MIPS | Assembler programming

Peter Stallinga

peter@stallinga.org

MIPS: Assembler programming (paperback)
Peter Stallinga
v. 1.0 (June 9, 2018)

ISBN: 978-0-244-68801-1

Typefaces: Times Roman. Helvetica, Free Sans
Typesetting: LaTeX2e with TexLive in TexMaker
Graphical: Inkscape and PjotrSoft. All pictures made by the author, except the photo used for the cover (Weber VanHeber) and Pictures 4 (adapted from Wikipedia), 5 (Wikipedia), 10 (output of MARS 4.5 of Sanderson and Vollmar), and 17 (Wikipedia).

stallinga.org
Non-profit science organization

Contents

1 | Introduction

This book is part of a course of Computer Architecture at the first of a university. One part of that discipline is programing the hardware in a low level. While mainly talking about general concepts, it then explains a specific assembler language designed for a specific architecture, namely MIPS, which for didactic purposes is perfect in that it is a RISC-type architecture (reduced instruction set computer) which thus has a limited amount of instructions.

But where does all this fit in? Up front it has to be said that it is assumed here that the reader is comfortable with higher-level programming languages. Specifically, it is assumed that the reader knows the basic concepts of the C programming language:

- Data types
- Variables and constants
- Comment
- Input/output (`printf`, `scanf`)
- If, if-else and switch; conditional execution
- Loops: for, while, do-while
- Arrays and structs
- Functions (and recursivity)
- Passing by value and passing by reference

- Pointers

Especially the last item — pointers — is very important because basically *everything* in assembler is pointers, as we will see! We will also see that nearly all concepts of the list above are not part of assembler. There are no functions. There are no arrays and structures. Variables do not exist. No looping instructions exist. All these concepts we have to implement ourselves, but we will see that the assembler is already prepared to implement these concepts and we will learn how to implement them one by one. We will conclude that the link between C and assembler is quite strong.

Now the main question is, *why* should we want to learn to write programs in assembler, if we already know how to program in a higher-level programming language?

1. Understanding the level below makes us write code in the level above better. For example, if we know that divisions are slower implemented in assembler compared to multiplications, we might want to replace a C instruction `a = b/5.0` by `a = 0.2*b`.

2. In cases when hardware is limited, we are forced to optimize the code to increase *efficiency of using memory space and computing time*. This means going to the low level of assembler.

If you want to jump to the 'goodies', because you came here only to learn how to program in assembler, you can directly jump to Chapter 4. But the thing I want to address now is: where does this all fit in? That is, in our knowledge of the universe and in the way we think in general? And how exactly did we wind up programming in assembler and what will we do with this knowledge? In this chapter some background will be given about computing and how it fits in the layers of knowledge of a university course of Informatics (information processing).

One thing is the information itself, the ideas we are going to process with our hardware. The hard facts — 'numbers' — that we will process to come up with processed information. Maybe the temperature data on the planet processed to come up with a prediction what weather it will be tomorrow. Or maybe an analysis of the stock market to see if we can determine a pattern. This being only monitoring the world, maybe we actually want to use the computer to control the world in things as simple as maintaining the temperature inside a car at a desired value. As we will see in Chapter 2, these 'data' (the 'numbers') only exist in our heads. What exists in reality is the hardware state, described by electronic properties such as voltages, currents, and charges. The link between the two, the state of the hardware and its behavior on the one hand and the interpretation of that state by concepts in our head, is the realm of Computer Architecture. A gate has 5 volt at its output port, which we interpret as 'true' or '1', etc. The hardware seems to follow the logic in our head. In fact, the hardware is *designed* to implement the logic of our head. A well-designed architecture can efficiently and rapidly process the information in the way we imagined it.

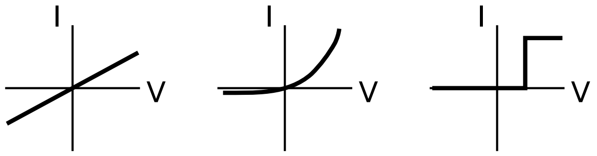
The other thing is the hardware, the physical object that processes our information. The first observation is that it is a so-called finite-state machine, meaning that it can be in one of a limited number of states. This number of states is large, but finite. To give an idea: a computer with 1 GB of memory has $2^{8000000000}$ different possible states. Large, but not infinite. The finiteness limitation is especially felt for smaller memory units. The contents of a memory cell or a register in MIPS is 32 bits and this has only 2^{32} different possible states (about 4 billion). This is especially felt when doing floating-point calculations. Whereas in integer calculations ($\mathbb{Z}$) the limitations of our computer being a finite-state machine are to a certain point rather irrelevant and only limit the range of calculations, for floating-point calculations these limitations are severe and we have to keep them in mind. A single 'float' of 32 bits can take only 2^{32} different values, there where the number of real numbers ($\mathbb{R}$) is infinite, even if we were to limit the range of the numbers (for instance only between 0 and 1).

While not inherently necessary, modern computers are all electronic. This means that the state of the machine is defined in terms of electronic properties. It has not always been like that. Imagine, the first automatic processor, the Difference Engine of Charles Babbage in the beginning of the 19th century, was a fully mechanical machine. However, since the 20th century computers are electronic, first with vacuum tubes and later with transistors and integrated circuits (of transistors).

We can thus place this entire thing in the knowledge tree of science. Layers of knowledge of Informatics:

The starting layer is Physics. A short while after the Big Bang, so the theory goes, particles were created that consisted of quarks that later condensed into electrons, protons and neutrons. Especially the electrons interest us here. They are charged particles and can thus be manipulated by electrical fields. At this layer of knowledge we speak of Particle Physics, which is not very relevant for an Informatics engineer, but also about the Maxwell Equations, which govern the behavior of the charged particles. And then especially the electrons, which interest us here foremost. While, in principle, we can also make computers using the positively-charged particles, the protons, this is less convenient because protons are about three orders of magnitude heavier than the negatively-charged electrons and 'protonics' is thus expected to be significantly slower than 'electronics'. This layer of knowledge is the realm of Electronic engineers (and Physics alike), but of somewhat less interest for the Informatics engineer.

The next layer of knowledge is Electronics. Here we learn concepts of 'current' (which is the movement of charge; how much charge — coulomb — passes a cross-section per second) and 'voltage' (which is the amount of potential energy that is stored in a coulomb of charge). We now start suffering here from the most-irritating error ever made by a scientist, namely attributing a negative charge to the electron instead of a positive one. This is so annoying that we always have to imagine that if we have a current from A to B, we have, in fact, a flow of electrons from B to A. That is, if we still want to have some link to the layer of Physics. Most Electronic engineers prefer to make a level of abstraction and just talk about current as if it is a mere mathematical property, forgetting that currents consists of moving electrons. It is possible to get away with this and such abstraction of ignoring underlying levels of knowledge is quite common, as we will see. Electronic engineers now talk about Ohm's Law ($R = V/I$) and power consumption ($P = V^2/R$) and the likes. Moreover, they talk about capacitance ($C = Q/V$) and inductance ($L = dI/Vdt$). (R is resistance, V is voltage, Q is charge, I is current, C is capacitance, L = inductance, t is time, and $I = dQ/dt$). We can call this level 'linear electronics' since all properties scale linearly: if the voltage increases by a factor 2, the current will also increase by a factor 2, etc. See



Picture 1: Linear electronics, non-linear electronics and digital electronics (I-V curves)

Picture 1.

This brings us immediately to the next level. If linear electronics exist, also non-linear electronics exist. Actually, here is where it starts getting interesting. A so-called diode does not have a linear current-voltage relation, but an exponential behavior instead. The current grows exponentially with the applied voltage (or the voltage grows logarithmically with the applied current; for an electronics engineer it is all the same). This is called Ebers-Moll equation, named after two German scientists,

$$I = I_0 \left[\exp\left(\frac{V}{V_\mathrm{T}} \right) - 1 \right] \tag{1}$$

Even more interesting is a diode, which has a current-voltage relation between A and B controlled by a third connection, C. The resistance (and current) between A en B is thus controlled by a voltage placed at C, and we thus have a trans-resistor, or transistor. An entire world of electronics opened up by the invention of this non-linear behavior. While vacuum tubes — the 'audion' — of De Forest already had this behavior, especially the miniaturization of the transistor made it popular. Signals could be amplified and 'analog' electronics in general surged, for instance radios and televisions. In all these systems the signals are analog in that *any* value between the supply voltages can exist.

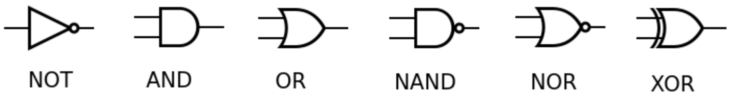
For the Informatics engineers the real fun starts with highly non-linear electronics. By combining transistors in certain ways circuits can be made that are amplifying so much that basically only two (saturation) states exist because the voltages at the output can never exceed the supply voltages (see Picture 1). We can call this binary or 'digital electronics'. We enter the realm of Informatics, because we can assign logical values to these two states and process information in a binary, digital way. Moreover, the digital processing of information begins here, because we can make circuits (logic gates) that have two inputs and one output, the output depends on the logical states of the two inputs. We can imagine here OR-gates, NOR-gates, AND-gates, XOR-gates and NAND-gates. Most university courses offer lectures in digital systems, or digital electronics, that treat such systems from the electronics point of view or from the logical point of view. The latter deals with things such as Karnaugh maps to implement any logic based on simple gates, while the former talks about things like CMOS (complementary [channel-type] metal-oxide-semiconductor transistors) to find the most power-efficient implementation of the desired basic logic functions. We see here that this level of knowledge is mixed. It is where Electronic engineers meet with Informatics engineers and they talk about functionality and electronic implementation of that functionality.

With two input lines and one output line, there are exactly 16 possible logic circuits. Some of them are silly, because they do not depend on any of the inputs (the output always being the logic state 'false' or always the logic state 'true'), or only on one input (copying it, or inverting it), and some of them are redundant (that is, more redundant than others, since any functionality can be implemented by, for instance, only NANDs). That leaves basically the five logic circuits mentioned above: AND, NAND, OR, NOR, XOR. They are shown in bold here below and marked with an *.

The 16 possible 2-input 1-output logic gates:

A	B	FALSE	**A AND B**	A AND NOT B	A	B AND NOT A	B	**A XOR B**	**A OR B**	**A NOR B**	A NXOR B	NOT B	(NOT B) OR A	NOT A	(NOT A) OR B	**A NAND B**	TRUE
0	0	0	**0**	0	0	0	0	**0**	**0**	**1**	1	1	1	1	1	**1**	1
0	1	0	**0**	0	0	1	1	**1**	**1**	**0**	0	0	0	1	1	**1**	1
1	0	0	**0**	1	1	0	0	**1**	**1**	**0**	0	1	1	0	0	**1**	1
1	1	0	**1**	0	1	0	1	**0**	**1**	**0**	1	0	1	0	1	**0**	1
			*					*	*	*						*	

Note that the names of the devices is what we, humans, give them, to somehow make sense out of their behavior. An AND-gate, for example, is named as such because *if* we assume a high voltage is 'true' (written



Picture 2: The one-input-one-output inverter (NOT) gate and the five basic two-input-one-output logic gates (AND, OR, NAND, NOR, XOR)

down as '1' here) and applied at both entrances then, and only then, the output voltage is high, meaning 'true' or '1'. We'll get back to this distinction about physical level and logic abstraction later in the coming chapters.

Combining more transistors allows for the implementation of more advanced functionality such as flip-flops (memories), latches, clocks, etc. And also advanced processing circuits such as ALUs (arithmetic logic units) and even CPUs (central processing units). I would like to refer the reader here to the book of Tanenbaum, *Structured Computer Organization*, where the architecture of computers is very well described.

With increased complexity of the functionality more and more transistors are needed. Miniaturization of the transistors made it possible to pack ever more transistors per square centimeter; we all know the famous Moore's Law — named after Gordon Moore — that predicts that the number of transistors per area doubles every 2 years, a speed of innovation that is still going on in 2018. We have reached a level of integration that is some tens of billions of transistors on a single 'chip'. Some very powerful circuits can be built with so many transistors and it no longer makes sense of talking about individual gates, let alone transistors. We must make another level of abstraction if we want to keep on understanding what is going on in our hardware. Here is where our knowledge layer of Computer Architecture starts kicking in. A central processor has several input lines and output lines. It can be imagined (sic) as a logic array where we have two sets of input data elements, and one set of lines that define the functionality selected. We can recognize here information as data and a 'program', which consists of supplying a combination of functionalities to 'execute' on the data.

In the first approach, the program and data were supplied to the processor in the form of logic states at the entrance of the processor, for instance mechanical switches supplying voltages 'high' and 'low'. We can represent such programs symbolically by 0s and 1s. The work of an engineer was to translate desired functionality into a set of 0s and 1s to be supplied to the machine. We call such programs therefore 'machine language'. A program for calculating the product of two numbers might be

```
1001101011100101
0111100101101101
1110000010110100
```

Of course, this is hardly legible to the engineer and mistakes must have been quite frequent. One engineer must have coined the idea that doing such repetitive work — it often consisted of doing the same translations of a human-readable logic program to a machine readable machine language — might actually be done by the machine itself. (Take that for a machine making machines). While at first the idea was considered ludicrous — "Why having the machine do something that can perfectly be done by a human being?!" — the paradigm of computing must have shifted from doing as little as possible by the computer to doing as much as possible by the computer. In the 21st century we say, "Why have a human doing work that can perfectly be done by a computer?!" The idea of a 'compiler' or 'translator' was born. The machine running a program (written in machine language, of course) was fed human-readable 'code' that was translated into machine-readable machine-language data and then run (or only stored somewhere, for later use).

The first versions of these meta-languages were still rather close to the machine language and only mnemonics were used for the 'instructions' (functionalities selected). So, the machine code for adding two registers '01110111' was written as `add`. This type of programming is called second-generation programming languages, or (macro) 'assembler', exactly the layer of knowledge of this book. It is a level of programming quite close to the hardware level. (There can also be a level of programming inside the ALU and the control logic, which is called micro-assembler, which will not be covered by this book).

Of course, the hierarchy of Informatics does not stop here, but at this point it is nice to take a look back at where we have arrived. Basically, by writing the code 'add', etc., we control the flow of electrons in our processor. Of course, in no way is it necessary for an Informatics engineer to know that electrons are flowing in the processor. The only thing an engineer needs to know is the *functionality* of the machine and not how it is implemented! An Electronics engineer needs to know about what is going to be done with the electronics, as well as knowing how to implement it in the Physics layer. A Physics graduate is basically just doing philosophy and could not care less what is being done with the knowledge acquired. No scientist — that is, J. J. Thomson — ever thought, "Let me discover the electron, so that we can add two numbers fast."

The engineers soon must have discovered that very often the same functionality was implemented. It was always things like for-loops, or if-then-else structures. People must have started writing programs in meta-assembler, nearly English. Programs were designed by the well known fluxograms, then written down in English, as in something like

```
for (i=0; i<10; i++){
      if (i % 2 == 0)
           printf("even")
      else
           printf("odd")
}
```

which would then be translated by an engineer into assembler and fed to the computer. Well, they must have thought, if assembler can be translated into machine-language by the machine, why not let the machine translate the near-English-source code directly?! This created the so-called third-generation programming languages, of which FORTRAN (Formula Translator), BASIC (Beginners All-Purpose Symbolic Instruction Code), Pascal (named after French Mathematician Blaise Pascal), and C are the most famous. This is normally the 'starting level' of the engineer and scientist alike. It is well possible to never look back at the levels below, but learning assembler is useful for the reasons given earlier.

After having learned high-level 'imperative' programming, students then normally go on and learn to write in object-oriented languages such as C++, Delphi, or Java. Obviously, this now falls way outside the realm of Computer Architecture and the subject of this book. Even more distanced are the subjects of applications such as writing in 'frameworks', where several different programming languages can be joined. As an example may serve Android Studio, that combines writing in Java, HTML (XML) and Javascript all in one IDE (integrated development environment). But don't forget that when you are connecting to Facebook on your mobile telephone, that it is all based on object-oriented programming, that is based on third-level programming languages, that is based on assembler, that is based on machine language, that is based on integrated logic circuits, that is based on non-linear electronics, that is based on electronics, that is based on physics. At the end, it is all because of the Big Bang.

2 | Number systems

We all use number systems in daily life. The most famous in modern world is the one based on the number 10, the so-called decimal system. The first myth is that this is a very adequate number, because it is nice 'round'. However, note that *any* base number, when expressed in that base is written as 10. For example, in the binary system (base 2), 2 is written as . . . 10.

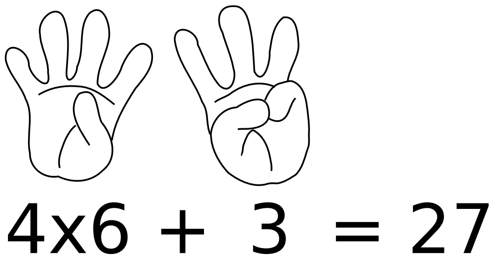
The second myth is that 10 is good for a base system because "we have ten fingers". Well, if the number of fingers were to determine what number system to use, it would be 6 or 11. To show why this is: Imagine each hand shows a digit, for instance your left hand shows the units and your right hand the multitudes of 6. You can start counting: decimal=RL (right hand, left hand):

1=01, 2=02, 3=03, 4=04, 5=05, 6=10, 7=11, 8=12, 9=13, 10=14, 11=15, 12=20, 13=21 . . .

and so forth. This is very convenient. A base-6 hexal number system works very well when communicating with your hands, see Picture 3 for an example of how to represent 27 (base 10) with our hands (base 6). So: *five* fingers per hand implies that base-*six* is ideal. Probably for this reason the base-6 number system survived for a long time, with England being the most famous case, since most people in the world have two hands with five fingers each.

The disadvantage of the hexal system is that numbers get large faster. To give an example, nearly 1 million (999 999) in the decimal system has 6 digits while in the hexal system it has 8 (33 233 343).

Other number systems that were popular were base-20. This unit



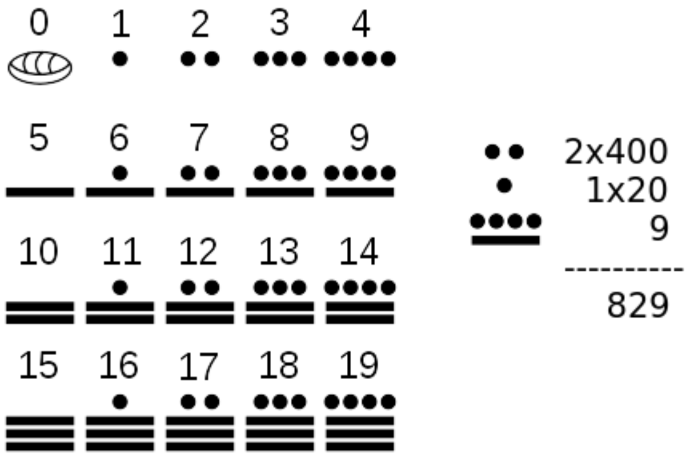
Picture 3: An example of the hexal (base 6) number system with counting on our hand. One hand is used for the units and the other hand for the multiples of the base number 6

is called a 'score' in English. Take for example this funny Limerick:

A Dozen, A Gross, And A Score,
Plus Three Times The Square Root Of Four,
Divided By Seven,
Plus Five Times Eleven,
Equals Nine Squared Plus Zero, No More.

(A dozen is 12, a gross is a dozen dozens, $12 \times 12 = 144$). Some languages still remind us of this score-based system. French speak of 'quatre vingts' for eighty, giving a calculation in the number $80 = 4 \times 20$. Likewise, the Danes talk of 'tres' and 'firs' to indicate sixty ($= 3 \times 20$) and eighty ($= 4 \times 20$), respectively. Confusingly, other Danish numbers are based on the decimal system: 'tyve' (20), 'tredive' (30), 'fyrre' (40). And 'halvtreds' does not mean half 'tres' (60/2=30), but halfway 40 and 60, thus 50. Are you still following it? Well, the Danes seem capable of seeing the logic.

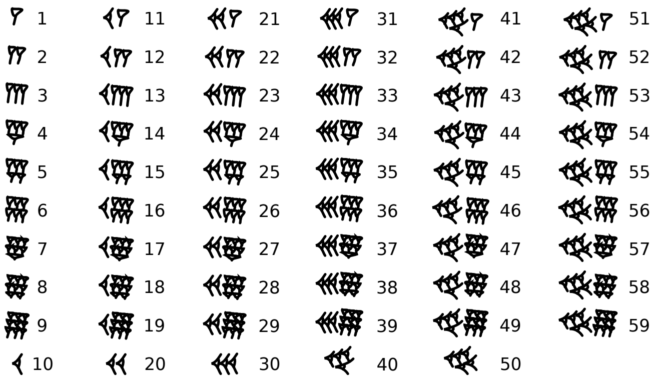
Interestingly, also the Mayas used this system of scores, see Picture 4. A unit was represented by a dot and five dots was written as a horizontal bar. Subsequent units, multiples of the base number 20, were stacked on top. In this way, the number 829 ($= 2 \times 20^2 + 1 \times 20^1 + 9 \times 20^0$), for example, would be written as two dots above one dot above four dots and a bar. Note that they also had a symbol for zero, which was needed by their stacking method. How else could one distinguish between 801 (two dots on top of a dot) and 41 (two dots on top of a dot)?



Picture 4: The number system of the Mayas based on scores (20). Their unit was represented by a dot, with five of them written as a horizontal bar. The numbers 0 to 19 are shown here. Subsequent digits were stacked vertically on top of each other. An example is the number 829 ($= 2 \times 20^2 + 1 \times 20^1 + 9 \times 20^0$) would be written as two dots above one dot above four dots and a bar. Curiously, they also had a symbol for the number zero, probably because the stacking method requires it.

Now, for all you conspiracy thinkers — those that believe in aliens — the question arises how two distinctly separated civilizations (the Mayas and the Danes), that had no contact with each other (since they were separated by the Atlantic Ocean), both seemingly independently decided on 20 for their number system.

The best system was probably invented by the Babylonians. They used a combination of base 10 and base 6, and this makes it divisible by 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, and 30. That is especially useful when doing divisions with results after the floating point. Compare for example the difficulty of our decimal system to write out 1/3. It gives a result with an infinite number of digits, 0.3333... ($= 3/10 + 3/10^2 + 3/10^3 \ldots$). While for the Babylonians, 1/3 is simply 20/60, or a single digit '20' after the floating point.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59

Picture 5: The Babylonian number system based on 10 and 6. A zero was represented by a space, some place without any symbol. (Source: Wikipedia)

Note that our geometry and timing is still done in the Babylonian number system. Especially here you can see the power of this number system. A full 360° circle can be divided into 2 angles of 180°, 3 angles of 120°, 4 angles of 90°, 5 angles of 72°, 6 angles of 60°, 8 angles of 45°, 9 angles of 40°, 10 angles of 36°, 12 angles of 30°, 15 angles of 24°, 18 angles of 20°, 20 angles of 18°, 24 angles of 15°, 36 angles of 10°, 40 angles of 9°, 45 angles of 8°, 60 angles of 6°, 72 angles of 5°, 90 angles of 4°, 120 angles of 3°, 180 angles of 2°, and 360 angles of 1°.

Maybe the worst number system was invented by the Romans. (Makes you wonder how they could be so crafty and build bridges and roads everywhere). The problem with their system is that it does not have a unique base number, nor does it follow the method of ordering the significance of the digits, as for instance is done by the Mayas (highest significant digit on top) or by modern counting (most-significant digit on the left). The Romans had mixed significance, as we all know. Placing a small unit before a large unit puts a negative weight on it. Where 'I' indicates 1 and 'C' stands for 100 (clearly bigger), 99 is represented by placing the I before the C. Moreover, the concept of digits (units multiplied with a base raised to some power) does not even exist. It creates a system where the same number can be represented in more than one way. As an example, 99 can also be written as XCIX, which is $-10 + 100 - 1 + 10$.

I: 1
V: 5
X: 10
L: 50
C: 100
D: 500
M: 1000

Examples:

2018 = MMXVIII = 1000 + 1000 + 10 + 5 + 1 + 1 + 1
2019 = MMXIX = 1000 + 1000 − 1 + 10 + 10

This Roman system we had better forget as soon as possible! Instead, more useful, are standard sign-magnitude number systems

- Digits have weight. The most-significant digit is on the utmost left, the least significant digit on the utmost right
- Each step to the left has a weight that is increased by the base number and each step to the right the weight is reduced by the base number
- Negative numbers are preceded by a '$-$' sign. Positive numbers can (but don't have to) be preceded by a '$+$' sign. Alternatively, numbers can be limited to positive, unsigned' values only

As an example, if the number base is x, then the number represented by $\pm abcde_{\mathrm{x}}$ is $\pm(a \times x^4 + b \times x^3 + c \times x^2 + d \times x^1 + e \times x^0)$. For example, $12345_6 = 1 \times 6^4 + 2 \times 6^3 + 3 \times 6^2 + 4 \times 6^1 + 5 \times 6^0 = 1865_{10} = 0 \times 10^4 + 1 \times 10^3 + 8 \times 10^2 + 6 \times 10^1 + 5 \times 10^0$, where the convention was used to write the base as a subscript.

Note that floating point numbers follow this scheme. After the floating point (floating comma in some countries), the digits get divided by the base: $\pm abc.de_{\mathrm{x}}$ is $\pm(a \times x^2 + b \times x^1 + c \times x^0 + d \times x^{-1} + e \times x^{-2})$. Conversion between base systems can become impossible in floating point numbers. Some numbers, like 0.3_6, are still possible to convert to base 10, namely $3 \times 6^{-1} = 5 \times 10^{-1} = 0.5_{10}$. But what about 0.1_6? In base-10 it is an infinite string: $0.166666\ldots_{10}$. The reason why Babylonians used the base-60 number system; it is more likely a division can be written out with a finite number of digits.

Binary numbers

”There are 10 types of people:
Those that know binary and those that don’t!”

The most important of all number systems for informatics is base-2. That is because the underlying hardware works with binary state electronics. Any output of any gate can be either low or high. Any capacitor is either full or empty. It does not matter at this moment which one we will ascribe to the logic (sic) ’1’ and which one a logic ’0’. Now, if we have a set of transistors each in a certain state designated by ’1’ or ’0’ we can imagine (sic) that they represent a binary number. Note that the number does not exist in the computer!

Now, this needs some explanation. How can it be that numbers do not exist in the computer? Well, it can be said even stronger: numbers do not exist in the world! They only exist in our heads. They are part of mathematical — that is, *imaginary* — worlds. The only thing that exists in reality are things that are in the realm of Physics, and all these things can be expressed in terms of the seven basic S.I. units (kilogram, meter, second, ampere, mol, candela, kelvin) or their derivatives. Do not confuse a sole number with a quantity which has unit ’mol’. If there are two people in the room, in fact there are $2.0/N_\mathrm{A}$ mol people in the room, with N_A the number of Avogadro, $N_\mathrm{A} = 6.022\times 10^{23}/\mathrm{mol}$.

Likewise, if we have a number (binary or hexadecimal, or whatever) in a computer, what we in fact have is merely a set of gate states (as in high voltage or low voltage) that we can — in our heads — represent with a number. To show why this makes sense: the same combination of gate states, for instance a 32-bit register, can simultaneously be thought of as a binary number, a decimal integer, an ASCII character, or a single-precision floating point number.

Why it is useful to do this abstraction is that, *if* we assume these to represent numbers, the logic of the hardware (ALU, arithmetic logic unit) follows exactly the logic we have in our heads of how it should behave if they are numbers. The behavior of our computer is consistent with our model in our heads. We can thus think as a computer, or a computer ’thinks’ the same way we do.

Returning to the subject, binary numbers are very useful in informatics because the gates work with two possible states. Note that not necessarily computers have to use this architecture. Russians are famous for having developed the so-called ternary computers that use ternary logic (three possible values) and trits instead of the more common binary logic (two possible values) and bits in their calculations. Apparently, it had some advantages, such as lower power consumption and lower production cost (Wikipedia). Obviously, for such computers, it makes much more sense to represent the information as ternary numbers. However, this technology has died out and we will no longer refer to it here.

What we have to remember, however, at this point is that there is a symbolic link between a physical set of gate states, the binary representation, and the non-binary interpretation of what information is, in fact, stored there. For example, for a 4-bit gate output we have the following approaches:

Physical levels:

5 volt, 0 volt, 5 volt, 5 volt

Boolean logic levels (assuming false is low voltage, true is high voltage):

true, false, true, true

Binary logic levels (assuming '0' is low voltage, '1' is high voltage):

'1', '0', '1', '1'

Binary value (assuming 0 is '0' and 1 is '1'):

1011

Decimal value (assuming binary value is unsigned int with MSB left):

11

Hexadecimal value (idem):

B

With only the first (physical) level really existing and the other just figments of our thought in our heads. Note that it is here assumed that high voltage = true = '1' = 1. In any step this assumption can be different, as in, for instance, high voltage = 'false'. As long as the behavior of the hardware is consistent with the symbolic translation, it is correct. An AND-gate should have a physical behavior that is consistent with the truth table of the logic-AND function.

An important observation to make here: A combination of n binary gates — or n 'bits' — can take $p = 2^n$ different possible 'values' or output combinations. Reasoning the other way around: we need at least $n = \log_2(p)$ bits (gates) to represent a number that has p possible values. So, for instance, with 3 bits we can 'store' integer numbers from 0 to 7. (Or from 27 to 34, if we'd want that). Reasoning the other way around, to store the 26 letters of the English alphabet, we need at least $\log_2(26) = 4.7$ bits. That is, 5 bits, since partial bits do not exist.

An 8 bit register or memory address can thus store $2^8 = 256$ different values. If they are unsigned integers including zero, they'd span from 0 to 255. If we represent them in binary, the rightmost bit, the least-significant bit (LSB) has weight $2^0 = 1$ and the leftmost bit, the most-significant bit (MSB) has weight $2^7 = 128$. Likewise, 32-bit registers store integer numbers ranging from 0 to 4,294,967,295. As an example for a 4-bit unsigned integer:

$$\begin{aligned} 1101 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 4 + 1 = 13 \end{aligned}$$

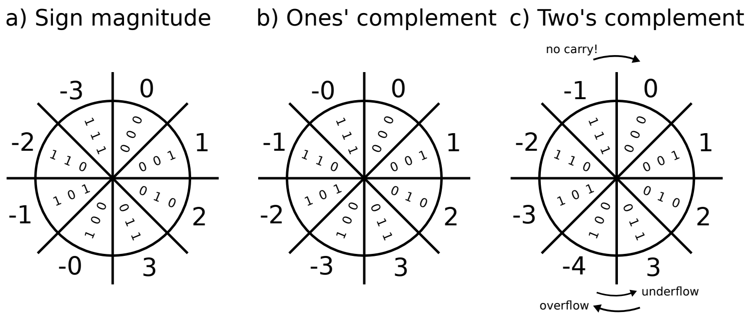
Negative numbers

So far so good. The problems start when we want to interpret the bit patterns to include negative numbers as well and have the hardware capable of dealing with them, that is, having the ALU perform gate operations that are consistent with the formalism of the gate voltages representing numbers that can be positive as well as negative.

As a first thought, we may think that using — 'sacrificing' — one bit for the sign, and continuing to use rest for the magnitude (which now has a smaller range, 0 unto $2^{n-1} - 1$), is a good idea. This scheme is called sign-magnitude. See picture 6a for a three-bit example. Assuming the above binary combination is a sign-magnitude representation of an integer number, with the MSB representing the sign (0='+', 1='−'), it would give:

$$\begin{aligned} 1101 &= -(1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\ &= -(4 + 1) = -5 \end{aligned}$$

Note the peculiar property of sign magnitude that there are two numbers zero, namely +0 (0000) and −0 (1000). More problematic is that we can no longer use the same hardware for these sign-magnitude



Picture 6: Three ways of representing signed numbers; 3-bit examples. a) In sign-magnitude, the MSB is the sign bit, 0 for '+' and 1 for '−', while the rest of the bits form a normal unsigned n−1-bit number. b) In ones' complement, to find a negative number, simply all bits are inverted. c) In two's complement, the MSB has a negative weight -2^{n-1} while the other bits have positive weight $+2^{n-2}$ to $+2^0$

numbers and the unsigned integer numbers. As an example, imaging adding 1 to −2 using hardware we learned from digital electronics classes:

binary		*unsigned*		*sign-magnitude*
0001	=	1	=	+1
1010	=	10	=	−2
1011	=	11	=	−3

The same problem we have in the alternative ones' complement. In this scheme, we just invert all the bits to get the negative number, see Picture 6b for a 3-bit example. Like in sign-magnitude, positive numbers start with a 0 and negative numbers start with a 1. Moreover, also here we have two possible ways to represent zero, namely $+0 =$ 0000 (all 0s) and $-0 =$ 1111 (all 1s). While the calculations at first sight seem to be going better, we have this peculiar result that occurs whenever there is a 'carry':

binary		*unsigned*		*ones'-complement*
0011	=	3	=	+3
1110	=	14	=	−1
0001	=	1	=	+1

This can be solved by adding the carry that was ignored. Adding it will make the final result +2, which is correct. However, we would like to use the same hardware for unsigned integer operations and signed integer operations and not have to resort to additional operations when adding signed integers. The perfect solution for that is the two's-complement representation of numbers.

Two's-complement is formed by giving a negative weight -2^{n-1} to the MSB, and positive weights $+2^{n-2}$ to $+2^0$ to the other bits until the LSB. See Picture 6c for a 3-bit example. The number −1 is thus formed by a combination of all 1s. And this gives another way of rapidly looking at things:

- If the first bit (MSB) is 0, treat the number as a normal unsigned int, with every (other) bit its proper weight. Example, 00000100: The positional weight of the only 1 is 4, therefore the number is +4.

- If the first bit (MSB) is 1, the number is equal to −1 plus every 0 (sic) weighted by its positional weight negatively. Example, 11111011: The positional weight of the only 0 is 4, therefore the number is $-1 - 4 = -5$.

This might come in handy sometimes when we want rapid answers. Especially for large bit numbers. (A 32-bit two's-complement int,

11111111111111111111111111111011

is also −5)

Note that two's complement has only one version of zero. It moreover follows the same hardware logic as unsigned integers. Interestingly, the carry can be ignored (see Picture 6c). On the other hand, a phenomenon occurs halfway the bit pattern, when the MSB changes $0 \to 1$,

binary		*unsigned*		*two's-complement*
0111	=	7	=	+7
0001	=	1	=	+1
1000	=	8	=	−8

this effect is called 'overflow', and similarly, 'underflow' occurs when subtracting numbers (or adding a negative number) resulting in a change of MSB bit $1 \to 0$.

To find the 2's-complement of a number we can use either of the two following algorithms,

- Invert all bits and add 1. Example:

3 =	00000011	
invert:	11111100	
add 1:	11111101	=−3

- Starting from the right (LSB) simply copy until the first 1 encountered. From then on, but excluding this one, invert all bits.

For a programmer it is not important to exactly know *how* this is implemented in hardware. The only thing that matters for us is that signed integer numbers in MIPS are represented in two's-complement. We can consider the hardware itself as a black box. If the reader is interested, I recommend the book of Tanenbaum, *Computer Architecture.*

Hexadecimal

The hexadecimal number system is very often used in informatics, it is ubiquitous, for a very simple reason: it is simply joining 4 binary bits and attributing a symbol to it. This for a very simple and unique reason: to save space. Hexadecimal is simply shorthand binary. So, we have the simple look-up table as in Table I.

In this table, the A does not represent the letter A of the English alphabet, but rather the bit combination 1010 written in hexadecimal. (As we will see in a moment, the letter 'A' in the English alphabet is coded in ASCII in a different way).

Table I: Look-up table for hexadecimal

binary	*decimal*	*hexadecimal*	*BCD*
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	8
1001	9	9	9
1010	10	A	-
1011	11	B	-
1100	12	C	-
1101	13	D	-
1110	14	E	-
1111	15	F	-

The table also shows binary-coded decimal (BCD), which is similar to hexadecimal but the last combinations are not used. However, a computer calculating in BCD is not the same as a computer calculating in binary (and thus hexadecimal). As an example, in BCD, 07+07 = 14: or 0000 0111 + 0000 0111 = 0001 0100, while in binary (hex) it is 00000111 + 00000111 = 00001110, or 0x07 + 0x07 = 0x0E.

Here we used the convention of preceding the hexadecimal number by 'zero-x' (0x) to distinguish the hexadecimal numbers from other number systems. As an example, 0x12 is $1 \times 16^1 + 2 \times 16^0 = 18$ in decimal representation.

ASCII

Another way of representing data is by ASCII (American standard code for information interchange). Or, to say it the other way around:

instead of storing in 8 bits a binary number, signed or unsigned, we can also store there a letter of text. Simply by convention (!) we can attribute binary patterns to letters of the English alphabet, for instance, 01000001 is equal to the letter 'A' and 01100001 to the letter 'a'. And it now becomes very clear why we could make the statement that the numbers or the letters do not exist in the computer, but only in our heads. Because, how could it otherwise be that the exact same physical state of the computer, with 01000001 in a memory address — describing the combination of output voltages or charge states — contains the short unsigned byte 65 as well as the letter 'A'? This is only possible if the physical states really exist and the *interpretation* of these states — the numbers or letters — is only in our heads. The computer follows our thoughts and when we process 7+7, the bit pattern for 14 comes out:

```
  00000111 = 7
  00000111 = 7
  ======== +
  00001110 = 14
```

That is to say, if we use the same hardware to process the ASCII characters '7' + '7' we get (see the ASCII table in Appendix C):

```
  00110111 = '7'
  00110111 = '7'
  ======== +
  01101110 = 'n'
```

which, in decimal, 55 + 55 = 110, which is the letter 'n' in ASCII. So, 7+7 = 14, while '7'+'7' = 'n'. The computer hardware does not care what is in our heads and if it makes sense there! As the popular saying goes, word processors are often WYSIWYG, "what you see is what you get". Computer hardware is YWIYGI, "you wanted it, you got it!". You asked for doing an `add` instruction with ASCII data and that's what you get.

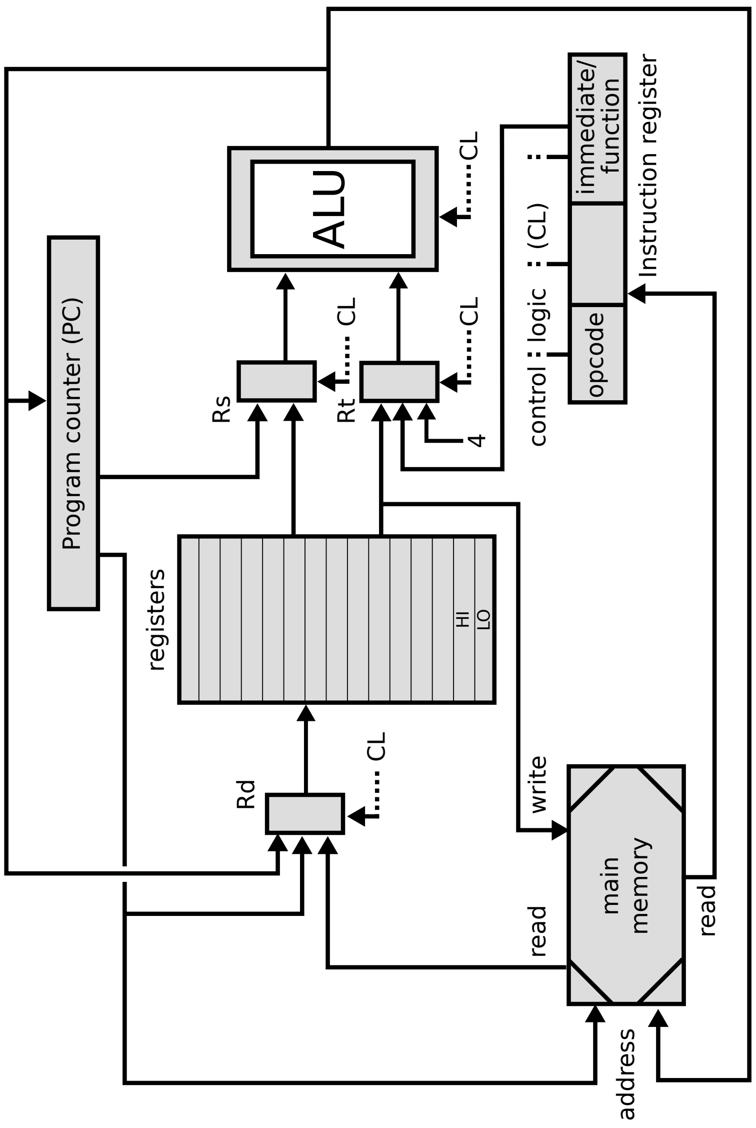
3 Architecture of MIPS

We now come to the specific architecture of a MIPS computer. In this chapter we will not bother ourselves with understanding how exactly the described behavior is implemented in hardware. We just have to know at this stage how the machine works logically. For that we have to analyze the data flow diagram. Picture 7 shows this schematically. It all evolves around the ALU (arithmetic logic unit), which is doing the basic calculation. Or better to say 'logic operations', it merely being a sphisticated logic array. In a nutshell, this is what happens at each step of a program:

- The program counter (PC) contains the address of the next instruction to be executed. The program counter is a 32-bit register and can thus address 2^{32} different addresses. The main memory is organized in byte units, and thus MIPS can address 4 GB of main memory.

- The 32-bit contents of the 4 consecutive bytes of memory pointed to by the PC are fetched and placed in the instruction register (IR).

- The opcode (and possibly the function code) of the instruction in the instruction register determine *what* is going to be performed. These are the first 6 bits of the instruction and control the hardware (CL standing for 'control logic'). It 'steers' the ALU into doing the correct logic operation. For instance, with opcode 000000 the logic operation will be mathematically add



Picture 7: Basic data flow diagram of the MIPS architecture

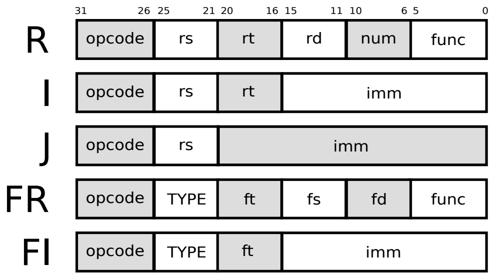
the two operands, Rs ('source register') added to Rt ('target register') at the input of the ALU.

- The rest of the instruction code then determines *which* operands will be used in the logic operation. This is also controlled by the control logic. For instance Rs will be register `$t0` and Rt will be register `$t1`.
- One of the operands can also be 'immediate', which means that the value is part of the instruction and should be copied from the last bits of the instruction register.
- The control logic implied in the instruction also specifies where the resulting value, if any is generated, should be stored. This is the destination register Rd.
- An operation can also be a simple advancing of the program, making the program counter point to the next instruction, at PC+4. If the program counter was not changed by a jump instruction, this instruction is executed and a new cycle starts
- Mostly the operands are registers of the register file. These consist of 32 registers of 32 bits each and some additional functional registers described in a moment.
- For some instructions the output is written to main memory instead.

See Picture 8 for the format of the instruction register.

That is basically it. This is what we have to work with. With this we have to do all our calculations. Writing programs in MIPS consists of shoving information around, adding or subtracting things, doing simple logical operations on data, etc. It is like shunting (or switching) rolling stock to form trains. Noteworthy, MIPS has a limited instruction set. It is of the RISC architecture (reduced instruction set computer), meaning that it has few instructions, but every instruction is fast. This in contrast to CISC architectures (complex instruction set computers), which has many instructions, but each relatively slow.

Simple as it may seem, when we start programming we soon realize the thing is very powerful indeed. In the next chapter it will be



Picture 8: Possible formats of the instruction register. Three different types, R, J and I. All instructions start with a 6-bit opcode, R (register) instructions specify a source (rs), a target (rt) and a destination (rd) register. Some further specify a number (num), for instance the amount of bits to shift, or specify a subfunction (func). I (immediate) instructions have one of operands included in the instruction. J (jump) instructions can specify a relative address in immediate form in the instruction or use an address in a register (rs). The FR and FI instructions are for floating point operations, to be discussed later

explained how with the simple 6-bit instruction set ($2^6 = 64$ different instructions) we can implement advanced programs. We will see how we can implement the concepts of higher-level programming languages such as C and FORTRAN in MIPS instructions.

Before we do that, next to be described here in this chapter is the register file. As said above, it consists of 32 registers of 32 bit. While they can be labeled `$0` until `$31`, mostly they are written in their logical names, which indicate their function (see Table II):

- The first register is called `$zero`. The contents of this register are hardwired to always contain 0, that is, 32 bits of 0. This might not make sense at first sight, but don't forget that MIPS is a RISC architecture, and the engineers had to save on instructions. As such, MIPS does not have, for example, an instruction for copying the contents of one register to another. The way it is implemented is by adding zero to a register and storing the result of the 'calculation' in the destination register. In other words,

```
move Rd Rs
```

Table II: MIPS registers

0	`$zero`	8	`$t0`	16	`$s0`	24	`$t8`
1	`$at`	9	`$t1`	17	`$s1`	25	`$t9`
2	`$v0`	10	`$t2`	18	`$s2`	26	`$k0`
3	`$v1`	11	`$t3`	19	`$s3`	27	`$k1`
4	`$a0`	12	`$t4`	20	`$s4`	28	`$gp`
5	`$a1`	13	`$t5`	21	`$s5`	29	`$sp`
6	`$a2`	14	`$t6`	22	`$s6`	30	`$fp`
7	`$a3`	15	`$t7`	23	`$s7`	31	`$ra`

is implemented with

```
add Rd Rt $zero
```

Fortunately, MIPS compilers understand this and we can freely use the `move` instruction. We have to realize that this instruction is not implemented in MIPS, but translated by the compiler and is thus a so-called pseudo instruction.

- Register 1 is called `$at` and is reserved for the compiler. It is used, for instance, to load a 32 bit address in two steps into a register, storing the intermediate value in `$at`.

- Registers 2 and 3 (`$v0` and `$v1`) are used by functions to return values (similar to the C-instruction `return(value)`).

- Similarly, registers 4 through 7 (`$a0` through `$a3`) are used to pass information to functions as arguments, either directly as integer values (passing by value) or as addresses to values in memory (passing by reference).

- The next registers, 8 through 25 are 'freely usable' registers, which are divided into two groups, the so-called t-registers (8-15 and 24-25, `$t0`-`$t7` and `$t8`-`$t9`) and s-registers (16-23, `$s0`-`$s7`). The difference between the two is: who is going to be responsible for temporarily saving the values on the stack when functions are called, the calling code — the 'caller' — or the called code — the 'callee'. This will be better explained in the section about functions in MIPS (Page 74 of Section 4.7 of Chapter 4).

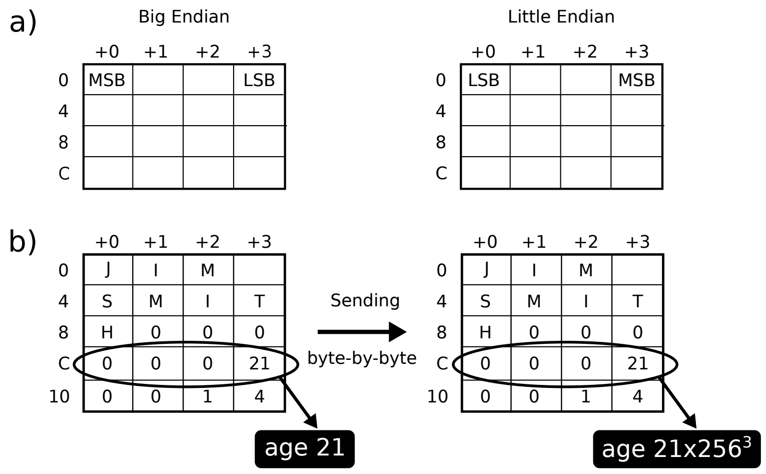
- The last registers are for the kernel (26 and 27; `$k0` and `$k1`), the global pointer (28; `$gp`), stack pointer (29; `$sp`), frame pointer (30; `$fp`) and return address (31; `$ra`). The stack pointer and return address are used when implementing functions and will be discussed in Section 4.7 of Chapter 4. The kernel registers, global pointer and frame pointer will not be discussed in this book.

- Apart from these general 32-bit registers are two registers exclusively used for arithmetic, namely `HI` and `LO` that are each 32 bit and store the results of integer multiplications and divisions.

Finally, there remains to be described the organization of memory in MIPS architectures. A problem is introduced by the fact that the CPU is organized as 32 bits (instruction register, program counter and all other registers), but the main memory is 8-bit. That means that two addresses are 8-bit apart. Because the communication bus of the CPU with the memory is also 32 bits wide, a problem arises in that we do not know how the information is stored into memory. Of the 4 bytes to store in a memory write, where does the most-significant byte (MSB) and least-significant byte (LSB) go?

There exist two possibilities, as shown in Picture 9a that represents part of memory here shown semi-linearly. A write or read from memory will communicate a complete line of this memory, for instance from address 0 to address 3, each containing 8 bits. In so-called big endian, the MSB goes to the lowest address, while in little endian this memory contains the LSB. This naming comes from the country of Jonathan Swift's book *Gulliver's Travels.* In that country the biggest debate of politicians was how one should open an egg at breakfast in the morning; opening it at the big end or at the little end. Of course, it is a completely irrelevant debate, but both parties, the Big Endians and the Little Endians alike, took it very serious.

Also for us it is completely irrelevant. As long as it is done consistently, it makes no difference whatsoever if the architecture is big endian or little endian — in fact, a software engineer does not even have to worry about it; it is the job of the hardware engineer — but



Picture 9: a) The difference between Big Endian and Little Endian storage of information is caused by the fact of having a different size CPU and communication bus architecture compared to that of main memory. In this case, MIPS has a 32-bit architecture while the addressing distance of memory is 8-bit. In Big Endian the MSB is stored at the smallest memory address, while in Little Endian this memory element is occupied by the LSB. b) It goes wrong when two computers of different endianness communicate byte-by-byte; the LSB becomes the MSB, vice versa

a problem might arise when we communicate between computers of different architectures. If we fetch (32-bit) information from memory and start communicating it byte-by-byte to another computer of different endianness that fills its memory with it, things might go wrong. Picture 9b shows an example. The big-endian computer on the left has stored the information of `JIM SMITH, age 21, office 260` into memory (note the padding of the name by 0s; this in order to align the data to addresses being necessarily a multiple of 4) and starts communicating it byte-by-byte to the little-endian computer on the right. It sends the memory in a string, first address 0, then 1, etc. Thus: first a J, then an I, etc. The little-endian computer receives a J and stores it in address 1, etc. So far so good. At the end it has completely copied the file of Jim Smith. Now the right computer wants to print the age of Jim. The poor guy has suddenly aged quite a lot. The LSB (21) has become the MSB, and Jim is now $21 \times 256^3 = 352,321,536$ years old, and the building in which he works must be rather big, his office being $4 \times 256^3 + 1 \times 256^2 = 67,174,400$. We have to keep this in mind when communicating with other computers.

4 | MIPS assembler language implementation

It is now finally time to start coding! In this chapter we will do that. For that we have to define some key issues. We have to understand what, in essence, is happening in our architecture hardware when we run a program.

The hardware loads the contents of the 4 consecutive memory addressed, pointed to by the program counter (`pc`), into the 32-bit instruction register and executes it. Then it loads the next instruction that is at `pc+4` if the previous instruction did not cause a jump in the program.

Now about the instructions themselves. MIPS is an example of a RISC (reduced instructions set computer) architecture. This means that the number of different instructions is rather limited. Each one is fast, but not very powerful. MIPS has a total of only 64 different basic instructions (see Appendix A). That means that specifying which one to use takes 6 bits ($2^6 = 64$. Note that some instructions use additional bits to define a sub-function, thereby leaving less space for the rest of the instruction). As an example, `addi` is specified by `001000`. This binary number that is in the beginning of the instruction we call the 'opcode', whereas the more human-readable `addi` is called the 'mnemonic'. Our MARS compiler translates our mnemonics into opcodes so that our life is a little easier. Imagine, in the old days engineers did not have compilers and they had to write the opcodes by hand instead of the mnemonics. You should appreciate the work done by developers of compilers such as MARS.

The next part of the instruction is specifying the operands, of which there can be either one, two or three. Imagine we want to add the contents of register `$t0` to `$t1` and store it in `$t2`. These registers are then the operands. The nomenclature is to call them the source, the target and the destination operand, as in: adding the source to the target and store it at the destination. The complete instruction consists of the choice of operation (mnemonic) plus operands. Since we have 32 registers available, specifying a register as operand takes 5 bits ($2^5 = 32$). As we will see, instead of the value to be used being stored in a register, the operand value can also be part of the instruction itself. These instructions are called 'immediate'. An operand can also be a something that is in main memory, the address of which can either be specified as immediate or be contained in a register.

That brings us to the memory organization. We have to observe that both the instructions as well as the data reside in the same main memory. In fact, technically speaking, code is fully indistinguishable from data; an instruction is 32 bit, the bus is 32 bit and a register is 32 bit. An instruction can be treated as data. As we will see, data can also be memory addresses.

However, there is one small difference between data and code and that makes that the organization of the memory follows a so-called Von Neumann architecture. It means that program instructions and data are separated in memory. Each residing in its own block of memory, the former named the 'code segment' of memory (normally starting at address `0x00400000`) and the latter the 'data segment' (normally starting at address `0x10010000`). The code cannot change the code, it can only change the data! That is, *technically* it is possible, but it is a 'mortal sin' in programmers ethics to do so. Data is never code and code is never data. The code can only change contents of the registers or of memory addresses in the data segment, never in the code segment. Even if it is technically possible, an educated programmer will never write a program that breaks this fundamental law in programming.

In MIPS, the two segments are identified by the assembler directives `.data` and `.text` for data segment and code segment respectively. If we declare something in the data segment by writing a name (label) for it followed by a semicolon, for instance a word with the `.word` directive, as in (note the semicolon)

```
  myword:  .word 64
```

the compiler reserves space in the data segment, places the (optional) value (64 in this case) in it and remembers the label name (`myword`), a pointer — an address; *not* the value itself — to it in a table it keeps on the side while assembling our program.

Similarly, the use of a label in the code segment just stores the label name and current position — the address in the code segment of the next instruction — in the table that is kept on the side. For example,

```
  mycode:
      li $t0, 4
```

will keep the address of instruction `li $v0, 4` in the label-look-up table.

Every time the assembler now encounters a reference to the label (a pointer to data or code), it just uses (in most cases simply substitutes) the value it has saved in this label-look-up table. After having completed the assembling of our code and translated our instructions into machine language, the names of the labels have been lost and the label-look-up-table no longer exists.

Related, if we want to define a 'constant', label a value, we can do this with the `.eqv` assembler directive. As an example,

```
  myvalue .eqv 64
```

It just remembers the combination label and *value* 64 in the table. In this case, after our three declarations the label-look-up table now looks like this, two pointers to memory (one to data and one to code) and one value:

Label	*Value*
`myword`	`0x10010000`
`mycode`	`0x00400000`
`myvalue`	`0x00000040`

Note that the declarations above are fully equivalent, they all just create label-value pairs in the table, but the first one resembles the declaration of a variable in the C language, while the latter resembles the `#define` C-compiler directive defining constants. (The middle one having no C equivalent). For MIPS it makes no difference. If we wish, we can jump to our constant, `j myvalue`, which the compiler might even accept if the value coincidentally is within the code segment

Table III: Some compiler directives

Anywhere

`#`	ignore rest of line (comment)
example: `# this text will be ignored`	

Data segment

`label:  .word value(s)`	reserve 4 bytes in data segment for every value given, place value(s) in data segment, store (label,address) in compiler table
example: `numbers:  .word 1, 2, 3`	
example: `numbers:  .word 0:129 # reserve 130*4 bytes space`	
`.eqv label value`	store (label,value) in compiler table
example: `.eqv four 4`	
`label:  .ascii "string"`	store ASCII string in data segment and remember (label,address) in compiler table
example: `name:  .ascii "Peter"`	
`label:  .asciiz ''string''`	same as above but add `NULL` (0x00) to end of string

Code segment

`label:`	store (label,next code address) in compiler table
example: `main:`	

range. (YWIYGI, ”you wanted it, you got it!”). Very likely though, the compiler will warn us: ”Jump target word address beyond 26-bit range”, or so. (Actually, the compiler used by the author, MARS, does refuse to compile this turd of programming).

Input/output (system calls) and memory access

A computer without output is as silly as the concept of WOM (write-only memory). A computer without input is strange, but possible, but without output is silly. Therefore, we first have to learn to generate output. As a tradition in programming, the first program we write in a new language will print ”Hello world!” For that we need to have access to input and output functions that in the MARS simulator of MIPS are part of the environment; no need to write them ourselves. These are so-called system calls which are summarized in Appendix C. The method consists of choosing the system call we want by placing the correct number of the system call in register `$vo`, place any arguments in the a-registers, and then issue a `syscall`. As Appendix C shows, printing a string is system call number 4 and we have to place the address of the null-terminated string in register `$a0`. That’s all there is to it. The program below, `hellow.asm`, shows how this works. First define a null-terminated string by the definition `.asciiz` in the data segment, then we prepare the registers and issue a syscall. Note that to cleanly end the program, we issue a ’syscall 10’, which will print the message
”`-- program is finished running --`”
and stop execution.

```
########################################################
#                                                      #
#   MIPS assembler program that prints "Hello world!"  #
#                                                      #
########################################################

#data segment with 'variables'; starts at 0x10010000
.data

hellow: .asciiz "Hello world!"

#code segment with instructions;  # starts at 0x00400000
.text

start:
```

```
li $v0, 4
la $a0, hellow
syscall  # 4: print string

#terminate program:
stop:
li $v0, 10
syscall
```

Output:

```
Hello world!
-- program is finished running --
```

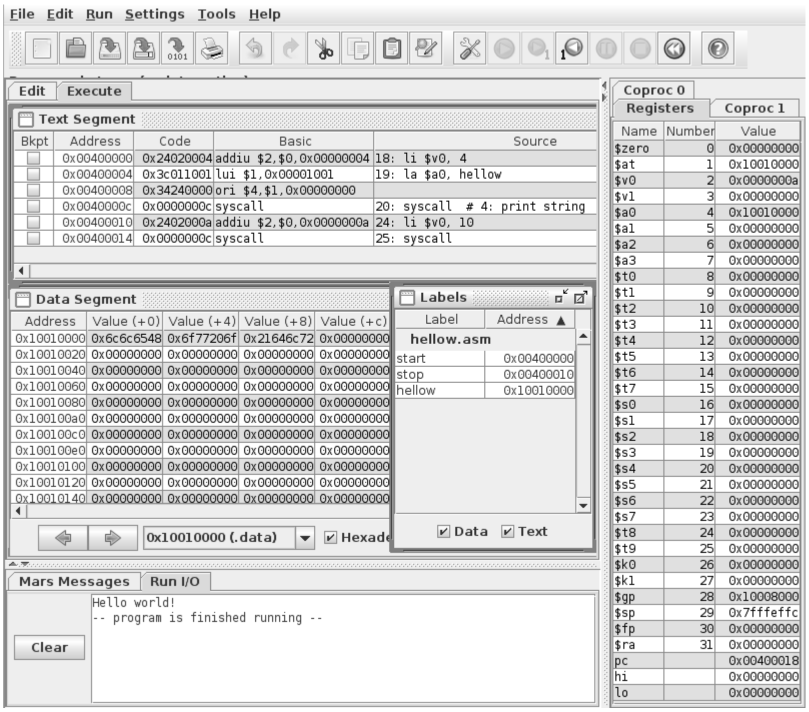
See Picture 10 how this looks in the MARS environment. Analyzing the program we see that assembler does not have any variables. The only thing assembler has is values and addresses. When we 'declare' a variable such as `hellow`, what, in fact, is done is

- Space is reserved in main memory (on the heap) enough to store the data
- The assembler remembers a pointer to this space and saves this in a table, together with the name of the label we have given to it
- Every time we use the label in further reference, the label is looked up in the table and the compiler substitutes it with the value

In this case the label-look-up table looks like this (see panel Labels in Picture 10):

Label	*Value*
`start`	0x00400000
`stop`	0x00400010
`hellow`	0x10010000

The first two labels are pointers to instructions in the code segment. The last is a pointer to the string in the data segment. When the instruction `la $a0, hellow` is encountered, this is replaced by the



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24020004	addiu $2,$0,0x00000004	18: li $v0, 4
☐	0x00400004	0x3c011001	lui $1,0x00001001	19: la $a0, hellow
☐	0x00400008	0x34240000	ori $4,$1,0x00000000	
☐	0x0040000c	0x0000000c	syscall	20: syscall # 4: print string
☐	0x00400010	0x2402000a	addiu $2,$0,0x0000000a	24: li $v0, 10
☐	0x00400014	0x0000000c	syscall	25: syscall

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x6c6c6548	0x6f77206f	0x21646c72	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000

Label	Address ▲
hellow.asm	
start	0x00400000
stop	0x00400010
hellow	0x10010000

```
Hello world!
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x10010000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400018
hi		0x00000000
lo		0x00000000

Picture 10: Example of the compiled and ran program `hellow.asm`

compiler with a `li $a0, 0x10010000`, loading the value directly into the register. We will get back to this later. Let us here analyze basic instructions of MIPS. Starting with the simplest of all, `move`.

Moving around — shunting — data in the registers is done by the move instruction (`move`), which might be confusing, since it does not actually *move* anything, but rather *copies* things. (The source register retains the original value as well). The move instruction is a pseudo instruction in that it is not part of the MIPS instruction set, but MARS implements it with other MIPS instructions,

- `move $t0, $t1`: copy the contents of `$t1` to `$t0` (= `add $t0, $t1, $zero`)

If we want to directly load a number into a register, we can use 'load immediate', `li`. In that case, the value to store in the register is part of the instruction, this value we call `immediate`.

- `li, $t0, value`
  store the 32 bit `value` into register `$t0`

This needs some explanation. Since specifying one of the 64 opcodes of MIPS takes 6 bits, and specifying one of the 32 registers as destination takes 5 bits, storing directly a value of 32 bits into a register would take $6+5+32 = 43$ bits. That obviously does not fit into the 32 bits of the instruction register. The solution is that the compiler translates the `li` instruction into two separate instructions: first loading the upper part of `value` into the temporary register `$at` by 'load upper immediate' (`lui`) and then bitwise OR'ing the lower part of the `value` to it by 'or immediate' (`ori`), storing the final result in the destination register (`$t0` in this case):

```
  li $t0, 0x12345678
```

translates into

```
  lui $at, 0x00001234
  ori $t0, $at, 0x00005678
```

Note that if the value to load is less or equal than `0x0000FFFF` the `lui 0x00000000` is redundant and a `li 0x0000abcd`, for example, translates into a single instruction,

```
  ori $t0, $zero, 0x0000abcd
```

or 'add immediate unsigned', which is effectively the same,

```
  addiu $t0, $zero, 0x0000abcd
```

The instruction `li` is therefore also a 'pseudo instruction'. In this case the MARS compiler translates it into one or two MIPS instructions. For reasons of legibility of the program in other cases, the exact same pseudo instruction is also called 'load address',

- `la, $t0, value`
  exactly the same as 'load immediate' (`li`), but pronounced as 'load address'

The instructions are to have access to the code segment in main memory. While many things can be done at the registers — and working with registers is much faster than working with main memory!

— sometimes we will want to store our data. Moreover, we have only 32 registers available, which is rather few. With addresses of 32 bits, each capable of storing 8 bits of information, we can store $2^{32} = 4$ GB of data. So, we will load things from memory, calculate things as much as is possible in the fast registers and then store the end result back in memory. We thus need instructions for loading and storing information. The basic instructions are `lw` and `sw`, 'load word' and 'store word', respectively.

- `lw, $t0, addressvalue`
  load the 32-bit contents of the 4 consecutive addresses pointed to by `addressvalue` into register `$t0`

- `lw, $t0, offset($t1)`
  load the 32-bit contents of the 4 consecutive addresses pointed to by the pointer in `$t1` plus `offset` into register `$t0`

- `sw, $t0, addressvalue`
  store the contents of register `$t0` into the 4 consecutive addresses pointed to by `addressvalue`

- `sw, $t0, offset($t1)`
  store the contents of register `$t0` into the 4 consecutive addresses pointed to by the pointer in `$t1` plus `offset`

Also smaller sized load and store instructions exist: `sh`, `lh`, for storing and loading halfwords (16 bits) and `sb`, `lb`, for storing and loading bytes (8 bits).

Now let's take a look again at our compiled program `hellow.asm`, see Picture 10. We see that the compiler has translated our

```
 li $v0, 4
```

into

```
 addiu $2, $0, 0x00000004
```

which accomplishes the desired effect, adding 4 to 0 and placing it in register `$2`, which is `$v0`, as can be seen in the right side of the image. Indeed, the program shows that at finishing the program, the contents of `$v0` are `0x0000000a` which were placed there by a another `li` to invoke a system call `syscall 10` to end the program.

Now, let's look at this specific instruction `li $v0, 4` that was translated into `addiu $2, $0, 0x00000004`. (See the figure; the column named 'Basic'). If we look in the appendix with MIPS instructions (Appendix A), we see that `addiu` has an opcode equal to `001001`, after which follow five bits for the source register, five bits for the target register and 16 bits for the immediate value: the source register is `$zero`, or `$0` which is in 5 bits equal to `00000` and the target register is `$v0` (`$2`), which is `00010`, the immediate value is 4, which is `0000000000000100`, so the final instruction is:

mnemonic	source	target	immediate
addiu	$zero	$v0	4
001001	00000	00010	0000000000000100

Regrouping in units of 4 bits:

```
0010 0100 0000 0010 0000 0000 0000 0100
= 0x24020004
```

which is the compiled code, as can be seen in the figure (in the column named 'Code'). It is placed in the four addresses starting from `0x00400000` (see the column named 'Address'). When the program is run, the program counter is set to this value and the hardware fetches this `addiu` instruction, places it in the instruction register and executes it. That is, it latches the values of the register `$0` and the immediate value into the ALU, adds the two operands and latches the result into register `$v0`. It then adds 4 to the program counter.

The next two compiled code instructions (`la` that is translated into `lui` plus `ori`) load the address (`0x10010000`) of the text "Hello world!" of the data segment into register `$a0` (=`$1`). As can be seen in the Data Segment panel, the text is stored as (hexadecimal)

```
 6c   6c   65   48   6f   77   20   6f   21   64   6c   72    00
  l    l    e    H    o    w         o    !    d    l    r  [eos]
```

([eos] = string terminator, 00), which gives us an indication of how data is represented and stored by the MARS/MIPS environment and architecture.

At the end of the program we can verify in the Registers panel that:

- The program counter is `pc = 0x00400018`, which is the address of the last syscall plus 4
- The value of `$zero` is still `0`, since it is hard wired

- The value of `$v0` is `10`, set there for the last syscall (terminate execution)
- The value of `$a0` is `0x10010000`, still pointing to the string "Hello world!"

As can be seen, what at first seems rather complicated, turns out to be very simple after all.

Arithmetic

Basic arithmetic in MIPS consists of simple mathematical operations of addition (`add`), subtraction (`sub`), multiplication (`mult`) and division (`div`). Multiplying two 32-bit numbers can, in principle, result in a 64-bit integer. Such a result does not fit in a 32-bit destination register. To avoid this problem, MIPS engineers added two registers to the 32 standard 32-bit registers described before, namely `HI` and `LO`. These will contain the first 32 and last 32 bits of the multiplication, respectively. A similar problem occurs in divisions: the integer division result is stored in `LO`, while the remainder of the division is stored in `HI`. These can be moved to normal registers by instructions move-from-low (`mflo`) and move-from-high (`mfhi`).

Additionally, the ALU can perform some basic logic operations `or`, `nor`, `and`, `xor`). These mathematical and logic operation all take three operands: two input operands, of which one can be immediate (contained in the instruction) and one output register. (Question: why would it not make sense to have both input operands of the immediate type?). All these operations are straightforward and will not be explained here further, with the only comments that subtractions do not have an 'immediate' version (again: why not?) and that all mathematical operations can be done with either signed or unsigned numbers.

Apart from this, the ALU can shift the bits of a register left or right a number of places determined either by the immediate value or by the contents of a specified register, and can store the result in another register. They exist in two variants. The difference between the variants, that are called 'logic' and 'arithmetic', is that the former merely shifts the bits a certain amount of places, filling the created gaps with 0s, while in the arithmetic variants the high-order bits are sign extended, thus a right shift fills the utmost left bits with a copy of the utmost left bit before shifting; if, before shifting, the MSB was 1, they are filled with 1s, otherwise they are filled with 0s. This way, arithmetic shifting right is like divisions by 2, also for negative numbers. Note that left logic shifts and left arithmetic shifts are the same and therefore, only a logic instruction (mnemonic) exists.

An example of the effects of the three versions a 1-bit shift operation on a positive and a negative operand is given here in the table below:

mnemonic	operand	decimal	result	decimal

`srl 1`	`0x00000004`
	`0000...0100`
		`4`	`0x00000002`
			`0000...0010`
				`2`

`sra 1`	`0x00000004`
	`0000...0100`
		`4`	`0x00000002`
			`0000...0010`
				`2`

`sll 1`	`0x00000004`
	`0000...0100`
		`4`	`0x00000008`
			`0000...1000`
				`8`

`srl 1`	`0xfffffffc`
	`1111...1100`
		`-4`	`0x7ffffffe`
			`0111...1110`
				`2147483646`

`sra 1`	`0xfffffffc`
	`1111...1100`
		`-4`	`0xfffffffe`
			`1111...1110`
				`-2`

`sll 1`	`0xfffffffc`
	`1111...1100`
		`-4`	`0xfffffff8`
			`1111...1000`
				`-8`

Shift instructions where the number of bits to shift is not an immediate value contained in the instruction, bur given in a register instead, are specified by an extra 'v'. All variants of shifting instructions have opcode `000000`, but are differentiated through the specification of the function code, which for these instructions is the last 6 bits of the instruction. In the list here below, '`s`' means shift, '`r`' means right, '`l`' means left or logic, '`a`' means arithmetic, '`v`' means value (contained in a register).

mnemonic	opcode	function	direction, type, value is
`sll`	000000	000000	left, (logic & arith.), immediate
`srl`	000000	000010	right, logic, immediate
`sra`	000000	000011	right, arithmetic, immediate
`sllv`	000000	000100	left, (logic & arith.), in register
`srlv`	000000	000110	right, logic, in register
`srav`	000000	000111	right, arithmetic, in register

Sometimes we want to have the bits that leave the register on one side reappear on the other side. Such rotate instructions are not part of MIPS, but our MARS compiler can translate the pseudo-code easily

mnemonic	opcode	function	direction, type, value is
`rol`	-	-	rotate left
`ror`	-	-	rotate right

As an example, `rol $t1, $t0, 4` is implemented as

```
srl $at, $t0, 28
sll $t1, $t0, 4
or $t1, $t1, $at
```

which indeed does the job.

Exercises:

1. Write a MIPS program that asks two numbers and prints their sum, difference, product and quotient.
2. Write a MIPS program that asks a number and calculates the number multiplied by 10.5, with this multiplication not done by a `mult` instruction, but by shifting and summing.

Jump and branch. (goto, if ... then ...)

The next instructions we will look at are simple unconditional jumps and conditional jumps (see Picture 11). Most modern fourth-generation programming levels do not use this concept, with maybe the only exception being BASIC. The reason is that a program rapidly turns out to be spaghetti, and where the idea in programming is to write programs in as-close-as-possible-to English, writing in Italian is highly discouraged. The BASIC equivalent of a jump instruction is `GOTO label`, and the conditional jumps are `IF condition THEN GOTO label`. In the latter, the program can continue at two different paths, and therefore this technique is called 'branching'.

- `j label`
  This simply puts the value of the address label into the program counter: `label` $\rightarrow$ `pc`

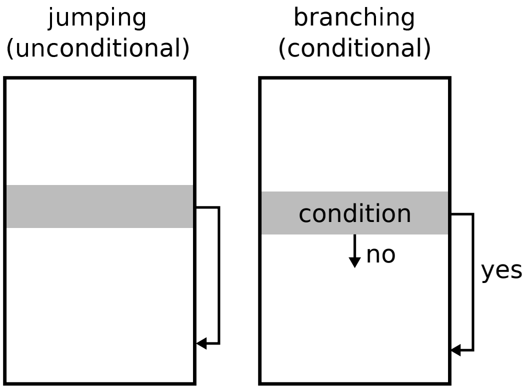
Note that the jump addresses are part of the instruction and these instructions are thus of the 'immediate' type described before. Since 6 bits are used for specifying the operation (the 'opcode'), only 26 bits remain for the address. MIPS developers were very smart. Realizing that instructions are always 4 bytes apart, the last two bits of the address are redundant (always `00`; in some cases we have to 'align' the code with `nop` — no-operation — instructions) and are thus implicit in the address specified. Still, that makes 28 bits of address, and only 2^{28} = 256 MB addressable of the total 4 GB of memory. All addresses are thus relative to the program counter (`pc`) when jumping. To be more precise, the address jumped to is the first 4 bits of the actual program counter (`p`) plus the 26 immediate bits (`i`) of the instruction multiplied by 4 (left-shifted two bits).

```
ppppiiiiiiiiiiiiiiiiiiiiiiiiii00 → pc
```

This way, the first 4 bits of the program counter represents something like a 'page' we are working on and for that reason we call this technique 'paging'. (Long) jumping to a different page has to be done in a different way. For example placing an address in a register and issuing a jump-on-register (`jr`).

Conditional jumping is called 'branching', for which we have this set of instructions:

- `branch-condition label`
  This puts the value of the address label into the program counter when the condition, comparing two registers, is true, otherwise it defaults to adding 4 to the program counter:



Picture 11: Difference between (unconditional) jumping and (conditional) branching

condition=true: `label` → `pc`
condition=false: `pc+4` → `pc`

The only two possible branch instructions are:
`beq $t0, $t1, addr`: Branch to `addr` if operands are equal
`bne $t0, $t1, addr`: Branch to `addr` if operands are not equal
All other branching variants are pseudo instructions:
`blt $t0, $t1, addr`: Branch to `addr` if `$t0` $<$ `$t1`
`bgt $t0, $t1, addr`: Branch to `addr` if `$t0` $>$ `$t1`
`ble $t0, $t1, addr`: Branch to `addr` if `$t0` $\leqslant$ `$t1`
`bge $t0, $t1, addr`: Branch to `addr` if `$t0` $\geqslant$ `$t1`
`beqz $t0, addr`: Branch to `addr` if `$t0` $= 0$
`bnez $t0, addr`: Branch to `addr` if `$t0` $\neq 0$
`bltz $t0, addr`: Branch to `addr` if `$t0` < 0
`bgtz $t0, addr`: Branch to `addr` if `$t0` > 0
`blez $t0, addr`: Branch to `addr` if `$t0` $\leqslant 0$
`bgez $t0, addr`: Branch to `addr` if `$t0` $\geqslant 0$

As an example,
`beqz $t0`: Branch if `$t0` $= 0$
is implemented as
`beq $t0, $zero`

And

`bgt $t0, $t1`: Branch if `$t0` $>$ `$t1`

is implemented in two instructions as

```
slt $at, $t0, $t1
bne $at, $zero
```

Here the instruction `slt` is writing the condition — 'less-than' — in a specified register

- `slt $t0, $t1`: Set (to 1) if `$t0` is less than `$t1` (set to 0 otherwise).

An instruction that has no 'greater than' or 'equal to' variants (I leave it to the reader to think why not), but has an 'immediate' version and can be for both signed and unsigned numbers: `slt`, `slti`, `sltu`, and `sltiu`.

The jump-to addresses for these conditional jump instructions work differently compared to the addresses of unconditional jumps (`j`) described earlier, yet also uses the same memory efficiency 2-bit-shifting technique. Conditional jump addresses are relative to the actual current value of the program counter, which, immediately after fetching the instruction from memory, is already updated to the next instruction at `pc+4`. So, if jumping, the program continues at `(address+4) + immediate`$\times$`4`, with `address` the address of the branching instruction. For simple jumps described earlier, the immediate value is 26 bit and addresses span $2^{26+2} = 256$ MB, as shown above. For conditional jumps, the specification of the registers to compare take up an additional 10 bits (5 bits for each register) and the immediate address specification is thus limited to only 16 bit, making the jump addresses in these instructions span only $2^{16+2} = 256$ kB; conditional jumps are relatively local. If we need longer jumps, we need to make a conditional jump to an instruction with an unconditional jump. If we need even longer jumps, we need to place the full 32-bit address in a register and issue a jump-register instruction (`jr`), which simply copies the 32-bit contents of the register to the 32-bit program counter.

Some final remarks about conditional branching. First of all, note that there does not exist in assembler an 'else' clause, nor does the concept of multiple choice (like '`switch`' in C, or '`case ... of`' in Pascal) exist. All conditional branching in assembler is either jumping to that address, or continuing with the next instruction at `pc+4`.

Finally, note here the important difference between high level languages such as C:

```
if (t0==t1)
      instructions-when-true
```

and assembler:

```
beq $t0, $t1, label
      instructions-when-false
```

They are in opposite order.

Here is a worked-out example of branching. It inputs two numbers and prints "same" if they are equal, and "different" otherwise.

Source code:

```
###############################################
#  MIPS assembler program that shows how      #
#    to implement branching                   #
###############################################

.data
number: .asciiz "Give a number: "
difftxt: .asciiz "Numbers are different\n"
sametxt: .asciiz "Numbers are the same\n"

.text
la $a0, number
li $v0, 4
syscall                  # print string
li $v0, 5
syscall                  # read int into $v0
move $t0, $v0
li $v0, 4
syscall                  # print string
li $v0, 5
syscall                  # read int into $v0
move $t1, $v0

li $v0, 4
beq $t1, $t0, same       # if ($t1==$t0)
different:               # false
  la $a0, difftxt
  syscall
  j terminateprog
same:                    # true
  la $a0, sametxt
  syscall
terminateprog:
li $v0, 10
syscall                    # terminate program
```

Output:

```
Give a number: 5
Give a number: 6
Numbers are different

-- program is finished running --
```

Loops; (for, while, do-while)

In high-level programming we normally have to our disposition the concept of loops, and mostly they can be divided into

- for: used for loops that have a number of iterations well known at the beginning of the loop and they are countable (thus integers are used).
- while-do: used for loops that have an a-priori undetermined number of iterations (as in: while input chars available do), and used for loops that use floating point numbers. The condition is checked in the *beginning* of the loop, so it might occur that not a single iteration is done and none of the instructions within the loop are executed.
- do-while: the same as while-do, but the condition is checked at the *end* of the loop, so the instructions within the loop are executed at least once.

The concept of loops is not available in assembler, but it is not very difficult to implement with the branching instructions learned before. The code below gives an example of how to implement a for loop, printing n times a certain text.

Source code:

```
###############################################
#  MIPS assembler program that shows how      #
#    to implement a for loop                  #
#    n times printing a text                  #
###############################################

.data
prompt: .asciiz "Give a number (n): "
hellow: .asciiz "Hello world!\n"

.text
  la $a0, prompt
  li $v0, 4
  syscall                 # print prompt
  li $v0, 5
  syscall                 # read int n into $v0
  move $t0, $v0

# we will implement the following C code:
#     for (i=0; i<n; i++)
#        printf("%s", benfica);
# i is stored in $t1
# n is stored in $t0

  la $a0, hellow
  li $v0, 4
  move $t1, $zero         # initial value of i=0
startloop:
  beq $t1, $t0, exitloop  # exit if end value is reached
  syscall                 # printf
  addi $t1, $t1, 1        # i++
  j startloop             # go back to start of loop
exitloop:
  li $v0, 10
  syscall                 # terminate program
```

Compiled program:

Address	Code	Basic	Line:	source code
0x00400000	0x3c011001	lui $,0x00001001	12:	la $a0, prompt
0x00400004	0x34240000	ori $4,$1,0x00000000		
0x00400008	0x24020004	addiu $2,$0,0x00000004	13:	li $v0, 4
0x0040000c	0x0000000c	syscall	14:	syscall
0x00400010	0x24020005	addiu $2,$0,0x00000005	15:	li $v0, 5
0x00400014	0x0000000c	syscall	16:	syscall
0x00400018	0x00024021	addu $8,$0,$2	17:	move $t0, $v0
0x0040001c	0x3c011001	lui $1,0x00001001	23:	la $a0, hellow
0x00400020	0x34240010	ori $4,$1,0x00000010		
0x00400024	0x24020004	addiu $2,$0,0x00000004	24:	li $v0, 4
0x00400028	0x00004821	addu $9,$0,$0	25:	move $t1, $zero
0x0040002c	0x11290003	beq $9,$8,0x00000003	27:	beq $t1, $t0, exitloop
0x00400030	0x0000000c	syscall	28:	syscall
0x00400034	0x21290001	addi $9,$9,0x00000001	29:	addi $t1, $t1, 1
0x00400038	0x0810000b	j 0x0040002c	30:	j startloop
0x0040003c	0x2402000a	addiu $2,$0,0x0000000a	32:	li $v0, 10
0x00400040	0x0000000c	syscall	33:	syscall

Output:

```
Give a number: 5
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!

-- program is finished running --
```

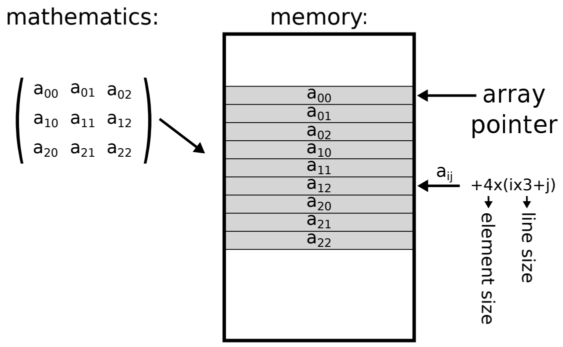
The compiled code is also shown here to highlight the relative branching address method described in the previous section. Especially to note here is the `beq` instruction at address `0x0040002c` on line 27: the `immediate` value in the instruction is only 3. Therefore the program (if `$t0` equals `$t1`) jumps to `(0x0040002c+4) + 4`$\times$`3` = `0x0040003c`, just beyond the for loop.

Exercise:

> Write a MIPS program that calculates the first 100 prime numbers.

Arrays and structures

A very useful concept in a high-level programming language is the joining of data in arrays or structures. We can imagine a mathematical



Picture 12: An array in programming is a mapping of a multi-dimensional object, such as a mathematical matrix shown here, to a linear memory 'vector'. We have to calculate the position of the element in memory based on its indexes. The address of element a_{ij} can be found as: `address = arraypointer + elementsize`$\times$(*i*$\times$`elements_per_line + `*j*)

vector, or matrix of elements, all of the same type, or a collection of data, a 'file' of a person in a database. From programming lectures we know that data of identical type are stored in arrays, where each element has an index, or indexes.

In assembler the concept of arrays and structures does not exist. Our program thus has to calculate where all the elements of data reside in memory. We have to map the structure of our data onto a one-dimensional array of our linear memory (see Picture 12). This is a simple calculation if we know the size of each element of our data. As an example, if we have a vector of words (4 bytes each), then the address of element i, assuming the first element is element 0, can be found as `vectoraddress+i`$\times$`4`. A two-dimensional array can be implemented as well, but the calculations are slightly more complicated. Here is an example that checks which point in space, with coordinates (x,y,z), has the smallest sum of coordinates $x+y+z$. We assume that the data are placed in memory as x_0, y_0, z_0, x_1, y_1, z_1, x_2 ... starting from address `array`. This means that to find the coordinate x of point i in space, we calculate `array+3`$\times$`4`$\times$`i`. The '4' comes from the size of an element (1 word is 4 bytes), the '3' from the size of a line in the matrix (1 point has 3 coordinates). For coordinate y and z we add 4 and 8 to this, respectively.

Source code:

```
###################################################################
#                                                                 #
#   MIPS assembler program that shows how to implement an array   #
#                                                                 #
###################################################################

.data
mintext: .asciiz "minimum sum: "
indextext: .asciiz "  at index: "

# an array of 10 times (x,y,z)
array: .word
   1, 10, 18,      # 0
   2, 2, 20,       # 1
   13, 13, 1,      # 2
   20, 20, 100,    # 3
   8, 9, 10,       # 4
   11, 12, 1,      # 5
   20, 1, 2,       # 6
   18, 8, 8,       # 7
   9, 9, 3,        # 8
   10, 9, 5        # 9

.text
# t8 stores minsum
li $t8, 999999

li $t0, 0
li $t1, 10

startloop:
beq $t0, $t1, exitloop

la $a0, array
mul $t3, $t0, 12
add $a0, $a0, $t3   # a0 = array + 3*4*i
lw $t4, 0($a0)      # x_i
lw $t5, 4($a0)      # y_i
add $t4, $t4, $t5
lw $t5, 8($a0)      # z_i
add $t4, $t4, $t5   # t4 = x_i + y_i + z_i
bgt $t4, $t8, continue  #jump if sum is larger than old minimum sum
#if not, then new minimum sum found
```

```
move $t8, $t4  # save new minimum
move $t7, $t0  # save index

continue:
addi $t0, $t0, 1    # increment i
j startloop

exitloop:
li $v0, 4
la $a0, mintext
syscall
li $v0, 1
move $a0, $t8
syscall
li $v0, 4
la $a0, indextext
syscall
li $v0, 1
move $a0, $t7
syscall

#terminate program
li $v0, 10
syscall
```

Output:

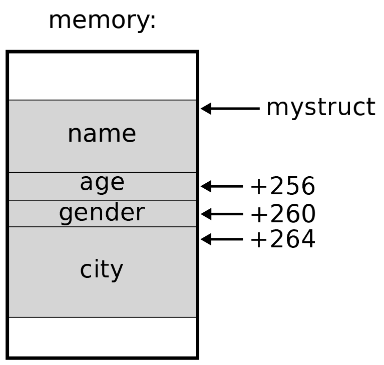
```
minimum sum: 21  at index: 8
-- program is finished running --
```

A structure is a set of data of (possibly) different type. Yet, the technique of finding a field of our 'struct' is very similar to finding an element in an array. The example below makes this more clear (see also Picture 13):

Source code:

```
###################################################################
#                                                                 #
#   MIPS assembler program that shows how to implement a struct   #
#                                                                 #
###################################################################

.data
nameprompt: .asciiz "name:"
```



Picture 13: An example of a structure (set of informations of different types) used for the programming example

```
ageprompt: .asciiz "age:"
genderprompt: .asciiz "gender:"
cityprompt: .asciiz "City:"

mystruct: .word 0:129
   # name:   256 chars ASCII = 64 word
   # age:    1 byte = 1 word
   # gender: 1 char = 1 word
   # city:   256 chars ASCII = 64 word
   #-------------------------+
   #         130 words =  520 bytes

.text

li $v0, 4
la $a0, nameprompt
syscall
la $a0, mystruct
li $a1, 256
li $v0, 8
syscall   # read string. $a0,= string address, $a1 = max length

li $v0, 4
la $a0, ageprompt
syscall
li $v0, 5
```

```
syscall   # read int into $v1
la $a0, mystruct
addi $a0, $a0, 256  # we have to calculate where the age int is
                    #  in the struct
sw $v0, 0($a0)

li $v0, 4
la $a0, genderprompt
syscall
la $a0, mystruct
addi $a0, $a0, 260 # we have to calculate where the gender byte is
                   #  in the struct
li $a1, 10 # it reads max 9 chars. Note: It may overwrite the
           #  adjoining city string!
li $v0, 8
syscall   # read string

li $v0, 4
la $a0, cityprompt
syscall
la $a0, mystruct
addi $a0, $a0, 264 # we have to calculate where the city string is
                   #  in the struct
li $a1, 256
li $v0, 8
syscall   # read string. $a0,= string address, $a1 = max length

#terminate program
li $v0, 10
syscall
```

Output:

```
name:John Doe
age:23
gender:m
City:Amsterdam
-- program is finished running --
```

Floating-point numbers

So far all the calculations were done with integer numbers, either exclusively positive ($\mathbb{N}$) or both positive and negative ($\mathbb{Z}$). However, from mathematics we know that there also exist real numbers ($\mathbb{R}$). And sometimes we would like to do calculations with these numbers. It then becomes important to realize that our computer is a finite state machine and registers and memory contents have a finite number of possible values. Whereas in integer calculations this limitation is, to a certain point, rather irrelevant and only limits the *range* of calculations, for floating-point calculations these limitations are severe and we have to keep them in mind. A single float of 32 bits can, for instance, take only 2^{32} (approx. 4 billion) different values, there where the number of real numbers is infinite, even if we were to limit the range of the numbers to an interval (for instance only between 0 and 1, an interval that contains an infinite number of numbers). Our calculations are bound to be incorrect. But how incorrect are they going to be? And is that acceptable, or not?

In science lectures we have learned the so-called scientific notation, which is a way of writing a number as a product of a fraction (f) and an exponent of 10,

$$n = f \times 10^e,$$

for instance 3.28×10^{21}. Because many computers were limited to writing text with ASCII-only, this scientific notation was also often written in ASCII in so-called engineering notation, `n = 3.28E21`. The wording 'floating point' is used in computer jargon, because the decimal point can 'float' between the digits, as in,

$$3.28 \times 10^{21} = 0.328 \times 10^{22} = 32.8 \times 10^{20}.$$

Very important to note at this moment is that any number that has a finite number of digits can always be described with only integers, without the floating point, by just floating the point in the fraction until its mantissa (the part after the floating point) contains only zeros (and can thus be omitted). As an example,

$$3.2800 \times 10^{21} = 328 \times 10^{19},$$

which is described by $f = 328$ and $e = 19$, both integer numbers. For this reason, we can implement floating point numbers with finite-state integer machinery such as our MIPS architecture.

In the same way, we can also always 'normalize' numbers by adjusting the exponent in such a way that the fraction falls within certain limits, for instance by forcing the part of the fraction before the floating point to be one non-zero digit, effectively limiting the range of the fraction between 1.000... and 9.999..., or in another scheme by limiting it to 0.100... and 0.999.... Such normalization will come in very handy because for binary numbers limiting the numbers in the same way from 1.000... to 1.111... means that the numbers always start with "1.", so that part can be omitted because it is information that is redundant!

Now lets take a specific example of floating point numbers with 3 digits for the fraction and 2 digits for the exponent, both also including a sign. What we can say is:

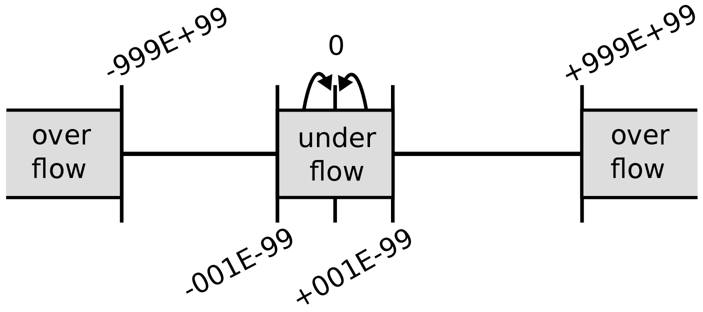
- The largest negative number is -999×10^{99}
- The smallest negative number is -001×10^{-99}
- Zero: $\pm 000 \times 10^{\pm xx} = 0$ (there are many ways of writing zero)
- The smallest positive number is $+001 \times 10^{-99}$
- The largest positive number is $+999 \times 10^{99}$

This defines 7 regions in the number scale. As can be seen, and as marked in Figure 14, some numbers are unattainable with this number system. If the result of a calculation is too big and falls beyond $\pm 999 \times 10^{99}$ this is called 'overflow'. A similar problem occurs when the number is too small and falls into the 'underflow' region. Most calculators treat this number simply as 0 in order not to generate an error.

Note that the absolute error in the numbers — the distance between two adjacent numbers — is varying from small numbers to big numbers, namely from 1×10^{-99} to 1×10^{99}, but the relative error is rather constant over the entire range, namely about 1/1,000. Still, this error makes that our floating-point calculations on a computer are not (always) exact. The rounding introduces errors.

We can also invent other combinations of number of digits for the fraction and the exponent. If we increase the number of digits for the fraction and less for the exponent, the relative error of our calculations drop, but, as a price to pay, the range of our numbers also drops. For instance, for a system with 4 digits for the fraction and 1 for the exponent:

- The largest negative number is -9999×10^{9}
- The smallest negative number is -0001×10^{-9}
- Zero: $\pm 0000 \times 10^{\pm x} = 0$



Picture 14: The seven ranges of base-10 floating-point numbers with 3 digits for the fraction and two for the exponent. When the number is too big (on either side), it is called 'overflow'. When it is too small it is called 'underflow', which is normally mapped to 0.

- The smallest positive number is $+0001 \times 10^{-9}$
- The largest positive number is $+9999 \times 10^{9}$

with a relative error of 1/10,000.

❇

The IEEE 754 standard for floating point numbers was developed to make it possible for engineers from various architectures talk with each other. It is based on fractions and exponents that are based on the binary number system, so

$$n = f \times 2^e,$$

with both f and e binary integers that were described before. IEEE 754 has the following features:

	1	8	23
single:	$\pm$	`...exp...`	`...frac...`

	1	11	52
double:	$\pm$	`......exp......`	`......frac......`

- It has three formats with different total bit-lengths: single (32 bits), double (64 bits), and extended (80) bits. As we will see, MIPS has hardware co-processors that can perform dedicated floating point operations with the first two types of numbers. Calculations with extended numbers have to be emulated with software.

- One sign bit, at the position of the MSB. 0 = positive, 1= negative.
- The exponent is 8 bits long for singles and 11 bits long for doubles.
- The exponent is written in the format 'excess 127' (for singles), and 'excess 1023' (for doubles). It means that the bits that represent the exponent are a binary positive integer to which 127 is subtracted (or 1023) to find the exponent used in the calculation. Example: If the exponent pattern is `exp=00001001`, this is equal to 9, so $e = 9 - 127 = -118$. (The bit pattern `11111111` is reserved for special use).
- The fraction bit pattern (`frac`) contains 23 bits (single) or 52 bits (double).
- Normalized fractions start with "1.", so it needs not be written, it is 'implied'; the bits only represent the digits after the floating point. This bit pattern f plus the leading "1." we call the significand: $s = 1.f$; the final number thus being (for singles and doubles respectively)

  $$\begin{aligned} n &= \pm(1.\text{frac}) \times 2^{\text{exp}-127}, \\ n &= \pm(1.\text{frac}) \times 2^{\text{exp}-1023}. \end{aligned}$$

  (Note the strange mixed writing of the above, binary for the significant and decimal for the exponent. It is done for clarity; not many people would understand $10^{\text{exp}-01111111}$ correctly, thinking it is a base-10 number). The significand is a number between `1.000...` and `1.111....`

As an example, the hexadecimal bit pattern `0x3f000000` is:

3	f	0	0	0	0	0	0
0011	1111	0000	0000	0000	0000	0000	0000

grouping the bits:

`sign`	`exp`	`frac`
`0`	`01111110`	`00000000000000000000000`

which translates into
sign: +
exponent: `exp` $- 127 = 126 - 127 = -1$
significand: 1.`frac` = 1.0
So the number is: $+1.0 \times 2^{-1} = 0.5$ (base 10).
Special bit patterns exist for: denormalized numbers in general, zero, infinity and `NaN` (not-a-number, for instance 1/0):

Denormalized:

`sign`	`exp`	`frac`
±	`00...00`	— non-zero —

Zero:

`sign`	`exp`	`frac`
±	`00...00`	`000000...000000`

Infinity:

`sign`	`exp`	`frac`
±	`11...11`	`000000...000000`

`NaN`:

`sign`	`exp`	`frac`
±	`11...11`	— non-zero —

(Denormalized numbers use excess-126 instead of excess-127 for the exponent).
The smallest normalized number is thus (`exp=1`, `frac=0`): $n = 1.0 \times 2^{1-127} = 1.175 \times 10^{-38}$.
The smallest denormalized number is (`exp=0`, `frac=0000...0001` [22 zeros]): $n = 2^{-23} \times 2^{0-126} = 2 \times 2^{-149} = 1.401 \times 10^{-45}$.
The largest normalized number is (`exp=11111110`, `frac=11...11`): $n \approx 2 \times 2^{254-127} = 2 \times 2^{128} = 3.403 \times 10^{38}$.

❇

Exercises:

1. What are the smallest and largest double floating point numbers?
2. What is the bit pattern for the single floating

point value 9.0?

3. What is the bit pattern for the single floating point value 6.125?
4. What is the bit pattern for the single floating point value $-5/32$?
5. What single floating point value is represented by the bit pattern `0x42e48000`?
6. What single floating point value is represented by the bit pattern `0x00800000`?
7. What single floating point value is represented by the bit pattern `0xff800000`?
8. What single floating point value is represented by the bit pattern `0xff800001`?

Answers: 2: `0x41100000`, 3: `0x40c40000`, 4: `0xbe200000`, 5: 114.25, 6: 1.175×10^{-38}, 7: $-\infty$, 8: NaN.

❇

Most MIPS architectures nowadays have dedicated co-processors to do the floating-point calculations. The co-processor is similar to the integer processor. It also has 32 registers, each 32 bits wide, `$f0`. . . `$f31`. These can thus store 32 single-precision floats, or 16 double-precision floats. In the latter case, two consecutive registers store the number. So, for instance, the pair {`$f4`,`$f5`} can store a 64-bit double-precision float. (The syntax for such instructions is only writing the first of the two registers, which should always be an even-numbered register; `$f0`, `$f2`, etc.)

It comes with a set of instructions specifically for floating point operations and exchanging bit patterns between the float-registers and regular int-registers.
The first group is main memory access:

- `lwc1 $f0, $t4`: copy 4-byte contents of address pointed to by `$t4` into register `$f0` of the co-processor.
- `lwd1 $f0, $t4`: copy 8-byte contents of address pointed to by `$t4` into registers {`$f0`,`$f1`} of the co-processor.

- `swc1 $f0, $t4`: copy 4-byte contents of register `$f0` of the co-processor into consecutive addresses pointed to by `$t4`.
- `swd1 $f0, $t4`: copy 8-byte contents of registers {`$f0`,`$f1`} of the co-processor into consecutive addresses pointed to by `$t4`.

There are no 'immediate' versions of loading values into floating-point registers.
Copying data in the co-processors, like copying in the main processor, is done with move instructions:

- `mov.s $f0, $f2`: Copy contents of single-register `$f2` into single-register `$f0`.
- `mov.d $f0, $f2`: Copy contents of double-registers {`$f2`,`$f3`} into double-registers {`$f0`,`$f1`}.

Moving data between processors. Note that these do not convert the data; they simply copy the bit pattern:

- `mtc1 $t0, $f0`: (Move to). Copy contents of int-register `$t0` into register `$f0` of the co-processor.
- `mfc1 $t0, $f0`: (Move from). Copy contents of float-register `$f0` into int-register `$t0`.

Data conversion:

- `cvt.TO-TYPE.FROM-TYPE $f0, $f3`: Convert from format `FROM-TYPE` to format `TO-TYPE` (either '`s`', '`d`', or '`w`'). Example:

  ```
  cvt.d.w $f0, $f3
  ```

  convert the integer in `$f3` to double and store the result in registers {`$f0`,`$f1`}. Both operands are in the floating point co-processor; it can thus store integers as well.

Conditional branching: In contrast to integer branching instructions that can go in a single step, floating-point branching always goes in two steps. In the first step the condition is calculated and in the second step a conditional jump is made on basis of the resulting condition value:

- `c.COND.TYPE $f0, $f4`: Sets condition flag true or false. `COND`: '`eq`', '`gt`', '`lt`', '`le`', or '`ge`'. `TYPE`: '`s`', or '`d`', Example:

  ```
  c.lt.d $f0, $f4
  ```

  Set condition flag to true if double {`$f0`,`$f1`} is less than double {`$f4`,`$f5`}.

- `bc1t $t4`: Branch if condition flag in co-processor-1 is true to address stored in register `$t4`. Example:

  ```
  bc1t $t4
  nop
  ```

  The `nop` (no operation) instruction is needed to align the next code to an address being a multiple of 4 (because such branching instructions are less than 4 bytes).
  `bc1f $t4`: Same as `bc1t`, but branches when condition is false.

Floating point arithmetic is done by the same mnemonics as for integer arithmetic by simply adding a specification of the format (`.s` or `.d`). Note that, also here, there is no 'immediate' variant of the instructions; the input operands are always registers.

- `OP.TYPE $f0, $f2, $f4`: Perform arithmetic operation `OP` ('`add`', '`sub`', '`mul`' or '`div`') of type `TYPE` ('`s`' or '`d`') on `$f4` and `$f2` and store the result in `$f0`. Example:

  ```
  add.d $f0, $f2, $f4
  ```

  Add the double in {`$f4`,`$f5`} to the double in {`$f2`,`$f3`} and store the result in {`$f0`,`$f1`}

Exercise:

Check your answers of the above exercises with MIPS assembler programs.

Answer. Example:
Source code:

```
###################################################################
#                                                                 #
#   MIPS assembler program to convert int to float                #
#                                                                 #
###################################################################

.text
li $t0, 9           # substitute your int here
mtc1 $t0, $f0       # move bit pattern to $f0
cvt.s.w $f12, $f0   # convert word in $f0 to float and put in $f12

li $v0,2
syscall             # print float in $f12

li $v0, 10
syscall

# check in Coproc 1:$f12 what the hex code is for the number
```

Output:

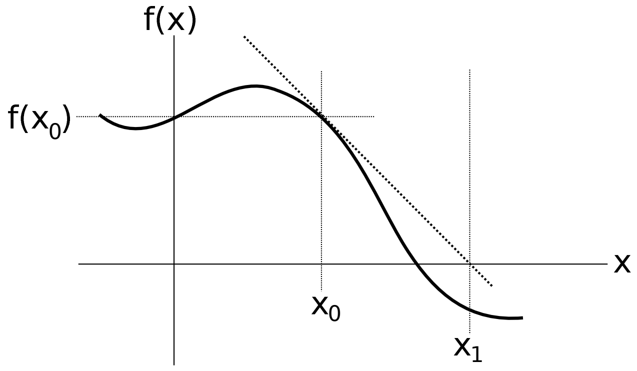
```
9.0
-- program is finished running --
```

We finish this section with a worked out example of floating point calculations. Namely a way to calculate any function with the method of Newton-Raphson. In this example we calculate the square root, a floating point function that is not implemented in hardware, so that we have to calculate it with software. This is not so difficult, as will be shown.

Calculating the square-root x of an input value (argument) A, or in other words $x = \sqrt{A}$, is the same problem as determining which x, when multiplied by itself, results in A, or in other words $x^2 = A$. This, in turn, is the same as finding the zero of a function

$$f(x) = x^2 - A$$

For this we can use the numerical recipe of Newton and Raphson. It consists of making successive guesses as to where the zero will be, based on the function value and its derivative at a certain point x. Picture 15 explains this. Starting at a point x_0, an estimation of where the zero might be is made on basis of the function value and derivative at



Picture 15: Method of Newton-Raphson for finding zeros in functions. Starting at a point x_0, a new estimation x_1 is found based on the function value and its derivative at x_0, namely $x_1 = x_0 - f(x_0)/f'(x_0)$

x_0. Successive iterations will lead to the x-value at which the function is zero. This will be then square root of the argument. In other words, at each step

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

In this case

$$x_{i+1} = x_i - \frac{x_i^2 - A}{2x_i}$$

These iterations use simple multiplications, divisions and subtractions, all implemented in MIPS hardware and part of its instruction set. The only thing that remains is to determine when we can stop; when the calculation is close enough. In other words, we have to implement a loop of the type repeat-until (or do-while) a certain condition is met, when two successive iterations give a value for x close enough, let us say less than the precision δx aimed at.

As an example, let's calculate the square root of 30.0 with the precision of $\delta x = 0.01$. We need to start somewhere, it could be anywhere (positive), so why not 30.0 itself? We then get the following sequence:

i	x_i	x_{i+1}	Δx
0	30.0	15.5	14.5
1	15.5	8.72	6.78
2	8.72	6.08	1.64
3	6.08	5.51	0.57
4	5.51	5.48	0.03
5	5.48	5.48	0.00

So, $\sqrt{30.0} = 5.48 \pm 0.01$. Below here is the entire Newton-Raphson root-calculation in MIPS, with 0.000001 precision. (Note that the multiplication of x_i by 2 is replaced by an addition of x_i to itself, because additions are much faster than multiplications):

Source code:

```
###################################################################
#                                                                 #
#  MIPS assembler program implements Newton-Raphson method        #
#    to calculate the square root of a number                     #
#                                                                 #
###################################################################

.data
answer: .asciiz "Its square root is: "
precision: .float 0.000001

.text
li $a0, 30           # argument: 30
mtc1 $a0, $f0        # move to co-processor
cvt.s.w $f0, $f0     # convert int to float
lwc1 $f1, precision  # load 0.000001 into $f1

# f0:  A
# f1:  precision
# f2:  xi

mov.s $f2, $f0  #x0 = A
dowhile:
# x(i+1) = xi - (xi*xi - A)/(xi+xi)
mul.s $f5, $f2, $f2   # $f5 = xi*xi
sub.s $f5, $f5, $f0   # $f5 = xi*xi-A
add.s $f6, $f2, $f2   # $f6 = 2xi
div.s $f5, $f5, $f6   # $f5 = (xi^2-A)/(2xi) = delta x
sub.s $f2, $f2, $f5   # $f2 = xi - (xi^2-A)/(2xi)
```

```
c.lt.s $f5, $f1
bc1f dowhile

# print result:
la $a0, answer
li $v0, 4
syscall
mov.s $f12, $f2
li $v0, 2
syscall              # print float in $f12
li $v0, 10
syscall              # end program
```

Output:

```
Its square root is: 5.477226
-- program is finished running --
```

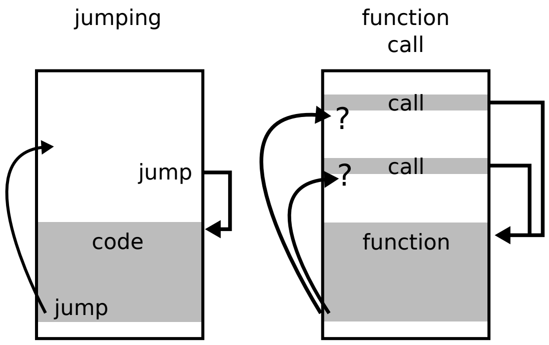
Exercise:

Write a program that finds out which of the pairs of coordinates is closest to each other? The distance is given by

$$d_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}.$$

The data of the 10 points in space are given by

```
.data
mylabel: .float
-86.197052961678, -11.611577927042, 61.992590409186,
13.27904907337, 19.916714939069, -20.722319227382,
-4.2258863849783, 45.297465287969, -90.569809463764,
-95.944184921756, 47.662103394222, 64.51404880929,
-21.807443326017, -24.698308698634, 75.762861069285,
-36.654951049497, 43.494575924847, 86.447702712523,
47.778522103541, -65.538547015254, 59.179507921132,
54.530144848978, -67.933306319424, -36.693280039158,
84.306270676353, 21.53150037385, 44.934044045448,
73.659364837401, 81.085378856605, -73.1797878870
```



Picture 16: Difference between simple jumping (`GOTO`) on the left and function calling (`GOSUB`) on the right. For function calls we have to remember what was the address of the code that was interrupted by the function call, otherwise we do not know where to return to

They are organized according to:
x_1, y_1, z_1,
x_2, y_2, z_2,
etc.

Functions and the stack

Procedures and functions — the difference between them is that functions return a value, where procedures don't – are code that can be called from anywhere within the program, including other procedures and functions. If a function calls itself, it is *called* recursive, an example of which will be given here too. The important thing of functions and procedures is that, after the code of them has finished, the program should continue at the point where it was interrupted by the function call, see Picture 16. It therefore has to save somewhere this information of where it was interrupted. We will see how this is done. In fact, MIPS is already well prepared for implementing this high-level programming concept, a concept that for instance in BASIC is `GOSUB`, there where simple jumps are `GOTO`.

In a first example we will use a very simple procedure that prints a text. Note that the address of the text is passed as an argument to the procedure. This avoids that the procedure is dependent on the rest of the program and in this way we can write code that can be recycled. If we do our work carefully, functions can be taken out of our program and inserted in other programs without any modification. In fact, we can keep a set of functions – a library of functions – in separate files, ready to be used when necessary. This ensures that we will not keep reinventing the wheel every time we need it.

However, the main difference between normal code we have seen so far and functions is that a function can be called from various places in the program, and when the function finishes the program should continue at the instruction directly following the code that called the function. Therefore, simple jump (`j`) instructions are not adequate, because they jump to a static address (the value of the label contained in it).

A call to the function is done by a so-called jump-and-link (`jal`) to that address. This is doing two distinct things:

- Calculate `pc+4` and save this in register `$ra`. `pc+4` $\rightarrow$ `($ra)`. It points to the first instruction after the `jal`.

- `j address`, or in other words, `address` $\rightarrow$ `pc`.

Now note the direct '4' at one of the input gates of the ALU in architecture of Picture 7. This now makes sense; the `jal` instruction is directly implemented in hardware and is thus faster than it would have been had it been implemented by software. Returning from a function is achieved by a jump-on-register, which returns to the main program.

- `jr $ra`. Or in other words, `($ra)` $\rightarrow$ `pc`.

To make functions independent of the main code, functions cannot use information of the main program! In high-level programming it means that functions are not allowed to use global variables. If information is to be used in a function, this information has to be passed to the function, either by directly supplying the value, or by supplying the address where the value is stored. The former is called passing by value, the latter passing by reference. Similarly, if the function generates a value, it should not store this in a global variable, but rather in the return value (or placed in the memory address that was passed to the function as an argument).

In assembler, four registers are used to pass arguments to functions, $a0 . . . $a3. If more arguments are needed, for instance an entire array, we have to pass the *address* to the information to the function. In the following example, the function is placed after the main code. It simply prints the string (an array of chars), the address of which is passed as an argument in $a0.

```
###############################################
#  MIPS assembler program that shows how      #
#    to implement a procedure                 #
###############################################

.data
text1: .asciiz "Text to print\n"

.text
  la $a0, text1    # argument passed to procedure
  jal procedure    # call procedure
      # stores pc+4 into $ra and makes a "j procedure"
      # returning from procedure, program continues
      # with next instruction:

# terminate program:
  li $v0, 10
  syscall

################  FUNCTIONS: ####################

procedure:
  ###############################################
  # arguments:                                  #
  #     $a0: address of null-terminated string  #
  # return value(s):                            #
  #     none                                    #
  ###############################################

  li $v0, 4
  syscall
  jr $ra  # return to address saved in $ra
```

Output:

```
Text to print

-- program is finished running --
```

The following example shows how to implement a function that receives arguments and returns a value. In this case it implements the function x^n, with x and n received in `$a0` and `$a1`, respectively, and the function returns the calculated value in `$v0`.

```
###############################################
#  MIPS assembler program that shows how      #
#    to implement a function                  #
###############################################

.data

.text
  li $a0, 3
  li $a1, 5
  jal power     # call function $v0 = power($a0, $a1)

# print result:
  move $a0, $v0
  li $v0, 1
  syscall

# terminate program:
  li $v0, 10
  syscall

################  FUNCTIONS: ####################

power:
  ###############################################
  # arguments:                                  #
  #     $a0: x                                  #
  #     $a1: n                                  #
  # return value(s):                            #
  #     $v0: x^n                                #
  ###############################################
  #   implemented with a while loop             #

  li $v0, 1
power_startloop:
  beqz $a1, power_exit
  mul $v0, $v0, $a0
  subi $a1, $a1, 1
  j power_startloop
power_exit:
  jr $ra
```

Output:

```
243

-- program is finished running --
```

Moreover, to make functions *fully* independent of the rest of the code, the state of the registers has to be unaltered by the function call. In MIPS we use the following convention:

- The t-registers are the responsibility of the calling code. The 'caller'. If the caller is still going to use these t-register values after the function call, the caller has to save them *before* calling the function and retrieve them *immediately after* returning from the function. Note that t-registers that will no longer be needed after the function call do not have to be saved. Note also that no s-registers have to be saved at all. "Not my problem!"

- The s-registers are the responsibility of the called code. The 'callee'. If the callee (function) is going to use these s-registers in the function, the callee has to save their values *before* using these registers and make sure to retrieve them all *before exiting* the function; the caller relies — or might rely; we have to assume they do! — on the fidelity of the s-register values. Note that s-registers that will not be used by the callee function do not have to be saved. Note also that no t-registers have to be saved at all by the callee. "Not my problem!"

These registers have to be saved somewhere in memory before the instructions of the function are executed and retrieved afterwards. The best place to do that is the stack. A stack differs from conventional memory — the 'heap' — in that, whereas all elements of the heap are always accessible at all times, only the top value of the stack — the latest one placed there — is accessible. This implements the LIFO-concept (last in, first out). We can thus place a value — 'push' — on top of the stack, or remove — 'pop' — one from the stack. Special instructions for popping and pushing items on the stack do not exist, but they can easily be implemented. There does exist a special stack pointer register (`$sp`). Pushing and popping is thus implemented as (an example of storing `$t0`):

- push:
  ```
  addi $sp, $sp, -4
  sw $t0, ($sp)
  ```
- pop:
  ```
  lw $t0, ($sp)
  addi $sp, $sp, 4
  ```

The stack is thus growing and shrinking every time we push and pop items. Note that we have to do this in the correct order (last in, first out) and also in the correct number; every item pushed on the stack has to be popped off it, otherwise the stack runs the risk of over- or underflowing. Note also that the order of changing the stack pointer and accessing the memory pointed to by the stack pointer is reversed in pops and pushes, as shown here above. The reason may be obvious.

The code below shows an example of a main code and a function that both use both `$t0` and `$s0` where it is shown which one is responsible for saving and restoring which of these register values. It loads the values of 3 and 4 into `$t0` and `$s0`, then copies these to `$a0` and `$a1` because the function expects them as arguments there. The caller code saves the t-register on the stack and the function the s-register it uses. As can be seen, the values are restored and when printed they have their original value of 3 and 4 at the end.

```
###############################################
#  MIPS assembler program that shows how      #
#    to use the stack                         #
###############################################

.text
  li $t0, 3
  li $s0, 4
```

```
  move $a0, $t0       # pass arguments
  move $a1, $s0       #   to function
  addi $sp, $sp, -4   # push $t0
  sw $t0, ($sp)       #   onto stack
  jal multi           # call function
  lw $t0, ($sp)       # pop $t0
  addi $sp, $sp, 4    #   from stack

# print result:
  move $a0, $v0
  li $v0, 1
  syscall
# check if $t0 and $s0 changed:
  move $a0, $t0
  syscall             # print $t0
  move $a0, $s0
  syscall             # print $s0

# terminate program:
  li $v0, 10
  syscall

################  FUNCTIONS: ####################

multi:
  ###############################################
  # arguments:                                  #
  #     $a0: int                                #
  #     $a1: int                                #
  # return value(s):                            #
  #     $v0 = $a0 * $a1                         #
  ###############################################

  addi $sp, $sp, -4   # push $s0
  sw $s0, ($sp)       #   onto stack
  move $t0, $a0
  move $s0, $a1
  mul $v0, $t0, $s0
  lw $s0, ($sp)       # pop $s0
  addi $sp, $sp, 4    #   from stack
multi_exit:
  jr $ra
```

Output:

```
1234
-- program is finished running --
```

Note that if the function is going to call other functions, we also need to save the return address on the stack before we issue a `jal`. Here is an example of a recursive function that calculates the factorial $n!$ of the argument n:

```
###############################################
#  MIPS assembler program that shows an       #
#    example of a recursive function          #
###############################################

.text
  li $a0, 5
  jal factorial

#print result:
  move $a0, $v0
  li $v0, 1
  syscall

# terminate program:
  li $v0, 10
  syscall

################  FUNCTIONS: ####################

factorial:
  ###############################################
  # arguments:                                  #
  #     $a0: int n                              #
  # return value(s):                            #
  #     $v0 = n!                                #
  ###############################################

  li $v0, 1
  beqz $a0, factorial_exit  # 0! = 1; exit
  addi $a0, $a0, -1
  addi $sp, $sp, -4         # save $ra
  sw $ra, ($sp)             #   onto stack
  jal factorial             # $v0 = (n-1)!
  lw $ra, ($sp)             # retrieve $ra
  addi $sp, $sp, 4          #   from stack
  addi $a0, $a0, 1
  mul $v0, $v0, $a0         # $v0 = n*(n-1)!
factorial_exit:
  jr $ra
```

Output:

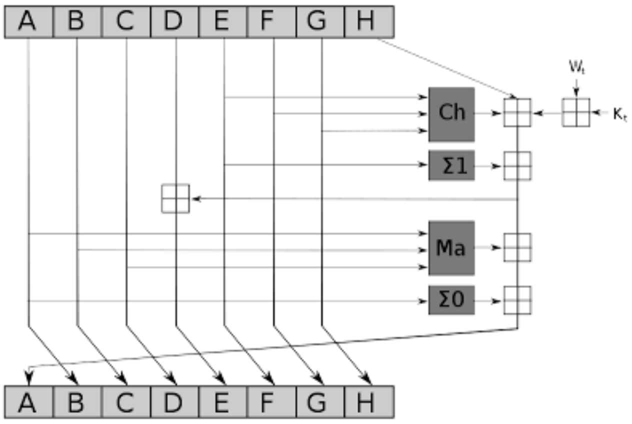
```
120
-- program is finished running --
```

Calculating blockchain

Time to look at an extensive example to finalize this book. Blockchain may serve very well. It consists of encrypting data by executing many simple logical operations on it. Operations like xor, shift-left, etc. Perfect for assembler programming. More so since blockchain is money — or can be money, in case of bitcoin — and time is money, so the faster our program is, the richer we get.

The most famous is the SHA-256 algorithm that is shown in Picture 17 (source of picture and description: ”Mining Bitcoin with pencil and paper: 0.67 hashes per day” on Ken Shirriff’s blog). It takes eight 32-bit pieces of data (A to H), and two key registers, Kt and Wt and performs the following actions on them:

- The Ma majority box looks at the bits of A, B, and C. For each position, if the majority of the bits are 0, it outputs 0. Otherwise it outputs 1. That is, for each position in A, B, and C, look at the number of 1 bits. If it is zero or one, output 0. If it is two or three, output 1.
- The Σ0 box rotates the bits of A to form three rotated versions, and then sums them together modulo 2. In other words, if the number of 1 bits is odd, the sum is 1; otherwise, it is 0. The three values in the sum are A rotated right by 2 bits, 13 bits, and 22 bits.
- The Ch ’choose’ box chooses output bits based on the value of input E. If a bit of E is 1, the output bit is the corresponding bit of F. If a bit of E is 0, the output bit is the corresponding bit of



Picture 17: Example of the SHA-256 algorithm of encrypting data (A-H). Picture from Wikipedia

G. In this way, the bits of F and G are shuffled together based on the value of E.

- The next box Σ1 rotates and sums the bits of E, similar to Σ0 except the shifts are 6, 11, and 25 bits.
- The plus boxes ⊞ perform 32-bit addition, generating new values for A and E. The input Wt is based on the input data, slightly processed. (This is where the input block gets fed into the algorithm). The input Kt is a constant defined for each round.

Example of a single pass of SHA-256: Starting with

```
A = 0x87564C0C
B = 0xF1369725
C = 0x82E6D493
D = 0x63A6B509
E = 0xDD9EFF54
F = 0xE07C2655
G = 0xA41F32E7
H = 0xC7D25631
```

```
  Wt = 0x6534EA14
  Kt = 0xC67178F2
```
we wind up with
```
  A = 0xE620B22B
  B = 0x87564C0C
  C = 0xF1369725
  D = 0x82E6D493
  E = 0xADCEF783
  F = 0xDD9EFF54
  G = 0xE07C2655
  H = 0xA41F32E7
```
That's it. 64 times repeating and if the first 17 bits of register A are zeros, then we have found a new block in the blockchain. If not, we have to start with a new key (a.k.a. 'nonce').

Exercise:

Write a MIPS program that calculates a blockchain: repeating starting with random keys and the data as given above until the first 17 bits of A are 0.

A | MIPS Instruction set

Mnemonic	Opcode (func)	Meaning	Type	Operation
`sll`	0 (0)	Shift left logical	R	`$rd = $rs`$\ll$`num`
`srl`	0 (2)	Shift right logical	R	`$rd = $rs`$\gg$`num`
`sra`	0 (3)	Shift right arithmetic	R	`$rd = $rs`$\gg$`num+msb`
`sllv`	0 (4)	Shift left logic. var.	R	`$rd = $rs`$\ll$`$rt`
`srlv`	0 (6)	Shift right logic. var.	R	`$rd = $rs`$\gg$`$rt`
`srav`	0 (7)	Shift right arithm. var.	R	`$rd = $rs`$\gg$`$rt+msb`
`add`	0 (20)	Add	R	`$rd = $rs+$rt`
`addu`	0 (21)	Add unsigned	R	`$rd = $rs+$rt`
`sub`	0 (22)	Subtract	R	`$rd = $rs-$rt`
`subu`	0 (23)	Subtract unsigned	R	`$rd = $rs-$rt`
`addi`	8	Add immediate	I	`$rt = $rs+imm`
`addiu`	9	Add immediate unsigned	I	`$rt = $rs+imm`
`mult`	0 (18)	Multiply	R	`HI,LO = $rs*$rt`
`multu`	0 (19)	Multiply unsigned	R	`HI,LO = $rs*$rt`
`div`	0 (1a)	Divide	R	`LO = $rs/$rt` `HI = $rs%$rt`
`divu`	0 (1b)	Divide unsigned	R	`LO = $rs/$rt` `HI = $rs%$rt`
`and`	0 (24)	And	R	`$rd = $rs&$rt`
`andi`	c	And immediate	I	`$rt = $rs&imm`
`or`	0 (25)	Or	R	`$rd = $rs\|$rt`
`nor`	0 (27)	Nor	R	`$rd = !($rs|$rt)`
`ori`	d	Or immediate	I	`$rt = $rs|imm`
`xor`	0 (26)	Xor	R	`$rd = $rs^$rt`
`xori`	e	Xor immediate	I	`$rt = $rs^imm`

Instruction types: R: registers, I: immediate

Mnemonic	Opcode (func)	Meaning	Type	Operation
`slt`	0 (2a)	Set if less than	R	`$rd = ($rs<$rt)?1:0`
`sltu`	0 (2b)	Set if less than unsigned	R	`$rd = ($rs<$rt)?1:0`
`slti`	a	Set if less than imm.	I	`$rt = ($rs<imm)?1:0`
`sltiu`	b	Set if less than imm. unsigned	I	`$rt = ($rs<imm)?1:0`
`j`	2	Jump	J	`PC=addr`
`jal`	2	Jump and link	J	`$ra=PC+4, PC=addr`
`jr`	0 (8)	Jump register	R	`PC=$rs`
`jalr`	0 (8)	Jump and link register	R	`$ra=PC+4, PC=$rs`
`beq`	4	Branch if equal	I	`$rs==$rt?  PC=PC+4+imm`
`bne`	5	Branch if not equal	I	`$rs!=$rt?  PC=PC+4+imm`
`blt`	-	Branch if less than	P	`$rs<$rt?  PC=PC+4+imm`
`bgt`	-	Branch if greater than	P	`$rs>$rt?  PC=PC+4+imm`
`ble`	-	Branch if less or equal	P	`$rs<=$rt?  PC=PC+4+imm`
`bge`	-	Branch if greater or eq.	P	`$rs>=$rt?  PC=PC+4+imm`
`move`	-	Move (copy)	P	`$rd=$rs`
`mfhi`	0 (10)	Move from high	R	`$rd=HI`
`mflo`	0 (12)	Move from low	R	`$rd=LO`
`mfc0`	16	Move from control	R	`$rd=CR[$rs]`
`lb`	20	Load byte	I	`$rt=$rt+M[$rs+imm]` *
`lbu`	24	Load byte unsigned	I	`$rt=$rt+M[$rs+imm]` *
`lhu`	25	Load half word	I	`$rt=$rt+M[$rs+imm]` *
`lui`	f	Load upper imm.	I	`$rt=$rt+imm` *
`lw`	23	Load word	I	`$rt=$rt+M[$rs+imm]`
`li`	-	Load immediate	P	`$rd=imm`
`la`	-	Load address (=`li`)	P	`$rd=imm`
`sb`	28	Store byte	I	`M[$rs+imm]=$rt` *
`sh`	29	Store half word	I	`M[$rs+imm]=$rt` *
`sw`	2b	Store word	I	`M[$rs+imm]=$rt`
`lwc1`	31	FP load single	I	`$ft = M[$fs+imm]`
`ldc1`	35	FP load double	I	`2×$ft = M[$fs+imm]`
`swc1`	39	FP store single	I	`M[$fs+imm] = $ft`
`sdc1`	3d	FP store double	I	`M[$fs+imm] = 2×$ft`
`c.eq.TYPE`	11 (32)	FP equal	FR	`COND = ($ft==$ft)?1:0`
`c.lt.TYPE`	11 (3c)	FP less than	FR	`COND = ($ft<$ft)?1:0`
`c.le.TYPE`	11 (3e)	FP less or equal	FR	`COND = ($ft<=$ft)?1:0`
`c.gt.TYPE`	-	FP greater than	P	`COND = ($ft>$ft)?1:0`
`c.ge.TYPE`	-	FP greater or eq.	P	`COND = ($ft>=$ft)?1:0`
`bc1t`	11^1	FP branch on true	FI	`COND? PC=PC+4+imm`
`bc1f`	11^0	FP branch on false	FI	`!COND? PC=PC+4+imm`

Instruction types: R: registers, I: immediate, P: pseudo-code, J: jump

*: appropriate bits only

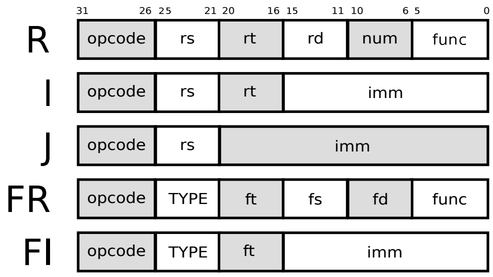
`TYPE` = s (=10), d (=11)

2×: 2 consecutive float registers

^{1}: ft=1, ^{0}: ft=0

Mnemonic	Opcode (func)	Meaning	Type	Operation
`add.s`	11 (0)	FP Add single	FR	`$fd = $fs+$ft`
`sub.s`	11 (1)	FP Subtract single	FR	`$fd = $fs-$ft`
`mul.s`	11 (2)	FP Multiply single	FR	`$fd = $fs*$ft`
`div.s`	11 (3)	FP Divide single	FR	`$fd = $fs/$ft`
`add.d`	11 (0)	FP Add double	FR	2×`$fd = `2×`$fs+`2×`$ft`
`sub.d`	11 (1)	FP Subtract double	FR	2×`$fd = `2×`$fs-`2×`$ft`
`mul.d`	11 (2)	FP Multiply double	FR	2×`$fd = `2×`$fs*`2×`$ft`
`div.d`	11 (3)	FP Divide double	FR	2×`$fd = `2×`$fs/`2×`$ft`

Instruction types: F: float, R: registers
`TYPE` = s (=10), d (=11)
2×: 2 consecutive float registers



B | Assembler directives

Directive	Meaning	Example
`.data`	Start of data segment	
`.kdata`	Start of kernel data segment	
`.text`	Start of code segment	
`.ktext`	Start of kernel code segment	
`.ascii <str>`	Store <str> in memory	`.ascii "Ajax"`
`.asciiz <str>`	Store <str>+00 in memory	`.ascii "Benfica"`
`.byte b1, b2 ..., bn`	Store byte(s) in memory	
`.half h1, h2 ..., hn`	Store half-word(s) in memory	
`.word w1, w2 ..., wn`	Store word(s) in memory	`float 1.0, 2.1, 3.6`
`.double d1, d2 ..., dn`	Store double-precision float(s) in memory	`.double 3.14E03`
`.eqv`	define constant (does not store it in memory)	`myvalue .eqv 64`

C | System calls

function	**$v0**	**argument(s)**	**return value(s)**
print integer	1	$a0 = integer	
print float	2	$f12 = float	
print double	3	$f12, $f13 = double	
print string	4	$a0 = address of null-terminated string	
read integer	5		$v0 integer read
read float	6		$f0 float read
read double	7		$f0 double read
read string	8	$a0 = address of buffer $a1 = max. length	
exit (terminate execution)	10		
print character	11	$a0 = character	
read character	12		$v0 character read
print integer in hexadecimal	34	$a0 = integer	
print integer in binary	35	$a0 = integer	
print integer as unsigned	36	$a0 = integer	
set random seed	40	$a0 = integer	
random int	41	$a0 random int	$a0 next random int

D | ASCII

Dec	Hex	Bin	Value	Meaning
0	00	0000000	NUL	Null character
1	01	0000001	SOH	Start of header
2	02	0000010	STX	Start of text
3	03	0000011	ETX	End of text (Ctrl-C)
4	04	0000100	EOT	End of transmission
5	05	0000101	ENQ	Enquiry
6	06	0000110	ACK	Acknowledge
7	07	0000111	BEL	Bell
8	08	0001000	BS	Back space
9	09	0001001	HT	Horizontal tab
10	0A	0001010	LF	Line feed *
11	0B	0001011	VT	Vertical tab
12	0C	0001100	FF	Form feed
13	0D	0001101	CR	Carriage return *
14	0E	0001110	SO	Shift out
15	0F	0001111	SI	Shift in
16	10	0010000	DLE	Data link escape
17	11	0010001	XON	Device control 1
18	12	0010010	DC2	Device control 2
19	13	0010011	XOFF	Device control 3
20	14	0010100	DC4	Device control 4
21	15	0010101	NAK	Not acknowledge
22	16	0010110	SYN	Synchronous idle
23	17	0010111	ETB	End of transfer block
24	18	0011000	CAN	Cancel
25	19	0011001	EM	End of medium
26	1A	0011010	SUB	Substitute (Ctrl-Z)
27	1B	0011011	ESC	Escape
28	1C	0011100	FS	File separator
29	1D	0011101	GS	Group separator
30	1E	0011110	RS	Record separator
31	1F	0011111	US	Unit separator

*: UNIX (Linux): newline is LF, MS-DOS (Windows): newline is CR+LF

Dec	Hex	Bin	Value	Dec	Hex	Bin	Value	Dec	Hex	Bin	Value
32	20	0100000	space	64	40	1000000	@	96	60	1100000	’
33	21	0100001	!	65	41	1000001	A	97	61	1100001	a
34	22	0100010	"	66	42	1000010	B	98	62	1100010	b
35	23	0100011	#	67	43	1000011	C	99	63	1100011	c
36	24	0100100	$	68	44	1000100	D	100	64	1100100	d
37	25	0100101	%	69	45	1000101	E	101	65	1100101	e
38	26	0100110	&	70	46	1000110	F	102	66	1100110	f
39	27	0100111	’	71	47	1000111	G	103	67	1100111	g
40	28	0101000	(	72	48	1001000	H	104	68	1101000	h
41	29	0101001	)	73	49	1001001	I	105	69	1101001	i
42	2A	0101010	*	74	4A	1001010	J	106	6A	1101010	j
43	2B	0101011	+	75	4B	1001011	K	107	6B	1101011	k
44	2C	0101100	,	76	4C	1001100	L	108	6C	1101100	l
45	2D	0101101	-	77	4D	1001101	M	109	6D	1101101	m
46	2E	0101110	.	78	4E	1001110	N	110	6E	1101110	n
47	2F	0101111	/	79	4F	1001111	O	111	6F	1101111	o
48	30	0110000	0	80	50	1010000	P	112	70	1110000	p
49	31	0110001	1	81	51	1010001	Q	113	71	1110001	q
50	32	0110010	2	82	52	1010010	R	114	72	1110010	r
51	33	0110011	3	83	53	1010011	S	115	73	1110011	s
52	34	0110100	4	84	54	1010100	T	116	74	1110100	t
53	35	0110101	5	85	55	1010101	U	117	75	1110101	u
54	36	0110110	6	86	56	1010110	V	118	76	1110110	v
55	37	0110111	7	87	57	1010111	W	119	77	1110111	w
56	38	0110100	8	88	58	1011000	X	120	78	1110100	x
57	39	0111001	9	89	59	1011001	Y	121	79	1111001	y
58	3A	0111010	:	90	5A	1011010	Z	122	7A	1111010	z
59	3B	0111011	;	91	5B	1011011	[	123	7B	1111011	{
60	3C	0111100	<	92	5C	1011100	\	124	7C	1111100	|
61	3D	0111101	=	93	5D	1011101	]	125	7D	1111101	}
62	3E	0111110	>	94	5E	1011110	ˆ	126	7E	1111110	˜
63	3F	0111111	?	95	5F	1011111	_	127	7F	1111111	delete

i Index