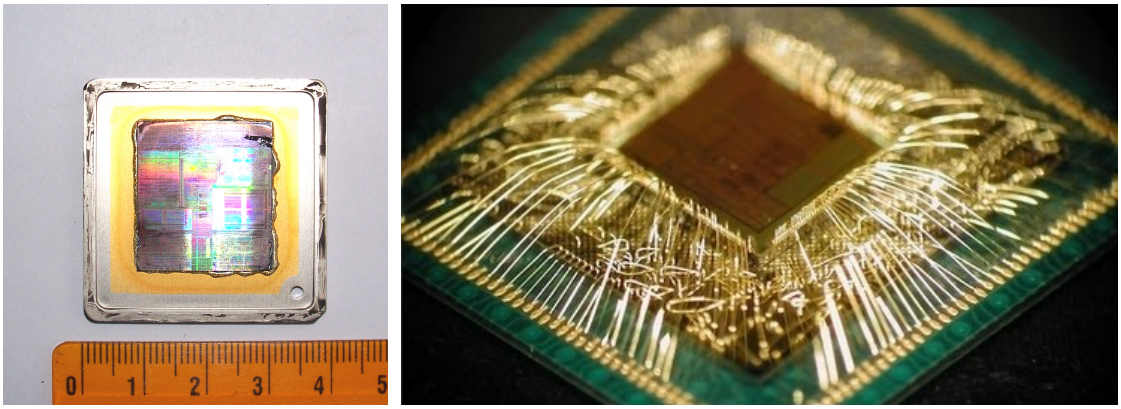
(a) (b)

**Figure 1.2:** From the silicon ingot to the packaged chip.

step, all the defective dies are discarded, and the amount of discarded chips is measured as the *production yield*, which is the percentage of working dies per each wafer. To keep this percentage as high as possible, special environments called *clean rooms* are specifically built with a controlled level of contamination, namely the number of particles of a given size (such as dust, or microbes) per cubic meter. To give an example, ambient air has commonly 35,000,000 particles per cubic meter of $0.5\mu m$ and larger. A clean room used to manufacture semiconductors has no particles of that size, and only 12 particles per cubic meter of $0.3\mu m$ and smaller. This is done by filtering air coming from the outside of the clean room, and the air inside of the clean room is constantly recirculated. Despite this, production yield of semiconductors can be as low as 30%—although this measure accounts as well for impurities in the monocrystalline silicon ingot.

The next step in the construction of a microprocessor (and generally of any chip) is packaging. A single die can be very small, with a size on the order of few millimeters. The picture on the cover of this book shows a magnified portion of an Intel 8086 die, while Figure 1.3(a) shows the die of an Intel Xeon in its packaging. The size of the Xeon die is $22 \cdot 23mm$, having a surface of $503mm^2$. This size is too small to make any connection with other components of the computing architecture, so the purpose of the packaging is to attach several wires, called *bond wires*, to specific points of the die and to the *pins* of the packaging, as shown in Figure 1.3(b). This process is not 100% safe, as new imperfections could come into the chips (bonding is usually a mechanical process), some bond wires might be missing, or the analog performance of the die could be altered by the package. Therefore, the last step of the production, called *final testing*, involves testing again the chip after it has been packaged. Only working chips are delivered to final customers.

When it comes to measuring the efficiency of a particular computing system, several measure units come handy. On the performance side, a common unit is the *FLOPS*, an acronym for FLoating-point Operations Per Second. This measure, which is particularly useful to understand how a computing system is useful for scientific applications, tells how many floating-point operations the computing system is able to carry out every second. With several approximations (for example, not accounting for memory and the time required to access it), the FLOPS of a given computing system can be com-



(a) The die of an Intel Xeon in its packaging.

(b) Wire bonding using gold wires.

**Figure 1.3:** A die in its package.

puted, in the general case, using the following Equation:

$$FLOPS = cores \cdot clock \cdot \frac{FLOPS}{cycle} \tag{1.1}$$

A common measure for modern off-the-shelf CPUs is to carry out 4 floating-point operations for each clock cycle. Thus, a single-core CPU with a 2.5 GHz clock can reach up to 10 billions FLOPS, that is 10 GFLOPS. Nevertheless, as we shall see throughout this book, modern CPUs have a clear division between the subsystem which carries out floating-point operations (which is called the Floating-Point Unit—FPU) and the subsystem which carries out the rest of the operations (which we will call the Processing Unit—PU). Therefore, FLOPS are a good measure of the performance of the FPU, rather than the whole CPU. In any case, for some applications which require a large number of FPU operations (these may be scientific applications, simulators, or game engines) this is still a relevant measure.

The performance of other applications, which don't rely heavily on floating-point operations, can be measured in terms of *Instructions per Second* (IPS). Again, this measure does not take into account memory-access latency, and furthermore it is non-trivial to select an application which exhibits a so-varied execution pattern to be representative of the actual performance. This is the main reason why there are several standard benchmark suites which can provide performance measurements to compare compute-intensive workloads on different computer systems. One of the most-famous benchmark suites for CPUs is the SPEC's benchmark, provided by the Standard Performance Evaluation Corporation (SPEC). This suite encompasses different applications which provide different execution profiles, and are explicitly thought to stress both the FPU and other parts of the CPU. Moreover, this set of applications is continuously evolving, so that if new computing architectures cannot be satisfactorily assessed with the current ones, new ones are introduced.

On the other hand, when it comes to measuring energy efficiency, a common measure is *Performance per Watt*. Literally, it measures the *rate of computation* (also called *throughput*) that can be delivered for every Watt of power consumed. Of course, this measure strongly depends on the measure used to compute the performance. Therefore, similarly to what we have just discussed, we can

talk about FLOPS per Watt, or Instructions per Watt. Similarly to the performance case, there are specific benchmarks to measure the energy efficiency of a system, such as the SPECpower or the Average CPU Power (ACP).

Currently, we have already reached computing systems which are able to reach a performance in excess of one PetaFLOP, that is a *quadrillion* floating-point operations per second. An organization called Top500.org is currently monitoring the newcoming computing systems which are able to reach or even break this performance record, although among these computing systems we often find large cluster of high-end machines. The fact that we already have systems able to reach several PetaFLOPS of performance has named our current era the era of *petascale computing*. Current applications of petascale computing involve the advancement of computations in fields such as weather and climate simulation, nuclear simulations, cosmology, quantum chemistry, lower-level organism brain simulation, and fusion science.

Nowadays, we're facing another transition. In particular, we are already producing computing systems able to break this performance, and we're thus transistioning from petascale to exascale systems. Nevertheless, in this ultimate transition we are facing new challenging issues. In fact, even though the previous transition (from terascale to petascale) could directly benefit from the aforementioned implications of Moore's law (namely, a new single microprocessor with a larger number of transistors was directly more powerful), we are now hitting limits imposed by fundamental physics, as shown by the *dynamic power equation*:

$$P = ACV^2 f \tag{1.2}$$

where $P$ is the power consumption, $A$ is the activity factor (i.e., the fraction of the circuit that is switching), $C$ is the switched capacitance, $V$ is the supply voltage, and $f$ is the operating frequency.

As we have shown in Figure 1.1, technology was able to scale because, while increasing the total number of transistors per chip, it was decreasing their size and capacitance. At the same time, an increase in the overall frequency was counter-balanced by a decrease in the supply voltage. This process, known as *Dennard scaling*, was actually the electrical basis for Moore's law.

In Figure 1.4 we see the trend of off-the-shelf processing units during the last 45 years. Moore's law's effect is clearly visible in the first part, as the number of transistors in a CPU keeps doubling, and the frequency follows the same trend. Yet, around year 2003 something happened: although the number of transistors presents the same trend, clock speed increase has stalled. This is connected to the fact that switching noise in the circuits poses a limit to supply voltage decrease, and current leakage (due to the extremely small size of transistors) causes the chip to heat up, requiring a flattening in the clock frequency to keep Equation (1.2) in balance. In fact, $130\,W$ of power consumption in a processor is considered an upper bound, the so-called *clock-frequency wall*: an increase in the clock frequency would create an unacceptable power consumption.

To overcome this limitation, since users are always demanding for more powerful microprocessors, the industry has dived into the *multi-core era*, where multiple CPU cores—highly similar to a traditional CPU in their internal design, yet equipped with an internal intercommunication network—are placed within the same chip. Although this new computing architecture presents new and more complicated aspects (which are nevertheless out of the scope of this book), it again allows us to progressively increase the available computing power. It is even interesting because, to some extent, the

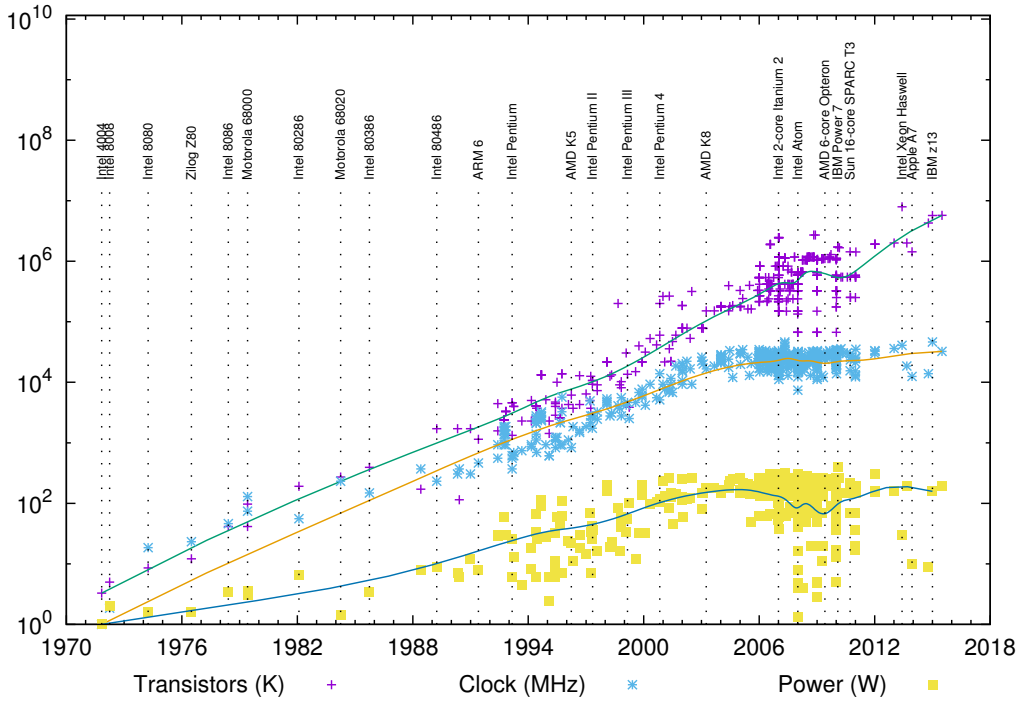



**Figure 1.4:** CPU specification trends: number of transistors, clock frequency, and power consumption. The dates when important historical CPUs have been introduced are marked on top of the plot.

implications of Moore's law have moved from describing the clock frequency increase of a single core, to counting the number of available cores in a single CPU.

Let it be a single processor or a core in a multi-core processing unit, the fundamental unit used in computer science is, again, the microprocessor. A microprocessor is a computer processor that incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits. According to at least the last 50 years of technology, a multiprocessor is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output. It is an example of sequential digital logic, as it has internal memory. Microprocessors operate on numbers and symbols represented in the binary numeral system.

Therefore, a microprocessor is a logical structure which interprets programs written by humans, which are translated into a machine-readable set of operations (the *machine language*) by a special program, the *assembler*[2]. As we have said at the beginning of this chapter, a microprocessor alone is of scarce utility. In fact, since its goal is just to interpret operations, the hardware cannot live without the software. A simplified scheme of the logical structure of the hardware and software structure is shown in Figure 1.5. This is a hierarchical structure, and is enforced by most of the modern general purpose computing architectures. In the outer rim we find *software applications*, which are the applications usually implemented by programmers, and which we are most accustomed to. A set of

[2]The action of transforming a program into the machine language is actually more complex if the original source of the program is written in higher-level languages. In this case, the assembler is only the last actor in the whole *compiling* operation.

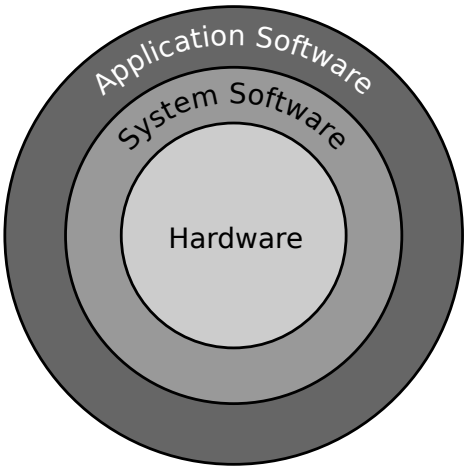
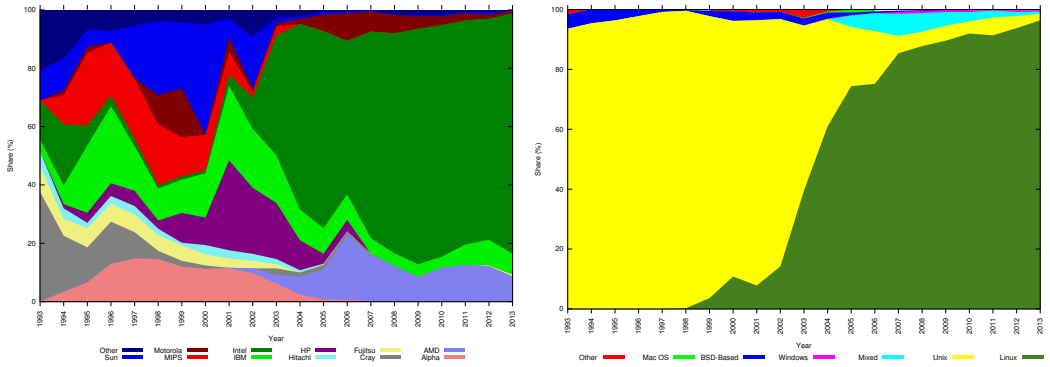



**Figure 1.5:** Simplified Hardware/Software coupling.

software components, on the other hand, are at a lower level, namely the *system software*. This is what is often referred to as the *operating system*, which is essentially a low-level software acting as an interface from the software to the underlying hardware. An operating system can be seen as a *compatibility layer*: in fact, the underlying hardware can be extremely different, offering different facilities which are triggered using different instructions (namely, different *machine instructions*). When we implement a software in a high-level language, we would like to make it run on *any* hardware family. This has been made possible, during the years, exactly through the operating systems: application-level software directly interacts with the operating system, which in its turn knows what is the currently used hardware, and translates the requests coming from the software into the specific operations that the currently-available hardware can understand. Therefore, to make the same software run on different hardware, the same (high-level) source code can be used, provided that it is *recompiled* for the correct target architecture and that an operating system version for that hardware is available. The operating system will as well be able to manage memory and interact with external devices in a uniform way, although memory and device implementations can be incredibly varied.

Throughout this book we will deal with software directly written in *assembly language*, and we will study how to design a CPU which is actually able to interpret the translation of the assembly instruction into specific machine instructions. This will be done so as to show how it is possible to programmatically drive the execution of a hardware device. There are many interconnections here between the hardware and the software, so we will build the final CPU step by step, introducing first the general architectural abstractions, then the basic hardware components, then we will show how we can represent various types of data in memory. Only after this we will start to show assembly instructions, their machine-instruction counterparts, and finally we will start mixing together the basic components in order to build a CPU which is able to interpret machine instructions.

In this process, we will concentrate on an assembly language which is proper of Intel CPUs. This language has been first devised (in a preliminary version) for the Intel 8080 CPU, back in the 1970's, and was then settled for the Intel 8086, a 16-bits CPU dating back to 1978. This assembly language has then been evolved, yet mostly in a full backwards-compatible way. Therefore, although with several shortcomings, a program written for the Intel 8086 could be theoretically run on more modern




(a) Vendors Share

(b) Operating systems share

**Figure 1.6:** Top 500 share of CPU vendors and Operating System families.

CPUs. This assembly language, along with the machine representation used to let the CPU interpret the instructions (which, similarly, has always been evolved in a backwards-compatible way) is often referred to as the x86 Instruction-Set Architecture (ISA). The careful reader should have already well understood that what the CPU actually interprets is the machine representation of the instructions, not the actual assembly instructions. Therefore, multiple assembly languages could be mapped to the same ISA. This is exactly what happens with the x86 architecture: different assembly languages exist, which are often referred to as *dialects* of the same ISA. The dialect that we will be using in this book is the AT&T one, which was developed at Bell Laboratories. Although this is not the "official" Intel syntax, it is the one used in most Unix systems to write assembly code for the x86 architecture. There is a historical reason behind this, as Unix was first developed at Bell Laboratories as well. Therefore, in the modern world, most Windows systems rely on the Intel syntax, while most Unix systems (GNU/Linux systems as well) use AT&T syntax. Due to the fact that the source codes for many Unix-like operating systems are open, becoming familiar with the AT&T syntax can be useful as well for operating system courses. Furthermore, AT&T syntax is sometimes a bit more cumbersome than Intel's, so we expect the transition from AT&T syntax to Intel's to be easier, should it be needed.

It is important to emphasize that the methodologies explained in this book are anyhow of general applicability. In fact, assembly languages for other CPU families, although different, must share the same principles, because CPUs must interpret programs which, despite their complexity, use most basic concepts. It is not a "learn-one/use-all" deal, but after having understood the basic principles, switching to programming a different CPU is not as complex as starting from scratch. The choice of using Intel CPUs with the AT&T syntax is further motivated by the fact that this synergy allows to directly program a very large part of modern most-powerful computers in the world. In fact, as it can be seen in Figure 1.6[3], more than 90% of 500 most powerful supercomputers in the world run Linux, and around 80% of them bundle an Intel CPU. If we sum to this latter percentage the number of AMD-based supercomputers (AMD shares most of the Intel ISA, so the same assembly language can be used to program them) the total count reaches 90% of available supercomputers.

Nevertheless, the x86 ISA is too complex to start with, especially the most modern implementa-

[3] The data to draw this plot has been obtained from Top500.org.

tions. Therefore, we will implement a subset of the machine instructions (and we will study only the assembly instructions related to them). To account for the modern time, this subset will be able to handle data up to 64 bits, and will allow us to interact with any kind of external device. This "reduced" ISA will be called the z64 ISA, and the corresponding CPU that we will be building throughout this book will be called the z64 CPU. The z64 has the advantage that any program written for it can be immediately re-assembled on an x86 CPU, although some caveats are necessary—mostly because modern operating systems are in between our software and the underlying hardware.

As we have said, a microprocessor is useless without appropriate software to drive its execution. Therefore, we will deeply study how to program a CPU using assembly language. Writing a computer program is like writing a recipe. If you have ever read a recipe, then you know its structure: you have the ingredients at the top, and below them you have the directions for how to deal with those ingredients. A computer program, especially an assembly program, shares much with this: at the beginning you have the stuff that you will be dealing with (the data), and below you will have the instructions to operate on them. The CPU will read the recipe and will follow through all the steps to cook the ingredients up. You are the head chef and you write the recipe. The CPU is more likely the sous chef, which does all the busy work. If the recipe is wrong, then the dinner won't be quite good. But for the recipe to be good and precise, you must first know how your kitchen is like. You could ask the CPU to mix all your vegetables using a blender, but what if your kitchen does not have one? This is exactly why, in order to write good programs, the programmer must perfectly know the kitchen he's cooking in: the underlying computing architecture.

## 1.1 Computing Architectures

A computing architecture is the description of the functionality, organization and implementation of a computer system. This description can come at two different levels: on the one hand we might find a description of the capabilities of the programming model, without a specific implementation. On the other hand, description of real-world computing architectures involves the ISA design, and the microarchitecture design—namely the actual implementation of the CPU internals, which is able to run the machine operations specified by the ISA.

In this Section we will be dealing only with the first category. The first computer architecture which we can find in history dates back to 1830's, when Charles Babbage and Ada Lovelace started describing the analytical engine. This was actually the first mechanical general-purpose computer, where specific input (both the data and the program) was given to a machine in the form of punched cards. This machine had a specific store, similar in concept to the modern memory, which was able to keep up to 1,000 floating-point numbers of up to 40 decimal digits. An arithmetic unit, called the *mill* was able to perform all four arithmetic operations, plus some comparisons and (in some designs of the machine) the square root. Similarly to how CPUs are designed today, the mill was relying on an internal program to carry out the operations, and this internal program could be altered by relying on several pegs. The programming language, represented by the punchs on the cards, was extremely similar to modern assembly languages, supporting branches and loops.

Another important example in history is the *Automatic Computing Engine* by Alan Turing, which

was proposed late in 1945. This architecture still used the term "engine", exactly in honour of Babbage's analytical engine. In fact, similarly to this initial design, the automatic computing engine was relying on punch cards to store the input data and program, and the engine was designed to be extremely general-purpose. This engine was actually the evolution of previous work by Turing himself, who started to study in 1936 a universal computer machine, later known as the Turing Machine. Nevertheless, due to secrecy on the work that Turing was carrying out, he was prohibited to explicitly tell that he was aware that such an engine could be implemented as an electronic device.

The same year, a bit earlier than Turing's Automatic Computing Engine, John von Neumann published a work titled "*First Draft of a Report on the EDVAC*". This document was made up of 101 pages, and it was mostly incomplete because it was handwritten by von Neumann while traveling on a train. Nevertheless, it already showed the complete logical organization of the elements of what later became the ENIAC, and paved the way to the definition of the *von Neumann architecture*, which is the logical organization that many modern CPUs still rely on. Similarly to previous proposals, the computing machine (which von Neumann divided into a *central arithmetic part* and a *central control part*) were relying on punched cards or magnetic wire to store the program and the data. Memory was again playing a major role, as it was used to *load* the program and the data from cards/wire, and to keep intermediate data required to transform the input into the desired output.

A different architecture which has been discussed in more recent years is the Harvard architecture. The most important difference between the Harvard architecture and the von Neumann one lies in that the former uses two different types of memory, one to keep data and one to keep the program. Although the difference could look small, this proves significant when considering the latency required to access the program and the data. In fact, dividing the two memories could end up in a significant speed increase, as those two different operations can be carried out concurrently. Modern CPUs do implement this kind of division internally: while the main memory is one, CPUs have two different internal memories (which we call *caches*) for data and programs.

All these architectures share anyhow one common principle: the program and the data is stored on some external medium (punched cards, magnetic wire, or disks/flash memories in more recent years), which is then loaded internally in the computing architecture, and finally executed. This makes all these architecture fall into the category of *stored-program computers*, which is a computer that stores program instructions in electronic memory. This was already described in Turing's 1936 idea of the universal computer machine, which had been surely studied by von Neumann as well. This makes a large difference with early computers, which were not reprogrammable, and were only able to run the same program a large number of time, possibly only changing the input data. The first computer which is universally recognized as being the first one able to run a stored program was the Small-Scale Experimental Machine (SSEM), built at the University of Manchester, which run its first program on 21 June 1948.

While Turing's work has had a great influence on the theory of computability and on theoretical computer science, von Neumann's and Harvard architectures are the most influential if we consider modern CPUs, so that is why they deserve a more in-depth discussion, before plunging into the design of an actual ISA and of a real implementation of a CPU.

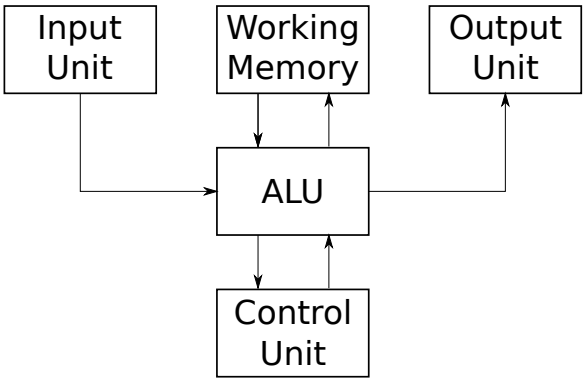



**Figure 1.7:** Von Neumann Architecture

### 1.1.1 Von Neumann Architecture

The basic components of a machine built according to the von Neumann architecture are, as shown in Figure 1.7:

- One input unit;
- One output unit;
- Working memory;
- An Arithmetic/Logical Unit (ALU);
- A Control Unit (CU).

The input unit allows to collect information from the outside world. Since the von Neumann falls into the category of stored-program computers, the input unit can be used to receive both input data and the program to be executed. Both the data and the program are stored into the working memory. This memory is a buffer of any desirable size, which is used as well to maintain temporary information which can be regarded as the result of partial processing of the program. Once the program is terminated, the user of the von Neumann architecture can be interested in retrieving the result of the computation. We note that in the most generic abstraction, the user of a von Neumann architecture can be both a human or another machine, even in a different location, connected via any kind of network. This is the purpose of the output unit: to present the result of a computation to the end user. Nevertheless, since the precise moment at which the computation is completed depends on the actual program (machines built according to the von Neumann architecture are general-purpose, thus the machine cannot a-priori know what kind of computation the user will ask the computing system to carry out), this output unit must be driven by the program itself.

The actual part of the architecture which carries out the computation is divided, as in many complex digital systems, into two subsystems: the Processing Unit (PU) and the Control Unit (CU). In the original specification of this architecture, the PU was often referred to as the Arithmetical/Logical Unit (ALU). Nevertheless, since computing systems have far evolved since the first specification, the ALU has become just a portion of the overall PU, as processing units are able to execute many more (and more complex) operations, with respect to more standard arithmetical and logical ones.

The PU is a set of logical components, like *registers*, *memory*, all the functional blocks which carry out operations on the content of registers and memory, and the interconnection structure which al-

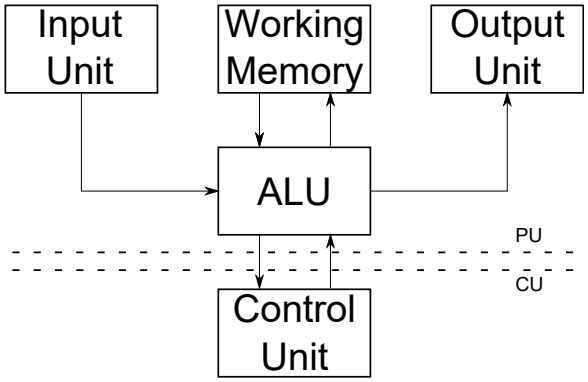



**Figure 1.8:** Specialisation of the von Neumann Machine according to the CU/PU organization.

lows to transfer data between registers, memory, and the functional blocks. On the other hand, the CU is a *sequential network*, namely a "logical set of steps and operations", which coordinates, according to the program (specified in machine language), all the activities of the PU. This coordination is carried out via *control signals*, namely pulses of frequency of electricity (in the case of electronic implementations of the von Neumann architecture) which represent a logical command for a given element of the PU. It is anyhow important to note that, in general, the activities of the CU are conditional to that of the PU: for example, after having asked via a control signal to carry out an operation, the PU might be interested in waiting until this operation is completed, before issuing the next command. Alternatively, the control signal issued by the CU might depend on the *state* of a certain component of the PU. This is done via *condition variables*, which are signals travelling from the PU to the CU.

The logical diversification of the von Neumann architecture into the CU and PU organization is depicted in Figure 1.8. In particular, generic control signals going from the CU to the PU, condition variables going from the PU to the CU, and the information flow between the various functional components are shown.

This architecture is of general applicability, as it only describes the basic components of a processing unit, without any constraint on the actual implementation. Nevertheless, taken as is, there is one shortcoming. In fact, the von Neumann architecture has a single working memory for both the program and the data. Therefore, machine instructions and data cannot be *fetched* from memory at the same time. This limits the throughput, namely the data transfer rate, between the CPU and memory compared to the amount of memory. Because program memory and data memory cannot be accessed at the same time, throughput is much smaller than the rate at which the CPU can work. This, which is called the *von Neumann bottleneck*, has given rise to the above mentioned different proposal, the Harvard Architecture.

### 1.1.2 Harvard Architecture

The Harvard Architecture owes its name to the IBM Automatic Sequence Controller Calculator (ASCC), which was called Mark I by the staff at Harvard University. The ASCC was a general-purpose electromechanical computer, developed in 1939, and used during the World War II. Ironically enough, one of the first programs to be run on top of the Mark I was written by John von Neumann.

This computer was programmed using punch cards, and it had a "surprising" computing power

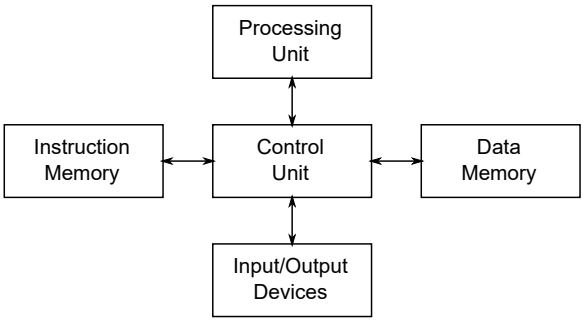



**Figure 1.9:** Organization of the Harvard Architecture.

able to carry out three addition/subtractions per second, while a multiplication lasted six seconds, and a division 15.3 seconds. It was able as well to compute logarithms and trigonometric functions, but these computations took minutes.

The most important advancement by the Harvard architecture entails having different pathways for data coming from devices, from instruction memory and from data memory, as shown in Figure 1.9, in contrast with the pure von Neumann architecture where the CPU can be accessing either data or instructions, since there is only one working memory.

In the Harvard architecture, the separation of the two pathways allows as well for completely uncoupled implementations of the two memories, which could be realized using different technologies. Moreover, this separation allows as well for the separation of the *addresses* of data and instructions, and their *privileges*: in fact, since it is not meaningful to modify the instructions of the program, having two different memories allows to have one of them (the instructions memory) which cannot be updated by the programs, which is something that can increase the security of a computer system.

On the other hand, this organization has the limit that if a program requires many instructions, but uses few data, the instruction memory could be filled up, preventing the program to be correctly run by the computer, even though there could be plenty of space in the other memory. In fact, this constraint is the reason why modern architectures do not strictly adhere to the Harvard one.

### 1.1.3 Modern Architectures

Modern architectures are a hybrid of von Neumann and Harvard architectures, often referred to as *modified Harvard architectures*. In particular, current architectures relax the strict separation between data and instruction memories, letting the computer use one single working memory for both. The strict separation is yet still enforced *inside* the CPU, where a small portion of very fast memory (called the *cache*) is organized according to the Harvard architecture and is thus divided into instruction cache and data cache. Thus, when accessing the internal cache memory, the CPU operates according to the Harvard architecture, while when accessing external main memory it behaves as a von Neumann machine.

Additionally, the pathway separation enforced by the Harvard architecture is exacerbated to the extent that many different interconnections are used according to a hierarchy which depends on the speed of the devices. A summary of this organization is reported in Figure 1.10. In this organization, the CPU is directly connected only to a unit called the *North Bridge*. This is a *coprocessor*, namely a

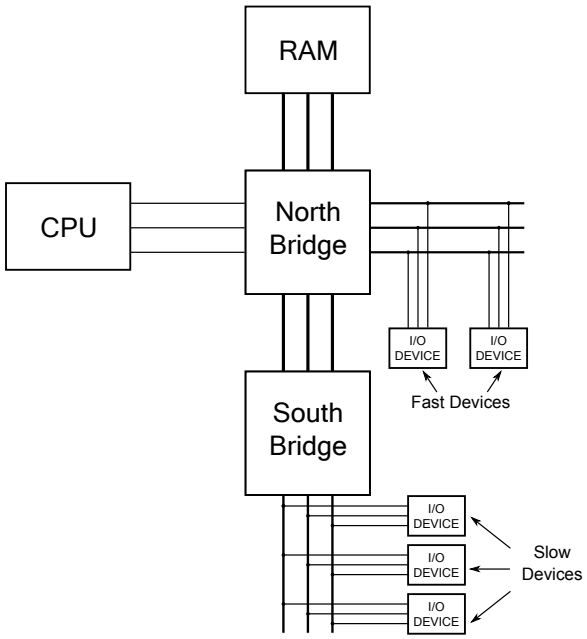



**Figure 1.10:** The organization of a modern computing architecture.

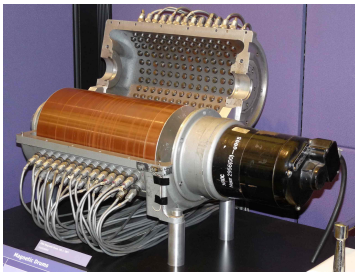
chip which does only one job, but very quickly. Its purpose is to communicate with memory, with fast devices, and with the *South Bridge*, another coprocessor. The latter is connected to all the slower devices in the system, and asks the North Bridge for permission to communicate with the CPU. The main advantage of this multiple-level organization is that the CPU is likely always busy: in fact, all the time which would be spent by the CPU waiting for a device is actually spent by either the North or the South Bridge, so that (statistically) the CPU can always find something to process in the whole system.

This is an architecture that we will build throughout this book. While we will start from a simple architecture where the CPU is directly connected to the working RAM, we will add step by step additional blocks to increase the performance of our computing architecture, eventually reaching this more complicated organization.

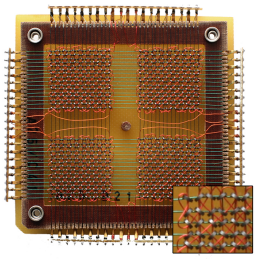
## 1.2 Working Memory

Working memory is an essential part of any computing architecture, both when organized according to the Harvard and to the von Neumann scheme. Working memory should be as large as possible, in order to keep all the instructions and all the data that characterize a program. At the same time, the technology used to build it should be as cheap as possible, otherwise large amounts of memory could not be included in a computing system.

In the early days, a computer could be equipped only with few kB of memory, due to the high production cost. Anyhow, the history of computing architectures has seen a gradual decrease in the cost per byte of memory, and this is one of the reasons why common off-the-shelf computers are now bundled with several GB of memory. This has been related to the fact that the components to build memory have been progressively miniaturized, and the production cycles have made the processing



(a) Magnetic drum memory, c. 1951.



(b) Core Memory, with detail of the toroids.


(c) Intel 1103 – DRAM Chip.

**Figure 1.11:** Early Working Memory implementations.

of raw materials always less costly.

The first implementations of working memory were using relays, mechanical counters or delay lines. The real first way to store data in a computing system in the form of working memory was the so-called *drum memory*, a large metal cylinder coated of ferromagnetic material memory invented by the Austrian Gustav Tauschek in 1932. Figure 1.11(a) shows an implementation realized in 1951. The ferromagnetic material was able to store data, in a very similar way to that of more modern spinning hard disk drives. On the cylinder, there was a row of read/write heads. The controller of the memory was simply selecting which head was to be used for reading/writing, while the whole drum was spinning.

The original drum memory by Tauschek had a total capacity of 62.5 kB. In between 1955 and 1975, a new type of memory became mainstream, called *core memory*, offering larger capacity. A large number of tiny magnetic toroids (the cores) were used to build this memory. Small wires were wrapped around them, which were used to read and write information. Each core was used to store one bit of information, having zeros or ones represented by their different magnetization—clockwise, or counterclockwise. The cores were organized in a grid, each one having wires which were able to magnetize them independently of all the others, sending pulses through specific wires. Figure 1.11(b) shows a $32 \cdot 32$ grid, able to keep 1024 bits of information, with a larger detail showing the toroids.

Core memory's functioning principle could be seen as a bit counter-intuitive in the modern days. In fact, the toroids were storing the information even when the component was powered off (thus, it was not a *volatile* working memory), but the reading process was *destructive*: reading a bit's value was clearing it, at the same time, thus any piece of information was reset after its access. Nevertheless, this technology had a large diffusion, as it reached a price as low as 1 cent per bit, and a size of 32 kbits per cubic meter was quite common.

In the late 1960's, the first Static Random Access Memory (SRAM) chip came to the market, and quickly eroded the core-memory market. SRAM is based on semiconductors, and uses bistable circuits (called flip-flops) to store bits. Differently from core memory, SRAM is volatile, meaning that if the chip is powered off, all the information is lost. Nevertheless, it was able to store much more information in a much smaller space, which was the main reason for its success. In the same years, the Dynamic Random Access Memory (DRAM) was invented. The main difference between SRAM and DRAM is density and price, the former being less dense and more costly. DRAM, which was invented

by IBM's engineer Robert H. Dennard—author as well of the Dennard's scaling model—around year 1968, came out to the market as a chip, the Intel 1103 (shown in Figure 1.11(c), in October 1970. After that time, DRAM became the most used technology to implement larger and larger working memories for computing architecture.

Independently of the technical realization of the working memory, *how* data are stored in memory determines the specific operations that a CPU has to undertake in order to correctly process the information. Usually, working memory is divided into *cells*, and the size of a single cell is usually set to one byte, composed of 8 bits[4]. Each cell is associated with an increasing number, starting from zero and reaching the last cell. This number is called the *address*, and it's used by the CPU and the programs to uniquely identify the content of the working memory. This representation is not necessarily compliant with the physical organization of working memory: while it could be divided into multiple chips, where each of them could even be of a different size, the CPU sees the union of all the physical memory as if it were organized as a unique continuous *tape*. This abstraction is referred to as the *flat memory model*, and significantly simplifies the way we write our programs, as we are only interested in the addresses, rather than where the data is physically stored—this is something which is left to the hardware for management.

Nevertheless, when it comes to represent data which is longer than one byte, there some issues come. In fact, if an address identifies a single cell of memory, which is one byte wide, what is the address of a larger data type? More in details, given an address, if we know that we will find there two consecutive bytes which represent a single piece of information, what is the *order* according to which this information is stored starting from this address? This issue, referred to as *memory endianness* (or *byte order*) is non-trivial, and might cause interoperability problems between different machines—which is something even more problematic when two different machines are connected over a network.

When we usually work with our computing systems, we are not necessarily aware of memory endianness, nor we care much about it when we write software in higher-level languages. Yet, being aware of how our architecture stores data in memory could give us much better control of what happens in our machine, could give us the possibility to write programs which exploit the actual order of bytes, or even prevent subtle bugs (especially when dealing with network programming) which could be extremely difficult to debug.

**Historical Note 1.2.1:** Endianness

In 1726, Jonathan Swift described in his satirical novel Gulliver's Travels tensions in Lilliput and Blefuscu: whereas royal edict in Lilliput requires cracking one's soft-boiled egg at the small end, inhabitants of the rival kingdom of Blefuscu crack theirs at the big end (giving them the moniker Big-endians). The

[4] Historically, a *byte* (term coined by Warner Buchholz in July 1956) is the number of bits used to encode a single character of text, and has therefore been considered the smallest unit which a CPU can process. Nevertheless, the size of a byte was considered hardware-dependent, and no standard existed in the early days to mandate its size—in fact, the term *octet* was used to identify a byte composed of 8 bits. Several architectures were using, in the early days, bytes composed of 4, 6, or 7 bits. The large development of 8-bits microprocessors in the 1970's popularized the common size of a byte to 8 bits, seeing it as a suitable power-of-two value allowing to represent integers from 0 to 255. The size of a byte was standardized to 8 bits only in 2008, with the IEC 80000-13 standard which defines the sizes of the units used in computer science.

terms little-endian and endianness have a similar intent.
Danny Cohen's "On Holy Wars and a Plea for Peace" published in 1980 ends with:

> *Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way. We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made.*

This trivial difference was the reason for a hundred-years war between the fictional kingdoms. It is widely assumed that Swift was either alluding to the historic War of the Roses or—more likely—parodying through oversimplification the religious discord in England and Scotland brought about by the conflicts between the Roman Catholics (Big Endians) on the one side and the Anglicans and Presbyterians (Little Endians) on the other.

## 1.3 Instruction Sets: CISC and RISC

When we write programs to be run by our computing architectures, we often rely on higher-level languages. These languages are forms of writing which allow us to write *directives* which we expect the CPUs to be able to carry on. Nevertheless, CPUs never speak the languages we use to write our software. In fact, CPUs are only able to understand sequences of zeros and ones, where each sequence represents a specific operation, which is often called an *instruction*. This representation of the instructions composed of only zeroes and ones is called *machine language*, and each operation is called a *machine instruction*. Thus, if the programs that we write cannot be interpreted by the CPUs which run them, there should be some sort of *translation* from what we are able to write and what the CPU can run. This process, which is called *compilation*, is left to some software, called the *compiler*, which scans through our code, trying to determine what are the operations that we are asking the CPU to carry out, and thus writing the most suitable sequence of machine instructions which correspond to our higher-level software.

There is an intermediate level to all this, which is the *assembly language*. This language is to the wider extent a programming language, which has anyhow the property that almost any statement in it corresponds to a machine instruction. Therefore, we can see the assembly language as a one-to-one mapping between machine instructions, which are composed only of zeroes and ones, and a more human-readable representation. In fact, most compilers work according to different steps: the first one, which is the proper *compilation* step, transforms higher-level statements in a set of *assembly instructions*, which are then transformed into their 0/1 representation by an additional program, the *assembler*.

But the right question now is: what are the assembly instructions that a compiler can generate starting from a given higher-level source code? Different CPUs speak different languages, so the

compiler must know what is the CPU that will eventually run a program—this is the *target* of the compilation process.

When it comes to the classification of CPUs, the two biggest families are *Complex Instruction Set Computers* (CISC) and *Reduced Instruction Set Computers* (RISC). This large differentiation tells whether a CPU can execute a large set of more complex instructions which often involve several steps of computations (CISC architectures) or only a smaller set of much simpler instructions can be run—in the latter case, the same program will be mapped to a larger number of instructions, which are usually much faster, yet.

This distinction is not very precise, but the general concept is that a system that uses a small, highly optimized set of instructions, rather than a more versatile set of instructions, is a RISC system. Conversely, the term CISC was retroactively coined in contrast to RISC, being seen as a sort of umbrella term for everything that is not RISC. Among the commercial architectures, the most famous CISC one is the Intel architecture, while many other famous brands, such as MIPS, ARM, or SPARC, fall into the category of RISC systems.

An additional categorization of computing architectures is based on the maximum number of operands that can be explicitly used by their instructions. According to this organization, the first categories embrace the so-called *zero-operands*, *zero-address*. These machines have instructions which only specify the operation to be performed, although instructions which have at least an operand, namely a memory address to retrieve data, are always present. For example, a simple operation like $c = a + b$ would be coded as:

1. read $a$ from memory
2. read $b$ from memory
3. compute an addition
4. write the result to memory to $c$

These architectures are called as well *stack* machines because they use a stack to carry out the operations. Every memory read places the read value on the top of the stack, and every executed operation reads the first one, two, or more elements on the stack. The result of every operation is put again on the top of the stack, so instructions which write to memory take the values again from the top of the stack.

The second category according to this classification is the *one-operand*, or the *one-address*, or even the *accumulator* machine. CPUs falling into this category offer instructions which specify a single operand, and were extremely common in the early days of computers, or even today in small microcontrollers (such as PIC microcontrollers). The operand specified by the instructions is called a *right operand*, meaning that it is usually a parameter which is read and not updated (as if it were on the right side of an equation). These architectures are called accumulator machines because the result of the operations are stored into one single destination, called the *accumulator register*, which is the implicit *left* operand. Again, an operation like $c = a + b$ would be coded like:

1. Load $a$
2. Add $b$

3. Store *c*

A third category is that of *two-operand* machines, and most of CISC and RISC machines fall into this category. Architectures developed according to this scheme refer to one of the operands of the instruction as the *source* operand, while the other is called the *destination* operand. Usually, an operation involves processing both the destination and the source operand, but the final result of the operation is stored into the destination operand, thus overwriting its content. In this situation, thus, it is not common to directly code an operation like $c = a + b$, since the most effective way would be to rewrite the code to have $a = a + b$. In any case, if the desired operation is exactly $c = a + b$, this would be coded as:

1. Add *a*, *b*
2. Copy *a* to *c*

if *a* is the destination operand. In any case, the content of the destination operand (*a* in the given example) would be overwritten.

The latter category is the *three-operand machines*, which try to overcome the fact that the destination operand's content is overwritten, and allow to specify an explicit left parameter of the operation. In this situation the operation $c = a + b$ would be directly encoded as:

1. Add a,b,c

Unlike two-operand or 1-operand architectures, this organization leaves all three values, *a*, *b*, and *c* available for further reuse, and thus could simplify the development of applications. Therefore, from a high-level point of view, three-operand architectures could look more promising. Nevertheless, instructions are in the end transformed into zeroes and ones. Representing three operands in a single instruction is "bit-consuming", and therefore many 16-bits CPUs are invariably two-operand machines. On the other hand, many 32-bits RISC CPUs are three-operand CPUs, such as the SPARC architecture, the MIPS architecture, the ARM architecture and the AVR32 architecture.

Intel's architecture is a different story, as it is a hybrid of the various categories. It is mostly a two-operands architecture, although there are some instructions which accept three operands, some which operate according to an accumulator-like strategy and thus accept only one operand. Other instructions accept no parameters, but in the reality they take them implicitly, as the operands could be specified only in a unique way, and thus the instructions do not explicitly mention them.

Consequently, additional operands to the operation could be implicitly specified (this is even more true for one-operand architectures, where the accumulator is always an implicit operand). Thus, the operation expressed by the instruction can have a number of parameters which is different from the one required by the logical or arithmetic operation associated with it, meaning that the *arity* of the instruction can be different from that of the operation.

Some CISC machines offer more than three operands, such as the VAX "POLY" polynomial evaluation instructions, but these are more exceptions than the rule.

Throughout this book, we will carefully look into many of Intel's operations, and the careful reader will be often able to notice this in practice.

# 2 Data Representation

Computers are ubiquitous and we depend on them for many everyday tasks because they are extremely *fast* at processing data. In fact, the only thing that a CPU can do is to process some data, according to the instructions given by the programmer. Both the instructions and the data are stored into one or multiple working memories. The fundamental question now is: *how* are data and instructions stored in memory? How can the CPU *differentiate* between data and instructions when reading something from memory?

The answer to the latter question is trivial: it can't. A computing system uses the *binary system* to represent and process data, so that every instruction, word, letter, or number is composed of a sequence of (fixed- or variable-length) sequences of zeroes and ones, namely a sequence of *bits*. A bit is the smallest piece of information that is required to distinguish between two "states", which could be the classical *Boolean values* (*true* and *false*), signs (+ or -), *yes* or *no*, *on* or *off*, and/or any other binary quantities defined by the programmer. A computer elaborates data represented using bits relying on a set of circuits which are mostly based on *storing* bits and *comparing* bits, in different ways. The fact that a CPU uses only two elements to represent the data is one of the reasons why computers are so fast—by far faster than men. In fact, computations have to account only for two symbols (which are mapped to two different levels of electric voltage) and allows to have circuits which are relatively simple. The decimal system which is used by humans, on the other hand, requires ten different symbols, and devising circuits which work at ten different levels of voltage is much more complicated.

The simplicity of the bit must be anyhow mapped to more complex data. We use computers to process text, images, integer number, real numbers, and so on. These data are represented *concatenating* multiple bits into what are often called *words*. A word is a sequence of bits, of a fixed length, which represent some data. The length of a word depends on the architecture, so that different CPUs can have words which are composed of a different number of bits. In the `z64` architecture, a word is composed of 16 bits, while 8-bits data are called a byte, conforming to the general usage of this term. When a CPU reads a sequence of bits from working memory it cannot tell the kind of data it is loading, because it only knows an initial address and a size. The interpretation of a sequence of bits

in memory is entirely up to the programmer. This means that the same sequence of bits could be interpreted in many different ways, for example the same sequence could be a positive integer number, a real number, or a pixel of an image. The only assumption the processor makes is that when it loads a word from the memory, and that word is representing an instruction, then it is a valid instruction.

We now have the bit to play around with, so let's play around a bit with it. In the following sections we will illustrate how we can use sequences of bits to represent different *data types*. Due to the diverse nature of data, we will see in the following Chapters that the CPU offers various instructions to manipulate them, exactly depending on the kind of involved data. In particular, since the CPU has no clue about what it is processing, the programmer has to chose the proper instructions which can manipulate the data in a coherent way.

## 2.1 Numeric Representation

Since computers represent data using bits, the representation of all numbers uses binary arithmetic. While this is perfectly true for integer numbers, both positive and negative, real numbers use a more complicated representation, which deserves a specific discussion. Of course, numbers (meant as *quantities*) exist before being bound to a specific representation, but a representation is needed by humans or machines in order to manipulate them. We never see a number, we only see its representation, but *any* representation is perfectly suitable to represent a number, and moving from one to another does not change the essence behind it.

Binary arithmetic could be considered tough, yet the decimal system which we are accustomed to is not as "natural" as we might think. History has seen many ancient civilizations which were using other numeral systems, such as the Babylonians, Chinese, and Mayas, who were representing numbers using only a smaller amount of elementary digits. To reach the modern decimal system, a long evolution took place, and this system has not completely superseded all previous ones. Consider, for example, time, angles, or geographic coordinates, which use the sexagesimal system, where 60 symbols are used to represent numbers. This numeral system dates back to the Sumerians in the 3rd millennium B.C., and passed through the Babylonians, to reach the modern age. Figure 2.1 reports the Babylonian numerals, which were used to count up to 60.

Another non-decimal numeral system which is still very in place are the British Imperial units—which takes as reference some common lengths of parts of the human body, for example the inch, the span, the foot, or the cubit—or the United States customary units—which are derived from the British Imperial units, but yet have some differences. This is something which could create even communication problems when people using different systems have to interact (think, for example, of a US guy telling a European guy that he should dress properly because the temperature is 30 degrees: the disaster is just behind the corner!).

A traditional numeral system still in place in Asia uses 12 elementary digits—this is called a *duodecimal system*. This originates from the fact that you can use your thumb to point at each bone of your other fingers in 12 different ways. Saan people in Africa can count only up to five. Pygmy from Africa, Xavantes from Brasil, and Aranda from Australia can count only up to two, and after that they only say "many". The importance of the number "two" is reflected as well in several ancient languages,

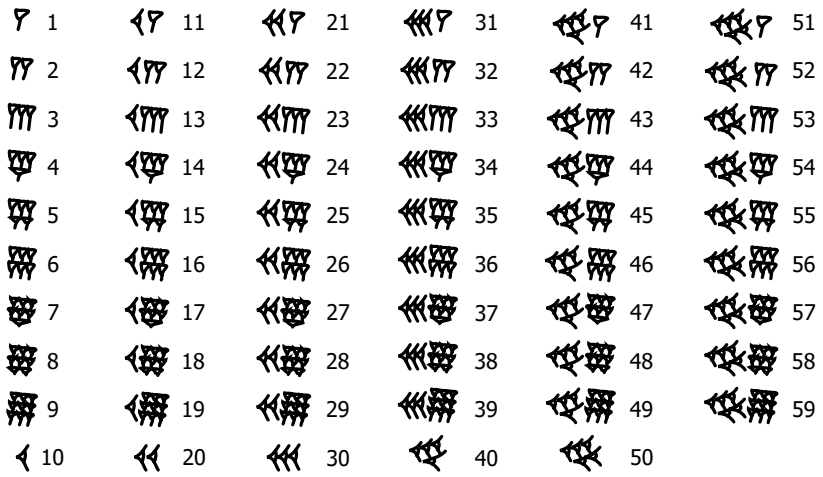



**Figure 2.1:** Babylonian numerals were used to count up to 60.

where the declension of names/adjectives and the conjugation of verbs had the *dual* form (as in Greek, Hebraic, or Arab), while some tribes in Oceania have even the *triple* form and the *quadruple* form. The introduction of the zero, associated with a null quantity, has its roots in India and then in Arabia only in recent times. In a world which is so chaotic about numbers, having computers which "reason" in binary shouldn't cause an uproar.

The fast manipulation of numbers is high on the list of things computers are good at. The simplest computations that computers can do are on integer numbers, therefore we shall start to understand how integer numbers can be represented using only binary digits.

### 2.1.1 Integer Numbers

Integers come in different variants and sizes. On the one hand, we have the *natural numbers*, which belong to the set $\mathbb{N}$. This set comprises all numbers from zero to $\infty$, and are commonly called *positive* numbers, while in computer science they are referred to as *unsigned integers*. On the other hand, we have the set $\mathbb{Z}$ of *integer* numbers, which start from $-\infty$ to $\infty$, and they are commonly called in computer science *signed integers*, or more shortly *integers*. Nevertheless, when we represent numbers in a computer, we use a sequence of bits, and representing infinite numbers would require an infinite sequence!

There is an upper bound, then, on the largest representable integer number, both in the signed and unsigned variant. This upper bound is related to the maximum length of the sequence of bits which the CPU can handle when performing numerical operations. While this is somewhat architecture-dependent, the common agreement is that we can have *long integers* which use 64 bits, integers which use 32 bits, *short integers* which use 16 bits, and bytes which use only 8 bits. Thus, if we consider only positive numbers, a 16-bits short integer can represent all the numbers in the range $[0, 2^{16}]$, meaning that we are not able to represent a number larger than 65.535 (since we have to represent the zero as well). This is something that a computer does not care about, as this is the only way it is able to work, so it is the responsibility of the programmer to choose a numeral representation which can store all the numbers which are of interest for the program being written.

Since we are so accustomed to the decimal system when it comes to counting, understanding the way integers are represented in binary passes through understanding how numbers are represented in decimal. The decimal system is a *positional* system: the basic symbols are only 10, from 0 to 9, but their actual value depends on the position they occupy in the notation. Not every numerical system is positional. Think, for example, of Roman numbers, where the symbol I, representing 1, can be added to the value of another symbol when it is placed on its right side (as in VI, 6), or subtracted when it is placed on the left side (as in IX, 9). The Arab numeral system[1], which is the most used nowadays, uses the rightmost digit to represent units, the second digit to represent tens, the third for hundreds, the fourth for thousands, and so on. The number 444 uses three times the digit '4', but due to their position they have different values, and thus this number sums to 4 hundreds, 4 tens, and 4 units (four hundreds forty four). In Roman numbering, several symbols are used to represent this number, which becomes CDXLIV. CD is read "500 - 100", XL is read "50 - 10", IV is read "5 - 1", so the final number is again 444, but the value of the symbols does not depend on their position.

Positional numeral systems, like the decimal one, need the zero to represent an "empty" or "null" place. For example, the number 404 has 4 hundreds, 0 tens, and 4 units. Omitting the zero would lead to 44, which is a completely different number.

This numeral system can be generalized saying that a digit's value is obtained by multiplying it by $10^i$, where $i$ is the position in the number, and every position's value is summed up. In fact, 444 can be written as:

$$4 \cdot 10^2 + 4 \cdot 10^1 + 4 \cdot 10^0 \tag{2.1}$$

or, more in general, the value of a number $a_n a_{n-1} \ldots a_2 a_1 a_0$ is given by:

$$\sum_{i=0}^{n} (a_i \cdot 10^i) \tag{2.2}$$

where $n$ is the number of positions used by a certain number. Note, in Equation (2.2), the recurring number 10. This number, which corresponds to the number of available symbols to represent a number is called the *base* (or *radix)* of the numeral system. In fact, we could in principles use any value as the base, provided that we have enough different symbols to use. If $b$ is the general base of a numeral system, then a number's value $x$ can be expressed as:

$$x = \sum_{i=0}^{n} (a_i \cdot b^i) \tag{2.3}$$

The binary system is no different, as it is a positional system which uses 2 as the base, and the symbols used are the values of the bit, namely zero and one. There are, however, other numeral systems that are often used in computer science, mostly because they are convenient to represent specific quantities in a very concise way. One of them is the *hexadecimal system* (usually abbreviated as *hex*), where numbers are represented in base 16. The 16 used symbols are numbers in the range $[0, 9]$, and then the letters A, B, C, D, E, and F. This base system is very common, as a hex digit corresponds to exactly 4 bits (since $2^4 = 16$), a unit often referred to as a *nibble*, and thus one byte can be concisely

[1] This system is called *Arab* for historical reasons, although it originated from India.

**Table 2.1:** Representation of numbers $[1, 16]$ in various numeral systems.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Decimal** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **Hexadecimal** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 |
| **Octal** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 20 |
| **Binary** | 1 | 10 | 11 | 100 | 101 | 110 | 111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 | 10000 |

represented using two hex digits. Similarly, another used numeral system is the *octal* (often abbreviated as *oct*), which represents numbers in base 8, thus using only the symbols 0–7. An octal digit represents three bits (since $2^3 = 8$), which is enough in many operating systems to represent different *permissions* to access resources such as files.

Since many of the presented bases share several common symbols, it is important to explicitly specify the base according to which a number is represented, whenever there could be an ambiguity. The notation that we adopt to do this is to surround in parentheses the number, and specify the base as subscript. Therefore, we can say that $(10)_{10} = (1010)_2 = (12)_8 = (A)_{16}$.

Subtraction and addition, in the binary system, follow the same rules of the decimal system. Therefore, if we want to sum $(110)_2$ with $(10)_2$ we proceed as follows:

$$\begin{array}{rcr} 110 & + & 6 \\ 10 & = & 2 \\ \hline 1000 & & 8 \end{array}$$

Similarly, if we want to subtract $(11)_2$ from $(1110)_2$, we proceed as follows:

$$\begin{array}{rcr} 1110 & - & 14 \\ 11 & = & 3 \\ \hline 1011 & & 11 \end{array}$$

Of course, the subtraction that we have shown here works only in the natural set of numbers $\mathbb{N}$, because we have not yet introduced how negative numbers are represented in binary *encoding*. We will discuss how computer architectures deal with negative numbers in Section 2.1.3, after having discussed how we can represent the same number in different bases.

### 2.1.2 Base Conversion

It is very important, then, to be quick at converting numbers back and forth between the decimal, binary, octal, and hexadecimal systems, as this is something which comes very handy when dealing with computing systems at a low level.

By the above discussion, it is clear that the same quantities are represented, using different bases, with different symbols. Table 2.1 shows the representations of numbers $[1, 16]$ in the most common bases which are of interest in computing architectures. It is very easy to see that the most "concise" representation is the hexadecimal, while the most "verbose" is the binary, exactly because hexadecimal has more symbols to use than binary.

Any number in any numeral system can be converted into another numeral system by reasoning about the relations between the bases. An important property to be used for conversion is:

**Definition** If $x$ is a number in base $b$, then $x$ is a sequence of digits $< b$

We can then rewrite Equation (2.3) as:

$$x = a_0 + b \cdot (a_1 + b \cdot (a_2 + b \ldots)) \tag{2.4}$$

From this equation, the procedure to convert a number to another base $b'$ becomes more evident. In particular, the index $n$ of the leading digit in the new base $b'$ is given by:

$$n = \lfloor \log_{b'} x \rfloor \tag{2.5}$$

Now, the digits of the number represented in the new base $b'$ can be computed as:

$$a_i = \left\lfloor \frac{r_i}{b'^i} \right\rfloor \tag{2.6}$$

where $r_i$ are the remainder of the successive integer divisions in Equation (2.6). In particular, $r_n = x$, and $r_{i-1} = r_i - a_i \cdot b'^i$.

There is anyhow a glitch in this algorithm. In fact, the used functions (namely division, floor function, and subtraction) must be computed accounting for the original base $b$ of the initial representation, which is a non-trivial problem. Since this could be counter-intuitive, usually conversions pass through base 10, if computed by humans, or through base 2, if computed by machines.

Let's see all this in practice, and let's try to convert $(57)_{10}$ to its binary representation. Referring to Equation (2.6), we have to repeatedly divide $(57)_{10}$ by $b'$, which is 2 in our case. Having multiple divisions is the same as dividing $(57)_{10}$ by $b'^i$, and the floor /reminder calculation can be mapped to computing the integer division modulus 2:

| | /2 | |
|---|---|---|
| 57 | 28 | a = 1 |
| 28 | 14 | a = 0 |
| 14 | 7 | a = 0 |
| 7 | 3 | a = 1 |
| 3 | 1 | a = 1 |
| 1 | 0 | a = 1 |
| 0 | | |

order of digits ↑

Equation (2.4) tells that the first division gives us the coefficient $a_0$, thus the representation in base 2 takes the last remainder as the most-significant digit. Thus we get that $(57)_{10} = (111001)_2$. We have stopped dividing by $b'$ when we have reached zero as the dividend. It is interesting to note that applying Equation (2.5) to $(57)_{10}$ we obtain $\lfloor \log_2(57) \rfloor = 5$, and in fact we have stopped after 6 iterations of our conversions (namely, after having computed coefficients $a_0$ to $a_5$).

To convert back from base 2 to base 10, we can simply apply Equation (2.3) setting $b = 2$, and we obtain:

$$57 = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

From this example we can derive an additional property of numbers represented in binary. If we consider that $a_0$ is always multiplied by $2^0$, we can assert that all *odd* numbers end with digit 1 in binary, while all *even* numbers end with 0.

Equations (2.6) and (2.3) can be used as well to compute the conversion of the fractional part of a number. In fact, when converting from base 10 to 2 a fractional number, we divide the current remainder iteratively by $2^{-1}$, $2^{-2}$, and so on, which is equivalent in practice to *multiplying* the fractional part by 2. Note that, in this case, the modulus operation does not hold anymore, and we therefore have to apply exactly Equation (2.6) for the conversion. To see this in action, let's look at another example, and let's try to convert to base 2 the fractional number $(0.6875)_{10}$:

| | *2 | | |
|---|---|---|---|
| 0.6875 | 1.375 | a = 1 | order of digits ↓ |
| 0.375 | 0.75 | a = 0 | |
| 0.75 | 1.5 | a = 1 | |
| 0.5 | 1.0 | a = 1 | |
| 0 | | | |

Since the conversion rule tells to divide by $b'^i$, but we have rather multiplied by $\frac{1}{b'^i}$ to simplify the math, the order according to which we determine what is the most significant digit of the fractional part is the opposite of the integral part. From this, we obtain that $(0.6875)_{10} = (0.1011)_2$.

Note that in case of fractional numbers Equation (2.5) does not tell anything, as it is not related to the fractional part of the number. In particular, this poses an additional question: does the conversion of fractional parts always terminate?

This opens up to a property of numeral representations which is interesting: a number which is periodic in a numeral system might not be periodic in another, and vice versa. To show this, let's try to convert $(0.1)_{10}$, which is an aperiodic number:

| | *2 | | |
|---|---|---|---|
| 0.1 | 0.2 | a = 0 | order of digits ↓ |
| 0.2 | 0.4 | a = 0 | |
| 0.4 | 0.8 | a = 0 | |
| 0.8 | 1.6 | a = 1 | |
| 0.6 | 1.2 | a = 1 | |
| 0.2 | 0.4 | a = 0 | |

In this conversion, we have stopped after 6 iterations. At that point, the result of the last multiplication was not zero, but it was a value that had been already encountered during the conversion. This means that $(0.1)_{10} = (0.0\overline{0011})_2$, which is a periodic number.

Similarly to the integral part, to convert the fractional part in base 2 back to base 10 Equation (2.3) can be used. For example, to convert $(0.1011)_2$ to the base 10 representation, we can compute:

$$(0.6875)_{10} = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}$$

As mentioned, converting from binary to octal and hex bases are operations that are usually done quite often, especially for integer numbers. Therefore, although the conversion could be made by passing through base 10, it is interesting to note that there are several "shortcuts" to avoid this longer procedure.

To convert an octal number $(a_n \dots a_2 a_1 a_0)_8$ composed of $n$ digits, each single digit $a_i$ can be converted into its binary form (using three bits) and later be concatenated. For example, let's look at the number $(325)_8$. We have that:

- $(3)_8 = (011)_2$
- $(2)_8 = (010)_2$
- $(5)_8 = (101)_2$

thus we have that $(325)_8 = (11010101)_2$. The same is true to convert a number represented in the hexadecimal base to binary base, considering that every digit must be converted to 4 bits. For example, considering the number $(A13)_{16}$ we have:

- $(A)_{16} = (1010)_2$
- $(1)_{16} = (0001)_2$
- $(3)_{16} = (0011)_2$

thus we have that $(A13)_{16} = (101000010011)_2$. To make the conversions in the opposite directions, namely from base 2 to 8 or 16, the steps to take are a bit different. In particular, the number of digits of the binary representation must be made a multiple of the destination base by adding trailing zeroes, and then the digits should be grouped into groups of 3 or 4 bits, depending on whether we are converting to base 8 or 16, respectively. For example, the following conversions hold:

$$(1101001101)_2 = (001|101|001|101)_2 = (1515)_8$$
$$(1101001101)_2 = (0011|0100|1101)_2 = (34D)_{16}$$

This handy property holds because 2, 8, and 16 are all powers of 2. This relation, in fact, holds for any conversion when the bases of multiple numeral systems are powers of the same number. In fact, in this case, the conversion can be operated on groups of digits due to the following properties:

- A one-digit number in base $a^b$ corresponds to a number of $b$ digits in base $a$;
- A number of $c$ digits in base $a^b$ corresponds to a number of $bc$ digits in base $a$, and to a number of $b$ digits in base $a^c$;
- A number of $cd$ digits in base $ab$ corresponds to a number of $bcd$ digits in base $a$, and to a number of $bd$ digits in base $ac$.

For example, if we take bases 3 and 9, we have that:

$$(37)_9 = 3 \cdot 9^1 + 7 \cdot 9^0 = (1 \cdot 3^1 + 0 \cdot 3^0) \cdot 3^2 + (2 \cdot 3^1 + 1 \cdot 3^0) \cdot 3^0 = 1 \cdot 3^3 + 0 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 = (1021)_3$$

More in general, this technique could be applied to other base conversions, provided that the source base is in the form $a^b$ and the destination base is in the form $a^c$, by composing a transformation from $a^b$ to $a$ and then from $a$ to $a^c$. For example:

$$(5627)_8 = (5\,6\,2\,7)_8 = (101\,110\,010\,111)_2 = (101110010111)_2 = (10\,11\,10\,01\,01\,11)_2 = (2\,3\,2\,1\,1\,3)_4 = (232113)_4$$

The same result could be obtained by grouping the digits: each group of $c$ digits in base $a^b$ correspond a group of $bc$ digits in base $a$ and a group of $b$ digits in base $a^c$. Therefore:

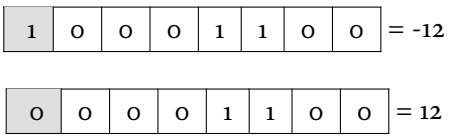
- $(56)_8 = (5\,6)_8 = (101\,110)_2 = (10\,11\,10)_2 = (2\,3\,2)_4 = (232)_4$
- $(27)_8 = (2\,7)_8 = (010\,111)_2 = (01\,01\,11)_2 = (1\,1\,3)_4 = (113)_4$

and therefore:

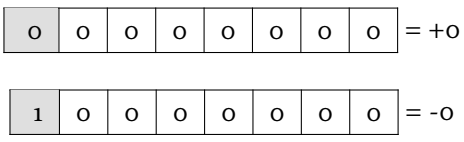
$$(5627)_8 = (56\,27)_8 = (232\,113)_4 = (232113)_4$$

### 2.1.3 Negatives

So far, we have only dealt with positive numbers. When it comes to representing negative numbers, several *encodings* could be used[2]. The most trivial one resembles the fact that we add the minus sign '-' before a negative number. Since we cannot represent the minus sign in binary, we dedicate a bit to this. In particular, given a word of length $n$, 1 bit is used to encode the sign (this bit is referred to as the *sign bit*), and $n-1$ bits are used to represent the absolute value (or the *magnitude*) of the number in base 2. The sign bit is set to zero for positive numbers, and is set to one for negative numbers. This encoding is called *sign-magnitude*, and was used by some early computers, such as the IBM 7090. For example, using 8 bits, we could have the following encoding:




It is interesting to note that, by using this strategy, there are two different encodings for the zero, one positive and one negative:




This simple representation is anyhow largely inefficient, despite its natural relation to common usage of numbers. In fact, the circuits which are able to perform calculations using these representations require a large number of components, which is a cost-ineffective solution.

Another encoding of negative numbers is called *ones' complement*, and is obtained by inverting all the bits in the binary representation. For example, if we encode numbers using 4 bits, the positive

[2] We call the representation of negative numbers *encoding* to differentiate it from the mathematical representation of positive numbers in base 2.

**Table 2.2:** Positive and negative numbers that can be represented in a ones' complement numeral system using 4 bits.

| **Number** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| + | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| - | 1111 | 1110 | 1101 | 1100 | 1011 | 1010 | 1001 | 1000 |

number $(4)_{10}$ is directly transformed into $(0100)_2$, while $(-4)_{10}$ becomes $(1011)_2$. If we have a number $x$, its ones' complement is denoted $\bar{x}$. Again, similarly to the sign-magnitude representation, we have two different values for the *positive zero* and the *negative zero*, which are respectively (again in a 4-bits system) $(0000)_2$ and $(1111)_2$.

Therefore, a ones' complement numeral system is a system in which negative numbers are represented as the *arithmetic negative* of the value of the positive number. In such a system, a number is negated (converted from positive to negative or vice versa) by computing its ones' complement. When a number is represented using a word of $n$ bits, the ones' complement numeral system can represent integers in the range $[-2^{n-1} - 1, 2^{n-1} - 1]$.

In a 4-bits system, the possible representable numbers are shown in Table 2.2. It is interesting to note that the most-significant bit has a role similar to the one it had in the sign-magnitude representation: numbers starting with one are negative, numbers starting with 0 are positive. Nevertheless, the remaining $n - 1$ bits do not represent the binary encoding of the absolute value of the number. Again similarly to the sign-magnitude encoding, the ones' complement has two zeroes, the positive and the negative one.

The arithmetic behind this representation is a bit different from the traditional one. If we want to sum two different numbers, we still align them and sum one digit after the other:

| | | |
|---|---|---|
| 0001 0110 | + | 22 |
| 0000 0011 | = | 3 |
| 0001 1001 | | 25 |

but we have to account for the possibility that a carry comes out of the most significant position, especially when dealing with negative numbers:

| | | |
|---|---|---|
| 1111 1110 | + | -1 |
| 0000 0010 | = | 2 |
| 1 0000 0000 | | 0 |

This latter result is clearly wrong. This is because an *end-around carry* condition was met: the carry coming out of the addition must be "wrapped" around, and summed to what is now the partial result of the operation. Computing 0 + 1 = 1 gives the correct result.

This same situation happens with subtractions, where *end-around borrows* might arise:

| | | |
|---|---|---|
| 0000 0110 | - | 6 |
| 0001 0011 | = | 19 |
| 1 1111 0011 | | -12 |

Again, this is the wrong result, and to make it correct the borrow should be "wrapped" around and subtracted from the partial result, giving $-12 - 1 = -13$.

An interesting property of the ones' complement encoding is that, although less intuitively, the bit complement of a positive value is a number with the same magnitude of the negative value—which is a necessary condition to have all the arithmetic operations behave in the expected way. In fact, for example, $19 + 3 = 19 - (-3)$:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | 0001 0011 | - | 19 | |
| 0001 0011 | + | 19 | | 1111 1100 | = | -3 | |
| 0000 0011 | = | 3 | | 1 0001 0111 | - | 23 | |
| 0001 0110 | | 22 | | 0000 0001 | = | 1 | |
| | | | | 0001 0110 | | 22 | |

where in the second operation an end-around borrow condition was met again. The circuitry required to compute operations using ones' complement were much simpler, and many more architectures had used this encoding, such as the PDP-1, the CDC 160 series, the CDC 6000 series, or the UNIVAC 1100. Nevertheless, the presence of multiple zeroes was wasting some of the possible encodings of numbers, and the end-around borrow and carry conditions were causing the hardware to perform anyhow additional checks and operations.

Therefore, another encoding for integer numbers was invented, which gets the name of *two's complement*. This is the encoding which offers the simplest and most efficient hardware implementation, and is therefore widely adopted, for example by Intel x86, Motorola m68k, MIPS, SPARC, ARM, DEC Alpha, and the z64 processors use two's complement to represent and process negative numbers.

The two's complement of a number $x$ represented using a word of $n$ bits is defined as the complement with respect to $2^n$, namely the result of $2^n - x$. Computing a number's two's complement can be automatized in a more efficient way reasoning about the ones' complement of a number. If we take the number $x = (0110)_2$ and it's ones' complement $\bar{x} = (1001)_2$, we can see that the addition of these two values is $(1111)_2$. This is true in all cases, given the definition of a ones' complement. Thus, any sum in the form $x + \bar{x}$ is a sequence of all ones. But if we take a word of length $n$ which is composed by all ones, its value in decimal is $2^n - 1$. Thus we have that:

$$\begin{aligned} x + \bar{x} &= 2^n - 1 \\ x + \bar{x} + 1 &= 2^n \\ \bar{x} + 1 &= 2^n - x \end{aligned}$$

Since $2^n - x$ is the formula to compute $x$'s two's complement, we can see that the same value could be computed by calculating the ones' complement $\bar{x}$ of $x$ and then adding one. Computing $\bar{x}$ is much simpler from a circuit-design point of view, as it only entails negating each bit.

Nevertheless, an additional property to convert a number to its two's complement can be devised. In fact, if we try to apply the last rule to compute the two's complement of $(0110)_2$, we can see that:

- The value $\bar{x} = (1001)_2$

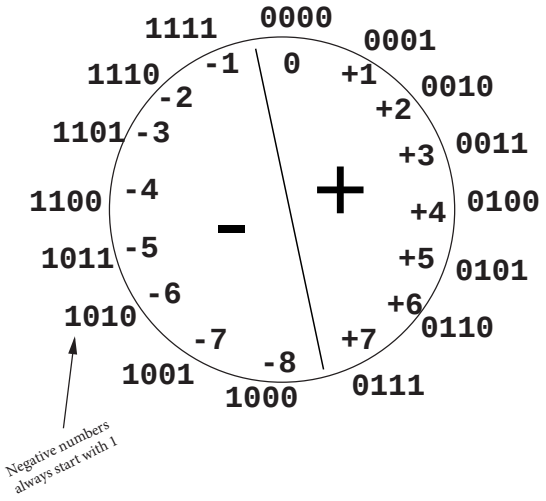



**Figure 2.2:** Mapping of positive and negative values to binary strings in a 4-bits two's complement numeral system.

- Adding one leads to $(1010)_2$.

If we look carefully at the result, we can see that the two least significant bits of the two's complement are 10, exactly as in the initial value $(0110)_2$. This is not by chance: in fact, when computing $\bar{x}$ all the trailing zeroes are negated to ones. Adding one causes a cascading carry, until the first zero is reached. But this first zero was a one in the initial representation (again, due to the inversion of the bits). Therefore, adding one to $\bar{x}$ always restores the initial value of the least-significant bits, up to the first one. Considering this, an additional practical rule can be devised to convert a number to its two's complement:

**Practical Rule:** To compute the two's complement of a number, start from the least-significant bit. Leave untouched all bits until the first one is found. Then, invert all the remaining bits.

In a circuitry, this method is no faster than computing $\bar{x} + 1$, as both methods require working sequentially from the least-significant bit towards the most-significant one, propagating logic changes.

A two's-complement numeral system encodes positive and negative numbers in a binary encoding. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two. The mapping of positive and negative values in a 4-bits two's-complement numeral system is depicted in Figure 2.2. This picture is the geometric visualization of the fact that $n$-bit binary numbers represented using the two's complement are a *cyclic group* $\mathbb{Z}/2^n$, where taking a negative number is mapped to a *reflection* on the circle. From the Figure it can be seen as well that the range of representable numbers using $n$ bits is $[-2^{n-1}, 2^{n-1} - 1]$. Differently from ones' complement, two's complement representation can represent one additional negative number, mainly because in this representation there is only one zero.

The value $x$ of a two's complement number $\bar{\bar{x}}$ represented as a word of length $n$ in the form $a_{n-1}a_{n-2}\ldots a_0$ is given by Equation (2.7):

$$x = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \tag{2.7}$$

The main difference with Equation (2.3) is exactly the weight of the most significant digit $a_{n-1}$. Compared to the sign-magnitude representation, the first bit of a two's complement value maintains both the information associated with the sign, and a part of the weight of the number representation.

When moving a certain two's complement binary encoding from a word of size $n$ to a word of size $m > n$, the sign bit must be preserved. For example, when the encoding using four bits of $(-5)_{10} = (1011)_2$ is converted to an 8-bits encoding, the leading bits should be set to 1, to preserve the information that the number is negative, thus leading to 11111011. If we compute again the two's complement of this encoding, we obtain 00000101, which is $(5)_{10}$, thus the information has been kept correctly, and the common semantics has been preserved.

As we have seen, converting a number to its two's complement gives an encoding that represents the negative number. If the two's complement of the negative number is computed again, the positive number is found back again. There is one exception to this rule, which is the most negative number. In a 4-bits system, the most negative number is $(-8)_{10} = (1000)_2$. Taking again the two's complement gives again $(1000)_2$. This is due to two reasons. On the one hand, the most intuitive explanation is that $(8)_{10}$ cannot be represented in this numeral system. On the other hand, the mathematical explanation is complementary to the fact that the negative of zero is again zero (namely, there is no positive and negative zero in this representation). Since the number zero is invariant under taking negatives, and two's complement is cyclic group of order 2, at least some other number must be invariant under taking negatives. This number is the one placed at the opposite position of the zero on the circle depicted in Figure 2.2. This result is known as the *orbit-stabilizer theorem*, and the most negative number is sometimes called the *weird number* due to this exception.

Some examples of two's complement conversions are the following:

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0 |
|---|---|---|---|---|---|---|---|---|

| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $= 2^{n-1} - 1$ |
|---|---|---|---|---|---|---|---|---|

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = -1 |
|---|---|---|---|---|---|---|---|---|

When it comes to arithmetic operations, addition can be carried out as usual. For example, adding 15 and -5 represented using 8 bits gives:

| 0000 1111 | + | 15 |
|---|---|---|
| 1111 1011 | = | -5 |
| 0000 1010 | | 10 |

It is interesting to note that during the computation of the addition, there is a carry which comes out of the most significant 8-th bit. Nevertheless, since this carry goes into the non-existent 9-th position, the obtained result is correct.

Concerning subtraction, the advantage of using two's complement is that the operation does not have to take into account the sign of the operands. If we want to subtract $(-5)_{10}$ from $(15)_{10}$ we have:

| | | |
|---|---|---|
| 0000 1111 | - | 15 |
| 1111 1011 | = | -5 |
| 0001 0100 | | 20 |

To summarize, the operations that a CPU should carry out to compute additions depends on the numerical encoding of negatives:

- If two's complement representation is used, subtraction requires only inverting the bits of the subtrahend and setting a carry into the rightmost bit.
- Using ones' complement representation requires inverting the bits of the subtrahend and connecting the carry out of the most significant bit to the carry in of the least significant bit (end-around carry).
- Using sign-magnitude representation requires only complementing the sign bit of the subtrahend and adding, but the addition/subtraction logic needs to compare the sign bits, complement one of the inputs if they are different, implement an end-around carry, and complement the result if there was no carry from the most significant bit.

Using two's complement requires the smallest number of operations, and this is why it is so widely used in modern CPUs. Nevertheless, comparing a negative value and a positive value represented according to the two's complement scheme, could lead to the opposite result, due to the fact that a negative value has always the most-significant bit set.

There are additional encodings for negative numbers, which are less used to represent numbers or to carry out operations in CPUs, but that come handy in some other scenarios, like representing real numbers. One of these representations is called *excess-k*, *offset binary*, or *biased* representation. This way of representing negative numbers uses a pre-specified number $k$ as a *biasing value*. Then, the number zero is represented using the binary value of $k$, a binary number composed of all zeroes is the minimal negative value, and a binary number composed of all ones is the maximal positive value. There is no standard way to determine $k$, although usually when representing numbers using $n$ bits, $k$ is usually set to $k = 2^{n-1}$. In this way, a number zero is represented with a binary number with the most-significant bit set to 1, and all the other digits set to zero. Additionally, the order for comparison between numbers is always preserved, which is something that is not the case using two's complement.

The last numeric encoding which allows to represent negative numbers is called *base -2*, or the *negabinary* system. In this system the base is set to -2. Negative numerical bases were first considered by Vittorio Grünwald in his work *Giornale di Matematiche di Battaglini*, published in 1885. Grünwald gave algorithms for performing addition, subtraction, multiplication, division, root extraction, divisibility tests, and radix conversion. In negabinary, any integer number $x$ can be represented as:

$$x = \sum_{i=0}^{n} a_i (-2)^i \tag{2.8}$$

Some numbers in negabinary have the same representation in binary, such as $17 = 2^4 + 2^0 = (-2)^4 + (-2)^0$ and are thus represented as 10001 in both numeral systems. Converting a number to

**Table 2.3:** Summary of binary representations using 4 bits, and their numeric value using different encodings.

| Binary | Unsigned | Sign-magnitude | Ones' complement | Two's complement | Excess-8 | Base -2 |
|---|---|---|---|---|---|---|
| 0000 | 0 | 0 | 0 | 0 | -8 | 0 |
| 0001 | 1 | 1 | 1 | 1 | -7 | 1 |
| 0010 | 2 | 2 | 2 | 2 | -6 | -2 |
| 0011 | 3 | 3 | 3 | 3 | -5 | -1 |
| 0100 | 4 | 4 | 4 | 4 | -4 | 4 |
| 0101 | 5 | 5 | 5 | 5 | -3 | 5 |
| 0110 | 6 | 6 | 6 | 6 | -2 | 2 |
| 0111 | 7 | 7 | 7 | 7 | -1 | 3 |
| 1000 | 8 | -0 | -7 | -8 | 0 | -8 |
| 1001 | 9 | -1 | -6 | -7 | 1 | -7 |
| 1010 | 10 | -2 | -5 | -6 | 2 | -10 |
| 1011 | 11 | -3 | -4 | -5 | 3 | -9 |
| 1100 | 12 | -4 | -3 | -4 | 4 | -4 |
| 1101 | 13 | -5 | -2 | -3 | 5 | -3 |
| 1110 | 14 | -6 | -1 | -2 | 6 | -6 |
| 1111 | 15 | -7 | -0 | -1 | 7 | -5 |

the negabinary representation can be performed again by using Equation (2.6). While it could appear a bit cumbersome to represent numbers using negabinary, several optimized circuits have been devised in history to implement arithmetic operations, and it has been used as well in the design of special-purpose hardware, such as digital filters or the fast Fourier transform.

From this discussion we have seen once again that simply looking at a sequence of bits stored in working memory cannot tell what that information is representing. In fact, as it can be seen in Table 2.3, the same sequence of bits can represent very different values. It is then the role of the programmer to properly instruct the CPU, when writing his code, on how to deal with the given information.

### 2.1.4 Real Numbers

There are many real-life quantities that cannot be stored accurately in integers. Lengths are something which are hardly integer, for example, or even prices of goods, temperatures, frequencies of musical notes, or speeds. Representing such quantities with an integer representation would lose their fractional part. Or even more, as we have said, there is a maximum number that can be represented given a word of $n$ bits. If we need to represent a bigger quantity, we simply cannot.

There is a completely different encoding for numbers, which was mainly devised to represent rational number, but works quite well to overcome all the integers limitations as well. This is the *floating point* representation, which is a formulaic representation to approximate a real number having a good trade-off between range and precision. A number represented as floating point is in the following form:

$$significand \cdot base^{exponent} \tag{2.9}$$

where $significand \in \mathbb{Z}$, $base \in \mathbb{N}$, and $exponent \in \mathbb{Z}$. The significand is also called *mantissa*. For example, the number 1.2345 is divided into the three components as following:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

The name floating point refers to the fact that with this representation the decimal point of a real number can "float around", as it can be placed anywhere in the representation, as the exponent is changed accordingly. There are several variants to represent a floating point in binary, such as the size of the mantissa and of the exponent. Here we will discuss the IEEE 754 32-bits standard[3] dating back to 1985, which is what modern architectures use to represent floating point numbers and compute operations on them.

As it will be clear by the enconding of a floating point number in binary, operations tend to be quite involved. Even simple additions require several steps. Historically, there were chips separated from the main CPU which were used just to execute floating point operations. These chips were called *math coprocessors*, and one of Intel's most famous ones is the 8087, which was the chip to which the 8086 was offloading mathematical computations. Intel's ISA which describes how floating point units (a different name for a math coprocessor) are organized is often called x87, coming from the name of the 8087. Since the 80486, x87 units have been bundled within the main CPU, thus no longer requiring a separate chip to do maths.

IEEE 754 specifies different types of numbers:

- *Zeroes*: there is both a +0 and a -0 in floating point representation. Most operations are devised so that they behave in the same way, independently of the sign of the zero, but for example dividing a number by +0 or by -0 gives a different infinity result;
- *Infinities*: they are the result of a division by zero, or of an *overflow* (namely, the computation of a number of so large magnitude that they cannot be represented). As mentioned, two different infinities are described by the standard, namely $+\infty$ and $-\infty$;
- *NaNs*: they are special values, which are *Not a Number*. They represent the result of every operation that does not have a meaningful result in terms of a finite number or infinity, such as $\infty - \infty$, or $0/0$, or $\sqrt{-1}$;
- *Normal numbers* (or *normalized numbers*): these are the vast majority of non-zero numbers that can be represented by the IEEE 754 encoding standard;
- *Subnormal numbers* (or *denormalized numbers*, or *denormals*): there are values which are extremely close to zero. As we will see, they pose a special issue regarding rounding errors.

Since operations on real numbers can be quite complex, and not all the domains of every function which can be computed on real numbers are the same, the standard specifies a certain number of *exceptions* which inform the programmer that something went wrong during a computation. The programmer can anyhow "silence" these exceptions, if he thinks that they are not meaningful for the software he is writing. The exceptions described by the IEEE 754 standard are:

[3] The 64-bits standard is no different, except for the size of the mantissa and of the exponent. For the sake of simplicity, mainly to deal with shorter bit strings, we have opted for the 32-bits version.

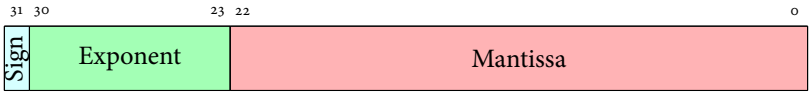



**Figure 2.3:** Floating Point encoding using the 32-bits version of the IEEE 754 standard.

- *Invalid operation*: this exception is fired when an operation which is not mathematically correct is computed, or if the exceptions are disabled, the result of the operation is a NaN;
- *Overflow*: this exception tells that the result of an operation is too large in magnitude to be represented by a floating point number. If the generation of exceptions is disabled, the result is $+\infty$ or $-\infty$;
- *Division by zero*: This is raised when computing $x/\pm 0$ when $x \neq 0$. The silent result is $\pm\infty$ depending on the sign of the zero;
- *Underflow*: Similarly to the overflow, this exception is fired when the magnitude of the result of an operation is too small to be computed correctly. This is a generally harmless condition for most of the software;
- *Inexact*: This exception tells that the "real" result could not be represented using IEEE 754 standard, and therefore the actual value returned by an operation is the result of a rounding.

In the case of a word of 32 bits, the structure of the encoding of a floating point number according to the IEEE 754 standard is reported in Figure 2.3. A number is thus organized into three different fields:

1. sign $s$, 1-bit wide
2. exponent $e$, 8-bits wide
3. mantissa $m$, 23-bits wide

The sign of the encoding resembles the sign of the sign-magnitude format, and thus is 1 for negatives and 0 for positives. The exponent is encoded using excess-127 format, although some values of the representation are reserved for special values like infinities and NaNs. In particular, the range of exponents which can be represented is $[-126, 127]$. A summary of the values of the exponent and the way the number is interpreted is reported in Table 2.4. Again, given that the exponent is encoded in excess-127, a given exponent $E$ is mapped to the exponent $e$ which is stored in the binary representation as:

$$e = E + 127 \tag{2.10}$$

From this discussion, it is evident that the decimal value of a number represented according to the IEEE 754 floating point encoding is:

$$(-1)^s \cdot 2^E \cdot 1.m \tag{2.11}$$

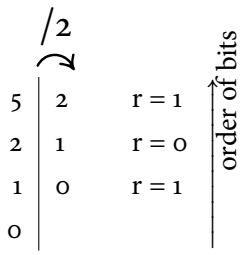
The value of the mantissa depends on whether the number is normalized or denormalized. In the vast majority of cases, numbers are normalized, and thus the value of the mantissa is between $(1.1)_2$

**Table 2.4:** Different types of value which can be represented according to the IEEE 754 format.

| $e$ | $m$ | **Type of value** |
|---|---|---|
| $[1, 254]$ | any | $(-1)^s \cdot 2^e \cdot 1.m$ – normalized numbers |
| 0 | $\neq 0$ | $(-1)^s \cdot 2^{-126} \cdot 0.m$ – denormalized numbers |
| 0 | 0 | $(-1)^s$ – signed zero |
| 255 | 0 | $(-1)^s \cdot \infty$ – signed infinity |
| 255 | $\neq 0$ | `NaN` – Not a Number |

and $(1.11111111111111111111111)_2$, namely between $(1.5)_{10}$ and $(1.99999998808)_{10}$. Any significand which does not fall in this range should be transformed (and the exponent thus correspondingly adjusted) to make it fall in the given interval. In case the number is denormalized, according to what reported in Table 2.4, the exponent $e = 0$ (therefore -127 is not a valid exponent), an thus the mantissa is normalized between $(0.1)_2$ and $(0.11111111111111111111111)_2$, namely between $(0.5)_{10}$ and $(0.9999998808)_{10}$. Despite the fact that $e = 0$, the exponent in this case is interpreted as -126, thus the range of subnormal numbers when using 32 bits is $[-1.40130 \cdot 10^{-45}, +1.40130 \cdot 10^{-45}]$. It is interesting to note that when using normalized numbers, on the other hand, the closest number to zero which can be represented is $\pm 1.0 \cdot 2^{-126} \approx \pm 1.17549 \cdot 10^{-38}$. If we compare $1.40130 \cdot 10^{-45}$ and $1.17549 \cdot 10^{-38}$, we can see that there is no overlapping between the two different representations, which is a desirable property because no binary encodings are wasted in this way—and this is exactly why denormalized numbers pick -126 as the exponent, rather than -127. Nevertheless, the difference between the two numbers is not zero, therefore there is a gap between the two representations. These are real numbers that *cannot* be represented in the format, so any number in between $1.40130 \cdot 10^{-45}$ and $1.17549 \cdot 10^{-38}$ is rounded to either of the two.

Let us see how we can convert a decimal real number into a binary floating point number, namely $x = (-5, 828125)_{10}$. To convert $x$ in binary, we first determine the sign. This is a negative number, so $s = 1$. Then, we convert to binary the integral part, as we have done already before for any integer number:




so that we (obviously) obtain that $(5)_{10} = (101)_2$. We then convert the fractional part of the number, similarly as we did before:

| | | | |
|---|---|---|---|
| 0.828125 | 1.65625 | r = 1 | order of bits ↓ |
| 0.65625 | 0.3125 | r = 1 | |
| 0.3125 | 0.625 | r = 0 | |
| 0.625 | 1.25 | r = 1 | |
| 0.25 | 0.5 | r = 0 | |
| 0.5 | 1 | r = 1 | |
| 0 | | | |

from which we get that $(0.828125)_{10} = (0.110101)_2$. Combining the two results we then know that $x = (101.110101)_2$. As mentioned, the IEEE 754 standard requires (in general) to have the mantissa normalized, thus we must transform our number to the form $1.m$, which gives $x = (1.01110101 \cdot 2^2)_2$. Then we can omit the integral part of the representation (because we are representing a normalized number) and thus set the mantissa to $m = 01110101$. The exponent is $E = 2$, so we have to convert it to the excess-127 format, using Equation (2.10), having:

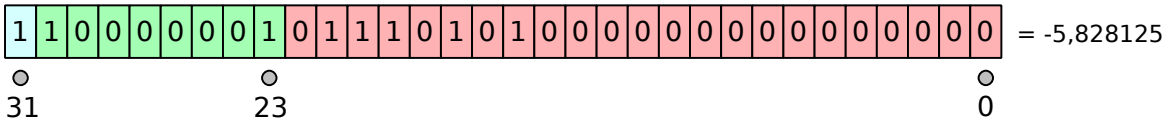
$$e = E + 127 = 2 + 127 = 129$$

This value should be now converted to binary, as usual:

/2

| | | | |
|---|---|---|---|
| 129 | 64 | r = 1 | order of bits ↑ |
| 64 | 32 | r = 0 | |
| 32 | 16 | r = 0 | |
| 16 | 8 | r = 0 | |
| 8 | 4 | r = 0 | |
| 4 | 2 | r = 0 | |
| 2 | 1 | r = 0 | |
| 1 | 0 | r = 1 | |
| 0 | | | |

thus the exponent will be $e = (10000001)_2$. At this point, we have determined the value of $s$, $e$, and $m$, so the final encoding is given by the concatenation of the three fields:

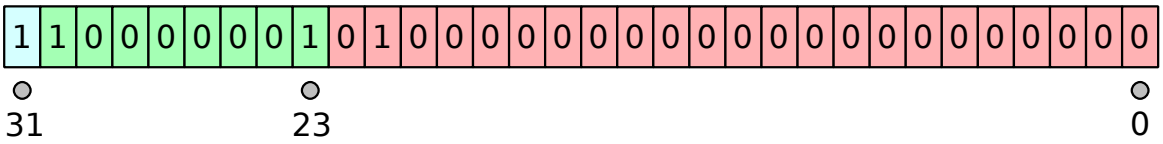



Sometimes it could be useful a more concise representation of the 32-bits number. To this end, we can rely on the hexadecimal base, an thus we can group the bits in quadruplets and make the conversion:

| 1100 | 0000 | 1011 | 1010 | 1000 | 0000 | 0000 | 0000 |
|---|---|---|---|---|---|---|---|
| C | 0 | B | A | 8 | 0 | 0 | 0 |

The number $(-5,828125)_{10}$ can be therefore represented as well as $(C0BA8000)_{16}$.

When converting a number in the IEEE 754 to the decimal representation, we can rely on Equation (2.11), but in order to do so, we first have to extrapolate the values of $s$, $e$, and $m$. Let's look at the following example:




Here we have that:

- $s = 1$
- $e = 2^7 + 2^0 = 129 \Rightarrow E = 129 - 127 = 2$
- $m = 2^{-2} = 0.25$

thus the final decimal value of the floating point number is:

$$x = (-1)^s \cdot 2^E \cdot 1.m = (-1)^1 \cdot 2^2 \cdot 1.25 = -5.0$$

so it appears clear that, despite the fact that this notation is used to represent real numbers, it is perfectly suitable for integer numbers as well.

#### Rounding, Absolute and Relative Representation Error, Density of Reals

As we have already mentioned, every time we try to represent a number $x$ as a floating point, we are actually representing a number $x'$ which could be possibly different from $x$ due to a rounding error and due to the fact that the number of bits in the mantissa is fixed. This rounding is uniquely identified by a *rounding function* $r(x)$, which is determined by a *running mode*. IEEE 754 mandates four standard rounding modes, which could be selected by the programmer of the floating point unit:

- *Directed rounding to* $+\infty$: $r(x)$ is the least floating-point value greater than or equal to $x$;
- *Directed rounding to* $-\infty$: $r(x)$ is the greatest floating-point value smaller than or equal to $x$;
- *Round to zero*: $r(x)$ is the floating-point value of the same sign as $x$ such that $|r(x)|$ is the greatest floating point value smaller than or equal to $|x|$;
- *Round to nearest*: $r(x)$ is the floating-point value closest to $x$ with the usual distance; if two floating-point values are equally close to $x$, then $r(x)$ is the one whose least significant bit is equal to zero.

When mixing together 32-bits floating point values and 64-bits floating point values, the round-to-nearest policy might produce different results when rounding first a 64-bits value, then converting it to 32-bits and rounding again, or when the value is first converted to 32 bits and then rounded. This issue is often referred to as *double rounding* problem. Nevertheless, the default policy is round to nearest, which is then the one used by most programs.

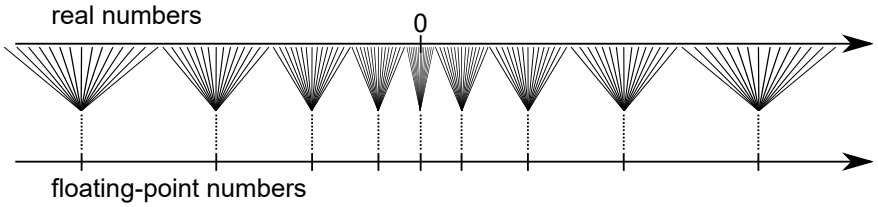



**Figure 2.4:** Schematic illustration of the correspondence between the set of real numbers and their floating-point representation in a computer.

It is therefore interesting to determine *how much* our representation $x'$ of $x$ is "wrong". We can therefore introduce the *absolute error*, defined as:

$$\varepsilon_A = x - x' \tag{2.12}$$

This is an algebraic quantity, which tells how much of the original information has been lost. At the same time we can introduce the *relative error*, which is an a-dimensional quantity which tells whether the error is small or big. This error is given by:

$$\varepsilon_R = \frac{\varepsilon_A}{x} = \frac{x - x'}{x} \tag{2.13}$$

Let's see an example using floating point numbers in base 10. Let's consider the number $x = 3.5648722189$, and let's assume that we want to represent it using only 4 fractional digits. We have that $x \approx \bar{x} = 3.5648$. With this approximation we have a relative error given by:

$$\varepsilon_R = \frac{x - \bar{x}}{x} = \frac{3.5648722189 - 3.5648}{3.5648722189} = 0.000020258$$

which tells, roughly speaking, that the error is not so large. There is another interesting relation which tells that $-log_{10}(\varepsilon_R) \simeq$ the number of fractional digits which are not affected by any error. In fact, in our example, $-log_{10}(0.000020258) = 4.69$

To conclude our discussion about floating-point numbers, let's reason once again about the fact that when we represent a number $x$ we actually represent a number $x'$. If we apply the function $r(x)$ according to the directed rounding to $-\infty$ or the directed rounding to $+\infty$ policies we obtain, from the same value $x$, two different values $x'_-$ and $x'_+$. This means that *every* real number in the interval $[x'_-, x'_+]$ is mapped to either $x'_-$ or to $x'_+$, and by definition we have an *infinite set* of real numbers falling into that interval. This situation is depicted in Figure 2.4, and tells us that while the set of real numbers is fully connected, the set of floating-point numbers is sparse.

Additionally, since the number of bits which are used to represent a floating-point value are fixed, it means that the cardinality of the set floating-point numbers is finite as well. In particular, as it can be seen in Figure 2.5, since the smallest positive value which can be represented (using the denormalized notation) is $0.5 \cdot 2^{-126}$, and the largest (normalized) value is $(1 - 2^{-24}) \cdot 2^{128}$, there are several "holes" in the possible representable numbers. In particular, close to zero, there are numbers which cannot be represented and, depending on the actual value of a non-zero number $x$ and the rounding policy, $r(x)$ could be mapped to zero. This condition is called underflow (signed or unsigned, depending on whether the initial value $x$ was positive or negative), and could be significantly crucial for software,

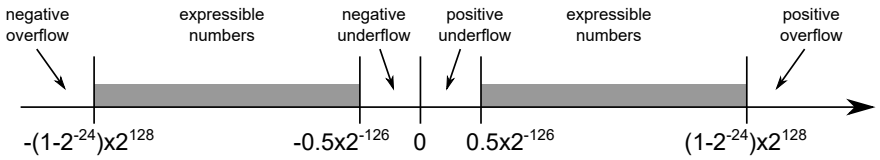



**Figure 2.5:** Bounds on the set of floating-point numbers and the conditions of overflow and underflow.


**Figure 2.6:** Density of floating point values, which decreases when going farther from zero.

specifically for mission-critical one. Think, for example, of a software which is written thinking that a certain variable is never zero, due to the nature of the algorithm, but this value becomes zero due to a rounding error. If this value is used as the divisor, the software could crash due to a divide by zero exception. Similarly to this scenario, if a value $x$ is greater/smaller than $\pm(1 - 2^{-24}) \cdot 2^{128}$, we face the overflow condition (positive or negative, depending again on the original sign of $x$).

Another interesting property of floating point numbers is their *density*. Thus, floating-point systems have the convenient property that the density of representable numbers in a range is related to the magnitude of the numbers in that range (namely, representable numbers are not equally spaced): the smaller the magnitude of the numbers in a range, the higher the density of floating-point numbers in that range, and the density halves at regular intervals. As an example, using 32-bits IEEE 754 representation, roughly half of all representable floating-point numbers are between -1 and 1, and the number of floating-point values between 65,536 and 131,072 (both powers of 2) is the same.

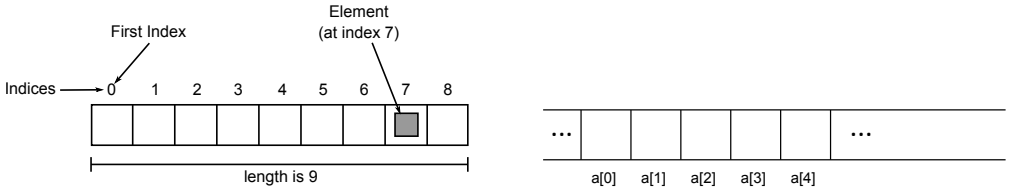
This is related to the fact that when a number is close to zero, the exponent is small, and thus a change of a small-significant bit in the mantissa causes a "small jump" from one representable number to the next one. On the other hand, when the exponent is large, the same change in the mantissa causes a "larger jump". To illustrate this, let's consider two values for the mantissa, $m_1 = 0.001$ and $m_2 = 0.010$, and similarly two unbiased exponents, namely $E_1 = 1$ and $E_2 = 16$. We can build four positive floating-point numbers from these parameters (setting in both cases $s = 0$), namely:

- $x_{m_1,E_1} = (2.25)_{10}$
- $x_{m_2,E_1} = (2.5)_{10}$
- $x_{m_1,E_2} = (73,728)_{10}$
- $x_{m_2,E_2} = (81,920)_{10}$

The change in the mantissa, in both numbers, is only of 0.001, which is a small increase. Nevertheless, the difference between $x_{m_2,E_1}$ and $x_{m_1,E_1}$ is small, namely $(0.25)_10$, while the difference between $x_{m_2,E_2}$ and $x_{m_1,E_2}$ is much larger, namely 8,192.

## 2.2 Arrays, Matrices, Pointers, and Structures

Among the almost infinite number of structures which can be devised to represent any kind of organised or unorganised data, the simplest one is the *array*[4], consisting of a collection of *elements*, each one being identified by a unique index. Usually, array indices are integers, on which a strict order is defined, so this order is logically transferred to the content of the array, making it a *sorted* data structure. This ordering makes arrays a logically *linear* structure, so they are usually graphically represented as in Figure 2.7(a).




(a) Graphical Representation (b) Memory Storage

**Figure 2.7:** An Array data structure

Usually, the index is restricted to a certain range of consecutive integers (or consecutive values of some enumerated type), and the address of an element is computed by a linear transformation on the indices. Consider in fact that primitive types have different sizes—a `char` can be represented using one byte, while an `int` might require a 4 bytes— and we can build arrays of any of them. The memory model used by standard von Neumann architectures is the flat memory model (as discussed in Section 1.1.1), so that an array `a` is actually represented in memory as in Figure 2.7(b).

Therefore, if we associate with `a` the *address* of the first element of the array, the displacement operator `[]` will allow to get the *address* of the $i$-th element of the array, given the index $i$. This happens because, in high-level programming languages, `a` carries the *type* of the elements of the array, which can be easily mapped to their *size*. Therefore, in the general case, the following equality holds:

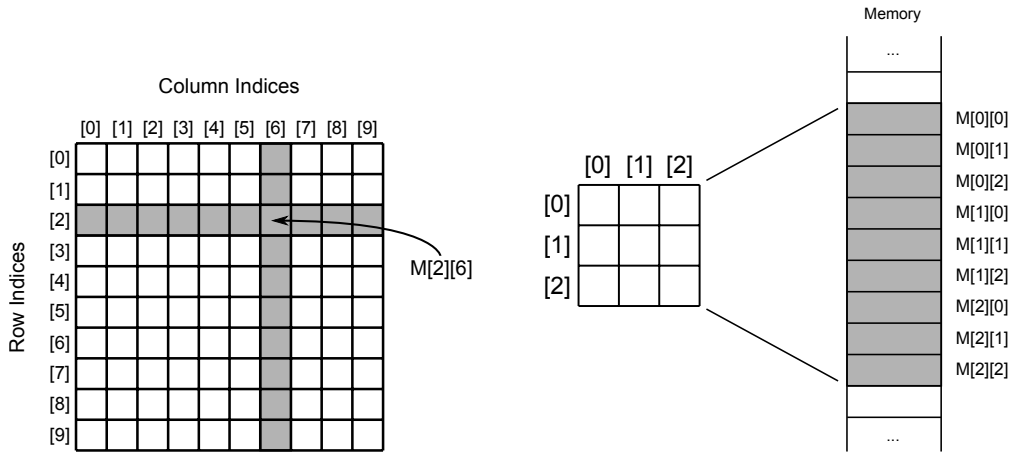
$$\texttt{a[i]} = \texttt{a} + \texttt{size} * \texttt{i} \tag{2.14}$$

In assembly programming the notion of type is not carried with the variable—actually we will see in Chapter 6.2 that the *size* of a variable is used when declaring a variable, to define how much space in memory it will take, but this information is not kept in the program. Therefore, when accessing different data items, different flavours of the same instructions shall be used (or different computations shall be done in advance) so as to tell the CPU how much memory must be actually accessed.

Concerning the indices, from a high-level programming language point of view, there are three ways in which the elements of an array can be indexed:

- **zero** (zero-based indexing): The first element of the array is indexed by subscript 0 (which is proper, e.g., of the C programming language). In this case, an array of $N$ elements has valid indices which go from 0 to $N - 1$.

[4] In computer science, the term *vector* is often used in an interchangeable way to identify an array, although there is not a right correspondence with the mathematical equivalent. A more correct analogy there would be with *tuples*.




(a) Graphical Representation (b) Memory Storage

**Figure 2.8:** A Matrix data structure

- **one** (one-based indexing): The first element of the array is indexed by subscript 1 (which is proper, e.g., of Fortran or Matlab). The range of valid indices, here, is from 1 to $N$.
- $n$ ($n$-based indexing): The base index of an array can be freely chosen (supported, e.g., by Pascal). This latter case encompasses also some languages which allow as subscripts more complex data types, like strings in the case of, e.g., PHP.

When programming in assembly, the only very meaningful way to address arrays is zero-based indexing[5]. In fact, from the above discussion on the displacement operator, and considering again Figure 2.7(b), it is clear that to access an array in assembly the only information needed is its starting address, the involved index and the size of the elements. Hence, since the address of the array is actually the same address as the first element's, the general Equality (2.7(b)) holds as well for the first element if it is associated with $i = 0$.

There is actually an additional piece of information which is required to properly handle an array: its *size*. If we look again at Figure 2.7(b), we can imagine that after the last element of the vector any other variable could be stored, perfectly adjacent to it. Therefore, to avoid accessing other variables' data thinking that it's part of the array, the current index should be matched against the total size. Yet, while high-level programming languages can do this automatically (like Java, which can automatically throw an `OutOfBoundException`), when programming in assembly this must be done by hand. Moreover, it is not even possible to manually determine the size at runtime as it is possible in some circumstances in C executing a statement like[6] `sizeof(a) / sizeof(a[0])`. In fact, upon the initialization of the vector, a *separate* variable should be used to keep the total number of vector's elements.

[5] Edsger W. Dijkstra, in his article "*Why numbering should start at zero*" argued that for style and simplicity, zero-based arrays should be preferable over one-based, and languages using the latter were much more error prone. Indeed, half-open intervals compose pretty well, so having arrays the indices of which are in the range $[0, n)$ makes arithmetic easier when splitting, concatenating, or counting elements, which are very common operations as we will discuss in this Section.

[6] This is only possible when dealing with an array value that has *not* decayed to a pointer: A pointer does not record anything about the array used to initialize it!

Arrays can have multiple dimensions, thus it is not uncommon to access an array using multiple indices. Two-dimensional arrays are usually referred to as *matrices*, due to their similarity with their mathematical counterpart, when depicting their graphical representation, like the one in Figure 2.8(a). Arrays with a higher number of dimensions are more generally called $n$-dimensional arrays.

Let us now consider the following matrix (i.e., a two-dimensional array) `M`:

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix} \tag{2.15}$$

This matrix must provide access to any valid element, for example 21, stored in the first column and second row. One common (zero-based) syntax used to access this element is `M[1][0]`, thus two indices are used for a two-dimensional array, three for a three-dimensional array, and n for an $n$-dimensional array. The number of indices needed to specify an element is called the *dimension*, *dimensionality*, or *rank* of the array.

Nevertheless, `M[1][0]` is not the only available syntax which a high-level programmer could use to access element 12, as other programming languages would require `M[0][1]` instead. This difference is associated with the memory representation used to store the values, more precisely their order. In the general case, a matrix (and in a similar way any $n$-dimensional array) can be stored in memory as in Figure 2.8(b). In fact, an $n$-dimensional array is necessarily stored in memory as a linear array, due to memory's flat model. The difference in the two writings, namely `M[0][1]` and `M[1][0]`, depend on whether the linearization process follows the rows or the columns, giving rise to the names *row-major order* and *column-major order*[7].

In particular, matrix `M` as of Equation 2.15 would be stored as follows in the two orders:

| Index | Column-Major | Row-Major |
|---|---|---|
| 0 | 11 | 11 |
| 1 | 21 | 12 |
| 2 | 12 | 13 |
| 3 | 22 | 21 |
| 4 | 13 | 22 |
| 5 | 23 | 23 |

When going to higher dimensions, the concept can be easily generalised. In particular, for a $d$-dimensional $N_1 \times N_2 \times \cdots \times N_d$ array with dimensions $N_k$ ($k = 1 \dots d$), a given element is specified by a tuple $\langle n_1, n_2, \dots, n_d \rangle$ of $d$ (zero-based) indices $n_k \in [0, N_k - 1]$.

In row-major order, the *last* dimension is contiguous, so that the memory-offset of this element is given by:

[7] Although it may appear that this difference is only theoretical, it had significant practical implications because two extremely famous programming languages, namely Fortran and C, were built using this two different representation. The result has been that, even if the assembly code produced by compilers of the two languages was perfectly compatible, this data representation made them not, forcing programmers to continuously transpose matrices in order to call C routines from Fortran code, and vice versa, significantly hampering the performance of the final program.

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1)\cdots))) = \sum_{k=1}^{d} \left( \prod_{i=k+1}^{d} N_i \right) n_k \tag{2.16}$$

In column-major order, the *first* dimension is contiguous, so that the memory-offset of this element is given by:

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d)\cdots))) = \sum_{k=1}^{d} \left( \prod_{i=1}^{k-1} N_i \right) n_k \tag{2.17}$$

Nevertheless, this is not the only possible memory representation of matrices (and, in general, of $n$-dimensional arrays). In fact, this representation is possible only when the total size is known at compile time. Indeed, the exact space for the involved data is reserved, while if the size depends, for example, on some user input, this space requirement cannot be known in advance. In this specific case, in higher-level languages like C the initialization is done as in Example **??** using the malloc memory allocator.

**Example 2.2.1:** example:multidimensional-arrays

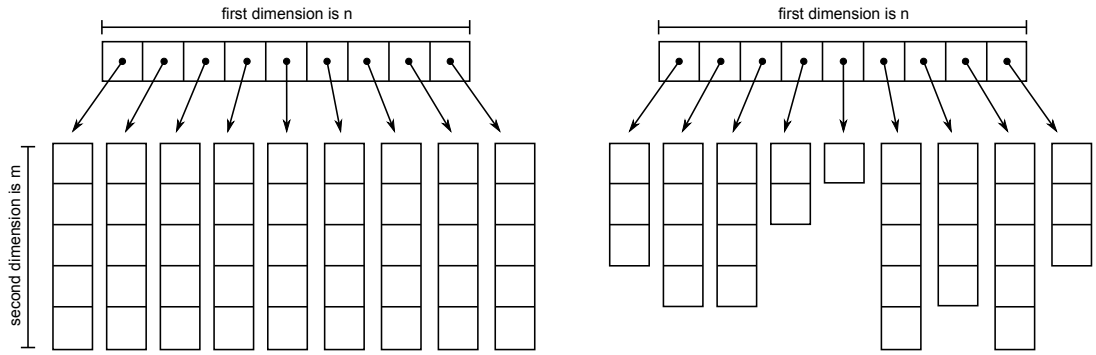
Allocation of two-dimensional arrays of `int` in C

```c
int N;  // The number of rows and columns, taken from the user

// Matrix elements are accessed via a 2-level pointer
int **M;

// Allocate an array of row pointers
if( (M = malloc(N * sizeof(int *)) ) == NULL) {
    /* error */
}

// Iterate to allocate each row
for (i = 0; i < N; i++) {
    if ( (M[i] = malloc(N * sizeof(int)) ) == NULL) {
        /* error */
    }

    /* initialize row content here */
}
```

The outcome of this code example is shown in Figure 2.9(a), and the logic behind it is quite straightforward: first allocate space for pointers to each row of the matrix, and then allocate (separately!) space for each row. Everything is glued using pointers.

The question is: if we know that the matrix will be composed by `N`×`N` elements, why couldn't we simply allocate a *linear* vector of `N`×`N` elements, resembling the contiguous allocation that we have seen so far? There are two answers to this questions. One is that the programmer could be actually interested in reserving memory for an array of arrays, rather than a matrix. The difference here is subtle, but the idea is what is depicted in Figure 2.9(b): the "matrix" has rows of different lengths.

In this second case, each row of the matrix will have a different length. So it is important to find out




(a) Regular Size (b) Variable Size

**Figure 2.9:** Dynamic Matrices

a proper way to know what is the length of each of them, during accesses. To keep this information, a second array of integers could be used, to store the length of each row. Otherwise, if the elements of the matrix belong to a certain domain, an element out of this domain (often named *sentinel value*) could be used to mark the end of the row. For example, if a matrix of non-negative integers is being used, the value -1 could be a good candidate to mark the end. Anyway, in both cases, the memory footprint is one additional integer for each row. What drives the choice is the algorithm that is being implemented, yet as we will discuss in Section 11.2, keeping data which are accessed close in time contiguous in space as well is likely to increase performance, so the solution using an element out of the domain of interest to mark the end of rows could be more efficient in the general case.

The second answer to the previous question is related mostly to some syntactic sugar given by high-level programming languages. In fact, if we allocate a matrix as an `N×N` contiguous array, then we would not be allowed by the language to access elements using the `M[i][j]` syntax. Instead, whenever a pointer is found, the `[]` operator is interpreted by the compiler as "take the pointer, dereference it, and jump to the element associated with the passed index".

Nevertheless, this behaviour of the `[]` operator is different from that we have seen before. In fact, when accessing a contiguous matrix, the double offset `[i][j]` is translated to a displacement from the initial address of `M` which is computed as:

$$address = i * N * size + j * size \tag{2.18}$$

while in the latter case the content of the item at index $i$ in the first array is actually accessed. This difference in the behaviour of the `[]` operator is a facility offered by some programming languages to access $n$-dimensional arrays in the same way independently of their memory shape, but often leads to the misconception that *arrays* and *pointers* are perfectly interchangeable.

Instead, while there are strong relations between pointers and arrays, they are very different. In fact, a pointer to an array can be visualized as in Figure 2.10. Comparing it with Figure 2.7(b), we can easily spot that there is an additional level of indirection. In the general case, a pointer to *any* data structure is simply a variable large enough to store an address, and therefore its size is *independent* of the pointed data structure.

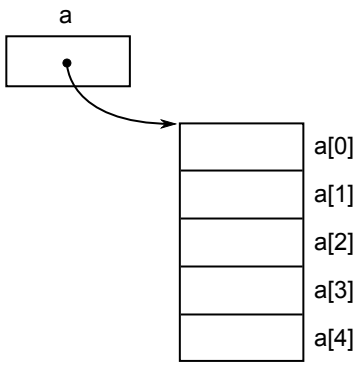



**Figure 2.10:** Arrays and Pointers: the difference

The strong relation between arrays and pointers is that the variable name used to declare an array can be easily decayed to a pointer to the first element. This assignment is therefore perfectly legit:

```
int a[4] = {0, 1, 2, 3};
int *p = a;
```

so that accessing to `a[1]` and `p[1]` will both generate the address where `1` is stored. In general, according to the previous example, `a` and `&a` will generate the same exact address. In C, nevertheless, `&a` and `a` have a different meaning. In fact, while the generated address is the same, `a` will be translated to a pointer to the first element of the array, while `&a` will be evaluated as the address of the *whole* array, meaning that the type of `a` is `int *`, while the type of `&a` is `int (*)[size]`. Therefore, pointers arithmetic will give different outcomes, as in Figure 2.11.

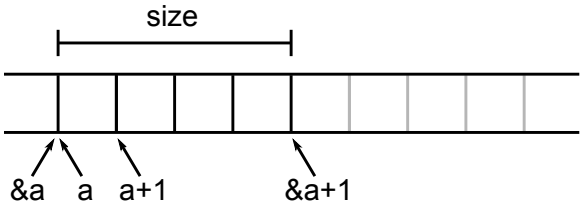



**Figure 2.11:** Arrays and Pointers: the difference

This confusion can sometimes produce wicked bugs, when using high-level languages. For example, the code snippet in Example 2.2.2 is perfectly legit for any C compiler. By using the `extern` keyword, the programmer is simply telling that the array of integers that the software wants to access is defined somewhere else, specifically in another module, and that the access should be done by using pointers. Yet, if the actual arrays is declared as:

```
int array[] = {0, 1, 2, 3, 4};
```

the programmer might think that the "equivalence" between arrays and pointers will make everything perfectly correct. Instead, the variable `array` is represented in memory as a real array, namely as in Figure 2.7(b), while the compiler will emit code which works with pointers, which are stored in memory as in Figure 2.10. Thus, since the two pieces of code are joined together by the linker, no

error (not even a warning!) will be issued during compiling[8], and the final program will likely crash. In fact, the first element of the array stores the value 0, which will be interpreted as an address, making the CPU access memory address 0. In most computing architectures, address 0 is reserved, usually associated with the `NULL` pointer[9], thus causing the fault.

**Example 2.2.2:** Accessing Array via pointers

```
#include <stdio.h>

extern int *array;

int main(void) {
    printf("Third element of the array is %d\n", array[2]);
    return 0;
}
```

The CPU, nevertheless, has no idea about anything of this, as it works only with addresses. Therefore, these differences which sometimes can cause confusions are not present when directly implementing assembly code to be executed by the processor, making assembly programming a bit more easy, provided that the programmer has a clear view of data stored in memory. Yet, in some situations, direct assembly code has to interact with modules or libraries implemented in higher-level languages, and in that situation it is of vital importance to know how compilers are translating these various language features in machine instructions.

To conclude our overview on arrays, we recall that once declared, a vector has a fixed size. Very high level languages, like C++ or Java, offer facilities to transparently use resizable arrays. Other languages like C give the possibility to use memory allocation functions like `realloc` to reserve a new memory area (of higher or smaller size) and make a copy of the content of the old array. In assembly, there is no direct way to do this, again because the CPU has no notion of what data is stored at which memory address. Then, if a programmer does really want to use resizable arrays in assembly without relying, for example, on external calls to `realloc`, he has to keep track manually of what memory regions are already used and which are free, so as to "mark" a free region as used and manually make a copy of the content of the old array. This could be a complex operation, and the goal of libraries is exactly to reuse code.

High-level languages like C provide *structures* to represent and organize more heterogeneous data. In fact, arrays are just a way to deal with group of values rather than single elements. In assembly, data structures are simply block of contiguous memory where single data items, arrays and references are stored, always using the same internal organization.

In C, a structure is declared like the following:

```
struct book {
```

[8] If it is not clear why this is so, we encourage the reader to jump to Appendix D for a comprehensive explanation of the compiling process of programs.

[9] The z64 architecture does not associate address 0 with `NULL`, yet it is anyhow a reserved address, as it will be discussed in Chapter 7.

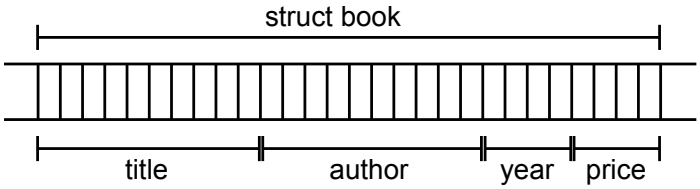



**Figure 2.12:** Structure in Memory

```
  char title[10];
  char author[10];
  int publication_year;
  float price;
};
```

where the internal variables are called *members*, which can have different types and sizes. Once a variable of structured type is declared, its organization in memory is as in Figure 2.12, and each member can be accessed using a syntax like `var.member`. In particular, the compiler translates the dot `.` operator to a set of assembly instructions which count how many bytes should be skipped from the beginning of the structure in memory, to find the initial address of the member.

In assembly there is no explicit notion of a structured data type, so they must be manually managed. Specifically, as mentioned, a structure is declared reserving a block of contiguous memory (to this end, the z64 has several *directives*, which will be discussed in Section 6.2), and the initial address of this memory area is referred to as the *base address of the struct*. To access a member, its *displacement* must be known in advance, which is possible because the structure has a fixed organization in memory. Then, the initial address of the member can be easily computed by summing this displacement to the base address of the structure, so as to allow a direct access to memory to read the content of a member.

Of course, this organization can be extended (even though it would be a bit more complicated) to implement algorithms able to work on *arrays of structures*, which combine the notion of arrays and structures that we have seen so far. In fact, they consist of contiguous memory areas, the total size of which is a multiple of one single structure size, and accesses are made by first computing the base address of one element of the array starting from its index. In Section 6.6, after having covered the basics of assembly programming on the z64 CPU, we will illustrate using several examples how this operations can be done programmatically.

## 2.3 Characters and Strings

A character is probably the most common primitive data type used when programming, as the communication between the machine and humans is necessarily character-oriented. Since CPUs are only able to manage bits, a character is a sequence of bits which, conventionally, is associated with a specific symbol. In the common practice, some characters are associated with *control symbols* (or *non-printing character*) which are used to represent control information rather than symbols.

**Historical Note 2.3.1:** Non-Printing Characters

In computer science and telecommunications, a *control character*, or a *non-printing character*, is a binary value which is not mapped to a written symbol, rather it is a command which, when used in in-band signaling, causes effects different than printing symbols.
The control characters were designed to fall into a few groups: printing and display control, data structuring, transmission control, and miscellaneous. Printing control characters were first used to control the physical mechanism of printers, the earliest output device. Among these, the carriage return character (CR) causes it to put the character at the edge of the paper at which writing begins, and the new line (or line feed) (LF/NL) causes the device to put the printing position on the next line. They are used as well on console screens, to move the cursor to the new line.
Separators (File, Group, Record, and Unit: FS, GS, RS and US) were adopted to structure data, usually on a tape, in order to simulate punched cards. These symbols have become much less useful in the modern days.
The transmission control characters were intended to structure a data stream, and to manage retransmission or graceful failure, as needed, to cope with transmission errors. They were thought as a means to transfer text on a network in a coherent way, having in the same representation both the data interesting for the humans, and the ones interesting for the machines in charge of the transmission. A large number of control characters fall under this category. Among them, the start of heading (SOH) character was to mark a non-data section of a data stream—the part of a stream containing addresses and other housekeeping data. The start of text character (STX) marked the end of the header, and the start of the textual part of a stream. The end of text character (ETX) marked the end of the data of a message.
An interesting miscellaneous control character was the *bell* character (BEL), intended to cause an audible signal in the receiving terminal, usually through a buzzer installed on the computer.

By far the most common character representation is the *American Standard Code for Information Interchange* (ASCII), although in most recent years new standards came out, like Unicode, mainly to extend the range of supported symbols, especially to allow rendering non-latin alphabets. Nevertheless, due to its simplicity, we will only be concerned with ASCII, considering Unicode simply an extension of this standard.

Traditional standard ASCII codes are made up of seven bits, allowing to represent 128 different values. Values in the range $[0, 31]$ and 127 are control codes, while the remainder are associated with printable symbols: letters, digits, punctuations, and so on. Since one byte is eight bits wide, when one ASCII character is mapped to one byte[10] there is one spare bit. In the early days, this bit was used to maintain a parity bit (which will be discussed in Section 2.5) to check for errors, which could be due for example to tape demagnetization. Then, output devices became more sophisticated, being able to display more characters, so the eighth bit started to be part of the actual representation, leading to the *extended ASCII*. The complete extended ASCII encoding is shown in Table 2.5. In this representation, characters associated with a numeric representation in the (extended) range $[128, 255]$ are often referred to as *top-bit set characters*, and they comprehend accented letters (to allow for interaction as well with non-English languages using the latin alphabet), some mathematical symbols and *box-drawing* characters.

[10] Indeed, traditionally the primitive `char` type is one byte wide.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000 | NUL | 033 | ! | 066 | B | 099 | c | 132 | ä | 165 | Ñ | 198 | ã | 231 | þ | | | | |
| 001 | Start Of Header (SOH) | 034 | " | 067 | C | 100 | d | 133 | à | 166 | ª | 199 | Ã | 232 | Þ | | | | |
| 002 | Start Of Text (STX) | 035 | # | 068 | D | 101 | e | 134 | å | 167 | º | 200 | ╚ | 233 | Ú | | | | |
| 003 | End Of Text (ETX) | 036 | $ | 069 | E | 102 | f | 135 | ç | 168 | ¿ | 201 | ╔ | 234 | Û | | | | |
| 004 | End Of Transmission (EOT) | 037 | % | 070 | F | 103 | g | 136 | ê | 169 | ® | 202 | ╩ | 235 | Ù | | | | |
| 005 | Enquiry | 038 | & | 071 | G | 104 | h | 137 | ë | 170 | ¬ | 203 | ╦ | 236 | ý | | | | |
| 006 | Acknowledge (ACK) | 039 | ' | 072 | H | 105 | i | 138 | è | 171 | ½ | 204 | ╠ | 237 | Ý | | | | |
| 007 | Bell | 040 | ( | 073 | I | 106 | j | 139 | ï | 172 | ¼ | 205 | ═ | 238 | ¯ | | | | |
| 008 | Backspace (BS) | 041 | ) | 074 | J | 107 | k | 140 | î | 173 | ¡ | 206 | ╬ | 239 | ´ | | | | |
| 009 | Horizontal Tab | 042 | * | 075 | K | 108 | l | 141 | ì | 174 | « | 207 | ¤ | 240 | - | | | | |
| 010 | Line Feed **(LF)** | 043 | + | 076 | L | 109 | m | 142 | Ä | 175 | » | 208 | ð | 241 | ± | | | | |
| 011 | Vertical Tab | 044 | , | 077 | M | 110 | n | 143 | Å | 176 | ░ | 209 | Ð | 242 | ‗ | | | | |
| 012 | Form Feed **(FF)** | 045 | - | 078 | N | 111 | o | 144 | É | 177 | ▒ | 210 | Ê | 243 | ¾ | | | | |
| 013 | Carriage Return **(CR)** | 046 | . | 079 | O | 112 | p | 145 | æ | 178 | ▓ | 211 | Ë | 244 | ¶ | | | | |
| 014 | Shift Out | 047 | / | 080 | P | 113 | q | 146 | Æ | 179 | │ | 212 | È | 245 | § | | | | |
| 015 | Shift In | 048 | 0 | 081 | Q | 114 | r | 147 | ô | 180 | ┤ | 213 | ı | 246 | ÷ | | | | |
| 016 | Dataline Escape (DLE) | 049 | 1 | 082 | R | 115 | s | 148 | ö | 181 | Á | 214 | Í | 247 | ¸ | | | | |
| 017 | DC 1 (XON) | 050 | 2 | 083 | S | 116 | t | 149 | ò | 182 | Â | 215 | Î | 248 | ° | | | | |
| 018 | DC 2 | 051 | 3 | 084 | T | 117 | u | 150 | û | 183 | À | 216 | Ï | 249 | ¨ | | | | |
| 019 | DC 3 (XOFF) | 052 | 4 | 085 | U | 118 | v | 151 | ù | 184 | © | 217 | ┘ | 250 | · | | | | |
| 020 | DC 4 | 053 | 5 | 086 | V | 119 | w | 152 | ÿ | 185 | ╣ | 218 | ┌ | 251 | ¹ | | | | |
| 021 | Negative Acknowledge (NAK) | 054 | 6 | 087 | W | 120 | x | 153 | Ö | 186 | ║ | 219 | █ | 252 | ³ | | | | |
| 022 | Synchronous Idle | 055 | 7 | 088 | X | 121 | y | 154 | Ü | 187 | ╗ | 220 | ▄ | 253 | ² | | | | |
| 023 | End Of Transmission Block | 056 | 8 | 089 | Y | 122 | z | 155 | ø | 188 | ╝ | 221 | ¦ | 254 | ■ | | | | |
| 024 | Cancel | 057 | 9 | 090 | Z | 123 | { | 156 | £ | 189 | ¢ | 222 | Ì | 255 | | | | | |
| 025 | End Of Medium | 058 | : | 091 | [ | 124 | \| | 157 | Ø | 190 | ¥ | 223 | ▀ | | | | | | |
| 026 | Substitude | 059 | ; | 092 | \ | 125 | } | 158 | × | 191 | ┐ | 224 | Ó | | | | | | |
| 027 | Escape **(ESC)** | 060 | < | 093 | ] | 126 | ~ | 159 | ƒ | 192 | └ | 225 | ß | | | | | | |
| 028 | File Seperator | 061 | = | 094 | ^ | 127 (DEL) | ⌂ | 160 | á | 193 | ┴ | 226 | Ô | | | | | | |
| 029 | Group Seperator | 062 | > | 095 | _ | 128 | Ç | 161 | í | 194 | ┬ | 227 | Ò | | | | | | |
| 030 | Record Seperator | 063 | ? | 096 | ` | 129 | ü | 162 | ó | 195 | ├ | 228 | õ | | | | | | |
| 031 | Unit Seperator | 064 | @ | 097 | a | 130 | é | 163 | ú | 196 | ─ | 229 | Õ | | | | | | |
| 032 | *SPACE* (SP) | 065 | A | 098 | b | 131 | â | 164 | ñ | 197 | ┼ | 230 | µ | | | | | | |

**Table 2.5:** Extended ASCII Map

When several characters are stored contiguously in memory, they form an array of characters which is called a *string*. Strings are a necessary datatype, as they are fundamental to interact with humans. The main problem when dealing with strings is the fact that it is hard to impose a predefined length as, depending on the situation, the text to be represented (for example on the screen) can be whichever. Therefore, the common representation of strings relies on a specific sentinel value, namely the `NUL` ASCII character, associated with the decimal value 0. This sentinel value is called *string terminator*, in high level languages is often represented with the character `'\0'`, and strings using this representation get the name of *null-terminated strings*. For example, in memory, the string `"Computer"`[11] is stored as in Figure 2.13.

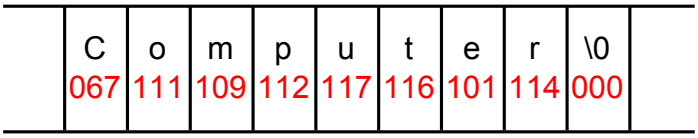



**Figure 2.13:** A String in Memory

The C language follows this representation, but it's not the only one. For example, most Fortran compilers represent a string as an array of characters with the length stored as an integer along with the array. This variability in representation of string is of no real problem, until some assembly code generated from one compiler has to interact with the code generate by another compiler. Again, the CPU has no idea about what is stored at a specific memory address, so dealing with strings in assembly can be made consistent by the programmer himself. Great care should be taken, nevertheless, when interacting with libraries written by others.

## 2.4 Bit Fields and bitmaps

All the datatypes that we have discussed so far have a length which is at least a multiple of one byte. Yet, bytes are composed by single bits, and (although very small) one single bit can be enough to store a piece of significant information.

As we have already discussed, a bit's value can be either 0 or 1, and this is enough, for example, to distinguish among some property of a system to be true or false. While high-level programming languages allow the usage of boolean variables to store the truth or the untruth of something, a single bit being used to represent this information is called a *flag*. As we will discuss thoroughly in Section 5.4, CPUs heavily rely on flags to store status information about what is the outcome of the execution of some instructions, or to keep information about what execution mode is currently active.

Three fundamental operations should be supported to correctly use a bit field:

- *Bit setting*: a specific bit is set to the value one;
- *Bit clearing*: a specific bit is set to the value zero;
- *Bit testing*: the value of a specific bit from the bit field is extracted to check its value.

[11] When programming, the common way to represent a single character is by surrounding the letter with single quotes, while strings are surrounded by double quotes. When a double quote is used around letters, the string terminator `\0` is transparently inserted by the compiler, so a string declared as `"string \0"` will actually have two `NUL` bytes at the end.

Since processing units often are not able to address single bits from working memory—the smallest addressable size is usually one byte—flags are aggregated into bytes (or words, longwords, and quadwords), which take the name of *bit fields*. To read the value of one bit in a register or in a memory location (or to alter its value), some more complex operations should be performed, which use the notion of *masks*, but we will discuss how these operations can be implemented in Section 6.10, after having seen the basics of assembly programming.

The benefit of using bit fields is that they are more concise for representing multiple true/false values, but the number of elements which can be stored is limited by the size of the primitive data type used to keep the bit field. In fact, a bit field is distinguished from a bit array in that the latter is used to store a large set of bits indexed by integers and is often wider than any integral type supported by the language. Bit fields, on the other hand, typically fit within a machine word, and the denotation of bits is independent of their numerical index.

When composing multiple bit fields together, a possibly unbounded number of true/false values can be stored. Nevertheless, in this case a two-level identification of the specific bit must be considered: first, the elementary datatype in the array should be identified. Then, in the element, the specific bit should be determined. A bit vector is often referred to as a *bitmap*, which is a data structure used, for example, to represent in a very concise way the usage of large portions of virtual memory in memory allocators.

When the size of a bit vector is squared, the data structure is often called *bit board*, as this representation was commonly used in programs that play board games, such as chess, reversi, or checkers. In this representation, each bit represents a position on the board or a state of the game. The compact structure was very important in early days entertainment systems, as it allowed for a very wise usage of memory and very fast manipulation of game rules. In fact, by having the possibility to check multiple positions on the board by looking at a single word, the software can answer some questions or execute some queries very quickly. Some of the questions which could be answered with a single memory load were something like "*does the white player have any pawns at the edge of the board*?". Nevertheless, the massive compression of information and encoding was often leading to programs which were not easy to debug or maintain.

## 2.5 Codes

## Questions and Exercises

**Exercise 2.1 (*General Questions*):**

1. Explain what is the difference between an array and a pointer.

# 5 Design of a Multi-Cycle CPU

Throughout this Chapter we will develop the design of the first version of the z64 CPU, starting from the von Neumann architecture. To this end, we will first establish the functional and cost/performance motivations that lead the definition of the z64 architecture, and then we will enter the details of each component's project, using combinational and sequential logic networks. In particular, the von Neumann architecture will be considered in a slightly modified fashion. In fact, as shown in Figure 5.1, some of the logical building blocks of the architecture are mapped to the actual CPU. These building blocks are of course the Control and the Processing Units, as they comprise all that is required to carry out the actual operations by the CPU. Additionally, a portion of the working memory is directly placed within the CPU: this working memory is what we usually call the *registers*. This choice allows us to have a very small portion of the memory which can be accessed for the execution of the operations at the same speed according to which the CPU operates—remember that working memory is usually orders of magnitude slower than the operation speed of the CPU.

Single-cycle processors suffer from a very poor speed performance, and a very low data throughput. In fact, due to its combinational nature, before switching to the execution of the next instruction, all the gates must have stabilized their output. This means that the clock frequency must be set according to the duration of the longest instruction, which can be much longer than the average time

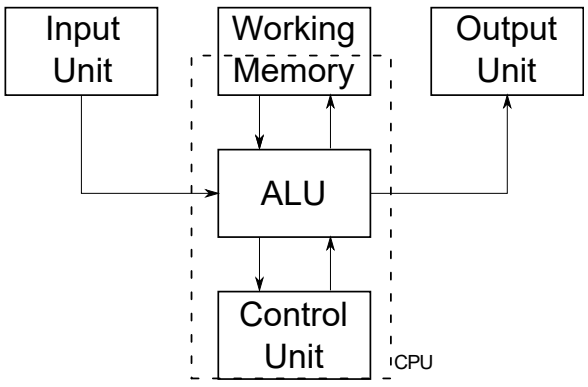



**Figure 5.1:** Von Neumann Architecture: the PU, CU and a part of Working Memory are placed in the CPU.

taken by other instructions to complete their operation.

On the other hand, a multi-cycle CPU breaks each instruction into several fundamental parts, and the clock cycle is then set to the duration of the longest part. In this kind of project, where control signals travel for a shorter distance, the complexity of the hardware can be reduced with respect to the single-cycle CPU—in fact, as we will see, the same components can be used for different purposes—and the overall execution speed can be increased. The basic part into which an instruction is broken up are commonly the following three:

- **Instruction Fetch** (IF): during this phase, the CPU queries the memory to obtain the binary representation of the next instruction to be executed. During this phase, the content of a special purpose register, called the *instruction pointer*, is used to retrieve (fetch) from memory the machine representation of the current instruction which, as we will see, is 64-bits long. At the same time, its content is incremented by 8, which is exactly the number of bytes used to represent an instruction. In this way, when the fetch phase is completed, RIP already points to the address of the next instruction.
- **Instruction Decode** (ID): once the machine-representation of the instruction is copied into another special-purpose register, called the *instruction register*, the Control Unit determines what are the steps required to complete its execution, so that the right set of control signals is determined.
- **Execution** (EX): this phase comprises all the operations that are related to the actual execution of the operation. During this phase, different steps can be executed, depending on the nature of the instruction, as determined during the ID phase. In particular, if some operands are located in memory, it is accessed again to retrieve them. The number of operands which can be stored in memory depends on the actual semantic power offered by the instruction set of the CPU. As we will see, the z64 allows only for an operand to be located in memory and in the internal registers. By the design which we will present in this chapter, this reduces to up to one additional memory access for all the instructions of the ISA. This step is mandatory only for a subset of the instructions. Then, all the operands of the instructions are passed to the computing circuitry (namely, the ALU or the shifter), and the result of the operation is computed. Again, this phase is mandatory only for a subset of the operations. Finally, if the instructions requires so, the result of the operation is written back into an internal register of the CPU, or to memory.

Concerning again hardware reuse, by this differentiation of the execution phases of one instruction, we can see that for example the same ALU can be used both for updating the instruction pointer during the IF phase, and to perform arithmetic computations during the EX phase. Moreover, different types of instructions might have less phases. Think, for example, of an instruction moving data from memory, and an instruction which changes the control flow of the program, by jumping to an instruction at a given address. They will both have the same IF and ID phases, but the number of steps in the EX phase will be different. Additionally, every phase could require a different number of control signals in different sub-phases (what we will call *micro-operations*, or $\mu$-ops) depending on the complexity of the involved logic. In this sense, different instructions will require a different number of clock cycles to complete their operation, which is another reason why the overall execution speed

of a program can be increased by this design.

As in every complex digital system's project, the z64 architecture is developed considering, at the same time, *functionalities*, *cost*, and *performance* of the overall final architecture. Given the high complexity of the functions to be carried out by the z64 CPU, its architecture (and consequently its project) is divided into two main entities, which interact among each other, as it has already been discussed in Chapter 1. On the one hand, we have the *Control Unit* (CU), whose goal is to supervise the execution of every function supported by the CPU, while on the other hand we have the *Processing Unit* (PU), which is ruled by the CU. The PU is composed of all the available registers, functional blocks (which operate on registers' content), and the interconnection structure, which is used to transfer data among registers and functional blocks. The CU is instead a sequential network which, driven by the assembled program, coordinates all the activities of the PU.

The CPU therefore is a logical structure that interprets programs written in machine language (which, we recall, is generated by the assembler starting from an assembly program). This is done through the execution of a set of microprograms, done by the CU, whose outputs (the commands) act on the PU (including the memory and the devices) to move or manipulate data. Each microprogram corresponds to a machine instruction and the microcode is identified by the opcode of the instruction.

It is easy to imagine that the number of control functions can be very high. Therefore, the easiest way to implement the CU is by relying on *microprogramming*, as discussed in Chapter 4. The problem of devising an efficient project for the CU is thus boiled down to determining an efficient algorithm which implements all the control functions which are required to execute every instruction which should be supported by the CPU (and, in the second place, the kind of microprogramming organization, and the proper selection of a microlanguage to be used in the algorithm). As for the PU, the major difficulty lies in the proper selection of its complexity degree. In general, the higher the complexity of the PU, the simpler will be the CU, and therefore we must pay the highest attention when designing the components of the PU. Since we have selected the von Neumann model as the reference architecture, our CPU will be able to execute only one single instruction at a time, and our PU can be therefore designed by relying on a restricted amount of hardware.

When presenting the possible architectures which can carry on the execution of z64 assembly programs, we will first present several choices that have been taken historically. In this way, the reader will be able to understand the guidelines used to design several real CPUs which have been developed in the past years. In fact, the architectural choices have been made considering the technology available at the moment and a cost/performance tradeoff. Therefore, the same von Neumann machine, despite being realized using the same logic-networks methodology, has been implemented using different technologies, starting from electromechanical switches and valves, up to semiconductors at different integration scales (LSI, VLSI, WSI, …).

We will therefore present the simplest solution first, which corresponds in terms of complexity to a computing system similar to the early 8-bits ones, such as the Zilog's Z80[1] or Intel's 8080, and we will later come to more complex solutions, so as to increase the overall performance, namely the programs' execution speed. Nevertheless, since our ultimate goal is to realize a 64-bits processing unit

[1] Zilog's Z80 is a very important CPU in the history of computing systems, as after 40 years since its introduction into the market it is still used as a microcontroller, for example in the Japanese railway system.

with an instruction set which is similar to the x86_64's (although extremely reduced), even the first implementation will provide 64-bits registers which can be accessed by the programmer as a whole (64 bits), in the lower half (32 bits), the lower fourth (16 bits), and in the lower eighth (8 bits). This allows the programmer to write software which uses 8, 16, 32, and 64-bits data, without requiring to implement in the processing unit multiple registers. Additionally, this possibility allows to run, on more recent CPUs (such as 64-bits ones), software written for older units (such as 8-bits ones) without the need to *recompile* the source code[2] (or *reassemble*, if the source is directly written in assembly). For the sake of simplicity, we will assume in the beginning that all data are only 64-bits long. This hypothesis will be later relaxed, once the basic architectural choices will be presented.

As a first approximation, a program's execution speed depends on the speed of each component of the CPU, and on the transfer speed of information (namely, data and instructions). Furthermore, as we will discuss later in Section 11.2, its speed depends as well on the *temporal* and *spatial* relations of data and instructions, during the execution of the program itself. In this part, we will only focus on optimizing programs' execution speed considering the speed of each component and data transfer, as it has been done in the early processing units. At the time, this choice was justified as well by the technological maturity, specifically by the level of integration that could be reached in a semiconductor chip.

To reduce the transfer time, the optimal solution would require to allocate all the logical components of the computing architecture on a single chip. Nevertheless, this solution is not viable, due to space constraints, so that the number of components that can be hosted on a single chip is far from being optimal. This limitation has led in the 70's to the development of conventional computing architectures (namely, those compliant with the von Neumann model) where the control subsystem, the computing subsystem, and a very limited portion of memory were grouped in a single entity which was named *central processing unit* (CPU). The organisation of a conventional computer with a CPU compliant with the von Neumann model, with a suitable interconnection structure to transfer information, can be specialised as in Figure 5.2.

Given the division between CU and PU of our processing unit, we must carefully consider the *cost* and the *performance* of each choice regarding their design, considering as well the effect (in terms of interactions) that each choice can have on the other subsystem.

## 5.1 Processing Unit implementation, under the hypothesis of 64-bits data

As we have mentioned, the z64 CPU is a conventional processing unit, and is therefore composed of:

- a computing unit (PU);
- a local memory;
- an interconnection structure;
- a control unit (CU).

[2] *Compiling* is the (automatic) translation from a human-readable form of code, such as the assembly language, to the corresponding machine representation. Throughout this book, we will go into details of this process, with Appendix D giving a more complete view on this complex task.

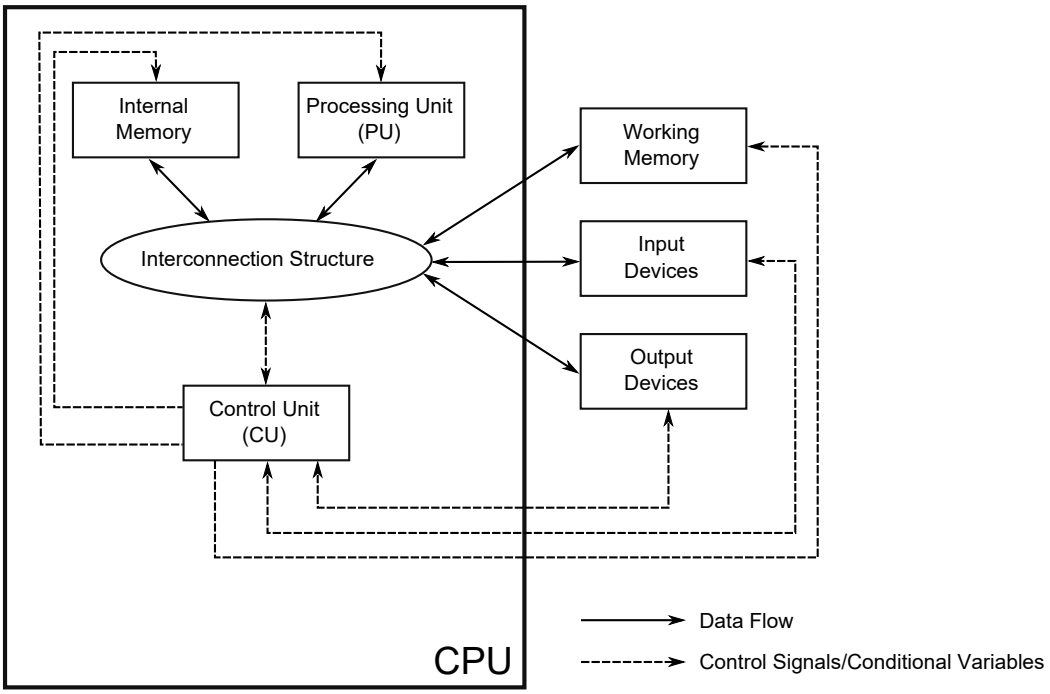



**Figure 5.2:** Logical Organisation of a Conventional Computer with a Processing Unit

The functional units which are part of the z64's PU are the *arithmetic logical unit* (ALU), and the rotation and shift unit (shifter). The interconnection structure is necessary to allow the communication among the local memory and the functional units.

The local memory is composed of a set of *general purpose registers*, a special register keeping the status flags (the *status register*) which are *visible* to the programmer (namely, they are accessible) by relying on the assembly instructions which can be interpreted by the CPU. Additionally, the local memory is composed of a set of registers which can be used only by the CU, and are therefore *invisible* to the programmer, although he can ask the CU to modify some of them according to some predefined rules. They are mainly used to interpret machine instructions, to keep partial results of operations, or to maintain the operands of particular functions which are implemented by components of the PU, such as the ALU.

Additionally, to fully respect the logical organization of the von Neumann architecture, the z64 CPU is connected as well to working memory and devices.

### Registers visible to the programmer

z64 is a 64-bits processing unit, which is equipped with 16 *physical* general-purpose registers, which are mapped to 64 *virtual* registers. The physical registers are all 64-bits wide, while the virtual registers are mapped to subportions of them, and therefore allow to operate as well on data which is 1-, 2-, or 4-bytes wide. Table 5.1 presents the list of physical registers, where their names are shown along with their binary encoding. The encoding will be directly reflected in the instruction's binary representation, allowing the Control Unit of the CPU to activate the specific logic related to accessing the registers in read/write mode, depending on the actual instruction being executed. In some cases, the

**Table 5.1:** z64 general purpose physical registers, with their encodings and common usage.

| Name | Encoding | Common Usage |
|---|---|---|
| RAX | 0000 | Accumulator Register |
| RCX | 0001 | Counter Register |
| RDX | 0010 | Data Register |
| RBX | 0011 | Base Register |
| RSP | 0100 | Stack Pointer |
| RBP | 0101 | Base Pointer |
| RSI | 0110 | Source Register |
| RDI | 0111 | Destination Register |
| R8 | 1000 | General-Purpose Register |
| R9 | 1001 | General-Purpose Register |
| R10 | 1010 | General-Purpose Register |
| R11 | 1011 | General-Purpose Register |
| R12 | 1100 | General-Purpose Register |
| R13 | 1101 | General-Purpose Register |
| R14 | 1110 | General-Purpose Register |
| R15 | 1111 | General-Purpose Register |

same name can be used to identify both the generic register (namely, the physical register, independently of the size associated with the operation being carried out on it) and the virtual register. To be clear about the naming convention that we are using throughout this book, if we refer to the generic physical register we will use the notation R15, while if we will specifically refer to the virtual register (for example, in a 64-bits operation) we will use the notation r15.

Each register has a common usage, which is shown in the third column of the table. This does not mean that a specific register *must* be used only according to that rule—they are general-purpose, nevertheless! Yet some (optimized) instructions provided by the ISA rely on these rules, and breaking them could result in an incorrect program behaviour. Additionally, if the *semantic* of the registers is respected, the whole program could be expressed by using a smaller amount of instructions, thus providing on the long run a higher performance and a reduced memory impact[3].

To access all virtual registers, different symbolic names are used, as shown in Figure 5.3. For example, to access the 32 least significant bits of the rax register, the register name eax should be used. Similarly, ax allows to access the least significant 16 bits, and `al` the least significant 8 bits. As another example, the register name r8 can be used to access the 64-bits wide R8 register, while the 32-bits, 16-bits, and 8-bits subportions are accessed with the names r8d, r8w and r8b, respectively. Concerning the DI register, the four variants are accessed using the names rdi, edi, di, and dil.

Looking carefully at Figure 5.3 we can note all these inconsistencies in how symbolic names are used to access subportions of the registers. This is a historic sedimentation, which can be understood only retracing the evolution of the Intel x86 architecture. In fact, the first processors produced by Intel were 16-bits CPUs, offering only 8 general purpose registers, exactly the first 8 shown in Table 5.1. At the time, the names of the registers where the mnemonics which could be used when writing assembly

[3]This is even more true in *real* Intel x86_64 ISA, where there are more compact (even in terms of byte length) instructions, which are optimized according to these rules.

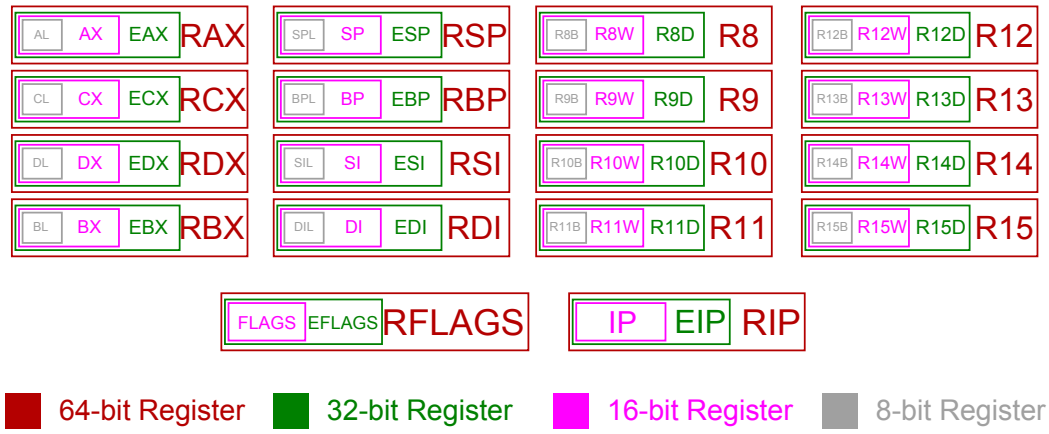



**Figure 5.3:** All available z64 registers and their subfields.

code to access their content. Accesses to four of these registers, namely AX, CX, DX, and BX, could be as well split into two different parts: the *low* bits and the *high* bits, which were giving rise to 8 general purpose 8-bits registers. To differentiate these accesses, the X in the mnemonic name was replaced by either L or H, depending on whether the programmer was accessing the low-order or high-order 8 bits. When Intel later developed 32-bits CPUs, all the registers where *extended* to 32-bits. Yet, to allow backward compatibility, the old 16-bits mnemonics where used to access the low-order 16 bits of each registers, and the letter E (for *extended*) was put before the names, to access all the 32 bits. In this evolution, the programmer did not have the possibility to directly access the high-order 16 bits of the registers, maybe because this was not seen as a common operation, and to do so, more complex bitwise operations should be used—we will discuss these operations in Section 6.10. Again, for backward compatibility, the suffixes L and H were kept, so the programmers were able to access bits 8–15 of four general purpose registers (AX to BX), but not the high-order 16 bits. Then, when AMD introduced the 64-bits extension to Intel's 32-bits ISA, they introduced 8 additional registers (R8 through R15), and allowed to access all the low-order 8 bits of all general purpose registers. To distinguish between the "old" 8 registers and the "new" 8 registers, they decided to keep the L suffix for the remaining four old general purpose registers, giving rise to the names SPL, BPL, `sil`, and `dil`. To address the subportions of the "new" 8 general purpose registers, they adopted the names commonly used by Intel to identify data of 8, 16, and 32 bits: byte, word, and double-word (which we will call throughout this book as a *longword*, to be consistent with the Assembly language which we will use to program the z64 CPU), giving rise to the suffices B, W, D. Therefore, if we consider the R8 register, we can access the three subportions with the symbolic names R8D, R8W, and R8B. In this evolution scenario, x86-compliant CPUs still give the possibility to access the AH, CH, DH, and BH, mainly for backward compatibility of the assembled software—this is something that, for simplicity and scarceness of applicability in general-purpose programming, we have dropped in the z64 CPU: there are no AH, CH, DH, and BH registers.

Therefore, as again shown in Figure 5.3, the z64 CPU offers 64 "virtual" registers, taken from the 16 "physical" registers described in Table 5.1: 16 byte-wide registers, 16 word-wide registers, 16 longword-wide registers, and 16 quadword-wide registers.

**Historical Note 5.1.1:** Register Names

CTC Datapoint 2200

Early in the 70's, Intel released the 8008 CPU. It was one of the first microprocessors, working with 8-bits wide data. It was extremely simple and it required a lot of additional chips to be used (for example, to address 16 kB of memory, the 14-bits address was stored in an external buffer, called *Memory Address Register*—MAR). It was designed for a specific purpose, namely for the Datapoint 2200 terminal, by Computer Terminal Corporation (CTC), and much of the modern instruction set organization of x86 CPUs owes to the design choices made at the time. This device has been incredibly important for Computer history, as it originated the first 8-bits microprocessor on a single chip, and it was an incredibly performing terminal, able to connect to various mainframes by loading different emulators from a tape device.

CTC had a major role in designing the instruction set of the Intel 8008, with Victor Pool and Harry Pyle leading the project. The hardware architecture, on the other hand, was designed by Federico Faggin and Hal Feeney from Intel. Nevertheless, the production of this chip was late, and CTC did not include into its Datapoint 2200, but Intel was later granted the possibility to sell this chip to other partners. This gave birth to a *primordial* x86 ISA, which later influenced the Intel 8080, which spread the x86 instruction set in the world.

Although the Intel 8008's instruction set was designed as a CISC one, CTC opted for introducing as well a set of instructions optimized (both in terms of memory usage and execution latency) resembling an *accumulator machine*. In modern days, since there is this heritage from the initial 8008 ISA, we still can find in x86 CPUs instructions which have implicit operands, namely registers which are accessed by the instruction but which do not appear in the actual mnemonic or representation.

Modern computing architectures no longer suffer from strict memory limits and poor CPU performance. Nevertheless, in past days, correctly using registers according to their original purpose and relying on "accumulator machine"-like instructions could make the difference between a more efficient and a less efficient program. This is still reflected in the names of the registers, which are the following.

**The Accumulator Register** (AX)

Although x86 are not accumulator machines, due to the aforementioned historical reasons the AX register mimics an accumulator register. All the operations can involve all the general purpose registers in Table 5.1, yet some instructions privilege the usage of the accumulator register. For example, in the x86 assembly language there are 9 basic instructions (`add`, `adc`, `and`, `cmp`, `or`, `sbb`, `sub`, `test`, and `xor`) which have a special 1-byte opcode to express operations between the accumulator and a constant. More specialized operations, like multiplication, division, or sign extension, can be executed only in the accumulator.

Additionally, since there are operations which privilege the accumulator, the x86 ISA offers special instruction to move data to that register—something which is proper of accumulator machines. For example, there are 16 1-byte `xchg` instructions that exchange the data between the accumulator and all the other general purpose registers. They could be not very useful in general programming, but show that the original project of the 8086 was preferring this register for the operations. There is additionally a special 1-byte `mov` instruction to move data into the accumulator register from a constant memory location.

**The Data Register** (DX)

This is the register which is most coupled with the accumulator. All the instructions which work with large-size data such as multiplication, division, and conversion instructions like `cwd` (convert word to doubleword), `cdq` (convert doubleword to quadword), and `cqo` (convert quadword to *octoword*, which is 128-bits data) store the most-significant bits in the data register, and the least-significant bits in the accumulator.
When programing, the EDX register can be seen as the 64-bits estension of EAX, or (similarly) RDX can be seen as the 128-bits extension of RAX.
DX plays an important role in instructions to interact with external devices as well, as we will see in Chapter 7. While AX keeps the data to be transferred to/from a device connected to the CPU, DX keeps the *address* of the device.
Many operations, on x86 architectures, can be carried out using only AX and DX registers.

**The Counter Register** (CX)
The CX register could be seen as the ubiquitous `i` (as *index*) variable of higher-level languages. All the instructions which *count* something, use this register. For example, all string instructions (like `movs` and `stos`) use CX to count the number of memory locations which must be moved or set to a specific value—these operations will be discussed thoroughly in Chapter 6. Differently from the `z64` CPU, real x86 CPUs allow this counting by placing *prefixes* before the instructions or block of instructions. These counting prefixes are `loop`, `loopz`, `loopnz`, `rep`, `repe`, and `repne`.
An additional instruction using CX is `jcxz`, which executes a *jump* to a different instruction only if the value of CX is zero.

**The Destination Register** (DI)
A common task when programming is to loop over a set of instructions which generate data and store them in memory. Usually, in higher-level languages, this can be done by using pointer, so that after the memory write operation the pointer must be updated. The destination register serves exactly this purpose, keeping the memory pointer. In particular, this register keeps the memory address used by all string instructions, which will be described in Chapter 6. As an example, an important instruction is `stos`, which copies data from the accumulator to the memory address stored in the destination register, a number of times specified by the counter register. This is, for example, a good way to implement straight in assembly the higher-level `memset()` function.
The value stored in DI is incremented or decremented automatically after each memory access, depending on the value stored in the *direction flag*.

**The Source Register** (SI)
The source register is the counterpart of DI. The only difference is that the pointer kept by this register is used to read memory, rather than to write to memory. There aren't many instructions which read large blocks of memory at once, so this register can be usually used as a temporary register to keep intermediate values of routines, being "more general" than other general-purpose registers.
Similarly to DI, SI is incremented or decremented automatically after each memory access, depending on the value of the *direction flag*.

**The Stack Pointer** (SP) **and the Base Pointer** (BP)
These are the only registers which are used 99% of the time according to their original meaning, because

they are at the heart of the function calls in x86 architectures. As we will see in Chapter 6, when calling a function, the *return address* of the calling function is stored on the parameters' stack. Once in the callee function, the base pointer is updated (pointing to the top of the stack), so that *local variables* can be easily located in memory.

**The Base Register** (BX)
This is the only "real" general-purpose register. It takes its name from the fact that early 16-bits x86 processors where limiting the usage of BX as the *base register* for addressing memory (see Section 5.6.1 for a discussion of the z64/x86 addressing modes). Therefore, an instruction like `movw %ax, (%bx)` was legal, while `movw %ax, (%dx)` was not. When Intel switched to 32-bits CPUs, this constraint was relaxed, so this rule is no longer in place. Therefore, its original specific meaning is no longer important, and it can be freely used a general-purpose register.

### Essential Registers

Any CPU implementation requires at least three registers which are essential to let the CU carry out the execution of any program, even under the assumption that no portion of the working memory is moved into the actual CPU (as the z64 does, by providing a set of 64 virtual registers internal to the CPU). We have already named some of them in this chapter, and they are the *instruction pointer* (IP), the *instruction register* (IR), and the *flags register* (FLAGS).

The *instruction pointer* register is a 64-bits register which keeps the memory address of the next instruction to be executed, allowing the CPU to fetch it easily from the working memory and to keep track of the evolution of the program. The common symbolic name for this register is RIP, conforming to the convention used to access other 64-bits registers. Nevertheless, it is not meaningful to consider subportions of it, because memory addresses in the z64 are always composed of 64 bits.

The programmer is not allowed to write directly into it, but its content is modified indirectly at every instruction's execution. Specifically, normal instructions increment the value of RIP by the size of the instruction itself, so that during the execution of any instruction, RIP always points to the forthcoming one. Additionally, two instructions can overwrite the content of RIP, namely `call` and `jmp`. The purpose of these two instructions is to explicitly alter the control flow of the program being run, by calling a subroutine or by jumping to a different portion of the code. Similarly to the `call` instruction which transfers the control flow to a different address, the `ret` instruction updated the value of RIP putting back the address of the instruction next to the `call` instruction which changed the flow of the program. The exact behaviour of these instructions will be analysed and discussed later in Section 6.9, yet they are the only means for a programmer to alter the content of RIP[4].

The *instruction register* is an internal buffer used by the CPU to store the instruction that is currently being executed. In particular, as we have mentioned at the beginning of this chapter, the execution of an instruction follows several different steps. At a certain point, the CPU loads the next instruction to be executed from memory, and make a temporary copy of it in IR. This is a 64-bits

[4] Another instruction which modifies the value of RIP is the `iret` instruction, which is used to return from the execution of a *device driver*, as it will be discussed in Chapter 7.

wide register, from which a large set of lines go to the CU and to several elements of the PU. These lines, which can be regarded as both *condition variables* and *control signals*, depending on their meaning, aim at informing the CU of the nature of the instruction to be processed, and to give additional information to other components of the PU.

The FLAGS register keeps track of *state variables* which record information related to the last-executed *arithmetic/logical instruction*. The way these state variables are used as *condition variables* by the CU heavily depends on the nature of the instructions which are executed, and the content of this register is so fundamental to the implementation of the CPU that it deserves a specific in-depth discussion, which will be carried out in Section 5.4

**Registers hidden to the programmer**

To speedup and to simplify the execution of some instructions, the z64 CPU relies on other registers which are *hidden* to the programmer, namely TEMP1 and TEMP2. These registers are used to keep the operands which will be used by the ALU or the Shifter. The presence of these registers, in our architecture, is mandatory. In fact, for example, the ALU is supposed to be able to compute the sum of two numbers. By our architecture, so far data are only stored in registers. Nevertheless, as we will see shortly, in order to reduce the complexity of the PU, a possible solution is to rely on one single interconnection network among all the components of the CPU. By using this solution, two operands from two different registers cannot be transferred to the ALU at the same time. Therefore, in order to allow it to read, at the same time, the two operands, we rely on *buffer registers* (namely, TEMP1 and TEMP2) where the operands are copied, one by one.

Two additional hidden registers, which we will use in the design of our CPU, are the *memory-address register* (MAR) and the *memory-data register* (MDR). These registers, similarly to TEMP1 and TEMP2, cannot be directly accessed by the programmer, but are used by the CU to communicate with the portion of the working memory which is external to the CPU. This will be dealt with in Section 5.6.

Overall, the block diagram of the z64 with the components described so far is presented in Figure 5.4.

### 5.1.1 Interconnection Structure

The typical set of operations which are performed on data stored into registers comprises:

1. Data transfer among different registers;
2. Data transfer to/from main memory from/to registers;
3. Logical/arithmetic operations.

Rotating and shift instructions are always coordinated by the CU, and are committed to the shifter, while the arithmetic/logic operations are committed to the ALU. Of course, these operations require to transfer data from the registers to the functional units, and the results back to the registers, an operation which is again coordinated by the CU.

To support data transfer among the different components of the CPU, an *interconnection structure* is necessary. Nevertheless, different interconnection organisations can be adopted in a CPU, some of

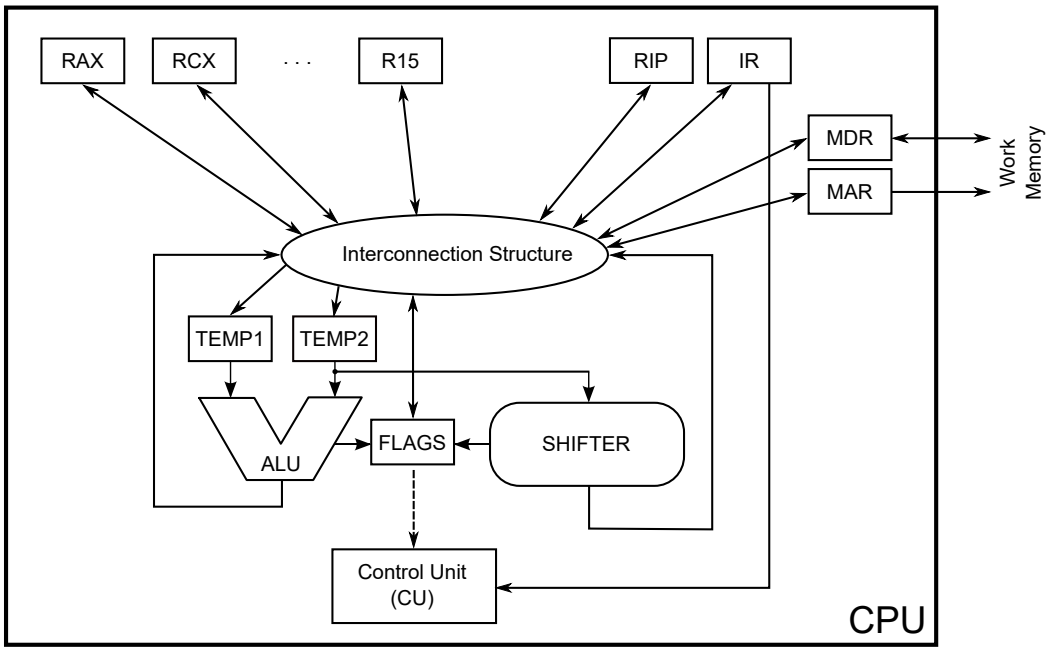



**Figure 5.4:** Block Diagram of the z64 Processing Unit.

which have been proven successful according to the technological evolution in the various families of processing units. In this Section we will introduce different implementations of the interconnection structure, and we will select the one which will be used to implement the z64 CPU. Additionally, considering that the processing unit is not the only component of a computing architecture, as described by the von Neumann architecture, we will show how it is possible to create an interface between this (internal) interconnection structure and the (external) main memory.

### Data Transfer Operations among Registers

Any data transfer which involves two registers has a *direction* which defines a *source* and a *destination* register. If we call $R_s$ (the source register) and $R_d$ (the destination register) any two distinct registers in the CPU (even hidden registers, such as TEMP1, TEMP2, or RIP), we denote the transfer of the content of the source register into the destination register as:

$$R_s \rightarrow R_d$$

This operation is coordinated by the CU. In order to support its execution, the CU enables the write of the content of the destination register and, at the same time, enables the read from the source register. At the end of this operation, the content of $R_d$ will be equal to that of $R_s$, while the previous content of $R_d$ is lost. To let this operation take place, in case the registers (as it has already been discussed in Section 3.3) are made of D flip-flops, the two registers can be connected as in Figure 5.5. In this case, data transfer is executed in one single step, namely the CU must only generate the control signal $W_d$.

As we have already shown, the z64 CPU hosts a large number of registers. Therefore, since any register can be considered as the source, and any register can be considered as the destination, we

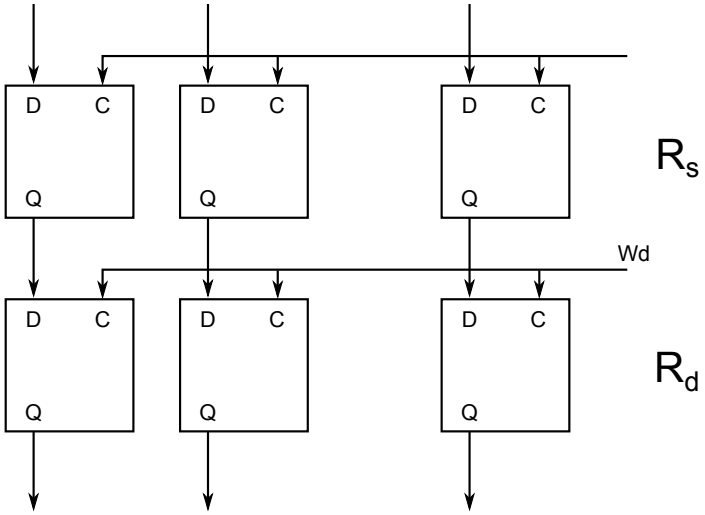



**Figure 5.5:** Interconnection between two registers

should provide for a connection between all of them. There are several interconnection structures which are suitable for this case. Let us consider the two extreme solutions, namely the ones that maximise and minimise the contemporary number of data transfers. If we have $n$ registers, in the former case we can have up to $\frac{n}{2}$ contemporary data transfers[5] (or $n - 1$ contemporary data transfers if we copy the same value from one register to all the other ones), while in the latter case we can have exactly *one*. The former solution requires a direct connection between any couple of registers, requiring as many multiplexers as the number of available registers $n$, and is depicted in Figure 5.6.

Control signals $\mathtt{m}_{11}, \ldots, \mathtt{m}_{1\log_2 n}, \mathtt{m}_{n1}, \ldots, \mathtt{m}_{n\log_2 n}$, and $\mathtt{W}_1, \mathtt{W}_2, \ldots, \mathtt{W}_n$ are generated by the CU and are used, respectively, to select source registers and enable writing on destination registers. Therefore, the number of inputs for each multiplexer is $\log_2 n + (n-1) \cdot b$, where $b$ is the number of bits in the registers, and therefore the complexity of the interconnection network is on the order of $b(n-1)$, which is non-negligible.

The opposite solution, namely the one which minimises the number of data transfers, minimises as well the complexity of the circuit. In fact, if we suppose that only one data transfer can be executed during one single execution step, we could devise one single transmission channel which is shared by all the registers, and is used only by the source and destination registers during one step. This solution can be implemented using one single set of transmission lines (namely, $b$ lines) to which each register is interfaced using a *three-state buffer*[6], as depicted in Figure 5.7. This structure is called *internal data bus*.

Control signals $\mathtt{B}_1, \ldots, \mathtt{B}_n$ enable the data transfer from every single register to the bus lines. Therefore, to avoid undesired electrical effects (such as short circuits), it is necessary to enable only one register at a time. Control signals $\mathtt{W}_1, \ldots, \mathtt{W}_n$, on the other hand, are used to enable writing the data being transferred on the bus to the destination register.

[5] In fact, it is pointless to provide for a connection to allow for a copy from one register to the same register.

[6] We recall that a three-state buffer is a component which allows an output port to assume a high impedance state in addition to the zero and one logic levels, effectively removing the output from the circuit. This allows multiple circuits to share the same output line or lines.

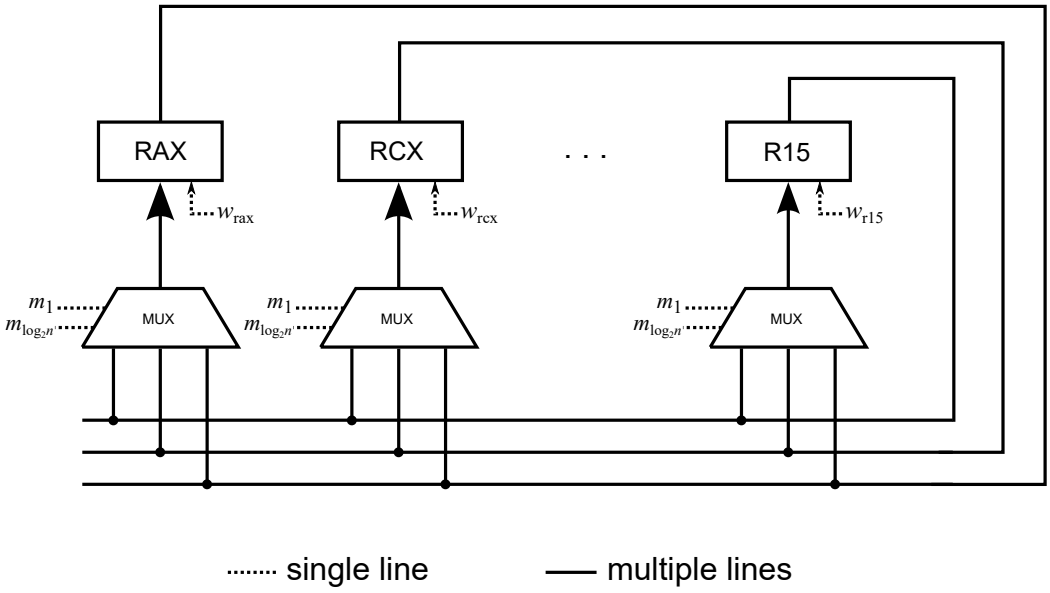



**Figure 5.6:** Interconnection structure using $n$ Multiplexers.

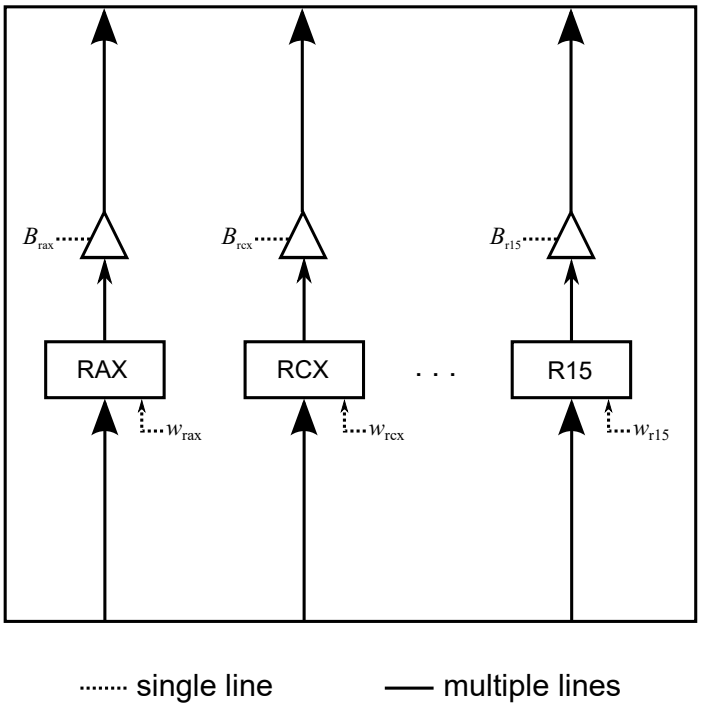



**Figure 5.7:** Bus interconnection.

Of course, in between these two extreme solutions, there is a plethora of intermediate interconnection topologies which allow to group different sets of registers together.

### 5.1.2 Data Transfer Operations to/from Computing Circuitry

Some arithmetic instructions (e.g., `add` or `sub`) and some logic instructions (e.g., `and` or `or`) have two source operands—therefore they are referred to *binary operations*[7]—while other instructions (e.g., `neg`, `not`, and all rotating instructions) have only one. Therefore, while the ALU must be able to process two data at once, the shifter requires an interconnection to one data source only—thus they are named *unary operations*. The result of both kind of operations must then be placed into a destination register.

Let us suppose that the CPU is equipped with a certain number of registers, which are operationally equivalent for the ALU and the shifter—namely, any operand of the computing circuitry can be stored into any register. Therefore, if we name any three registers $R_{S_1}$ (first source operand), $R_{S_2}$ (second source operand), and $R_D$ (destination operand), we can denote a generic binary operation `op` as:

$$R_{S_1} \texttt{ op } R_{S_2} \Rightarrow R_D.$$

The execution of this operation is managed by the CU, which enables reading data from source registers, instructs the computing circuitry to execute the required operation `op`, and enables writing on the destination register. On the other hand, in case of unary operations, we are only required to consider two generic registers, namely $R_S$ as the source register, and $R_D$ as the destination register. In this case, a generic unary operation `op` can be denoted as:

$$\texttt{op } R_S \Rightarrow R_D.$$

Similarly to the case of binary operations, the execution of unary ones is managed by the CU, which enables reading data from the only source register, instructs the computing circuitry to execute the required operations, and in the end enables writing the result into the destination register.

In general, any register could be connected to both the ALU and the shifter, and therefore it should be necessary to provide for a suitable interconnection structure. Among the various possible ones, we shall describe two completely opposite solutions, which nevertheless resemble what we have already seen concerning the interconnection among different registers in case of data movement instructions. In fact, the first solution minimizes the time required to execute operations, while the second one minimizes the number of required hardware components, having consistently a performance decrease.

The first solution is depicted in Figure 5.8. It requires a direct connection between each register and both computing elements (and vice versa), and requires as many multiplexers as the registers, so

[7] The term *binary* should not be confused with the representation used by computing architectures to store data. Rather, this kind of operations is named binary as it involves *two* operands (binary comes from late Latin *binarius*, meaning *consisting of two*).

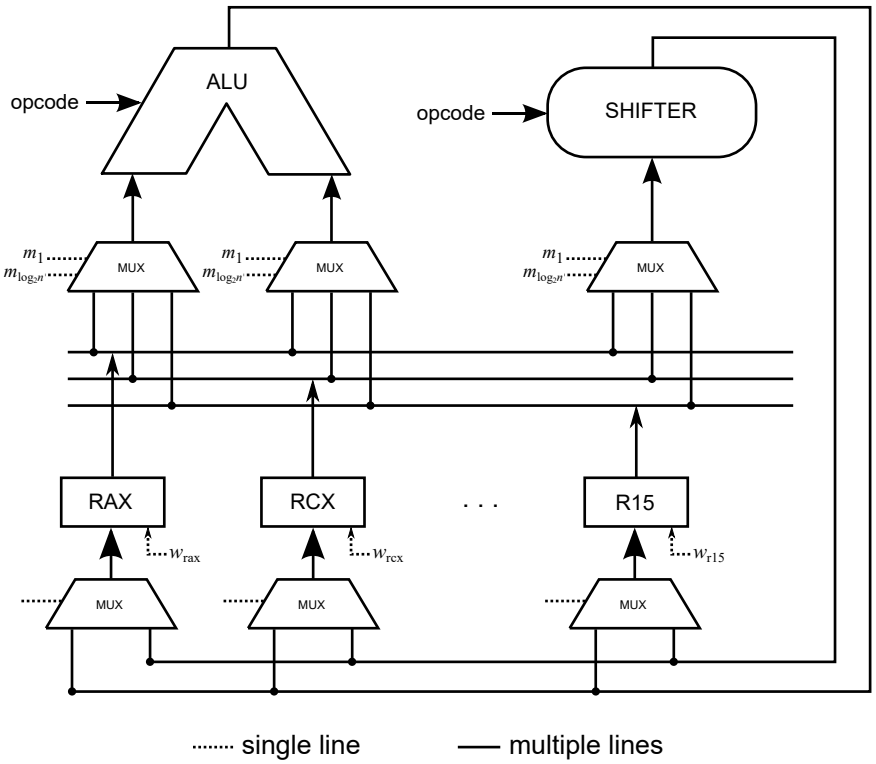



**Figure 5.8:** Interconnection structure between ALU, shifter, and registers, using Multiplexers.

as to connect the ALU's and the shifter's output with the inputs to the registers, plus three additional multiplexers to connect registers' outputs with the ALU's and the shifter's inputs. Therefore, in the case of this architectural solution, the execution of a binary operation is accomplished according to the following algorithmic steps[8]:

- The CU conveniently drives the input multiplexers to the ALU, so as to enable data tranfer from the actual $R_{S_1}$ and $R_{S_2}$;
- The CU instructs the ALU with proper control signals of what operation must be computed on the two input operands;
- The CU conveniently drivers the input multiplexers to the registers, so as to store the result of the computation into the proper $R_D$.

We emphasize that if the interconnection structure shown in Figure 5.8 is used, then the ALU and the shifter could execute operations in parallel, provided that the destination operand of the two operations is different. In fact, there is not one single direct interconnection among each register and the two computing elements. Furthermore, due to the high similarity with the interconnection structure shown in Figure 5.6, this one could be used as well to realize data transfer between registers. This could be done by having the shifter support a *no operation* feature, meaning that a shift of zero position is performed on input data, so as to present as output the same data which were received as input.

[8] We note however that the same operation can be carried out in one single step, if the destination is different from the source, or if registers' content is updated on the falling edge of the clock.

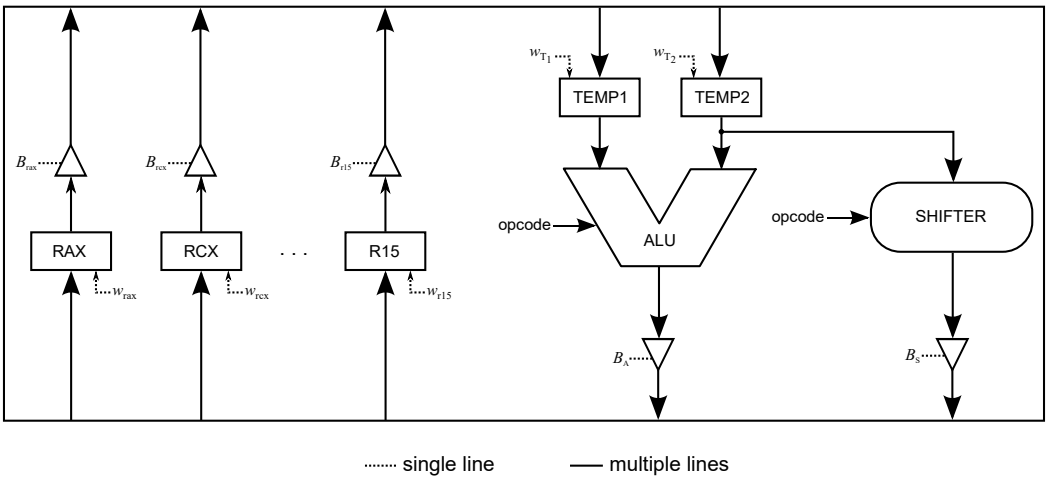



**Figure 5.9:** Interconnection structure using a bus.

The second solution, which is depicted in Figure 5.9, resembles the one shown in Figure 5.7 to interconnect all the registers for data transfer, using a single shared bus. Also in this case, due to the single transmissive resource—namely, bus lines—it is necessary to provide each element of the circuit with a three-state buffer, so as to avoid short circuits. Additionally, we note that both the ALU and the shifter, as implemented in Sections 3.5.4 and 3.5.3 respectively, are combinational networks. Therefore, in order to let them compute properly the result of a binary operation, the gates should stabilize having the values of both operands available at once. With the proposed interconnection, this is not possible. The same problem arises with unary operations, if we consider that the source register, at the same time, should be read (to take the input data) and written (to store the result)—this will surely lead the whole system to an inconsistent (possibly oscillating!) state.

Therefore, for this second solution to work properly, an additional couple of registers is required, namely `TEMP1` and `TEMP2`, which have already been shown in Figure 5.4. We note that, logically, three registers should be required: two for the input data to the ALU, and one to the shifter. Nevertheless, this second solution does not allow for parallel processing on top of both the ALU and the shifter, due to the lines which are shared to store the result. Thus, it is perfectly safe to condensate two registers into one—therefore `TEMP2` can be seen as the *second* input register to the ALU, or the *only* input register to the shifter. According to this strategy, executing a binary operations requires the following algorithmic steps:

- The CU enables writing on `TEMP1`, and at the same time enables the three-state buffer required to read the value stored into the first source operand;
- The CU enables writing on `TEMP2`, and at the same time enables the three-state buffer required to read the value stored into the second source operand (which can possibly be the same as the first operand, as no limitations are placed on the nature of source operands);
- The CU instructs the ALU with proper control signals of what operation must be computed on the two input operands (this time stored into `TEMP1` and `TEMP2`);

- The CU allows to store the result of the operation in the destination operand, by enabling the three-state buffer coming from the ALU, and enabling writing on the destination register.

In this case, differently from the previous solution, the temporal execution of micro-operations is required to be strictly controlled, as the CU must serialize the access on a unique shared resource (namely, the internal data bus). Additionally, it is interesting to note that, if on the one side we have reduced the required hardware (by eliminating a large number of multiplexers), on the other the time complexity to execute the same elementary binary operation is increased. The incautious reader might rebut that the multiplexers have been replaced by the same number of three-state buffers—so what's the deal here? We have the same amount of components, yet we have a more complex execution! We refer this reader to Section 3.5, in order to note that internally the number of gates required to realize the second solution is smaller. Additionally, the resulting combinational network is much simpler, thus requiring a smaller clock interval to stabilize the network, hence allowing for an increased frequency of the overall CPU.

If we compare this second solution with the one provided in Figure 5.7, we note that the interfaces to write and read to/from registers has not been changed. Therefore, the two schemes could be merged together, therefore having one single internal data bus which can be used both for moving data between different registers, and to execute operations onto their content.

## 5.2 Introducing the z64 Instruction Set

So far, we have only discussed how we can interconnect at the hardware level the basic components that we have introduced in Chapter 3, but we have not discussed how we can ask the CU to activate the specific ones that we need to carry on one operation.

To understand how a CPU can be actually programmed, we start introducing here the way in which instructions are *encoded*. When we write an assembly program, we use a language which is more similar to a *natural* one, where each operation is associated with a name (the assembly *instruction*), and all the required operands are expressed using some notation. While we will go much deeper into the details of this syntax in the following chapters, here we will introduce a couple of instructions which can be used to drive the execution of a tiny subset of the presented hardware. These instructions are: one which can be used to move data around between registers (a *move* instruction, represented by the mnemonic name `mov`) and one which asks the ALU to perform the addition of two integer numbers (represented by the mnemonic name `add`). These are both binary operations, but if we look back at the definition of a binary operation given in Section 5.1.2:

$$R_{S_1} \texttt{ op } R_{S_2} \rightarrow R_D.$$

we can see that three operands are actually needed. In the z64 assembly language, these three operands are reduced to only two, as the destination register $R_D$ and the second source register $R_{S_2}$ are always the same. Therefore, we can rewrite the generic syntax of the binary operation as:

$$\texttt{op } R_{S_1}, R_{S_2}$$

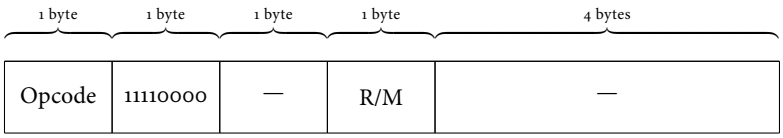



**Figure 5.10:** z64 Instruction Format for the `mov` and `add` instruction, with 64-bits register operands

which tells that the operation op uses the operands $R_{S_1}$ and $R_{S_2}$, and the result of the operation will be stored in $R_{S_2}$. In this sense, in the z64 notation, $R_{S_1}$ is called the *source* operand, and $R_{S_2}$ the *destination* operand.

Similarly, the same is true for unary operations. Therefore, the generic unary operation:

$$\texttt{op}\ R_S \rightarrow R_D.$$

in the z64 notation is rewritten (using the only *destination operand*) as:

$$\texttt{op}\ R_D.$$

When we write an assembly program, the notation which is used to write the instructions (the assembly language) cannot be interpreted by the CPU as is. In fact, the CPU can only decode a set of binary digits. Therefore, in order to feed our program into our CPU, we rely on the *assembler*, a program which "reads" our human-readable version of the program, and transforms it into a binary representation (the *machine code*) which can be decoded by the CPU. In particular, this machine-code representation will be used by the CU of the CPU to identify what are, at a given clock cycle, the control signals required to drive the hardware components to execute the instruction.

Therefore, there must be some sort of agreement between the assembler and the CPU on how the semantic information kept by the assembly instruction must be translated into a string of bits which can be correctly interpreted by the CU. This is exactly the *instruction format*, which defines what bits of each instruction represent a given piece of information of the instruction, be it the operands or the actual execution steps which must be carried out for it. The initial structure for the machine representation of our two instruction, `mov` and `add` using only 64-bits register operands, is the one presented in Figure 5.10.

All the instructions which we will present throughout this book can be grouped into *classes* of instructions, depending on the set of operations (namely, the control signals) that the CU will generate to carry on its execution. Our two first instructions, `mov` and `add`, belong to classes 1 and 2, respectively. This is reflected in the first byte of the instruction format, namely the *operation code* (Opcode). Generally speaking, the opcode can be regarded as a "directory index" for the CU, namely an information which tells the CU "where" to find, among its internal states, the ones associated with the execution of this specific instruction. As mentioned, the grouping of instructions brings to a two-level organization of this opcode, which is reflected in Figure 5.11: this byte is divided into the Class and Type sub-fields.

Since the `mov` instruction belongs to class 1, the Class field will be set to $(0001)_2$, while for the `add` instruction (which belongs to class 2), this same field will be set to $(0010)_2$. In each of these classes,

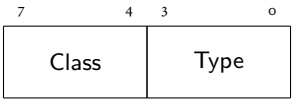



**Figure 5.11:** Opcode Byte Format

there are several instructions. The exact instruction within the class is identified by the Type field, which for both the `mov` and the `add` instruction is $(0000)_2$ (they are the first instruction within each class). Thus, the full opcodes for the `mov` and the `add` instruction are shown in Figure 5.12. Although there is only one bit difference in the binary representation of the opcodes for the two instructions, this is enough to correctly drive the selection of a different microprogram to carry on their execution.


(a) `add` opcode (a) `mov` opcode

**Figure 5.12:** Opcode Byte Format

The second byte of the binary representation is set for both instructions to $(11110000)_2$. This is called the *mode byte* (Mod), and we will explain the meaning of its bits later on in Section 5.5. For our discussion here, it is enough to say that this byte tells the CU that we are dealing with register operands, all of which are 64-bits wide. Similarly, the third byte is set to a *don't care* condition (represented by the — sign), which tells that its content is not considered at all for these two instruction (with these operands) by the CU, when interpreting the instruction. Therefore, the values of its bits can be set to either zero or one—the CU *doesn't care*! Commonly, in case don't care conditions are met by the assembler for the value of some bits, the assembler emits zero's for them. This is the same, in our case, for the last four bytes of the binary representation.

On the other hand, the fourth byte, which is called the *Register/Memory* byte (R/M) is essential for our `mov` and `add` instructions. In case of an operation which involves register operands, this 1-byte field is split into two 4-bits subportions, as shown in Figure 5.13. Each 4-bits subfield keeps one of the 16 binary representation of the registers, as shown previously in Table 5.1. The first sub-field is interpreted by the CU as the source register, while the second indicates the destination register, and the values stored in the fields can be efficiently used by the CU (we'll show shortly how) to activate the $B_i$ and/or $w_i$ signals, to allow reading from and writing to registers.

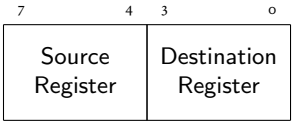
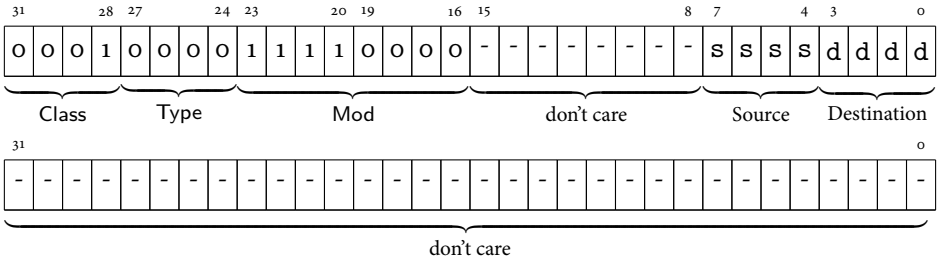


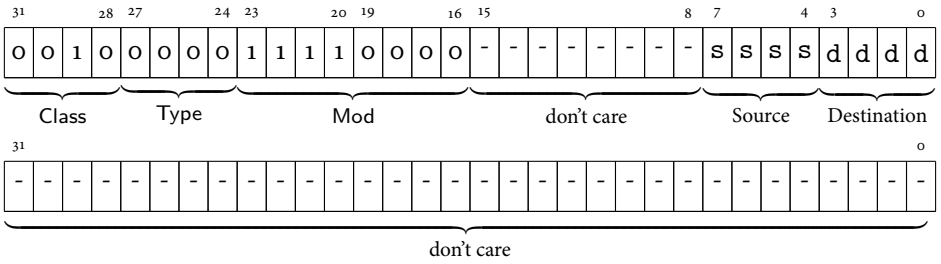

**Figure 5.13:** R/M Byte Format for `mov` and `add` instructions

If we then call `ssss` the four-digits encoding of the source register, and `dddd` the one of the destination register, the complete machine-code representation for the `mov` and `add` instructions which we are discussing are shown in Figure 5.14.

By the representation of these two instructions we can already draw some conclusions about the machine-code representation of the z64 instruction set. Several instructions (which are likely associated with a reduced number of $\mu$-ops for their execution) waste some bits. In fact, both the `mov` and




(a) `mov` instruction




(b) `add` instruction

**Figure 5.14:** Machine-code representation of the `mov` and `add` instructions

the `add` instruction have five don't care bytes. Nevertheless, as we will see throughout this book, this solution allows to speed up the decode phase of the execution—this will be even more the case when we will introduce *pipelining* in Chapter 13.

When writing a program, a common operation is to set a variable to a predefined value. In higher-level languages, this is a straightforward statement, which has a direct mapping to an assembly instruction (a `mov` instruction). Yet, four bytes are not enough to store, for example, a 64-bits immediate integer. Therefore, the machine-code representation of the z64 assembly language will be later extended to become of *variable size*[9]. This will give enough space to store data of any-size directly in the instruction's binary representation, making assignments like `x = 0` of simple execution for the CPU.

In any case, the two assembly instructions which we have so far introduced are enough to start showing how we can efficiently connect together the basic hardware components that have been already presented, and to show how the CU can rely on the various byte fields to drive their execution to carry out the involved operations.

## 5.3 Interconnection Structure of the z64 CPU

The z64 CPU Instruction Set allows to move only one piece of data at a time. Therefore, the bus interconnection which we have described in the previous Section looks more suitable, as the multiplexer-based one would be definitely underused, while the other allows to reduce the amount of required hardware and allows for an increased clock frequency of the overall CPU.

The global interconnection structure of the z64 CPU could be directly derived from the one de-

[9] The real x86 instruction set amplifies this to a higher extent, having instructions which can have a variable length starting from 1 byte, up to 16 bytes.

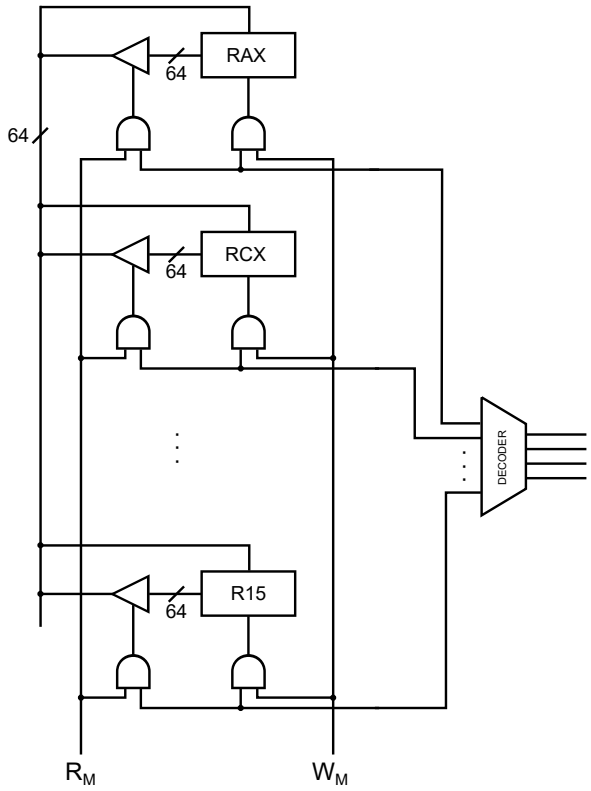



**Figure 5.15:** Registers organized as a small memory bank.

picted in Figure 5.9. Nevertheless, this organization can be further optimized, by organizing all registers in a *bank*, as depicted in Figure 5.15.

In this case, there is only one output line (obviously controlled by a three-state buffer) and one single input line to/from the internal data bus. In order to select one specific register (either in read or write mode) we rely on a decoder. To enable reading or writing the registers, two control signals ($W_M$ and $R_M$, to enable write or read mode respectively) are used.

With respect to the previous solution, namely the one depicted in Figure 5.9, we save several control signals which must be generated by the CU—and therefore, the corresponding lines in the CPU. In fact, we reduce from $2n$ control signals ($n$ for reading and $n$ for writing) to $2 + \log_2 n$ signals (one for reading, one for writing, and $\log_2 n$ to select one specific register). Nevertheless, as we shall see, one single instruction representation might carry the encoding of more than one register (e.g., the first source operand, the second source operand, and the destination operand). Therefore, in order to use the organization presented in Figure 5.15, in which only one decoder is provided, we must select beforehand one of the possible codes identifying a register from IR. This can be done by placing a multiplexer which, operated by the CU, allows to let one specific register identifier to pass through the lines, towards the decoder.

Additionally, some instructions do not explicitly provide the source/destination register. This is specifically for a set of instructions which always operate on the same registers, e.g. `push`, `pop`, `in`, and `out`. In this situation, by decoding the opcode of the instruction, the CU is able to identify any involved register. Therefore, in case of a `push` or `pop` instruction, the CU must generate control signals

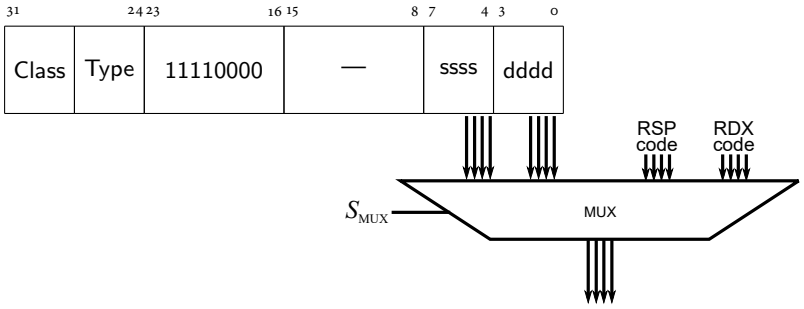



**Figure 5.16:** Circuit to select source/destination operands.

which select the RSP register. On the other hand, in the case of instructions such as `in` and `out`, the CU must generate control signals which select the DX register[10], in addition to the regular source and destination registers.

Our architectural solution so far uses a multiplexer to select one among the two possible register encodings which are stored into IR, namely the source and destination registers. Therefore, we can simply augment this logic, relying on a 4-input multiplexer, as depicted in Figure 5.16. The additional 2 sources will be the encodings of RDX and RSP, hard-coded into the circuit. The CU generates control signals to drive the multiplexer so as to select the encoding needed by the current instruction.

We note, however, that using only one multiplexer has a drawback. In fact, in this way, we can select only one register encoding from IR at a time. Therefore, data movement instructions which involve a register as both the source and the destination operands cannot be immediately mapped to CPU microoperations. To solve this issue, we must rely on a temporary register. If we use `TEMP2` as the temporary register for this data movement operation, the CU of the CPU will be able to instruct the shifter to perform a *zero-bit* shift, leaving unaffected the input data, which is copied verbatim on the output port of the shifter—and therefore the input data (from the temporary register) can be stored into the destination register.

By this discussion, we can define the PU architecture for the `z64` CPU as depicted in Figure 5.17.

## 5.4 The FLAGS register

The special FLAGS register contains several bits which either record special attributes of instructions' execution, or make the system operate according to different execution modes. This difference in the meaning of the bits in the FLAGS register places them in the category of either *status* or *control* bits. Similarly to the RIP case, the user can read the value of this register, but he cannot modify it directly as a whole, by for example storing some data into it[11]. However, depending on the nature of the flag bits (namely, control and status bits), there is the possibility to alter their status by using various instructions. Concerning the status bits, they are indirectly altered by the outcome of any arithmetic

[10] In the final architecture which we are designing, DX is a 16 bits register. For simplicity, we are now assuming that all registers are 64-bits wide. Therefore, our design will actually generate control signals to select RDX. When the reader will be familiar with the `z64` architecture, we will show how to select the 16 least-significant bits of the `D` register, thus DX.

[11] There is one instruction which allows to actually directly modify the value of the FLAG register, namely `popf` (see Appendix A). This instruction actually stores into FLAGS the content of the top stack element, requiring the programmer to push on the stack a suitable value before using it.

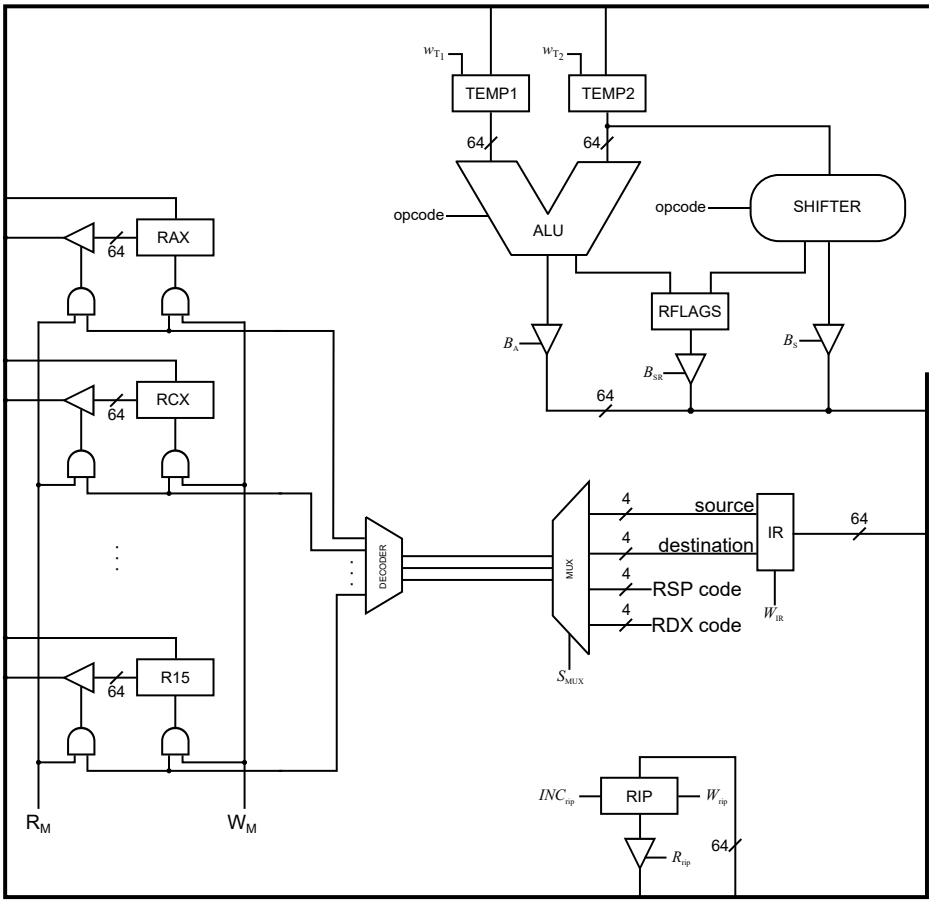



**Figure 5.17:** PU Architecture of the z64 CPU, without interfaces to external memory

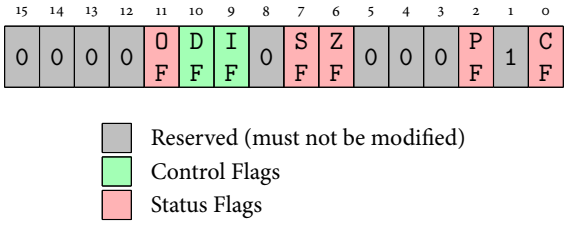



**Figure 5.18:** `flag` register

**Table 5.2:** FLAGS register bits and their meaning.

| Bit Number | Abbreviation | Description | Category |
|---|---|---|---|
| 0 | CF | Carry Flag | Status |
| 1 | 1 | Reserved | |
| 2 | PF | Parity Flag | Status |
| 3 | -- | Reserved | |
| 4 | -- | Reserved | |
| 5 | -- | Reserved | |
| 6 | ZF | Zero Flag | Status |
| 7 | SF | Sign Flag | Status |
| 8 | -- | Reserved | |
| 9 | IF | Interrupt Enable Flag | Control |
| 10 | DF | Direction Flag | Control |
| 11 | OF | Overflow Flag | Status |
| 12 | -- | Reserved | |
| 13 | -- | Reserved | |
| 14 | -- | Reserved | |
| 15 | -- | Reserved | |

instruction, to keep track of the outcome of the last-executed operations. On the other hand, they can be explicitly modified by *bit-manipulation instructions*, which allow to set/clear their values. These are the `setX`/`clrX` instructions. In Figure 5.18 and in Table 5.2 we report what is the meaning of each bit of the FLAGS registers. In general, when a flag contains 1, it is referred to as being *set*. On the other hand, when it contains zero, it is referred to as being *cleared*.

Conforming to the naming conventions of the `z64` registers, the FLAGS register is 16-bits wide, and it is therefore meaningful to talk of the EFLAGS and RFLAGS registers (while an 8-bits flags register is meaningless, due to the fact that fundamental bits are present in the 16-bits version). The remaining bits of RFLAGS are thus short-circuited to zero[12].

The CPU updates singularly the value of each control flag depending on the result of logical/arithmetic operations. As we have seen in Section 3.5, the processor relies on a couple of circuits called Arithmetic/Logical Unit (ALU) and Shifter to carry out this kind of operations, and the result of any operation is feed into a combinational network to update the value of the status bits.

The *carry flag* (`CF`) keeps track of the fact that, when computing an arithmetic operation, the last full adder outputs a carry bit—but it is then used to store information related to other non-arithmetic operations. In case of an arithmetic operation, to update `CF` the following rule is enforced: *The carry*

[12] Except for bit 1, which is always set by design.

*flag is set if the sum of two numbers causes a carry out of the most significant (leftmost) bits added*. If we consider two unsigned integers represented using 4 bits, for example 15 and 1, their sum is:

```
1111 + 0001 = 0000
```

Of course, the result of this operation should be 16, but there is no possibility to represent this number using only 4 bits. Nevertheless, the sum operation would produce an additional carry bit, which is used to update the value of `CF`. Therefore, this bit is interesting when dealing with unsigned integers, to check for an overflow.

When dealing with subtraction, the situation is different. The `z64` ALU performs subtractions by computing the sum between the second operand and the two's complement of the first operand. On some architectures, along with the carry flag the *borrow flag* is present, which is set whenever the result of a subtraction does not fit into the unsigned bit range. The `z64` CPU does not have a borrow flag, but the same result can be obtained by reasoning on the value of `CF`. To make an example, let us consider the operation 0-1. They are both non-negative values, and therefore they perfectly fit in the range of unsigned bit range. This subtraction is computed by the ALU as:

```
0000 - 0001 = 0000 + 1111 = 1111
```

Here, the situation is such that the result is -1, which in unsigned arithmetic would be considered as the value 15. Nevertheless, this operation could be part of a larger integer subtraction, which would take this as just a part of the final result (see page 229 for a discussion and an example of this). Therefore, we are actually interested in the value of the non-present borrow flag, which is nevertheless computed as the complement of `CF`, meaning that if `CF` is 0, the value of the borrow is 1 and vice versa.

In fact, in the given example we see that (not considering the unsigned representation of the data) we actually need a borrow bit to perform the original subtraction. If, on the other hand, we want to compute 1-1, the ALU computes:

```
0001 - 0001 = 0001 + 1111 = 0000
```

which actually sets `CF`, telling us that the (hypothetical) value of the borrow flag is zero: no borrow bit is needed.

`PF`, the *parity flag*, indicates if the number of set bits in the least significant byte of the result (depending on the size of the operands) is odd or even. This means that for 16-, 32-, and 64-bits values, only the least significant bits are used to compute the result. `z64`'s `PF` is an even parity flag, so it is set if the number of ones found in the involved byte is even. The value to which `PF` is set is computed via an XOR sum of the bits, yielding zero for even parity and 1 for odd parity. Therefore, this result is then negated and stored into `PF`. This is particularly useful, for example, to check for a transmission error or to compute CRC (see Section 2.5).

The *zero flag* `ZF` is set if all the bits of the results (depending on the size of the operands) are zero. This is simply computed by OR'ing all the bits of the result, and then negating the outcome of the operation. In fact, if any of the bits in the result is not zero, the result of the OR operation yields one, which when negated yields zero. This status bit is used to check whether the result of an operation is

exactly zero. As we will show in Section 6.3.2, this flag can be used as well to determine whether two operands store the same value.

SF, the *sign flag*, stores the sign of the result of the operation. Of course, this status flag is meaningless if the operands are representing unsigned integers, while in case they are signed its value simply stores the most significant bit of the result. Note that since the CPU has no clue about the signedness of the operands, this bit is always set/cleared upon the completion of any operation (even logical one), so it is the responsibility of the programmer to check it only when its content is meaningful, according to the implemented program.

The (signed) *overflow flag* OF tells whether the result from a signed operation overflows. It is important to note that OF should not be confused with CF, as they represent the same outcome but starting from different conditions, namely if the operands are signed or unsigned. OF is only meaningful when using signed operands. The rules for setting/clearing it are the following two:

1. *If the sum of two numbers with the sign bits off yields a result number with the sign bit on, then* OF *is set*. Let us consider two 4-bits numbers expressed in two's complement notation, namely 4 and 5. Their sum gives the following result:

   ```
   0100 + 0101 = 1001
   ```

   Note that the result of the sum is not 9, since this number cannot be represented in two's complement notation using only four bits. In fact, this number should be interpreted as -7. Since the initial operands were positive, but the result is negative, we have an overflow, which is correctly captured by OF.

2. *If the sum of two numbers with the sign bits on yields a result number with the sign bit off, then* OF *is cleared*. Let us consider again two 4-bits number expressed in two's complement notation, namely -7 and -8. Their addition yields:

   ```
   1001 + 1000 = 0001
   ```

   Similarly to the previous example, the result here is positive (it is 1), and therefore we are actually facing an overflow, which is correctly captured by OF.

We want to stress that in case of signed arithmetic, detecting an overflow by checking CF yields to wrong results. In fact, the example given to illustrate rule 2 sets CF, while the example for rule 1 does not, although we are facing overflows in both scenarios.

A small logic gate array checking for the sign bits of the operands and the sign bit of the result can be easily used to update the value of OF. But there is a different way to compute the value of OF, which is the one adopted by the z64 CPU. Let us consider for the sake of simplicity numbers represented in two's complement notation using only 2 bits. We can then represent only -2, -1, 0, and 1. This method takes into account the binary carry coming into the leftmost place and the binary carry going out of it. We call them *last-bit carry in* and *last-bit carry out*, which are the carry bits coming into the last adder of the ALU and the carry bit coming from the last adder of the ALU (the last-bit carry out is actually what later becomes CF). An overflow in two's complement occurs when the carry bit coming *into* the most significant bit does not match the carry bit coming *out of* the most significant bit.

**Table 5.3:** Conditions to set OF

| | Operation | Carry In | Carry Out | Overflow? |
|---|---|---|---|---|
| 1 | 11 + 01 = 00 | 1 | 1 | ✗ |
| 2 | 01 + 01 = 10 | 1 | 0 | ✓ |
| 3 | 11 + 10 = 01 | 0 | 1 | ✓ |
| 4 | 10 + 01 = 11 | 0 | 0 | ✗ |

Table 5.3 reports results for four cases, when adding different 2-bits numbers. We can easily verify that an overflow occurs only when the two carry bits are different: checking for this condition only requires computing the result of the XOR operation among them. In fact, the z64 CPU relies on this rule to set OF because it is much simpler from a circuitry point of view. In fact, only a XOR gate is enough to update OF, while using the previous rules requires checking three bits with a more-complex logic.

Status bits are updated as well by shift operations, and the circuitry to update them must take into account as well this, but in a slightly different way. In section 3.5.3, we have seen that the shifter's opcode is divided into two different parts: on the one hand, we have the actual operation (arithmetical shift, logical shift, or rotation) and the direction, on the other we have the number of positions the shifting operation moves the bits around. In the z64 architecture (you can refer to Section A.4 for a complete description of this class of instructions), this information is either encoded in bits 0–6 of the instruction (namely, the 7 least-significant bits of the Displacement field of the machine representation), or in the CX register, depending on the opcode. The different source is selected by using a multiplexer, which is driven by the 1-bit $S_{shpos}$ control signal, issued by the CU.

All Shifter-related operations can be gathered into two different groups of operations: *shift* instructions, and *rotating* instructions. From an encoding point of view, the difference between these instructions (as is clearly shown in Section A.4) lies in the most significant bit of the Type field: shift instructions have this bit cleared, while rotating instructions have this bit set. This is quite useful when dealing with the update of the FLAGS register. In fact, the status flags are updated according to these different rules:

- *Shift instructions* update CF so that is contains the value of the last bit shifted out of the operand, as shown previously in Figure 3.1. This is true both when shifting to the right, and when shifting to the left. SF, ZF, and PF are set according to the result, exactly as in arithmetic/logical operations. The value of OF depends on the direction of the shift. When moving bits to the left, OF is set to zero if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same). Note that a 1-position left shift is equal to multiplying by 2, so OF in this way is correctly set to the signed overflow. If shifting more than one position, OF is again computed according to this algorithm, but the value is no longer representative of a signed overflow, thus we say that OF is *undefined* in this case. When dealing with right-direction shifts, we differentiate the value of OF again according to the type of operations. In case of a *logical* shift, OF is set to the most-significant bit of the original operand (i.e., this instruction sets the overflow flag if the sign changes). On the other hand, in case of

an *arithmetical* shift, OF is always cleared, because unsigned overflow can never happen when dividing by two a signed number. Again, the value of OF cannot be considered reliable if a shift operation moves bits by more than one position.

- *Rotating instructions* are easier, as they shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions recirculate through the result. These instructions do not affect SF (this is not an arithmetic operation, so it is meaningless to say that the result is negative), ZF (if the operand were not zero, it is not zero even if we reshuffle the bits), and PF (the bits around the operand, again, are the same). On the other hand, OF is changed if the sign of the operand changes. Similarly to the previous case, CF is updated with the bit which is shifted into it, again according to what is depicted in Figure 3.1.

All these cases can be easily managed by a combinational network, which computes all the updates of the flags in the possible cases, and then selects the correct one depending on the actual opcode. First of all, the FLAGS register must be updated according to the discussed rules for arithmetic and logic operations *only* if the result is coming from the ALU. This is so only if the instruction which the CPU is executing is an arithmetical/logical one. While the full instruction encoding for this class of instructions is discussed in Section A.3, we highlight here that their class is 2. On the other hand, the shift and rotation instructions belong to class 3. The CU emits the $W_{flags}$ command only for instructions belonging to class 2 and 3. Therefore, it is sufficient to read from IR bit 28, which is bit 0 of the opcode class. This bit is set to 0 for class 2 instructions, and to 1 for class 3 instruction. This is enough to drive (using an additional NOT gate) a couple of three-state buffers, to select whether the updated bits are coming from a result generated by the ALU or by the shifter.

As mentioned, the z64 CPU supports an additional class of instructions which allows to directly manipulate all status flags. These instructions, belonging to Class 4, are the *flag bit manipulation instructions*. These instructions are `clc` and `stc` (which clear and set, respectively, CF), `clp` and `stp` (which clear and set, respectively, PF), `clz` and `stz` (which clear and set, respectively, ZF), `cls` and `sts` (which clear and set, respectively, SF), `clo` and `sto` (which clear and set, respectively, OF). In this class, we have two additional instructions (`cli`/`sti`, and `cld`/`std`) which allow to manipulate as well the DF and IF control flags, to change the operating mode of the CPU. The differences between these operating modes will be discussed later in this book, when we will see how to implement complex operations in assembly, and how to interact with the external world via devices.

Implementing the PU part to support the execution of class-4 instructions is quite easy, given their machine encoding. While the whole class is described in Section A.5, for the sake of simplicity let's see how the `clc`/`stc` pair of instructions is encoded. Again, all these instructions are 64-bits long, but all the bits, except for the Opcode field, are don't care conditions.


(a) `clc` opcode (a) `stc` opcode

**Figure 5.19:** `clc` and `stc` Opcode bytes

The Opcode bytes of the two instructions are shown in Figure 5.19. The Class field is of course set

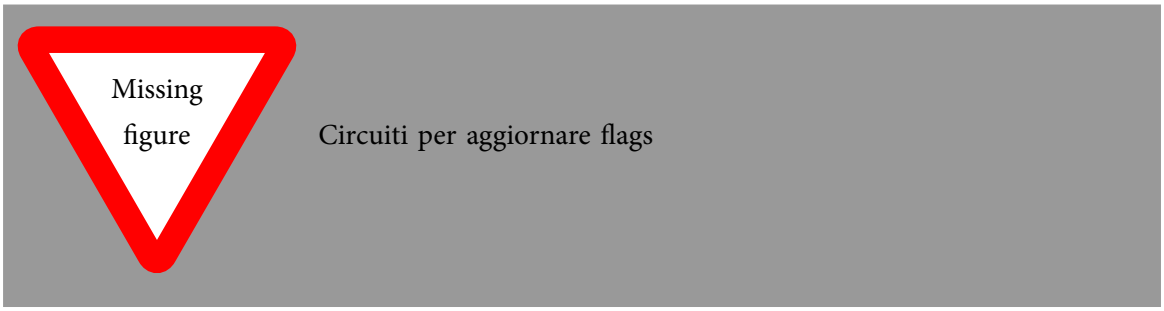



**Figure 5.20:** Combinatorial Network used to update FLAGS.

**Table 5.4:** z64 instruction suffixes

| Suffix | Operand Type | Bit Width |
|---|---|---|
| `b` | Byte | 8 bit |
| `w` | Word | 16 bit |
| `l` | Long | 32 bit |
| `q` | Quad-word | 64 bit |

to $(4)_2$, while the Type field is organized in this way: the first bit is set to zero for clear operations, while it is set to one for set operations. The latter three bits of Type are used to identify what bit in FLAGS is affected by the operation. This can be very easily mapped to a combinational network which updates the values of FLAGS.

From all the above description, we can illustrate the circuitry to update the FLAGS register and drive the execution of ALU- and Shifter-related instructions as in Figure 5.20.

## 5.5 Extending the PU to deal with variable-size data

The z64 instruction set allows to directly manipulate operands which are 8, 16, 32, and 64 bits long. Therefore, the register bank must allow to read/write *subportions* of their whole content (which, as we have seen in Section 5.1, is 64-bits long).

In order for the assembler to know how to properly set the size field of the instruction, a *suffix* should be added at the end of the instruction's mnemonic. For example, considering the `add` instruction, if we want to add the content of the `%eax` register to `%ebx`, explicitly specifying that we are interested in executing a 32 bit add operation, we will write:

```
addl %eax, %ebx
```

On the other hand, if we want to perform a 16-bit add operation between the `%ax` and the `%bx` registers, we will write:

```
addw %ax, %bx
```

In Table 5.4 the four suffixes associated with 1-byte, 2-bytes, 4-bytes, and 8-bytes operations are shown. If the suffix is not specified, and there are no memory operands for the instruction, the

operand size is inferred by the assembler from the size of the *destination* register operand, so that, for example, the following two instructions are perfectly equivalent:

```
addl %eax, %ebx
add %eax, %ebx
```

In Figure 5.10, we had hard-coded the value `11110000` into the second byte of the machine-code representation of the instruction, simply telling that this information is used by the CU to drive the PU so that the instructions are all executed using 64-bits operands. In fact, the second byte of the machine-code representation is called the Mode byte, and is used to specify the size of the source and destination operands in the first four bits. Up to now, we are only interested in these four bits, which are further divided into two 2-bits fields, as shown in Figure 5.21. For the sake of simplicity, we can set the remaining four bits to don't care conditions.

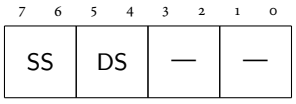



**Figure 5.21:** Mode Byte fields which specify the size of the operands

The first sub-fields of the Mode byte are called SS and DS, which stand for *Source Size* and *Destination Size*, respectively. These fields can keep four values each, which are interpreted as:

- `00`: the operand is 8-bits wide;
- `01`: the operand is 16-bits wide;
- `10`: the operand is 32-bits wide;
- `11`: the operand is 64-bits wide;

In fact, in Figure 5.10 the value `1111` was exactly telling that both the source and the destination operands were to be interpreted as 64-bits operands. Therefore, to fully identify the register operand involved in an operation, both in terms of register name and data size, the CU needs to extract from IR 4 bits which identify the register, and 2 bits to identify the size of the involved data. If the CU wants to extract the information associated with the source operand, the bits describing the register are bits 4–7 in `ir` and the bits describing the size are bits 22–23. On the other hand, to extract the information associated with the destination operand, the register is identified by bits 0–3 of IR, and the size is identified by bits 20–21.

Therefore, in order to select one of the overall 64 registers—we consider here to be two virtually-different registers, for example, RAX and AX— it is possible to use two decoders, as shown in Figure 5.22, one using four inputs (to select one of the sixteen 64-bits register), the other using two inputs (to select one of the four formats). The inputs to these decoders come directly from IR.

Let us consider as an example the RAX register only. This register is divided, as shown in Figure 5.23, into 4 subportions:

- The first subportion (AL) is composed of 8 flip-flops, and is connected to bits $\texttt{IB}_{0,7}$ of the Internal Data Bus IB;

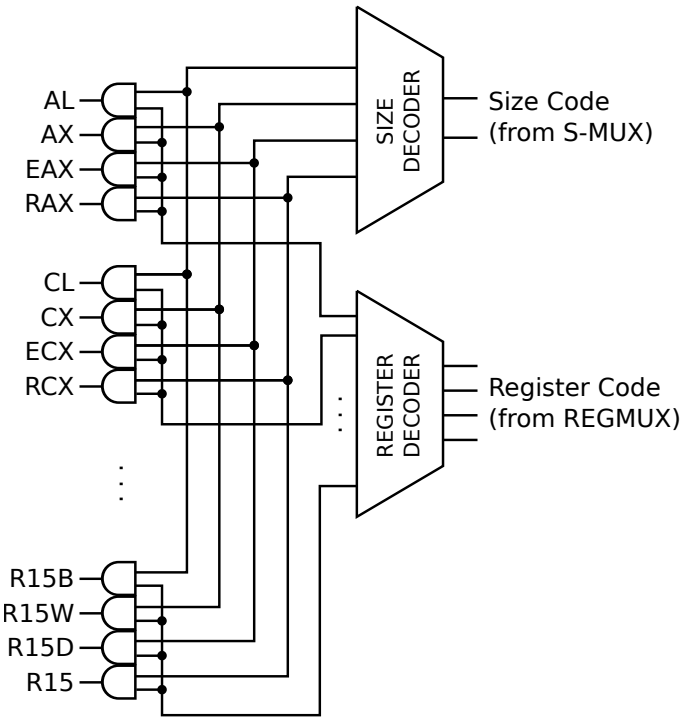



**Figure 5.22:** Using two decoders to select a subportion of a register.

- The second subportion (AX) is composed of 16 flip-flops. The first 8 flip/flops are the same as those of AL, and represent the 8 least-significant bits of AX. The remaining 8 flip/flops are connected to bits $\texttt{IB}_{8,15}$ of the Internal Data Bus IB;

- The third subportion (EAX) is composed of 32 flip-flops. The first 16 are the same as those of AX, and represent the 16 least-significant bits of EAX. The remaining 16 flip-flops are connected to bits $\texttt{IB}_{16,31}$ of the Internal Data Bus IB.

- The fourth subportion corresponds to the whole RAX register. Similarly to the other cases, it is composed of 64 flip-flops, among which the first 32 are the same as those of RAX, and the remaining 32 are directly connected to bits $\texttt{IB}_{32,63}$ of the Internal Data Bus IB.

The connection between the flip-flops and the Internal Bus is used for both reading and writing. When writing on the bus (thus, when reading from the register), the outputs of the flip-flops are driven by three-state buffers, again to avoid short circuits. On the other hand, when reading from the bus (thus, when writing on the register) the interconnection is direct. In fact, as mentioned in Section 3.3, we rely on Latch-D flip-flops, whose write enabling is driven by flip-flop control signals generated by the CU.

The different subportions of the RAX register are used in different cases, depending on the actual size of the data movement operation. In particular, the 8 flip-flops of AL are always used, independently of the size of the data movement operation. On the other hand, the most-significant flip-flops of AX are only involved in data movement operations when the sizes word, longword, or quadword are specified. Similarly, the most 16-significant flip-flops of EAX are involved only when the specified

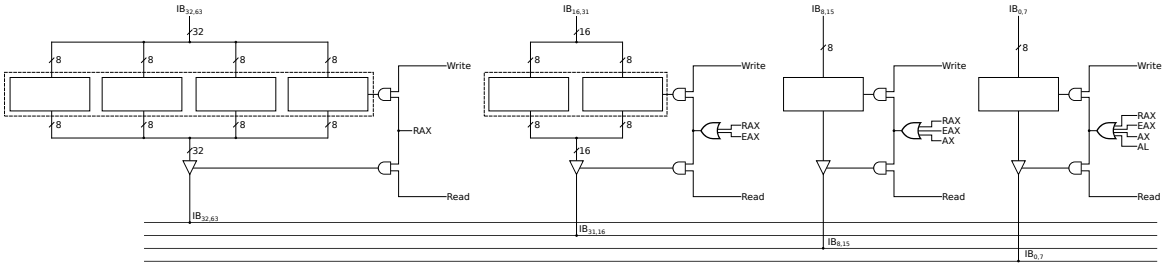



**Figure 5.23:** Combinational logic to select a subportion of RAX.

size is longword or quadword, and the 32 most-significat flip-flops of RAX are accessed only when the data-access size is a quadword.

Therefore, to select a subset of the available flip-flops, we can rely again on combinational logic, using OR gates whose outputs pass through AND gates, which receive as well control signals $\mathtt{R_M}$ and $\mathtt{W_M}$, as shown in Figure 5.23.

Extending this behaviour to all the other registers is straightforward, and selecting mode access for any other register could be carried out in the same way. This leads us to the definition of the overall final PU architecture of the z64 CPU as in Figure 5.24.

## 5.6 Accessing Memory

Since the z64 CPU is designed according to the von Neumann architecture, the CPU by itself becomes useless, as it has to constantly interact with main memory. In fact, even to simply execute an instruction, its first execution phase (the fetch phase) entails accessing memory to retrieve the machine code associated with it. Therefore, accessing main memory is the first operation which must be faced, and is actually the first interaction with the world outside the CPU that we will see.

While we will enter the details of the memory organization in Chapter 11, for the sake of simplicity we can assume now that it is logically organized as a simple linear vector of cells, where each cell is one byte. Each cell is associated with a number, and the numbers are assigned to cells in an increasing order, starting from zero. This number is the *address* of the cell, namely a unique identifier which can be used by the programmer to identify where some certain data are stored, and by the CPU to ask the memory chipset to transfer the content of the cell(s) to some register internal to the CPU itself. Therefore, since the address is encoded as a binary integer of $n$ bits, the CPU (and therefore the programmer) is able to access all the cells starting from address zero up to $2^{n-1}$.

Main memory keeps both the data and the program's instructions, although for simplicity they are usually located in different zones of memory[13]. Due to the fact that during the fetch phase of an instruction the value of the program counter RIP is incremented by 8, all the instructions are usually stored contiguously in memory. If, for some reason, the programmer decides to create some "empty space" between blocks of instruction, specific instructions to jump from one block to the other should be used, otherwise the CPU will not be able to carry on the execution of the program, once the empty space is reached.

[13] Except for the case of *immediate data*, which we will discuss later, which are stored directly in the machine-code representation of the instructions accessing them.

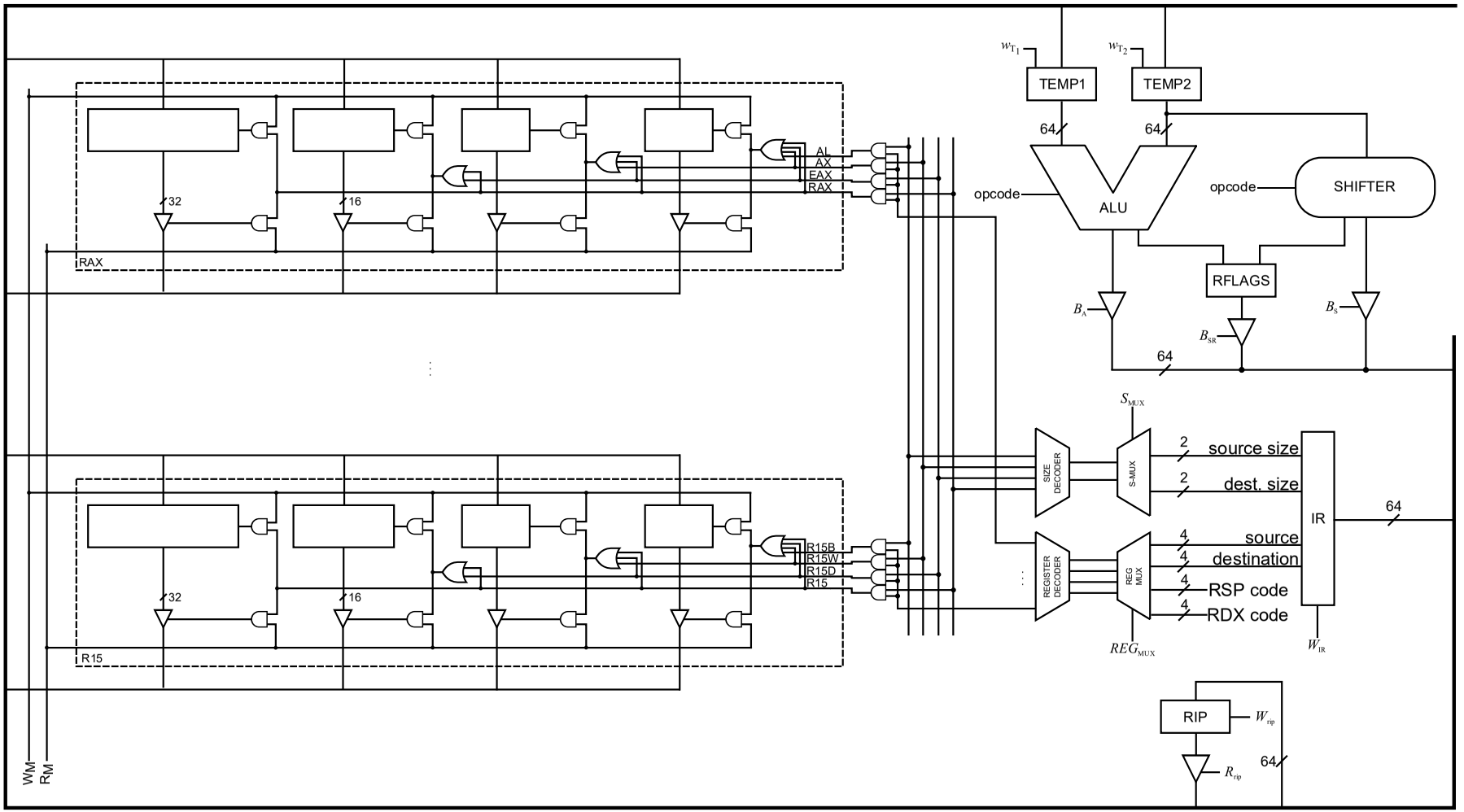



**Figure 5.24:** Final PU architecture, allowing to execute data movements of all the supported sizes.

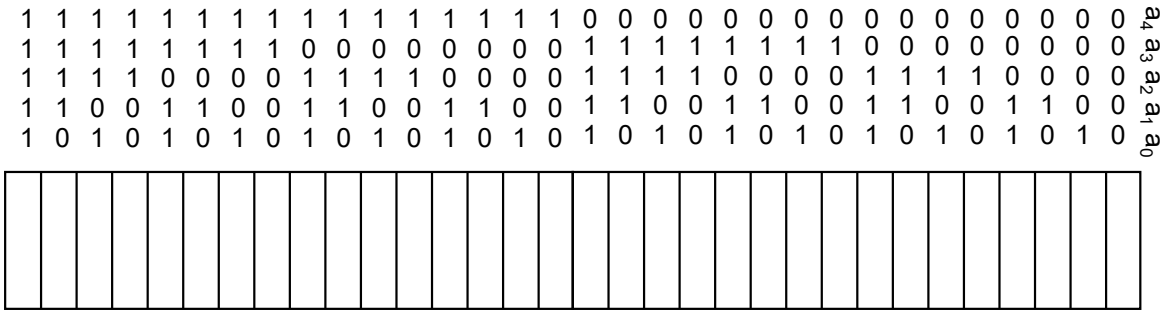



**Figure 5.25:** Logical organization as a vector of 16 cells of memory.

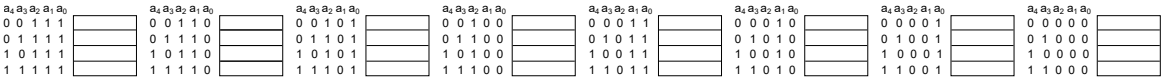



**Figure 5.26:** Memory organized into 16 blocks of cells.

As we have already introduced, all the instructions are 8-bytes long, while the data can be 1, 2, 4, or 8-bytes long (respectively a *byte*, a *word*, a *longword*[14], and a *quadword*). Each cell can be addressed using any of these four sizes (similarly to what we have seen when accessing subportions of the registers), so that in our linear vector of memory we can find a word followed by a byte, then three longwords, and then a quadword, each one representing different data which are of interest to our program. This means that instructions and data can be located at any position (namely, at any address) in memory.

Current technology does not allow to realize 64-bits memory modules, as commonly the market sells 8-bits modules. Even if in the future technology will allow for the construction of 64-bits modules, it is likely that at that time the internal CPU registers will be much wider than 64 bits. This has always been the case, and memory banks have been always built using less wide modules than the size of CPU registers.

Let's then take 8-bits modules, and let's try to organize them to let the programmer access data which are 8, 16, 32, and 64-bits wide. To do so, our memory architecture must be able to select, at the same time, 1, 2, 4 or 8 byte modules—this is called an *eight way parallel* memory. Figure 5.25 shows how we can logically organize as a vector 32 1-byte wide memory cells, while Figure 5.26 shows how we can organize the same amount of memory in having 8 different memory modules. In both figures, the addresses of each 1-byte memory cell are shown.

By Figure 5.26, we can see that all the memory cells within the same block share the least-significant bit. For example, the first module has memory cells identified by addesses of the form `--000`, which means that there is a *module address*, represented by the three least-significant bits of the address. At the same time, we can see that the $i$-th cell in every module share the first two bits of their addresses, and these bits are the binary encoding of the value $i - 1$, which we call *line address*. For example, the address of the second cell of each module is `01---`, while the address of the fourth cell is `11---`. It is exactly this kind of symmetry in the organization that will allow us to access at the same time the 8 memory modules.

[14] Intel's syntax refers to 4-bytes long data as *double words*. We are using here the nomenclature of the AT&T syntax, which will be later used to write assembly programs for the `z64`.

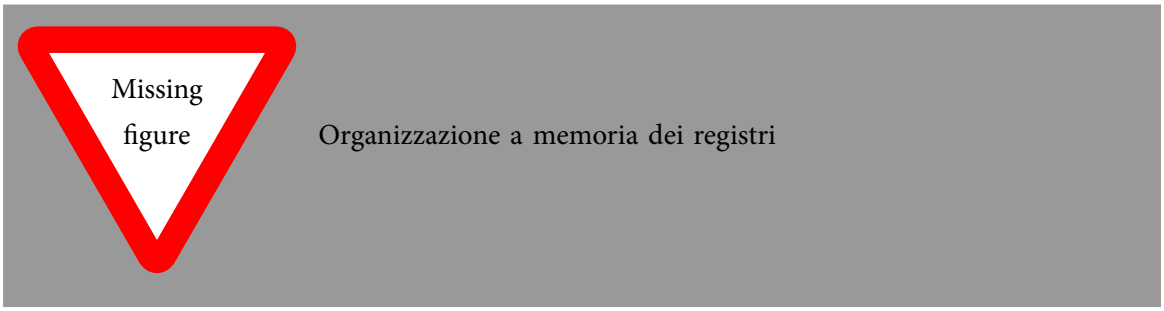



**Figure 5.27:** Memory realized using registers.

In a simplified way, memory modules could be developed using one single decoder and many registers composed of D flip-flops, as shown in Figure 5.27. The number of required flip-flops for each register depends on the desired size. For example, 500 MB of memory could be realized using 500.000.000 rows of 8-bit registers. Of course, from a technological point of view, this is not a viable solution, as it requires too much hardware and would produce a too-slow memory access.

An organization closer to reality is shown in Figure 5.28, where we present a *static*[15] *memory* module with four rows of registers, each one composed of 4 bits. If we now suppose that a flip-flop's size is $0.1\mu m$ ($10^{-7}m$), a 500-MB memory module organized as in Figure 5.28 would occupy an area of $2^{29} \cdot 10^{-7}m \times 8 \cdot 10^{-7}m$, which is a 50 meters-long chip. Of course, this is not a viable solution, so memories are usually organized in a rectangular shape (with similar edge size) where two decoders (a *row decoder* and a *column decoder*) are used to select the word of interest. The number of "registers-per-row" is mapped to a number of matrices of memory cells, which are put one on top of the other, to create the "width" of the row. Additionally, memories usually have an input and an output amplifier, which is used to adapt the different voltages used inside and outside the memory chip. For technical reasons, no more than 8 matrices can be stacked. This is related to the fact that if more matrices are stacked, then the silicon becomes too thick for a proper doping.

In order to properly work, these memory modules require:

- input lines where to specify the address of the cell which is involved in the data transfer. The precise number of these lines depends on the size of the memory module;
- lines used to transfer input/output data, namely the data which must be written into memory in case of a write operation, or the data read from memory which must be passed to the CPU in case of a read operation;
- a control signal used to notify that a read operation is being carried out on memory (Memory Read—MRD);
- a control signal used to notify that a write operation is being carried out on memory (Memory Write—MWR);

These lines are sufficient to let the modules work properly, but for efficiency reasons there is usually an additional control signal which enables specific memory modules—Chip Select, CS. This overall organization is shown in Figure 5.29.

[15] The term static differentiates it from *dynamic* memory, which must be periodically refreshed. Despite its name, static memory is still volatile in the conventional sense that data is eventually lost when the memory is not powered.

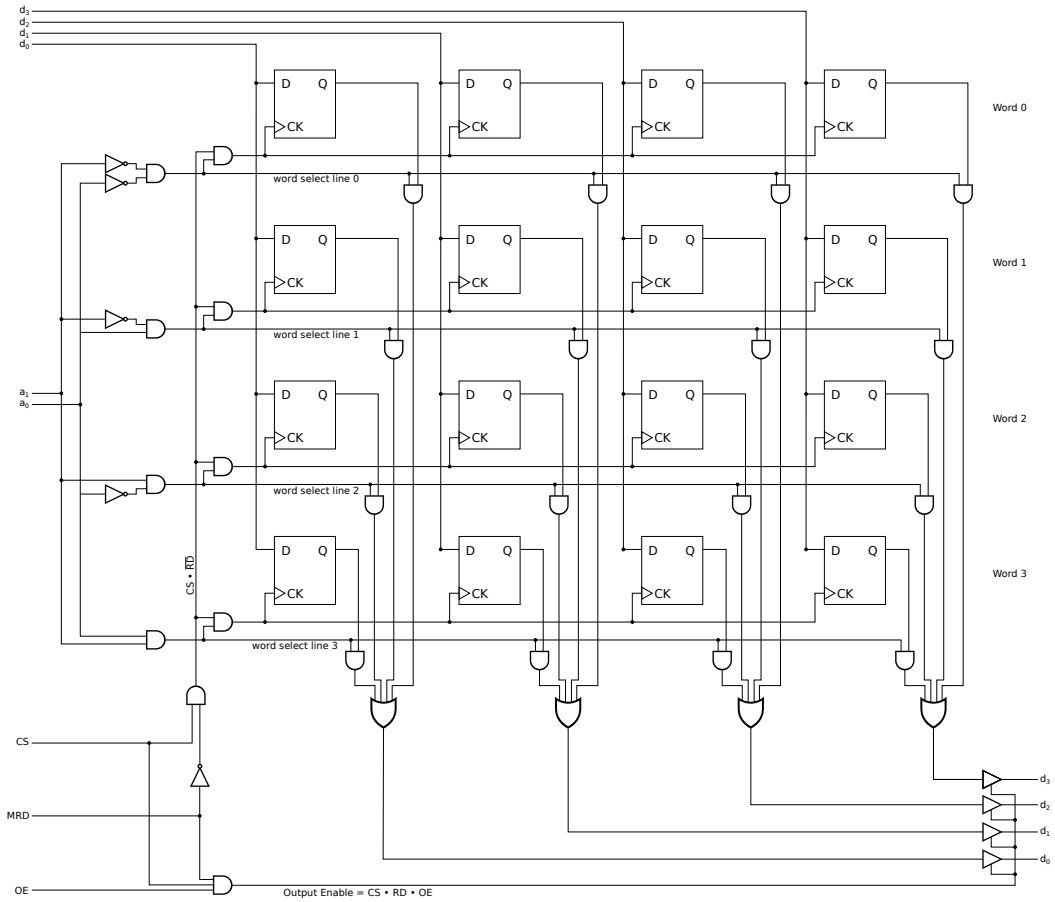



**Figure 5.28:** A static-memory module, having 4 rows made of 4 bits.

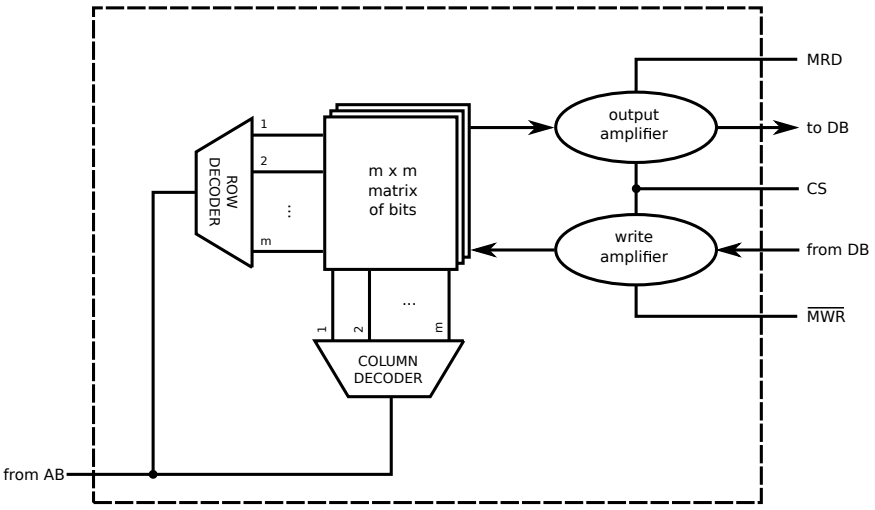



**Figure 5.29:** 1-byte Static RAM memory module.

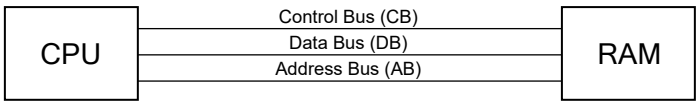



**Figure 5.30:** Simple CPU–RAM Computing Architecture

If using this architectural solution we want to realize a 500-MB memory bank, we need one 14-lines decoder and one 15-lines decoder for the input address, and 8 stacked memory matrices. In this case, the size of the chip would be $2^{14} \cdot 10^{-17} m \times 2^{15} \cdot 2^{-17} \times 8 \cdot 10^{-17}$, which is a chip of $2mm$ per edge, and negligible height. This is a much more viable solution!

These modules are nevertheless passive units: they are not able to perform any action without the intervention of an external unit—the CPU. In particular, it is the CPU's CU which determines when it is necessary to access memory, and it has to find some way to cross the boundaries of the CPU itself to ask the memory module to update its content or deliver some data which it is keeping. This means that the CU must have some "direct" connection to the memory, which allows to activate either the MRD or MWR control signals, to transmit the address of the memory area (1, 2, 4, or 8 bytes wide) that is involved in the read/write operation, and to receive/send data. In this sense, at this point, we can imagine a computing architecture which is organized having only the CPU and the memory, as in Figure 5.30, with three buses connecting them: a *control bus* (CB) with lines dedicated to the MRD and MWR signals, an *address bus* (AB) for the address, and an *internal data bus* (IB) for the data.

With this organization, we can organize the memory bank with 8 modules as in Figure 5.31. It is important to notice that the address bus is composed of 61 bits, as three of the available 64 lines are used by the PU to address one of the 8 memory modules. Nevertheless, using 61 bits leads to an available memory space of $2^{61}$ bytes, which is roughly speaking 2000 peta-bytes of memory! With the current technology, allowing the CPU to expect the system architecture to provide so much memory is unreasonable. In fact, in the real world of x86_64 architectures, the CPU only handles 48 bits (namely, there are 48 lines in the data bus) which gives a total addressable space of roughly 200 TB of memory—still a *huge* space, yet less futuristic. As we will see, to accommodate for this difference, every running program can address the whole 64-bits of *virtual* address space, and it is then a joint work between the operating system and a portion of hardware/firmware logic in the CPU that allows to correctly access *physical* memory, even it is less than the possible theoretical addressing space. For simplicity, until the point at which we will be introducing virtual memory management, we can assume that our computing architecture is equipped with $2^{48}$ bytes of memory, and that only one program is running at a time, so that it has full control on the whole available memory[16].

In Figure 5.31, each memory module is 1-byte wide, while the internal registers of the z64 CPU are 64-bits wide. To allow 8 different memory modules to access the DB at the same time, they must be connected to different lines. For example, we can connect module zero to lines 0–7, module three to lines 24–31, and module 7 to lines 56–63.

It is important to note that when addressing a single byte, it is necessary to select the right memory

[16] This situation actually resembles that of *embedded systems*, where there is commonly a tiny-sized RAM, and the processing unit is in charge of running only a single (or a few) programs. As well the operating systems in embedded systems are either missing, or they are very simplified. Moreover, many embedded systems do not rely on virtual memory, so that we can really see us as working on embedded systems at this point.

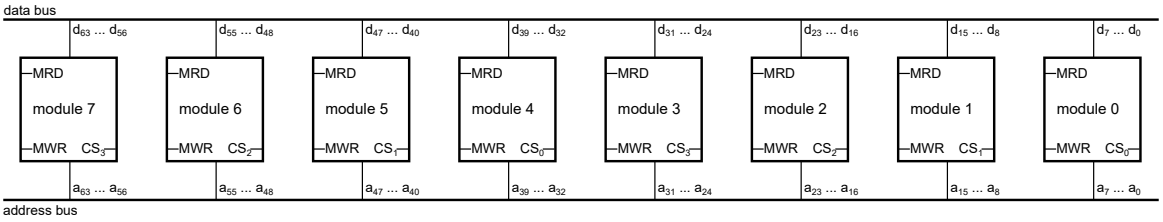
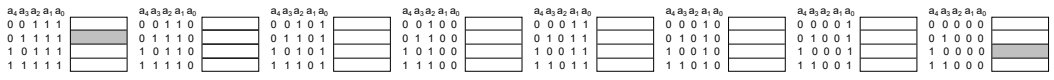
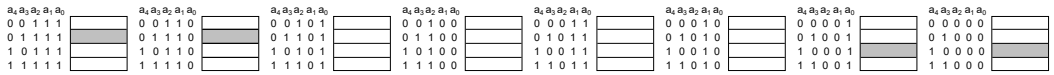



**Figure 5.31:** Organization of the memory bank with 8 modules.



(a) Example of a word with byte address spanning over the next line



(b) Example of a longword with byte address spanning over the next line

**Figure 5.32:** Example of misaligned memory accesses.

module. To this end, the chip select (CS) signals can be generated using the least-significant bits of the AB (which we call $A_2$, $A_1$ and $A_0$). Anyhow, computing the activation rule for the CS signals only from the least-significant bits can lead to errors. In fact, if we look back at Figure 5.26 and we think that we can access data which are 1, 2, 4, or 8 bytes long starting from *any* memory block, we can see that a word, a longword or a quadword might span over a different line (a single byte never creates this issue). This is called the *disalignment* problem, and heavily affects the design of the memory architecture. If on the one hand this could be easily solved at software level, by imposing that all the data (independently of their size) are stored starting from a cell located in module zero (see Figure 5.31), this "solution" to the disalignment problem generates on its turn the *fragmentation* problem: we waste a high amount of space in each row, as blocks 4–7 are used only if we are storing a quadword. This latter solution is somewhat more adopted in the modern days, as memory is becoming always less costly, and so it can make a bit more sense to waste a small portion of it for example to make our software run faster—this is a choice often adopted in programs which must run at a very high performance. In the early days, when memory was extremely costly, it was reasonable to design ways to address the disalignment problem at hardware, so as not to waste costly memory. This is some heritage that we still have nowadays, and it is why when we write a program (unless we're trying to optimize it for speed) we seldom care about whether we are generating an address which is aligned.

In Figure 5.32(a) we show a word whose address is `01111`. The two bytes composing this word are located at addresses `01111` and `10000`. The `z64` is a little-endian architecture[17], so at address `00111` we have the least significant byte, while at address `01000` there is the most-significant one. Similarly, Figure 5.32(b) shows a misaligned longword-based memory access at address `00100`, where the first byte is located at address `00100` (the least-significant one), and the last (the most-significant) is located at address `01001`, which is on the next line. In particular, this memory access involves two bytes on one line, and two bytes on the next.

[17] We recall that little-endian architectures store in memory the data starting from the least significant byte, while big-endian architectures store them the other way round, starting from the most significant byte.

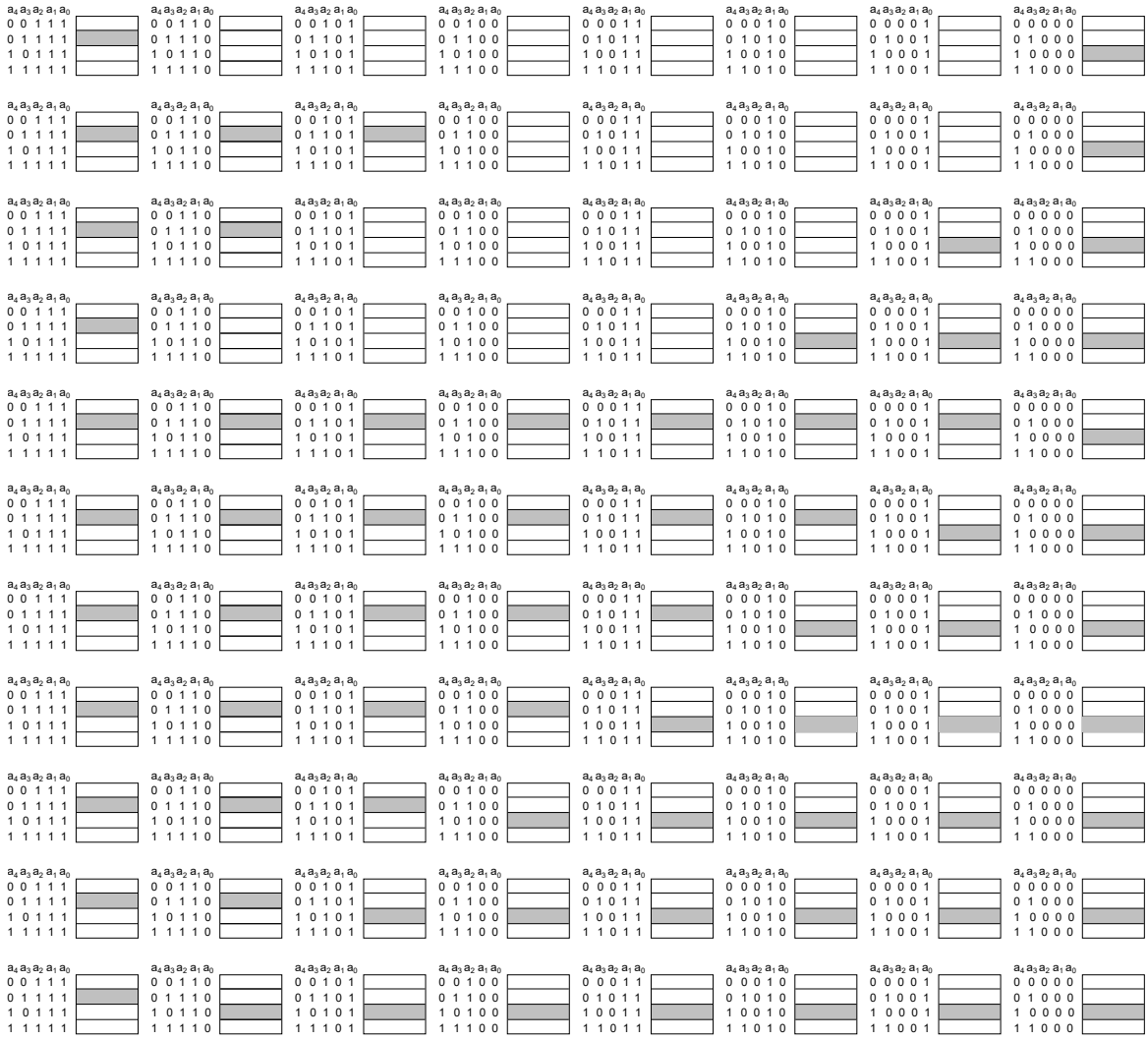


**Figure 5.33:** All possible combinations of misaligned memory, when using 8-blocks organization.

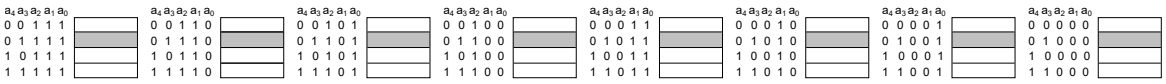



**Figure 5.34:** Row-address aligned memory access example for a quadword (using 64-bits addresses)

To access the word shown in Figure 5.32(a) we must be able to address both the byte at address `00111`, and the byte at address `01000`. Therefore, the CU generates two consecutive memory addresses on the address bus, and will read/write from/to the data bus only the associated part of the word. Additionally, in case of a read operation, the content of the data being read from memory must be correctly recomposed.

To correctly access the memory modules related to the bytes with the same row address, there are eight control signals in the z64 architecture: $MB_0$, $MB_1$, $MB_2$, $MB_3$, $MB_4$, $MB_5$, $MB_6$, and $MB_7$, which are connected to the various CS signals of the memory modules. Nevertheless, when the access is perfectly aligned, as in the example in Figure 5.34, the CU will generate only one address and will enable, at the same time, all the CS control signals.

When we will discuss memory modules in Chapter 11, we will show how it is possible, by relying on a simple combinational network, to avoid this "double read" from memory. To keep the discussion simple, we can assume now that the CPU only requires to send the address of the memory buffer to be read on the address bus, and the memory, *independently of alignment problems*, will write on the data bus, *after a fixed number of CPU clock cycles*, the read data. Similarly, in case of a write operation, the CPU simply sends the address and the data on the address and data buses, and the memory is then able to write the data, even if misaligned, in a fixed amount of time.

### 5.6.1 Addressing Modes

The address of the cell (or the cells) to access in read/write mode is specified by the programmer. In Section 5.5 we have already extended our instruction set to access the subportions of the internal registers. When dealing with memory operands, the z64 instruction set offers extremely flexible ways to access data in memory, which are called the *addressing modes* of the instruction set. Specifically, addressing modes supported by the z64 processing unit define how instructions are allowed to refer to data stored in main memory. Therefore, the instructions which have (explicit) memory operands must be compliant with the addressing mode definition of the CPU.

In particular, a memory location is evaluated by the CPU according to Equation (5.1):

$$\left[\left\{\begin{array}{c}\text{RAX}\\\text{RBX}\\\text{RCX}\\\text{RDX}\\\text{RSP}\\\text{RBP}\\\text{RSI}\\\text{RDI}\\\text{R8}\\\text{R9}\\\text{R10}\\\text{R11}\\\text{R12}\\\text{R13}\\\text{R14}\\\text{R15}\end{array}\right\}\right] + \left[\left\{\begin{array}{c}\text{RAX}\\\text{RBX}\\\text{RCX}\\\text{RDX}\\\text{RSP}\\\text{RBP}\\\text{RSI}\\\text{RDI}\\\text{R8}\\\text{R9}\\\text{R10}\\\text{R11}\\\text{R12}\\\text{R13}\\\text{R14}\\\text{R15}\end{array}\right\} * \left\{\begin{array}{c}1\\2\\4\\8\end{array}\right\} + [displacement]\right] \tag{5.1}$$

where four different variable appear, namely:

- a *base address*, stored within one of the general purpose register, which is therefore referred to as *base register*;
- an *index value*, stored within one of the general purpose registers, which is therefore referred to as *index register*;
- the *scale*, which is a multiplier of the index value. This is encoded directly into the machine representation of the instruction, and is evaluated only if the register value is specified;
- a *displacement*, directly encoded into the machine representation of the instruction.

In any instruction accessing memory, any of the numeric and the register parameters may be omitted. This means that a memory location can be identified by using only a displacement, or by using only a base address, or by using the base address and the index value and the scale—note that if we use the index, then we *must* specify a scale, and vice versa, as these two parameters are strictly coupled together.

In addition to this, there is another case where the z64 CPU has to perform a memory address, which is when it has to deal with *immediate data*. This kind of data is related to operations which mostly involve constants. In fact, it is quite common when writing software that some variables must be initialized to a pre-determined value. In assembly notation, if our variable is the RAX register, we would write this instruction:

```
movq $1000, %rax
```

Here the $ sign is used to tell the assembler that we want to use 1000 as a number rather than as a memory address. In fact, while writing `movq $1000, %rax` means that we want to store the number 1000 into RAX, writing `movq 1000, %rax` means that we want to access memory at address 1000, copy its content into the CPU, and store the value into RAX. Anyway, in both cases, we have a memory access. While in the latter case it is clear why, to understand the reason behind the memory access in the former we must ask ourselves a question: where is the CPU taking the value 1000 from?

This value is encoded within the machine instruction. In fact, as we have already mentioned, the z64 instruction set is variable. The reason behind the variability is due to the fact that one instruction is encoded using 8 bytes which are not enough to store... 8-bytes immediate data! When using such immediate data, the instruction format becomes as in Figure 5.35. Considering that during the fetch phase the value of RIP is incremented by 8, this means that in case immediate data are present, RIP is pointing exactly to them. Therefore, in order to let the CPU process this immediate data, an additional memory address, to the one pointed by RIP, must be executed, thus giving rise to a MEM phase.

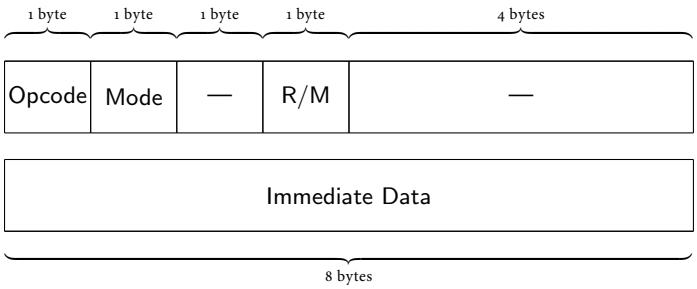



**Figure 5.35:** Machine instruction with immediate data

While we will discuss in the details how to use from a programmer's perspective the addressing modes in Section 6.4, after some practice with assembly programming, it is interesting here to understand how the addressing modes are encoded into a machine instruction, because this has effects on what control signals are present in the PU of the z64.

In particular, memory accesses are specified via the four least-significant bits of the Mod byte, and third byte of the machine instruction. Concerning the Mode byte, the four least-significant bits are divided into two 2-bits sub-fields, which are called *Displacement* (DI) and *Memory* (Mem). Mem tells whether the source and destination operand are register or memory operands: the first bit is associated with the source operand, the second bit with the destination, the value zero tells that the operand is a register, while the value 1 tells that the operand is a memory location. Considering that the z64 architecture does not allow to make a memory-to-memory data movement in a single instruction, the value 11 is illegal for the Mem field, and is therefore never generated by a z64 assembler.

The DI field tells whether the instruction is using a displacement, as of Equation (5.1), and/or an immediate data. Similarly to Mem, the first bit is associated to the displacement, the second bit to immediate data, and a value of zero tells that the corresponding field is not used/present, while 1 tells the contrary. In case immediate data are used, the I bit tells exactly that a memory access should be performed to retrieve them. Additionally, the usage of immediate data can be only as a source operand, as it is meaningless to tell that we want to execute an operation which has as the *destination* some *data*! Then, in this case, if the bit I is set, the CPU will not look at all the bits which are related to source operands in Mem.

When the displacement is used, its value is stored in the last 32 bits of the instruction. This gives rise to a possible memory access of up to $2^{32}$ bits, which is less than the total $2^{48}$ available to the z64 CPU. This is anyhow not a problem, as the value of the displacement can be summed to that of a base register, which then gives the possibility to span the entire memory from any point. It is nevertheless

the role of the programmer to take care of this situation and implement its code accordingly.

A different situation, which is related to memory saving, rises when the field DI tells that we have immediate data but no displacement, and the SS field in Mod tells that the size of the source operand (which is the immediate data) is up to 32 bits (namely a byte, a word, or a longword). In this case, the displacement field would be unused, and using the additional immediate field would be just a waste of space and time (due to the additional memory access). In this case, the immediate data is stored in place of the displacement, with a possible sign extension if its size is less than a longword.

Concerning the third byte of the machine instruction, we have always said so far that it was set to a don't care condition, because its goal is to specify what of the remaining parameters of Equation (5.1) are actually used by a memory access. Therefore, this byte is called the *Scale, Index, and Base* byte (SIB), and is organized as in Figure 5.36. $B_p$ and $I_p$ tell whether the addressing mode used by the current instruction is using a base and/or an index register, respectively. If $I_p == 1$, the field Index keeps the binary representation of the index register, according to the values shown in Table 5.1, and the field Scale keeps the scale value, the possible values of which are 1 (encoded as 00), 2 (encoded as 01), 4 (encoded as 10), and 8 (encoded as 11).

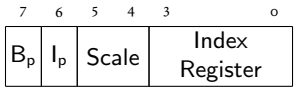



**Figure 5.36:** SIB Byte Format

Putting it all together, our machine instruction representation becomes as in Figure 5.37. Additionally, for the sake of clarity, Table 5.5 reports all the possible values of the Mod byte, which plays a central role in the execution of operations which involve data operands (such as movement or arithmetich/logical operations).

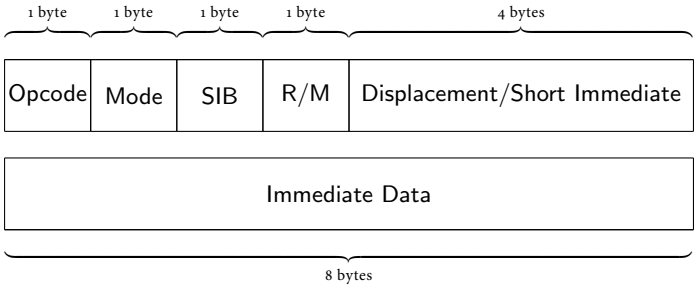



**Figure 5.37:** Machine instruction with immediate data

At this point, it is clear that to support the execution of these memory accesses the CPU must be able to extract from IR the encoding of three registers: the source/destination register, a base register, and an index register. Not all instructions will require all them, yet in case they are used, three different lines must be used. This requires a modification to the PU of the z64, exactly adding this additional line from IR to the multiplexer which selects the register. The multiplexer must be modified as well, as it has no longer to deal with two input codes. Thus, the number of lines of $S_{MUX}$ must be increased to 2, thus giving the possibility to select up to four different input values (even if only three will be legal: the CU will never generate a value $S_{MUX} = 11$).

**Table 5.5:** Mode Byte Fields Explanation

| Field | Value | Explanation |
|---|---|---|
| SS | 00 | Source is a byte |
| | 01 | Source is a word |
| | 10 | Source is a longword |
| | 11 | Source is a quadword |
| DS | 00 | Destination is a byte |
| | 01 | Destination is a word |
| | 10 | Destination is a longword |
| | 11 | Destination is a quadword |
| DI | 00 | Displacement is not used, immediate is not present |
| | 01 | Immediate is present |
| | 10 | Displacement is used |
| | 11 | Displacement is used and immediate is present |
| Mem | 00 | Both R/M Source and Destination are registers |
| | 01 | R/M Source is register, R/M Destination is memory |
| | 10 | R/M Source is memory, R/M Destination is register |
| | 11 | Impossible condition (generates a runtime error) |

We have to reason now about an additional issue of this organization. Going back to Equation (5.1), we can see that in order to determine the final memory address, some computations should be made. The z64 does not have a multiplication unit, therefore to multiply the value of the index register by the scale we must find out a different approach. The solution is straightforward, as the legal values for the scale are all powers of two. Thus we can resort to the shifter to make these multiplications: a scale value of 1 will result in a zero-positions shift, a value of 2 will result in a shift to the left by one position, a value of 4 will result in a shift to the left by two positions, and a value of 8 will result in a shift to the left by three positions. Similarly, to sum the values of the base register and of the displacement, we can simply use our available ALU. Nevertheless, when making these computations, we must consider the fact that these computations are not related to calculations required by the user directly. Executing arithmetic/logical operations has the side effect of updating the content of the FLAGS register. The programmer can be unaware of this, and therefore we have to preserve the content of FLAGS when computing a final memory address. This can be done easily by introducing the $W_{FLAGS}$ control signal, which acts as the load enable for the FLAGS register. This signal will be activated only when the ALU or the shifter are used to execute operations which are related to instructions explicitly written by the programmer, while when computing final memory addresses it will be disabled, so as to prevent any update to the content of the register.

All these modifications to the PU are reported in Figure 5.38. In our journey towards the design of a multi-cycle CPU, this is almost the last version. We still miss a way to let the CPU communicate with the external main memory, which is something that we will discuss in the next section.

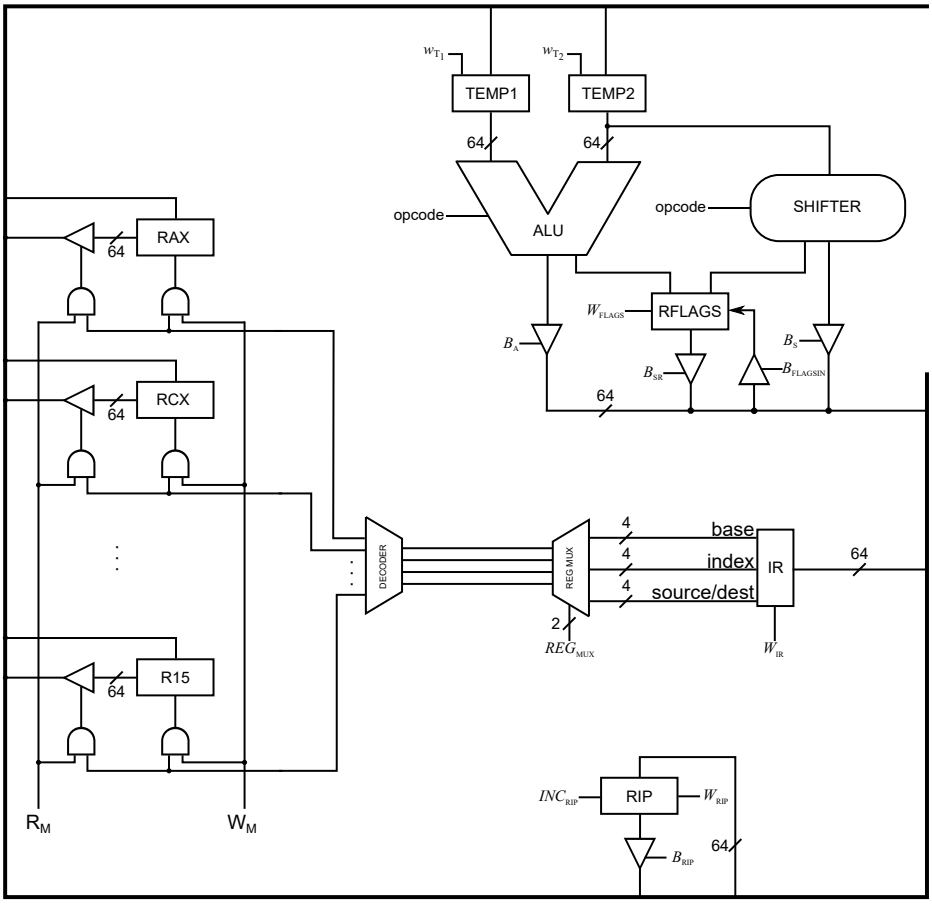



**Figure 5.38:** PU organization with 3 register lines from IR and $W_{FLAGS}$.

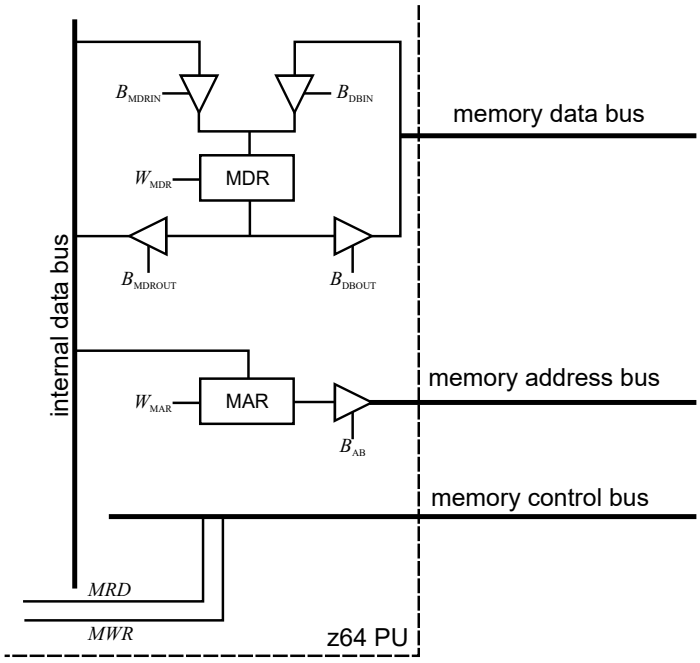



**Figure 5.39:** PU interface to main memory, in the case of 64-bits data.

### 5.6.2 Talking to the external main memory

As we have already discussed, the address of the memory cells involved in a memory access is determined by parameters directly encoded within the instruction. After that the CPU has computed the final address, it has to be transferred to the external main memory bank. As a consequence, we have to introduce an additional register within the z64 to keep the encoding of the target memory address. This is a 48-bits register (remember that the z64 does not allow to address more than $2^{48}$ cells of memory), it is called *Memory Address Register* (MAR), and it is used to drive the lines of the address bus.

Transferring the address to the memory bank is not enough to access memory: we must provide for an additional register which is used to receive data in case of a memory read, or to send data in case of a memory write. This buffer register is 64-bits wide, and is called *Memory Data Register* (MDR). To let the data flow to/from the MDR, it must be physically connected to memory, so its flip-flops are connected to the data bus. To control the direction of the data transfer, and to separate electronically the PU internal to the z64 to the outside world, we use a couple of three-state buffers. At the same time, the output of MAR to the address bus could be set to high impedance (and in fact, we will see in Chapter 7 that it is extremely important to have this possiblity), and a three-state buffer is then placed as well between it and the address bus. Overall, the interface between the PU and the external memory can be summarized as in Figure 5.39

## 5.7 The Control Unit

As we have already seen in Chapter 1, the organization of the CU is strictly coupled with that of the PU. It should be clearer now that this is due to the number and type of control signals used by the different parts of the hardware which is used to carry on the execution of an instruction, and their temporal sequence.

Considering that we are now designing a multi-cycle CPU, we can only execute one instruction at a time, but the execution of this instruction is anyway split into different phases. Different instruction might require more or less phases, but we can nevertheless set the clock period according to the duration of the longest phase, among all the instructions. Each one of these phases is referred to as a micro-operation, or $\mu$-op. During the execution of subsequent $\mu$-ops, even of the same instructions, the hardware components of the PU can be re-used. This allows for a simple implementation of the PU at the hardware level: we have one single data bus to connect all the components, and the same ALU can be used both to perform operations requested by the programmer, or to compute intermediate values required by the CPU, for example to compute the final value of a memory address—after having disabled the update of the FLAGS register, which is done setting the $W_{FLAGS}$ control signal to zero.

All these choices have a strong influence on the organization of the CU of the z64 CPU, and its functioning. First of all, given the high number of operations which are supported by the z64 CPU and the high number of temporal sequences (which are proportional to the number of states of the CU), it is convenient to design the PU using *microprogramming*.

To execute a program stored in main memory, a multi-cycle CPU designed according to the von Neumann architecture must map one instruction to a microprogram of the CU (which is stored in the internal permanent memory of the CU). The CPU is therefore a *logical structure* that *interprets* programs written in machine language (which, we recall, is generated by the assembler starting from an assembly program). This is done through the execution of a set of microprograms, done by the CU, whose outputs (the commands) act on the PU (including working memory and devices) to move or manipulate data. Each microprogram corresponds to a machine instruction and the microcode is identified by the opcode of the instruction.

Thus, the CU executes a microprogram for each instruction which is read from main memory. The microprogram is composed of a set of *microinstructions* whose number depends on the amount of $\mu$-ops to be executed according to a precise sequence. It is exactly the maximum number of possible $\mu$-ops and of possible $\mu$-ops that are executed in parallel which determines the possibility to opt for a so-called *horizontal*, *vertical*, or *diagonal* organization of the microcode.

The optimal choice for the organization of the microcode comes from a trade-off between two contrasting needs: good flexibility and high operational power of the microcode on the one side, reduced memory consumption of microprogram memory on the other. It is possible to devise structures of the microcode which privileges the former or the latter: if we give more importance to the small memory footprint on the internal memory bank, microinstructions are usually highly encoded and limited in the number of possible branches to take, which leads to the *vertical* organization of the microcode. On the other hand, if we give too much importance to the expressiveness of the microcode, which means that we target the maximum parallelization of $\mu$-ops and the maximum flexibility in

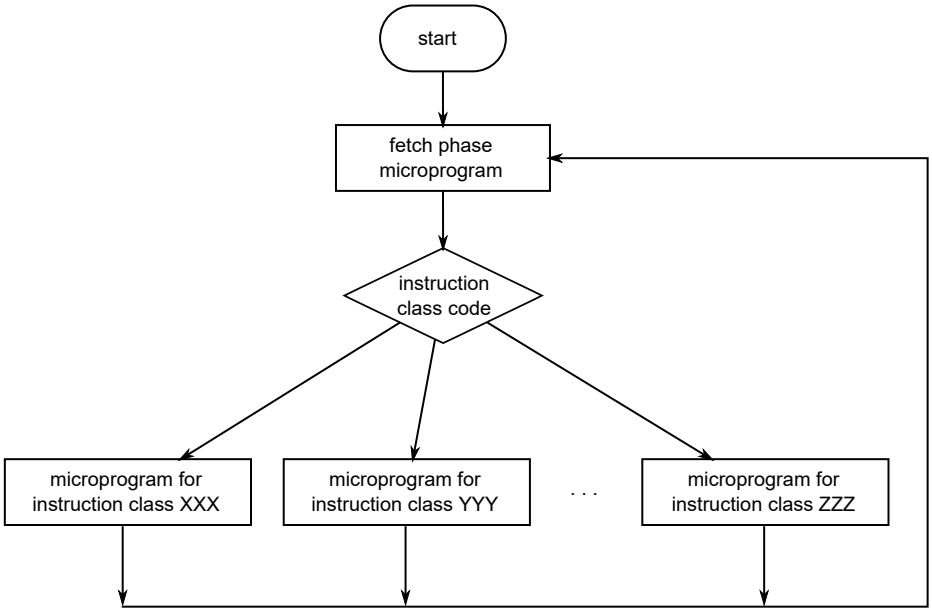



**Figure 5.40:** A possible high-level organization of the CU microcode.

the sequencing of the addresses, this is a *horizontal* organization of the microcode. Any intermediate solution is called a *diagonal* organization. Of course, a horizontal organization of the microcode must rely on a PU which has a high operational parallelism, namely a non-reduced redundancy of the hardware components. If this is not the case, the potential of the horizontal organization of the microinstructions will be under-used.

To interpret any program, the PU must be structured so as to map each machine instruction to one or more sequential procedures (namely, more distinct microprograms). To this end, the PU can be organized as a single ROM memory, where each microprogram is a sub-machine of one single finite state automata. Deciding on which microprogram to execute depends on the class of the instruction that the PU is currently executing, something which is discovered by the PU only during the decode phase. In fact, all the instructions are stored in main memory in the beginning, and their binary representations are copied to the IR register during the fetch phase. By the instruction format of the z64 instruction set, the CU can easily extract the class of the currently fetched instruction from the IR register during the decode phase (this is exactly why all the instructions keep the class field at the same bit position). If we do not take into account the possibility that the execution of an assembly instruction is interrupted by any external event (this is something that we will introduce in Chapter 7), the microcode can be organized as in Figure 5.40.

The execution of the microprogram of a single machine instruction is called *instruction cycle*. Each instruction cycle is composed of multiple elementary phases which entail the activation of control signals ($\mu$-ops) directed to internal units of the CPU (registers, ALU, shifter, …) and/or units external to the CPU (main memory, or external devices, as we shall see in Chapter 7). Each elementary phase is called *machine cycle*, and every instruction can be as well described in terms of the machine cycles needed to carry out its execution—in its turn, they describe the "speed" of the instruction's execution. The microcore associated with the interpretation of the instruction can be schematized as in

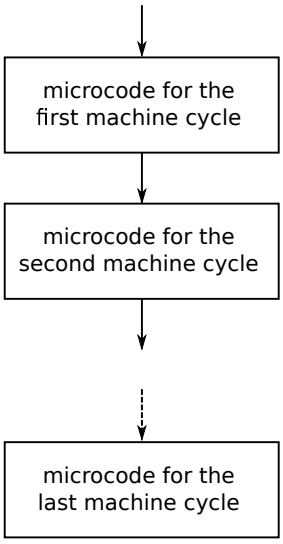



**Figure 5.41:** How to organize the microprogram associated with a generic machine instruction.

Figure 5.41.

To allow for the execution of the microprogram associated with the machine instruction the CU will receive as input the class of the decoded instruction. Thus the CU will have as many inputs as:

- The condition variables coming from outside of the CPU;
- The condition variables coming from inside of the CPU, such as the value of all the status flags (stored in the FLAGS register);
- The class (and the type) of the instruction (stored in the IR register);
- The addressing mode of the operands of the instruction (stored in the IR register).

The number of outputs of the CU depends on the number of control signals which are necessary to drive the PU of the CPU and all the external modules. The organization of the CU depends on the implementation cost, on the desired performance, and on the type of finite-state machine chosen (Mealy or Moore).

If this organization is satisfactory and no minimization of the cost has to be involved, we can use the organizations shown in Chapter 4, where the size of the ROM is determined by accounting for all the condition variables (inputs), all the control signals (outputs), and using a number of state variables suitable to encode all the possible states. In the simplest case, where we do not want to rely on reentrant code, the total number of states is given by the states of the fetch phase, and all the states of the sub-machines associated with each instruction to be interpreted. If the number of fetch states is $N_0$, and the number of states for the $i$-th instruction class is $N_i \quad \forall i \in [1, M]$, where $M$ is the total number of instruction classes, the number of state variables $k$ is:

$$k = \left\lceil \log_2 \sum_{i=0}^{M} N_i \right\rceil \tag{5.2}$$

Let us call $m$ the number of input variables, and $r$ the number of output variables. In the case of a Mealy machine, the size of the ROM is $2^{m+k}$ words, each one of $k + r$ bits, while in the case of a Moore machine, the ROM will be composed of $2^k$ words of $k2^m + r$ bits. In both cases, since the number of condition variables, control variables, and state variables are tens, the size of the ROM is non-negligible and thus is costly. To this end, we must reorganize the ROM of the CU, trying not to reduce its performance. A significant cost reduction can be obtained by reorganizing CU's inputs and outputs, since the final size and the length of each word depend on their number, as we have seen.

The first optimization comes from the fact that the microinstructions associated with the interpretation of one machine instruction can be physically allocated in adjacent cells. The code of the instruction class can be therefore used as the base address of a block of microinstructions. As a consequence, given one microinstruction, rather than encoding the whole address of the next one[18], we can only encode the displacement. In this case, it is important to note that the number of bits needed for the displacement must be dimensioned on the longest microprogram, among the ones for each instruction class and the fetch phase.

The number of different microprograms is equal to the number $M$ of different instruction classes, plus the fetch microprogram. Therefore the size of the *base register* to access a microinstruction is:

$$\left\lceil \log_2(M+1) \right\rceil \tag{5.3}$$

while the number of bits for the displacement register is:

$$\max \left\lceil \log_2 N_i \right\rceil \qquad \forall i \in [0, M] \tag{5.4}$$

A second optimization comes from a reduction of the number of condition variables that can be examined in each state. In fact, given the tasks of the CPU (which is designed according to the von Neumann architecture) in each state the CU can examine only a subset of the condition variables. For example, during the fetch phase, the CU will not examine any flag stored in the FLAGS register. Therefore, the various input signals can be masked by a selection circuit, driven by a specific field stored in the current microinstruction. Therefore, by increasing the size of the word by a small amount of bits (which we call SEL field, for *selection*), we can significantly reduce the number of input variables that the CU is receiving as input.

Figures 5.42 and 5.43 show two possible structures to implement these variants, using a Mealy and Moore machine, respectively.

Additional optimizations might come from a reduction in the number of control signal (outputs) generated by the CU. In fact, all the signals that exclude each other could be logically grouped. A specific binary encoding can be used so as to identify, within a specific group, which signal should be activated. For example, if we consider the control signals for the ALU, rather than using a number of control signals equal to the number $n$ of operations supported at the hardware level (in the case of the ALU of the z64 CPU, they are 10 operations), we can use $\lceil \log_2 n \rceil$ signals (which is 4, in our case). Of course, in order to let the ALU understand the operation command, its hardware must be slightly modified (for example, by using a decoder) in order to transform this command code back

[18] In case of a Mealy machine we can have multiple next instructions.

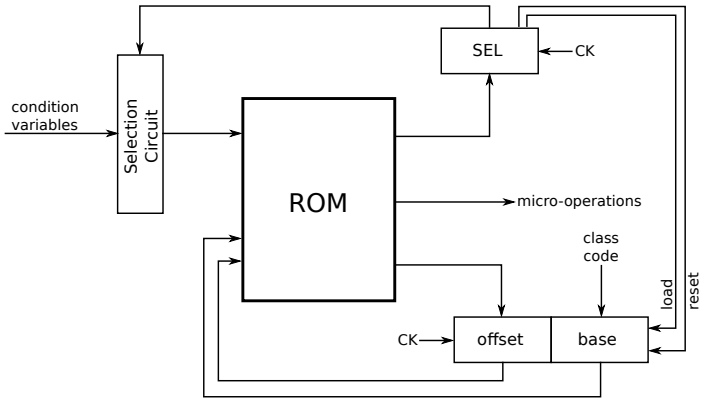



**Figure 5.42:** CU organization according to the Mealy machine model.

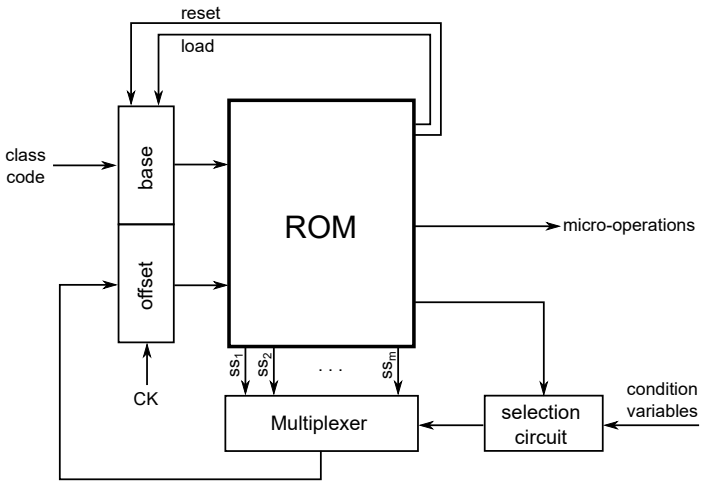



**Figure 5.43:** PU organization according to the Moore machine model.

into a control signal.

Organizing the microprogram as reentrant code can lead to further optimizations. In fact, different machine instructions can entail the execution of identical portions of the microcode. To exploit this redundancy, the CU should be able to *jump* to a specific instruction and then *jump back* to the point where this code block was *called*. In order to do so, the CU's microcode must introduce some *jump* and *return* microinstructions, and must be able to rely on some memory organized as a stack, where a microinstruction can store the return address of the called code block.

The speed according to which the CU can run its microcode (which is reflected in the clock speed of the CPU) depends on the type of machine used to build the CU (Mealy and Moore) and the stabilization time of the PU's combinational network. For the CU, this speed is independent of the state in which it currently is, while for the PU it highly depends on the functional units which are involved in the execution of the instruction. Therefore, if we fix the speed of the CU, the faster the hardware of the PU is, the smaller the system's clock speed. Considering that the dimensioning of the CPU frequency is based on the slower component, it is important to devise the whole structure of the hardware so that the disparity among the components' speed is not that high.

### 5.7.1 Additional changes to the PU

The addressing mode of the z64 CPU is quite complex and, as already shown, allows for a very versatile way to identify the position of an operand in memory. The CU determines what are the actual operands of the memory access by looking at the SIB byte, where the $B_p$ and $I_p$ flags are used as condition variables during the execution of the microprogram associated with the instruction accessing memory.

Anyhow, in order to compute the final address as for Equation (5.1), the CU must rely on the hardware offered by the PU, namely the ALU and the shifter. While it is quite easy that shift operations can be mapped to multiplications to compute the $index \cdot scale$ part, and that the ALU can easily sum the other operands, it can be a little more tricky to understand how to use the available temporary registers. In fact, there is one situation when the output of the ALU should be directed to one of the two temporary input registers. This is the case when the addressing mode is using all the four parameters (namely, base, index, scale and displacement), as the steps to undertake to compute the final address would be:

1. Copy the content of the index register to TEMP2
2. Compute a shift of $\log_2(scale)$ to the left and store the result in TEMP1
3. Copy the content of the base register to TEMP2
4. Compute the addition of TEMP1 and TEMP2, and store the result in TEMP1
5. Copy the displacement in TEMP2
6. Compute the addition of TEMP1 and TEMP2, and store the result in MAR to later access memory

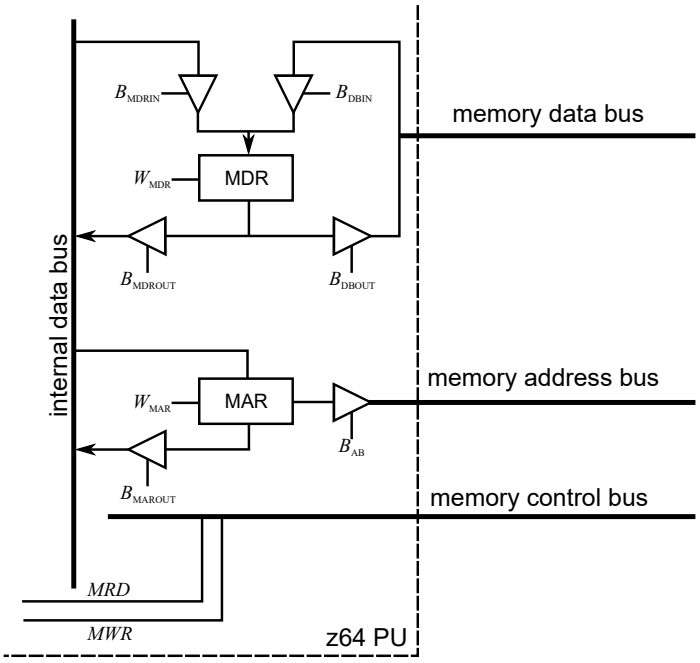



**Figure 5.44:** Insertion of an additional line (and the $B_{MAROUT}$ control signal) to let the content of MAR be transferred to the Internal Data Bus.

Step 4 has some problems. In fact, the ALU is a combinational network, and we have already discussed that, during a clock cycle, the inputs to the network must be stable so as not to hamper the correctness of the result. If we write the result of an operation by the ALU in one of its input registers, we fail to satisfy this condition[19].

To solve this issue, we modify the organization of the z64's PU adding an additional line coming out of the MAR register. This allows to use this internal register as a temporary buffer when computing a target memory address according to the addressing mode of the z64 CPU. Of course, to add this connection, a three-state buffer must be added as well, along with the $B_{MAROUT}$ control signal (coming from the CU) which enables reading its content.

### 5.7.2 Communication with Main Memory

### 5.7.3 The final z64 PU Architecture and some $\mu$-ops

Considering all the discussion that we had so far on the organization of the z64 PU, the final architecture is shown in Figure 5.48. By looking at this picture, we can now start to discuss some of the microprograms which are used by the z64 CU to carry out the execution of certain families of in-structions. A $\mu$-op is represented in the generic form `destination` ← `source`, where `destination` `source` can be any component of the z64 PU. Every $\mu$-op, which represent one (or, in certain cases,

[19] This problem could be solved if the CU read the content of the register at the rising edge of the clock, and wrote the result back at the falling edge. Nevertheless, this solution would require an additional discussion which is out of the scope of this book, and we therefore adopt a different one, which is more costly in the usage of hardware, but simpler from the logical point of view.

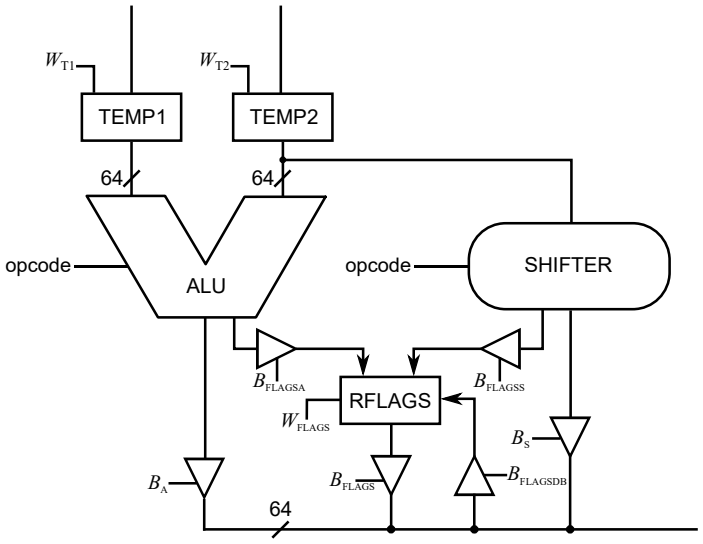



**Figure 5.45:** Three-state buffers to update the content of the FLAGS register with the updates from the ALU, the Shifter, or directly from the Internal Data Bus.

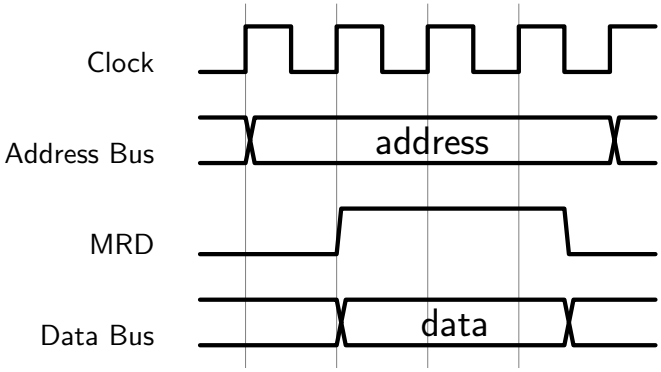



**Figure 5.46:** Memory Read Cycle

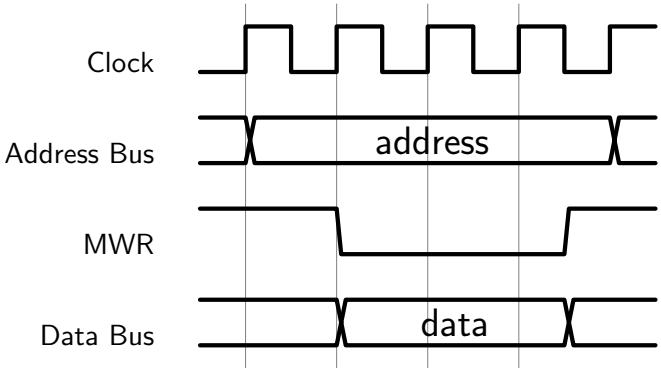



**Figure 5.47:** Memory Write Cycle

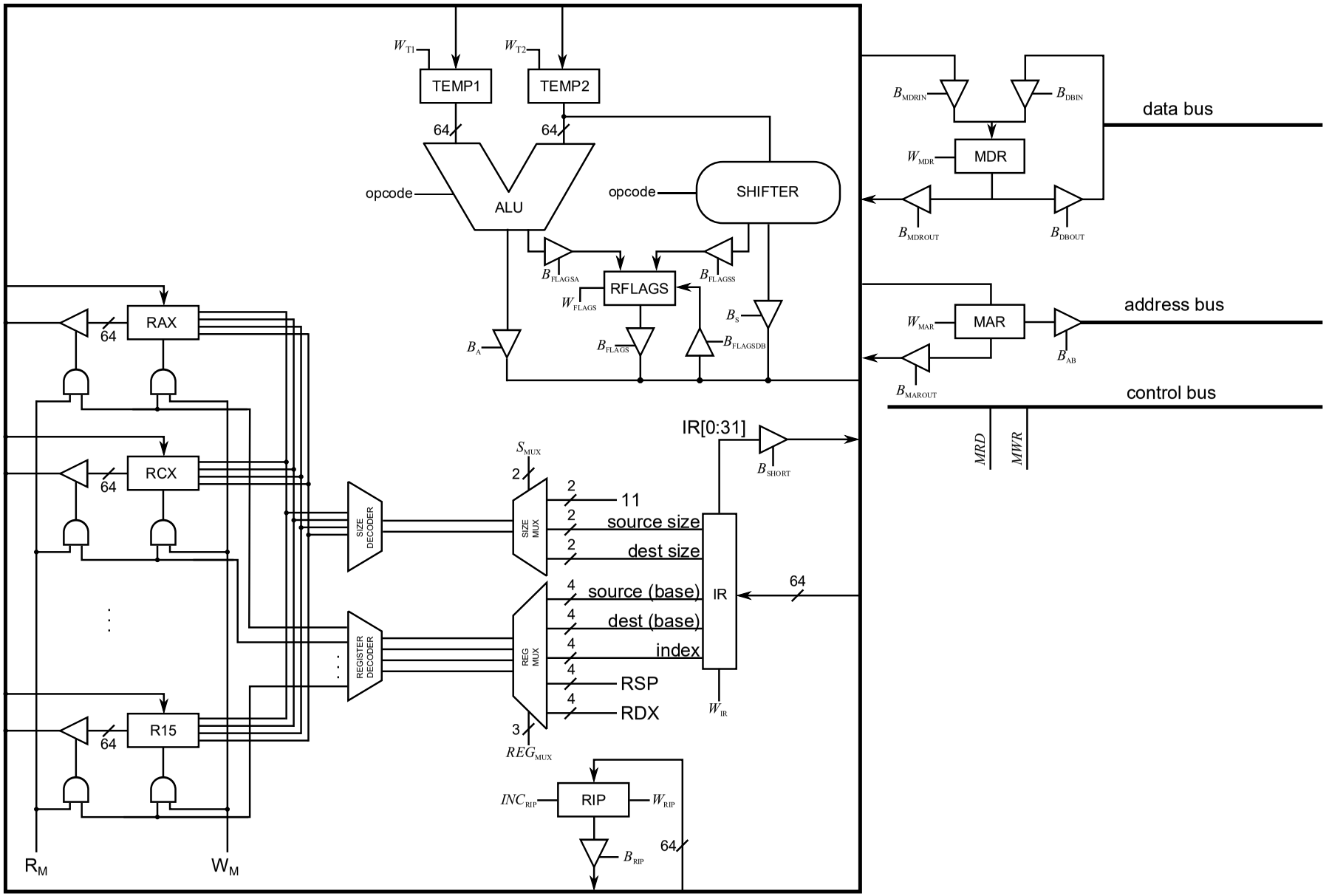



**Figure 5.48:** The final architecture of the z64 CPU.

multiple) state of the finite-state machine which implements the CU, is associated with a set of control signals which are set by the CU. It is important to note that *all* the control signals in the CPU must have some value: it is meaningless to say that they have *no* value, due to the combinatorial nature of the CPU. Nevertheless, for the sake of simplicity, we assume that any control signal which is not explicitly set, has a value of zero.

All the instructions, independently of what they do, start with the fetch phase, which is implemented in the z64 CU using the following $\mu$-ops, and the associated control signals:

1. `MAR` ← `RIP`
   - $B_{RIP} = 1, W_{MAR} = 1$
2. `MDR` ← (`MAR`); `RIP` ← `RIP + 8`
   - $INC_{RIP} = 1, B_{AB} = 1$
   - $B_{AB} = 1, MRD = 1$
   - $B_{AB} = 1, MRD = 1, B_{DBIN} = 1, W_{MDR} = 1$
3. `IR` ← `MDR`
   - $W_{IR} = 1, B_{MDROUT} = 1$

The fetch phase's goal is to load the next instruction from memory and increment the value of RIP. Therefore, the first $\mu$-op copies the content of RIP into MAR, as RIP keeps the address of the next instruction to be executed and MAR is the buffer register towards the address bus. This copy is performed by setting the $B_{RIP}$ control signal (which makes the content of RIP flow on the internal data bus) and setting as well the $W_{MAR}$ control signal, so that the data flowing on the internal data bus is actually stored into MAR

The second $\mu$-op executes the actual memory access. By the discussion given in Section 5.7.2, we know that in our first approximation of the system the memory is three times slower than the CPU. Therefore, this $\mu$-op is actually split into three different states, which activate different control signals. First of all, we set $B_{AB}$, which lets the address stored into MAR flow on the external address bus. In this way, the memory starts receiving the address, and can compute which are the cells that are currently keeping that data. In the second state (thus, in the second clock cycle), we set MRD on the control bus. This tells the memory that we are interested in reading the content of the memory. At the third clock cycle, the memory has placed the content of the data stored at that memory address on the data bus, so we can store them in MDR. This is done by setting $B_{DBIN}$, which connects MDR's input lines to the data bus, and $W_{MDR}$, which stores the data in the register.

Now that we have copied some content from working memory to the CPU, we have to place the loaded data into the proper register associated with the operation which we are carrying out. Since this is the fetch operation, MDR contains an instruction, and therefore its content must be copied to IR. This is done by the third $\mu$-op, which sets $B_{MDROUT}$ to let the content of MDR flow on the internal data bus, and $W_{IR}$, which stores what is currently on the internal data bus in IR.

At the moment, we have not yet incremented the value of RIP, which is something to do during the fetch phase. We could simply add another $\mu$-op to the fetch phase, but in order to reduce its duration (it's already 5 clock-cycles long!), we should note that during the execution of the second $\mu$-op, the hardware of the PU implementing the RIP register is not used. Therefore, the CU can emit as well, during the execution of the second $\mu$-op, the $INC_{RIP}$ control signal, which increases by 8 the value of RIP, since it is implemented as a counter register. $INC_{RIP}$ is kept set only during the first clock cycle of the second $\mu$-op, as one clock cycle of the CPU is enough to update the value.

Let us now consider a register-to-register data movement instruction, namely:

```
movq %rax, %rcx.
```

This instruction, after the fetch phase, has to move the content of the RAX register into the RCX register. While, from a technical point of view, it is possible to copy the content of RAX directly into RCX, we should account here for the possibility that it is perfectly legal for the programmer to write an instruction such as `movq %rax, %rax`—this maps to a no-operation. In this latter case, we are in a situation similar to the one discussed in Section 5.7.1. We cannot read the content of a register and move it to the same register's input lines. Now, there are two possibilities to solve this issue. On the one hand, we could modify the microprogram associated with the `mov` instruction, so that we explicitly check whether the source and the destination operands are the same—in that case we do not execute any data movement. Nevertheless, in all the other circumstances when the source and the destination are different, which is a more likely situation, we would pay the additional cost of checking this corner case. Thus, we adopt the other solution: we *always* pass through a temporary register, independently of what the source and destination are. Therefore, the $\mu$-ops for this instructions are the following ones (we do not report, again, the control signals associated with the fetch phase):

1. MAR ← RIP
2. MDR ← (MAR); RIP ← RIP + 8
3. IR ← MDR
4. TEMP2 ← RAX
   - $REG_{MUX} = 0, S_{MUX} = 1, R_M = 1, W_{T_2} = 1$
5. RCX ← TEMP2
   - $REG_{MUX} = 1, S_{MUX} = 2, S\_opcode = 000000, B_S = 1, W_M = 1$

As we can see, copying the content of a register into a temporary register is straightforward: the CU sets $REG_{MUX}$ to extract the encoding of the source register from IR, which activates through the *register decoder* the selection line for the actual register (RAX, in our case). Similarly sets to 1 the value of $S_{MUX}$, to drive the combinational network shown in Figure 5.23 in order to select the correct size of the source register (64 bits in our case). $R_M$ is set, to identify the fact that we are reading from the selected register, and $W_{T_2}$ is set so that a copy of RAX can be made into TEMP2, through the internal data bus.

Now, the temporary register TEMP2 was not chosen by chance. In fact, we have to copy the current content of TEMP2 back into RCX, but there is no direct connection from temporary registers' output to the internal data bus. Rather than adding more wires, we can use the already existing ones, namely those of the shifter, driving it to perform a zero-position shift, which does not alter the data. The output of the shifter is then connected to the internal data bus, so we can make a copy of the content of TEMP2 into RCX. Note that this operation does not add any overhead, because the clock frequency has already been set taking into account the latency of the shifter, and we are additionally saving some wires, thus keeping the overall cost a little lower. To ask the shifter to perform a zero-position shift, the CU sets its opcode ($S\_opcode$) to 000000, telling that we demand a left shift operation of zero bits. $B_S$ is set, so that the output of the shifter goes to the internal data bus, $REG_{MUX}$ is set to 1, telling that we are interested in the encoding of the destination register from IR, and $W_M$ is set, to tell that we want to write the data flowing on the internal data bus into the selected register.

Let us now consider a more complicated data movement instruction, namely:

```
movq %rax, 0xaaaa(%rax, %rcx, 8).
```

This instruction uses all the parameters of the z64 addressing mode to determine the memory location of the destination operand. In this case, during the decode phase, the CPU knows (by inspecting the Mem field of the Mode byte, which is 01 in this case) that the source is a register and the destination is a memory operand. Therefore, when it comes to computing the actual address of the destination, the CU inspects $B_p$ and $I_p$ of the SIB byte (both set in this case) and the value of DI in the Mode byte, to determine that all parameters are used. The computation of the actual address starts from right to left, and follows the steps that were discussed in Section 5.7.1. Therefore, the $\mu$-ops required to carry out the execution of this instruction are:

1. MAR ← RIP
2. MDR ← (MAR); RIP ← RIP + 8
3. IR ← MDR
4. TEMP2 ← RCX
   - $S_{MUX} = 0, REG_{MUX} = 2, R_M = 1, W_{T_2} = 1$
5. TEMP1 ← Shifter_Out[SHL, 000100]
   - $S_{opcode} = 000100, B_S = 1, W_{T_1} = 1$
6. TEMP2 ← RAX
   - $S_{MUX} = 2, REG_{MUX} = 1, R_M = 1, W_{T_2} = 1$
7. MAR ← ALU_OUT[ADD]
   - $A_{opcode} = 0000, B_A = 1, W_{MAR} = 1$

8. `TEMP1` ← `IR[0:31]`
    - $B_{SHORT} = 1, W_{T_1} = 1$
9. `TEMP2` ← `MAR`
    - $B_{MAROUT} = 1, W_{T_2} = 1$
10. `MAR` ← `ALU_OUT[ADD]`
    - $A_{opcode} = 0000, B_A = 1, W_{MAR} = 1$
11. `MDR` ← `RAX`
    - $S_{MUX} = 1, REG_{MUX} = 0, R_M = 1, B_{MDRIN} = 1, W_{MDR} = 1$
12. `(MAR)` ← `MDR`
    - $B_{AB} = 1$
    - $B_{AB} = 1, MWR = 1$
    - $B_{AB} = 1, MWR = 1, B_{DBOUT} = 1$

The fourth $\mu$-op uses the index register. As we have already discussed, the content of the index register can be used only on a 64-bits basis. This is because the machine instruction format does not have any space to specify what the size of the index register is, namely the *virtual* register to be used in order to perform the operation. Therefore, the CU, to account for this, cannot use any field from IR to tell the registers bank to access the whole 64-bits index register. In this case, $S_{MUX}$ is set to zero, which gives the hard coded value `11` as the size of the operand. This is interpreted by the combinational logic as if SS (or DS) from the Mode byte where specifying a quadword size, thus the network accesses the whole 64 bits of the register specified in the Index Register field of the SIB byte, which is selected setting $REG_{MUX}$ to 2.

The other $\mu$-ops are quite straightforward, as they mostly entail only moving data from one internal component to the other. It is interesting to note, anyhow, that when it comes to arithmetic/logical operations, such as the ADD required to accumulate the final address, the CU has to correctly set the opcode of the ALU to select the required operation, among the various supported by the circuit. This is the case, for example, of $\mu$-ops 7 and 10. The opcode passed to the ALU, for the case of an add, is 0000. If we refer to Figure 5.12, where the opcode for the `add` instruction was provided, we can see that the Type field for this instruction is exactly 0000. Therefore, when dealing with arithmetic/logical operations, the CU can simply pass the Type field of the instruction to the ALU, making the implementation of the finite-state machine much simpler.

The particular instruction that we have been using as an example in this case uses a memory operand as the destination. Therefore, the last $\mu$-op must access memory in write mode. This operation is carried out in a way which is very similar to that of the fetch phase, requiring three clock cycles to complete. The main difference lies in that in this case, rather than setting $MRD$, the $MWR$ control signal is set, so that the memory bank knows that it has to copy the content of the data bus into the memory cell(s) identified by the address stored in MAR.

Another common operation is related to copying constants into registers (or directly to memory). As we have already discussed, the location of immediate data depends on its size. In fact, if immediate data are up to 32-bits, they are directly encoded in the Short Immediate/Displacement field (of course, if no displacement is used by the addressing mode). On the other hand, if they are 64-bits wide, they are stored in the quadword immediately after the current instruction. Let's look at the $\mu$-ops generated by the CU in case of "small" immediate data, considering the following instruction:

```
movl $0xaaaa, %eax.
```

In this case, the CU executes the following microprogram:

1. MAR ← RIP
2. MDR ← (MAR); RIP ← RIP + 8
3. IR ← MDR
4. EAX ← IR[0:31]
   - $S_{MUX} = 2, REG_{MUX} = 1, W_M = 1, B_{SHORT} = 1$

It is interesting to note that only one additional $\mu$-op is required to copy the short immediate into the destination register. This is because the data are already available into IR, after the fetch phase. Let us now compare the execution of this instruction with the slightly different:

```
movq $0xaaaa, %rax
```

where only the size of the immediate operand has been changed. In this case, the CU executes the following microprogram:

1. MAR ← RIP
2. MDR ← (MAR); RIP ← RIP + 8
3. IR ← MDR
4. MAR ← RIP
   - $B_{rip} = 1, W_{MAR} = 1$
5. MDR ← (MAR); RIP ← RIP + 8
   - $INC_{RIP} = 1, B_{AB} = 1$
   - $B_{AB} = 1, MRD = 1$
   - $B_{AB} = 1, MRD = 1, B_{DBIN} = 1, W_{MDR} = 1$
6. RAX ← MDR
   - $S_{MUX} = 2, REG_{MUX} = 1, W_M = 1, B_{MDROUT} = 1,$

Note that, in this latter case, an additional memory access is required, which is in all similar to the fetch phase. In fact, the CU has to retrieve the data from memory, exactly at the address pointed by RIP. Nevertheless, the destination of this memory access is no longer IR, as the accessed information is not an instruction, rather the destination register (RAX, in our example). This is reflected by the fact that $\mu$-op 6 does not enable $W_{IR}$, rather it exploits the information already stored into IR to determine what is the destination register.

As we have seen, there are several different possibilities to execute a data movement instruction, depending on the nature of the operands. Yet, the microprogram for this instruction must be only one, as the CU jumps to a microprogram depending only on the class of the instruction. In fact, as we have discussed, the microprogram can internally jump to different branches, depending on several condition variables which are read from the state of the CPU, including specific bits in the IR register. Therefore, let's call `B` the encoding of the generic base register, `I` the encoding of the generic index register, `T` the generic scale, and `dest` the generic destination register of the computed address. The final parametric microprogram to execute a `mov` instruction with the source operand in memory becomes the following:

```
1  if D == 1 and B_p == 0 and I_p == 0
2      MAR ← IR[0:31]
3  else if D == 0 and B_p == 1 and I_p == 0
4      MAR ← B
5  else if I_p == 1
6      TEMP2 ← I
7      MAR ← SHIFTER_OUT[SHL, T]
8      TEMP1 ← MAR
9      if D == 1
10         TEMP2 ← IR[0:31]
11         MAR ← ALU_OUT[ADD]
12         TEMP1 ← MAR
13     endif
14     if B_p == 1
15         TEMP2 ← B
16         MAR ← ALU_OUT[ADD]
17     endif
18 endif
19 MDR ← (MAR)
20 dest ← MDR
```

At a first glance, we could say that there is some redundancy here. For example, in case we have both an index register and a displacement, we first copy a partial address to MAR, then back to TEMP1. Then, after the addition with the displacement, we copy the partial address first to MAR, and then back to TEMP1. In this case, the $\mu$-ops at lines 7–8 could be condensed into a single one, and the $\mu$-op at line 12 is useless. Nevertheless, we should empasize that this microprogram works *independently* of the parameters of the addressing mode: given any combination, the final address is always stored into MAR, to access memory.

This is a non-negligible optimization: in fact, there are many operations which access memory—not only `mov`s. Thus, we can modify the flow of the states of our CU so that, whenever we have to compute a memory address, we can *call* a microprogram which only does this, and always places into MAR the computed address. We only have to place in the ROM of our CU two versions of this mi-

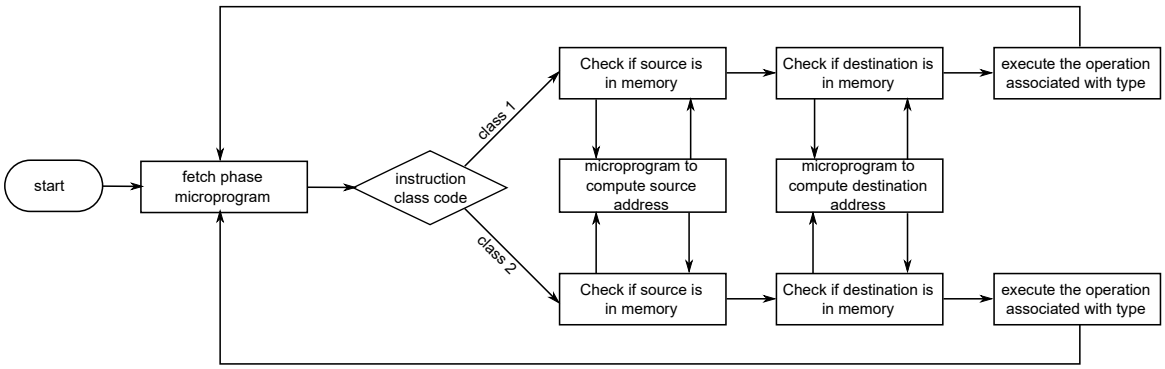



**Figure 5.49:** A callable microprogram to compute the final address of a memory operation, as used by Class 1 and Class 2 instructions.

croprogram: one to deal with source operands, one to deal with destination operands, as these two versions use different values for the $REG_{MUX}$ and $S_{MUX}$ control variables. To have this implementation working, as mentioned before, the CU must be able to keep track of what microinstruction called the microprogram to compute the memory address. This can be done by introducing a special-purpose register, which we call the *return register*, to track the execution of the microprogram, and a special microoperation which "calls" a subprogram, copying the value of the base and offset registers as shown in Figures 5.42 and 5.43 in the return register. The called microprogram should then finish with a new "return" microoperation, which restores the content of the offset and base registers, taking the old value from the return register.

In the z64 CPU, this allows to significantly reduces the size of several microprograms, and therefore to reduce the overall space used by the ROM. In fact, several instructions from Class 1, 2, and 3 might require to compute source and destination memory addresses. Figure 5.49 shows the organization of the microcode, focusing only on Class 1 and 2 instructions.

There are some instructions, which will be discussed later in this book, which can alter the flow of the program. One of them is the `jmp` instruction, which sums to the value of RIP after the fetch phase the content of the Displacement field of the instruction. This instruction is represented in assembly code as:

```
jmpq M
```

where `M` can be an absolute address or a certain "bookmark" in the code, which is called a *label*. Therefore, the programmer writes directly an address, and then the assembler transforms it into a displacement. The fact that the displacement is added to the value of RIP *after* the fetch phase is a trick to simplify the generation of the microcode. In fact, even if the assembler has to precompute the displacement, this does not add any additional runtime overhead. Instead, having to account for the size of the instruction at runtime does add an overhead. In this case, the microprogram executed by the CU in case of a `jmp` instruction is:

1. `MAR` ← `RIP`
2. `MDR` ← (`MAR`); `RIP` ← `RIP + 8`

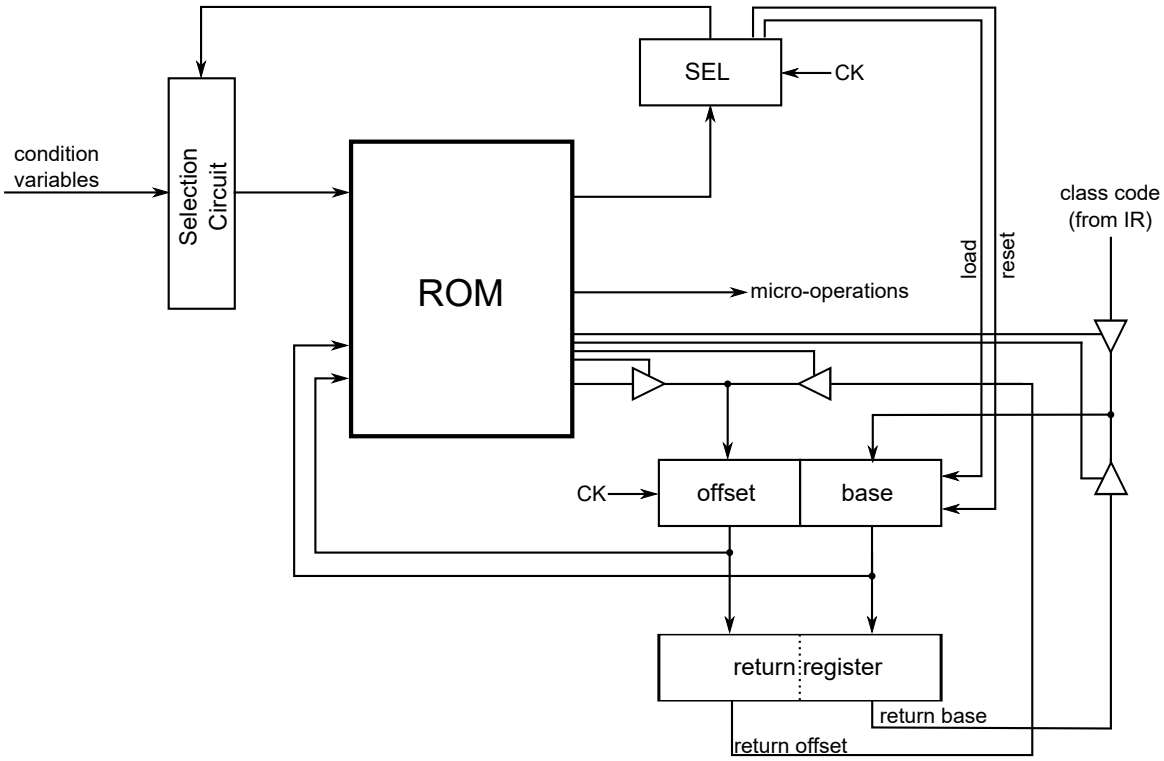



**Figure 5.50:** Modification of the Mealy Machine implementing the CU to allow calling one subprogram in the microcode.

3. `IR` ← `MDR`
4. `TEMP1` ← `RIP`
   - $B_{RIP} = 1, W_{T_1} = 1$
5. `TEMP2` ← `IR[0:31]`
   - $B_{SHORT} = 1, W_{T_2} = 1$
6. `RIP` ← `ALU_OUT[ADD]`
   - $A_{opcode} = 0000, W_{RIP} = 1$

## Questions and Exercises

### Exercise 5.1 (*General Questions*):

1. What information can be stored into temporary registers?
2. How many control signals are necessary to tell the ALU what operation it has to execute?
3. What are the steps required to execute a unary operation?
4. Given the diagram in Figure 5.31, how many bytes can be addressed?

5. The content of the SEL register in Figure 5.42 is referencing the current or the next $\mu$-op?
6. What is the purpose of CK in the diagrams in Figures 5.42 and 5.43?
7. What is the purpose of the load and reset signals shown in Figures 5.42 and 5.43?
8. Look at Figure 5.46. If the MRD signal is set active high during the first clock cycle, could it be possible to correctly read the data?

**Exercise 5.2 (*SR flip-flop based registers*):** Draw the schematic diagram of a register composed of SR flip-flops. Then, design the interconnection between two SR flip-flop registers.

**Exercise 5.3 (*Updating status flags*):** Draw the schematic diagram of a combinational network which computes the value to update `OF` according to the two rules described at page 89, where the sign bits of the operands and the sign bit of the result must be combined to determine the new value of `OF`.

**Exercise 5.4 (*Internal data transfer*):** Look at the diagrams in Figure 5.6 and 5.7. How many registers can be updated concurrently in the two different organizations? Design the diagram of a solution which allows to transfer the content of two registers in parallel.

**Exercise 5.5 ($\mu$-*ops*):** Given the PU architecture shown in Figure 5.17, enumerate the $\mu$-ops (including the fetch phase) generated by the CU to:

- transfer data from R8 to R10;
- sum a constant value 10 to memory address $(AABB)_{16}$;
- shift two positions to the left the content of EAX;
- add the content of the word stored in memory at address $(AAAA)_{16}$ to the content of BX.

**Exercise 5.6 (*Memory organization*):** Design a possible organization of a main memory composed of four 1-byte memory banks, where it is possible to retrieve only one byte at a time. Discuss as well the design choices made to let the access be executed as much efficiently as possible. Additionally, assuming that one flip-flop is square, and large $10^{-17}m$, compute the total size of the memory.

# 6 Assembly Programming

In this chapter, we will explain how to implement assembly software to be run on the z64 processing unit, as described in chapter 5. Assembly programming is a form of art, which requires the software developer to reason as the machine, in order to produce code which is efficient both from a performance and a memory-usage perspective.

When programming in assembly, instructions have a 1:1 mapping with binary words that the processing unit is able to interpret, and which in turn activate internal signals for data transfer and/or processing. Given this direct mapping, a carefully-written assembly routine could run much faster than the same routine written in a higher-level language, although modern compilers often try to do their best at optimizing the output code, often producing very impressive results.

We assume that the reader is already fluent in higher-level programming languages like C, and we will often show higher-level code snippets, to illustrate how various operations could be implemented.

Each assembly language has its own syntax. z64 assembly code uses the AT&T syntax[1], which generally expresses instructions in the form:

```
mnemonic source, destination
```

where `mnemonic` corresponds to one of the assembly instructions, which are listed in Appendix A. On the other hand, `source` and `destination` could be either a register, or a memory location, or even an immediate data, depending on the actual meaning of the involved instruction.

An assembly program consists of a set of instructions which operate on data. These instructions are actually translated to their corresponding binary form by a program, which is called the *assembler*. In order for the assembler to know how to properly set the size field of the instruction, a *suffix* should be added at the end of the instruction's mnemonic. For example, considering the `add` instruction, if we want to add the content of the `%eax` register to `%ebx`, explicitly specifying that we are interested in executing a 32 bit add operation, we will write:

```
addl %eax, %ebx
```

[1] AT&T Syntax is used as well by real-world assemblers, like GNU `gas`. Therefore, once the student becomes fluent in z64 programming, he will be able to easily implement software on real Intel-compliant CPUs, using the same syntax.

**Table 6.1:** z64 instruction suffixes

| Suffix | Operand Type | Bit Width |
|---|---|---|
| `b` | Byte | 8 bit |
| `w` | Word | 16 bit |
| `l` | Long | 32 bit |
| `q` | Quad-word | 64 bit |

On the other hand, if we want to perform a 16-bit add operation between the `%ax` and the `%bx` registers, we will write:

```
addw %ax, %bx
```

In Table 6.1 the four suffixes associated with 1-byte, 2-bytes, 4-bytes, and 8-bytes operations are shown. If the suffix is not specified, and there are no memory operands for the instruction, the operand size is inferred by the assembler from the size of the *destination* register operand, so that, for example, the following two instructions are perfectly equivalent:

```
addl %eax, %ebx
add %eax, %ebx
```

It is interesting to note that some operations might require an additional suffix, to allow the assembler to properly fill all the required instruction's fields. This is the case, for example, of *sign extension* instructions, where the processing unit is required to transform some data from one format to another. The `movsX` instruction tells what conversion should be done[2], so two suffixes are required, telling which is the source size and the destination size. If, for example, we want to take a 16-bits value from the `%ax` register, and move it to the `%ebx` register, we will write:

```
movswl %ax, %ebx
```

The `movsX` instruction can be used as well to make *in-place* sign extension, simply having the same register as source and destination operands, but using their different names specifying different sizes:

```
movswl %ax, %eax
```

This instruction takes the 16-bits value in `%ax` and simply sign-extends it to 32. Of course, this involves the activation of various control signals in the CPU, and the data actually move from the accumulator and then go back in there, but we can say that this instruction *virtually* makes an in-place extension.

Various combination of sizes can be used when extending the sign, provided of course that the destination size is bigger than the source's. To illustrate them, all the instructions shown in Table 6.2 are valid. The same discussion holds as well for the `movzX` instruction, which performs the same action but zero-extending the value.

The counterpart `mov` instruction, on the other hand, requires just one suffix to specify the size of the *destination* operand. If the source operand is of a smaller size, then the behaviour of the instruction is to zero-extend its value. For example, the following instruction:

[2] It is not possible, with this instruction, to have as a destination operand a memory address.

**Table 6.2:** Sign extension suffixes

| Instruction | Type of conversion |
|---|---|
| `movsbw %al, %ax` | Sign extend from byte to word |
| `movsbl %al, %eax` | Sign extend from byte to longword |
| `movsbq %al, %rax` | Sign extend from byte to quad-word |
| `movswl %ax, %eax` | Sign extend from word to longword |
| `movswq %ax, %rax` | Sign extend from word to quadword |
| `movslq %eax, %rax` | Sign extend from longword to quadword |

```
movl %ax, %eax
```

takes the 16 least significant bits from the accumulator, and zero-extends them to reach a data width of 32 bits. The remaining 32 more significant bits are not affected by the execution of this instruction. Therefore, when writing assembly code is important to pay attention to what operations were executed *before* accessing a register. Consider the following example, where two consecutive `mov` instructions are used to store data into `%eax` register:

```
movq $2048, %rax
movb $128, %al
```

The first instruction saves the value 2048 to the register, with zero extension up to 64 bits, and therefore assumes the following binary value:

| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Then, the execution of the second instruction zero-extends the value 128 up to one byte, stores it in the least significant 8 bits of the register, and leaves its value as:

| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

If later in the program the instruction `movl %eax, %ebx` is executed, the processing unit takes the least significant 32 bits from the accumulator, and moves them to the base register. The user might think that the value 128 has been copied to `%ebx`, rather after the execution of that instruction it will contain the value 2176, which is actually the or'ed value of 128 and 2048.

In z64 assembly, some characters have a special meaning. As already shown, whenever a register is accessed, its name must be preceded by the "%" symbol, otherwise the assembler will generate a syntax error. To place comments in the code, the symbol "#" must be used. In particular, a comment starts with a # and lasts until the end of the line is reached, and multi-line comments are not allowed.

Every *constant number* must be prefixed with a "$". Numbers can be expressed in decimal or hexadecimal notation. To distinguish between the different notations, the assembler looks at the initial characters representing the data, so that a number starting with `0x` is interpreted as a hexadecimal number, otherwise it is considered to be in decimal notation. On the other hand, if a number is found without the leading `$`, it is considered to be a *memory address*. Then, the following two instructions have a very different result:

```
movl $0xAABB, %eax
movl 0xAABB, %eax
```

The former instruction stores into `%eax` register *exactly* the value `0xAABB`, while the latter goes into memory and copies into register `%eax` the first 4 bytes found starting from address `0xAABB`.

## 6.1 Structure of a Program

As mentioned, an assembly program consists in a set of instructions which operate on data. Therefore, when implementing an assembly program, it is necessary to define as well the data on which the instructions will operate, similarly to what happens in any high-level language.

Nevertheless, there is a distinction in the way data are declared in an assembly program. Let's refer to Example 6.1.1, where a simple C program to compute the squared value of an integer declares data between two different functions. We could think that the actual data referred from variable `value`, declared at line 7, is located between the machine instructions representing the functions `square_int` and `main`. Rather, in real-world programs, this is never the case. This is due to various reasons, among which *sections protection* is very important. In fact, different memory regions of a program have different *privileges*, meaning that operations that can be executed on data depend on *where* they are located.

**Example 6.1.1:** Declaration of Variables in C

```
1  #include <stdio.h>
2
3  int square_int(int x) {
4      return x*x;
5  }
6
7  int value = 10;
8
9  int main(void) {
10     printf("%d\n", square_int(value));
11     return 0;
12 }
```

This becomes evident if we think that any data is represented in memory as a concatenation of binary words. If we look at the whole memory as a contiguous tape, we cannot make any difference between an integer number and an instruction: they are both a *sequence of bits*. What makes the

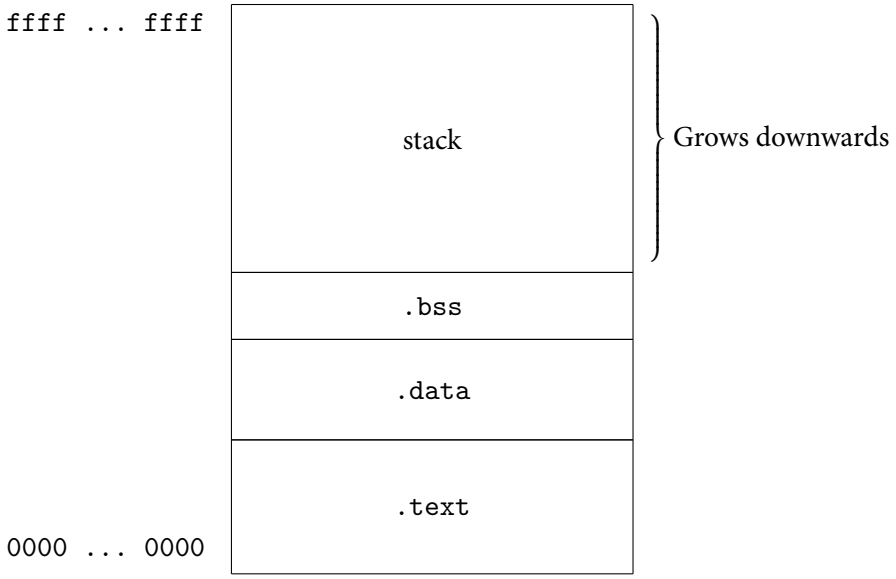



**Figure 6.1:** Map of a Program's Memory

difference between them, is how the processing unit has to interpret them. In fact, it looks awkward to say than an integer could be executed, or an instruction could be added to another. There is therefore a clear distinction between what the processing unit can do with the sequences of bits that it finds in memory, and this is reflected into the *privileges* associated with memory regions.

It is necessary to define permissions on large memory regions: it wouldn't be memory-efficient to set different permissions, for example, on a word basis, as the memory consumption would be too high. Therefore, similar data are clustered in contiguous memory regions, which are then given the same permissions. In this scenario, a memory region keeping data would be set as *readable* and *writeable*, but not *executable*, while a memory region containing instructions would be set as *readable* and *executable*, but not *writeable*.

This is reflected in the structure of a program, as shown in Figure 6.1, which is divided into different *sections*, each one containing homogeneous information (like data, or instructions). This structure, which is usually called *executable format*, could not be unique across different computing architectures and/or operating systems, although in real architectures few of them have become a de-facto standard. On the z64 architecture, since the whole memory is dedicated to a single program, the used executable format is simply a concatenation of the sections, which are:

- `.text`: it contains the instructions of the program. This is a section which is usually set to be executable and accessible in read mode only;
- `.data`: it contains all the variables used by the program which are initialized to a value different from zero;
- `.bss`: it contains all the variables used by the program which are initialized to zero;
- `.stack`: it contains temporary data instantiated/used by the program, and therefore its size can grow or shrink, depending on the program flow. The last element placed on the top of the stack is referred by the RSP register. Since at the beginning of any program the stack is empty, the RSP

register is initialized with the address `ffff ffff ffff ffff`. When a new element is placed into the stack, RSP is *decremented*, pointing to a lower address. The stack is then said to grow downwards, towards the address `0000 0000 0000 0000`.

Modern computing architectures do not allow a program to directly access memory, as they make the distinction between *physical memory* and *virtual memory*. Program only manipulate virtual memory addresses, which are the same for all of them, meaning that, for example, at the same virtual address one program could store an instruction, while another could store some data. It is then the role of both the CPU firmware and the operating system to translate a virtual address into a physical address, and therefore reach the correct memory location.

**Historical Note 6.1.1:** `.bss` section

The name `.bss` means "Block Started by Symbol", and was a "pseudo-operation" of an assembly language developed in the mid-1950s. This language was used by a software called UA-SAP (United Aircraft Symbolic Assembly Program), which was developed mainly by Roy Nutt and Walter Ramshaw to program the IBM 704, the first large-scale IBM computer equipped with a floating point unit, which was introduced into the market in 1954. It was essentially one of the first assemblers, which allowed programmers to implement their programs without having to write manually machine code, thus being much faster at implementation and making software more readable and maintainable.

Roy Nutt in 1953

Roy Nutt introduced in the UA-SAP a set of pseudo-operations, which were essentially mnemonics with no one-to-one mapping to machine instructions. The goal of the `BSS` pseudo-operation was to reserve a block of uninitialized space associated with a particular label (i.e., a symbol) for a given number of words. It was therefore a shorthand to avoid the cumbersome task of manually reserving a number of separate smaller data locations. Some other pseudo-operations which survived to time where `ORG`, abbreviation of "origin", defining the starting location in memory of the program's first executable instruction, `EQU` for "equating" a symbol name to an expression which could be later used in place of the expression itself.

In an example from the IBM 704 COMPUTER programming manual at MIT we found the original usage of the `BSS` pseudo-operation:

```
        ORG 1000
COMMON BSS 60
START  CLA COMMON+5
```

According to this example, the symbol `COMMON` is assigned the value 1000 and `START` is assigned the value 1060. Therefore, using the `BSS` pseudo-operation, the programmer is able to define, like in this case, an array of 60 bytes in a single instruction.

The reason why the `.bss` section has preserved this name is that in the program image on the hard drive (in many executable formats) this section occupies no space at all. When the operating system

then loads it to make it run, it allocates a contiguous space of zero-filled bytes, thus initializing all the symbols to zero, similarly to what the original BSS instruction was thought for (although, on IBM 704, the memory was not initialized to zero).

## 6.2 Assembly Directives

An assembly program involves more than instructions, as they have to deal with data. In general higher-level prorgamming, different types of data can be used, namely global variables or local variables. This distiction must be somewhat present as well in assembly programming. Moreover, each instruction and each piece of data is located at a certain address, so an assembly program must have the possibility to handle this.

Indeed, despite the fact that assembly instructions have a 1:1 mapping with machine instructions, there is some "knowledge" that the programmer relies on when implementing his program. In fact, a program must access data, either initialized or uninitialized, in order to support the execution flow of the program itself. As mentioned earlier, a program has a fixed structure, so data and instructions are located in separate parts of the source. The final mapping between instructions manipulating data and the data themselves is handled by the *assembler*, yet the programmer must have a way to instruct the assembler on how to properly *emit* symbols and instructions while generating the final executable.

This is done by relying on *assembly directives*, which are keywords reserved by the language or defined by the programmer, which are not associated with machine code instruction, rather they give additional information on what will follow in the program. Some directives are associated with variables internal to the assembler, others drive some state change within the assembler, others allow to create *symbolic names* withing the program. In some cases, a directive might have one or more arguments, which are represented as a comma-separated list of values.

**Labels.** A *label* is a textual mnemonic which is defined by the programmer and allows to keep track of the address of the following element. It is essentially a series of ascii characters followed by a column:

```
this_is_a_label:
```

For example, if the programmer wants to define a procedure, this is usually referred using a symbolic name within a `call` instruction, which we will discuss later. To define the name of the procedure, a label should be used in first place, placing the first instruction of the procedure right after it:

```
procedure:
    movq $0, %rax
```

A label can be used as well to define placemarks in the code of a given procedure, for example to implement loops. Although this is not necessary, a good practice is to name labels which are not defining procedures (and are therefore only used for internal code referring) starting with a dot. Therefore, an infinite loop could be implemented as follows:

```
.loop:
    jmp .loop
```

Labels can be used as well to create symbolic names for variables. The situation is analogous to that of procedures, meaning that a label placed before a data declaration will be associated with the address of the immediately following data.

**Location Counter.** An essential assembly directive is the *location counter*, which is an assembler's variable keeping track of the address at which data are placed and instructions are emitted. It is represented using a single dot (`.`), and whenever something in inserted in the binary representation of the program, be it an instruction or a piece of data, its value is increased of the amount of bytes consumed by the newly-generated element. By relying on the location counter, the assembler knows what part of the assemblying job it has done already.

In some circumstances it is necessary to arbitrarily set its value, for example to ensure that a given variable is located at a particular address. This can be done by explicitly setting the value of the location counter. For example, this statement:

```
. = 0x8000
```

tells the assembler that what will follow that line (be it, again, an instruction or some data) must be placed at *address* `0x8000`.

The location counter is useful as well to compute the size of a certain data structure statically. For example, in the following code, a text string is created—the involved directive is described later—and the location counter is used to define a constant value keeping the length of the string:

```
msg:
.ascii "Hello, world!\\n"
len = . - msg
```

The goal of the location counter is to skip portions of the address space, and therefore its value can only be increased. Attempting to set the location counter to a value which is smaller than the current one will generate an error. This is a conservative behaviour, to ensure that no previously-emitted variables or instructions are overwritten by erroneously setting the location counter to an address which has already been taken by data or instructions.

**.org address, fill.** This directive is an alternate way of moving the location counter. Therefore, the following two statements are perfectly equivalent:

```
. = 0x8000
.org 8000
```

On the other hand, using the `.org` directive the optional `fill` argument can be specified, telling the assembler that starting from the current value of the location counter to the new one, a series of bytes with `fill` value should be emitted.

**.equ symbol, expression.** This directive sets the value of `symbol` to `expression`. After this line, `symbol` can be used as a constant number/address in any instruction. The assembler replaces its symbolic name with its value.

In place of `.equ`, a flat assignment like the following one can be written:

```
symbol = expression
```

Note that when using both `.equ` and the flat assignment, the same `symbol` can be redeclared in different parts of the code.

**.byte expressions.** This directive is used to reserve memory for `expressions`, which can be one or more, separated by commas. It is useful to define single byte variables, or arrays of bytes, like in the following example:

```
var:   .byte 0
array: .byte 0, 1, 2, 3, 4, 5
```

**.word expressions.** This directive is used to reserve memory for `expressions`, which can be one or more, separated by commas. It is useful to define single word variables, or arrays of words, like in the following example:

```
var:   .word 0
array: .word 0, 1, 2, 3, 4, 5
```

**.long expressions.** This directive is used to reserve memory for `expressions`, which can be one or more, separated by commas. It is useful to define single long variables, or arrays of bytes, like in the following example:

```
var:   .long 0
array: .long 0, 1, 2, 3, 4, 5
```

**.quad expressions.** This directive is used to reserve memory for `expressions`, which can be one or more, separated by commas. It is useful to define single quadword variables, or arrays of bytes, like in the following example:

```
var:   .quad 0
array: .quad 0, 1, 2, 3, 4, 5
```

**.ascii "string".** The `.ascii` directive is used to store character strings in memory. By definition, a string is a sequence of characters followed by a string terminator null-byte. `.ascii` assembles each string into consecutive addresses.

**.fill repeat, size, value.** This directive can be used to fill memory regions with particular values. In particular, it emits `repeat` copies of `size` bytes, each one set to `value`.

`size` and `value` are optional. If the second comma and `value` are absent, `value` is assumed zero. If the first comma and following tokens are absent, `size` is assumed to be 1.

**.text.** This directive tells the assembler to assemble the following statements onto the end of the text section.

**.data.** This directive tells the assembler to assemble the following statements onto the end of the data section.

**.comm symbol , length.** The `.comm` directive declares a named common area in the bss section. This is an area which is `length` bytes long, and is associated with the symbolic name `symbol`.

**.driver idn.** Using this directive, a pointer to the next instruction is placed in the Interrupt Descriptor Table (IDT) in the corresponding `idn` entry, so that a hardware or a software interrupt associated with this number will be served by the following code. A complete description of the interrupt system will be given in Chapter 7.

**.handler idn.** This directive is associated with the same operation of **.driver**. It is present to allow the programmer to differentiate between interrupts generated via hardware or via software, as it will explained in Chapter 7.

## 6.3 Building Blocks of an Assembly Program

To start writing an assembly program, a minimal set of directives should be used, which are shown in example 6.3.1.

**Example 6.3.1:** Skeleton of a Program

```
1 .org 0x800
2
3 .data
4     # declare and initialize data
5
6 .text
7
8     # Program body
9     hlt  # To halt the execution of the program
```

As mentioned, `.org` allows to load the program at a specific memory address. It is important to note that the assembler will return an error if the initial value of the location counter is smaller than

0x800, because the initial part of memory is reserved. The reasons behind this are related to the way the z64 CPU interacts with I/O devices, and will be thoroughly discussed in Chapter 7.

The `.data` statement will allow to declare and initialize data, by using a combination of `.byte`, `.word`, `.long`, and `.quad` directives. As mentioned, to declare variables initialized to zero, the directive `.fill` can be used[3], while to reserve memory for strings the `.ascii` directive can be used after `.data`.

On the other hand, since `.equ` does not reserve any space, it can be used in any place of the program, since it is simply handled by the assembler. Of course, a constant must be declared using `.equ` prior to using it.

**Example 6.3.2:** Hello World

```
1  .org 0x800
2
3  .data
4
5  message: .ascii "Hello World!"
6  counter: .byte 0
7
8  .text
9  main:
10     movq $message, %rax
11     .repeat:
12     cmpb $0, (%rax)
13     jz .end
14     addb $1, counter
15     addq $1, %rax
16     jmp .repeat
17     .end:
18     hlt
```

A more substantial example is given in Example 6.3.2. Although the involved programming techniques will be discussed later, this *Hello World* example allows us to introduce some fundamental programming constructs. Since at this stage it is not possible for us to print the "Hello World!" string on screen, this simple program simply calculates the number of characters composing it.

Two different variables are declared: a text string and a byte counter (which is initialized to zero), which are associated with the symbolic names `message` and `counter`, respectively, by using two labels in the `.data` section. Then, in the `.text` section, the `main` program is declared. It is important to note that the execution of the program always starts from the `main` program, and therefore failing to declare it in the code will result in the impossibility by the processor to execute it.

On line 10, the address of the `message` string is loaded into the RAX register. We stress that if line 10 were reading `movq message, RAX`, after its execution the RAX register would be loaded with the value 'H', which is the first byte in the string (namely, the first character). On line 11, a label local to the main program is declared (note that its name starts with a dot). This will be used to implement a

[3] Remember that using `.fill` the data is placed in the `.bss` section rather than in `.data`, and therefore the usage of `.fill` overrides the previous `.data` statement.

loop, to scan through the string and sum up the number of characters.

Line 12 has a `cmpb` instruction, which makes a comparison between the first and the second operand by subtracting from the second one the first one, and setting the status flags in the FLAGS register accordingly. The second operand uses a base address to retrieve one byte (since the size suffix of the instruction is `b`) from the memory location stored into RAX which, as discussed, points to the first character of the string. Since the first operand of the instruction is zero, the `cmpb` instruction sets the status bits depending on the current value of the character being pointed by RAX.

The following instruction is `jz`, which performs a jump to the address associated with the `.end` local label only if ZF in FLAGS is set. Since the previous `cmpb` instruction subtracted zero from the second operand, the ZF is set if and only if the second operand's value is zero as well. By the definition of the string datatype which was given in Section 2.3, every string is terminated by a null-byte, and therfore the branch in the instruction on line 13 is taken only if a string terminator is currently pointed by RAX.

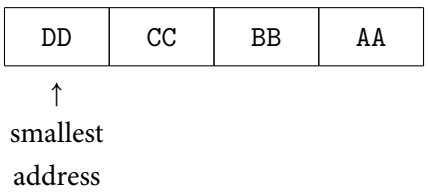
If the branch is not taken, the control passes to the instruction on line 14, which simply adds 1 to the current value of `counter`. Then, the following instruction adds 1 to the current value of RAX, making it point to the next character in the string. Note that the two `add` instruction use different size suffices, as `counter` is a 1-byte variable and RAX is a 64-bits register.

Line 17 finally implements the loop which closes the cycle with the previous declaration of the `.repeat` label, transferring back the control to line 11. Here, the `cmpb` instruction finds a different value of RAX from the previous iteration, and can therefore check whether the next character is a string terminator or not. Eventually, due to the fact that the string was declared using the `.ascii` directive (which ensures the presence of the string terminator), the branch on line 13 will be taken, making the CPU execute the `hlt` instruction on line 18, which terminates the execution of our "Hello World!" example.

We note that in this example we have only used the accumulator register, which as mentioned is commonly used to perform operation on data. Although perfectly correct, to be a bit more consistent, this program could have been written relying on the counter register to count the current character being processed, making line 12 read, for example, `cmpb $0, $message(, %rcx, 1)`.

### 6.3.1 The effect of Endianness on Data Declarations and Access

The z64 CPU addresses memory using little-endian byte ordering, and therefore multibyte values are stored with their least-significant byte at the lowest byte address. Therefore, if we consider the single 32-bits hexadecimal value `0xAABBCCDD`, its memory layout is:




Therefore, when using assembly directives to reserve space for data, the size of the declaration

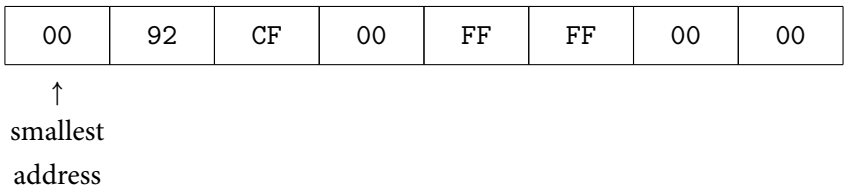
matters the memory representation. For example, consider the following declaration and its corresponding memory representation:

```
.long  0x00cf9200 = 00 92 CF 00
```

This is a classical little-endian memory representation, similar to the previous example. The difference arises when we rely on the concatenation of different values to declare an array:

```
.long  0x00cf9200, 0x0000ffff
```

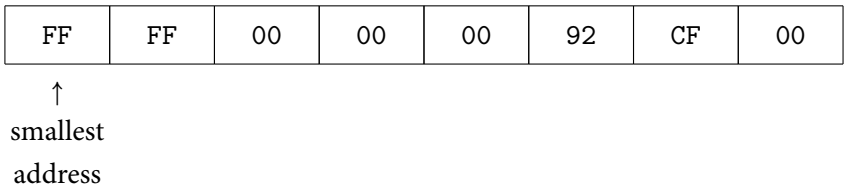
In this case, each value is considered separate from the others, since we are telling the assembler that we are interested in *two* long values. Therefore, the memory representation of this array will be:




meaning that the bytes of *each* value are swapped, but the array is not considered as a single data. Then, there is a strong difference between the following two declarations:

```
.long  0x00cf9200, 0x0000ffff
.quad 0x00cf92000000ffff
```

In fact, although they look similar, the representation in memory for the quadword is:




This is because we are telling the assembler that our second value is *one* quadword, and therefore it expects the CPU to load it as a whole. Since memory is addressed using little-endian byte ordering, all the bytes will be swapped. On the other hand, using again the same values, we can see that the declaration

```
.long  0x00cf9200, 0x0000ffff
```

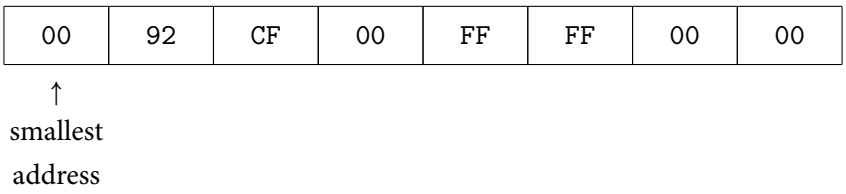
is equivalent to

```
.quad  0x0000ffff00cf9200
```

which is in turn equivalent to

```
.long  0x00cf9200
.long  0x0000ffff
```

as they are all represented in memory as:




When writing an assembly program, it is important to take care of the memory representation of data. In fact, assuming that two consecutive longwords can be *concatenated* into a quadword by simply reading 8 bytes starting from the address of the first longword will lead to an incorrect result.

While little-endianness could be regarded as an awkward choice, its effectiveness comes easily to mind when thinking of *casts*. When converting a primitive value stored in memory to a smaller data size, higher-order bytes are discarded. In case of little endianness, we have that the least-significant bytes come first. Therefore, when casting a value stored in memory to a smaller data size, the address at which che CPU has to perform a memory access does not change: only the number of bytes read changes. This allows for a much more efficient design of the CPU, which must not perform complex calculations to determine where to perform a read, given a certain memory address.

### 6.3.2 Comparing Data

In Example 6.3.2 we have seen that using the `cmp` instruction we can check whether a given value is zero or not. This is part of a more general technique which is used to compare data. In particular, regarding integer numeric representation of data, it is possible to compare any given two number to assess, for example, if one is bigger than the other.

This is of course again done by relying on status bits, since the only instruction which can be used for such comparison is `cmp`. Nevertheless, we must make a difference between signed and unsigned representations of integers, since the same bit string can represent either a very large number or a negative one.

#### Comparing Unsigned Integers

As mentioned, `cmp` subtracts the first operand from the second one, discarding the final result. When dealing with unsigned integers (thus we assume that the first and the second operands are both $\geq 0$) we should check the value of `CF` and `ZF` to determine the relations among the operands.

If we consider a generic instruction `cmpX source, dest`, the relations are defined according to what stated in Table 6.3. By recalling the discussion about how to properly set/clear status flags, which was given in Section 5.4, it is clear that for unsigned arithmetic only `CF` should be considered when comparing two different operands.

**Table 6.3:** Unsigned Arithmetic Comparison

| | Relation | First Check | Second Check |
|---|---|---|---|
| 1 | source > dest | CF = 0 | – |
| 2 | source ≤ dest | CF = 1 | – |
| 3 | source < dest | CF = 1 | ZF = 0 |
| 4 | source = dest | ZF = 1 | – |
| 5 | source ≠ dest | ZF = 0 | – |

To explain why the relations in Table 6.3 hold, we must first of all remember that cmp performs a subtraction of the source operand from the destination operand and then discards the result. Then, let us prove first of all the relations 1 and 2 by giving two examples. Considering two 4-bits integers, if we compare the values 2 (as the source) and 1 (as the destination), the CPU will compute:

```
0001 - 0010 = 0001 + 1110 = 1111
```

and CF is cleared. In fact, 2 is bigger than 1, and we can check for this condition by verifying that CF is set to zero. On the other hand, if we compare the values 1 (as the source) and 2 (as the destination) the CPU will compute:

```
0010 - 0001 = 0010 + 1111 = 0001
```

and CF is set. In fact, 1 is smaller than (*or* equal to!) 1, and we can check for this condition by verifying that CF is set to 1.

Then, relation 3 is a further specialization of relation 2. In fact, if we want to evaluate the strict inequality, we must first verify if source ≤ dest, and *then* verify that the operands are *not* equal. This is done by using a second cmp instruction and then testing whether ZF is zero, since this flag tells whether the result of an operation (a subtraction in this case) is zero, which is true only if the two operands are the same.

Similarly, relation 4 and relation 5 tell whether the operands are equal or not. This can be easily done by testing ZF after the cmp instruction.

There is no rule in Table 6.3 to check whether source is ≥ dest. This is because the rule is not unique, as it is the composition of relation 1 and relation 4. In fact, this comparison is done by checking whether CF = 0 *or* ZF = 1. On the other hand, it is possible to swap source and dest and evaluate relation 2.

A different way to check for inequalities comes exactly from the observation that cmp is actually a form of non-persistent subtraction. So, the following implications hold:

$$\texttt{dest} < \texttt{source} \Rightarrow \texttt{dest} - \texttt{source} < 0 \qquad \texttt{dest} > \texttt{source} \Rightarrow \texttt{dest} - \texttt{source} > 0$$

By the right side of the implication, we can clearly see that if the (discarded) result of the subtraction is negative, then source < dest, while if it is positive, then source > dest. This information is exactly stored in SF, which can be considered as an alternative condition to be tested for inequalities.

Finally, as it has been thoroughly discussed in Section 5.4, to determine whether an unsigned arithmetic operation has generated an overflow, CF should be checked. Depending on whether the operation is an addition or a subtraction, it should be tested for the value 1 or 0, respectively.

### Comparing Signed Integers

We have just seen that it is possible to use SF to discriminate on the result of a subtraction, given the mathematical fact that $B < A$ if and only if $B - A$ is negative. We could think that, since we are now moving to signed integers, it is even more correct to rely on this very same ploy to compare two values. Unfortunately, this is not always the case.

Suppose again that we are using 4-bits integers, yet this time they are signed. If we want to test whether $6 < -3$ by using SF, we compute:

```
0110 - 1101 = 0110 + 0011 = 1001
```

This result is negative and therefore sets SF, so we are lead into thinking that $6 < -3$ is true! What has happened here is that the result of $6 - (-3)$ should be 9, which is a number that cannot be represented in two's complement using 4 bits. In fact, an overflow has occurred. Thus, to generalize the rule which tells us to check whether SF is set to verify number comparisons, we realize that an overflow *always* changes the sign of a result, and so the general rule becomes:

1. If no overflow has occurred, then the inequality holds if SF is set.
2. If an overflow has occurred, then the inequality holds if SF is not set.

The status flag which tells us whether an overflow has occurred when dealing with signed operands is OF. To make the general rule more compact, we can say that the inequality holds if and only if OF $\oplus$ SF = 1.

Going back to the example $6 < -3$, we have there that since OF $\oplus$ SF = 0, the inequality does not hold, but we know that $6 > -3$. Then, since the `cmp` instruction subtracts the source operand from the destination, since this result was obtained by computing $6 - (-3)$, assuming that 6 is the destination and -3 is the source, we can state that if OF $\oplus$ SF = 0, then `source` < `dest`. Similarly, if OF $\oplus$ SF = 1, then `source` > `dest`. This result is not perfectly correct, as we are not actually taking into account equality—recall that this is done by checking ZF. Then, to refine the relations, we check whether OF $\oplus$ SF = 0—then `source` $\leq$ `dest`—or if OF $\oplus$ SF = 0—then `source` $\geq$ `dest`.

By this discussion, it now becomes clear that when dealing with signed integers (thus we assume that the first and the second operands are both represented using two's complement) we should check the value of OF, SF and ZF to determine the relations among the operands, and we can finally derive all the relations in Table 6.4.

Similarly to the case of unsigned arithmetic, to check for equalities and inequalities we must always check the value of ZF, which is also necessary to differentiate between strict and not strict inequalities. The main difference from the unsigned case is that in order to see if an operand is less than or equal (or greater than or equal) to the other, we must check two different status flags.

**Table 6.4:** Signed Arithmetic Comparison

| | Relation | First Check | Second Check |
|---|---|---|---|
| 1 | source ≥ dest | SF = OF | – |
| 2 | source > dest | SF = OF | ZF = 0 |
| 3 | source ≤ dest | SF ≠ OF | – |
| 4 | source < dest | SF ≠ OF | ZF = 0 |
| 5 | source = dest | ZF = 1 | – |
| 6 | source ≠ dest | ZF = 0 | – |

**Table 6.5:** Summary of comparison conditions

| Condition | Unigned Arithmetic | Signed Arithmetic |
|---|---|---|
| dest > source | C = 0 and Z = 0 | Z = 0 and N = V |
| dest ≥ source | C = 0 | N = V |
| dest = source | Z = 1 | Z = 1 |
| dest ≤ source | C = 1 or Z = 1 | Z = 1 or N ≠ V |
| dest < source | C = 1 | N ≠ V |
| dest ≠ source | Z = 0 | Z = 0 |

**Summary**

We have seen that in order to compare operands, the conditions to be verified are different when dealing with signed and unsigned arithmetic. This is because CPUs are agnostic with respect to the data representation used: they simply perform operations on bits, since the same string of bits can be associated with many different types of data. Then, it is the programmer's responsibility to tell how these bits should be interpreted, and in the context of comparisons this means that the programmer must use different instructions for conditional branches depending on what he is actually willing to do. In Table 6.5 we summarize the checks that should be performed in order to compare two different integers.

Intel processing units provide more advanced instructions to perform checks on the outcome of comparisons, which are nevertheless internally mapped to the same conditions which we have so far presented. In fact, there are actually instructions which should be used when dealing with signed arithmetic, and different ones for unsigned. For example, there are the `jb` (jump if below) and the `jl` (jump if less) which check whether the outcome of a compare states that the source operand is less than the destination in signed or unsigned arithmetic, respectively. In general, according to Intel, the terms "above" and "below" are associated with the `CF` flag and refer to the relation between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relation between two signed integer values.

Therefore, when using the instruction `jb`, the branch will be taken if and only if `CF` = 1, while the instruction `jl` will jump to the destination location only if (`SF` ⊕ `OF`) = 1, which is the exact same condition that we have derived in our analysis. Given this enhanced set of instructions, Intel

provides as well instructions which allow to take a branch checking for non-strict inequalities. For example, `jbe` (jump if below or equal) takes the branch if (`CF` or `ZF`) = 1. Of course, this is an unsigned arithmetic check, and to do the same when dealing with unsigned integers, the corresponding `jle` (jump is less or equal) should be used—this instruction, of course, takes the branch if ((`SF` ⊕ `OF`) or `ZF`) = 1.

If the program is run on the `z64` processing unit[4], comparing unsigned integers might be a bit quicker, as it requires checking for less conditions. Yet, the most important thing to keep in mind is that mixing signed and unsigned integers in comparisons could create errors which are tremendous to debug: a very large unsigned integer could "magically" become a negative signed value, when interpreted in a different way! High-level languages compilers often simply issue warnings for this, but the errors generated by this kind of comparison could unveil only rarely, for example depending on the input of a specific function. Then, when this happens, we might just skip to check that function, simply because it has "always" worked correctly so far. Additionally, the behaviour of an overflowing unsigned integer is always defined: adding one to the highest representable value leads to zero. In unsigned arithmetic, it depends on the underlying CPU and, in some cases, the compiler used. Therefore, some high-level code *assuming* that adding one to the highest value of a signed integer will give the smallest (negative) representable value, might miserably fail when moved on a different CPU.

## 6.4 Addressing Modes in Action

The simplest operation that can be performed by using this addressing method, is loading the value of a global variable into a register. This is done by relying on a `mov` instruction, where the source operand is a memory location using only the *displacement* variable from Equation (5.1):

```
movl var, %eax
```

Here, `var` is a label pointing to a long variable in memory, and its address is therefore encoded as a displacement in the instruction binary form. An example, on the other hand, using all the four parameters is:

```
movl -4(%ebp, %edx, 4), %eax
```

where the target memory address is computed by multiplying the value stored into EDX by 4, adding the result to the value stored into EBP, and then subtracting 4. In Section 6.8 we will show how this type of addressing can be used to retrieve from memory parameters passed during function calls.

The versatility of this addressing mode gives the possibility to translate easily to assembly instruction more complex high-level operations. For example, if in a high level language we want to access an element of a vector of integers, we would write code like:

```
int array[16];
...
```

[4]This is true as well for many CPUs which do not provide specific hardware to perform multiple bit checks in the same clock cycle.

```
int element = array[10];
```

This would be immediately translated into the following assembly instructions:

```
movl $10, %ecx
movl array(, %ecx, 4), %edx
```

In fact, `array` is a vector of 32-bits integers, and its 10-th element is stored 10*4 bytes after the initial address, which is pointed by the label `array`. By using the z64 addressing method, the CPU computes `array` + 10 * 4 in a single instruction, giving the possibility to scan fundamental data structures more easily.

## 6.5 Basic Programming Constructs

Every programming language offers a set of basic keywords which allow the programmer to control the flow of the program. The execution of these statements results in a choice between two or more paths in the code to be followed (*branching*), or in the repetition of a set of instruction a fixed number of times, or until a specific condition is met (*loops*). These higher-level constructs define *blocks* of code, namely a sequence of instructions which are executed one after the other, before a subsequent choice must be made, depending on the evolution of the algorithm.

In assembly programming, the essential building blocks to implement control flow statements are labels, which are used to define blocks of code (considering that if an instruction does not explicitly change the value of RIP, then the immediately next instruction is executed), and (conditional) jumps, which can exploit the outcome of comparisons, as we have introduced in Section 6.3.2.

In the following paragraphs, we will plunge into the art of assembly programming by discussing how it is possible to implement most common control flow statements using the z64 assembly language.

#### if-then-else

The simplest and most general branching construct is *if-then-else*, where one (ore more) conditions are evaluated, each one being associated with a block of code, which is executed only if the corresponding condition is met. In the most general case, this statement has the structure:

```
if(condition 1) {
    < block A >
} else if (condition 2) {
    < block B >
} else {
    < block C >
}
```

The construct usually begins with a `cmp` instruction, and a conditional jump is used to check whether `condition 1` is *not* met. In fact, if this negated condition is not met, it means that `condition`

1 is true, the branch is not taken, and the execution flow continues to the next instruction, which is actually the first instruction of `block A`.

On the other hand, if the negated condition is met, it means that the program should check whether `condition 2` is true. To this end, the conditional jump's target should be the preamble of `block B`, where a `cmp` and a conditional jump check whether the negated `condition 2` is met or not. This pattern should be repeated for each corresponding `else if` statement of the higher-level language, until the last block (`block C` in our general example) is reached. This last block has no preamble (so its label points exactly to the first instruction of the block), as if it had been reached by executing all the conditional jumps, it means that no previous (positive) condition has been met.

Of course, one last step is missing in the implementation of a correct if-then-else statement. If we consider this example:

```
if(x > 5) {
    < block A >
} else if (x > 3) {
    < block B >
} else {
    < block C >
}
```

and we assume that the value of x is 6, we then have the following execution flow: x is compared to 5, and the conditional branch checks whether $x \leq 5$. Since this is not the case, `block A` is executed. At the end of the last instruction of this block, the program finds the target instruction of the conditional branch which ensured that $x \leq 5$. So the value of x is checked against 3 using the inverse condition $x \leq 3$. This condition is not met, and therefore `block B` is executed as well! And furthermore, this behaviour is again repeated for `block C`.

The culprit here is a missing jump instruction at the end of each block. In fact, the actual semantic of the if-then-else statement is that one (and only one) block should be executed within the possible ones. Once a block is selected for execution, at the end of it an unconditional jump instruction should point the execution flow to the first instruction at the end of the whole if-then-else construct. A correct example is shown in Example 6.5.1, where C statements are placed aside of the corresponding assembly instructions implementing them. We use equality conditions in the if-then-else statement, as different comparisons deserve an additional discussion.

**Example 6.5.1:** if-then-else Construct

C code:

```
1
2  int x = 1;
3  int val;
4
5
6
7  if(x == 2) {
8
9      val = 2; // Block A
10
11
12
13 } else if (x == 1) {
14
15     val = 1; // Block B
16
17
18 } else {
19     val = 0; // Block C
20
21 }
```

Assembly code:

```
1  .data
2      x: .byte 1
3      val: .byte 0
4
5  .text
6      movb x, %al
7      cmpb $2, %al
8      jnz .elseif
9      movb $2, val ; block A
10     jmp .endif
11
12     .elseif:
13     cmpb $1, %al
14     jnz .else
15     movb $1, val ; block B
16     jmp .endif
17
18     .else:
19     movb $0, val ; block C
20
21 .endif:
```

As mentioned, comparisons are negated: for example, on line 7 of the C code the condition is `x == 2`, while in the corresponding assembly code we compare 2 with the value of AL, and if the result is not zero (meaning that they are *not* equal) the branch is taken. Additionally, to make the code correct, after each code block, a `jmp .endif` instruction is placed. `.endif` is a label placed at the end of the construct, pointing exactly to the first instruction not being part of the if-then-else.

Note line 6 in the assembly code: this instruction has no correspondence in the C code. It essentially loads the memory value from x and stores it in a fair-size register, namely AL. This register is then used in the following `cmp` instructions. This is an optimization for the execution of the program, originating from the observation that the code in the example never modifies the value of x. In fact, it is semantically equivalent to replace AL in the `cmp` instructions at lines 7 and 13, but doing so would require additional accesses to memory to retrieve a value which has been already loaded to the processor. A good assembly program always tries to reduce to the lowest the number of memory accesses, and does so by using registers. If a value should be used more than once, it is much better from a performance point of view to load it once, use it repeatedly, and then if it has been modified, store it back to memory[5].

So far, we have used the `cmp` instruction and conditional jumps to verify conditions on data, but

[5] Higher-level languages demand the task of identifying the frequency of accesses to variables to the compiler, although some languages offer *modifiers* to the definition of variables, explicitly telling that a variable will be accessed often, like `register` in C.

any instruction modifying status bits in FLAGS can be used, since conditional jumps are only checking the values of these bits to make a decision, independently of the instruction which has modified them. This is particularly useful if we deal with more complex conditions, for example mathematical expression. In fact, if we want to branch using a condition like `x - y < 0`, we could decide to write this code:

```
movb x, %al
movb y, %bl
subb %bl, %al
cmpb $0, %al
```

In this case, we want to check whether the result of `x - y` is negative, which is done using `js`/`jns` to inspect the value of `SF`. This flag is already set by `subb y, x`, and in fact `cmpb $0, x` simply subtracts zero from the result of `x - y`, making no change in the value of FLAGS.

Of course, this has an additional effect. Since many instructions can change the value of any status flag, if we don't use a conditional jump right after the instruction which has updated FLAGS, we might find flags updated twice, and take the wrong branch. Therefore, unless we are sure that an instruction does not change any status flag (which we can be, by checking Appendix A), it is convenient to use conditional branches right after having executed a `cmp` or a logical/arithmetic instruction.

Going back to Example 6.5.1, note that at the end of `block C` there is no `jmp .endif` instruction. This is because the common behaviour of the instruction is to jump to the next one, unless it is a control flow instruction. Writing code like the following:

```
      jnz elsebranch
      jmp here
here: nop ; block A
      jmp endif
elsebranch: nop ; block B
endif: halt
```

is a pure waste of clock cycles! The second instruction just does not add anything to the program itself (and replacing it with a `jz` would have not made it better!).

In higher-level languages, the programmer can rely on boolean variables. CPUs have no notion of boolean variables, rather they rely on unsigned integers (usually unsigned bytes). The common convention is that `false` = 0, and `true` ≠ `false`, meaning that a value is true if it is non-zero. Then, to check if a boolean variable is false, it can be compared to zero, as in Example 6.5.2.

**Example 6.5.2:** if-then-else Construct

C code:

```
boolean var = true;

if(var) {
    < block A >
} else {
    < block B >
}
```

Assembly code:

```
.data
    var: .byte 1

.text
    cmpb $0, var
    jz .false
    < block A >
    jmp .endif
    .false:
    < block B >
    .endif:
```

As last point of discussion, some high-level conditions cannot be mapped to checking a single status flag. This is, for example, the case of conditions on signed arithmetic. Let us define x = 3 and y = -2. We want to implement an assembly program that sets to 1 the byte at address 0x1280 if x > y. Recalling the discussion on signed arithmetic comparison, we have to check in this case whether OF = SF, but there is no single instruction to check two status flags at once[6]. Therefore, the high-level if-then-else statement must necessarily be split into more assembly level comparison, by checking, for example first if SF is set (cleared), and then if OF is set (cleared). In our example, since we are checking for a strict inequality, we must as well check the value of ZF. The algorithm is given in Example 6.5.3.

**Example 6.5.3:** Signed-Arithmetic Comparison

```
.org 0x800

.data

x: .word 3
y: .word -2

.text
    movw x, %ax
    movw y, %dx
    cmpw %dx, %ax
    jz .doNotSet

    js .sfIsSet
    jo .doNotSet  # executed if SF=0. If OF=1 then SF != OF
    jmp .set
    .sfIsSet:
```

[6] On real Intel architecture, conditional jump instruction to check these two bits at once are actually provided, exactly to simplify implementing flow controls like the one described here.

```
18      jno .doNotSet # executed if SF=1. If OF=0 then SF != OF
19
20      .set:
21      movb $1, 0x1280
22
23      .doNotSet:
24      hlt
25  END
```

### Loops

A second essential family of programming constructs is the loop, which is used to repeat a set of instruction multiple times. It is essential because the number of iterations can depend on a runtime variable, making it impossible to *unfold* the loop in all situations, and because *iteration* is one of the basic control structures which allows to implement *any* algorithm, as stated by the Böhm-Jacopini theorem.

At a high level, there are three essential looping construct, namely *for*, *while*, and *do/while*. From a low-level point of view, these three constructs reduce to only two, depending on where the termination condition is controlled—in fact, a *for* loop can be easily converted into a *while* loop, simply moving the initialization of the iteration variable outside of the loop, and updating it just before giving control back to the first instruction of the loop, namely the one that verifies the condition.

The assembly counterparts of *while* and *do/while* loops are based on the `jmp` instruction. The ability for the CPU to make RIP point to a previous instruction is the basis to implement an iteration. Then, it is necessary to devise a way to *leave* the loop, which means that we must use an additional conditional jump to leave the repetition. The basic structure of the while and do/while loops is shown in Example 6.5.4. We immediately note that the syntax to implement a do/while loop is much more compact that the plain while one, and this second solution is therefore to be preferred whenever possible.

**Example 6.5.4:** Assembly Loops

**While Loop**

C code:

```
while(<condition>) {
    <code>
}
```

Assembly code:

```
test: jnz skip
# <code>
jmp test
skip:
```

**Do/While Loop**

C code:

```c
do {
    <code>
} while(<condition>);
```

Assembly code:

```
begin: # <code>
jz begin
```

Generally speaking, a for loop has a limited number of iterations. This can be either directly encoded in the source, or it can be derived at runtime by some condition. The first case allows us to write a for loop in the following form:

```
                # %rax contains the number of iterations to repeat
  movq $0, %rcx # %rcx is the counter, the common 'i' variable
test: cmpq %rax, %rcx
  jz end
  # <code>
  addq $1, %rcx
  jmp test
end:
```

On the other hand, if we already now when implementing the code what is the number of iterations (say, 10), we do not need to compare the content of RCX with RAX, rather we can rewrite the compare instruction as `cmpq $10, RCX`.

Nevertheless, this code is exactly the translation of a high-level loop into assembly, and this is exactly why we had to differentiate between loops initialized with a fixed (hard coded) number of iterations, and one which is initialized. In fact, this difference is reflected in the actual `cmp` operands. Nevertheless, in a loop we are not actually interested in that the counter variable reaches a certain value, rather we want that the number of iterations is exactly a certain amount. Then, we can optimize the loop code by recalling that the FLAGS register is updated by any arithmetic operations, not only by a `cmp`, and so a `sub` instruction will actually set ZF when the counter variable reaches zero.

The final code of a loop, then, becomes:

```
loop:
  # <code>
  subq $1, %rcx
  jnz loop
```

This optimized code relies on the initialization of RCX, which is set to the amount of iterations required for the loop, be them hard coded or depending on a runtime computation.

## 6.6 Dealing with Arrays and Structures

When writing a program, implementing control flows is of no real value if there lacks the possibility to represent data for processing. As we have discussed in Section 2.2, the simplest data structures which we can use are arrays and matrices (or $n$-dimensional arrays).

The essential operation when dealing with arrays is the *iteration*, which consists in repeatedly accessing the elements, either for reading their content or for updating them. This operation can be performed in different ways, anyhow relying on several variants of the z64 addressing mode, and depending on the size of each element. In fact, as we have seen in Section 2.2, an array is simply a collection of elements, which can be either primitive types, or more complex data structures. In any case, iteration is implemented via some sort of loop, which must explicitly handle the size of the vector which, as we have seen, is usually stored in a separate variable.

The naïve solution to iterate over an array is shown in Example 6.6.1, where we exploit the fact that every array has a base address (which is the address of the first element) and that every item has the same size. In the example, we assume that we have declared an array of longwords of size `size` called `array`. We want to iterate on the array for reading, meaning that we want to copy the data from memory to an internal CPU register, to perform some action on it. When accessing in read mode, we copy data into the CPU registers so that the original copy in memory is not modified. Otherwise, any change to the data will produce a side effect on the original value, actually resulting in a write access.

Since we want to iterate over all the elements of the array, we know that we want to use indices in the range $[0, \texttt{size} - 1]$. Thus, at line 2 of the example we reset the counter register[7] in order to *logically* start from index 0—we do not actually use this register for accessing the array. Similarly, at line 1 we load the base address (i.e., the address of the first element of the array) into the 64-bit source RSI register. Since every memory location is identified by a 64-bit address, we are forced to use the 64-bit RSI register, as smaller virtual registers might not be able to keep the whole address.

This is enough to access the first element of the array, provided that we know the size of each entry. In Example 6.6.1, we assume that it is an array of longwords, therefore we rely on assemly instructions with the suffix `l` to operate on array's elements. In the actual loop shown in the example—the first instruction of the loop is marked by the label `.loop` we load the CX-th element from memory into the accumulator register, by relying on a `mov` instruction—see line 5. At this point, we can perform any sort of operation on the content of the CX-th element, remembering that we are working on a *copy* of the content. Therefore, to make any change to the content persistent, the altered value should be copied back.

Then, we move to the next element. Since we are accessing the array's memory relying only on the RSI base register (again, see line 5) which is thus acting as a pointer, we have to increment it by the size of an entry. As said, this is an array of longwords, therefore to access the next element we increment the RSI pointer by 4 bytes—see line 10. At the same time, since iterating over the elements of an array involves a loop, we must increment the counter CX register—see line 9. In this way, at the end of the loop's block, we can check the content of the CX register against the total number of elements (line

[7] We have used in this example the 16-bit counter CX, which allows to access a vector of up to 65,536 elements, just as an example. If more or less elements are present in the array, a different-size register should be used.

11). Considering that the CPU executes this check *after* each iteration of the lopop body, the content of CX will equal the total number of elements `$size` once RSI will point to the first element *out of the array's boundaries*. There is no possibility, anyhow, that this non-existant element will be accessed, since the `cmp` instruction at line 11 will set ZF, so that the conditional jump at line 12 will not iterate again through the loop body. Any previous iteration will find ZF cleared, so that the `jnz` instruction will effectively iterate.

**Example 6.6.1:** Naïve Array Iteration Algorithm

```
1     movq $array, %rsi    # Load the base address of the array in %rax
2     xorw %cx, %cx        # Set the counter to zero
3   .loop:
4     movl (%rsi), %eax    # Move the current element
5
6     # <access the entry copied in %eax>
7
8     addw $1, %cx         # Increment the item counter
9     addq $4, %rsi        # Move to the next longword, which is 4 bytes away
10    cmpw $size, %cx
11    jnz .loop
```

Let's appreciate again the difference between the two `addw` and `addq` instructions at lines 9 and 10. They are the same instruction, yet they have two different suffices. On the one hand, CX should be handled as a word, since we have assumed that our array has less than $2^16$ elements. On the other hand, RSI keeps an address, which is always 64-bit long, and therefore its content must be handled as a quadword. Again, the CPU cannot determine what is the *meaning* of the data kept in registers (in memory locations as well) and it is therefore up to the programmer to properly identify the *type* of the data handled by the CPU by relying on proper suffices.

For primitive datatypes, anyhow, the z64 addressing mode discussed in Section 5.6.1 comes handy. In Example 6.6.2 we see the same code as in previous Example 6.6.1, with the main difference at line 4. Here, rather than relying only on the base register, we use as well the index register along with a size of 8. As mentioned in Chapter 5, machine instructions do not have any field to specify the size of the index register, thus only quadrword registers can be used. This is the reason why we use RCX as the index register, rather than CX as in Example 6.6.1.

A direct effect of using the full addressing mode offered by the z64 CPU is that we have reduced the length of the code by one instruction. In fact, we do not use RSI as a pointer anymore, therefore we do not have to increase its value to make it point to the current entry in the array. The actual computation of its address is done transparently by the CPU microcode, when evaluating the address of the first operand of the `mov` at line 4. Therefore, although minimal, this variant of the code to loop over arrays is a bit more efficient than the naïve one, especially when run on pipelined CPUs, which we will study in Chapter 13. In any case, if the size of each element is larger than 8 bytes (as in the case of structs), then we must revert to the code in Example 6.6.1.

**Example 6.6.2:** Array Iteration Algorithm

```
1      xorq %rcx, %rcx # We need RSI to use it as an index
2      movq $array, %rsi
3  .loop:
4      movq (%rsi, %rcx, 8), %rax
5
6      # <access the entry copied in %rax>
7
8      addq $1, %rcx
9      cmpq $num, %rcx
10     jnz .loop
```

An additional optimization of the looping code can be obtained by using in a more clever way the actual addressing mode, as reported in Example 6.6.3. In particular, the loop in Example 6.6.2 relies on two different CPU registers: RSI for the base address, and RCX to determine what is the current element. If we think of a code snippet which scans a three-dimensional matrix, it's easy too see that a large number of general-purpose registers will be dedicated only to selecting the proper element to be accessed—an example of how to scan a matrix will be discussed later in this Section.

The z64 full addressing mode has an additional parameter, namely the offset. While this parameter was introduced in the addressing mode to displace within a certain data entry (as in the case of structs), it is a common practice to use it in place of the base register, in case the base address can be determined at compile time—this is the case of static arrays in main memory.

Example 6.6.3 shows a variant of the previous code to iterate over the array which relies on the *displacement* parameter rather than on the *base register* parameter. At line 3, with respect to Example 6.6.2, the RSI register is no longer used—no actual base register is used at all—and the offset is set to be `array`, i.e. the (fixed) base address of the array. Therefore, depending on the content of the RCX register (which we recall has a value in the range $[0, size - 1]$), the addressing mode at line 3 in Example 6.6.3 will first compute the offset in bytes within the array (as RCX·8) and then will sum up the base address of the array, as if it were a displacement[8].

**Example 6.6.3:** Array Iteration Algorithm without Base Register

```
1      xorq %rcx, %rcx # We need 64-bit RCX to use it as an index
2  .loop:
3      movq array(, %rcx, 8), %rax
4
5      # <access the entry copied in %rax>
6
7      addq $1, %rcx
8      cmpq $num, %rcx
9      jnz .loop
```

[8] This is a common practice adopted as well by standard compilers. Indeed, inverting the semantic of the base register and the displacement in case of a global variable allows to save a non-minimal number of registers in compute-intensive application, and should be therefore the preferred syntax whenever possible.

Note that the base address is spelled `array` rather than `$array`. As mentioned, a symbolic name starting with the `$` sign identifies a costant value, while a symbolic name alone identifies a memory address. Since the first operand of the `movq` instruction at line 3 targets a memory location, the convention is to use a displacement without the `$` sign. Indeed, an instruction like `movq $array(, %rcx, 8), %rax` would generate an error by the assembler.

There could be numberless variants to the above code. First of all, any of the looping constructs described in Section 6.5 could be used to iterate over an array. Other variants involve the way according to which the end of the array is identified.

Example 6.6.4 shows how to iterate over an array without relying on a counter and on the size of the array itself—this is a complete program, not just a code snippet. The trick used when relying on this approach is to have in memory an additional variable (called `endarr` in this example) which falls in the memory layout just after the last element of the array itself. This approach allows to increase the size of the array just by inserting/removing elements from the declaration of the array in the `data` section, thanks to the fact that symbols that are declared contiguously in a given section, are contiguous in memory as well.

The `lea` instruction (*load effective address*) at line 12 is used to compute the address of the current element of the array. The difference between a `mov` and a `lea` is that the `mov` actually performs a memory access, while a `lea` only evaluates the address according to the used addressing mode and copies this address in the destination operand. Therefore, after having executed the `lea` instruction, the RAX register keeps the address of the current element, which can be compared to the address of `$endarr`.

An additional variant entails computing the address of the first element out of the boundaries of the array starting from the number of its elements. This variant is reported in Example 6.6.5, where the instructions to convert the number of elements of an array into the address of its end are shown.

**Example 6.6.4:** Array Iteration Algorithm without Array Size

```
1  .org 0x800
2
3  .data
4      array: .long 1,2,3,4,5,6,7,8,9,10
5      endarr: .long 0xdeadc0de
6
7  .text
8      movq array(, %rcx, 8), %rax
9
10     # <access the entry copied in %rax>
11
12     leaq array(, %rcx, 8), %rax
13     addq $1, %rcx
14     cmpq $endarr, %rax
15     jnz .loop
16     hlt
```

**Example 6.6.5:** Computing the Address of the Array End

```
1 movq $array, %rax
2 movq $num, %rcx
3 shlq $3, %rcx
4 addq %rax, %rcx
```

In this example, we assume that we are dealing with an array of quadwords. The conversion starts by loading into RAX the base address of the array, and into RCX the element count. Since each element is 8-byte wide, we have to multiply the size of the array by 8. The z64 CPU does not offer an instruction to perform a multiplication, but since $8 = 2^3$, we can perform the multiplication by left shifting the content of the RCX register of three positions[9] (line 3). Adding to this value the base address gives the address of the end of the array.

Let's now give a glance to some more complicated examples of how arrays could be used. In Example 6.6.6 we find a code snippet which shows how to scan only odd elements within an array of quadwords. In this particular example, we rely on a pointer register, as in previous Example 6.6.1. After having accessed one element, the pointer RSI register is incremented by 16 bytes. Here, 8 bytes are to skip the element which has already been read, and bytes are to skip the next element.

At the same time, we show in this example an additional variant to the technique used to iterate over arrays, which is related to how we check for the array end. In particular, we load into RCX the total number of elements of the array (line 2), and we decrement its content until we reach zero (line 6). Since we are accessing only odd elements of the array, at line 6 we decrement RCX by 2.

We emphasize that after having subtracted 2 from RCX, there is no need to execute a `cmp` instruction to check the value of RCX. Indeed, any logical/arithmetic instruction set the FLAGS register accoringly to the result of the operation, therefore we can check the proper flags to determine if we have reached the end of the array right after the execution of the `sub` instruction. This allows to remove an additional instruction, compared to the previous code fragments which we have analyzed.

**Example 6.6.6:** Scanning Odd Elements of an Array

```
1     movq $array, %rsi
2     movq $num, %rcx
3 .loop:
4     addq (%rsi), %rax
5     addq $16, %rsi
6     subq $2, %rcx
7     jc .loop
```

[9] This is a common practice on many computing architectures: some computing architectures could execute shift operations faster than a multplication, while on some modern architectures relying only on shift operations might empty the instruction cache. When writing core assembly components, it is important to profile, in order to determine what is the most efficient solution. On the other hand, if a component is not core and multiple instructions are available, it is a better practice to write in a *semantically correct* way, giving up with micro-optimizations, namely shifting when a shift is meant, multiplying when a multiplication is meant. This will produce more readable code.

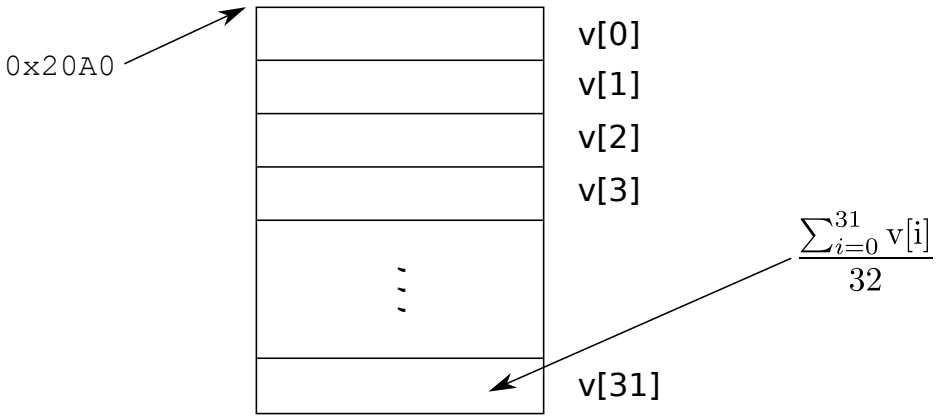



**Figure 6.2:** Computing the Average of an Array

Since we are skipping elements, there are two conditions which force the CPU to leave the loop: either if RCX reaches zero, or if its value becomes negative. We can anyhow consider the content of RCX as an unsigned value, since a negative element of an array has no actual meaning[10]. Considering the checks that were presented in Table 6.5, we continue in the loop if RCX $\geq$ 2, meaning that we continue in the loop if there are at least 2 more elements to be checked. In case of unsigned values, this entails checking only if CF is set—more precisely, if the "borrow" bit is not set. As already said before, this same technique to determine whether to continue with other iterations can be used in any other structure of code. In general, counting to zero is "much easier" to count to the array size.

We now try to implement a program which solves the following assignment:

> *Given an array* v *of 32 unsigned bytes, stored at address* 0x20A0*, compute the average of all elements and overwrite the last entry of the vector with the average. In case of an overflow, the value* −1 *should be stored in the element* v[31].

A solution to this assignment is presented in Example 6.6.7, and it can be visualized as in Figure 6.2. The solution involves scanning the whole array, summing up all the elements into the accumulator register, and dividing the final sum by the number of elements. Since we have to explicitly take into account the possibility of an overflow, we must perform check must be performed after every add instruction. Indeed, in one addition generates an overflow, executing another one will "lose" this information, since CF is updated again.

In the example, the array is declared using the .fill directive, specifiying that we want to reserve an array of 32 bytes initialized to zero—the third argument to the .fill directive is not specified, thus defaulting to zero.

[10] Anyhow, a negative element of an array is a legal syntax in many languages. For example, in C an array can be dereferences as array[-1]. This would be mapped to an address coming before the actual start of the array. Whether this is correct or not, depends on the actual program.

**Example 6.6.7:** Computing the Average

```
.org 0x20a0
.data
    array: .fill 32, 1
    .equ dim, 32
    .equ log2dim, 5
.text
    xorq %rax, %rax      # We use the accumulator to keep the partial result
    movq $dim, %rcx
  .loop:
    addb array(, %rcx, 1), %al  # sum the i-th element
    jc .error           # Check for an overflow
    subq $1, %rcx
    jnz .loop
    shrb $log2dim, %al # Divide by the number of elements
    jmp .write
  .error:
    movb $0xFF, %al      # Write -1 (in two's complement) in the last element
    movq $dim, %rcx
  .write:
    subq $1, %rcx
    movb %al, array(, %rcx, 1)
    hlt
```

Again on divisions. Since the z64 has no instruction to execute a division, we rely on the `log2dim` constant. Since $\log_2 32 = 5$, we know that dividing by 32 is equivalent to a right shift of five positions. This is exactly how the division is taken in the example. When writing on the last element, we decrement RCX by one, since the control flow leaves the previous loop when RCX reaches the number of the first element out of the array. The remainder of the code should be straightforward now, so we leave to the reader the task of understanding every single line.

We now try to implement an assembly program to sort the elements of an array of unsigned integers. We consider one of the simplest algorithms to sort an array, namely the *selection sort*. According to this algorithm, the array is divided into two parts at any time: the *active* part and the *sorted* part—the latter, when the algorithm starts, is empty. Then, the algorithm selects the smallest element (hence the name) in the *active part* of the array and moves it to its correct position, performing a simple swap. The swapped element is always moved to the active part, so it will be sorted eventually. The overall pseudocode of the selection sort is shown in Algorithm 6.1.

The algorithm relies on two nested loops. The outer one essentially enlarges the sorted part of the array and identifies which is the position where the minimum (of the active part) should be placed. The inner one scans the (current) active part to select the minimum. This is reflected in the assembly implementation of the algorithm, which is presented in Example 6.6.8. In this example, we declare an array of 4096 elements and a constant to keep its size. The array is declared using the `.comm` directive, which places it in the `bss` section. For the sake of brevity, we do not report any initialization of the vector, which could be either done statically (i.e., by relying on the `.byte` directive) or at program startup. Initializing a program requires some sort of interaction with the external world, therefore we will discuss this in a more thorough way in Chapter 7.

**Algorithm 6.1** Selection Sort Pseudocode

```
1:  procedure SelectionSort(A: array of elements to reorder)
2:      for i ← 0 to n − 2 do
3:          min_pos ← i                                  ▷ Divide the array into active and sorted parts
4:          for j ← i + 1 to n − 1 do                    ▷ Scan the active part
5:              if A[j] < A[min_pos] then
6:                  min_pos ← j
7:              end if
8:          end for
9:          if min_pos ≠ i then                          ▷ Swap the elements if the first is not the minimum
10:             tmp ← A[i]
11:             A[i] ← A[min_pos]
12:             A[min_pos] ← tmp
13:         end if
14:     end for
15: end procedure
```

**Example 6.6.8:** Selection Sort

```
.org 0x800
.data
    .comm   array, 4096 # byte array
    .equ    size, 4096 # size of the array

.text
    xorq %rbx, %rbx # rbx is the index of the position where to store the
        minimum

  .outer_loop:
    leaq 1(%rbx), %rcx # %rcx contiene l'indice dell'elemento da confrontare
    movq %rbx, %rdx # start scanning from the active part

  .inner_loop:
    movb array(, %rdx, 1), %al # load the current minimum
    cmpb array(, %rcx, 1), %al # compare with current element
    jc .skip # if array[%rdx] < array[%rcx]
    jz .skip
    movq %rcx, %rdx # new minimum found: save its index

  .skip:
    addq $1, %rcx
    cmpq $size, %rcx
    jnz .inner_loop

    cmpq %rbx, %rdx # Check if a new minimum has been found
    jz .dontswap
    movb array(, %rbx, 1), %r8b # swap elements
    movb array(, %rdx, 1), %r9b
    movb %r9b, array(, %rbx, 1)
    movb %r8b, array(, %rdx, 1)

  .dontswap:
    addq $1, %rbx
    cmpq $size, %rbx
    jnz .outer_loop
    hlt
```

We use two general-purpose registers to keep the indices of the position where to put the minimum in the active part (RBX in the example, which corresponds to the variable *i* in Algorithm 6.1) and the current element in the active part which is being compared with the current minimum (RCX in the example, which corresponds to the variable *j* in Algorithm 6.1). RBX is initialized to zero at the beginning of the program, before entering the outer loop. This is because we consider the sorted part as empty at the beginning of the execution.

In the outer loop, we initialize RCX to RBX + 1, in accordance to line 4 of Algorithm 6.1. To do this, we rely on the `lea` instruction, used in a way which drifts from the semantics it was meant for. The `lea` instruction is used to evaluate an address taking into account the addressing mode. The instruction at line 10 evaluates the "address" `1(%rbx)`, which is composed of a base register and a displacement. The "address" is evaluated exactly to RBX + 1. This is an optimized way to execute an addition and a data movement to the destination register RCX in a single instruction. By exploiting a `lea`, it is possible to embed up to two additions, one multiplication (by 1, 2, 4 or 8), and one data movement into a single instruction. Therefore, it is quite common to rely on a `lea` to do math rather than to compute addresses, as this allows to reduce the number of instructions that the CPU has to execute.

Before entering the inner loop, we copy the index of the first element of the active part (namely RBX) into RDX. The RDX register thus corresponds to the *min_pos* variable in Algorithm 6.1, and is updated every time that a new minimum is found. Overall, the three registers are used as follows:

- RBX: is the index in the array which is associated with the first position of the active part. This is the position where the minimum identified in the active part will be copied at the end of the inner loop.
- RCX: is the counter used to scan through the active part of the array, to find the minimum.
- RDX: is the index of the current minimum in the active part. At the beginning of the inner loop, it is initialized to the same value as RBX, assuming that the initial element in the active part *could* be the actual minimum.

In the inner loop, we first load into AL the element at the RDX-th position, namely the currently identified as the minimum. We then compare the content of AL with the element at the RCX-th position, namely the current element found during the scan of the active part of the array. We note that we are force to use the AL register, because common `z64` instructions do not allow to have both the source and the destination operand as memory operands. Therefore, we copy one operand from memory to the AL register, and we then compare the register with the second memory operand.

We have found a new minimum if the array element at position RCX is smaller than the element at position RDX. We therefore perform the opposite check in the code, to determine whether we have to skip the `mov` instruction at line 18 which updates the index of the current minimum. In both cases, we increment RCX and, if additional elements are present, we iterate the inner loop. When RCX has reached the index of the last element of the array, by the algorithm's construction we have found the minimum in the active part of the array, and its index is stored into RDX.

At this point, we check if RBX is equal to RDX. In the positive case, it means that the minimum is the first element of the active part, and therefore we do not have to move it. This is an important

check, from a performance point of view, and comes directly from the pseudocode in Algorithm 6.1. If we look at the pseudocode, the `if` clause at line 9 is not mandatory: the algorithm would be correct anyhow. The point here is that, in order to exchange the elements, we execute four memory accesses: two to read and update `A[`$i$`]`, and two to read and update `A[`$min\_pos$`]`. Memory accesses are costly, so performing additional checks on the values of CPU registers can increase the overall performance of the algorithm.

With respect to Algorith 6.1, we do not have any memory instruction which allows to implement line 11 with a single instruction, again because we cannot have both the source and the destination operands as memory operands. Therefore, the correct assembly translation of the array elements exchange is to load both memory locations into two general purpose registers, and store the values back in inverse order (lines 27–30 of Example 6.6.8). In this context, we are forced to clobber two general purpose regsiters, as no optimization is possibile. Both if we swap or not, at the end we increment RBX, meaning that the sorted part of the array has been increased by one element. If the active part is still non-empty, we iterate again over the outer loop.

To conclude our sequence of examples regarding assembly programs which deal with array, we present an example which solves the following assignment:

> *Given two arrays of signed longwords,* `a1` *and* `a2`*, build a third array* `a3` *which keeps in the* $i$*-th position the maximum between* `a1[`$i$`]` *and* `a2[`$i$`]`.

A program to solve this problem is presented in Example 6.6.9. The important point here is that each entry of the arrays is a *signed* longword. Therefore, while scanning and comparing the elements, we must rely on the signed arithmetic checks which are presented in Table 6.5. This requires additional labels in the code. In fact, since we have to determine whether `SF` = `OF` or `SF` ≠ `OF`, we branch multiple times, depending on the actual flags set by the `cmp` instruction at line 16.

Regarding the approach used to make a copy in `a3`, the code in the example tries to minimize both the number of instructions and the number of memory accesses. The trick is at lines 15, 25, and 28. Initially, EAX keeps the $i$-th element in `a2`. If this is the maximum, the instruction at line 28 copies the value into the $i$-th element of `a3`. Otherwise, a single additional memory access is performed to copy the $i$-th element from `a1` to EAX, and then the *same* instruction to update `a3` is executed. In general, this is an approach that compilers follow as well: whenever possible, prevent clobbering registers which keep values that could be useful with high likelihood in the near future.

**Example 6.6.9:** Maximum Element between two Arrays of Signed Longwords

```
1 .org 0x800
2
3 .data
4     a1: .long  0,-1, 2,-3, 4,-5, 6,-7, 8,-9
5     a2: .long -9, 8,-7, 6,-5, 4,-3, 2,-1, 0
6     a3: .fill 10, 4
7     .equ dim, 10
8
```

```
.text
    xorq %rcx, %rcx

  .loop:
    cmpq $dim, %rcx
    jz .end
    movl a2(, %rcx, 4), %eax    # %eax keeps the current element from a2
    cmpl a1(, %rcx, 4), %eax    # compare current elements in a1 and a2
    js .SFset                                   # a2[i] - a1[i] < 0
    jno .a2higher                               # v2[i] - v1[i] > 0, without
        overflow
    jmp .a1higher                               # with overflow

  .SFset:
    jo .a2higher                                # with overflow

  .a1higher:
    movl v1(, %rcx, 4), %eax    # %eax keeps the element from a1

  .a2higher:
    movl %eax, v3(, %rcx, 4)

    addq $1, %rcx                               # Go to the next entry
    jmp loop

  .end:
    hlt
```

We now discuss how to handle matrices. As discussed in Section 2.2, an $(m, n)$ matrix is stored in memory as a contiuouss array of $m \cdot n$ elements[11]. A matrix is therefore a multi-dimensional array stored in memory as a linear array. When scanning a matrix, it will be faster to do so in the order in which the array is stored. We have already discussed the two different strategies, namely the *column-major* and the *row-major* ones. Different higher-level programming languages rely on either memory organization. For example, C relies on the column-major representation, while Fortran relies on the row-major one.

This means that the programming language used to implement a scanning procedure has an effect on what is the most efficient way to implement the routing. In Fortran, considering an 6.6.10

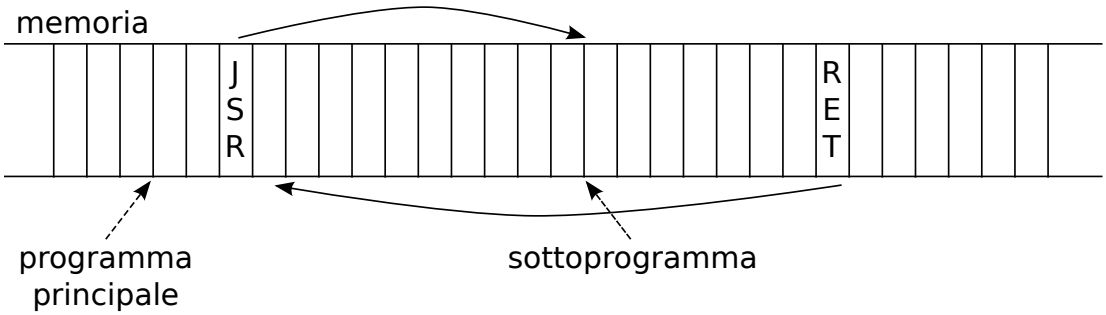
[11]In case the matrix is dinamically allocated, relying on facilities such as the `malloc` library, it usually consumes more memory and the data might not be contiguous. We do not consider here the possibility that memory is allocated dinamically.

## 6.9 Subroutines

- Possono essere considerate in maniera analoga alle funzioni/metodi dei linguaggi di medio/alto livello
- Garantiscono una maggiore semplicit, modularit e riusabilit del codice
- Riducono la dimensione del programma, consentendo un risparmio di memoria utilizzata dal processo
- Velocizzano l'identificazione e la correzione degli errori
- Per eseguire un salto a sottoprogramma si utilizza l'istruzione *jump to subroutine* (`JSR`)
- La sintassi la stessa del salto incondizionato:

```
JSR sottoprogramma
```

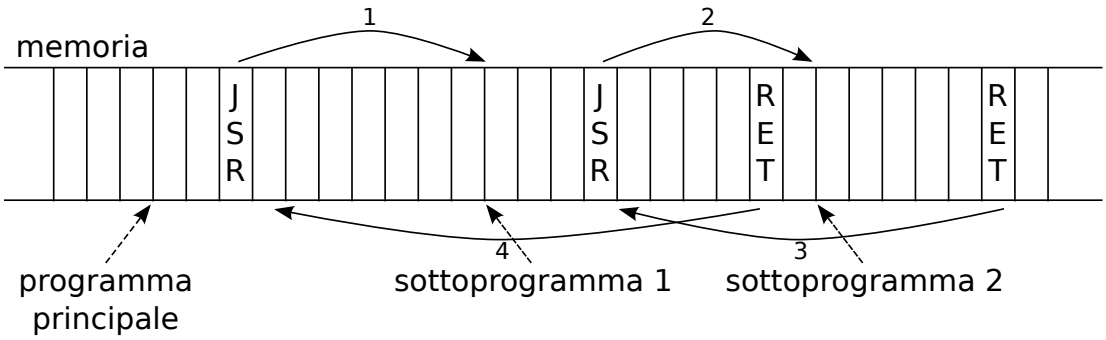
- L'esecuzione del sottoprogramma termina con l'istruzione *return* (`RET`), che fa "*magicamente*" riprendere il flusso d'esecuzione dall'istruzione successiva alla `JSR` che aveva attivato il sottoprogramma (ritorno al programma *chiamante)*




Differenza tra `JMP` e `JSR`

- A differenza della `JMP`, il microcodice della `JSR` memorizza l'indirizzo dell'istruzione successiva (*indirizzo di ritorno*) prima di aggiornare il valore contenuto nel registro `PC`
- L'unico posto in cui pu memorizzare quest'informazione la memoria
- Quest'area di memoria deve essere organizzata in modo tale da gestire correttamente lo scenario in cui un sottoprogramma chiama un altro sottoprogramma (*subroutine annidate*)
- La struttura dati utilizzata per questo scopo si chiama *stack* (pila)

Subroutine annidate




### 6.9.1 Stack Frames

Lo stack

- Gli indirizzi di ritorno vengono memorizzati automaticamente dalle istruzioni `JSR` nello stack
- ffk una struttura dati di tipo **LIFO** (*Last-In First-Out*): il primo elemento che pu essere prelevato l'ultimo ad essere stato memorizzato
- Si possono effettuare due operazioni su questa struttura dati:
  - **push**: viene inserito un elemento sulla sommit (*top*) della pila
  - **pop**: viene prelevato l'elemento affiorante (*top element*) dalla pila
- Lo stack pu essere manipolato esplicitamente
  - Oltre gli indirizzi di ritorno inseriti dall'istruzione `JSR` possono essere inseriti/prelevati altri elementi

Qui a differenza del PD32 si deve parlare anche di ebp

La finestra di stack

- Abbiamo visto come creare variabili globali e come definire costanti
- Dove "*vivono*" le variabili locali (o automatiche)?
- Una subroutine pu utilizzare lo stack per memorizzare variabili locali

```c
void function() {
    int x = 128;
    ...
    return;
}
```

```
function:
     push #128
     ...
     pop R0 ; !!!
     ret
```

- Dopo aver fatto il push di tutte le variabili, si pu accedere ad esse tramite `R7`:

$$(R7),\ 4(R7),\ 8(R7),\ \ldots$$

### 6.9.2 Calling Conventions

Le convenzioni principali permettono di passare i parametri tramite:

- Lo stack
- I registri

Per me si possono adottare entrambe, prediligendo quella per registro (pi tipica dell'x64, ma comunque pi semplice da capire).

Generalmente il valore di ritorno viene passato in un registro (`eax`) perch la finestra di stack viene distrutta al termine della subroutine. Se la subroutine chiamante vuole conservare il valore di `eax`, deve memorizzarlo nello stack prima di eseguire la `call`.

Convenzioni di chiamata

- Affinch una subroutine chiamante possa correttamente dialogare con la subroutine chiamata, occorre mettersi d'accordo su come passare i parametri ed il valore di ritorno
- Le *calling conventions* definiscono, per ogni architettura, come opportuno passare i parametri
- Le convenzioni principali permettono di passare i parametri tramite:
  - Lo stack
  - I registri
- Generalmente il valore di ritorno viene passato in un registro (per esempio R0) perch la finestra di stack viene distrutta al termine della subroutine
  - Se la subroutine chiamante vuole conservare il valore nel registro, deve memorizzarlo nello stack prima di eseguire la `jsr`

**Example 6.9.1:** Esempio: calcolo del fattoriale

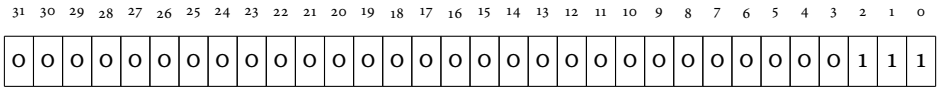
```
1  org 400h
2     numero dl 20
3     risultato equ 1200h
4
5  code     ; Programma principale (main)
6       movl numero, R1
7       jsr FATT
8       jnc corretto
9       movl #-1, risultato
10      halt
11    corretto:
12      movl R0, risultato
13      halt
14
15
16 ; Subroutine per la moltiplicazione
17 ; Parametri in R1 e R2
```

## 6.10 More Complex Operations

### Bit-wise Operations

An interesting family of operations which can be realized out of the box in assembly is that of *bit-wise operations*, which are necessary particularly in lower-level programming such as writing device drivers, low-level graphics, communications protocol packet assembly, and decoding. A bit-wise operation marks a change in the bit representation of general bit strings at the level of their individual bits. The basic operations are *bit extraction*, *bit setting*, *bit resetting*, and *bit inversion* which actually involve reading the value of a specific bit, or changing its value. The interesting fact about bit-wise operations is that setting one or multiple bits in a bit string does not require more operations, making them extremely fast, and perfectly suitable to store information in a more compact way. All bit-wise operations rely on bit-wise instructions, which are `not`, `and`, `or`, and `xor`.

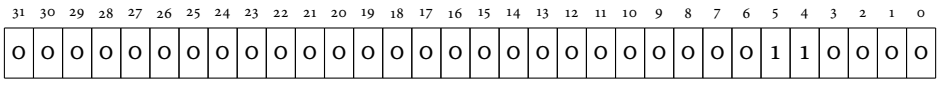
Bit extraction is the operation of reading the value of one or more specific bits from a string. Let us suppose that we are interested in knowing the value of the three least-significant bits. To do this, we have first of all to build a *bit mask*, namely an additional value which allows us to manipulate the original operand. Using a mask, multiple bits in a string can be set or cleared at the same time in a single bit-wise operation. In our example, the bit mask that we are interested in is:




By executing an and operation between the original bit string and our bit mask, all the original bits in places where there is a 0 in the mask will be hidden away. In order to code this, we should convert the bit mask to its corresponding hexadecimal (or decimal) value–which is 7—and use it like in the following instruction:

```
andl $0x7, %eax
```

supposing that the original bit string is stored in RAX. Of course, if we are interested in reading the value of bits in the middle of a bit string, we can do that by building a different bit mask. For example, to read the value of bit 4 and 5, we can use the following value:




which is 0x30 in hexadecimal and 48 in decimal. Then, the extraction can be implemented as

```
andl $0x30, %eax
```

Then, we could be possibly interested in having these bits in positions 0 and 1, and to do so we can use a shift operation:

```
shrl $4, %eax
```

At this point, the value of the bits can be used for further operations, or tested by using a `cmp` instruction. Nevertheless, if we are only interested in *testing* the value of specific bits, a different approach is to use the `test` instruction offered by the `z64` cpu, which computes the bit-wise logical and of the source and the destination operands, then discarding the result. Thus, for example, to test the value of bits 4 and 5 we will use:

```
testl $0x30, %eax
```

This will set `ZF` if the result of the bit-wise and operation is zero. Thus, having `ZF` = 1 means that bits 4 and 5 in the original bit strings where *both* zero, while if `ZF` = 0 means that *either* bit 4 or bit 5 where 1. To check for single bits, we must either rely on the aforementioned couple of `andshr` instructions, or on multiple `test` instructions.

Bit setting/resetting entails setting one (or more) specific bits to 1 or 0 respectively in a bit string, while bit inversion changes the value of one (or more) bits. To this end, we must again use bit masks as before (in order to identify which are the bits that are manipulated), and we will use as well different assembly instructions. Specifically, to set a bit we will use `or`, to reset a bit we will use `and`, while to flip a bit we will use `xor`.

If we consider a 32-bit string, to set the most significant bit we will use the bit mask `0x80000000`, which is

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

with the instruction

```
orl $0x80000000, %eax
```

From the logical properties of the or operation, we can see that the bits in the original string corresponding to zeros in the mask will be left untouched (due to the fact that *anything* $\vee$ 0 = *anything*), while the most significant bit will be set to 1 independently of its original value.

On the other hand, if we want to set to 0 the most significant bit, the involved mask will be different. In fact, if we combine `0x7FFFFFFF`:

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

with a logical and:

```
andl $0x7FFFFFFF, %eax
```

all the bits in the original string corresponding to a 1 in the mask will be left untouched (due to the fact that *anything* $\wedge$ 1 = *anything*), while the most significant one will be set to 0 independently of its value (since *anything* $\wedge$ 0 = 0).

To flip a bit, on the other hand, we rely on the `xor` instruction. Similarly to the bit setting operation, we build a mask with 1 in places corresponding to bits that we want to flip, and zeroes in any other position. For example, to flip the most significant bit, we still use `0x80000000` as the mask:

```
xorl $0x80000000, %eax
```

This works because *anything* $\oplus$ 0 result the original value of the bit (if it is 1, the result is 1, if it is 0, the outcome is 0). On the other hand, *anything* $\oplus$ 1 will invert both 0 and 1 as the leftmost operand.

Usually, in assembly programming, we require to set the content of a register to 0. The simplest instruction to do this is, for example:

```
movq $0, %rax
```

Although this approach is perfectly correct, it is unoptimized under several aspects. First of all, as we have already discussed, the immediate value of the instruction takes up additional memory for the representation of the instruction. Therefore, if several registers must be reset to zero in different places of the code, then the value 0 will be repeated several times in the program image, making it bigger without a real reason. Then, in a multicycle CPU, as the one that we have been building so far, reading this data from memory requires additional cycles, making it less efficient. And this is just to reset a register!

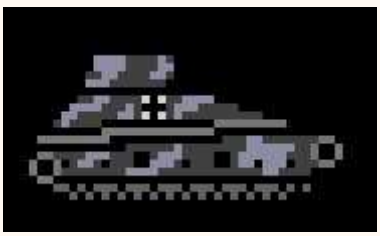
We can rely on bit-wise instruction to make this very common operation much faster, namely on bit flipping. In fact, we know that both 0 $\oplus$ 0 and 1 $\oplus$ 1 give zero as their result. Then, xor'ing one register's content with the same register's content will yield zero, independently of the original content of the register. Then, the most efficient way to clear the content of a register is:

```
xorq %rax, %rax
```

**Historical Note 6.10.1:** Bit Blitting

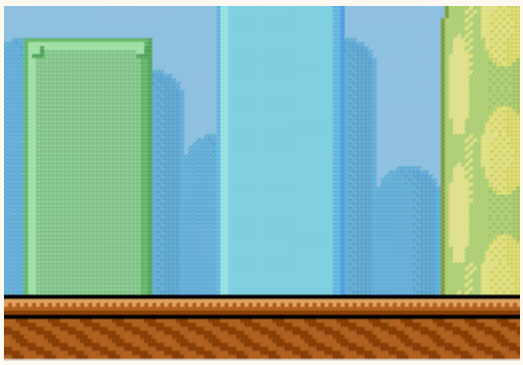
Bit-wise operations are incredibly useful, due to their speed. They have been applied to a large set of fields, among which an interesting one is *old-school* 2D platform games.
In this kind of games, there are several images which must be composed quickly, for example to show characters moving over a fixed (or slightly-changing) background. To do this operation quickly, a component external to the main CPU was adopted, which was essentially a graphical co-processor specifically designed to perform *raster operations*, based on the or, and, not, and xor operators.
The term *bit-boundary block transfer* (bit blit) refers to the composition of several bitmaps to update the flow of graphics. At least three images were involved in the process. The first one, the background, was modified to insert additional elements on top of it, by first applying an initial bitmap (the *mask*) used to clear a portion of the background image, and then applying the actual graphical element, usually referred to as a *sprite*.

Mask

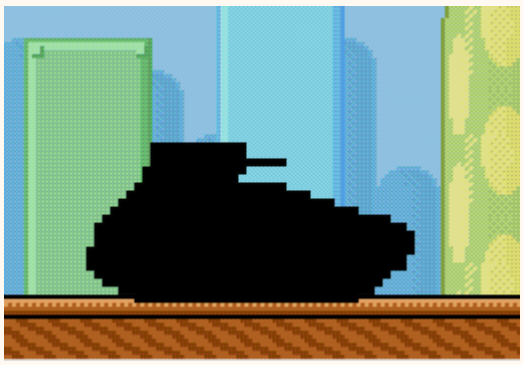


Sprite

The mask is usually an image with a white background, and a black shape of the final sprite to be drawn. The co-processor receives a memory address pointing to the mask, and a set of coordinates defining where the image should be applied. Then, the unit repeatedly performs the bit-wise and operation over every pixel of the mask and the background. Since white is represented using all ones, the and operation will not alter the original background image. Instead, since black is represented using all zeros, the and operation will clear out the shape of the final object.

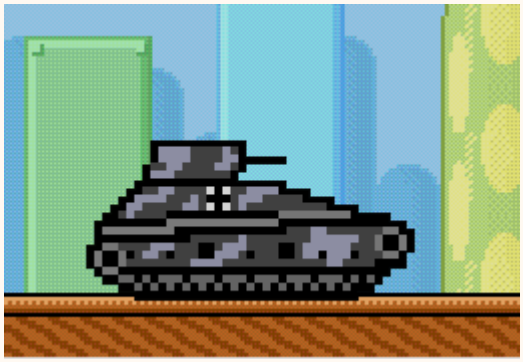


Background



Masked Background

Then, the co-processor receives the address of the sprite image, to be placed on the same coordinates of the background, and executes a bit-wise or operation over the pixels of the masked background and the sprite. Since the sprite has a black background, and since black is represented using all zeroes, the surrounding color of the sprite will not affect the background. But since we have already cleared the shape of the object (meaning that all the pixels' value is already zero), the or operator will set the pixels of the masked area to that of the sprite.



Background with sprite applied

Since old-school games were essentially using 8 or 16 bits to represent colors, graphical co-processors were built with at least 32-bits registers to perform these operations, allowing to process multiple pixels at a time. This allowed for a much faster generation of graphics, allowing refresh rates which were making games well interactive.

The problem with bit blitting was that the borders of the composed images were quite sharp. This was not a problem when the resolution of screens was quite low, yet modern graphic hardware/software has replaced bit-wise operations with more accurate ones, taking into account as well, for example, *alpha channels* to allow for a smoother compositing. Additionally, the advent of GPUs, able to perform

an extremely large amount of computations per second, has allowed for the implementation of more complex operations on the final *frame buffer*. Yet, bit-wise operations are still used to draw interactive highlight rectangles, or to show mouse pointers, on some systems.

**switch-case**

In several circumstances the number of `else if` branches in an `if` statement could be very high. High-level programming languages provide the `switch-case` statement which allows for a simpler syntax to implement multiple choices.

While this is because with a high number of branches the `switch-case` statement is much more readable than a long chain of `if/else-if`, actually its assembly counterpart can be much more efficient. In fact, as we have seen, an `if-then-else` statement check each condition of each branch, until a match is found. Entering the last branch of such a statement might require a very high number of comparisons, hampering the overall performance of the program.

**Example 6.10.1:** `switch-case` statement

C code:

```c
int x;

switch(x) {
   case 0:
      < block >
      break;

   case 1:
      < block >
      break;

   case 2:
      < block >
      break;

   default:
      < block >
}
```

Assembly code:

```
org 400h
    branchTable dl 0, 0, 0
    var dl 2
code
    ; inizializza branch table
    xorl R0, R0
    movl #branchTable, R0
    movl #case0, (R0)+
    movl #case1, (R0)+
    movl #case2, (R0)

    ; Carica il valore della variabile
        su cui fare switch
    movl var, R0
    lsll #2, R0

    ; Effettua il salto condizionato
    movl branchTable(R0), R0
    jmp (R0)

  case0:
    ; <codice>
    jmp fine
  case1:
    ; <codice>
    jmp fine
  case2:
    ; <codice>
    nop
  fine: halt
end
```

In Example 6.10.1 a sample `switch-case` construct is shown. While it might appear much more complex than the more traditional `if-then-else`, as we will show its execution could be immediate, independently of the branch which a specific comparison would take. The essence of the `switch-case` lies in the notion of *branch tables* and in the versatility of the addressing mode offered by the z64.

The branch table is actually a vector of addresses which correspond to the initial instruction of every block of the `switch-case`. To determine which is the address to jump to, the programmer should use the value of the variable on which to switch. In Example 6.10.1 we face the simplest situation: all the possible cases are associated with values of the variable `x` which are contiguous and bounded by one single upper value. To implement the switch, the programmer must first of all compare `x` value with the upper bound (2 in the example), and if its value is higher, than a jump to the address of the `default` branch is executed.

On the other hand, if `x` is lower of equal to the upper bound, its value is used as the index to

access the branch table, from which the address of the first instruction of the block is retrieved. This address is then used in a register branch, moving the control flow immediately to that instruction. Similarly to the `if-then-else` case, right before the first instruction of the next block, a jump to the end of the statement should be placed, which corresponds to the higher-level `break`. In fact, when in higher-level languages a case is not ended by a `break` statement, the semantic is to follow through the execution of the next block, which is exactly what would happen in the assembly counterpart.

In this simple situation the efficiency and simplicity of the `switch-case` statement is crystal clear. Yet, often this simple situation does not hold, and implementing a `switch-case` might require a bit more care, for example when the legal values of `x` associated with cases are not contiguous. This is shown in Example 6.10.2, where the same assembly code as before could be actually used. In fact, to deal with this specific example, we can simply leave the code untouched, and insert in the branch table entries associated with non-existing cases, where the retrieved address simply points to the `default` case.

**Example 6.10.2:** Non-contiguous `switch-case` statement

C code:

```c
int x;

switch(x) {
  case 0:
     < block >
     break;

  case 4:
     < block >
     break;

  case 8:
     < block >
     break;

  default:
     < block >
}
```

Assembly code:

```
org 400h
    branchTable dl 0, 0, 0
    var dl 2
code
    ; inizializza branch table
    xorl R0, R0
    movl #branchTable, R0
    movl #case0, (R0)+
    movl #case1, (R0)+
    movl #case2, (R0)

    ; Carica il valore della variabile
        su cui fare switch
    movl var, R0
    lsll #2, R0

    ; Effettua il salto condizionato
    movl branchTable(R0), R0
    jmp (R0)

  case0:
    ; <codice>
    jmp fine
  case1:
    ; <codice>
    jmp fine
  case2:
    ; <codice>
    nop
  fine: halt
end
```

But what happens if few cases are associated with values distributed in a very wide range? If the high-level code were:

```
switch(x) {
  case 100:
     < block >
     break;

  case 10:
     < block >
     break;
```

```
    case 1000:
        < block >
        break;

    default:
        < block >
}
```

the size of the table would be too large. In fact, it would require at least 1000 quadword entries, although only 3 were keeping an address different from `default`'s. This waste of memory would be unacceptable, and the best solution here is to fall back to the `if-then-else` code construct: the efficiency would slightly drop, but the tradeoff between performance and memory consumption would be nevertheless positive[12].

Playing with these variables, namely the time required to find the correct block to execute and the amount of memory used to build the table, it is very easy to find scenarios where there isn't any win-win situation. In fact, if we have a high number of cases, with keys which are widely distributed, we either end up with a very large branch table, or we would spend too much time *linearly searching* for the correct block. In this case, it is better to take a completely different path: it is still useful to build a branch table, which should have nevertheless entries only for really-existing blocks. If each entry is organized as a $\langle key, address \rangle$ tuple, then to find the correct block entry address a binary search could be used. This ensures that the amount of memory used to store the table is linear with the number $n$ of cases, while the time required to execute the jump is $O(\log n)$, which is actually the best combination that we can offer, and it is actually the choice made by some efficient compilers. We shown in Example 6.10.3 this kind of `switch-case`.

**Example 6.10.3:** `switch-case` with binary search

```
1  org 400h
2      branchTable dl 0, 0, 0
3      var dl 2
4  code
5      ; inizializza branch table
6      xorl R0, R0
7      movl #branchTable, R0
8      movl #case0, (R0)+
9      movl #case1, (R0)+
10     movl #case2, (R0)
11
12     ; Carica il valore della variabile su cui fare switch
13     movl var, R0
14     lsll #2, R0
15
```

[12] Indeed, compilers make this choice automatically, trying to find the best tradeoff. This means that even if a programmer uses the `switch-case` construct, it could be assembled to an `if-then-else`-like one, or vice versa.

```
16      ; Effettua il salto condizionato
17      movl branchTable(R0), R0
18      jmp (R0)
19
20    case0:
21      ; <codice>
22      jmp fine
23    case1:
24      ; <codice>
25      jmp fine
26    case2:
27      ; <codice>
28      nop
29    fine: halt
30  end
```

# 7 Interacting with the Outside World

So far, we have seen how we can create a CPU that is able to carry out useful work—basic operations which, wisely interweaved can create the complex programs that we use everyday.

Nevertheless, *computing* is not only about *processing*. A software which cannot be controlled in some way can be of little practical use. This requires to have devices such as a mouse or a keyboard, to interact with our machines. RAM is, by its construction, volatile. Shutting off a computer makes all its content disappear, so we need devices such as hard disks or USB drives to store data in a stable manner. If we need to send a document to someone we might want to use email or have a hard copy to hand over. Thus we need network interfaces or printers.

To make the long story short, to have computing architectures which are of real use, we need devices. Therefore, we must find a way to have the CPU interact with the outside world. As we will see in this chapter, this conceptually no more complicated than having the CPU interact with working memory. Nevertheless, devices can be of any complexity, and heavily varied among each other—in no way the implementation of a network interface is similar to that of a mouse!

Having to think of a different way for a CPU to interact with different devices would be anyhow the wrongest choice. In fact, this would create incompatibility among vendors, and would increase the complexity of the internal organization of a CPU to an extent that it would be too hard to design or maintain. Therefore, to have a device interact with a CPU, it's all about *interfaces*. The interface is a sort of "agreement" among CPU and device vendors: a set of control signals is defined, which can be used by the CPU to *programmatically interact* with the device. This means that different devices can use the same signals in a different way, depending on what the device is actually doing. The CPU is instructed to interact with the device in the correct way by a piece of software, often developed by the device vendor, which is commonly known as the *driver*. Having a device installed on a computing system without the corresponding driver installed makes the device of little to no use, simply because the CPU is not going to respect the correct *protocol* which allows to properly use the interface with the the peripheral.

Since the introduction of the first computer, the interfaces have been often changed. This is related to the bus that connects the CPU with the device: how many control signals are supported to have

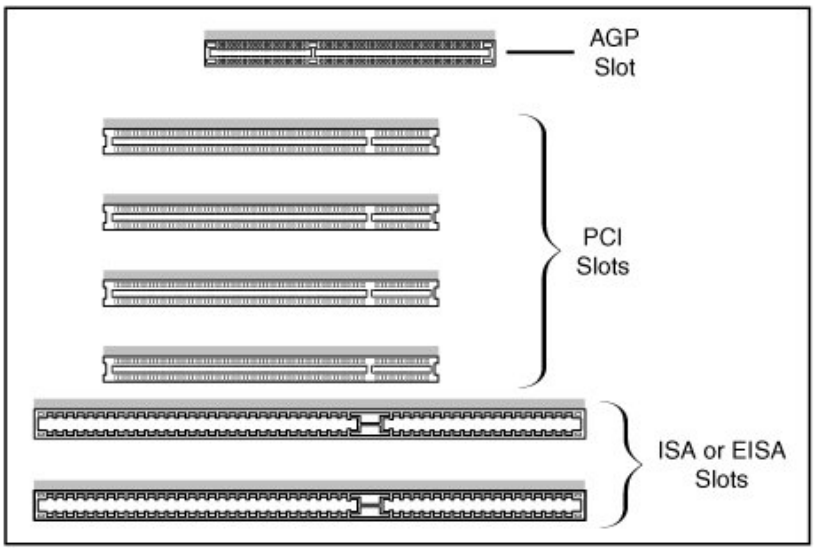



**Figure 7.1:** Some CPU/device interfaces and their corresponding slots.

the CPU interact with the device correctly? What is the width of the words that can be exchanged among them? The answers to this question tell, for example, how many "pins" will be visible on the slot associated with a given type of bus. Figure 7.1 shows some common slots for the ISA/EISA, AGP, and PCI buses. Another difference which strongly characterizes a certain type of bus is the amounts of data that it can transfer at one time and the speeds at which they can do it. Neverthless, this latter aspect is mostly related to electronic aspects, which are out of the scope of this book.

Another difference according to which we can classify devices connected to a computing system is whether they *process* or *produce* data. A keyboard only produces data, as every keystroke must carry the information about which key was pressed by the user, and the CPU should be informed of this to correctly respond to the user's command. A screen, in the general case, only processes information, as it receives some data to show and does not give back anything to the CPU. Therefore, depending on the flow of the data, we can classify devices as *input* or *output* ones. An input device is such that it produces data to be processed by the CPU. An output device receives data produced by the CPU so as to process them. This is a CPU-centric view, meaning that the classification of a device in the input or the output class depends on whether the data flow to or from the CPU. Of course, some devices can work both as input and output devices. Think for example (again!) of a network interface: outbound packets come from the CPU (and the device thus works as an output one), while inbound packets coming from the network are "generated" by the device for the CPU to process them, making it as well an input one. Input and output devices are commonly referred to using the collective name of "I/O devices".

In this chapter, we will describe how our z64 CPU can interact with input and output devices. To make the discussion simpler, we will start with the logical organization depicted in Figure 7.2, where a dedicated bus is used to create an interconnection with I/O devices. Due to its simplicity, this was a typical organization of early computing systems, still used in some microcontroller-based environment. Since this early design, the organization has significantly changed, introducing some dedicated hardware (or even dedicated CPU) which handle interaction with I/O devices. We will

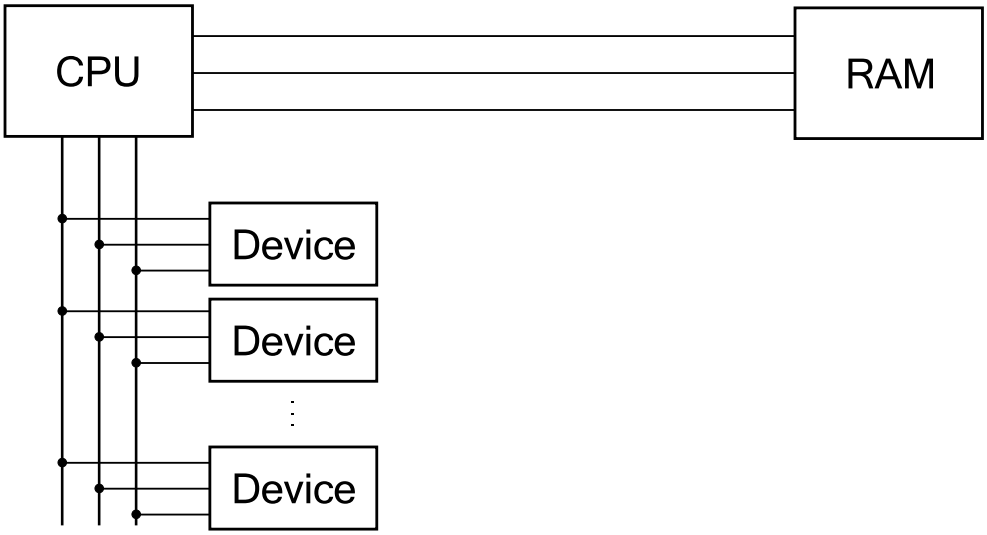



**Figure 7.2:** CPU with a dedicated interconnection for IO devices.

**Table 7.1:** Typical speeds of I/O devices

| Class | Working data unit | Exchange speed |
|---|---|---|
| Magnetic Tapes | Byte | Up to 30 Mcar/sec |
| Magnetic Disks | Byte | Up to 300 Mcar/sec |
| Serial Printers | Byte | 200–1200 car/sec |
| Parallel Printers | Byte | 1K–100 Kcar/sec |
| CRT Screens | Byte | 300–19.2 Kcar/sec |
| Analog/Digital Converters | 8–16 bits words | 10–10 Mwords/sec |
| USB 1.0 | Byte | 1.5 Mcar/sec |
| USB 2.0 | Byte | 60 Mcar/sec |

introduce the nomenclature proper of I/O interconnection, and we will later move to a more sophisticated interconnection known as *memory-mapped I/O*, which is still supported in the modern days from a programmatic point of view, although the hardware is virtualizing its behaviour to make it compatible with more complex organizations.

The ways a CPU can interact with I/O devices depend on several aspects: the speed of each component, the "urgency" to carry out a certain task, and the degree of efficiency we require for the functioning of the CPU. Since their origin, CPUs have been much quicker than I/O devices, even order of magnitudes faster. In particular, by Table 7.1 we can see that the speed at which devices exchange data is 3 to 8 orders of magnitude slower than the speed at which CPUs produce/process data. This can create issues in data transfer.

The time to exchange data between CPUs and devices should be as short as possible, to avoid incurring in non-negligible delays. This issue has been formally stated by Gene Amdahl, a former engineer at IBM who helped to design first mainframe computers. His famous "Amdahl's Law" relates the *speedup S* with the speed of CPUs and I/O devices. Amdahl noted that that while the speed of a CPU was significantly increasing as time was passing by—remember as well Moore's Law, and its effects depicted in Figure 1.4—the speed of devices was hardly keeping the pace. In particular, if we

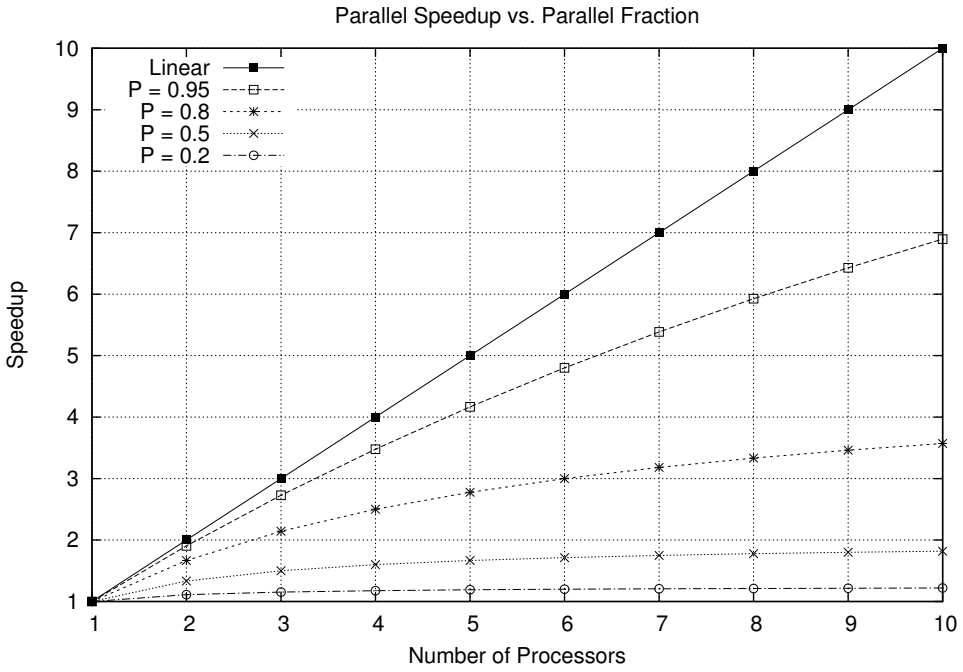



**Figure 7.3:** Speedup according to Amdahl's Law.

can divide the time taken by a program to run into two parts, a fraction $f$ of time spent in processing activities on the CPU, and a the remainder $1 - f$ spent in interactions with devices, even if we have a CPU $K$ times faster, the maximum speedup is:

$$S = \frac{1}{(1-f) + \frac{f}{K}} \tag{7.1}$$

The performance bottleneck is therefore related to the $1 - f$ fraction of the program, namely the interactions with I/O devices. Figure 7.3 shows the maximum achievable speedup when changing the fraction $f$ of time spent in processing activities on the CPU. As shown, if this fraction is 80% of the total execution time of the program, even if we consider an infinitely-fast CPU the maximum speedup achievable is:

$$\lim_{K \to +\infty} S = \frac{1}{1 - 0.80} = 5. \tag{7.2}$$

Therefore, no matter how fast is our CPU, this program can never run more than 5 times faster. And for many contexts, 80% is a significantly high estimate.

```
mov data, %eax
mov D, %edx
outb
```

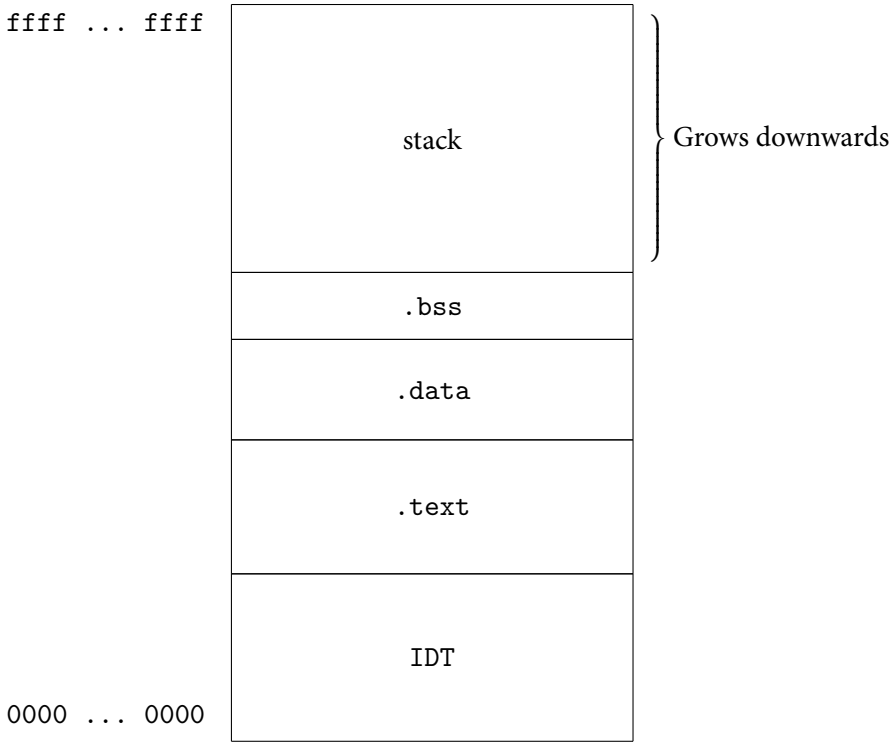



**Figure 7.4:** Map of a Program's Memory when Loaded for Running

# z64 Instruction Set Reference

The instructions used for z64 programming can be grouped into 8 different classes, depending on the actions that they perform. Each instruction is encoded in a bit string, allowing the z64 processing unit to actually execute them. As thoroughly discussed in chapter 6, to allow for an easier programming, instead of bit strings there are several mnemonics and names which can be used in place of the instructions' binary representation, and it is the role of the *assembler* to translate the textual representation to bits.

In this appendix, we will overview all the instructions which can be used to program the z64 CPU, by providing tables for each class of instructions, and showing what is the actual binary representation for each of them, as translated by the assembler. For each class of instructions, we explicitly show what are the operands that are expected for any given instruction, and what are the status flags that can be affected by its execution. When the outcome of an instruction gives a pre-defined value of a flag, that value is shown. Otherwise, if it depends on the operands, the symbol ⇳ is used.

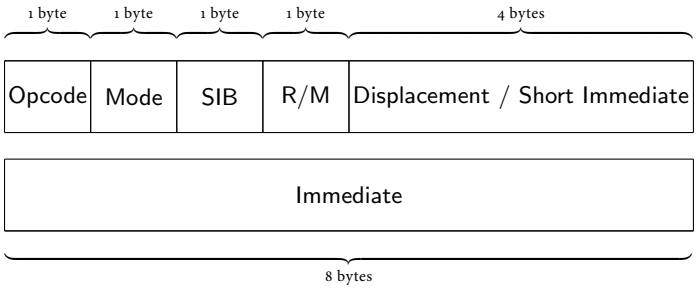



**Figure A.1:** z64 Instruction Format

The size of z64 instructions is variable, depending on the operation associated with it and the actual operands involved, meaning that an instruction can be either 64 or 128 bits long. The general organization of one instruction is shown in Figure A.1, where the various bytes composing it are presented[1]. The Opcode byte tells the CPU Control Unit what is the microcode to support instructions'

[1]Class 7 instruction do not respect this general format, as it will be shown in Section A.8.

execution, while the Mode byte gives additional information about the specific (class-dependent) operating mode. This is mostly related to memory accesses, due to the highly-general supported addressing mode, described in Section 5.6.1. The SIB byte gives information about the scale, the index and the base of a memory operand, while if the instruction requires explicit operands (namely, operands which must be encoded in the binary representation), they are encoded into the R/M (Register/Memory) byte.

The Displacement block is used to keep a 32-bits wide offset to target a memory address. This is used in data movement operations to refer specific variables in memory (for example, an array's initial address) or in program flow control instructions to specify (as an offset) the destination address to be stored into the `rip` register after the instruction's execution. Some instruction have an operand which is a Short Immediate, namely a 32-bits wide immediate, encoded in the instruction. This can be placed in the Displacement/Short Immediate block. The Immediate block allows to encode within an instruction some 64-bits wide data, which will be handled by the instruction itself during its execution.

In Figure A.2 the sub-blocks of the Opcode byte are shown. Class is a 4-bits code which identifies what class the instruction belongs to (as described in this appendix). The z64 processor has 8 different instruction classes, and therefore the most significant bit of the Class field is always forced to zero. This allows for an optional extension of the instruction set, allowing to support the implementation of other 8 additional classes. On the other hand, the Type field (again 4-bits wide) allows to select within a given class the exact instruction that the CPU will execute when a specific opcode is found in the program.

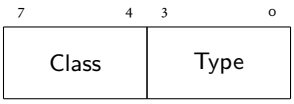



**Figure A.2:** Opcode Byte Format

The Mode byte sub-blocks are depicted in Figure A.3. The sub-blocks SS and DS contain the source operand and the destination operand sizes, respectively. They keep the binary representation of the size suffixes which are placed at the end of instructions' mnemonics, and they have the same value in case the instruction has just one size suffix (like in the case of `mov` instructions) or they keep two different values if the instruction requires two different sizes (like in the case of `movzX` instructions). DI is a 2-bits field which tells whether the instruction is using a displacement, an immediate data, or both, while Mem tells whether each of the operands of the instruction should be interpreted as memory or register operands. In case one of the two operands are memory operands, then the CPU interprets the content of the following SIB byte, to determine what parameters of the general addressing mode should be used to compute the target memory address of the operation. All the possible configurations of these bits, and their meaning, are reported in Table A.1.

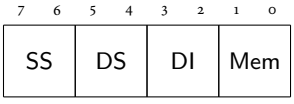



**Figure A.3:** Mode Byte Format

**Table A.1:** Mode Byte Fields Explanation

| Field | Value | Explanation |
|---|---|---|
| SS | 00 | Source is a byte |
| | 01 | Source is a word |
| | 10 | Source is a longword |
| | 11 | Source is a quadword |
| DS | 00 | Destination is a byte |
| | 01 | Destination is a word |
| | 10 | Destination is a longword |
| | 11 | Destination is a quadword |
| DI | 00 | Displacement is not used, immediate is not present |
| | 01 | Immediate is present |
| | 10 | Displacement is used |
| | 11 | Displacement is used and immediate is present |
| Mem | 00 | Both R/M Source and Destination are registers |
| | 01 | R/M Source is register, R/M Destination is memory |
| | 10 | R/M Source is memory, R/M Destination is register |
| | 11 | Impossible condition (generates a runtime error) |

The SIB byte specifies additional information about the operands, and is organized as in Figure A.4. $B_p$ and $I_p$ tell whether the addressing mode used by the current instruction is using a base and/or an index register, respectively. If $I_p == 1$, the field Index keeps the binary representation of the index register, according to the values shown in Table 5.1, and the field Scale keeps the scale value, the possible values of which are 1 (encoded as 00), 2 (encoded as 01), 4 (encoded as 10), and 8 (encoded as 11).

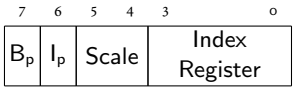



**Figure A.4:** SIB Byte Format

The R/M byte, as shown in Figure A.5, has space for two register codes. The interpretation of both the Source (Base) Register and the Destination (Base) Register depends on the value of the Mem field of the previous Mode byte, and the $B_p$ bit of the previous SIB byte. Specifically, they can either identify a plain general purpose register, or they can tell that the same register should be used as a base register. This depends on whether one of the two operands should be interpreted as a memory operand or not (as determined by the Mem field) and on whether the memory addressing is using a base register or not (as determined by the $B_p$ bit).

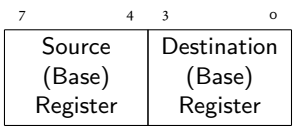



**Figure A.5:** R/M Byte Format

To give an example, the instruction `mov 0, RAX` will be encoded setting Mem = 00, telling that the SIB byte should not be interpreted and that the Destination (Base) Register value 0000 should be considered as a plain register, while the instruction`mov 0, (RAX)` will be encoded setting Mem = 01, telling that the destination operand is a memory operand. Therefore, the SIB byte is interpreted by the CPU, where it will find the $B_p$ bit set, telling that the Destination (Base) Register value 0000 should be considered as a base register.

In the following sections, each instruction class will be described, and the possible operands combinations will be shown, along with their binary encoding. In this presentation, several codes will be used to identify instruction operands classes:

`B` — the operand is a general purpose register, a memory address, or an immediate value. If it is a memory address, the address is computed from any of the following values: a base register, an index register, a scaling factor, a displacement. If it is an immediate value, its value is represented using 64 bits, and is located immediately after the instruction encoding.

`E` — the operand is either a general purpose register, or a memory address. If it is a memory address, the address is computed from any of the following values: a base register, an index register, a scaling factor, a displacement.

`G` — the operand is a general purpose register.

`K` — the operand is a constant unsigned numeric value, up to $2^{32} - 1$. Its encoding is stored in the Displacement field of the instruction.

`M` — the operand is a memory location, which is encoded a displacement from the value of RIP after the fetch of the instruction.

`P` — the operand is an I/O port, represented as an unsigned integer up to $2^{16} - 1$.

`Imm` — the operand is a short immediate.

When presenting the encoding of the instructions, the `E` and `B` operands are made explicit, to allow for an easier translation to the corresponding binary form. Additionally, generic instruction suffixes for representing the size of operand will be placed after the instruction mnemonic. In this context, the following additional codes will be used:

`B` — Base register.

`D` — Destination register.

`I` — Index register.

`O` — Displacement (offset).

`S` — Source register.

`T` — Scale.

`X` — Operand size.

`Y` — Additional operand size (if required).

Similarly, in the binary encoding lowercase letters will be used to represent the binary representation of the aforementioned codes. Each letter will be repeated as many times as required for the corresponding field in the representation. The codes used for this representation are:

`b` — Base register.

`d` — Destination register.
`i` — Index register.
`k` — Constant unsigned numeric value, up to $2^{32} - 1$.
`m` — Memory location, represented as displacement from the value of RIP after the fetch of the instruction.
`o` — Displacement (offset).
`p` — I/O port, represented as an unsigned integer up to $2^{16} - 1$.
`s` — Source register, or short immediate (depending on the instruction operand).
`t` — Scale.
`x` — Operand size.
`y` — Additional operand size (if required).
`-` — Don't care bit. It is a bit which is not interpreted by the CPU, and which is generally set to zero by the assembler.
✓ — There is immediate data encoded into the instruction.
✗ — No immediate data is encoded into the instruction.

## A.1 Class 0: Hardware Control Instructions

Hardware control instructions are a class of instructions which allow to operate on the state of the CPU.

These instructions have a quite simple representation, due to the fact that they do not need particular operands. Most of the bits are therefore don't care values, as the microcode for their execution does not depend on any parameters. There is no type-0 instruction belonging to this class, since its encoding would be done using 32 zero bits, making it impossible for the CPU to distinguish it from uninitialized data.

All instructions of this class have the Class field of the opcode set to 0. The available instructions in this class are reported in Table A.2, along with the values of the Type value, and the operands classes.

**Table A.2:** Class 0 Instructions

| Type | Mnemonic | Operands | O S Z P C | Description |
|---|---|---|---|---|
| 1 | `hlt` | – | - - - - - | Place the CPU in low-power mode, until the next interrupt is fired |
| 2 | `nop` | – | - - - - - | No operation |
| 3 | `int` | – | - - - - - | Call to Interrupt Handler |

### `hlt` — Halt

**Description.** Stops instruction execution and places the processor in a Halt state. An enabled interrupt will resume the execution, and the saved instruction pointer RIP points to the instruction following the `hlt` instruction.

The Halt state is a "low-power" mode, where the CPU sits doing nothing until an external device wakes it up. The execution of this intruction allows the processor to significantly reduce its power usage and heat output, which is why it is commonly used instead of busy waiting for sleeping and idling.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
Enter HALT state
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | | $Imm_{64}$ |
| `hlt` | 0000 0001 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

**Historical Note A.1.1:** `hlt` in real architectures

In real x86/x86_64 processing units, `hlt` is a privileged instruction, therefore it cannot be executed by user programs, rather it is the role of the operating system to use it to enter the power saving mode. This instruction is nevertheless generated by some standard compilers relying on particular implementations of the standard library, right at the end of the execution of the `main` function. Since its execution produces a fault when executed in non-privileged mode, if the program for some reason does not terminate at the end of the `main` function, it will cause a protection fault which will kill the process. Rather, if the program is for some reason running in privileged mode, it will stop the processor until the next interrupt, which will eventually trigger the operating system to remove the process.

### `nop` — No Operation

**Description.** This instruction performs no operation. It is a 32 bit instruction which takes up space in the instruction stream but does not impact machine context, except for the RIP register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `nop` | `0000 0010` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |

### `int` — Call to Interrupt Handler

**Description.** The `int` instruction executes a software-generated call to the interrupt handler specified within the operand. The value stored in the operand can range from 0 to 255, encoded as an 8-bit unsigned short immediate, and is used to find the corresponding handler pointer in the IDT.

The execution of this instruction pushes on the stack the FLAGS registers, and then the address of the next instruction. This setup of the stack can be used to return control from the interrupt handler to the original code by relying on the standard `iret` instruction.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
MDR ← FLAGS
MAR ← RSP
(MAR) ← MDR
RSP ← RSP - 8
FLAGS[IF] ← 0
MDR ← RIP
MAR ← RSP
(MAR) ← MDR
RSP ← RSP - 8
TEMP2 ← IR[Short Immediate]
RIP ← ALU_OUT[<< 8]
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `int S` | `0000 0011` | `---- ----` | `---- ----` | `---- ----` | `0000 0000` | `0000 0000` | `0000 0000` | `ssss ssss` | ✗ |

## A.2 Class 1: Data Movement Instructions

Data movement instructions create a copy of given data to/from any combination of register/memory operands, and they can include as well immediate data as the source. The only limitation is that the `mov`

instruction cannot execute a direct memory-to-memory data copy. To perform this data movement, the counterpart `movs` instruction must be used. To this class belong as well the stack manipulation and type conversion instructions. The former are commonly used to place parameters on stack before calling subroutines, to save the value of local variables, or to preserve the value of registers among different drivers' calls. The latter are used to convert byte into words, words into longwords, and longwords into quadwords. They are specially useful for converting signed and unsigned integers to larger integers, as they can extend the sign. In this case, the source size cannot be bigger than the destination size.

All instructions of this class have the Class field of the opcode set to 1. The available instructions in this class are reported in Table A.3, along with the values of the Type value, and the operands classes.

**Table A.3:** Class 1 Instructions

| Type | Mnemonic | Operands | O S Z P C | Description |
|---|---|---|---|---|
| 0 | `mov` | B, E | - - - - - | make a copy of `B` into `E` |
| 1 | `movsX` | E, G | - - - - - | make a copy of `E` into `G` with sign extension |
| 2 | `movzX` | E, G | - - - - - | make a copy of `E` into `G` with zero extension |
| 3 | `lea` | E, G | - - - - - | evaluate the addressing mode, store the result in `G` |
| 4 | `push` | E | - - - - - | copy the content of `E` into the top of the stack |
| 5 | `pop` | E | - - - - - | copy the content of the top of the stack into `E` |
| 6 | `pushf` | – | - - - - - | copy into the top of the stack the FLAGS register |
| 7 | `popf` | – | - - - - - | copy into the FLAGS register the top of the stack |
| 8 | `movs` | – | - - - - - | perform a memory-to-memory copy |
| 9 | `stos` | – | - - - - - | set a memory region to a given value |

### `mov` — Move

**Description.** Copies the source operand to the destination operand. The source operand can be an immediate value, a general-purpose register, or a memory location, while the destination operand can be a general-purpose register or a memory location. Both operands must be of the same size, as described by the suffix specified at the end of the instruction's mnemonic, which can be a byte, a word, a longword, or a quadword. Only one memory operand can appear at a time, either as the source or the destination. If the source and/or the destination operand is a general purpose register, then its size must always match that of the size suffix. In case the source operand is an immediate data, its size is truncated to the size of the destination operand (as specified by the suffix).

**Operation.** The operation of this instruction depends on whether memory operands are involved or not, and what fields of the general addressing method are used to identify memory locations.

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| movX S, D | 0001 0000 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX S, (B) | 0001 0000 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX S, (B, I, T) | 0001 0000 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX S, O(B, I, T) | 0001 0000 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movX S, (, I, T) | 0001 0000 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX S, O(, I, T) | 0001 0000 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movX S, O | 0001 0000 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movX I, D | 0001 0000 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| movX I, (B) | 0001 0000 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| movX I, (B, I, T) | 0001 0000 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| movX I, O(B, I, T) | 0001 0000 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| movX I, (, I, T) | 0001 0000 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| movX I, O(, I, T) | 0001 0000 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| movX I, O | 0001 0000 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| movX (B), D | 0001 0000 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX (B, I, T), D | 0001 0000 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX O(B, I, T), D | 0001 0000 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movX (, I, T), D | 0001 0000 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movX O(, I, T), D | 0001 0000 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movX O, D | 0001 0000 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### `movsX` — Move with Sign-Extension

**Description.** Copies the content of the source operand to the destination operand and sign-extends the value to 16, 32, or 64 bits. The size of the converted value depends on the operand-size attributes, as specified by the second size suffix added to the instruction's mnemonic.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| movsXY S, D | 0001 0001 | xxyy 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movsXY (B), D | 0001 0001 | xxyy 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movsXY (B, I, T), D | 0001 0001 | xxyy 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movsXY O(B, I , T), D | 0001 0001 | xxyy 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movsXY (, I, T), D | 0001 0001 | xxyy 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| movsXY O(, I, T), D | 0001 0001 | xxyy 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| movsXY O, D | 0001 0001 | xxyy 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### `movzX` — Move with Zero-Extension

**Description.** Copies the content of the source operand to the destination operand and zero-extends the value to 16, 32, or 64 bits. The size of the converted value depends on the operand-size attributes, as specified by the second size suffix added to the instruction's mnemonic.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `movzXY S, D` | `0001 0010` | `xxyy 0000` | `00-- ----` | `ssss dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `movzXY (B), D` | `0001 0010` | `xxyy 0010` | `10-- ----` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `movzXY (B, I, T), D` | `0001 0010` | `xxyy 0010` | `11tt iiii` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `movzXY O(B, I , T), D` | `0001 0010` | `xxyy 1010` | `11tt iiii` | `bbbb dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `movzXY (, I, T), D` | `0001 0010` | `xxyy 0010` | `01tt iiii` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `movzXY O(, I, T), D` | `0001 0010` | `xxyy 1010` | `01tt iiii` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `movzXY O, D` | `0001 0010` | `xxyy 1010` | `00-- ----` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |

### `lea` — Load Effective Address

**Description.** Computes the effective address of the source operand and stores it in the destination operand. The source operand is a memory address specified with one of the addressing modes; the destination operand is a general-purpose register.

**Operation.** The operation of this instruction depends on whether memory operands are involved or not, and what fields of the general addressing method are used to identify memory locations.

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| **Instruction** | **Encoding** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `leaX (B), D` | `0001 0011` | `--xx 0010` | `10-- ----` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `leaX (B, I, T), D` | `0001 0011` | `--xx 0010` | `11tt iiii` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `leaX O(B, I , T), D` | `0001 0011` | `--xx 1010` | `11tt iiii` | `bbbb dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `leaX (, I, T), D` | `0001 0011` | `--xx 0010` | `01tt iiii` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `leaX O(, I, T), D` | `0001 0011` | `--xx 1010` | `01tt iiii` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `leaX O, D` | `0001 0011` | `--xx 1010` | `00-- ----` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |

### `push` — Push byte, word, longword or quadword onto the stack

**Description.** Decrements the stack pointer and then stores the source operand on the top of the stack. The operand size determines the amount the stack pointer is decremented (1, 2, 4, or 8 bytes). If the source operand is an immediate value, and its size is less than the one expressed by the instruction's suffix, then its value is sign-extended onto the stack.

The instruction `push RSP` pushes the value of the RSP register as before the instruction was executed, thus if a `push` instruction uses a memory operand in which the RSP register is used for computing the operand address, it is computed before the RSP register is decremented.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| **Instruction** | **Encoding** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `pushX S` | `0001 0100` | `xxxx 0000` | `00-- ----` | `ssss dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `pushX (B)` | `0001 0100` | `xxxx 0010` | `10-- ----` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `pushX (B, I, T)` | `0001 0100` | `xxxx 0010` | `11tt iiii` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `pushX O(B, I , T)` | `0001 0100` | `xxxx 1010` | `11tt iiii` | `bbbb dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `pushX (, I, T)` | `0001 0100` | `xxxx 0010` | `01tt iiii` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `pushX O(, I, T)` | `0001 0100` | `xxxx 1010` | `01tt iiii` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `pushX O` | `0001 0100` | `xxxx 1010` | `00-- ----` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |

### `pop` — Pop byte, word, longword or quadword from the stack

**Description.** Loads the value from the top of the stack to the location specified with the destination operand, and then increments the stack pointer. The destination operand can be a general-purpose register, or a memory location. The operand size determines the amount the stack pointer is decremented (1, 2, 4, or 8 bytes). If the source operand is an immediate value, and its size is less than the one expressed by the instruction's suffix, then its value is sign-extended onto the stack.

If the RSP register is used for addressing a destination operand in memory, the `pop` instruction computes the effective address of the operand after it increments the ESP register. In case the pop

operation empties the stack (namely, the value of RSP becomes 0x00), this value is used for computing the memory location. It is left to the programmer to check whether the stack is empty or not, before executing a pop operation.

The instruction pop RSP increments the stack pointer before data at the old top of stack is written into the destination.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| popX S | 0001 0101 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| popX (B) | 0001 0101 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| popX (B, I, T) | 0001 0101 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| popX O(B, I , T) | 0001 0101 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| popX (, I, T) | 0001 0101 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| popX O(, I, T) | 0001 0101 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| popX O | 0001 0101 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### pushf — Push FLAGS register onto the Stack

**Description.** Decrements the stack pointer by a number of bytes specified by the instruction suffix, and pushes the entire content of the FLAGS register. When specifying a push of size 4 or 8 bytes, the remaining bits of the pushed register are zeroed. An instruction suffix specifying a byte size will generate a syntax error.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| pushfw | 0001 0110 | xx01 ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| pushfl | 0001 0110 | xx10 ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| pushfq | 0001 0110 | xx11 ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `popf` — Pop FLAGS register onto the Stack

**Description.** Pops a word, a longword, or a doubleword (depending on the instruction suffix) from the top of the stack, and stores the value in the FLAGS registers. Then increments the value of the stack pointer. These instructions reverse the operation of `pushfX` instructions.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | | $Imm_{64}$ |
| `popfw` | 0001 0111 | xx01 ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `popfl` | 0001 0111 | xx10 ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `popfq` | 0001 0111 | xx11 ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `movs` — Move Data from String to String

**Description.** Moves the byte, word, longword, or quadword (depending on the instruction suffix) specified with the source operand to the location specified with the destination operand. Both the source and destination operands are located in memory. The address of the source operand is read from the RSI register. The address of the destination operand is read from the RDI register. The registers, must be loaded correctly before the move string instruction is executed.

After the move operation, the source and destination registers are incremented or decremented automatically according to the setting of the DF flag in the FLAGS register. (If the DF flag is 0, the registers are incremented; if the DF flag is 1, they are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for longword operations, or by 8 for quadword operations. Additionally, the RCX register is always decremented by 1.

The value of the RIP register is not updated until the RCX register reaches the value 0, meaning that the `movs` instruction can perform a memory-to-memory copy of any given size. It is important to note that this instruction works on a (byte/word/longword/quadword) block basis, therefore if the destination memory area overlaps the source area, care must be taken when using it, as it could produce an undefined result.

An example to copy (in forward mode) a block of `size` bytes from address `S` to address `D` on a 8-bytes block basis is provided below:

```
movq $S, %rsi
movq $D, %rdi
movl $size/8, %rcx
```

```
cld
movsq
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| movsX | 0001 1000 | xx-- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `stos` — Store String

**Description.** Stores a byte, word, longword or quadword (depending on the instruction suffix) from the AL, AX, EAX or RAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the RDI register. The registers, must be loaded correctly before the store string instruction is executed.

After the store operation, the destination register is incremented or decremented automatically according to the setting of the DF flag in the FLAGS register. (If the DF flag is 0, the register is incremented; if the DF flag is 1, it is decremented.) The register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for longword operations, or by 8 for quadword operations. Additionally, the RCX register is always decremented by 1.

The value of the RIP register is not updated until the RCX register reaches the value 0, meaning that the `stos` instruction can perform a memory initialization of any given size.

An example to initialize to zero (in forward mode) a block of `size` bytes starting at address `D` on a 8-bytes block basis is provided below:

```
movq $0x0, %rax
movq $D, %rdi
movl $size/8, %rcx
cld
stosq
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| stosX | 0001 1001 | xx-- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

## A.3 Class 2: Arithmetic and Logical Instructions

The instructions belonging to this class allow to execute arithmetic operations on binary integer numbers represented in two's complement. The z64 CPU is equipped with an ALU which supports only addition and subtraction, while multiplication and division should be implemented either with ad-hoc software routines, or using a Floating Point Unit (FPU) which is connected to the CPU as an external device, resembling a mathematical coprocessor (see Appendix B). Along with these mathematical operations, the z64 ALU supports the execution of most common logical operations. They are bitwise operations, meaning that the result is computed by executing the elementary logical operation on the corresponding bit of the source and destination operands. The execution of these instructions can update the value of the bits in the FLAGS register.

Similarly to data movement instructions, class 2 instruction can handle any combination of source/destination operands, with the only constraint that only one memory operand can appear in one instruction. To operate on two different memory operands, one of them must be first explicitly loaded into a general purpose register.

All instructions of this class have the Class field of the opcode set to 2. The available instructions in this class are reported in Table A.4, along with the values of the Type value, and the operands classes.

**Table A.4:** Class 2 Instructions

| Type | Mnemonic | Operands | O | S | Z | P | C | Description |
|---|---|---|---|---|---|---|---|---|
| 0 | add | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Stores in E the result of E + B |
| 1 | sub | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Stores in E the result of E - B |
| 2 | adc | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Stores in E the result of E + B + CF |
| 3 | sbb | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Stores in E the result of E - (B + neg(CF)) |
| 4 | cmp | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Compares the value of B and E by evaluating E - B, the result is then discarded |
| 4 | test | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Computes the bit-wise logical and of B and E, the result is then discarded |
| 5 | neg | E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Replaces the value of E with its two's complement |
| 6 | and | B, E | 0 | ⇕ | ⇕ | ⇕ | 0 | Stores in E the result of the bit-wise and between B and E |
| 7 | or | B, E | 0 | ⇕ | ⇕ | ⇕ | 0 | Stores in E the result of the bit-wise or between B and E |
| 8 | xor | B, E | 0 | ⇕ | ⇕ | ⇕ | 0 | Stores in E the result of the bit-wise xor between B and E |
| 9 | not | E | 0 | ⇕ | ⇕ | ⇕ | 0 | Replaces the value of E with its one's complement |

### `add` — Add

**Description.** Adds the destination operand and the source operand and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one `add` instruction, and there is no other instruction to perform such an operation.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format (as also specified by the instruction suffix).

The `add` instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the `OF` and `CF` flags to indicate an overflow (carry) in the signed or unsigned result, respectively. The `SF` flag indicates the sign of the signed result.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The `OF`, `SF`, `ZF`, `CF`, and `PF` flags are set according to the result of the operation.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `addX S, D` | 0010 0000 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX S, (B)` | 0010 0000 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX S, (B, I, T)` | 0010 0000 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX S, O(B, I, T)` | 0010 0000 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `addX S, (, I, T)` | 0010 0000 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX S, O(, I, T)` | 0010 0000 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `addX S, O` | 0010 0000 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `addX I, D` | 0010 0000 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `addX I, (B)` | 0010 0000 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `addX I, (B, I, T)` | 0010 0000 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `addX I, O(B, I, T)` | 0010 0000 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| `addX I, (, I, T)` | 0010 0000 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `addX I, O(, I, T)` | 0010 0000 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| `addX I, O` | 0010 0000 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| `addX (B), D` | 0010 0000 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX (B, I, T), D` | 0010 0000 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX O(B, I, T), D` | 0010 0000 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `addX (, I, T), D` | 0010 0000 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `addX O(, I, T), D` | 0010 0000 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `addX O, D` | 0010 0000 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### `sub` — Subtract

**Description.** Subtracts the source operand from the destination operand and stores the result in the destination operand. The destination operand can be a register or a memory location; the source

operand can be an immediate, register, or memory location. However, two memory operands cannot be used in one instruction. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The `sub` instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the `OF` and `CF` flags to indicate an overflow in the signed or unsigned result, respectively. The `SF` flag indicates the sign of the signed result.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The `OF`, `SF`, `ZF`, `CF`, and `PF` flags are set according to the result of the operation.

**Binary Encoding.**

| **Instruction** | **Encoding** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `subX S, D` | 0010 0001 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX S, (B)` | 0010 0001 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX S, (B, I, T)` | 0010 0001 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX S, O(B, I, T)` | 0010 0001 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `subX S, (, I, T)` | 0010 0001 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX S, O(, I, T)` | 0010 0001 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `subX S, O` | 0010 0001 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `subX I, D` | 0010 0001 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `subX I, (B)` | 0010 0001 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `subX I, (B, I, T)` | 0010 0001 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `subX I, O(B, I, T)` | 0010 0001 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| `subX I, (, I, T)` | 0010 0001 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| `subX I, O(, I, T)` | 0010 0001 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| `subX I, O` | 0010 0001 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| `subX (B), D` | 0010 0001 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX (B, I, T), D` | 0010 0001 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX O(B, I, T), D` | 0010 0001 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `subX (, I, T), D` | 0010 0001 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| `subX O(, I, T), D` | 0010 0001 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| `subX O, D` | 0010 0001 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### `adc` — Add with Carry

**Description.** Adds the destination operand, the source operand, and the carry flag (`CF`), then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the `CF` flag represents a carry from a previous operation (often, an addition). When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The adc instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The adc instruction is usually executed as part of a multibyte or multiword addition in which an add instruction is followed by an adc instruction.

For example, to perform a sum using 128-bits integers, the value of the original integers can be split into two couples of general purpose registers like in the following example:

```
movq $operand_1_high, %rax
movq $operand_1_low, %rbx
movq $operand_2_high, %rcx
movq $operand_2_low, %rdx
addq %rbx, %rdx
adcq %rax, %rcx
```

so that the registers RDX and RCX will keep the high bits and low bits (respectively) of the result of the sum.

**Operation.**

```
MAR ← RIP

MDR ← (MAR); RIP ← RIP + 8

IR ← MDR
```

**Status flags affected.** The OF, SF, ZF, CF, and PF flags are set according to the result of the operation.

**Binary Encoding.**

| **Instruction** | **Encoding** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `adcX S, D` | `0010 0010` | `xxxx 0000` | `00-- ----` | `ssss dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX S, (B)` | `0010 0010` | `xxxx 0001` | `10-- ----` | `ssss bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX S, (B, I, T)` | `0010 0010` | `xxxx 0001` | `11tt iiii` | `ssss bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX S, O(B, I, T)` | `0010 0010` | `xxxx 1001` | `11tt iiii` | `ssss bbbb` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `adcX S, (, I, T)` | `0010 0010` | `xxxx 0001` | `01tt iiii` | `ssss ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX S, O(, I, T)` | `0010 0010` | `xxxx 1001` | `01tt iiii` | `ssss ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `adcX S, O` | `0010 0010` | `xxxx 1001` | `00-- ----` | `ssss ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `adcX I, D` | `0010 0010` | `xxxx 0100` | `00-- ----` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `adcX I, (B)` | `0010 0010` | `xxxx 0101` | `10-- ----` | `---- bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `adcX I, (B, I, T)` | `0010 0010` | `xxxx 0101` | `11tt iiii` | `---- bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `adcX I, O(B, I, T)` | `0010 0010` | `xxxx 1101` | `11tt iiii` | `---- bbbb` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `adcX I, (, I, T)` | `0010 0010` | `xxxx 0101` | `01tt iiii` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `adcX I, O(, I, T)` | `0010 0010` | `xxxx 1101` | `01tt iiii` | `---- ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `adcX I, O` | `0010 0010` | `xxxx 1101` | `00-- ----` | `---- ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `adcX (B), D` | `0010 0010` | `xxxx 0010` | `10-- ----` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX (B, I, T), D` | `0010 0010` | `xxxx 0010` | `11tt iiii` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX O(B, I, T), D` | `0010 0010` | `xxxx 1010` | `11tt iiii` | `bbbb dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `adcX (, I, T), D` | `0010 0010` | `xxxx 0010` | `01tt iiii` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `adcX O(, I, T), D` | `0010 0010` | `xxxx 1010` | `01tt iiii` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `adcX O, D` | `0010 0010` | `xxxx 1010` | `00-- ----` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |

### `sbb` — Integer Subtraction with Borrow

**Description.** Adds the source operand and the complement of the carry (`CF`) flag, and subtracts the result from the destination operand. The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. However, two memory operands cannot be used in one instruction. The state of the `CF` flag represents the complement of a borrow from a previous operation (often a subtraction), as discussed in Section 5.4.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The `sbb` instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the `OF` and `CF` flags to indicate a borrow in the signed or unsigned result, respectively. The `SF` flag indicates the sign of the signed result. The `sbb` instruction is usually executed as part of a multibyte or multiword subtraction in which a `sub` instruction is followed by a `sbb` instruction. For example, to perform a subtraction using 128-bits integers, the value of the original integers can be split into two couples of general purpose registers like in the following example:

```
movq $operand_1_high, %rax
movq $operand_1_low, %rbx
movq $operand_2_high, %rcx
movq $operand_2_low, %rdx
subq %rax, %rcx
sbbq %rbx, %rdx
```

so that the registers RDX and RCX will keep the high bits and low bits (respectively) of the result of the subtraction.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The OF, SF, ZF, CF, and PF flags are set according to the result of the operation.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| sbbX S, D | 0010 0011 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX S, (B) | 0010 0011 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX S, (B, I, T) | 0010 0011 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX S, O(B, I, T) | 0010 0011 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| sbbX S, (, I, T) | 0010 0011 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX S, O(, I, T) | 0010 0011 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| sbbX S, O | 0010 0011 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| sbbX I, D | 0010 0011 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| sbbX I, (B) | 0010 0011 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| sbbX I, (B, I, T) | 0010 0011 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| sbbX I, O(B, I, T) | 0010 0011 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| sbbX I, (, I, T) | 0010 0011 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| sbbX I, O(, I, T) | 0010 0011 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| sbbX I, O | 0010 0011 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| sbbX (B), D | 0010 0011 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX (B, I, T), D | 0010 0011 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX O(B, I, T), D | 0010 0011 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| sbbX (, I, T), D | 0010 0011 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sbbX O(, I, T), D | 0010 0011 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| sbbX O, D | 0010 0011 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### cmp — Compare two Operands

**Description.** Compares the first source operand with the second source operand and sets the status flags in the FLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand. The result is then discarded.

The outcome of the cmp instruction can be used by relying on jX and jnX instructions.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The OF, SF, ZF, CF, and PF flags are set according to the result of the operation.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `cmpX S, D` | `0010 0100` | `xxxx 0000` | `00-- ----` | `ssss dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX S, (B)` | `0010 0100` | `xxxx 0001` | `10-- ----` | `ssss bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX S, (B, I, T)` | `0010 0100` | `xxxx 0001` | `11tt iiii` | `ssss bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX S, O(B, I, T)` | `0010 0100` | `xxxx 1001` | `11tt iiii` | `ssss bbbb` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `cmpX S, (, I, T)` | `0010 0100` | `xxxx 0001` | `01tt iiii` | `ssss ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX S, O(, I, T)` | `0010 0100` | `xxxx 1001` | `01tt iiii` | `ssss ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `cmpX S, O` | `0010 0100` | `xxxx 1001` | `00-- ----` | `ssss ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `cmpX I, D` | `0010 0100` | `xxxx 0100` | `00-- ----` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `cmpX I, (B)` | `0010 0100` | `xxxx 0101` | `10-- ----` | `---- bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `cmpX I, (B, I, T)` | `0010 0100` | `xxxx 0101` | `11tt iiii` | `---- bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `cmpX I, O(B, I, T)` | `0010 0100` | `xxxx 1101` | `11tt iiii` | `---- bbbb` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `cmpX I, (, I, T)` | `0010 0100` | `xxxx 0101` | `01tt iiii` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `cmpX I, O(, I, T)` | `0010 0100` | `xxxx 1101` | `01tt iiii` | `---- ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `cmpX I, O` | `0010 0100` | `xxxx 1101` | `00-- ----` | `---- ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `cmpX (B), D` | `0010 0100` | `xxxx 0010` | `10-- ----` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX (B, I, T), D` | `0010 0100` | `xxxx 0010` | `11tt iiii` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX O(B, I, T), D` | `0010 0100` | `xxxx 1010` | `11tt iiii` | `bbbb dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `cmpX (, I, T), D` | `0010 0100` | `xxxx 0010` | `01tt iiii` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `cmpX O(, I, T), D` | `0010 0100` | `xxxx 1010` | `01tt iiii` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `cmpX O, D` | `0010 0100` | `xxxx 1010` | `00-- ----` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |

### `test` — Logical Compare

**Description.** Computes the bit-wise logical and of first source operand and the second source operand and sets the `SF`, `ZF`, and `PF` status flags according to the result. The result is then discarded. The `CF` and `OF` flags are set to zero, `SF` is set to the most significant bit of the result of the and. If the result of the and is 0, then the `ZF` flag is set to 1, otherwise it is set to 0. The parity flag is set to the bitwise xnor of the result of the and. If an immediate is used as one source operand, its value is zero extended to the size of the other operand.

The outcome of the `test` instruction can be used by relying on `jX` and `jnX` instructions. This instruction is particularly useful to test the value of specific bits. For example, the following code snippet

```
testq $64, %rax
jz is_set
```

checks whether bit number 6 ($2^6 = 64$) of the RAX register is set to 1 or not. Similarly, if the programmer wants to check if a particular register is set to zero, he could use the following code

```
testq %rax, %rax
jz is_zero
```

which is semantically equivalent to

```
cmpq $0, %rax
jz is_set
```

but produces shorter instructions, which require less memory in the final executable, due to the fact that the immediate operand o is not stored into the instruction[2].

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The OF, SF, ZF, CF, and PF flags are set according to the result of the operation.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| testX S, D | 0010 0101 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX S, (B) | 0010 0101 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX S, (B, I, T) | 0010 0101 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX S, O(B, I, T) | 0010 0101 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| testX S, (, I, T) | 0010 0101 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX S, O(, I, T) | 0010 0101 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| testX S, O | 0010 0101 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| testX I, D | 0010 0101 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| testX I, (B) | 0010 0101 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| testX I, (B, I, T) | 0010 0101 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| testX I, O(B, I, T) | 0010 0101 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| testX I, (, I, T) | 0010 0101 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| testX I, O(, I, T) | 0010 0101 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| testX I, O | 0010 0101 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| testX (B), D | 0010 0101 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX (B, I, T), D | 0010 0101 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX O(B, I, T), D | 0010 0101 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| testX (, I, T), D | 0010 0101 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| testX O(, I, T), D | 0010 0101 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| testX O, D | 0010 0101 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### neg — Two's Complement Negation

**Description.** Replaces the value of the destination operand with its two's complement. This operation is equivalent to subtracting the operand from o. The destination operand is located in a general-purpose register or a memory location.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

[2] This programming style is similar to resetting a register to zero using the instruction `xorq %rax, %rax` rather than `movq $0, %rax`.

**Status flags affected.** The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, and PF flags are set according to the result.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| negX S | 0010 0110 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| negX (B) | 0010 0110 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| negX (B, I, T) | 0010 0110 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| negX O(B, I , T) | 0010 0110 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| negX (, I, T) | 0010 0110 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| negX O(, I, T) | 0010 0110 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| negX O | 0010 0110 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### and — Logical And

**Description.** Performs a bit-wise and operation on the destination and source operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. However, two memory operands cannot be used in one instruction. Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| andX S, D | 0010 0111 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX S, (B) | 0010 0111 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX S, (B, I, T) | 0010 0111 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX S, O(B, I, T) | 0010 0111 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| andX S, (, I, T) | 0010 0111 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX S, O(, I, T) | 0010 0111 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| andX S, O | 0010 0111 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| andX I, D | 0010 0111 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| andX I, (B) | 0010 0111 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| andX I, (B, I, T) | 0010 0111 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| andX I, O(B, I, T) | 0010 0111 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| andX I, (, I, T) | 0010 0111 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| andX I, O(, I, T) | 0010 0111 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| andX I, O | 0010 0111 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| andX (B), D | 0010 0111 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX (B, I, T), D | 0010 0111 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX O(B, I, T), D | 0010 0111 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| andX (, I, T), D | 0010 0111 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| andX O(, I, T), D | 0010 0111 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| andX O, D | 0010 0111 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### `or` — Logical Inclusive Or

**Description.** Performs a bit-wise inclusive or operation between the destination and source operands and stores the result in the destination operand. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. However, two memory operands cannot be used in one instruction. Each bit of the result of the `or` instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The `OF` and `CF` flags are cleared; the `SF`, `ZF`, and `PF` flags are set according to the result.

**Binary Encoding.**

| **Instruction** | **Encoding** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `orX S, D` | `0010 1000` | `xxxx 0000` | `00-- ----` | `ssss dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX S, (B)` | `0010 1000` | `xxxx 0001` | `10-- ----` | `ssss bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX S, (B, I, T)` | `0010 1000` | `xxxx 0001` | `11tt iiii` | `ssss bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX S, O(B, I, T)` | `0010 1000` | `xxxx 1001` | `11tt iiii` | `ssss bbbb` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `orX S, (, I, T)` | `0010 1000` | `xxxx 0001` | `01tt iiii` | `ssss ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX S, O(, I, T)` | `0010 1000` | `xxxx 1001` | `01tt iiii` | `ssss ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `orX S, O` | `0010 1000` | `xxxx 1001` | `00-- ----` | `ssss ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `orX I, D` | `0010 1000` | `xxxx 0100` | `00-- ----` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `orX I, (B)` | `0010 1000` | `xxxx 0101` | `10-- ----` | `---- bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `orX I, (B, I, T)` | `0010 1000` | `xxxx 0101` | `11tt iiii` | `---- bbbb` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `orX I, O(B, I, T)` | `0010 1000` | `xxxx 1101` | `11tt iiii` | `---- bbbb` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `orX I, (, I, T)` | `0010 1000` | `xxxx 0101` | `01tt iiii` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✓ |
| `orX I, O(, I, T)` | `0010 1000` | `xxxx 1101` | `01tt iiii` | `---- ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `orX I, O` | `0010 1000` | `xxxx 1101` | `00-- ----` | `---- ----` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✓ |
| `orX (B), D` | `0010 1000` | `xxxx 0010` | `10-- ----` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX (B, I, T), D` | `0010 1000` | `xxxx 0010` | `11tt iiii` | `bbbb dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX O(B, I, T), D` | `0010 1000` | `xxxx 1010` | `11tt iiii` | `bbbb dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `orX (, I, T), D` | `0010 1000` | `xxxx 0010` | `01tt iiii` | `---- dddd` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |
| `orX O(, I, T), D` | `0010 1000` | `xxxx 1010` | `01tt iiii` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |
| `orX O, D` | `0010 1000` | `xxxx 1010` | `00-- ----` | `---- dddd` | `oooo oooo` | `oooo oooo` | `oooo oooo` | `oooo oooo` | ✗ |

### `xor` — Logical Exclusive Or

**Description.** Performs a bit-wise logical exclusive-or (xor) operation on the source operand and the destination operand and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The `OF` and `CF` flags are cleared; the `SF`, `ZF`, and `PF` flags are set according to the result.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| xorX S, D | 0010 1001 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX S, (B) | 0010 1001 | xxxx 0001 | 10-- ---- | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX S, (B, I, T) | 0010 1001 | xxxx 0001 | 11tt iiii | ssss bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX S, O(B, I, T) | 0010 1001 | xxxx 1001 | 11tt iiii | ssss bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| xorX S, (, I, T) | 0010 1001 | xxxx 0001 | 01tt iiii | ssss ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX S, O(, I, T) | 0010 1001 | xxxx 1001 | 01tt iiii | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| xorX S, O | 0010 1001 | xxxx 1001 | 00-- ---- | ssss ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| xorX I, D | 0010 1001 | xxxx 0100 | 00-- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| xorX I, (B) | 0010 1001 | xxxx 0101 | 10-- ---- | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| xorX I, (B, I, T) | 0010 1001 | xxxx 0101 | 11tt iiii | ---- bbbb | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| xorX I, O(B, I, T) | 0010 1001 | xxxx 1101 | 11tt iiii | ---- bbbb | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| xorX I, (, I, T) | 0010 1001 | xxxx 0101 | 01tt iiii | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✓ |
| xorX I, O(, I, T) | 0010 1001 | xxxx 1101 | 01tt iiii | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| xorX I, O | 0010 1001 | xxxx 1101 | 00-- ---- | ---- ---- | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✓ |
| xorX (B), D | 0010 1001 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX (B, I, T), D | 0010 1001 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX O(B, I, T), D | 0010 1001 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| xorX (, I, T), D | 0010 1001 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| xorX O(, I, T), D | 0010 1001 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| xorX O, D | 0010 1001 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

### `not` — One's Complement Negation

**Description.** Performs a bit-wise not operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The `OF` and `CF` flags are cleared; the `SF`, `ZF`, and `PF` flags are set according to the result.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| notX S | 0010 1010 | xxxx 0000 | 00-- ---- | ssss dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| notX (B) | 0010 1010 | xxxx 0010 | 10-- ---- | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| notX (B, I, T) | 0010 1010 | xxxx 0010 | 11tt iiii | bbbb dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| notX O(B, I , T) | 0010 1010 | xxxx 1010 | 11tt iiii | bbbb dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| notX (, I, T) | 0010 1010 | xxxx 0010 | 01tt iiii | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| notX O(, I, T) | 0010 1010 | xxxx 1010 | 01tt iiii | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |
| notX O | 0010 1010 | xxxx 1010 | 00-- ---- | ---- dddd | oooo oooo | oooo oooo | oooo oooo | oooo oooo | ✗ |

## A.4 Class 3: Rotate and Shift Instructions

Rotate and shift instructions are bitwise operations that modify the content of their single operand, which can be a general purpose register only. They modify the order of the bits, and are mainly used to perform multiplications and divisions (by powers of two), to test the value of particular bits of the operand, or in more complex algorithms (like graphics, or cryptography). Rotate and shift instructions can move the bits to the right or to the left. Shift instructions belong to two main classes:

- *logical*: they shift the bits of the operand towards the specified direction for the specified number of bit positions, with zeros entering from the other direction. The `CF` flag receives the last bit leaving the operand.

- *arithmetic*: they shift the bits of the operand towards the specified direction for the specified number of bit position, respecting the arithmetic two's complement representation rule of the sign. They therefore ensure sign extension when shifting to the right, and overflow control when shifting to the left.

It is important to remember that shifting an operand to the right of $k$ positions is equivalent to dividing its value by $2^k$, and similarly shifting it to the left by $k$ positions is equivalent to multiplying its value by $2^k$. Previously, Figure 3.1 illustrated the operation of the various instructions of this class, while Table A.5 summarises the instructions and their operands.

**Table A.5:** Class 3 Instructions

| Type | Mnemonic | Operands | O | S | Z | P | C | Description |
|---|---|---|---|---|---|---|---|---|
| 0 | `sal` | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Multiply by 2, $k$ times |
| 1 | `sal` | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Multiply by 2, RCX times |
| 0 | `shl` | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Multiply by 2, $k$ times |
| 1 | `shl` | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Multiply by 2, RCX times |
| 2 | `sar` | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Signed divide by 2, $k$ times |
| 3 | `sar` | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Signed divide by 2, RCX times |
| 4 | `shr` | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Unsigned divide by 2, $k$ times |
| 5 | `shr` | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Unsigned divide by 2, RCX times |
| 6 | `rcl` | K, G | ⇕ | - | - | - | ⇕ | Rotate left, $k$ times |
| 7 | `rcl` | G | ⇕ | - | - | - | ⇕ | Rotate left, RCX times |
| 8 | `rcr` | K, G | ⇕ | - | - | - | ⇕ | Rotate right, $k$ times |
| 9 | `rcr` | G | ⇕ | - | - | - | ⇕ | Rotate right, RCX times |
| 10 | `rol` | K, G | ⇕ | - | - | - | ⇕ | Rotate left, $k$ times |
| 11 | `rol` | G | ⇕ | - | - | - | ⇕ | Rotate left, RCX times |
| 12 | `ror` | K, G | ⇕ | - | - | - | ⇕ | Rotate right, $k$ times |
| 13 | `ror` | G | ⇕ | - | - | - | ⇕ | Rotate right, RCX times |

Concerning `sal` and `shl`, they are only two different mnemonics for the same instructions, since the execution of the instructions involves the same steps, and they are in fact encoded with the same Type field. Additionally, for each instruction, two variants exist, one explicitly accepting the number of bit positions to shift/rotate (class `K` operand), and one using the register RCX to read this value. The maximum allowed value for `K` is 64, and its binary representation is stored in the `Displacement` field of the instruction.

`sal/sar/shl/shr` — **Shift**

**Description.** Shifts the bits in the destination operand to the left or right by the number of bits specified in the second operand, or in the RCX register. Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can only be a register. The count operand can be an immediate value or the RCX register.

The shift arithmetic left (`sal`) and shift logical left (`shl`) instructions perform the same operation, and are mapped to the same binary encoding. They shift the bits in the destination operand to the left (towards more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared

The shift arithmetic right (`sar`) and shift logical right (`shr`) instructions shift the bits of the destination operand to the right (towards less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The `shr` instruction clears the most significant bit (see Figure 3.1); the `sar` instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand.

The `sar` and `shr` instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the `sar` instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the `sar` instruction provides a result which is different from that of real arithmetic. The "quotient" of the `sar` instruction is rounded towards negative infinity, which is apparent only for negative numbers. For example, when dividing -9 by 4, the result is -2 with a remainder of -1. If the `sar` instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the `sar` instruction stores only the most significant bit of the remainder in the CF flag.

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the `sar` instruction, the OF flag is cleared for all 1-bit shifts. For the `shr` instruction, the OF flag is set to the most-significant bit of the original operand.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for `shl` and `shr` instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts; otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| salX K, D | 0011 0000 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| salX D | 0011 0001 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| shlX K, D | 0011 0000 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| shlX D | 0011 0001 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| sarX K, D | 0011 0010 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| sarX D | 0011 0011 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| shrX K, D | 0011 0100 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| shrX D | 0011 0101 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `rcl`/`rcr`/`rol`/`ror` — Rotate

**Description.** Rotates the bits of the destination operand the number of bit positions specified in the count operand and stores the result in the destination operand. The destination operand can only be a register; the count operand is an unsigned integer that can be an immediate or a value in the RCX register.

The rotate left (`rol`) and rotate through carry left (`rcl`) instructions shift all the bits towards more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (`ror`) and rotate through carry right (`rcr`) instructions shift all the bits towards less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The `rcl` and `rcr` instructions include the CF flag in the rotation. The `rcl` instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The `rcr` instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the `rol` and `ror` instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other. See Figure 3.1 for a visualization of the operations.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except `rcl` and `rcr` instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive or of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive or of the two most-significant bits of the result.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates, it is undefined for multi-bit rotates. The SF, ZF, and PF flags are not affected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| rclX K, D | 0011 1000 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| rclX D | 0011 1001 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| rcrX K, D | 0011 1010 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| rcrX D | 0011 1011 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| rolX K, D | 0011 1100 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| rolX D | 0011 1101 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |
| rorX K, D | 0011 1110 | xxxx ---- | ---- ---- | ---- dddd | 0000 0000 | 0000 0000 | 0000 0000 | 0kkk kkkk | ✗ |
| rorX D | 0011 1111 | xxxx ---- | ---- ---- | ---- dddd | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

## A.5 Class 4: Flag Bits Manipulation Instructions

Status and control bits can be manipulated relying on instructions belonging to this class, which allow to set or reset each one individually.

**Table A.6:** Class 4 Instructions

| Type | Mnemonic | Operands | O S Z P C | Description |
|---|---|---|---|---|
| 0 | clc | – | - - - - 0 | |
| 1 | clp | – | - - - 0 - | |
| 2 | clz | – | - - 0 - - | |
| 3 | cls | – | - 0 - - - | |
| 4 | cli | – | - - - - - | |
| 5 | cld | – | - - - - - | |
| 6 | clo | – | 0 - - - - | |
| 7 | stc | – | - - - - 1 | |
| 8 | stp | – | - - - 1 - | |
| 9 | stz | – | - - 1 - - | |
| 10 | sts | – | - 1 - - - | |
| 11 | sti | – | - - - - - | |
| 12 | std | – | - - - - - | |
| 13 | sto | – | 1 - - - - | |

### `clc` — Clear Carry Flag

**Description.** Clears the `CF` flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[CF] ← 0
```

**Status flags affected.** The `CF` flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `clc` | 0100 0000 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `clp` — Clear Parity Flag

**Description.** Clears the PF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[PF] ← 0
```

**Status flags affected.** The PF flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `clp` | 0100 0001 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `clz` — Clear Zero Flag

**Description.** Clears the ZF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[ZF] ← 0
```

**Status flags affected.** The ZF flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `clz` | 0100 0010 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `cls` — Clear Sign Flag

**Description.** Clears the SF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[SF] ← 0
```

**Status flags affected.** The SF flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `cls` | 0100 0011 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `cli` — Clear Interrupt Flag

**Description.** `cli` clears the IF flag in the FLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore external interrupts.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[IF] ← 0
```

**Status flags affected.** The IF flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `cli` | 0100 0100 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `cld` — Clear Direction Flag

**Description.** Clears the DF flag in the FLAGS register. When the DF flag is set to 0, string operations increment the index registers (RSI and/or RDI).

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[DF] ← 0
```

**Status flags affected.** The DF flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| cld | 0100 0101 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### clo — Clear Overflow Flag

**Description.** Clears the OF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[OF] ← 0
```

**Status flags affected.** The OF flag is set to 0. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| clo | 0100 0110 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### stc — Set Carry Flag

**Description.** Sets the CF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[CF] ← 1
```

**Status flags affected.** The CF flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `stc` | 0100 1000 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

## `stp` — Set Parity Flag

**Description.** Sets the `PF` flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[PF] ← 1
```

**Status flags affected.** The `PF` flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `stp` | 0100 1001 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

## `stz` — Set Zero Flag

**Description.** Sets the `ZF` flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[ZF] ← 1
```

**Status flags affected.** The `ZF` flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `stz` | 0100 1010 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `sts` — Set Sign Flag

**Description.** Sets the SF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[SF] ← 1
```

**Status flags affected.** The SF flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `sts` | 0100 1011 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `sti` — Set Interrupt Flag

**Description.** Sets the interrupt flag (`IF`) in the FLAGS register. After the `IF` flag is set, the processor can respond to external interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if a `sti` instruction is followed by a `ret` instruction, the `ret` instruction is allowed to execute before external interrupts are recognized. If the `sti` instruction is followed by a `cli` instruction (which clears the `IF` flag), the effect of the `sti` instruction is negated.

The `sti` instruction delays recognition of interrupts only if it is executed with `IF` = 0. In a sequence of `sti` instructions, only the first instruction in the sequence is guaranteed to delay interrupts. In the following instruction sequence, interrupts may be recognized before `ret` executes:

```
sti
sti
ret
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[IF] ← 1
```

**Status flags affected.** The `IF` flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| sti | 0100 1100 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### std — Set Direction Flag

**Description.** Sets the DF flag in the FLAGS register. When the DF flag is set to 1, string operations decrement the index registers (RSI and/or RDI).

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[DF] ← 1
```

**Status flags affected.** The DF flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| std | 0100 1101 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### sto — Set Overflow Flag

**Description.** Sets the OF flag in the FLAGS register.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
FLAGS[OF] ← 1
```

**Status flags affected.** The OF flag is set to 1. Other flags are unaffected.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| std | 0100 1110 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

## A.6 Class 5: Program Flow Control Instructions

Instructions belonging to this class do not perform any operation on data, rather they are used to modify the sequential execution order of the program's instructions. This can be done by either jumping to a different instruction in the same procedure, or by *calling* another one.

All instructions of this class have the Class field of the opcode set to 5. The available instructions in this class are reported in Table A.7, along with the values of the Type value, and the operands classes.

La jmp assoluta

**Table A.7:** Class 5 Instructions

| Type | Mnemonic | Operands | O S Z P C | Description |
|---|---|---|---|---|
| 0 | jmp | M | - - - - - | Perform a relative jump |
| 1 | jmp | *G | - - - - - | Perform an absolute jump |
| 2 | call | M | - - - - - | Perform a relative subroutine call |
| 3 | call | *G | - - - - - | Perform an absolute subroutine call |
| 4 | ret | – | - - - - - | Return from subroutine |
| 5 | iret | – | ⇕ ⇕ ⇕ ⇕ ⇕ | Return from interrupt |

### jmp — Jump

**Description.** Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be a (signed) memory displacement, or a general-purpose register.

There are two different natures for a jmp instruction, as they can be either absolute or relative. A relative jmp find in its operand the (signed) memory displacement used to identify the memory address from which the program flow should restart after the instruction's execution is completed, while an absolute jmp has as the operand a general purpose register, where the destination memory address is already stored.

A relative offset is generally specified as a label in assembly code, but at machine-code level it is encoded as a signed 32-bits displacement. Concerning the binary encoding and operation, the different incarnations of a jmp instruction use different opcodes, and the target address in case of a relative jmp is represented as a displacement to be summed to value of RIP after the instruction fetch phase. On the other hand, an absolute jmp finds in the register operand the final memory address from which to continue the execution.

There is a difference in the way these instruction are written in an assembly program, which is related to the usage of the * operator, which is used by the absolute jmp and is placed before the register operand. Therefore in the following snippet the first instruction is relative, while the second is absolute:

```
jmp label
jmp *%rax
```

**Operation.**

MAR ← RIP

```
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
if is absolute jmp
     RIP ← RIP + DISPLACEMENT
else
     RIP ← REGISTER
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `jmp M` | `0101 0000` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jmp *R` | `0101 0001` | `---- ----` | `---- ----` | `---- rrrr` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |

### `call` — Call Procedure

**Description.** Saves the procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be a (signed) memory displacement, or a general purpose register.

When executing a call, the processor pushes the value of the RIP register (which contains the address of the instruction following the `call` instruction) on the stack, for later use as a return-instruction pointer. The processor then branches to the first address of the called procedure, by summing the displacement to the current value of RIP, or by replacing the current value of RIP with the one stored into the general purpose register specified as operand.

Similarly to the `jmp` instruction, there are both the absolute and relative `call` instruction, which are again associated at the assembly level with the presence or absence of the `*` operator:

```
call label
call *%rax
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
MAR ← RSP
MDR ← RIP
(MAR) ← MDR
RSP ← RSP - 8
if is absolute call
     RIP ← RIP + DISPLACEMENT
```

```
else
     RIP ← REGISTER
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `call M` | 0101 0010 | ---- ---- | ---- ---- | ---- ---- | mmmm mmmm | mmmm mmmm | mmmm mmmm | mmmm mmmm | ✗ |
| `call *R` | 0101 0011 | ---- ---- | ---- ---- | ---- rrrr | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `ret` — Return from Procedure

**Description.** Transfers program control back to a return address located on the top of the stack. The address is usually placed on the stack by a `call` instruction, and the return is made to the instruction that follows the `call` instruction.

The processor simply pops from the stack the return address into RIP, and begins program execution at the new address.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `ret` | 0101 0100 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### `iret` — Interrupt Return

**Description.** Returns program control from an interrupt handler to a program or procedure that was interrupted by an external interrupt, or a software-generated interrupt .

The processor pops the return instruction pointer, and FLAGS image from the stack to the RIP and `flags` registers, respectively, and then resumes execution of the interrupted program or procedure.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** All the flags and fields in the FLAGS register are potentially modified.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| ret | 0101 0101 | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

## A.7 Class 6: Conditional Flow Control Instructions

This class encompasses all the instructions which allow to alter the control flow of the program only if a particular condition (verified on any of the status flags in the FLAGS register) is met, differently from jump instructions belonging to Class 5 where the branch is always taken.

This class is fundamental, as it allows to implement both *selection* among different subportions of the program, and *iteration* of portions of the code, which are the two essential features required by a CPU to run any kind of algorithm.

All instructions of this class have the Class field of the opcode set to 6. The available instructions in this class are reported in Table A.8, along with the values of the Type value, and the operands classes.

**Table A.8:** Class 6 Instructions

| Type | Mnemonic | Operands | O S Z P C | Description |
|---|---|---|---|---|
| 0 | jc | M | - - - - - | Jump to M if CF is set |
| 1 | jp | M | - - - - - | Jump to M if PF is set |
| 2 | jz | M | - - - - - | Jump to M if ZF is set |
| 3 | js | M | - - - - - | Jump to M if SF is set |
| 4 | jo | M | - - - - - | Jump to M (displacement) if OF is set |
| 5 | jnc | M | - - - - - | Jump to M (displacement) if CF is not set |
| 6 | jnp | M | - - - - - | Jump to M (displacement) if PF is not set |
| 7 | jnz | M | - - - - - | Jump to M (displacement) if ZF is not set |
| 8 | jns | M | - - - - - | Jump to M (displacement) if SF is not set |
| 9 | jno | M | - - - - - | Jump to M (displacement) if OF is not set |

### jX — Jump if flag X is set

**Description.** Performs a jump instruction if status flag X in the FLAGS register is set. The destination of the jump can only be expressed as a longword displacement from the current value of RIP.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `jc M` | `0110 0000` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jp M` | `0110 0001` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jz M` | `0110 0010` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `js M` | `0110 0011` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jo M` | `0110 0100` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |

### `jnX` — Jump if flag X is not set

**Description.** Performs a jump instruction if status flag X in the FLAGS register is zero. The destination of the jump can only be expressed as a longword displacement from the current value of RIP.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | SIB | R/M | Displacement (Offset) | | | | $Imm_{64}$ |
| `jnc M` | `0110 0101` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jnp M` | `0110 0110` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jnz M` | `0110 0111` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jns M` | `0110 1000` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |
| `jno M` | `0110 1001` | `---- ----` | `---- ----` | `---- ----` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |

## A.8 Class 7: I/O Instructions

I/O instructions are used to exchange data with devices connected to the computer, and therefore are similar in logic to data movement instructions, but are different due to the fact that they require different steps to perform the data movement[3].

Instructions belonging to this class override the default format of the instructions, as it was presented in Figure A.1, due to the fact that an I/O port (16-bits wide) can be specified as an operand, rather than registers. The generic I/O instruction format is shown in Figure A.6.

[3] In some computing architectures, on the other hand, there is no difference between these two types of data movement, and transfers to I/O devices are executed using the same data movement instructions, a technique called *memory-mapped I/O*.

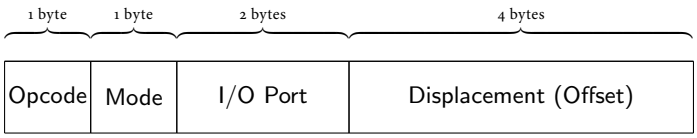



**Figure A.6:** z64 I/O Instruction Format

All instructions of this class have the Class field of the opcode set to 7. The available instructions in this class are reported in Table A.9, along with the values of the Type value, and the operands classes.

**Table A.9:** Class 7 Instructions

| Type | Mnemonic | Operands | O S Z P C | Description |
|---|---|---|---|---|
| 0 | `in` | | - - - - - | Copy data from an I/O to the accumulator |
| 1 | `out` | | - - - - - | Copy data from the accumulator to an I/O port |
| 2 | `ins` | | - - - - - | Copy data from an I/O port to a memory location |
| 3 | `outs` | | - - - - - | Copy data from a memory location to an I/O port |
| 4 | `start` | P | - - - - - | Clear the `STATUS` flip/flop and start the execution of device on I/O port `P` |
| 5 | `clear` | P | - - - - - | Clear the `STATUS` flip/flop of device on I/O port `P` |
| 6 | `jr` | P, M | - - - - - | Jump to `M` if flip/flop `READY` of device on I/O port `P` is set |
| 7 | `jnr` | P, M | - - - - - | Jump to `M` if flip/flop `READY` of device on I/O port `P` is cleared |
| 8 | `wait` | | - - - - - | Block the execution until the `WAIT` signal is cleared |

### `inX` — Input from Port

**Description.** Copies the value from the I/O port specified within the DX register to the accumulator register. The size suffix added to the instruction tells what is the size of the actual data movement, and allows to differentiate between AL, AX, EAX, and RAX. The value stored in DX allows to address I/O ports from 0 to 0xFFFF.

To copy one byte to a specific I/O port, code similar to the following should be used. After its execution. the byte coming from the device can be accessed in AL.

```
movw $ioport, %dx
inb
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement | | | | $Imm_{64}$ |
| inX | 0111 0000 | xx-- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### outX — Output to Port

**Description.** Copies the value from the accumulator register to the I/O port specified within the DX register. The size suffix added to the instruction tells what is the size of the actual data movement, and allows to differentiate between AL, AX, EAX, and RAX. The value stored in DX allows to address I/O ports from 0 to 0xFFFF.

The processor ensures that the data movement to the I/O device connected to the specified I/O port has been completed before executing the next instruction. To write data to a device connected to an I/O port, code similar to the following should be used.

```
movw $ioport, %dx
movb $data, %al
outb
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement(Offset) | | | | $Imm_{64}$ |
| outX | 0111 0001 | xx-- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### insX — Input from Port to String

**Description.** Copies the value from the I/O port specified within the DX register to the memory location starting at the address stored in RDI on a block basis, repeating the data movement RCX times. The size suffix added to the instruction tells what is the size of one block being copied. The value stored in RCX tells how many data blocks of a given size must be copied. The value stored in DX allows to address I/O ports from 0 to 0xFFFF.

To copy a string from a specific I/O port, code similar to the following should be used. After its execution. the byte coming from the device can be accessed in memory starting at the address stored in RDI.

```
movw $ioport, %dx
movq $data, %rdi
movq $size, %rcx
inb
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| outX | 0111 0010 | xx-- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ---- ---- | ✗ |

### outsX — Output from String to Port

**Description.** Copies the value from the memory location starting at the address stored in RDI on a block basis, repeating the data movement RCX times, to the I/O port specified within the DX register. The size suffix added to the instruction tells what is the size of one block being copied. The value stored in RCX tells how many data blocks of a given size must be copied. The value stored in DX allows to address I/O ports from 0 to 0xFFFF.

To copy one string to a specific I/O port, code similar to the following should be used. Memory pointed by RDI must be already initialized to the required value. The processor ensures that the data movement to the I/device connected to the specified I/O port has been completed before executing the next instruction.

```
movw $ioport, %dx
movq $S, %rsi
movq $size, %rcx
outb
```

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| `outX` | `0111 0011` | `xx-- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |

### `start` — Activate a Device

**Description.** Resets the `STATUS` flip/flop in the device connected to the I/O port specified in the operand. This instructions enables the `start` pin on the I/O Control Bus, and can be used by a device interface to communicate to the Device Control Unit that a new operation should be started.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| `start P` | `0111 0100` | `---- ----` | `pppp pppp` | `pppp pppp` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |

### `clear` — Clear Interrupt Cause

**Description.** Resets the `STATUS` flip/flop in the device connected to the I/O port specified in the operand. This instructions enables the `clear` pin on the I/O Control Bus, and should be used by a device interface to clear the cause of an interrupt.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| `clear P` | `0111 0101` | `---- ----` | `pppp pppp` | `pppp pppp` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |

### `jr` — Jump if Device is Ready

**Description.** Jump to the memory location specified in the second operand as a displacement (offset) from the value of RIP after the instruction's fetch phase, if the `STATUS` flip/flop of the device connected to the I/O port specified in the first operand is set.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| `jr P, M` | `0111 0110` | `---- ----` | `pppp pppp` | `pppp pppp` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |

### `jnr` — Jump if Device is not Ready

**Description.** Jump to the memory location specified in the second operand as a displacement (offset) from the value of RIP after the instruction's fetch phase, if the `STATUS` flip/flop of the device connected to the I/O port specified in the first operand is cleared.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| Instruction | Encoding | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| `jr P, M` | `0111 0111` | `---- ----` | `pppp pppp` | `pppp pppp` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | `mmmm mmmm` | ✗ |

### `wait` — Wait while pin `wait` is enabled

**Description.** Do not execute the next instruction until the pin `wait` is disabled. This instruction allows to wait for any busy device to terminate its operation.

**Operation.**

```
MAR ← RIP
MDR ← (MAR); RIP ← RIP + 8
IR ← MDR
```

**Status flags affected.** None.

**Binary Encoding.**

| **Instruction** | **Encoding** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Opcode | Mode | I/O Port | | Displacement (Offset) | | | | $Imm_{64}$ |
| `jr P, M` | `0111 1000` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | `---- ----` | ✗ |

## A.9 Summary of the Opcode Table

In Table A.10 we present the complete instruction set of the z64 processor. The number on each row is the first nibble of the opcode (thus, it is the instruction class), while the number on each column is the second nibble (thus, the type). If an instruction's opcode is, for example, `0x4B`, then its mnemonic and the corresponding class of parameters can be found on row 4, column B.

**Table A.10:** Opcode Table

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | hlt | nop | int S | – | – | – | – | – | – | – | – | – | – | – | – |
| 1 | mov B,E | movsX E,G | movzX E,G | lea E,G | push E | pop E | pushf | popf | movs | stos | – | – | – | – | – | – |
| 2 | add B,E | sub B,E | adc B,E | sbb B,E | cmp B,E | test B,E | neg E | and B,E | or B,E | xor B,E | not E | – | – | – | – | – |
| 3 | sal/shl K,G | sal/shl G | sar K,G | sar G | shr K,G | shr G | rcl K,G | rcl G | rcr K,G | rcr G | rol K,G | rol G | ror K,G | ror G | – | – |
| 4 | clc | clp | clz | cls | cli | cld | clo | stc | stp | stz | sts | sti | std | sto | – | – |
| 5 | jmp M | jmp G | call M | call G | ret | iret | – | – | – | – | – | – | – | – | – | – |
| 6 | jc M | jp M | jz M | js M | jo M | jnc M | jnp M | jnz M | jns M | jno M | – | – | – | – | – | – |
| 7 | in | out | ins | outs | start P | clear P | jr P,M | jnr P,M | wait | – | – | – | – | – | – | – |